UCLouvain

epl

École polytechnique de Louvain

Computer-aided musical composition

Constraint programming and music

Author: **Baptiste Lapière**
Supervisor: **Peter Van Roy**
Readers: **Alex Mattenet, Jean Bresson**
Academic year 2019–2020
Master [120] in Computer Science and Engineering

Abstract

This master's thesis presents the design of a tool destined to generate scores respecting rules stated by the user. It uses constraint programming in order to model the specification of rules through a general CSP that can be extended with additional constraints. As this is a yet unsolved challenge, the focus is set on rhythms generation with an unknown number of musical events. On one hand, OpenMusic is used to handle the musical part of this work. It is a visual programming environment based on Lisp designed to program music. On the other hand, the constraint programming part will be handled by Gecode, a C++ toolkit to develop constraint-based applications.

In order to provide a powerful and expressive constraint system, an interface is created between OpenMusic and Gecode. With the help of these two main tools, a program is designed that contains an extendable base rhythm CSP and represents a good starting point towards a general constraint-based CAC tool.

Contents

1 Introduction **1**

2 Theoretical background **4**

2.1 Computer-Aided Composition 4
2.2 Constraint programming 4
2.2.1 Definitions 4
2.2.2 Example CSP 5
2.2.3 Symmetries 6
2.3 A few musical notions 7
2.4 Music and constraints 8
2.5 Previous work 9

3 Tools **12**

3.1 OpenMusic 12
3.1.1 OpenMusic patches 13
3.2 Gecode 16
3.2.1 Search space 16
3.2.2 Variables and arguments 16
3.2.3 Constraints 17
3.2.4 Branching 17
3.2.5 Search 17

4 Models and constraints **18**

4.1 Modeling a rhythm CSP 18
4.1.1 The rhythm problem 18

4.1.2 First approach: avoid the problem 19
4.1.3 Second approach: rhythm from a graph 21
4.1.4 Third approach: Events described by their position and duration 24
4.1.5 Conclusion 31
4.2 Comparison of the models 32
4.3 Constraints on rhythm 33
4.3.1 Sample constraints 34
4.3.2 Limitations 37
4.4 Conclusion 37

5 GiL: Interface between Gecode and OpenMusic 39
5.1 C Wrapper 39
5.2 Lisp Wrapper 41
5.3 Example program 41
5.4 Conclusion 43

6 Implementation 44
6.1 Rhythm-Box: a rhythm CSP in OpenMusic 44
6.1.1 *rhythm-box* box 44
6.1.2 *constraint* boxes 45
6.1.3 *search-next* box 46
6.2 Base CSP 48
6.2.1 Adding constraints 48
6.3 Examples programs with Rhythm-Box 49
6.4 Summary 49

7 Conclusion 52
7.1 Further Work 53
7.1.1 Completing GiL 53
7.1.2 Improving Rhythm-Box 55
7.1.3 Discussion: including pitch 57

Bibliography 59

A Rhythm-Box **61**
A.1 Tutorial: searching rhythms 61
A.2 Constraints catalogue 65
A.3 Tutorial: create a new constraint 66
A.4 Source code 68

B GiL **78**
B.1 How to extend GiL 78
B.1.1 Tutorial: adding a constraint 78
B.1.2 Branching strategies 80
B.1.3 Expressions 80
B.2 Source code 80
B.2.1 C Wrapper 80
B.2.2 Lisp Wrapper 107

C Experiments on the model **123**
C.1 Experiments 123
C.1.1 Explicit position and duration 123
C.1.2 Explicit duration only 124
C.1.3 Explicit position only 126

Chapter 1

Introduction

The importance of computers in musical composition is continuously growing since the first experiments of computer-assisted composition in 1957. They are involved in many steps of musical production. One of the applications of computer in music is Computer-Assisted Composition (CAC), a particular field music informatics that aims to generate scores from algorithms.

On the other hand, constraint programming is a promising paradigm that has already proven itself efficient in practical applications such as planning and scheduling [Barták, 1999]. As stated by E. C. Freuder: [Frühwirth, Thom and Abdennadher, Slim, 2003]:

> “Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.”
>
> — Eugene C. Freuder, Inaugural issue of the *Constraints journal*, 1997

This citation is now more than twenty years old, and nowadays sophisticated constraints systems and efficient constraints solvers are available. This master’s thesis aims to use the power of constraint programming to create a constraint-based CAC tool. Such applications already exist but are all limited in some ways. The common drawback of the existing tools involves the generation of rhythms.

In this context, the goal is to create a tool that solves rhythm constraint satisfaction problems to generate rhythms. Gecode, a powerful and expressive

framework for constraint-based applications [Schulte et al., 2019], will be used to handle the constraint-programming aspect the tool, and OpenMusic [Amoric, 2006] will be the environment in which the tool is built in order to process the musical aspect.

Results

As outcome, this master's thesis provides three contributions:

- The first contribution is the creation of GiL (Gecode interface Lisp), an interface between OpenMusic and Gecode. This is the first step to create a tool that uses Gecode in OpenMusic. GiL allows to call Gecode functions in Lisp, and OpenMusic can evaluate Lisp programs. GiL may not be a complete interface for now, but extending it to cover most of Gecode features is not a hard task.
- The second contribution is a model of the rhythm problem. The model can generate rhythms of fixed duration without specifying the number of musical events or of variables to constrain. It is created as a base CSP that can be extended with additional constraints.
- Finally, the model has been integrated into OpenMusic to provide visual and usable results of the generated rhythms. A tool has been created, Rhythm-Box, that encapsulates a base general rhythm problem and lets the user add constraints. Rhythm-Box only handles rhythms, but it can easily be extended to solve problems that involve pitch as well and is a good starting point to further explore the possibilities of constraint programming in CAC.

Outline

Chapter 2 introduces the necessary notions to read this master's thesis. Then, it presents a brief survey of the work done in the field of constraint-based CAC. Next, chapter 3 introduces OpenMusic, the musical programming environment in which the tool is created, and Gecode, the constraint programming framework used to model and solve the rhythm problems. How the general rhythm problem was

modelled, and the different approaches used to reach the chosen representation, is described in chapter 4. GiL, the interface between OpenMusic and Gecode, is introduced in chapter 5 and the implementation of the Rhythm-Box tool as well as some measurements are presented in chapter 6. Finally, chapter 7 concludes this master's thesis.

Chapter 2

Theoretical background

2.1 Computer-Aided Composition

Computers play an important role in today's music. Broad-sense computer music includes every intervention of computers in the music production process, including mixing, editing, or even sound synthesis and effect processing. Computer-Aided Composition (CAC) in particular refers to the generation of music scores by a computer. CAC is a field born from the work of L. Hiller in in 1957, who created the first computerized composition [Assayag, 1998]. The easy access to a personal computer and the evolution of programming languages, as well as the creation of standards as MIDI allowed to improve CAC and popularize its use.

Several strategies for algorithmic composition have been explored in this framework, including Constraint-Based CAC that uses a rule-based approach to emulate compositional rules [Anders, 2007] and that this master's thesis aims to further explore.

2.2 Constraint programming

2.2.1 Definitions

Constraint programming is a programming paradigm that lets the user specify relations between objects instead of setting their value [Frühwirth, Thom and Abdennadher, Slim, 2003].

These relations are called constraints. Posting a constraint on one or several variable(s) will reduce the space of its (their) possible values, which is called the domain of the variable. The process of domain reduction is called constraint propagation. Once the variables and constraints are defined, partial information on the value of the variables is available. The idea of constraint programming is to allow to reason with this concept of partial information in a higher-level programming language.

Problems solved with constraint programming are Constraints Satisfaction Problems (CSP). A CSP is defined by a set of variables $V_0...V_n$, a set of domains $D_0...D_n$ where D_i is the set of possible values of V_i, and a set of constraints $C_0...C_p$ that are predicates on the variables. A variable V_i can be instantiated to a value v_i. A solution of the CSP is an instantiation of all its variables that verifies the constraints, i.e. where $\bigwedge_{i=0}^{p} C_i(v_0...v_n)$ is true [Truchet, 2004]. The responsibility for finding solutions lies with the constraint solver. To do so, it assigns values to variables from their domain and tests if the result is a solution. The decision of which value is first assigned to which variable depends on the search heuristic. Therefore, the challenge for the user is to efficiently model the real world's problem with sets of variables and constraints. A good model can minimize the possible values and reduce the search time, while a poor model can quickly lead to a search explosion.

2.2.2 Example CSP

A simple example of a problem that can be expressed as a CSP is the Latin Square problem. A Latin square a $n \times n$ matrix where each cell contains a symbol among n different symbols. Each row and each column of the matrix contain exactly one of each symbol, i.e. all symbols in each row and all symbols in each column are distinct. If we represent the symbols with integers, a Latin square can be described by the following CSP:

Variables

M a $n \times n$ matrix of integer variables of domain $[0, n[$.

Constraints

$C_1(M) = \forall i \in [0, n[: \mathit{alldiff}(M_{i0}...M_{i(n-1)})$

0	1	2	3
1	0	3	2
2	3	0	1
3	2	1	0

1	0	3	2
0	1	2	3
2	3	0	1
3	2	1	0

0	2	1	3
1	3	0	2
2	0	3	1
3	1	2	0

0	3	2	1
3	0	1	2
2	1	0	3
1	2	3	0

Figure 2.1: Four equivalent Latin squares

$$C_2(M) = \forall j \in [0, n[\, : \, \mathit{alldiff}(M_{0j}...M_{(n-1)j})$$

The predicate $\mathit{alldiff}(v_0...v_n)$ is a widely used shorthand for $\forall i, j \in [0, n], i \neq j \,:\, v_i \neq v_j$ that allows to express in a single constraint that all the variables $v_0...v_n$ have different values.

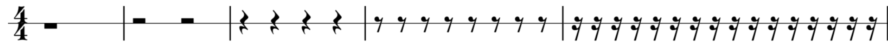
This model has many solutions for a given n, mainly because a Latin square can have several equivalent arrangements. We can indeed produce a Latin square from another by permuting rows and columns. Figure 2.1 from [Schulte et al., 2019] shows an example of four equivalent Latin squares of size 4×4.

2.2.3 Symmetries

The equivalent solutions in the Latin square are called symmetries [Rossi et al., 2006]. Symmetries are a crucial problem of constraint programming; they occur when the search executed by the solver visits equivalent states of the problem. According to the OEIS[1], the 4 Latin square example has 576 possible solutions, but only 4 distinct solutions. Increasing the size of n in this case quickly multiplies the number of distinct solutions s, and the number of possible solutions is equal to $n!(n-1)! * s$. Such a fast equivalence explosion can be very problematic as the

[1]On-Line Encyclopedia of Integer Sequences, an online database of integer sequences created and maintained by Nail Sloane.

(a) Notes duration



(b) Rests duration

Figure 2.2: Events duration

search will be considerably slow because it has to visit a vast number of equivalent sates.

2.3 A few musical notions

In order for the reader to understand the following sections, some musical notions and vocabulary are required. This section is neither a musical theory lesson nor a complete glossary of musical terms; it only describes a few concepts used in the following chapters.

Voice A music score is composed of voices. A voice is typically an instrument that plays its part of the musical piece.

Bar and time signature Each voice is divided into bars. Bars are time subdivisions of known duration that contain the musical events. To know the duration of the bar, one must look its time signature, which is a fraction representing the number of events (numerator) of given duration (denominator) that fit in the bar. For example, $\frac{4}{4}$ means that 4 quarter notes can fit in the bar.

Duration In occidental music, an event duration is always relative to the whole note that represents a single note that fills the standard $\frac{4}{4}$ bar. Figure 2.2 shows the division of a bar in whole note, half note, quarter notes, eighth notes and sixteenth notes, each being half the duration of the previous one.

Dot When an event is dotted, it means that its duration is augmented by a half.

Note A note is an event with a pitch.

Rest A rest is an event where no sound is played.

Interval An interval is the vertical distance between two notes, i.e. their pitch difference. Intervals give the colour of the sound.

Chord A chord is a juxtaposition of notes played at the same time.

MIDI MIDI is a protocol dedicated to music and to communication between virtual instruments, controllers and software. The pitch of the note is measured in *midic* or in *midicent*.

Midicent By convention, CAC softwares use MIDI standard to express pitch. The value 60 is used for middle C, 61 for C#, and so on. Raising the MIDI number by one corresponds to raising the pitch by a semitone. To handle microtonal music, some CAC environments like OpenMusic use *midicent* as unit, which correspond to the MIDI number (i.e the semitone) divided by 100.

2.4 Music and constraints

Musical composition often occurs in a framework of compositional rules. Such rules have been stated to express musical knowledge for centuries. According to T. Anders, "prominent examples are the explanations in composition textbooks (e.g. discussing counterpoint, harmony, musical form or instrumentation). These textbooks primarily specify the properties of a suitable result without detailing how to procedurally achieve this outcome." [Anders, 2007]. This way of specifying properties inspires the use of a rule-based approach to generate music.

In this context, constraint programming finds a natural place in music composition. Constraints are an efficient way to simply model a complex musical problem. For example, to compose his *Tombeau de Marin Mersenne* [Amoric, 2006], Michel

Amoric used a set of rules stated by Mersenne that indicates how to create harmony, such as :

- Not repeating the same interval more than three times;
- Not using consecutive fifths or parallel octaves[2];
- Not using the melodic tritone[3];

Amoric declared these constraints to express the Mersenne's rules and used a solver to generate a chords sequence to be used in his musical piece.

Examples of compositional rules are not limited to classical music. Rhythm and blues popularized the continuous use of the backbeat, a "forceful stroke [...] played on beats two and four of a drum pattern" [Beck, 2013], also widely use in rock 'n roll and other popular musics. This can be understood as a rule stating that the second and fourth beats of the bar must be accentuated. Modern popular music also has its rules, with the use of a percussive bass sound mashing every beats of the bar, or the continuous repetition of a short melody. Examples are numerous and it is impossible to provide an exhaustive list of possible rules. This huge corpus of compositional rules motivates the use of a rule-based approach in CAC.

2.5 Previous work

Several tools have been developed in different frameworks and environments to allow the use of constraint-based CAC. The following proposes a list of the most used works on the topic since the second half of the '90s.

PWConstraints M. Laurson developed a rule-based programming language called PWConstraints built on top of Patchwork[4] to solve musical problems

[2] Two parallel octaves are two consecutive octaves and are, as well consecutive fifths, often forbidden.

[3] Other name for augmented fourth or diminished fifth, an interval of exactly six semitones (i.e. three tones).

[4] Patchwork is a tool for computer-aided musical composition that provides a graphical interface to Lisp. It is the ancestor of OpenMusic.

[Laurson, 1996]. It is composed of two constraint systems, PMC — a general-purposed tool for constraint programming — and score-PMC, specialized in solving musical CSPs. Given a fixed rhythmic structure, it finds solutions to fill it with pitch information.

PWConstraint has also been ported to OpenMusic by O. Sandred in 1999 under the name of OMCS.

Situation Developed by A. Bonnet and C. Rueda [Bonnet and Rueda, 1999], Situation is a constraint system for solving CSPs involving musical objects in the OpenMusic (see section 3.1) environment. It was first suited for harmonic problems and later extended to solve rhythmic problems as well. The music is represented as a set of objects containing a certain number of points and by the distances — internal distances between two points of the same objects and external distances between two points from different objects. The user has to specify input parameters such as the desired number of objects, the possible ranges of the points, etc. Objects and points can be interpreted at will. They can be chords and notes or rhythmical plans and impact points, for example. This gives the tool flexibility in addition to its convenience, yet the support for rhythmical structures is limited, according to O. Sandred [Sandred, 2010].

OMClouds This tool extending OpenMusic proposed by C. Truchet [Truchet, 2004] differs from the other in the sense that it focuses on a heuristic search strategy that improves a random solution to a CSP. Ease of use and flexibility are featured, following the philosophy of OpenMusic, by providing a way to express constraints with visual programming and by displaying the current state of the search, allowing the user to stop it at any intermediate partial solution. Compared to the other systems, OMClouds can only solve a limited set of CSPs.

Strasheela This efficient constraint-based music composition system was created by T. Anders [Anders, 2007] and is based on Oz programming language. It can solve complex problems involving rhythmic and harmonic searches at the same time. Strasheela uses a text-based syntax, which requires the composer to have a deep understanding of the programming language.

PWMC O. Sandred developed PWMC [Sandred, 2010] with the motivation of solving musical CSPs involving interdependent constraints on rhythms and pitches in PWGL. PWMC uses three subdomains that form the domain of its variables: pitch, expressed in MIDI numbers; duration expressed in ratios; and metric, representing the signature of a bar as a pair of numbers. A variable can only take value from one of the subdomains. A note is the combination of a pitch and a duration and therefore uses two variables. The user then specifies the number of voices and the number of variables. These variables are then spread among pitch, duration and metric.

This master's thesis aims to explore how to combine some advantages of the different presented tools and to get rid of some drawbacks they carry. The idea is to provide composers with a tool that gathers the following characteristics:

- easy to use: the tool shouldn't require a composer to know any programming languages;
- musically sound: the tool shouldn't ask the composer to decide values that don't have a musical sense such as the number of variables in the CSP;
- general: the tool should be able to provide any kind of melody, i.e. rhythm and pitch together. The main challenge discovered in the review of existing tools is the generation of rhythms without defining in advance the number of variables; the focus will be set on creating a tool that generates rhythms, and that can be extended to include the pitch.

Chapter 4 describes how the general CSP was modelled as well as a sample set of constraints, both destined to be used in the tool, and chapter 6 describes its implementation.

Chapter 3

Tools

This chapter provides a short introduction to the two main tools used within the framework of this master's thesis: OpenMusic and Gecode.

3.1 OpenMusic

OpenMusic [Agon, 1998] is an object-oriented visual programming language developed at IRCAM (Institut de Recherche et Coordination Accoustique/Musique). It is built on top of Common Lisp and CLOS[1] and provides a visual interface for both, allowing easy use of existing CL/CLOS code. OpenMusic can be used to create any-purpose object-oriented and functional visual programs but provides more specialized libraries to implement musical data and behaviour in order to be an environment for music composition.

OpenMusic is composed of a few core elements. The main interface is the *workspace*, which is a virtual desktop containing files and programs as well as a folder ensuring the persistence of the environment. the content of the workspace is file tree of *patches*, *maquettes* and lisp files. Another essential element is the *Lisp Listener*, which displays the results of program evaluations, among others. Most of the work will occur within patches.

[1]Common Lisp is a specification of the functional list-processing language Lisp and includes CLOS that brings object-oriented paradigm to the language.

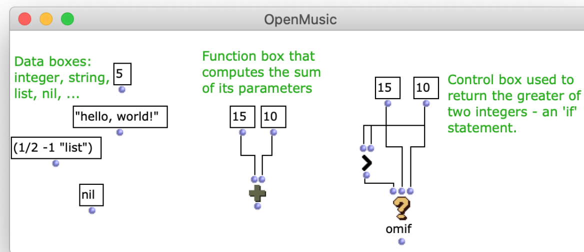


Figure 3.1: Examples of OpenMusic boxes

3.1.1 OpenMusic patches

A patch is a visual representation of a program. Within a patch, objects and functions are represented by boxes with inlets and outlets, interconnected by lines that represent data flow.

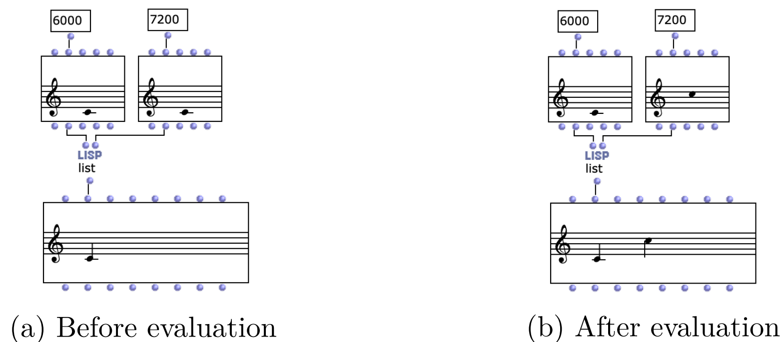
Boxes

Boxes are the main components of the patch; they are functional components of the visual program, i.e. values, objects, functions, and so on.

Data boxes Data boxes represent primitive Lisp types, such as integer, list or string, to name a few. They do not have inlets and have only one outlet: their value. Examples of data boxes are shown on figure 3.1.

Function boxes These boxes are containers for sets of operations and have at least one outlet and generally at least one inlet. The inlets are the parameter of the function contained in the box, and the outlets are the results. Figure 3.1 shows two examples of function, connected to their input values, including a control box. Control boxes are a subcategory of function boxes. They contain specific functions that represent decision and control over the execution path.

Object boxes Object boxes are factories that produce instances of classes. Their inlets are the setters of the class, and their outlets the getters. The first inlet of an



(a) Before evaluation (b) After evaluation

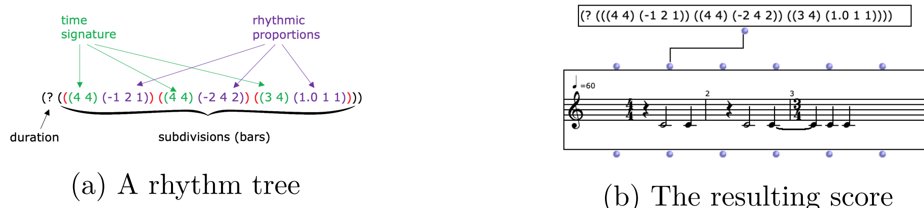
Figure 3.2: Two notes and a chord-seq objects

object box is always the object itself, called *self*. If it used as input, it automatically bypasses the other inlets and sets the values of the object copying the input values. The state of the object is often displayed on the box itself. For example, figure 3.2 (b) shows two *note* objects and a *chord-seq* object. The two notes have their field *midic*, the second inlet, set to respectively 6000 and 7200, and their *midic* outlets are put together in a list passed as parameter to the chord-seq *lmidics* inlet.

A few objects are categorized as *Score objects*. Objects of this category represent notes, chords, sequences of chords, etc. Score objects can be opened in a musical editor that allows modifying the musical elements individually. They also can be played in order to hear their result.

Abstraction boxes These boxes are programs, for example, patches or lisp functions. Such use of patches requires *in* and *out* boxes that allow having input and output for a patch. This allows the user to encapsulate previously made programs and consider them as a black box in new ones.

Note: rhythm trees In OpenMusic, the rhythm is represented as a specific tree. In practice, it is a Lisp list of nested lists. Figure 3.3 shows an example of rhythm tree and its effect on a score object. The first element is the duration of the tree expressed in whole notes. A ? lets OpenMusic compute it itself. In the example, the duration is equal to 11/4 whole notes. The second element is the subdivisions, composed of bars. Each bar has a time signature represented by a pair



(a) A rhythm tree

(b) The resulting score

Figure 3.3: A rhythm tree and its representation on a voice object

(*numerator*, *denominator*) and list of rhythmic proportions. The latest defines the duration of the events, depending on the duration of the bar given by the signature and on the duration on the other events. Different rhythmic proportion lists can have the same meaning, as shown in the example with the first and second bar.

An event duration can be positive to represent a note, negative to represents a rest, and is tied to the previous event if it is followed by a .0.

Evaluation

Connecting boxes alone is not enough to change their state. Boxes need to be evaluated in order to update their state depending on their input. Meanwhile, they are set to their default value. The process of evaluating a box refers to the call to its internal functions and parameters. Figure 3.2 (a) shows the small program of figure 3.2 (b) before it is evaluated.

The evaluation is performed bottom-up and left to right: when a box is evaluated, it needs its input to be evaluated first. The evaluation starts from the evaluated box, and then is propagated to its leftmost inlet; it climbs the connection to the box whose output is connected to the inlet, which is then evaluated, and so on. When all the necessary evaluations are performed for this first inlet, it goes to second, then the next, until all the needed values and object are evaluated. This evaluation tree corresponds to the visual program.

It is possible to change the evaluation mode of a box. This is useful in iterative processes, for example, where a box is evaluated multiple times, but some of all of its parameters should not be reevaluated. In that case, it is possible to lock the evaluation of the parameters.

All these elements provide a rather complete and comfortable environment for musical composition, as well as an ideal field to perform programming experiments in musical composition. OpenMusic has many more components and features that are not covered in this master's thesis.

3.2 Gecode

Gecode [Schulte et al., 2019] is an open-source C++ environment for constraint programming. It is developed and updated since 2005 by C. Schulte et al. and offers a constraint system and an efficient solver. Gecode provides a wide set of features that can be extended to match the users' needs. The following sections present some of these features.

3.2.1 Search space

The space in Gecode is where the variables and constraints are defined. A constraint-based problem is typically modelled by creating a class extending the *Space* class that defines the variables and constraints, and then creating a *SearchEngine* (for example, a DFS) that will search for solutions in the space.

Specialized types of space exist that serve the purpose of finding a best solution to CSP. For example, the *IntMaximizeSpace* is a subclass of *Space* that will maximize a specific cost variable.

3.2.2 Variables and arguments

Gecode provides four types of variables. Integer variables, Boolean variables — which are integers with maximum domain $[0, 1]$ — Float variables and Set variables. The variables come with functions to support their definition, such as expressive ways of defining sets domains.

Gecode also provides argument variables arrays, a system that allows creating intermediate variables, for example, to store the result of an expression. They differs from the normal variables in the sense that they are not updated in the search: they are just holders of an expression and their domain is passively restrained

during constraint propagation.

3.2.3 Constraints

Constraints are posted in the space as well as the variables. Gecode provides a wide predefined set of constraints, including more than 70 constraints from the Global Constraint Catalog[2] and many others. For an exhaustive list of constraints, the reader should refer to the Gecode website[3].

A convenient feature of Gecode constraints system is that to post relation constraints within a very expressive way. The following example[4] illustrates a composite relation that would require to create four sub-relations and three intermediate variables without this convenience.

```
//Express that if x is greater than y, then z is
//equal to the sum of x and y.
rel(*this, (x > y) >> (z == x + y));
```

3.2.4 Branching

Variables that need their value to be searched are branched using a dedicated branching function. This function allows to choose a strategy on which variable to assign next and to what value, and can also include support for symmetry breaking. Branching strategies are posted in the *Space* class.

3.2.5 Search

Gecode provides search engines to perform the actual assignation of variables depending on the strategies defined by the branching options. In practice, at each step of the search, the space is copied. Then the variables in the copy are updated according to the constraints and to which variable has been assigned.

[2]Global Constraint Catalog is a report that aim to present and describe global constraints. It can be found at `http://sofdem.github.io/gccat/gccat/index.html` (visited the 31st of May 2020.

[3]`https://www.gecode.org/`

[4]The first argument of the shown function is the space in which the variables and constraints are defined.

Chapter 4

Models and constraints

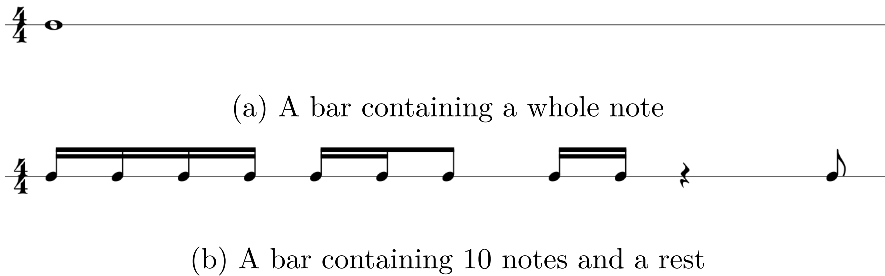
Most of the previous work has put a strong focus on solving pitch problems, which makes sense since it is a very complex process of musical composition, where most of the rules of music take action. While O. Sandred and T. Anders include rhythm in their CSPs as well [Anders, 2007, Sandred, 2010], an issue remains: the number of variables somehow have to be known in advance, which limits the possibilities for the output rhythm.

There is much room left to explore and model rhythm CSPs. For this reason, this master's thesis focuses exclusively on rhythm generation. The first part of this chapter proposes three approaches to model rhythm CSPs based on a core problem: the number of variables. It is followed by an experiment to decide which model is the most efficient and habitable. The last part introduces a sample set of additional constraints that can extend the base CSP expressed in the model deemed most promising.

4.1 Modeling a rhythm CSP

4.1.1 The rhythm problem

Rhythm is a particular issue when it comes to model a musical CSP. The number of events, i.e. notes or rests, can greatly vary in a melody or even in a single bar. For example, the rhythm sequences (a) and (b) in figure 4.1 have the same duration, and yet have a very different number of events. Figure 4.1(a) shows a bar filled



(a) A bar containing a whole note

(b) A bar containing 10 notes and a rest

Figure 4.1: Two bars of same duration with different numbers of events

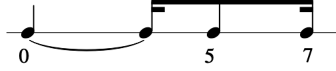
with only one note, while figure 4.1(b) shows a bar containing 11 events. This problem may seem trivial, but taken that there is no theoretical lower bound to event duration, and the upper bound being only defined by the total duration of the rhythm sequence, each sequence could have an infinity of possible fillings. And, as the composer is not supposed to know in advance the number of events she wants in her future score, there is no way of knowing exactly how many variables will be needed to represent rhythm. The issue is that a constraint solver can not solve itself the number of variables it needs (it would require a previous solver whose goal is to determine this number).

Once the rhythm variables — they can be seen as a horizontal sequence of variables — are created, the pitch variables — vertical sequences of variables — need to be created as well to find notes or chords. But this task is more straightforward since chords are typically composed of one to four notes, sometimes more, up to 8 notes, which is a drastically smaller range than the rhythm. However, handling pitch is left for future researches.

4.1.2 First approach: avoid the problem

Some rhythmic problems don't require to represent events by variables, but rather to find a parameter of an operation on an input rhythm. For example, to write *Aschenblume* [Amoric, 2006], the composer Mauro Lanza creates rhythmic sub-patterns from an original rhythm, using the modulo operator. Let m an integer greater than 1 and smaller than the duration of the original pattern p. The position of each pulse in p modulo m gives a sequence of integers which, when the duplicates

(a) The original pattern, with pulses at position 0, 5, 13, 15 and 21



(b) The sub-pattern generated with a m equal to 8.

(c) The new pattern generated from the repetition of the subpattern (the last repetition must be truncated to fit in the original duration)

Figure 4.2: An example of modulo subpattern with $m = 8$

are removed, gives a new sequence of pulse positions of duration m. A sub-pattern is generated by inserting notes at the indicated positions. This sub-pattern is then repeated until it reaches the length of p, and truncated if necessary. Figure 4.2 shows an example with $m = 8$. The positions are in terms of the denominator of the time signature (here 16, which means that position x corresponds to the $(x + 1)^{th}$ sixteenth note of the bar). Mauro Lanza then searches a m that generates an output with certain properties, for example, that minimizes the rhythmic density.

The problem can be modeled as a CSP using only a few variables (values are taken from the example of figure 4.2):

Known values

$p = 0, 5, 13, 15, 21$, the sequence of original pulses.

$n = 5$ the number of original pulses.

$drt = 22$ the duration (in sixteenth notes) of the original pattern.

Variables

m an integer variable of domain $[2, drt/2]$, the modulo.

d an integer variable of domain $[0, drt[$, a prediction of the output rhythmic density.

Constraints

$C(d, p_0 ... p_{n-1}) = d = distinct(p_0\%m ... p_{n-1}\%m)$

The function $distinct(x)$ returns the number of distinct values in x. Given this CSP, it suffices to minimize d, and it will output values for m that can be passed as a parameter in a function that creates the modulo sub-pattern and fit it in the duration of the original sequence.

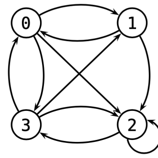
This approach can bypass the need of knowing the number of needed variables to compose the rhythm sequence. But while it suits well specific problems like the Mauro Lanza's modulo sub-pattern example, creating a tool that provides a way to build such CSP and auxiliary methods that come with it (such as building the sub-patterns and fit them in the pattern) basically means asking the composer to become a programmer, or to limit the number of available CSPs to a library of predefined ones, which are two strong limitations.

4.1.3 Second approach: rhythm from a graph

The idea comes from the exploration of the possibilities that Gecode has to offer, notably to post constraints on integer variables with the use of graphs. From there, the goal is to generate a rhythm sequence from a graph. Gecode gives the possibility to post constraints called *circuit* on an array of integer variables [Schulte et al., 2019]. These constraints consider the array as a graph and use the values of the variables of the array as its edges. For an array x of n variables and for $0 \leq i, j < n$, the graph has nodes from 0 to $n-1$, and if $j \in x_i$, then the graph contains an edge $i \to j$. This automatically constrains the domain of each variable in x to range between 0 and j.

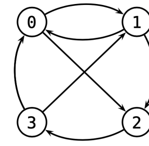
On such an array, the constraint $circuit(x)$ will constrain the values of x in a way their corresponding edge in the graph form a hamiltonian circuit.

It is possible to add the notion of costs for the edge. The user can define a matrix that represents the cost of each edge of the graph. For a cost matrix c, two integer variables arrays x and y and an integer variable z, $circuit(c, x, y, z)$ constrains the value of x to behave as described above, y to be the costs of the circuit and z to be the sum of the elements in y, i.e. the total cost of the circuit. Figure 4.3 from [Schulte et al., 2019] shows an example of variables representation with a graph



$x_0 \in \{1,2,3\}, x_1 \in \{0,2\}, x_2 \in \{2,3\}, x_3 \in \{0,1,2\}$
$y_i \in \{-100,\dots,100\} \quad (0 \le i < 4)$
$z \in \{-100,\dots,100\}$,

(a) Representation of the variables by the graph.



$x_0 \in \{1,2\}, x_1 \in \{0,2\}, x_2 \in \{3\}, x_3 \in \{0,1\}$
$y_0 \in \{3,5\}, y_1 \in \{4,9\}, y_2 \in \{5\}, y_3 \in \{-7,8\}$
$z \in \{5,\dots,27\}$,

(b) Result after propagation of the *circuit* constraint.

Figure 4.3: An example of the graph representation of the variables and of the propagation of the *circuit* constraint with cost.

and the result when the constraint *circuit* is applied with a cost.

Notes representation

Reasoning on the cost of the edges allows us to represent notes. If we define the cost to be the duration of a musical event, the resulting array of values from y is a sequence of duration that can correspond to a rhythm. The sequence is always of the same size, but the total duration varies from $|x| * min(y)$ to $|x| * max(y)$.

As an example, consider the user is asked to choose a list that represents the allowed notes durations in the problem, and she chooses (1/2 1/4 1/8 1/16 1/32). The output with the shortest duration will then be the sequence (1/32 1/32 1/32 1/32 1/32 1/32), and the longest one will be (1/2 1/2 1/2 1/2 1/2 1/2), 16 times longer than the previous one. The problem of the number of variables is then still present. Still, having in mind that rhythm is often composed of a repeated short pattern, a solution is to copy p times the output sequence given by the values of y, p being the smallest integer such that $duration(sequence) \geq D$ where D is a duration chosen by the user or a variable, and then truncate the result to obtain a sequence of duration D.

When truncating, it must be taken into account that not all note durations are allowed. For example, with a total duration $D = 1$, a duration domain $dom_d = (1/2$

1/4 1/8 1/16 1/32) and the output sequence $o = (1/4\ \ 1/4\ \ 1/8\ \ 1/8\ \ 1/16$ $1/2)$, we have to truncate o because its total duration is 21/16, but cannot simply reduce 1/2 to 3/16 because the latest (which is the remainder) is not in dom_d. We must choose a sequence of values from dom_d with a sum equal to the remainder. A strategy must be selected to choose which values will be taken from the dom_d to replace the remainder. The values could be selected randomly as long as they fit in the gap, or the largest/smallest possible value first, etc. Unfortunately, no matter the strategy used, it can sometimes be impossible to find a combination of values from dom_d that perfectly fill the gap, and the program might be forced to choose a value outside of the domain in order to respect the duration.

Using list processing once the CSP has found a solution can be problematic. It significantly limits the set of possible constraints. Worst, it can end up not respecting a constraint posted by the user. Let's consider the user wants that a note cannot be of the same length as the previous one. It would be expressed as $\forall i \in [1, n[:\ y_i \neq y_{i-1}$. She chooses $dom_d = (1/2\ \ 1/4\ \ 1/8\ \ 1/16\ \ 1/32)$ and $D = 1$. Then the output sequence $o = (1/8\ \ 1/4\ \ 1/16\ \ 1/8\ \ 1/16\ \ 1/8)$ is a solution to the CSP. Its duration is 6/8, thus we copy o one time and truncate the result, and obtain $o^* = (1/8\ \ 1/4\ \ 1/16\ \ 1/8\ \ 1/16\ \ 1/8\ \ 1/8\ \ 1/8)$. The constraint posted by the user is violated, and therefore the result should not be acceptable. But as the multiplication of the output sequence is performed once the CSP has found a solution, there is no way to consider all the constraints put by the user at this stage.

Rests representation

Another issue is that of the rests. The variable array x can be used to represent rests. Let the value x_r included in the domain of the variables in x. If the value of a variable x_i is lower than x_r, then it is a rest; otherwise, it is a note. x_r can also be defined as a variable to let the user post constraints on the number of rests.

Summary

Rhythms can be generated from a graph using the possibilities of Gecode. To solve the problem of the number of variables, an output sequence from the CSP is considered as a pattern to repeated a number of times that makes it fit in the duration of the final output rhythm. Event durations are represented by the costs of the edges in the graph, and a threshold on the values of the nodes decides if they are rest or not. While it does provide a way of generating rhythms with a small set of variables (i.e. one edge variable and one node variable a for each possible event duration), it limits the set of possible constraints heavily and makes them hard to express, and may even not respect them eventually, which makes it an unacceptable way of modelling the rhythms.

4.1.4 Third approach: Events described by their position and duration

Representing the musical event as a tuple of values describing its state allows posting constraints directly on each component of its state. The chosen representation of the event is inspired by the Strasheela music representation [Anders, 2007]: $event(pos, drt)$ where pos is the position of the note in the note sequence and drt is its duration. This representation can be extended to a note by adding a pitch value: $note(pos, drt, pitch)$, or even to a chord by adding a list of pitches: $chord(pos, drt, pitches)$. The notion of event is sufficient here because we focus only on the rhythm. This section presents the step by step construction of the model.

Fixed number of events

The first model is an array of n events, with their order in the array corresponding to the order in the resulting sequence. This way, for a duration array x, we always know that $pos(x_i) = i$. Because representing the positions this way means that positions are relative to each other in the sequence, we have to consider the representation of rests. Otherwise, we don't have any clue that a note is supposed to be played directly after another, or if a long silence comes in between. As the

positions in this model are related to each other, it is important to know if the event at position i is a rest or a note. We define here a rest as a negative duration. Hence, the following events $e(p, 4)$ and $e(p, -4)$ are to events of equal duration at position p, the first being a played note and the second being a rest. A minimal set of constraints can define a general way to build rhythms given the following data:

Let N and D two integers representing the signature N/D of the desired bar and s the inverse of the shortest desired note duration (for example, if the user wants the shortest possible note to be a sixteenth note, then s is equal to 16) We define here a *tick* to be the unit of time equal to the duration $1/s$. Let *duration* the total duration of the bar in terms of the shortest possible note, i.e. the number of ticks that fit in the bar: $duration = s * N/D$. Then the following CSP describes all possible rhythm bars of n musical events with signature N/D and s as their shortest possible note length:

Variables

$drt_0...drt_{n-1}$ an array of n integer variables with domain $[-duration, duration] \setminus \{0\}$.

Constraints

$C_1(drt_0...drt_{n-1}) = \sum_{i=0}^{n-1} |drt_i| = duration$

One should note that it works only for a predetermined fixed number n of events. We aim to get rid of this restriction because specifying the number of events in a bar does not carry much musical sense.

Bounded number of events

To extend the problem, let's consider a range of events. The core CSP remains essentially the same, except that this time, we ask the user to choose a minimum number of events n_{min} and a maximum n_{max}. Furthermore, we add the duration 0 to the domain of the variables to help us determine whether an event is to be included in the output sequence or not: an event of duration 0 is considered non-existent. We need another constraint that will express that the number of variables in drt equal to 0 is lower or equal to the difference $n_{max} - n_{min}$. The CSP becomes:

Variables

$drt_0 ... drt_{n_{max}-1}$ an array of n_{max} integer variables with domain $[-duration, duration]$.

Constraints

$C_1(drt_0 ... drt_{n_{max}-1}) = \sum_{i=0}^{n-1} |drt_i| = duration$

$C_2(drt_0 ... drt_{n_{max}-1}) = count(drt_i ... drt_{n_{max}-1}, 0) \leq n_{max} - n_{min}$

The function $count(vars, x)$ represents the number of values in $vars$ that are equal to x. What remains to be done after a solution is found is to exclude the duration variables set to 0. This raises an issue of symmetries: some solution will be equivalent to each other. The larger the range, the more there will be equivalent solutions. For example, the following sequences will all be equivalent:

$$\{1, 2, 1, 1, 0\} \equiv \{1, 2, 1, 0, 1\} \equiv \{1, 2, 0, 1, 1\} \equiv \{1, 0, 2, 1, 1\} \equiv \{0, 1, 2, 1, 1\}$$

The number of symmetries depends on the chosen n_{max} and on the difference d ranging from 0 to $n_{max} - n_{min}$. If no other constraints are specified, we see that the number of possible symmetries for a single output sequence of length $n_{max} - d^*$ (i.e. there is d^* events that are non-existent) is $A_{n_{max}}^{d^*}$. An example solution could be to force the non-existing events to be put at the beginning or the end of the sequence with additional constraints. The drawback would be that it limits the use of certain constraints the user could want to use. For example, the constraint $\forall i \in [1, n[: \ drt_i < drt_{i-1}$ automatically implies a rhythm filled with rests if we cast all the non-existing durations at the beginning of the sequence.

Now, asking the user to choose a minimum and a maximum number of notes does not make more musical sense than asking her to choose a fixed amount. The ideal would be to let her choose a characteristic of the rhythm that allows computing these bounds. Note that with the signature N/D and the shortest allowed duration s, we can already find a minimum number n_{min} of events equal to 1, the situation when there is a single event of the same duration as the bar, and a maximum n_{max} equal to $s * N/D = duration$, the case when there are only events of the shortest duration (i.e. each tick starts an event with a duration of 1 tick). This represents a difference d ranging from 0 to $(s-1) * N/D$, resulting in a huge possible amount of symmetries if the user is looking for a long bar with small events.

There are several options to tackle this issue. The first could be to ask the user to choose a longest duration l that would increase the value of n_{min} to $duration/l$. The second one is to create a density handle represented by a float h_d ranging from 0 to 1 that the user can set so that $n_{max} = h_d * duration$. Finally, both previous options can be combined.

Variation: explicit position

Until now, we considered an event position to be its position in the array, i.e. $pos_i = i$, meaning that the position of an event is relative to the position of the other events. But what if the user wants to emphasize a particular note of the bar, for example with a backbeat accentuating the second and fourth pulses? Currently, it is required for a given event to compute the sum of the durations of the previous events to find its position.

A simpler solution is to explicitly express the position with a variable. We then need a second variable to represent the event. The CSP will become:

Variables

$drt_0 ... drt_{n_{max}-1}$ an array of n_{max} integer variables with domain $[-duration, duration]$.

$pos_0 ... pos_{n_{max}-1}$ an array of n_{max} integer variables with domain $[0, duration[$.

Constraints

$C_1(drt_0 ... drt_{n_{max}-1}) = \sum_{i=0}^{n-1} |drt_i| = duration$

$C_2(drt_0 ... drt_{n_{max}-1}) = count(drt_0 ... drt_{n_{max}-1}, 0) \leq n_{max} - n_{min}$

$C_3(drt_0 ... drt_{n_{max}-1}, pos_0 ... pos_{n_{max}-1}) = \forall i \in [1, n_{max}[: \; pos_i = pos_{i-1} + drt_{i-1}$

$C_4(pos_0) = pos_0 = 0$

The constraint C_3 forces an event to occur exactly when the preceding event stops, and the constraint C_4 ensures there is an event at the beginning of the sequence. As a consequence, the position pos_i of an event e_i doesn't mean the e_i is the pos_i^{th} element of the sequence anymore, but instead that e_i starts at the pos_i^{th} tick of the bar.

While this base CSP represents rhythms correctly, the domain of the position variables makes it not that flexible: as it makes it impossible to have non-existing

events after the event at position $duration - x$ and of duration x, some combinations of additional constraints may be affected, or even not possible to satisfy. A small change of domain from $[0, duration[$ to $[0, duration]$ and the additional constraint $\forall i \in [0, n_{max}[:\ pos_i = duration => drt_i = 0$ are enough to avoid this.

Variation: Implicit duration

Another variation is to express only the position explicitly and to deduce the duration. Obviously, the duration of the event that occurs at pos_i ticks is equal to $pos_{i+1} - pos_i$. The last event duration is given by $duration - pos_{n_{max}}$ This way of computing the duration doesn't allow to know if the event is a rest or not. A new variable is needed, with value 0 or 1, to describe this. In order to know that an event is non-existent, its position should be compared to the previous position. If they are equal, the event has a duration of 0 and therefore does not exist. The CSP becomes:

Variables

$pos_0 ... pos_{n_{max}-1}$ an array of n_{max} integer variables with domain $[0, duration]$.
$rest_0 ... rest_{n_{max}-1}$ an array of n_{max} boolean variables

Constraints

$C_1(pos_0 ... pos_{n_{max}-1}) = count(pos_1 - pos_0 ... pos_{n_{max}-1} - pos_{n_{max}-2}, 0) \leq n_{max} - n_{min}$
$C_2(pos_0 ... pos_{n_{max}-1}) = \forall i \in [1, n_{max}[:\ pos_i \geq pos_{i-1}$
$C_3(pos_0) : pos_0 = 0$
$C_4(pos_0 ... pos_{n_{max}-1}) = \forall i \in [1, n_{max}[:\ pos_i = duration => pos_i = pos_{i-1}$

Constraints C_1 ensures that the number of non-existing events is well fitted, and C_2 constrains the positions to be greater or equal than the previous ones, and their domain prevents them from being equal or greater than the duration, which means that the sum of the durations will never exceed the total duration. C_3 forces the first position to be at the first tick of the sequence (it is always an existing event in this model). Finally, C_4 prevents positions of existing events to be equal to duration, while allowing it for non-existing events.

Figure 4.4: Example of backbeat rhythm

Comparison of the variations: the backbeat example

In order to compare the three variations, let's take the backbeat example. Constraints will be posted for each variation to express the fact the positions corresponding to the second and fourth pulses of the bar are accentuated (in this example, the accentuation will be to force them to be at least longer than the duration of a pulse). Figure 4.4 show an axemple of output from the backbeat CSPs.

The first task is to find the second and fourth pulses. We need to find the duration of a pulse and which tick corresponds to which pulse. It is quite simple: the duration d_p of a pulse is given by $d_p = s/D$, and therefore the ticks that correspond to the beginning of a pulse are given by $(i-1) * d_p,\ i \in [1, N]$.

Variation 1 As this model does not explicitly express the position, we have to compute the position in ticks $p_i = \sum_{j=0}^{i-1} drt_j$. The constraints are:

$$C_{b1}(drt_1...drt_{n_{max}-1}) = \forall i \in [1, n_{max}[:\ \sum_{j=0}^{i-1} drt_j = d_p \Rightarrow drt_i \geq d_p$$
$$C_{b2}(drt_1...drt_{n_{max}-1}) = \forall i \in [1, n_{max}[:\ \sum_{j=0}^{i-1} drt_j = 3 * d_p \Rightarrow drt_i \geq d_p$$
$$C_{b3}(drt_1...drt_{n_{max}-1}) = count(\sum_{i=0}^{1} drt_i ... \sum_{i=0}^{n_{max}-1} drt_i, dp) = 1$$
$$C_{b4}(drt_1...drt_{n_{max}-1}) = count(\sum_{i=0}^{1} drt_i ... \sum_{i=0}^{n_{max}-1} drt_i, 3 * dp) = 1$$

Constraint C_{b1} allow to compute the positions; C_b2 and C_b3 constrains the durations at positions corresponding to the second and fourth pulses to last longer than a pulse (starting at pos_1 because the first position pos_0 is always on the first pulse); C_b4 and C_b5 ensure there is an event starting at positions corresponding to the second and fourth pulses. It is inconvenient to have to compute the sum of the preceding durations to find the position. Still, it is counterbalanced by the fact that it allows posting constraints on positions without having them represented as variables.

Variation 2 The constraints for this model are straightforward:

$$C_{b1}(pos_1...pos_{n_{max}-2}) = count(pos_1...pos_{n_{max}-2}, d_p) = 1$$
$$C_{b2}(pos_1...pos_{n_{max}-1}) = count(pos_1...pos_{n_{max}-1}, 3 * d_p) = 1$$
$$C_{b3}(pos_1...pos_{n_{max}-1}, drt_1...drt_{n_{max}-1}) = \forall i \in [1, n_{max}[: \; pos_i = d_p \Rightarrow drt_i \geq d_p$$
$$C_{b4}(pos_1...pos_{n_{max}-1}, drt_1...drt_{n_{max}-1}) = \forall i \in [1, n_{max}[: \; pos_i = 3 * d_p \Rightarrow drt_i \geq d_p$$

Constraints C_{b1} and C_{b2} constrain the sequence to include an event at the second pulse while constraints C_{b3} and C_{b4} constrain the duration of the event at the second pulse lasts longer than a pulse duration.The expression is easier, but the number of variables is doubled compared to the first variation. Even though the domain of the supplementary variables is half the domain of the durations, this can significantly increase the number of variables to explore.

Variation 3 In this model, constraining the position is straightforward, and constraining the duration of the event at a given position means constraining the next position. In our example:

$$C_{b1}(pos_1...pos_{n_{max}-2}) = count(pos_1...pos_{n_{max}-2}, d_p) = 1$$
$$C_{b2}(pos_1...pos_{n_{max}-1}) = count(pos_1...pos_{n_{max}-1}, 3 * d_p) = 1$$
$$C_{b3}(pos_1...pos_{n_{max}-2}) = \forall i \in [2, n_{max} - 1[: \; pos_{i-1} = d_p \Rightarrow pos_i \geq pos_{i-1} + d_p$$
$$C_{b4}(pos_1...pos_{n_{max}-1}) = \forall i \in [2, n_{max} - 1[: \; pos_{i-1} = 3 * d_p \Rightarrow pos_i \geq pos_{i-1} + d_p$$
$$C_{b5}(pos_{n_{max}-1}) = pos_{n_{max}-1} = 3 * d_p \Rightarrow pos_{n_{max}-1} \leq duration - d_p$$
$$C_{b6}(pos_1...pos_{n_{max}-1}, rest_1...res_{n_{max}-1}) = \forall i \in [1, n_{max}[: \; pos_{i-1} = d_p \Rightarrow \neg rest_i$$
$$C_{b7}(pos_1...pos_{n_{max}-1}, rest_0 1...res_{n_{max}-1}) = \forall i \in [1, n_{max}[: \; pos_{i-1} = 3 * d_p \Rightarrow \neg rest_i$$

Constraints C_{b1} and C_{b2} are the same as in variation 2; C_{b3} and C_{b4} constrain the position following the second or fourth pulse to occur after a time greater than the duration of a pulse., and C_{b5} plays this role for the last position; finally, C_{b6} and C_{b7} ensure that the events at the desired pulses are not rests. The constraints are easy to express, but there are many cases to think of than with the other variations. This model has the same number of variables as the one in variation 2, but the $rest$ variables have a small domain (0 or 1) and have a lesser effect on the increase of the search time and space.

Summary

This subsection provided a step by step description of how the model of musical events as tuples $event(pos, drt)$ is built. To tackle the issue of the number of variables, the risk is taken to have (potentially many) unused variables in the CSP, and to try to limit their number with musically-sound parameters. The outcome of the process is that three possible models have drawbacks and advantages in terms of search time and space and of ease of expression. These variations are (1) represent only the duration of the event with variables and deduce the position, which requires the fewer variables — and thus, the smallest theoretical search time and space — but where constraints are the hardest to express; (2) represent both position and duration of events with variables, making the expression of constraints the easiest but theoretically requiring the most search time and space; and (3) represent only the position of the event with variables and deduce the duration, which is between the two first variations in terms of expression and search time and space. A test comparing the efficiency of these three variations is provided in section 4.2. This approach seems to be the most promising among the three approaches presented.

4.1.5 Conclusion

This section showed and discussed three ways of modelling rhythm CSPs. The first one is by creating any CSP with an output that can be interpreted as a rhythm. While it can be efficient for specific problems, it has the inconvenient that each CSP produced that way is specific to the particular situation, and can not easily be extended. The second way was to use graphs to generate a rhythm pattern destined to be repeated until it fits in the right duration. It allowed highlighting that repetition of such generated patterns could lead to constraints violation. The third model used a representation of each musical event of the bar by a tuple containing variables. It allows the most flexibility of expression amongst the three methods but may result in a considerable search tree.

In the last model, three way of expressing the tuples were explored: the first one is to explicitly express the duration of the events and deduce their position,

Figure 4.5: An example of ternary backbeat that can be produced by the test programs

requiring fewer variables but making it difficult to express constraints. The second representation explicitly expresses durations and positions. It allows to express constraints easier than the first method but requires the most variables. Lastly, the third repression explicitly expresses the positions and deduces the durations. Its ease of constraint expression and its number of variables are situated between the two previous representations.

The next section shows some tests that have been conducted in order to decide which of the three variations should be used for the remaining of this work.

4.2 Comparison of the models

A small experiment has been conducted to decide which of the three variants of the model will be used to implement the rhythm part of the tool. A simple problem, similar to the backbeat example, has been implemented with Gecode in each of the three variations.

In the test example, the program is tasked to find a single $\frac{12}{8}$ bar, with an accent on the fourth and ninth eighth notes (which correspond to a ternary backbeat). The note is accentuated by letting it last exactly three eighth notes. Figure 4.5 shows a solution rhythm. The shortest authorized event length is $\frac{1}{16}$, and the bar can contain 5 to 8 events. The source code of the CSPs can be found in appendix C.1.

A test consisted of letting the CSP find all solutions to the problem and measuring the execution time, the number of distinct solutions found, and the number of equivalent solutions found. The test has been conducted 20 times for each experiment. The results can be found in table C.1 of appendix C.1. In the following, variations will be referred to as $V1$ for the first variation (only explicit durations), $V2$ for the second variation (explicit position and duration) and $V3$ for the third

one (explicit position only).

The three CSP find 108080 solutions to the problem, and find the same solutions. Using $V1$ or $V2$ finds 30,800 equivalent solutions, while $V3$ finds 68,960 equivalents solutions.

Regarding the execution time, $V2$ is always the slowest and has an average search time of 533.25 ms, $V1$ is always between the two others with an average search time of 405.6 ms and $V3$ is always the fastest, with an average of 162.8 ms.

While $V3$ is clearly faster than the others, it generates much more symmetries, which are parasite solutions. Besides, it revealed itself to be the most error-prone model of the three variations because of the many limit cases to think of. A natural choice then leads to $V1$ variant since the only difference in the results is that it is faster than the $V2$ model. The choice seems even better with the ability of Gecode to let the user define argument variables that will serve to store the positions and makes it as habitable as $V2$.

4.3 Constraints on rhythm

The previous sections described the general model of the rhythm CSP and an experiment to decide which variation of the model to use. For the remaining of this chapter, the term rhythm CSP will refer to $V2$, the expression of events with explicit position and duration. This variation as been chosen despite the results of the above experiments because it is more readable than the others, and because pos_i can be understood as an abstraction of $\sum_{j=0}^{i-1} drt_i$.

While this model already include a small set of constraints that define the general behaviour of a rhythm sequence, it is possible to add constraints to the CSP in order to further specify its outcome.

4.3.1 Sample constraints

The following constraints are chosen in the mindset that a composer would probably write a short rhythm and use rules to modify it, but the set of possible constraints is not limited to this particular approach. To represent this, let's consider that $pulse^p$ and $pulse^d$ are two lists of integers, respectively the positions of the pulses (in ticks) of the input rhythm and their durations, and n_{pulses} is the number of pulses of the input.

Simple constraints examples

At least pulses ($C_{\geq p}$) This constraint forces the output rhythm to contain at least the pulses of the input. It is expressed as follow:

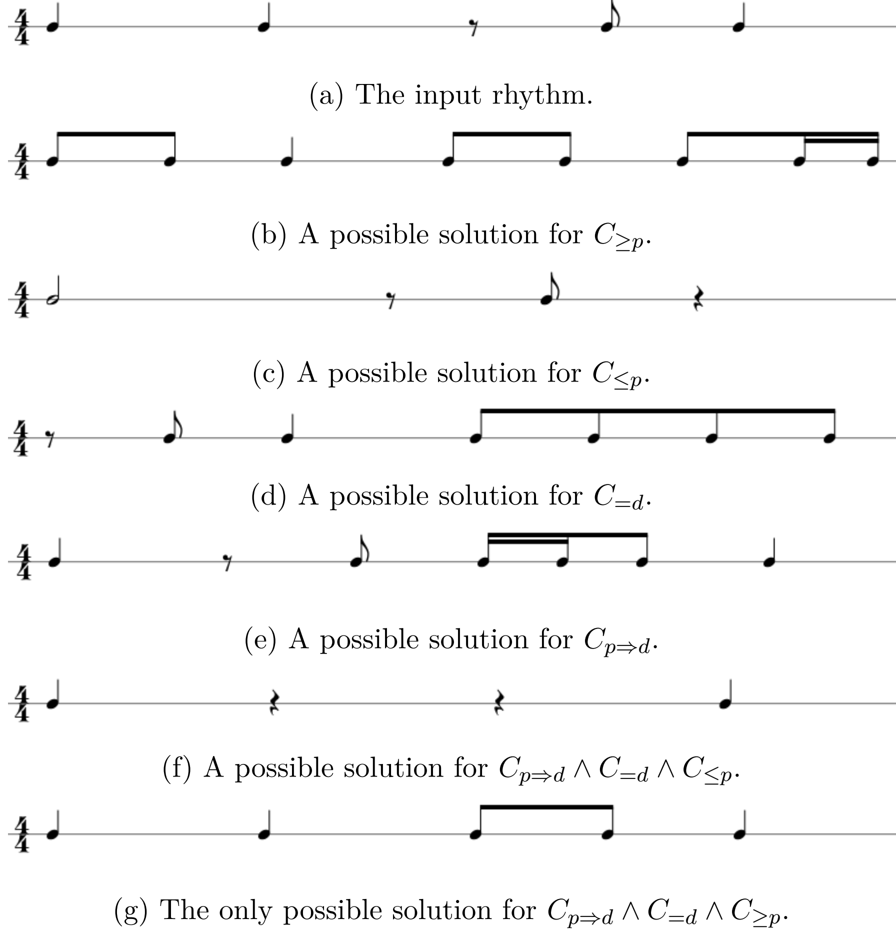
$C_{\geq p1}(pos_0...pos_{n_{max}-1}) = \forall p \in pulse^p \ : \ count(pos_0...pos_{n_{max}-1}, p) = 1$
$C_{\geq p2}(pos_0...pos_{n_{max}-1}, drt_0...drt_{n_{max}-1}) = \forall p \in pulse^p, \ \forall i \in [0, n_{max}[: \ pos_i = p \Rightarrow drt_i > 0$

$C_{\geq p1}$ constrains the output to have at least one position equal to each input pulse position and $C_{\geq p2}$ ensures that the events at these positions exist ($drt_i \neq 0$) and are not rests ($drt_i > 0$). It is obviously also possible post this constraint with only a subset of the input pulses.

At most pulses ($C_{\leq p}$) This constraint forces the output rhythm to contain at most the pulses of the input. It is expressed by changing the domain $[0, duration[$ of the positions variables by $pulse^p$.

Same durations ($C_{=d}$) This constraint forces the output rhythm to contain only durations of the pulses of the input. It is expressed by changing the domain $[-duration, duration]$ of the positions variables by $[-duration, 0] \cup pulse^d$.

Same duration at pulse ($C_{p \Rightarrow d}$) This constraint ensures that the notes that occur at the positions of the input pulses have the same durations as the input pulses. It is expressed as follow:



(a) The input rhythm.

(b) A possible solution for $C_{\geq p}$.

(c) A possible solution for $C_{\leq p}$.

(d) A possible solution for $C_{=d}$.

(e) A possible solution for $C_{p \Rightarrow d}$.

(f) A possible solution for $C_{p \Rightarrow d} \wedge C_{=d} \wedge C_{\leq p}$.

(g) The only possible solution for $C_{p \Rightarrow d} \wedge C_{=d} \wedge C_{\geq p}$.

Figure 4.6: Examples of solutions of the base CSP extended with simple constraints.

$C_{p \Rightarrow d}(pos_0 ... pos_{n_{max}-1}, drt_0 ... drt_{n_{max}-1}) = \forall i \in [0, n_{pulses}[, \; \forall j \in [0, n_{max}[\; : \; pos_j = pulse_i^p \Rightarrow drt_i = pusle_i^d$

These simple examples can be combined to express rules like *the output must have the same pulse positions but their duration are exchanged* or *the output must keep some pulses and their duration and delete others*. Figure 4.6 shows example of outputs of the base CSP augmented with one or a combination of these simple constraints. It is easy to imagine other simple constraints, like keeping the same number of pulses, or using only durations that are not present in the input. It is also possible to simply reason on rests the same way.

(a) The input rhythm, in which the pattern must be inserted in place of the first pulse.

(b) The pattern to insert in place of the first pulse.

(c) The solution of the insert pattern constraint.

Figure 4.7: Example of the base CSP extended with the insert pattern constraint. The pattern has a duration of $\frac{1}{4}$ and the pulse where it must be inserted has a duration of $\frac{1}{2}$, so its events durations are doubled.

A more complex example

A less trivial example is to insert a given pattern in the input in place of the chosen pulse. The duration of the pattern is extended or shrunk by multiplying all its notes duration by a ratio r equal to the duration of the pulse divided by the duration of the pattern. Figure 4.7 shows an example where a pattern is inserted at the first pulse, which is twice as long at the pattern. Let $pulses^p$ the positions of the pulses in the input, $rests^p$ the positions of the rests in the input, ptr the pattern expressed as a list of n durations and p the pulse to replace by ptr (all values are in ticks), and let $drt(x)$ the duration of an event x in the input. We have $r = \frac{drt(p)}{\sum_{k=0}^{n-1} ptr}$, as well as the following constraints:

$C_{ptr1} = C_{\geq p}$

$C_{ptr2} = C_{\geq r}$

$C_{ptr3}(pos_0...pos_{n_{max}-1}, drt_0...drt_{n_{max}-1}) = \forall i \in [0, n_{max}[,\ \forall j \in [0, n[\ :\ pos_i = p \Rightarrow drt_{i+j} = r * ptr_j$

$C_{ptr4}(pos_0...pos_{n_{max}-1}, drt_0...drt_{n_{max}-1}) = \forall p' \in pulses^p \setminus p \forall i \in [0, n_{max}[\ :\ pos_i = p' \Rightarrow drt_i = drt(p')$

$C_{ptr5}(pos_0...pos_{n_{max}-1}, drt_0...drt_{n_{max}-1}) = \forall r' \in rests^p \forall i \in [0, n_{max}[\ :\ pos_i = r' \Rightarrow drt_i = drt(r')$

Constraint C_{ptr1} is the same as $C_{\geq p}$: it ensures that there is at least the same number of pulses in the solution. $C_{ptr2} = C_{\geq r}$ expresses the same with rests, to prevent the CSP to insert new notes in place of rests in the solution. C_{ptr3} is responsible for the insertion of the pattern and C_{ptr4} and C_{ptr5} constrain the solution to have the same duration for events other than p as in the input. Note that if a shrunk note of *ptr* is shorter than the shortest desired note duration s (introduced in section 4.1.4, the problem has no solution, because the constraint forces a duration variable to have a value outside of its domain.

4.3.2 Limitations

A limitation of this model and its constraints, in terms of usability for a composer, is that it becomes quickly over constrained. For example, keeping the rhythm close to the input requires many constraints. Therefore, if a tool uses it, it prevents the user to first constrain the rhythm to keep it simple, and then add constraints to make it more complex. This forces a tool using this model to provide mechanisms to restart a search from the output of the previous found solution to take the new constraints into account.

4.4 Conclusion

This chapter showed that rhythm CSPs are a challenging topic and proposed and discussed three approaches to model them. The first one is by creating any CSP with an output that can be interpreted as a rhythm. While it can efficiently specific problems, it has the inconvenient that each CSP produced that way is specific to the particular situation, and can not easily be extended. The second approach is to use graphs to generate a rhythm pattern destined to be repeated until it fits in the right duration. It allowed to highlight that repetition of such generated patterns could lead to constraints violation. The third model uses a representation of each musical event of the rhythm sequence by a tuple containing variables. It allows the most flexibility of expression amongst the three method, but may result in a huge search tree.

In the last approach, three way of expressing the tuples were explored: the first one is to explicitly express the duration of the events and deduce their position. This requires the less variables but makes it uneasy to express constraints. The second representation explicitly expresses durations and positions. It allows to express constraints way easier than the first method, but requires the most variables. Lastly, the third repression expresses explicitly the positions and deduces the durations. Its ease of constraint expression and its number of variables are situated between the two previous representations.

Each variation has been tested in a simple example wth Gecode, and the first variation was selected:

Variables

$drt_0...drt_{n_{max}-1}$ an array of n_{max} integer variables with domain $[-duration, duration]$.

Constraints

$C_1(drt_0...drt_{n_{max}-1}) = \sum_{i=0}^{n-1} |drt_i| = duration$

$C_2(drt_0...drt_{n_{max}-1}) = count(drt_i...drt_{n_{max}-1}, 0) \leq n_{max} - n_{min}$

This variation is the best compromise between execution time, number of symmetries and habitability.

Next, some example constraints where proposed for the third approach. Simple constraints can quickly be expressed and combined to create more sophisticated ones. A limitation was found that the CSP becomes quickly over constrained, and that it lacks flexibility for the predicted use of constraining an existing rhythm to obtain new ones, and then try to add more constraints to change the results. This limitation can be bypassed with a smart management of the tool using this model.

Chapter 5

GiL: Interface between Gecode and OpenMusic

The first step towards the implementation of a constraint-based tool for musical composition in OpenMusic is to include the constraint environment in OpenMusic. As OpenMusic can evaluate Lisp programs, the interface should mainly wrap the C++ Gecode library to Lisp. An interface called Gelisp was already made by M. Toro et al. [Toro et al., 2016] that also included a graphical interface to be used in OpenMusic. However, this interface has not been updated since 2008 while Gecode and OpenMusic both have evolved, as well as CFFI[1], the Lisp library used to link C++ and Lisp. Updating Gelisp would have been out of the scope of this master's thesis. The remaining of this chapter introduces GiL — *Gecode interface Lisp* — a small wrapper of Gecode to Lisp greatly inspired by Gelisp. The GiL source code can be found in appendix B.2 or on Github at `https://github.com/blapiere/GiL`.

5.1 C Wrapper

CFFI provides an interface between Lisp and C, but not C++. The first part of the implementation of GiL is then to provide a C external library that executes C++ Gecode functions. In Gecode, everything is modelled within a subclass of the *Space* class that represent the problem. Hence, a subclass *WSpace* (for "Wrap

[1] The *Common Foreign Function Interface*, see `https://common-lisp.net/project/cffi/`

Space") containing methods to add variables and post constraints and branching can be the general holder for the CSPs. Choice has been made to extend the subclass *IntMinimizeSpace* to provide support for optimization. When created, the space is cast as a void pointer that CFFI can understand. Each time a function in the external C library that uses a method of the space wrapper is called, the void pointer is then recast as the space wrapper to call the method, as in the code snippet below.

```
//Create a new space.
void* computation_space() {
        return (void*) new WSpace();
}

//Add a Gecode::IntVar to the space sp, ranging from min to max.
int add_intVar(void* sp, int min, int max) {
        return static_cast<WSpace*>(sp)->add_intVar(min, max);
}
```

Variables As variables are defined as instances of a variable class, they won't be able to be referenced the usual way in C. The *WSpace* stores them in a vector (each type of variable, namely integer, boolean, float and set, has its own vector), and defines methods to post constraints and branching via their indices in the vector instead of their reference. In the code snippet above, an integer variable is created. A Gecode *IntVar* is pushed in the integer variables vector of the wrapper and its index in the vector is returned, to be able to reference it in later functions. Currently, only integer and boolean variables are supported.

Constraints and branching Posting constraints and branching are wrapped similarly. For each constraint, one or more methods are created in the space wrapper to post it in the space.

Search engines Search engines have their own simple wrapper. They contain a reference to an actual instance Gecode search engine, and provides methods to call their *next* function, that returns a space holding a solution to the CSP.

The C wrapper is divided into two stages: the first stage is the definition of the *WSpace* that contains all the calls to Gecode functions inside of a particular

child class of *Space*, called the Space Wrapper. The second stage is the external C library that calls the methods of this wrapper, getting rid of the C++ features, called the Gecode Wrapper. The functions defined in the Gecode Wrapper will then be used by the Lisp wrapper.

5.2 Lisp Wrapper

The Lisp part of the interface wraps the C functions from the Gecode Wrapper external C library in two stages. The first stage is the definitions of the foreign functions using CFFI, and is a list of Lisp function, each calling a specific Gecode Wrapper function.

The second stage is the interface with the user. It takes advantage of CLOS generic methods system to provide a more readable and flexible library, closer to the Gecode original way of defining variables and posting constraints and branching. The next section shows an example of non trivial CSP modelled with GiL in OpenMusic.

5.3 Example program

This section shows an example CSP built with GiL in OpenMusic that solves the *all intervals* problems [Toro et al., 2015]: Given an integer n, the program has to find a sequence of n different pitches forming n distinct intervals. The all intervals problem can be modelled as follow:

Variables

$v_0 ... v_{n-1}$ an array of n integer variables with domain $[0, n[$.

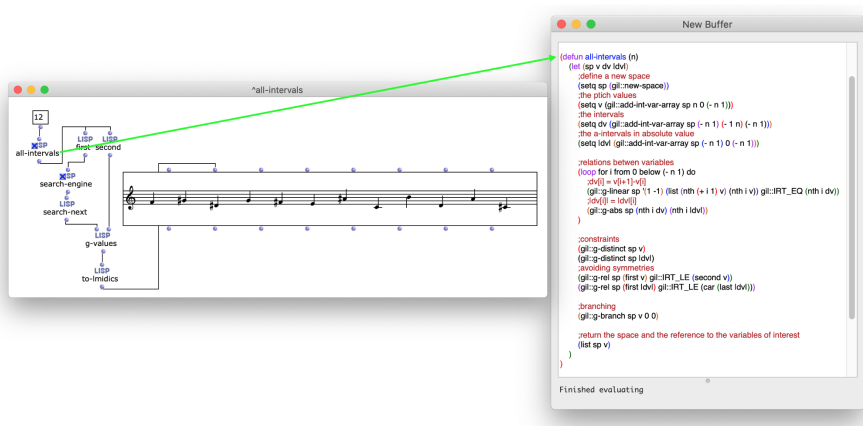
Constraints

$C_1(v_0 ... v_{n-1}) = \mathit{alldiff}(v_0 ... v_{n-1})$

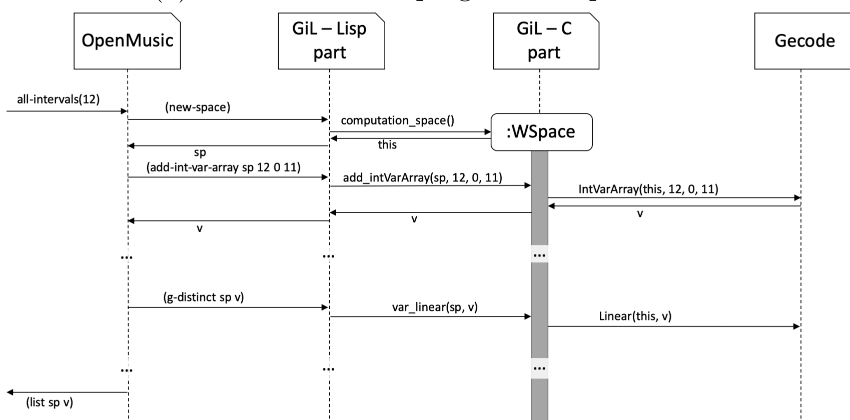
$C_2(v_0 ... v_{n-1}) = \mathit{alldiff}(|v_1 - v_0| ... |v_{n-1} - v_{n-2}|)$

$C_3(v_0, v_1) = v_0 < v_1$

$C_2(v_0, v_1, v_{n-2}, v_{n-1}) = |v_1 - v_0| < |v_{n-1} - v_{n-2}|$



(a) The all intervals program in OpenMusic



(b) Interactions between OpenMusic, GiL and Gecode. WSpace is a particular case since it extends the Gecode *IntMinimizeSpace* class.

Figure 5.1: All intervals program

Figure 5.1 shows (a) the program in OpenMusic and (b) a sequence diagram that represents some interaction between OpenMusic, GiL and Gecode. In the box *all-intervals* with parameter $n = 12$, a new Gecode search space is created and added some variables and constraints. The sequence diagram shows the interactions for the space creation, the declaration of a new variables array and posting a constraint on this array.

5.4 Conclusion

GiL provides an interface between Gecode and Lisp — and not directly OpenMusic, that can seemlessly include and evaluate Lisp expression and programs. It allows to use most of the integer variables features and some of the boolean variables features from Gecode. It is currently an incomplete interface, and work remains to be done in order to make it a complete interface. However, it wraps enough of Gecode features to be usable in the context of this master's thesis.

Chapter 6

Implementation

This chapter shows the implementation of the rhythm composition tool. It introduces Rhythm-Box, a constraint-based assistant for musical composition of rhythms in OpenMusic, developed as a result of this master's thesis. It shows the different components of the tool before diving into details. Finally, it suggests two way to extend Rhythm-box to a complete assistant for musical composition by adding the notion of pitch.

6.1 Rhythm-Box: a rhythm CSP in OpenMusic

Rhythm-Box is a tool created as a result of this master's thesis that uses a constraint-based approach to generate rhythms in OpenMusic. Its source code is available in appendix A.4 or on Github at `https://github.com/blapiere/Rhythm-Box`. It is composed of 3 core components:

6.1.1 *rhythm-box* box

The *rhythm-box* box is the heart of the tool. It is an OpenMusic method that takes two to five parameters:

- a list of constraints;
- a voice object[1];

[1]Voice objects in OpenMusic are score objects that contain a sequence of chords arranged

- an optional shortest event duration — if not set, the method takes the shortest event duration among the events of the input voice;

- an optional longest event duration — if not set, the method takes the longest event duration among the events of the input voice;

- a density handle that allows reducing the maximum number of notes in the output without increasing the shortest event duration. By default, this parameter is always at its maximum, which means it does not reduce the rhythmic density.

Given the above parameters, the *rhythm-box* creates a search space and defines the base CSP (as explained in section 6.2). The decision was taken that all problems solved by the tool would take their root in the input voice object, which is a melody already written by the composer. All the needed values (see section 4.1.4) are computed from the voice data unless the user explicitly specifies them. Attention has been paid that the user does not have to enter values that do not have a musical sense[2]. However, the user may use a dummy voice object to create rhythm from scratch: the only constraint is to set the signatures of the bars.

6.1.2 *constraint* boxes

Once the search space and the base CSP are set, the box adds the additional constraints. These constraints are defined by specific boxes, which are also methods that output a function and a list of arguments to be passed to this function, that is called inside the *rhythm-box*.

Currently, Rhythm-Box supports a set of constraints such as:

- the output rhythm must contain at least the same number of pulses as the input;

- the output rhythm must contain at most the same number of pulses as the input;

according to a rhythm tree.

[2]Yet the density handle is a float ranging between 0 to 1 that carry only indirect musical sense.

- the output rhythm must contain a pulse at a given position with duration equal/not equal/greater (or equal)/lower (or equal) to a given duration;
- the output rhythm must contain contain a pulse where there is one in the input;
- the output rhythm can not contain a pulse where there is no pulse in the input;
- the output rhythm must be the same as the input with a pattern inserted in place of the selected pulses.

This small list is not exhaustive, and the complete list of constraints, with their name to use them in OpenMusic, is available in appendix A.2. It is also possible to define new constraints, as shown in the tutorial in appendix A.3.

6.1.3 *search-next* box

After the additional constraints are posted, a search engine is created inside the *rhythm-box*. It is then output (together with the references to the duration variables and some values necessary to rebuild a rhythm tree) and passed as a parameter in the *search-next* box. This box produces a new voice object each time it is evaluated, with the rhythm tree corresponding to the next solution of the CSP.

An example of CSP using Rhythm-Box is shown on figure 6.1. It shows the following elements:

- base CSP: the *rhythm-box* is marked with a small blue cross, meaning that its evaluation is locked. If the evaluation of the *rhythm-box* is not locked when searching for solutions, a new CSP will be created each time the *search-next* box is evaluated, resulting in always finding the same first solution.
- constraint boxes: the constraints are OpenMusic methods that return an instance of the class *constraint*. They are put together in a list to and passed to the *rhythm-box*. The first constraint forces the output to use only the note durations from the input (the rest durations are free), and the second one constrains the output to have pulses where there are pulses in the input.

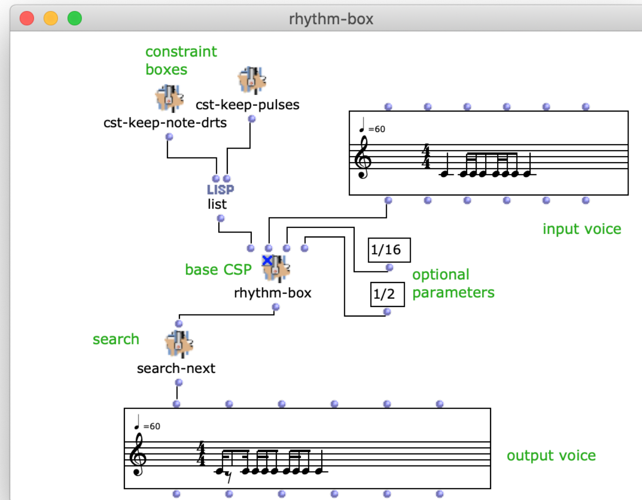


Figure 6.1: A CSP that finds rhythms that have at least the same pulses as the input and where the notes have duration from the input only.

- input voice: the voice object, i.e. the rhythm from which the CSP is built.

- optional parameters: here, the shortest authorized duration is set to $\frac{1}{16}$ and the longest authorized duration to $\frac{1}{2}$.

- search: the *search-next* is an OpenMusic method that takes the result of the *rhythm-box* (i.e. a list containing the search engine, the reference of the duration variables, the input and the shortest authorized duration). At each evaluation, it finds the next solution of the CSP if any, and instantiate a new voice object with the solution and the values from its arguments.

- output voice: this voice object contains the solution to CSP, recomputed as a voice from the input data.

A tutorial available in appendix A.1 provides more information on how to use Rhythm-Box.

6.2 Base CSP

The base CSP is modelled as described in section 4.1.4 using GiL. The program creates the values explained in that section from the input voice object data. Concerning the variant of the model, despite the apparent supremacy of the *explicit duration only* model, the fact that arguments array are not yet available in GiL makes it harder to express if the user wants to avoid the creation of unnecessary variables as much as possible. In this context, the model used is the *explicit position and duration* variant, that lets the user express constraints on positions more easily, and that do not provoke as many symmetries as the *explicit position only* variant.

However, once argument arrays are supported in GiL, changing the model to *explicit duration only* will be a small matter. Only a few lines have to be changed as below:

```
;replace the following line
(setq pos (add-int-var-array sp nmax 0 duration))
;by
(setq pos (int-var-args sp nmax 0 duration))
;and add the following constraints
(loop for i from 1 below nmax do ;pos[i] = pos[i-1]+drt[i-1]
    (g-sum sp (nth i pos) (first-n i drt)))
```

Where *(g-sum v vars)* constrains v to be equal to the sum of the $vars$, and *(first-n n l)* returns the n first elements of the list l. After this small operation, the model is changed, and the rest of the code continues to work properly without needing any change.

6.2.1 Adding constraints

The additional constraints the user may post are encapsulated in the *constraint* class. This class has two fields: a function and a list of arguments. After the base CSP is defined, the *rhythm-box* loops through the input constraints objects and call their function:

```
;loop through the constraints and post them
(loop for cst in csts do
    (post-constraint
```

```
        sp cst pos drt drt* N Dn s* l duration nmin nmax input)
)

;where the function post-constraint is the following:
(defun post-constraint
    (sp cst pos drt drt* N Dn s l duration nmin nmax input)
        (funcall (cstf cst)
            sp pos drt drt* N Dn s l duration nmin nmax input
            (args cst)
        )
)
```

A constraint function must take all the values of the CSP and the input voice as parameters in addition to its arguments list to allow a general constraints posting system, where the constraints can use the values computed in the *rhythm-box* to perform their own computations.

6.3 Examples programs with Rhythm-Box

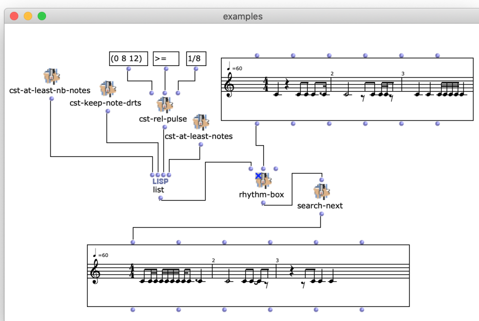
Figure 6.2 shows two use cases of the Rhythm-Box. The first case generates rhythms from an input voice object that already contains a pre-composed rhythm. The program finds rhythms that have at least the notes from the input, all the notes have duration that exist in the input and the first, ninth and twelfth notes last longer than an eighth note.

The second case only use a rhythm tree of three bars in $\frac{4}{4}$ with one note each, but does not take the content of the input bars into account. The only constraints are that there are at least 10 pulses and that pulses 0, 2, 4, 6, 8 and 10 are between a quarter note and an eighth note.

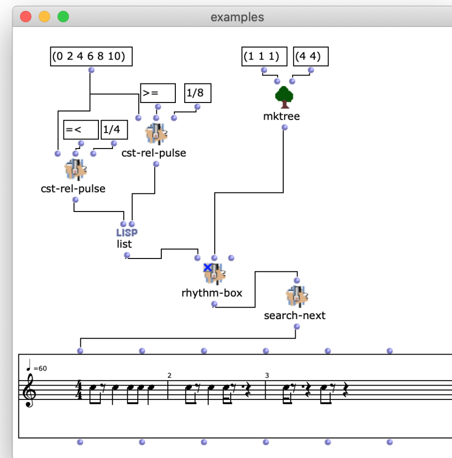
6.4 Summary

Rhythm-Box is a tool of constraint-based computer-assisted composition focused on rhythms. Rhythm-Box is divided into a few core components, namely the base CSP in the *rhythm-box* box, the additional constraints and the search box.

It is currently incomplete and presents some weaknesses, but it represents a good starting point to create a more complete, general and practical tool. Some of its



(a) Rhythms generated from a voice object



(b) Rhyhtms generated from scratch

Figure 6.2: Two ways of using Rhythm-Box: (a) shows a use case where an input rhythm is first composed and the program finds variations, and (b) shows a use case where only the number of bars and the time signatures are used to find new rhythms.

limitations are due to GiL limitations, which is yet an incomplete interface between OpenMusic and Gecode. This section introduced its functioning and its limitations, as well as ways to improve it.

Chapter 7

Conclusion

The main goals of this master's thesis were (1) create a tool that generates musical rhythm in OpenMusic by (2) modelling a general rhythm constraint-satisfaction problem that can be extended by adding new constraints with the aid of Gecode, and thus (3) create an interface between OpenMusic and Gecode.

Chapter 4 showed a step by step presentation on how the general rhythm CSP was modelled and gave some example of constraints. The model has the inconvenient of generating a lot of equivalent rhythms, but it can generate most possible rhythms.

In chapter 5, the GiL library was introduced, that creates an interface between OpenMusic and Gecode. GiL does not entirely wrap Gecode, but the main features required to reach the objectives are handled.

Finally, the Rhythm-Box — the tool itself — was presented in chapter 6. Despite some drawbacks both in usability and flexibility, it can efficiently generate rhythm from an input previously composed rhythm or from scratch. It features a small set of additional constraints destined to be extended.

There are still many improvements that can be done and researches that can be carried out. Concerning the model, the next step would be to include the notions of dynamics and pitch. Once these notions are modelled, multi-voice CSPs can be taken into consideration.

The interface between Gecode and OpenMusic can be completed by handling more types of variables and by covering the entirety of the constraints available in

Gecode. The branching strategies should be treated as well, including symmetry breaking. The expression system should also be included to allow expressive relation constraints, and the missing argument arrays can be added to optimize memory management. Finally, it could provide a visual interface in OpenMusic to ease its use for OpenMusic users.

Regarding the tool, its interface can be improved to provide more comfort to the user: the whole tool can be wrapped in a single box, and let the user start a new search from a part of a solution by selecting the events she wants to save. The symmetry problem should be handled, and the solutions that are equivalent to previous results should not be shown to the user. It could also be able to use variable total durations, number of bars, and time signature. Finally, a significant improvement would, of course, be the addition of pitch in the program.

The field is open to many more constraints, especially once the pitch is included: harmony, polyphony, the notion of what is possible with what musical instrument, and so on. Each musical genre has its own set of possible constraints.

Compared to everything that can be thought of, the tools and models presented seem quite limited. Yet, they represent a good starting point towards the creation of a completely general constraint-based assistant for musical composition.

7.1 Further Work

7.1.1 Completing GiL

Currently, GiL is an incomplete Gecode wrapper. While it is complete enough to be used in the context of this master's thesis, having it improved could make it a stand-alone tool and could improve the results of this work. It has the limitations described below.

Integer and boolean variables For now, only integer and boolean variables are supported, but they are not entirely supported. Some constraints remain to be wrapped, and the ones already wrapped could not work in specific cases. The

boolean variables are also poorly supported, and only a few constraints are available for them. Further work should include:

- Handling all the cases of the currently wrapped constraints;
- Complete the boolean variable support;
- Wrap all the remaining Gecode integer and boolean constraints.

Set and float variables No support is provided for float and set variables. Further work should include the wrapping of these kinds of variables similar to the wrapping of the integer and boolean variables, and provide the set of constraints and their different uses as Gecode does.

Branching strategies For now, while GiL provides branching functions with the parameters ready to define the strategy to use for branching, they do not influence the actual strategy used. The Space Wrapper actually always uses the same default strategy, which to set first the variable with the smallest domain, starting by its minimum value. Further work should include a wrapper for branching strategies. Besides, support for symmetry breaking can also be included.

Expressions Gecode expressive relation model presented in 3.2.3 is not available. A small trick to approach it is that GiL allows creating boolean variables from integers relations, e.g. $b = x > y$, and from there can post relation constraints on the boolean variables. However, complete support would allow far more readable and expressive models.

Argument variables arrays The argument arrays are currently not available in GiL. For the moment, it is required to create usual variables if the user wants to reference the result of an operation on variables. The implementation of argument arrays would improve the memory management of GiL. To provide support of argument arrays, either a vector of arguments should be added to the *WSpace* class, and specific use cases of all the constraints must be implemented, or a system should be found to store arguments in the same array as variables.

Visual interface Finally, a convenient feature of GiL would be to provide a visual interface in OpenMusic to ensure ease of use for non-programmers using OpenMusic.

Appendix B.1.1 shows a tutorial on how to improve the wrapper.

7.1.2 Improving Rhythm-Box

The Rhythm-Box is currently a limited tool. It can find many different rhythms, but some weaknesses are yet to be corrected.

Total duration and signatures

The first lack of flexibility is that the base CSP is such that the output sequence will always have the same duration as the input, and will always be sliced with the same bars. In the current state, the *rhythm-box* compute the total duration in ticks of the rhythm and uses it to define the maximum and minimum numbers of events n_{max} and n_{min} in the output. Using the input duration this way forbids to represent it with a variable, since n_{max} and n_{min} have to be known in order to define the events variables.

However, one can imagine a way of predicting and bounding the output duration with a similar method that was used to predict and bound the number of events of the output rhythm.

Branching

As GiL only provides a default branching strategy, the Rhythm-Box is forced to comply with this strategy, that makes the CSP look first for the shortest events. This strategy can make the tool inconvenient, especially when the user wants a rhythm with many different durations, because the solutions with only short durations will be given first. Even worse, the shortest events are the rests because they are defined as events with a negative duration. The CSP will then search for long rests before short notes.

When the branching definition is available in GiL, the Rhythm-Box should include a branching parameter. In order to make it musically sound, it should be expressed in terms of event preferences ("prefer short event durations") instead of using branching vocabulary.

Symmetries

As shown in the experiment (see 4.2), the number of symmetries, i.e. of unnecessary equivalent solutions, can be important in a lightly constrained CSP. Currently, the program shows all the solutions it finds even though an equivalent solution might have already been shown.

To eliminate this inconvenience, the first correction is to keep a solution in memory each time it is found. When the *search-next* box finds a new solution, it is compared to the previously found solutions. If it is equivalent to a previous solution, then it is discarded and the box searches the next solution until it finds a new distinct solution. This correction will prevent the user from having to look repeatedly through the same solutions but does not improve the search speed.

A second and more sophisticated correction would be to perform a branch-and-bound search that checks the equivalence during the search, and once a solution is found. It would let the program prune entire parts of the search tree and improve the search speed, and at the same time, improve the user comfort.

Interface and features

Currently, the user interface is not so practical. For example, the need to lock the evaluation of the *rhythm-box* can lead to frustration in case the user forgets it. Furthermore, all this box tree should be encapsulated in a single box to make it more appealing and understandable.

An important improvement that can be added to this tool is to let the user select a part of the solution and begin a new search, free of the previous constraints, that can not influence the selected part. This feature would allow the user to save parts

of the rhythm she likes and thinks should be further modified. In the same time, it would allow to not have to remember which constraint had which effect on the result if the user wants to modify only parts the result.

Events are limited

Events defined by a position and a duration are a poor representation of a musical element. The model should be extended to include the notion of magnitude, or velocity in MIDI terms, that represent the strength the note is played with. And of course, events should eventually include the notion of pitch in order to provide a more complete tool. The next subsection discusses how the pitch could be added to go from a Rhythm-Box to a Music-box.

7.1.3 Discussion: including pitch

A complete constraint-based assistant for musical composition should obviously include the notion of pitch to create notes, chords and melodies. This subsection suggests two different approaches to include pitch. The first approach is to create a second CSP dedicated to pitch only, that takes the output of the rhythm CSP as a parameter. The second is to extend the model of events tuples from $event(pos, drt)$ to $event(pos, drt, pitches)$, where $pitches$ is a list of pitch variables.

In both approaches, the number of variables used to define each chord must be predicted and bounded. A pitch variable would then be an integer variable with a domain equal to the desired register expressed in midicents and 0. A note in the chord with a pitch of 0 is a non-existing note. To limit the number of symmetries, a pitch in the chord that is not equal to 0 should be strictly greater than the previous pitch.

First approach: a two-stage CSP

In this approach, the number of events that are not rests is already known. For each of these events, n pitch variables are created. n should be a typical number of notes in a chord but should be modifiable by the user. Each of the pitch variables is assigned to a value during the search according to the constraints.

The advantage of this approach is that the number of symmetries is reduced since there is no pitch variable assigned to rests. It would also allow different strategies for the model of the rhythm: everything that is needed is a rhythm tree in the input. However, it would be preferable that GiL provides a visual interface to let the user define its own specific CSPs, for example. The major inconvenient is that the tool is forced to have a two-step functioning: first finding a rhythm and then looking for pitches. In practice, according to O. Sandred, the composition process usually does not separate rhythm from pitch [Örjan Sandred, 2009].

Second approach: extend the model

The events are currently a pair of two variables: the position and the duration of the event. The idea would be to extend this representation to include pitch, by adding a list of variables that represents a chord. The events would then become: $event(pos, drt, pitch$.

This approach uses only one CSP for both rhythm and pitch. Starting from the rhythm CSP, we extend it with n pitch variables for each event, resulting in $n_{max} * n$ additional variables. The search can then be performed on rhythm and pitch simultaneously.

This approach can generate a huge number of unnecessary variables. A high n_{max} does not necessary means a great number of chords in the output: there can be more rests than notes, and there can be only a few long events. In order not to waste time and memory, the user should efficiently define the base CSP.

Bibliography

[Agon, 1998] Agon, C. (1998). *OpenMusic: Un langage visuel pour la composition musicale assistée par ordinateur.* PhD thesis, Université Pierre et Marie Curie / Paris 6, Paris.

[Amoric, 2006] Amoric, M. (2006). Writing a Homage to Mersenne: Tombeau de Marin Mersenne for Theorbo and General MIDI. In Delatour, editor, *The OM composer's book 1.* Ircam, Centre Pompidou.

[Anders, 2007] Anders, T. (2007). *Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System.* PhD thesis, School of Music & Sonic Arts, Queen's University Belfast.

[Assayag, 1998] Assayag, G. (1998). Computer assisted composition today. In *1st symposium on music and computers, Corfu.*

[Barták, 1999] Barták, R. (1999). Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, volume 4, pages 555–564. MatFyzPress Prague.

[Beck, 2013] Beck, J. H. (2013). *Encyclopedia of percussion.* Routledge.

[Bonnet and Rueda, 1999] Bonnet, A. and Rueda, C. (1999). OpenMusic, Situation. *IRCAM, Paris.*

[Frühwirth, Thom and Abdennadher, Slim, 2003] Frühwirth, Thom and Abdennadher, Slim (2003). *Essentials of constraint programming.* Springer Science & Business Media.

[Laurson, 1996] Laurson, M. (1996). PatchWork, PWConstraints. *IRCAM, Paris, first english edition.*

[Rossi et al., 2006] Rossi, F., Van Beek, P., and Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

[Sandred, 2010] Sandred, Ö. (2010). PWMC, a constraint-solving system for generating music scores. *Computer Music Journal*, 34(2):8–24.

[Schulte et al., 2019] Schulte, C., Tack, G., and Lagerkvist, M. (2019). Modeling and programming with Gecode.

[Toro et al., 2015] Toro, M., Rueda, C., Agón, C., and Assayag, G. (2015). Gelisp: A library to represent musical csps and search strategies. *arXiv preprint arXiv:1510.02828*.

[Toro et al., 2016] Toro, M., Rueda, C., Agon, C., and Assayag, G. (2016). GELISP : A Framework to represent musical constraint satisfaction problems and search strategies. *Journal of Theoretical and Applied Information Technology*, 86(2):327–331.

[Truchet, 2004] Truchet, C. (2004). *Contraintes, recherche locale et composition assistée par ordinateur*. PhD thesis, Université Pierre et Marie Curie / Paris 6, Paris.

[Örjan Sandred, 2009] Örjan Sandred (2009). Approaches to Using Rules as a Composition Method. *Contemporary Music Review*, 28(2):149–165.

Appendix A

Rhythm-Box

This appendix brings more explanations on how to use Rhythm-Box. First, it shows a small tutorial to build a simple rhythm CSP. Then, it proposes the catalogue of all the available Rhythm-Box constraints. Thirdly, it shows how to create user-defined constraints and add them to the base CSP. Finally, the source code of Rhythm is presented and described.

A.1 Tutorial: searching rhythms

This section shows how to use the Rhythm-Box tool to generate rhythms from an input rhythm. The first steps are to install Gecode[1] and OpenMusic[2] and then download and install the GiL and Rhythm-Box OpenMusic libraries from `https://github.com/blapiere/GiL` and `https://github.com/blapiere/Rhythm-Box`[3]. Pay attention, currently only computers that can support dylib files can use GiL.

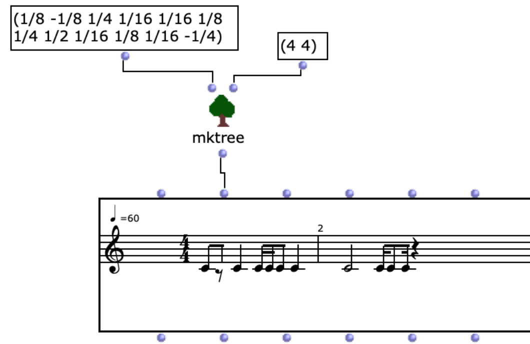
Once this is done, open OpenMusic and create a new patch. Create a voice object (eihter double click on the background of the patch and write "voice" or select it from the menu “Classes > Score > VOICE”) and add a rhythm tree. If you are not familiar with the rhythm trees, you can use the the OpenMusic function

[1]`https://www.gecode.org/download.html`

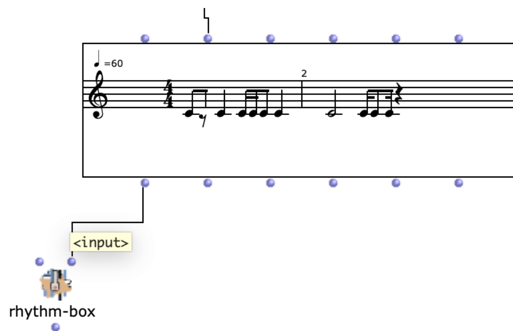
[2]`https://openmusic-project.github.io/openmusic/`

[3]To add libraries to OpenMusic: `http://support.ircam.fr/docs/om/om6-manual/co/UserLibraries.html`

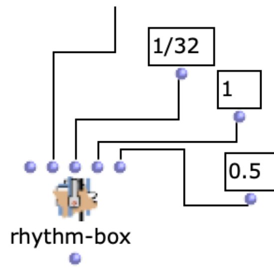
"MKTREE" to generate one from a list of ratios and a list of time signatures.



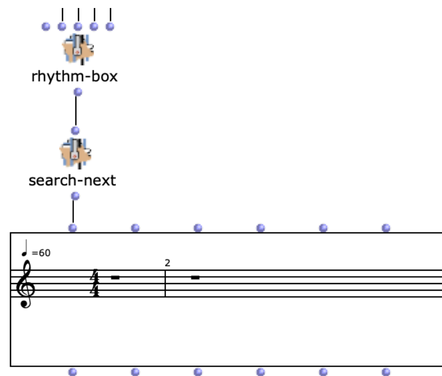
The next step is to create the rhythm-box. Double-click on the patch background and write "rhythm-box". The method will appear, with initially two inlets: the constraints and the input. Link the *self* or *tree* outlet of the voice to the *input* inlet of the rhythm-box.



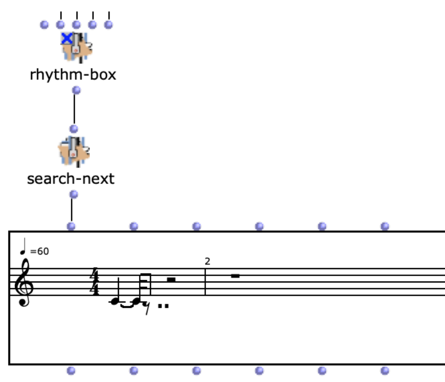
If you evaluate the rhythm-box now, a base CSP is created with data from the input. But let's modify this data. In our new rhythm, we want the minimum duration of an event to be 1/32. The default minimum duration is the one of the the input (here 1/16), so we have to specify it. Reveal the optional *s* inlet of the rhythm-box by selecting it and pressing ALT+→, and add a primitive 1/32 box. Finally, as the 1/32 shortest duration will allow the space to generate up to 64 events and since we don't want that many events, we will set the *hd* inlet, i.e. the density handle, to 0,5, which will reduce the the maximum number of events to its half.



In that state, the rhythm-box can already generate rhythms. Add a search-next box connect the output of the rhythm-box to its input. Then, add a new voice object and connect the output of the search-next box to its *self* inlet. You can evaluate the new voice (press V while selecting it) and generate rhythms.



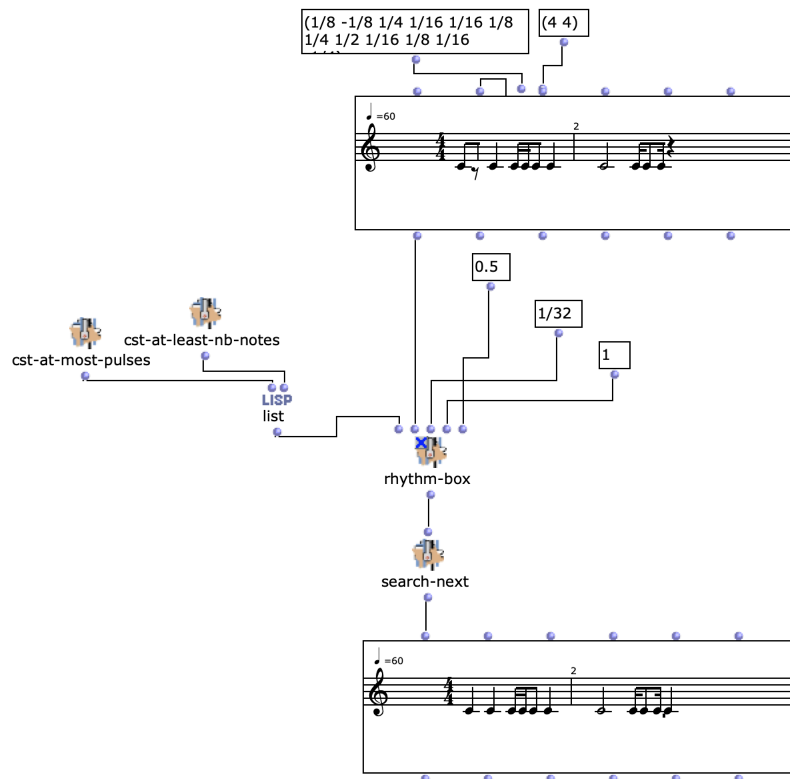
You will notice that the search always returns the same score without any note. The reason is simple: in this configuration, a silent rhythm is always the first generated solution. And each time the output score is evaluated, the evaluation process evaluates the rhythm-box as well, creating a new identical CSP with the same first solution at each evaluation. To solve this inconvenience, the evaluation of the rhythm-box must be blocked once it is first evaluated. Select the rhythm-box and press B. A small blue cross appears in its top-left corner, meaning its evaluation is blocked. You can now search for rhythms by evaluating the output voice.



The results are not so interesting for the moment. It is time to add constraints to our base CSP. Let's say we want a rhythm with the at least the same number of notes as the input, but with at most the same pulses (i.e. if there is a pulse in the output, there must be a pulse at this position in the input).

Create a Lisp list, select it and press ALT+→ twice to reveal two inlet. Then, create a "cst-at-least-nb-notes" and a "cst-at-most-pulses" boxes, and link their output to the list inlets. You can now link the list outlet to the *csts* inlet of the rhythm-box.

To evaluate the addition of the new constraints, you have to first unlock the rhythm-box evaluation, then evaluate it, and then lock it again. You can now evaluate the output score and see the results.



This tutorial is over. A catalogue of available constraints can be found in the next section. Keep in mind the locking/unlocking of the rhythm-box when changing constraints or input.

A.2 Constraints catalogue

This section presents the exhaustive list of currently available constraints.

- **cst-rel-pulse** The specified positions must contain a note of duration with the specified relation to the specified value (e.g. equal to 1/4 or greater than 1/16).

- **cst-keep-pulses** The output rhythm must include at least the pulses contained in the input rhythm.

- **cst-at-most-pulses** If there is a pulse in the output, there must be a pulse at this position in the input.
- **cst-at-most-nb-notes** The output rhythm can not contain more notes than the input.
- **cst-at-least-nb-notes** The output rhythm can not contain less notes than the input.
- **cst-keep-note-drts** The output rhythm can not have events of different duration than the durations present in the input.
- **cst-keep-nb-pulses** The number of pulses in the output is exactly the same as the number of pulses in the input.
- **cst-at-least-notes** All the notes from the input are in the output (their positions can change, but not their durations).
- **cst-insert-pattern** The output rhythm is the same as the input, but with the specified pattern inserted in place of the specified pulses. The length of the inserted pattern is dependant of the duration of the pulse it replaces.

A.3 Tutorial: create a new constraint

This section shows how to create new constraints for the Rhythm-Box. The example will be constraining the output rhythm to have all its notes duration greater than the previous note duration in the rhythm, and to have a note at position 0 with duration equal to an input duration.

Open the Lisp Editor (menu "Windows > Lisp Editor"). The first step is to express our new rule in terms of constraint. In our case, it would be:

$\forall i \in [1, n_{max}[:\ drt_i > 0 \Leftrightarrow drt_i > drt_{i-1}$

Create a Lisp function that expresses this constraint using GiL. All the constraint functions must use the same arguments list containing all the input parameters of the CSP plus a list containing their own arguments. To ease the writing, begin by defining the function with only the needed parameters:

```
(in-package :om)

;<value> is the duration of the first event (e.g. 1/4)
;<sp> is search space of the problem
;<drt> is the list of references to the duration variables
;<s> is the shortest note duration (required to convert a ratio into
; a tick duration)
;<nmax> is the maximum number of events in the output
(defun my-cst (value sp drt s nmax)
    (gil::rel (first drt) gil::IRT_EQ (* s value))
    (loop for i from 1 below nmax do
        ;first create the boolean expressions.
        (let ((b1 (gil::add-bool-var-expr sp (nth i drt) gil::IRT_GR 0))
              ;if an event duration is greater than 0 (i.e. it exists and is not a rest) ...

              (b2 ((gil::add-bool-var-expr sp (nth i drt) gil::IRT_GR (nth (- i 1) drt)))))
              ;... then the preceding event is of shorter duration.

            ;and then express the equivalence relation between them.
            (gil::g-op sp b1 gil::BOT_EQV b2))
    )
)
```

Now, you can create a function that calls that function and that takes all the needed parameters. Note that our specific parameter that defines the duration of the first note (“value” in the code above) is hidden in the “args-list” parameter and axtracted when the my-cst function is called.

```
(defun post-my-cst (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (my-cst (first args-list) sp drt nmax)
)
```

The last step is to create an OpenMusic method box that takes a ratio in input and returns a constraint. All constraint in Rhythm-Box are instances of the class *constraint*, that hold a constraint function and its list of specific parameters.

```
(defmethod! cst-my-cst (value)
    :initvals '(1/4)
    :indoc '("a␣ratio")
    :icon 262
    :icon 262
    :doc "
This␣is␣my␣first␣constraint.
```

```
"
    (make-instance 'constraint
        :cstf #'post-my-cst
        :args (list value))
)
```

The new constraint is ready to be used in an OpenMusic program.

A.4 Source code

The source code of Rhythm-Box is spread among three files:

- **rhythm-csp.lisp** defines the base CSP and the searcg support in the methods rhythm-box and search-next.
- **rhythm-csts.lisp** is the source code of all the constraint boxes.
- **rhythm-utils.lisp** is a small library of useful methods to handle rhythms and rhtythms trees.

rhythm-csp.lisp

The definition of the base CSP and some additional methods.

```
(in-package :om)

(defclass constraint ()
    ((cst-function :initform nil :initarg :cstf :accessor cstf)
    ;a constraint function that always have a space as 1st arg and a list as 2nd arg
    ;should this function have return values, they HAVE TO under the form of a list
    ;of var-ref objects.

     (args :initform nil :initarg :args :accessor args))
    ;the list args to be the 2nd argument of the cst-function
)

(defun post-constraint (sp cst pos drt drt* N Dn s l duration nmin nmax input)
    "Call␣the␣costraint␣function␣with␣the␣rhythm-box␣data␣and␣the␣constraint␣arglist."
    (funcall (cstf cst) sp pos drt drt* N Dn s l duration nmin nmax input (args cst))
)

; <csts> is a list of <constraint> object
; <input> is either a voice object or a rhythm tree
; <s> is a fraction - the shortest authorized event duration
; <l> is a fraction - the longest authorized event duration
; <hd> is a float between 0 and 1 that indicates the maximum rhythmic density
(defmethod! rhythm-box (csts input &optional (s nil) (l nil) (hd 1.0))
    :initvals (list nil (make-instance 'voice) nil nil 1.0)
    :indoc '("a␣list␣of␣constraints" "a␣voice␣or␣a␣rhythm␣tree" "the␣shortest␣desired␣note␣duration␣e.g.␣
        1/16"
            "the␣longest␣desired␣note␣duration␣e.g.␣1/2" "a␣float,␣the␣ratio␣of␣maximum␣density")
    :icon 921
```

```
    :doc "
Create␣a␣search␣space␣for␣the␣musical␣CSP␣and␣post␣general␣constraints␣that␣define␣general␣rhythm␣CSP.
"
    (let ((sp (gil::new-space))
          (N (get-N input))
          (Dn (get-D input))
          (s* (or (convert s) (shortest input)))
          duration nmin nmax csp pos drt drt* se)

        ;1: create the variables (depending on the input and options)
        (setq duration (* s* (/ N Dn)))
        (setq nmin (ceiling (or (or/ N (or* Dn l)) 1))) ;nmin = N/Dl or 1
        (setq nmax (round (* hd duration)))

        (setq drt (gil::add-int-var-array sp nmax (- duration) (or (or* l s*) duration)))
        (setq drt* (gil::add-int-var-array sp nmax 0 duration))
        (setq pos (gil::add-int-var-array sp nmax 0 duration))

        ;2: post the constraints
        ;general constraints

        ;2.1: drt*[i] = |drt[i]|
        (loop for i from 0 below nmax do
            (gil::g-abs sp (nth i drt) (nth i drt*)))

        ;2.2: sum(drt*) = duration
        (gil::g-linear sp (create-coeffs nmax 1) drt* gil::IRT_EQ duration)

        ;2.3: count(drt, 0) <= nmax-nmin
        (gil::g-count sp drt 0 gil::IRT_LQ (- nmax nmin))

        ;2.4: pos[i] = pos[i-1]+drt[i-1]
        (loop for i from 1 below nmax do
            (gil::g-linear sp '(1 1) (list (nth (- i 1) pos) (nth (- i 1) drt*)) gil::IRT_EQ (nth i pos)))
        ;2.4': pos[0] = 0
        (gil::g-rel sp (nth 0 pos) gil::IRT_EQ 0)

        ;pos[i] = duration => drt[i] = 0
        (loop for i from 0 below nmax do
            (let ((b1 (gil::add-bool-var-expr sp (nth i pos) gil::IRT_EQ duration))
                  (b2 (gil::add-bool-var-expr sp (nth i drt) gil::IRT_EQ 0)))
                (gil::g-op sp b1 gil::BOT_IMP b2 1)))


        ;additional constraints
        (loop for cst in csts do
            (post-constraint sp cst pos drt drt* N Dn s* l duration nmin nmax input)
        )

        ;branching
        (gil::g-branch sp pos 0 0)
        (gil::g-branch sp drt 0 0)

        ;search engine
        (setq se (gil::search-engine sp nil))

        ;return
        (list se drt input s*)
))

(defmethod! search-next (l)
    :initvals (list nil)
    :indoc '("a␣rhythm-space")
    :icon 330
    :doc "
Get␣the␣next␣solution␣for␣the␣rhythm␣csp␣described␣in␣the␣input␣rhythm-space.
"
```

```
    (let ((se (first l))
         (drt* (second l))
         (input (third l))
         (s* (nth 3 l))
         sol drt rtree)

        ;Get the values of the solution and build the tree
        (setq sol (gil::search-next se))
        (if (null sol) (error "No␣solution␣or␣no␣more␣solution."))
        (setq drt (gil::g-values sol drt*))
        (if (typep input 'voice)
            (setq rtree (build-rtree (tree input) drt s*))
            (setq rtree (build-rtree input drt s*)))

        ;return a voice object based on the input
        (if (typep input 'voice)
            (make-instance 'voice
                :tree rtree
                :chords (chords input)
                :tempo (tempo input)
                :legato (legato input)
                :ties (ties input))
            (make-instance 'voice
                :tree rtree
                :chords '((7200)))
    )
))
```

rhythm-csts.lisp

The implementation of the constraint boxes.

```
(in-package :om)

;;;;;;;;;;;;;;;;;;;;;;
; SAMPLE CONSTRAINTS ;
;;;;;;;;;;;;;;;;;;;;;;

; Constraints are defined as instance of the class <constraint>, which is a container
; for a function that post the gecode constraints and its parameters.
;
; To create a constraint, one must create an OM method (i.e. via defmethod!) that returns
; the constraint instance.
; For readability reasons, the following constraints always return a function called "post-<cst>"
; that calls the <cst> by rearranging the args.


; Constraint REL-PULSE
; The specified positions must contain a note of duration with the specified relation
; to the specified value (e.g. equal to 1/4 or greater than 1/16).

(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)
    )
)
```

```
(defun rel-pulse (lp rel value sp pos drt Dn s nmax)
    (let ((dp (* value s))
          pulses)

        (setq pulses (mapcar #'(lambda (x) (* x dp)) lp))
        (loop for x in pulses do
            (gil::g-count sp pos x gil::IRT_GQ 1)
            (loop for i from 0 below nmax do
                (let  ;b1: if the pos is equal to x...
                     ((b1 (gil::add-bool-var-expr sp (nth i pos) gil::IRT_EQ x))
                      ;b2: ... then the drt is related to dp
                      (b2 (gil::add-bool-var-expr sp (nth i drt) (rel-to-gil rel) dp))
                      ;b3: ... and the drt is greater than 0 (i.e. is a pulse)
                      (b3 (gil::add-bool-var-expr sp (nth i drt) gil::IRT_GR 0)))
                (gil::g-op sp b1 gil::BOT_IMP b2 1)
                (gil::g-op sp b1 gil::BOT_IMP b3 1))
        ))
    )
)

(defun post-rel-pulse (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (rel-pulse (car args-list) (second args-list) (third args-list) sp pos drt Dn s nmax)
)

(defmethod! cst-rel-pulse (pos &optional (rel '=) (drt 1/4))
    :initvals '((0) = 1/4)
    :indoc '("a␣list␣of␣positions" "a␣relation␣operator␣(e.g␣=)" "a␣note␣duration␣(e.g.␣1/4)")
    :icon 262
    :doc "
Constrains␣the␣output␣rhythm␣sequence␣to␣include␣a␣note␣of␣duration␣related␣to␣the␣drt.

-␣The␣relation␣is␣defined␣by␣the␣relation␣operator␣argument␣and␣is␣=,␣=/=,␣<,␣=<,␣>␣or␣>=␣.
-␣The␣drt␣is␣fraction␣(1/4␣corresponds␣to␣a␣quarter␣note).
"
    (make-instance 'constraint
        :cstf #'post-rel-pulse
        :args (list pos rel drt))
)

; Constraint KEEP-PULSE
; The output rhythm must include at least the pulses contained in the input rhythm.

(defun keep-pulses (sp pos drt s nmax input)
    (let ((pulses (only-pulse-ticks (tree2posticks input s))))
        (loop for p in pulses do
            (gil::g-count sp pos p gil::IRT_GQ 1)
            (loop for i from 0 below nmax do
                (let ((b1 (gil::add-bool-var-expr sp (nth i pos) gil::IRT_EQ p))
                      (b2 (gil::add-bool-var-expr sp (nth i drt) gil::IRT_GR 0)))
                    (gil::g-op sp b1 gil::BOT_IMP b2 1))
            ))
))

(defun post-keep-pulses (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (keep-pulses sp pos drt s nmax input)
)

(defmethod! cst-keep-pulses ()
    :icon 262
    :doc "
Constrains␣the␣output␣rhythm␣sequence␣to␣include␣a␣pulse␣where␣the␣input␣tree␣or␣voice␣does.
"
    (make-instance 'constraint
        :cstf #'post-keep-pulses
        :args nil)
)
```

```
; Constraint AT-MOST-PULSES
; The output rhythm contains only pulses from the input
(defun at-most-pulses (sp pos s duration nmax input)
    (let ((pulses (only-pulse-ticks (tree2posticks input s))))
      (loop for i from 0 below nmax do
            (gil::g-dom sp (nth i pos) (append pulses (list duration)))
       )
))

(defun post-at-most-pulses (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (at-most-pulses sp pos s duration nmax input)
)

(defmethod! cst-at-most-pulses ()
    :icon 262
    :doc "
Constrains␣the␣output␣rhythm␣sequence␣to␣not␣include␣pulse␣where␣there␣is␣no␣pulse
in␣the␣input.
"
    (make-instance 'constraint
        :cstf #'post-at-most-pulses
        :args nil)
)

; Constraint AT-MOST-NB-NOTES
; The output rhythm has at most the same number of notes as the input.

(defun at-most-nb-notes (sp drt nmax input)
    (let ((drt>0 (gil::add-int-var-array sp nmax 0 1)))
        (loop for i from 0 below nmax do
            (let ((b1 (gil::add-bool-var-expr sp (nth i drt) gil::IRT_GR 0))
                  (b2 (gil::add-bool-var-expr sp (nth i drt>0) gil::IRT_EQ 1)))
                (gil::g-op sp b1 gil::BOT_IMP b2 1)))
    (gil::g-count sp drt>0 1 gil::IRT_LQ (n-pulses input)))
)

(defun post-at-most-nb-notes (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (at-most-nb-notes sp drt nmax input)
)

(defmethod! cst-at-most-nb-notes ()
    :icon 262
    :doc "
Constrains␣the␣output␣rhythm␣sequence␣to␣have␣at␣most␣the␣same␣number␣of␣pulses␣as␣the␣input.
"
    (make-instance 'constraint
        :cstf #'post-at-most-nb-notes
        :args nil)
)

; Constraint KEEP-NB-PULSES
; The output rhythm must have the same number of pulses as the input

(defun keep-nb-pulses (sp drt nmax input)
    (let ((note? (gil::add-int-var-array sp nmax 0 1)))
        (loop for i below nmax do
            (let ((b1 (gil::add-bool-var-expr sp (nth i drt) gil::IRT_GR 0))
                  (b2 (gil::add-bool-var-expr sp (nth i note?) gil::IRT_EQ 1)))
                (gil::g-op sp b1 gil::BOT_EQV b2 1)))
        (gil::g-count sp note? 1 gil::IRT_EQ (n-pulses input))
    )
)

(defun post-keep-nb-pulses (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (keep-nb-pulses sp drt nmax input)
)
```

```
(defmethod! cst-keep-nb-pulses ()
    :icon 262
    :doc "
Constrains␣the␣output␣rhythm␣sequence␣to␣have␣the␣same␣number␣of␣pulses␣as␣the␣input.
"
    (make-instance 'constraint
        :cstf #'post-keep-nb-pulses
        :args nil)
)

; Constraint KEEP-NOTE-DRTS
; The notes in the output sequence have only durations taken from the intput.

(defmethod get-ratios ((input list)) (tree2ratio input))
(defmethod get-ratios ((input voice)) (get-ratios (tree input)))

(defun keep-note-drts (sp drt s duration nmax input)
    (let ((neg (loop for d from (- duration) to 0 collect d))
          (pos (remove-if-not #'(lambda (x) (> x 0)) (mapcar #'(lambda (x) (* s x)) (get-ratios input)))))

        (loop for i from 0 below nmax do
            (gil::g-dom sp (nth i drt) (append neg pos)))
    )
)

(defun post-keep-note-drts (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (keep-note-drts sp drt s duration nmax input)
)

(defmethod! cst-keep-note-drts ()
    :icon 262
    :doc "
Constrains␣the␣notes␣in␣the␣output␣rhythm␣sequence␣to␣have␣only␣duration␣from␣the␣input.
"
    (make-instance 'constraint
        :cstf #'post-keep-note-drts
        :args nil)
)

; Constraint AT-LEAST-NOTES
; The output sequence contains at least the notes (i.e. events with positive duration) of the input. Notes
; in the input occur exactly once in the output, but other notes can exist in the output.

(defun distinct (l &optional (acc nil))
    (cond
        ((null l) acc)
        ((member (car l) acc) (distinct (cdr l) acc))
        (t (distinct (cdr l) (append acc (list (car l)))))
    )
)

(defun group-drts (input)
    "Create␣a␣list␣of␣pairs␣(x␣drt)␣where␣x␣is␣the␣number␣of␣occurrences␣of␣drt␣in␣the␣input"
    (let ((drt (remove-if-not #'(lambda (x) (> x 0)) (get-ratios input))))
        (mapcar #'(lambda (x) (list x (count x drt))) (distinct drt)))
)

(defun at-least-notes (sp drt s nmax input)
    (let ((ticks (mapcar #'(lambda (x) (list (* s (first x)) (second x))) (group-drts input))))

    (loop for p in ticks do
        (gil::g-count sp drt (first p) gil::IRT_GQ (second p))))
)

(defun post-at-least-notes (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (at-least-notes sp drt s nmax input)
)
```

```
(defmethod! cst-at-least-notes ()
    :icon 262
    :doc "
Constrains␣the␣output␣sequence␣to␣contain␣at␣least␣the␣note␣durations␣from␣the␣input.␣There␣can␣also␣be␣
     other␣notes␣with
other␣durations.
"
    (make-instance 'constraint
        :cstf #'post-at-least-notes
        :args nil)
)

; Constraint INSERT-PATTERN
; Constrains the output rhythm to include a user-defined pattern at each select pulse of the input.
; The pattern is adapted in length according to the duration of the note at the pulse.

(defun insert-pattern (p* l sp pos drt s nmax input) ;TODO: also insert input RESTS !!!!
    "p␣is␣the␣list␣of␣ratios␣defining␣the␣pattern␣to␣insert
␣␣␣␣␣l␣is␣a␣list␣bools:␣l[i]␣<=>␣p␣is␣inserted␣at␣pulse␣i"
    (let ((ps (only-pulse-ticks (tree2posticks input s))) ;positions of pulses in ticks
          (ds (mapcar #'(lambda (x) (* s x)) (remove-if-not #'(lambda (x) (> x 0)) (get-ratios input))))
               ;durations of pulses in ticks
          (p (mapcar #'(lambda (x) (* s x)) p*)) ;pattern in ticks
          (np (length p*))
          dp minp)
        (setq dp (apply '+ (mapcar #'abs p)))
        (setq minp (apply 'min (mapcar #'abs p)))
        (setq ns (length ps))

        (loop for i from 0 below (length ps) do
            (gil::g-count sp pos (nth i ps) gil::IRT_GQ 1)
            (let ((r (/ (nth i ds) dp)))
                (if (and (nth i l) (>= (* minp r) 1))
                    (loop for j from 0 below (- nmax np) do
                        (let ((b1 (gil::add-bool-var-expr sp (nth j pos) gil::IRT_EQ (nth i ps))))
                            (loop for k from 0 below np do
                                (let ((b2 (gil::add-bool-var-expr sp (nth (+ j k) drt) gil::IRT_EQ (floor (*
                                     r (nth k p))))))
                                    (gil::g-op sp b1 gil::BOT_IMP b2 1)))))
                  ;else
                 (loop for j from 0 below (- nmax dp) do
                        (let ((b1 (gil::add-bool-var-expr sp (nth j pos) gil::IRT_EQ (nth i ps)))
                                (b2 (gil::add-bool-var-expr sp (nth j drt) gil::IRT_EQ (nth i ds))))
                             (gil::g-op sp b1 gil::BOT_IMP b2 1)))))))
)

(defun xor (a b) (or (and a (not b)) (and (not a) b)))

(defmethod pulses2bools ((l null) p)
    (loop for i from 0 below (length p) collect t))

(defmethod pulses2bools ((l list) p)
    (let ((l* (mapcar 'abs l)))
    (loop for i from 0 below (length p) collect (xor (member (+ i 1) l*) (< (first l) 0)))))

(defmethod pulses2bools ((l fixnum) p) (pulses2bools (list l) p))

(defun post-insert-pattern (sp pos drt drt* N Dn s l duration nmin nmax input args-list)
    (insert-pattern (first args-list) (pulses2bools (second args-list) (get-pulse-places input))
                    sp pos drt s nmax input)
)

(defmethod! cst-insert-pattern ((pattern list) &optional (pulses nil))
    :initvals (list '(1/8 1/8) nil)
    :indoc '("a␣pattern␣described␣by␣a␣list␣of␣ratios␣(e.g.␣(1/16␣1/8␣1/16)"
             "a␣list␣of␣integers,␣the␣indices␣of␣the␣pulses")
```

```
    :icon 262
    :doc "
Constrains␣the␣output␣rhythm␣sequence␣to␣include␣the␣pattern␣<pattern>␣at␣each␣pulse␣specified␣in␣<pulses>.␣
     The
duration␣of␣the␣pattern␣in␣dependant␣of␣the␣duration␣of␣the␣note␣at␣the␣pulses.␣For␣example,␣inserting
p␣=␣(1/16␣1/8␣1/16)␣(i.e.␣the␣duration␣of␣p␣is␣1/4)␣at␣a␣pulse␣where␣the␣note␣lasts␣1/4,␣results␣in␣replacing
1/4␣by␣(1/16␣1/8␣1/16).␣If␣note␣lasted␣1/8,␣it␣would␣have␣been␣replaced␣by␣(1/32␣1/16␣1/32),␣and␣by␣(1/8␣1/4␣
     1/8)
if␣it␣lasted␣1/2.␣If␣the␣result␣would␣include␣a␣note␣shorter␣than␣the␣shortest␣duration␣authorized␣in␣the␣
     space,
then␣the␣pattern␣is␣not␣inserted␣at␣this␣pulse.

-␣<pattern>␣must␣be␣a␣non␣null␣list␣expressed␣in␣ratios.
-␣<pulses>␣must␣be␣a␣list␣of␣integers,␣indices␣of␣the␣pulses␣in␣the␣input␣rhythm.␣1␣is␣the␣first␣pulse␣of␣the
input␣rhythm,␣2␣the␣second␣pulse,␣and␣do␣on.␣The␣pattern␣is␣inserted␣at␣pulses␣denoted␣in␣<pulses>.␣<pulses>␣
     may
also␣be␣a␣list␣of␣negative␣integers,␣where␣-1␣is␣the␣first␣pulse␣of␣the␣input␣rhythm.␣In␣that␣case,␣the␣
     pattern
is␣not␣inserted␣␣at␣pulses␣denoted␣in␣<pulses>.␣If␣<pulses>␣is␣nil,␣then␣the␣pattern␣is␣inserted␣at␣every␣
     pulse.
"
    (make-instance 'constraint
        :cstf #'post-insert-pattern
        :args (list pattern pulses))
)
```

rhythm-utils.lisp

The library of auxiliary methods used in the rest of the program.

```
(in-package :om)


(defun get-proportions (bar)
    "Get␣the␣rhythmic␣proportions␣of␣a␣bar␣((N␣D)␣ps)."
    (cadr bar)
)

(defun get-bars (tree)
    "Get␣the␣bars␣of␣a␣tree␣(?␣bars)"
    (cadr tree)
)

(defun convert(drt)
    "Convert␣a␣musical␣note␣duration␣to␣a␣tick␣duration."
    (and drt (/ (denominator drt) (numerator drt)))
)

(defun create-coeffs (n x)
    "Return␣a␣list␣of␣n␣elements␣equal␣to␣x."
    (cond ((<= n 0) nil)
          (t (cons x (create-coeffs (- n 1) x))))
)

(defun or* (x y)
    "Return␣x*y␣if␣neither␣are␣nil."
    (and x y (* x y))
)

(defun or/ (x y)
    "Return␣x/y␣if␣neither␣are␣nil."
    (and x y (/ x y))
)
```

```
;;;;;;;;;;;;;;;;;
; GET SIGNATURE ;
;;;;;;;;;;;;;;;;;

(defun get-signature (bar)
    "Get␣the␣signature␣(N␣D)␣of␣a␣bar␣((N␣D)␣ps)."
    (car bar)
)

(defmethod get-N ((x voice))
    "Get␣the␣N␣part␣of␣the␣signature␣of␣the␣tree␣of␣the␣input␣voice␣as␣if
␣␣␣␣it␣was␣a␣long␣single␣bar."
    (apply '+ (mapcar #'(lambda(z) (first (get-signature z))) (get-bars (tree x))))
)

(defmethod get-N ((x list))
    "Get␣the␣N␣part␣of␣the␣signature␣of␣the␣input␣tree␣as␣if␣it␣was␣a␣long
␣␣␣␣single␣bar."
    (apply '+ (mapcar #'(lambda(z) (first (get-signature z))) (get-bars x)))
)

(defmethod get-D ((x voice))
    "Get␣the␣D␣part␣of␣the␣signature␣of␣the␣tree␣of␣the␣input␣voice."
    (second (get-signature (first (get-bars (tree x)))))
)

(defmethod get-D ((x list))
    "Get␣the␣D␣part␣of␣the␣signature␣of␣the␣tree␣of␣the␣input␣voice."
    (second (get-signature (first (get-bars x))))
)

;;;;;;;;;;;;;;;;;;;;;
; SHORTEST DURATION ;
;;;;;;;;;;;;;;;;;;;;;

(defun smallest-in-bar (bar)
    "Return␣the␣shortest␣note␣duration␣of␣the␣bar."
    (let ((sign (get-signature bar))
          (ps (mapcar #'abs (get-proportions bar))))
        (* (/ (second sign) (first sign)) (apply '+ ps)))
)

(defun smallest (x)
    "Return␣the␣shortest␣note␣duration␣of␣the␣input␣tree."
    (apply 'max (loop for b in (get-bars x) collect
        (smallest-in-bar b)))
)

(defmethod shortest ((x voice))
    "Return␣the␣shortest␣note␣duration␣of␣the␣tree␣of␣the␣input␣voice."
    (smallest (tree x))
)

(defmethod shortest ((x list))
    "Return␣the␣shortest␣note␣duration␣of␣the␣input␣tree."
    (smallest x)
)


;;;;;;;;;;;;;;;;;;;;;;;;;;
; POS IN TICKS FROM TREE ;
;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod tree2posticks ((tree list) s)
"Convert␣the␣tree␣to␣a␣list␣of␣event␣positions␣in␣ticks.

WARNING:␣when␣using␣this␣method,␣position␣are␣considered␣starting␣from␣1␣to␣be␣able␣to
```

```
represent␣the␣fact␣the␣first␣element␣can␣be␣a␣rest."
    (let ((ratios (tree2ratio tree)))
        (loop for i from 0 below (length ratios) collect
            (let ((x 0))
                (loop for j from 0 below i do
                    (setq x (+ x (* s (abs (nth j ratios))))))
                (if (> (nth i ratios) 0) (+ x 1) (- (+ x 1)))))
))

(defmethod tree2posticks ((tree voice) s) (tree2posticks (tree tree) s))

(defun only-pulse-ticks (ticks)
"Filter␣a␣list␣of␣positions␣in␣ticks␣to␣keep␣only␣the␣positions␣of␣pulses."
    (cond ((null ticks) nil)
          ((< (car ticks) 0) (only-pulse-ticks (cdr ticks)))
          (t (cons (- (car ticks) 1) (only-pulse-ticks (cdr ticks)))))
)

(defun only-rest-ticks (ticks)
"Filter␣a␣list␣of␣positions␣in␣ticks␣to␣keep␣only␣the␣positions␣of␣rests."
    (cond ((null ticks) nil)
          ((> (car ticks) 0) (only-pulse-ticks (cdr ticks)))
          (t (cons (- (abs (car ticks)) 1) (only-pulse-ticks (cdr ticks)))))
)

;;;;;;;;;;;;;;;;;;;;;;;
; BUILD TREE FROM SOL ;
;;;;;;;;;;;;;;;;;;;;;;;

(defun 0-erase (l)
"Filter␣a␣list␣to␣keep␣only␣the␣non-zero␣elements."
    (cond
        ((null l) nil)
        ((= 0 (car l)) (0-erase (cdr l)))
        (t (cons (car l) (0-erase (cdr l))))
    )
)

(defun build-rtree (tree drt s)
"Build␣a␣rhythm␣tree␣from␣a␣list␣of␣durations␣in␣ticks."
    (let ((signs (mapcar #'get-signature (get-bars tree)))
           drts)

        (setq drts (mapcar #'(lambda (x) (/ x s)) (0-erase drt)))
        (reducetree (mktree drts signs)))
)
```

Appendix B

GiL

This appendix provides more information abput GiL. It first suggests way of extending GiL with new constraints and features, and then shows the source code of the current GiL version. An up-to-date version can be found on Github at `https://github.com/blapiere/GiL`.

B.1 How to extend GiL

B.1.1 Tutorial: adding a constraint

This tutorial show an example of constraint wrapping from gecode to GiL, using the $abs(x, y)$ that expresses the $y = |x|$. Adding a use case of a constraint is a four-step process. The first step is to add a method that post this constraint in the class *WSpace*:

```
//space_wrapper.hpp
class WSpace : public IntMinimizeSpace
        ...
        void abs(int x, int y);
        ...

//space_wrapper.cpp
//Call the Gecode function abs() on this space and the integer
//variables at indices <x> and <y> of the IntVar vector.
void WSpace::abs(int x, int y) {
        Gecode::abs(*this, get_int_var(x), get_int_vars(y));
}
```

Then, a function must be created in the external C library (i.e. the Gecode Wrapper) with a void pointer parameter that will be cast to a *WSpace* pointer,

that calls the *abs* method:

```
//gecode_wrapper.hpp
extern "C"
        ...
        void abs(void* sp, int x, int y);
        ...

//gecode_wrapper.cpp
//Call the above-defined abs method of the WSpace referenced
//by <sp>.
void abs(void* sp, int x, int y) {
        return static_cast<WSpace*>(sp)->abs(x, y);
}
```

The third step is to to create a CFFI function in the lisp part of the interface to call the C function. The CFFI only specifies the name of the C function called, the name of the new Lisp function, the return type, the documentation and the type of the arguments:

```
;ll-gil.lisp
(cffi::defcfun ("abs" abs) :void
    "Post the constraint that |vid1| = vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)
```

Finally, the last step is to create a Lisp method that uses the *int-var* class (in this case), and get the *vid* field of the varibales (i.e. their index in the integer variables vector of the *WSpace*) to call the foreign function via CFFI:

```
;ui-gil.lisp
(defmethod g-abs (sp (v1 int-var) (v2 int-var))
    (abs sp (vid v1) (vid v2))
)
```

The next steps are to add other use cases, for example wher x is a fixed integer and y is an integer variable. The user should pay attention to the names of the function: some C functions names are not allowed in the CFFI foreign functions definition; For example, it is highly probable that the *abs* name is not authorized, and a call to the lisp function *abs* will provoke a memory error.

The user should also remember that any modification of the C++ files requires a new compilation in order the changes to take effect.

B.1.2 Branching strategies

In Gecode, branching strategies are set as in the following example:

```
//Post branching on the integer variables in x, beginning by the
//firt variable with the smallest domain (INT_VAR_SIZE_MIN) and
//assigning its trying its smallest value first (INT_VAL_MIN)
branch(*this, x, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
```

The third and fourth arguments are functions that return instances of strategy classes that extend the classes *VarBranch* and *ValBranch* respectively Some of the strategy functions require arguments to work properly. To wrap the strategies, the branching methods of the *WSpace* should convert their strategy selectors arguments to call to those functions. The challenge is to find a way to represent all the different types of arguments these functions can have in a way that can be carried from C to Lisp through CFFI.

B.1.3 Expressions

In order to include support for expressions in GiL, the wrapper should implement its own "minimodel" (see chapter 7: *Modeling convenience: MiniModel* in [Schulte et al., 2019]) that would allow to perform operations on variables. In practice, it would decompose a complex operation into atomic operation corresponding to temporary variables and post the constraint afterward.

B.2 Source code

The source code is divide into two main parts: the C Wrapper, responsible to wrap Gecode C++ methods into C functions, and the Lisp Wrapper that calls the functions of the C Wrapper in Lisp.

B.2.1 C Wrapper

The C Wrapper is responsible of converting C++ functions into C function. It composed of four files:

- **space_wrapper.hpp** declares the C++ classes that will wrap Gecode behaviour: WSpace encapsulates methods to create search spaces, add variables and post constraints and branching; WbabEngine and WdfsEngine encapsulate the creation of BAB and DFS engines and their method to search the next solution.

- **space_wrapper.cpp** implements the classes declared in space_wrapper.hpp.

- **gecode_wrapper.hpp** declares the external C library that will be called by the Lisp Wrapper.

- **gecode_wrapper.cpp** implements the functions defined in gecode_wrapper.hpp. Each of them calls a method of the classes defined in space_wrapper.hpp.

space_wrapper.hpp

Declares classes that encapsulate methods to create search spaces, add variables, post constraints and branching, and create search engine.

```
#ifndef space_wrapper_hpp
#define space_wrapper_hpp

#include <vector>
#include <iostream>
#include <stdlib.h>
#include <exception>
#include "gecode/kernel.hh"
#include "gecode/int.hh"
#include "gecode/search.hh"
#include "gecode/minimodel.hh"
#include "gecode/set.hh"

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

class WSpace: public IntMinimizeSpace {
protected:
    vector<IntVar> int_vars;
    vector<BoolVar> bool_vars;
    int i_size;
    int b_size;
    int cost_id;

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

    /**
     Return the IntVar contained in int_vars at index vid.
     */
    IntVar get_int_var(int vid);

    /**
```

```
     Return the BoolVar contained in bool_vars at index vid.
     */
    BoolVar get_bool_var(int vid);


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

    /**
     Return an IntVarArgs of size n, containing the n IntVars contained in
     int_vars at indices vids.
     */
    IntVarArgs int_var_args(int n, int* vids);

    /**
     Return a BoolVarArgs of size n, containing the n BoolVars contained in
     bool_vars at indices vids.
     */
    BoolVarArgs bool_var_args(int n, int* vids);

    /**
     Return an IntArgs of size n, containing the n values in vals
    */
    IntArgs int_args(int n, int* vals);

    /**
     Return the expression int_rel(vid, val)
    */
    BoolVar bool_expr_val(int vid, int int_rel, int val);

    /**
     Return the expression int_rel(vid1, vid2)
    */
    BoolVar bool_expr_var(int vid1, int int_rel, int vid2);

public:
    /**
     Default constructor
     */
    WSpace();

    //=========================
    //= Variables and domains =
    //=========================

    /**
     Add an IntVar to the WSpace ranging from min to max.
     In practice, push a new IntVar at the end of the vector int_vars.
     Return the index of the IntVar in int_vars
     */
    int add_intVar(int min, int max);

    /**
     Add an IntVar to the WSpace with domain dom of size s.
     In practice, push a new IntVar at the end of the vector int_vars.
     Return the index of the IntVar in int_vars
     */
    int add_intVarWithDom(int s, int* dom);

    /**
     Add n IntVars to the WSpace ranging from min to max.
     In practice, push n new IntVars at the end of the vector int_vars.
     Return the indices of the IntVars in int_vars.
     */
    int* add_intVarArray(int n, int min, int max);
```

```
/**
 Add n IntVars to the WSpace with domain dom of size s.
 In practice, push n new IntVars at the end of the vector int_vars.
 Return the indices of the IntVars in int_vars.
 */
int* add_intVarArrayWithDom(int n, int s, int* dom);

/**
 Return the number of IntVars in the space.
 */
int nvars();

enum {
    //Relations for BoolExpr
    B_EQ,
    B_NQ,
    B_LE,
    B_LQ,
    B_GQ,
    B_GR
};

/**
 Add a BoolVar to the WSpace ranging from min to max.
 In practice, push a new BoolVar at the end of the vector bool_vars.
 Return the index of the BoolVar in bool_vars
 */
int add_boolVar(int min, int max);

/**
 Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid, val).
 In practice, push a new BoolVar at the end of the vector bool_vars.
 Return the index of the BoolVar in bool_vars
 */
int add_boolVar_expr_val(int vid, int int_rel, int val);

/**
 Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid1, vid2).
 In practice, push a new BoolVar at the end of the vector bool_vars.
 Return the index of the BoolVar in bool_vars
 */
int add_boolVar_expr_var(int vid1, int int_rel, int vid2);

//======================
//= Posting constraints =
//======================

//=== INTVARS ===

/**
 Post a relation constraint between the IntVar denoted by vid and the val.
 */
void cst_val_rel(int vid, int rel_type, int val);

/**
 Post a relation constraint between the IntVars denoted by vid1 and vid2.
*/
void cst_var_rel(int vid1, int rel_type, int vid2);

/**
 Post a relation constraint between the n IntVars denoted by vids and the val.
*/
void cst_arr_val_rel(int n, int* vids, int rel_type, int val);

/**
 Post a relation constraint between the n IntVars denoted by vids and the the IntVar vid.
*/
```

```
void cst_arr_var_rel(int n, int* vids, int rel_type, int vid);

/**
 Post a relation constraint between the n IntVars denoted by vids.
*/
void cst_arr_rel(int n, int* vids, int rel_type);

/**
 Post a lexicographic relation constraint between the n1 IntVars denoted by vids1 and
 the n2 IntVars denoted by vids2.
*/
void cst_arr_arr_rel(int n1, int* vids1, int rel_type, int n2, int* vids2);

/**
 Post the constraint that the n IntVars denoted by vids are distinct
 */
void cst_distinct(int n, int* vids);

/**
 Post the linear constraint [c]*[vids] rel val.
 */
void cst_val_linear(int n, int* c, int* vids, int rel_type, int val);

/**
 Post the linear constraint [c]*[vids] rel_type vid.
*/
void cst_var_linear(int n, int* c, int* vids, int rel_type, int vid);

/**
 Post the constraint that |vid1| = vid2.
 */
void cst_abs(int vid1, int vid2);

/**
 Post the constraint that dom(vid) = d, where d is a set of size n.
 */
void cst_dom(int vid, int n, int* d);

/**
 Post the constraint that vid is included in {vids[0], ..., vids[n-1]}
 */
void cst_member(int n, int* vids, int vid);

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

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

/**
 Post the constraint that vid1 / vid2 = vid3
 and vid1 % vid2 = div4
 */
void cst_divmod(int vid1, int vid2, int vid3, int vid4);

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

/**
 Post the constraint that vid = min(vids).
 */
```

```
void cst_arr_min(int n, int* vids, int vid);

/**
 Post the constraint that vid = argmin(vids).
 */
void cst_argmin(int n, int* vids, int vid);

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

/**
 Post the constraint that vid = max(vids).
 */
void cst_arr_max(int n, int* vids, int vid);

/**
 Post the constraint that vid = argmax(vids).
 */
void cst_argmax(int n, int* vids, int vid);

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

/**
 Post the constraint that sqr(vid1) = vid2.
 */
void cst_sqr(int vid1, int vid2);

/**
 Post the constraint that sqrt(vid1) = vid2.
 */
void cst_sqrt(int vid1, int vid2);

/**
 Post the constraint that pow(vid1, n) = vid2.
 */
void cst_pow(int vid1, int n, int vid2);

/**
 Post the constraint that nroot(vid1, n) = vid2.
 */
void cst_nroot(int vid1, int n, int vid2);

/**
 Post the constraint that vid = sum(vids).
 */
void cst_sum(int vid, int n, int* vids);

/**
 Post the constraint that the number of variables in vids equal to val1 has relation rel_type
 with val2.
 */
void cst_count_val_val(int n, int* vids, int val1, int rel_type, int val2);

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

/**
 Post the constraint that the number of variables in vids equal to vid has relation rel_type
 with val.
```

```
 */
void cst_count_var_val(int n, int* vids, int vid, int rel_type, int val);

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

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

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

//=== BOOLVARS ===

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

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

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

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

//==========================
//= Exploration strategies =
//==========================

/**
 Post a branching strategy on the n IntVars in vids, with strategies denoted by var_strategy and
 val_strategy.
 */
void branch(int n, int* vids, int var_strategy, int val_strategy);

/**
 Post a branching strategy on the n BoolVars in vids, with strategies denoted by var_strategy and
 val_strategy.
 */
void branch_b(int n, int* vids, int var_strategy, int val_strategy);

//==================
//= Search support =
//==================

void cost(int vid);

virtual IntVar cost(void) const;
```

```
    WSpace(WSpace& s);

    virtual Space* copy(void);

    //=====================
    //= Getting solutions =
    //=====================

    /**
     Return the current values of the variable denoted by vid.
     */
    int value(int vid);

    /**
     Return the current values of the n variables denoted by vids.
     */
    int* values(int n, int* vids);

    //======================
    //= Printing solutions =
    //======================

    /**
     Print the n variables denoted by vids.
     */
    void print(int n, int* vids);
};

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

class WbabEngine {
protected:
    BAB<WSpace>* bab;
public:
    WbabEngine(WSpace* sp);
    ~WbabEngine();

    /**
     Search the next solution for this search engine.
     */
    WSpace* next();
};

class WdfsEngine {
protected:
    DFS<WSpace>* dfs;
public:
    WdfsEngine(WSpace* sp);
    ~WdfsEngine();

    /**
     Search the next solution for this search engine.
     */
    WSpace* next();
};

#endif
```

space_wrapper.cpp

Implements the methods declared in space_wrapper.hpp.

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

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

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

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

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

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

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

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

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

/**
 Return an IntArgs of size n, containing the n values in vals
*/
IntArgs WSpace::int_args(int n, int* vals) {
    IntArgs c(n);
    for(int i = 0; i < n; i++)
        c[i] = vals[i];
    return c;
```

```
}

/**
 Return the expression int_rel(vid, val)
*/
BoolVar WSpace::bool_expr_val(int vid, int int_rel, int val) {
    switch(int_rel) {
        case B_EQ: return expr(*this, get_int_var(vid) == val);
        case B_NQ: return expr(*this, get_int_var(vid) != val);
        case B_LE: return expr(*this, get_int_var(vid) < val);
        case B_LQ: return expr(*this, get_int_var(vid) <= val);
        case B_GQ: return expr(*this, get_int_var(vid) >= val);
        case B_GR: return expr(*this, get_int_var(vid) > val);
        default:
            cout << "Wrong␣expression␣type␣in␣BoolVar␣creation." << endl;
            return BoolVar();
    }
}

/**
 Return the expression int_rel(vid1, vid2)
*/
BoolVar WSpace::bool_expr_var(int vid1, int int_rel, int vid2) {
    switch(int_rel) {
        case B_EQ: return expr(*this, get_int_var(vid1) == get_int_var(vid2));
        case B_NQ: return expr(*this, get_int_var(vid1) != get_int_var(vid2));
        case B_LE: return expr(*this, get_int_var(vid1) < get_int_var(vid2));
        case B_LQ: return expr(*this, get_int_var(vid1) <= get_int_var(vid2));
        case B_GQ: return expr(*this, get_int_var(vid1) >= get_int_var(vid2));
        case B_GR: return expr(*this, get_int_var(vid1) > get_int_var(vid2));
        default:
            cout << "Wrong␣expression␣type␣in␣BoolVar␣creation." << endl;
            return BoolVar();
    }
}


//========================
//= Variables and domains =
//========================

/**
 Add an IntVar to the WSpace ranging from min to max.
 In practice, push a new IntVar at the end of the vector int_vars.
 Return the index of the IntVar in int_vars
 */
int WSpace::add_intVar(int min, int max) {
    int_vars.push_back(IntVar(*this, min, max));
    return i_size++;
}

/**
 Add an IntVar to the WSpace with domain dom of size s.
 In practice, push a new IntVar at the end of the vector int_vars.
 Return the index of the IntVar in int_vars
 */
int WSpace::add_intVarWithDom(int s, int* dom) {
    int_vars.push_back(IntVar(*this, IntSet(dom, s)));
    return i_size++;
}

/**
 Add n IntVars to the WSpace ranging from min to max.
 In practice, push n new IntVars at the end of the vector int_vars.
 Return the indices of the IntVars in int_vars.
 */
int* WSpace::add_intVarArray(int n, int min, int max) {
```

```
    int* vids = new int[n];
    for(int i = 0; i < n; i++)
        vids[i] = this->add_intVar(min, max);
    return vids;
}

/**
 Add n IntVars to the WSpace with domain dom of size s.
 In practice, push n new IntVars at the end of the vector int_vars.
 Return the indices of the IntVars in int_vars.
 */
int* WSpace::add_intVarArrayWithDom(int n, int s, int* dom) {
    int* vids = new int[n];
    for(int i = 0; i < n; i++)
        vids[i] = this->add_intVarWithDom(s, dom);
    return vids;
}

/**
 Return the number of IntVars in the space.
 */
int WSpace::nvars() {
    return i_size;
}

/**
 Add a BoolVar to the WSpace ranging from min to max.
 In practice, push a new BoolVar at the end of the vector bool_vars.
 Return the index of the BoolVar in bool_vars
 */
int WSpace::add_boolVar(int min, int max) {
    bool_vars.push_back(BoolVar(*this, min, max));
    return b_size++;
}

/**
 Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid, val).
 In practice, push a new BoolVar at the end of the vector bool_vars.
 Return the index of the BoolVar in bool_vars
 */
int WSpace::add_boolVar_expr_val(int vid, int int_rel, int val) {
    bool_vars.push_back(bool_expr_val(vid, int_rel, val));
    return b_size++;
}

/**
 Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid1, vid2).
 In practice, push a new BoolVar at the end of the vector bool_vars.
 Return the index of the BoolVar in bool_vars
 */
int WSpace::add_boolVar_expr_var(int vid1, int int_rel, int vid2) {
    bool_vars.push_back(bool_expr_var(vid1, int_rel, vid2));
    return b_size++;
}


//======================
//= Posting constraints =
//======================

//=== INTVAR ===

/**
 Post a relation constraint between the IntVar denoted by vid and the val.
 */
void WSpace::cst_val_rel(int vid, int rel_type, int val) {
    rel(*this, get_int_var(vid), (IntRelType) rel_type, val);
```

```
}

/**
 Post a relation constraint between the IntVars denoted by vid1 and vid2.
*/
void WSpace::cst_var_rel(int vid1, int rel_type, int vid2) {
    rel(*this, get_int_var(vid1), (IntRelType) rel_type, get_int_var(vid2));
}

/**
 Post a relation constraint between the n IntVars denoted by vids and the val.
*/
void WSpace::cst_arr_val_rel(int n, int* vids, int rel_type, int val) {
    rel(*this, int_var_args(n, vids), (IntRelType) rel_type, val);
}

/**
 Post a relation constraint between the n IntVars denoted by vids and the the IntVar vid.
*/
void WSpace::cst_arr_var_rel(int n, int* vids, int rel_type, int vid) {
    rel(*this, int_var_args(n, vids), (IntRelType) rel_type, get_int_var(vid));
}

/**
 Post a relation constraint between the n IntVars denoted by vids.
*/
void WSpace::cst_arr_rel(int n, int* vids, int rel_type) {
    rel(*this, int_var_args(n, vids), (IntRelType) rel_type);
}

/**
 Post a lexicographic relation constraint between the n1 IntVars denoted by vids1 and
 the n2 IntVars denoted by vids2.
*/
void WSpace::cst_arr_arr_rel(int n1, int* vids1, int rel_type, int n2, int* vids2) {
    rel(*this, int_var_args(n1, vids1), (IntRelType) rel_type, int_var_args(n2, vids2));
}

/**
 Post the constraint that all IntVars denoted by vids are distinct
 */
void WSpace::cst_distinct(int n, int* vids) {
    distinct(*this, int_var_args(n, vids));
}

/**
 Post the linear constraint [c]*[vids] rel_type val.
 */
void WSpace::cst_val_linear(int n, int* c, int* vids, int rel_type, int val) {
    linear(*this, int_args(n, c), int_var_args(n, vids), (IntRelType) rel_type, val);
}

/**
 Post the linear constraint [c]*[vids] rel_type vid.
*/
void WSpace::cst_var_linear(int n, int* c, int* vids, int rel_type, int vid) {
    linear(*this, int_args(n, c), int_var_args(n, vids), (IntRelType) rel_type, get_int_var(vid));
}

/**
 Post the constraint that |vid1| = vid2.
 */
void WSpace::cst_abs(int vid1, int vid2) {
    abs(*this, get_int_var(vid1), get_int_var(vid2));
}

/**
```

```
 Post the constaraint that dom(vid) = d.
 */
void WSpace::cst_dom(int vid, int n, int* d) {
    dom(*this, get_int_var(vid), IntSet(d, n));
}

/**
 Post the constraint that vid is included in {vids[0], ..., vids[n-1]}
*/
void WSpace::cst_member(int n, int* vids, int vid) {
    member(*this, int_var_args(n, vids), get_int_var(vid));
}

/**
 Post the constraint that vid1 / vid2 = vid3.
 */
void WSpace::cst_div(int vid1, int vid2, int vid3) {
    div(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
}

/**
 Post the constraint that vid1 % vid2 = vid3.
 */
void WSpace::cst_mod(int vid1, int vid2, int vid3) {
    mod(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
}

/**
 Post the constraint that vid1 / vid2 = vid3
 and vid1 % vid2 = div4
 */
void WSpace::cst_divmod(int vid1, int vid2, int vid3, int vid4) {
    divmod(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3), get_int_var(vid4));
}

/**
 Post the constraint that min(vid1, vid2) = vid3.
 */
void WSpace::cst_min(int vid1, int vid2, int vid3) {
    Gecode::min(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
}

/**
 Post the constraint that vid = min(vids).
 */
void WSpace::cst_arr_min(int n, int* vids, int vid) {
    Gecode::min(*this, int_var_args(n, vids), get_int_var(vid));
}

/**
 Post the constraint that vid = argmin(vids).
 */
void WSpace::cst_argmin(int n, int* vids, int vid) {
    Gecode::argmin(*this, int_var_args(n, vids), get_int_var(vid));
}

/**
 Post the constraint that max(vid1, vid2) = vid3.
 */
void WSpace::cst_max(int vid1, int vid2, int vid3) {
    Gecode::max(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
}

/**
 Post the constraint that vid = max(vids).
 */
void WSpace::cst_arr_max(int n, int* vids, int vid) {
```

```
    Gecode::max(*this, int_var_args(n, vids), get_int_var(vid));
}

/**
 Post the constraint that vid = argmax(vids).
 */
void WSpace::cst_argmax(int n, int* vids, int vid) {
    Gecode::argmax(*this, int_var_args(n, vids), get_int_var(vid));
}

/**
 Post the constraint that vid1 * vid2 = vid3.
 */
void WSpace::cst_mult(int vid1, int vid2, int vid3) {
    mult(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
}

/**
 Post the constraint that sqr(vid1) = vid2.
 */
void WSpace::cst_sqr(int vid1, int vid2) {
    sqr(*this, get_int_var(vid1), get_int_var(vid2));
}

/**
 Post the constraint that sqrt(vid1) = vid2.
 */
void WSpace::cst_sqrt(int vid1, int vid2) {
    Gecode::sqrt(*this, get_int_var(vid1), get_int_var(vid2));
}

/**
 Post the constraint that pow(vid1, n) = vid2.
 */
void WSpace::cst_pow(int vid1, int n, int vid2) {
    Gecode::pow(*this, get_int_var(vid1), n, get_int_var(vid2));
}

/**
 Post the constraint that nroot(vid1, n) = vid2.
 */
void WSpace::cst_nroot(int vid1, int n, int vid2) {
    nroot(*this, get_int_var(vid1), n, get_int_var(vid2));
}

/**
 Post the constraint that vid = sum(vids).
 */
void WSpace::cst_sum(int vid, int n, int* vids) {
    rel(*this, get_int_var(vid), IRT_EQ, expr(*this, sum(int_var_args(n, vids))));
}

/**
 Post the constraint that the number of variables in vids equal to val1 has relation rel_type
 with val2.
 */
void WSpace::cst_count_val_val(int n, int* vids, int val1, int rel_type, int val2) {
    count(*this, int_var_args(n, vids), val1, (IntRelType) rel_type, val2);
}

/**
 Post the constraint that the number of variables in vids equal to val has relation rel_type
 with vid.
 */
void WSpace::cst_count_val_var(int n, int* vids, int val, int rel_type, int vid) {
    count(*this, int_var_args(n, vids), val, (IntRelType) rel_type, vid);
}
```

```
/**
 Post the constraint that the number of variables in vids equal to vid has relation rel_type
 with val.
 */
void WSpace::cst_count_var_val(int n, int* vids, int vid, int rel_type, int val) {
    count(*this, int_var_args(n, vids), vid, (IntRelType) rel_type, val);
}

/**
 Post the constraint that the number of variables in vids equal to vid1 has relation rel_type
 with vid2.
 */
void WSpace::cst_count_var_var(int n, int* vids, int vid1, int rel_type, int vid2) {
    count(*this, int_var_args(n, vids), vid1, (IntRelType) rel_type, vid2);
}

/**
 Post the constraint that the number of distinct values in the n variables denoted by vids
 has the given rel_type relation with the variable vid.
 */
void WSpace::cst_nvalues(int n, int* vids, int rel_type, int vid) {
    nvalues(*this, int_var_args(n, vids), (IntRelType) rel_type, get_int_var(vid));
}

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


//=== BOOLVAR ===

/**
 Post the constraint that vid1 bool_op vid2 = val.
 */
void WSpace::cst_boolop_val(int vid1, int bool_op, int vid2, int val) {
    rel(*this, get_bool_var(vid1), (BoolOpType) bool_op, get_bool_var(vid2), val);
}

/**
 Post the constraint that vid1 bool_op vid2 = vid3.
 */
void WSpace::cst_boolop_var(int vid1, int bool_op, int vid2, int vid3) {
    rel(*this, get_bool_var(vid1), (BoolOpType) bool_op, get_bool_var(vid2), get_bool_var(vid3));
}

/**
 Post a relation constraint between vid and val.
 */
void WSpace::cst_boolrel_val(int vid, int rel_type, int val) {
    rel(*this, get_bool_var(vid), (IntRelType) rel_type, val);
}

/**
 Post a relation constraint between vid1 and vid2.
 */
void WSpace::cst_boolrel_var(int vid1, int rel_type, int vid2) {
    rel(*this, get_bool_var(vid1), (IntRelType) rel_type, get_bool_var(vid2));
}

//==========================
//= Exploration strategies =
```

```
//=========================

/**
 Post a branching strategy on the variables in vids, with strategies denoted by var_strategy and
 val_strategy.
 */
void WSpace::branch(int n, int* vids, int var_strategy, int val_strategy) {
    Gecode::branch(*this, int_var_args(n, vids), INT_VAR_SIZE_MIN(), INT_VAL_MIN()); //only this default
         branching for now.
}

/**
 Post a branching strategy on the n BoolVars in vids, with strategies denoted by var_strategy and
 val_strategy.
 */
void WSpace::branch_b(int n, int* vids, int var_strategy, int val_strategy) {
    Gecode::branch(*this, bool_var_args(n, vids), BOOL_VAR_NONE(), BOOL_VAL_MIN()); //default for now
}

//==================
//= Search support =
//==================

/**
 Define which variable, denoted by vid, will be considered as the cost.
 */
void WSpace::cost(int vid) {
    cost_id = vid;
}

IntVar WSpace::cost(void) const {
    return int_vars.at(cost_id);
}

WSpace::WSpace(WSpace& s): IntMinimizeSpace(s), int_vars(s.i_size), bool_vars(s.b_size), i_size(s.i_size),
     b_size(s.b_size), cost_id(s.cost_id) {
    //IntVars update
    vector<IntVar>::iterator itd, its;
    for(itd = int_vars.begin(), its = s.int_vars.begin(); itd != int_vars.end(); ++itd, ++its)
        itd->update(*this, *its);

    //BoolVars update
    vector<BoolVar>::iterator btd, bts;
    for(btd = bool_vars.begin(), bts = s.bool_vars.begin(); btd != bool_vars.end(); ++btd, ++bts)
        btd->update(*this, *bts);
}

Space* WSpace::copy(void) {
    return new WSpace(*this);
}

//=====================
//= Getting solutions =
//=====================

/**
 Return the current values of the variable denoted by vid.
 */
int WSpace::value(int vid) {
    return get_int_var(vid).val();
}

/**
 Return the current values of the n variables denoted by vids.
 */
int* WSpace::values(int n, int* vids) {
    int* vals = new int[n];
```

```
    for(int i = 0; i < n; i++)
        vals[i] = get_int_var(vids[i]).val();
    return vals;
}

//======================
//= Printing solutions =
//======================

void WSpace::print(int n, int* vids) {
    std::cout << "{";
    for(int i = 0; i < n; i++) {
        std::cout << get_int_var(vids[i]);
        if(i < n - 1) std::cout << ",␣";
    }
    std::cout << "}" << std::endl;
}

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

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

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

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

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

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

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

gecode_wrapper.hpp

Defines the C library that will be called by the Lisp Wrapper functions.

```
#ifndef gecode_wrapper_hpp
#define gecode_wrapper_hpp
```

```
#include <stdlib.h>

#ifdef __cplusplus
extern "C" {
#endif

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

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

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

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

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

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

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

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

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

/**
 Wraps the WSpace add_boolVar_expr_val method.
 */
int add_boolVar_expr_val(void* sp, int vid, int rel_type, int val);

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

```
int add_boolVar_expr_var(void* sp, int vid1, int rel_type, int vid2);

/**
 Wraps the WSpace cst_val_rel method.
 */
void val_rel(void* sp, int vid, int rel_type, int val);

/**
 Wraps the WSpace cst_var_relr method.
 */
void var_rel(void* sp, int vid1, int rel_type, int vid2);

/**
 Wraps the WSpace cst_arr_val_rel method.
 */
void arr_val_rel(void* sp, int n, int* vids, int rel_type, int val);

/**
 Wraps the WSpace cst_arr_var_rel method.
 */
void arr_var_rel(void* sp, int n, int* vids, int rel_type, int vid);

/**
 Wraps the WSpace cst_arr_rel method.
 */
void arr_rel(void* sp, int n, int* vids, int rel_type);

/**
 Wraps the WSpace cst_arr_arr_rel method.
 */
void arr_arr_rel(void* sp, int n1, int* vids1, int rel_type, int n2, int* vids2);

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

/**
 Wraps the WSpace cst_val_linear method.
 */
void val_linear(void* sp, int n, int* c, int* vids, int rel_type, int value);

/**
 Wraps the WSpace cst_var_linear method.
 */
void var_linear(void* sp, int n, int* c, int* vids, int rel_type, int vid);

/**
 Wraps the WSpace cst_abs method.
 */
void arithmetics_abs(void* sp, int vid1, int vid2);

/**
 Wraps the WSpace acst_div method.
 */
void arithmetics_div(void* sp, int vid1, int vid2, int vid3);

/**
 Wraps the WSpace cst_var_mod method.
 */
void arithmetics_mod(void* sp, int vid1, int vid2, int vid3);

/**
 Wraps the WSpace cst_divmod method.
 */
void arithmetics_divmod(void* sp, int vid1, int vid2, int vid3, int vid4);
```

```
/**
 Wraps the WSpace cst_min method.
 */
void arithmetics_min(void* sp, int vid1, int vid2, int vid3);

/**
 Wraps the WSpace cst_arr_min method.
 */
void arithmetics_arr_min(void* sp, int n, int* vids, int vid);

/**
 Wraps the WSpace cst_argmin method.
 */
void arithmetics_argmin(void* sp, int n, int* vids, int vid);

/**
 Wraps the WSpace cst_max method.
 */
void arithmetics_max(void* sp, int vid1, int vid2, int vid3);

/**
 Wraps the WSpace cst_arr_max method.
 */
void arithmetics_arr_max(void* sp, int n, int* vids, int vid);

/**
 Wraps the WSpace cst_argmax method.
 */
void arithmetics_argmax(void* sp, int n, int* vids, int vid);

/**
 Wraps the WSpace cst_mult method.
 */
void arithmetics_mult(void* sp, int vid1, int vid2, int vid3);

/**
 Wraps the WSpace cst_sqr method.
 */
void arithmetics_sqr(void* sp, int vid1, int vid2);

/**
 Wraps the WSpace cst_sqrt method.
 */
void arithmetics_sqrt(void* sp, int vid1, int vid2);

/**
 Wraps the WSpace cst_pow method.
 */
void arithmetics_pow(void* sp, int vid1, int n, int vid2);

/**
 Wraps the WSpace cst_nroot method.
 */
void arithmetics_nroot(void* sp, int vid1, int n, int vid2);

/**
 Wraps the WSpace cst_dom method.
 */
void set_dom(void* sp, int vid, int n, int* d);

/**
 Wraps the WSpace cst_member method.
 */
void set_member(void* sp, int n, int* vids, int vid);

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

```
 */
void rel_sum(void* sp, int vid, int n, int* vids);

/**
 Wraps the WSpace cst_count_val_val method.
 */
void count_val_val(void* sp, int n, int* vids, int val1, int rel_type, int val2);

/**
 Wraps the WSpace cst_count_val_var method.
 */
void count_val_var(void* sp, int n, int* vids, int val, int rel_type, int vid);

/**
 Wraps the WSpace cst_count_var_val method.
 */
void count_var_val(void* sp, int n, int* vids, int vid, int rel_type, int val);

/**
 Wraps the WSpace cst_count_var_var method.
 */
void count_var_var(void* sp, int n, int* vids, int vid1, int rel_type, int vid2);

/**
 Wraps the WSpace cst_nvalues method.
 */
void nvalues(void* sp, int n, int* vids, int rel_type, int vid);

/**
 Wraps the WSpace cst_circuit method.
 */
void circuit(void* sp, int n, int* c, int* vids1, int* vids2, int vid);

/**
 Wraps the WSpace cst_boolop_val method.
 */
void val_boolop(void* sp, int vid1, int bool_op, int vid2, int val);

/**
 Wraps the WSpace cst_boolop_var method.
 */
void var_boolop(void* sp, int vid1, int bool_op, int vid2, int vid3);

/**
 Wraps the WSpace cst_boolrel_val method.
 */
void val_boolrel(void* sp, int vid, int rel_type, int val);

/**
 Wraps the WSpace cst_boolrel_var method.
 */
void var_boolrel(void* sp, int vid1, int rel_type, int vid2);

/**
 Wraps the WSpace branch method.
 */
void branch(void* sp, int n, int* vids, int var_strategy, int val_strategy);

/**
 Wraps the WSpace branch_b method.
 */
void branch_b(void* sp, int n, int* vids, int var_strategy, int val_strategy);

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

```
/**
 Wraps the WbabEngine constructor.
 */
void* new_bab_engine(void* sp);

/**
 Wraps the WbabEngine next method.
 */
void* bab_next(void* se);

/**
 Wraps the WdfsEngine constructor.
 */
void* new_dfs_engine(void* sp);

/**
 Wraps the WdfsEngine next method.
 */
void* dfs_next(void* se);

/**
 Wraps the WSpace destructor.
 */
void release(void* sp);

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

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

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

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

gecode_wrapper.cpp

Implements the C library by calling methods declared in space_wrapper.hpp.

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

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

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

```
    return static_cast<WSpace*>(sp)->add_intVar(min, max);
}

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

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

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

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

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

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

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

/**
 Wraps the WSpace cst_val_rel method.
 */
void val_rel(void* sp, int vid, int rel_type, int val) {
    return static_cast<WSpace*>(sp)->cst_val_rel(vid, rel_type, val);
}

/**
 Wraps the WSpace cst_var_relr method.
 */
void var_rel(void* sp, int vid1, int rel_type, int vid2) {
    return static_cast<WSpace*>(sp)->cst_var_rel(vid1, rel_type, vid2);
}

/**
```

```
 Wraps the WSpace cst_arr_val_rel method.
 */
void arr_val_rel(void* sp, int n, int* vids, int rel_type, int val) {
    return static_cast<WSpace*>(sp)->cst_arr_val_rel(n, vids, rel_type, val);
}

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

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

/**
 Wraps the WSpace cst_arr_arr_rel method.
 */
void arr_arr_rel(void* sp, int n1, int* vids1, int rel_type, int n2, int* vids2) {
    return static_cast<WSpace*>(sp)->cst_arr_arr_rel(n1, vids1, rel_type, n2, vids2);
}

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

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

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

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

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

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

```
}

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

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

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

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

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

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

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

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

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

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

```
 */
void arithmetics_sqrt(void* sp, int vid1, int vid2) {
    return static_cast<WSpace*>(sp)->cst_sqrt(vid1, vid2);
}

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

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

/**
 Wraps the WSpace cst_dom method.
 */
void set_dom(void* sp, int vid, int n, int* d) {
    return static_cast<WSpace*>(sp)->cst_dom(vid, n, d);
}

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

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

/**
 Wraps the WSpace cst_count_val_val method.
 */
void count_val_val(void* sp, int n, int* vids, int val1, int rel_type, int val2) {
    return static_cast<WSpace*>(sp)->cst_count_val_val(n, vids, val1, rel_type, val2);
}

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

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

/**
 Wraps the WSpace cst_count_var_var method.
 */
void count_var_var(void* sp, int n, int* vids, int vid1, int rel_type, int vid2) {
    return static_cast<WSpace*>(sp)->cst_count_var_var(n, vids, vid1, rel_type, vid2);
}
```

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

/**
 Wraps the WSpace cst_circuit method.
 */
void circuit(void* sp, int n, int* c, int* vids1, int* vids2, int vid) {
    return static_cast<WSpace*>(sp)->cst_circuit(n, c, vids1, vids2, vid);
}

/**
 Wraps the WSpace cst_boolop_val method.
 */
void val_boolop(void* sp, int vid1, int bool_op, int vid2, int val) {
    return static_cast<WSpace*>(sp)->cst_boolop_val(vid1, bool_op, vid2, val);
}

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

/**
 Wraps the WSpace cst_boolrel_val method.
 */
void val_boolrel(void* sp, int vid, int rel_type, int val) {
    return static_cast<WSpace*>(sp)->cst_boolrel_val(vid, rel_type, val);
}

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

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

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

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

/**
 Wraps the WbabEngine constructor.
 */
```

```
void* new_bab_engine(void* sp) {
    WSpace* _sp = static_cast<WSpace*>(sp);
    return (void*) new WbabEngine(_sp);
}

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

/**
 Wraps the WdfsEngine constructor.
 */
void* new_dfs_engine(void* sp) {
    WSpace* _sp = static_cast<WSpace*>(sp);
    return (void*) new WdfsEngine(_sp);
}

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

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

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

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

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

B.2.2 Lisp Wrapper

The Lisp Wrapper calls the C functions declared in gecode_wrapper.hpp in Lisp using CFFI. It is composed of two files: **gecode-wrapper.lisp** is the lower-level API that calls directly the C functions, and **gecode-wrapper-ui.lisp** is a library

of methods that take advantage of the CLOS generic methods definitions to provide a more habitable way of calling the lower-level functions.

gecode-wrapper.lisp

A library of functions that calls C functions.

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

(in-package :gil)

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

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

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

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

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

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

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

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

(cffi::defcfun ("nvars" nvars) :int
    "Return␣the␣number␣of␣variables␣in␣the␣space."
    (sp :pointer)
)

;IntVar relation flags
(defparameter gil::IRT_EQ 0)    ; equality relation
(defparameter gil::IRT_NQ 1)    ; inequality
(defparameter gil::IRT_LQ 2)    ; Less or equal
(defparameter gil::IRT_LE 3)    ; Strictly lower
(defparameter gil::IRT_GQ 4)    ; Greater or equal
(defparameter gil::IRT_GR 5)    ; Strictly greater

(cffi::defcfun ("add_boolVar" add-bool-var-range) :int
    "Add␣a␣BoolVar␣ranging␣from␣l␣to␣h.␣Return␣the␣index␣to␣this␣BoolVar."
    (sp :pointer)
    (l :int)
    (h :int)
)

(cffi::defcfun ("add_boolVar_expr_val" add-bool-var-expr-val) :int
    "Add␣a␣BoolVar␣corresponding␣to␣the␣evalueation␣of␣rel-type(vid,␣val)."
    (sp :pointer)
    (vid :int)
    (rel-type :int)
    (val :int)
)

(cffi::defcfun ("add_boolVar_expr_var" add-bool-var-expr-var) :int
    "Add␣a␣BoolVar␣corresponding␣to␣the␣evalueation␣of␣rel-type(vid1,␣vid2)."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(cffi::defcfun ("val_rel" val-rel) :void
    "Post␣a␣variable/value␣rel␣constraint."
    (sp :pointer)
    (vid :int)
    (rel-type :int)
    (val :int)
)

(cffi::defcfun ("var_rel" var-rel) :void
    "Post␣a␣variable/variable␣rel␣constraint."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(cffi::defcfun ("arr_val_rel" arr-val-rel-aux) :void
    "Post␣a␣variable-array/value␣rel␣constraint."
    (sp :pointer)
    (n :int)
    (vids :pointer)
```

```
    (rel-type :int)
    (val :int)
)

(defun arr-val-rel (sp vids rel-type val)
    "Post␣a␣variable-array/value␣rel␣constraint."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (arr-val-rel-aux sp (length vids) x rel-type val))
)

(cffi::defcfun ("arr_var_rel" arr-var-rel-aux) :void
    "Post␣a␣variable-array/variable␣rel␣constraint."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (rel-type :int)
    (vid :int)
)

(defun arr-var-rel (sp vids rel-type vid)
    "Post␣a␣variable-array/variable␣rel␣constraint."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (arr-var-rel-aux sp (length vids) x rel-type vid))
)

(cffi::defcfun ("arr_rel" arr-rel-aux) :void
    "Post␣a␣variable-array␣rel␣constraint."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (rel-type :int)
)

(defun arr-rel (sp vids rel-type)
    "Post␣a␣variable-array␣rel␣constraint."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (arr-rel-aux sp (length vids) x rel-type))
)

(cffi::defcfun ("arr_arr_rel" arr-arr-rel-aux) :void
    "Post␣a␣variable-array/variable-array␣rel␣constraint."
    (sp :pointer)
    (n1 :int)
    (vids1 :pointer)
    (rel-type :int)
    (n2 :int)
    (vids2 :pointer)
)

(defun arr-arr-rel (sp vids1 rel-type vids2)
    "Post␣a␣variable-array/variable-array␣rel␣constraint."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids1))
          (y (cffi::foreign-alloc :int :initial-contents vids2)))
        (arr-arr-rel-aux sp (length vids1) x rel-type (length vids2) y))
)

(cffi::defcfun ("distinct" distinct-aux) :void
    "Post␣a␣distinct␣constraint␣on␣the␣n␣variables␣denoted␣in␣vids."
    (sp :pointer)
    (n :int)
    (vids :pointer)
)

(defun distinct (sp vids)
    "Post␣a␣distinct␣constraint␣on␣the␣variables␣denoted␣in␣vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (distinct-aux sp (length vids) x))
```

```
)

(cffi::defcfun ("val_linear" val-linear-aux) :void
    "Post␣a␣linear␣equation␣constraint."
    (sp :pointer)
    (n :int)
    (c :pointer)
    (vids :pointer)
    (rel-type :int)
    (val :int)
)

(defun val-linear (sp coeffs vars rel-type value)
    "Post␣a␣linear␣equation␣constraint.␣coeffs␣and␣vars␣must␣have␣the␣same␣number␣of␣elements."
    (let ((c (cffi::foreign-alloc :int :initial-contents coeffs))
          (x (cffi::foreign-alloc :int :initial-contents vars)))
        (val-linear-aux sp (length coeffs) c x rel-type value))
)

(cffi::defcfun ("var_linear" var-linear-aux) :void
    "Post␣a␣linear␣equation␣constraint."
    (sp :pointer)
    (n :int)
    (c :pointer)
    (vids :pointer)
    (rel-type :int)
    (vid :int)
)

(defun var-linear (sp coeffs vars rel-type vid)
    "Post␣a␣linear␣equation␣constraint.␣coeffs␣and␣vars␣must␣have␣the␣same␣number␣of␣elements."
    (let ((c (cffi::foreign-alloc :int :initial-contents coeffs))
          (x (cffi::foreign-alloc :int :initial-contents vars)))
        (var-linear-aux sp (length coeffs) c x rel-type vid))
)

(cffi::defcfun ("arithmetics_abs" ge-abs) :void
    "Post␣the␣constraint␣that␣|vid1|␣=␣vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)

(cffi::defcfun ("arithmetics_div" ge-div) :void
    "Post␣the␣constraint␣that␣vid3␣=␣vid1/vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi::defcfun ("arithmetics_mod" var-mod) :void
    "Post␣the␣constraint␣that␣vid1␣%␣vid2␣=␣vid3."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi::defcfun ("arithmetics_divmod" ge-divmod) :void
    "Post␣the␣constraint␣that␣vid3␣=␣vid1/vid2␣and␣vid4␣=␣vid1␣%␣vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
    (vid4 :int)
)
```

```
(cffi:defcfun ("arithmetics_min" ge-min) :void
    "Post␣the␣constraint␣that␣vid3␣=␣min(vid1,␣vid2)."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(defun set-member (sp vids vid)
```

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

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

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

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

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

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

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

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

(defun count-var-val (sp vids vid rel-type val)
```

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

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

(defun count-var-var (sp vids vid1 rel-type vid2)
    "Post␣the␣constraint␣that␣the␣number␣of␣variables␣in␣vids␣equal␣to␣vid1␣has␣relation
␣␣␣␣rel-type␣with␣vid2."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-var-var-aux sp (length vids) x vid1 rel-type vid2))
)

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

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

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

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

;BoolVar operation flags
(defparameter gil::BOT_AND 0)    ; logical and
(defparameter gil::BOT_OR 1)     ; logical or
(defparameter gil::BOT_IMP 2)    ; logical implication
```

```
(defparameter gil::BOT_EQV 3)    ; logical equivalence
(defparameter gil::BOT_XOR 4)    ; logical exclusive or

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

gecode-wrapper-ui.lisp

The user interface of GiL, that calls the lower-level functions with more elegance.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for int constraints ;
```

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

;REL
(defmethod g-rel (sp (v1 int-var) rel-type (v2 fixnum))
    "Post␣the␣constraint␣that␣v1␣rel-type␣v2."
    (val-rel sp (vid v1) rel-type v2))

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

(defmethod g-rel (sp (v1 list) rel-type (v2 null))
    (arr-rel sp (vid v1) rel-type))

(defmethod g-rel (sp (v1 list) rel-type (v2 fixnum))
    (arr-val-rel sp (vid v1) rel-type v2))

(defmethod g-rel (sp (v1 list) rel-type (v2 int-var))
    (arr-var-rel sp (vid v1) rel-type (vid v2)))

(defmethod g-rel (sp (v1 list) rel-type (v2 list))
    (arr-arr-rel sp (vid v1) rel-type (vid v2)))

;DISTINCT
(defmethod g-distinct (sp vars)
    "Post␣the␣constraint␣that␣the␣given␣vars␣are␣distinct."
    (distinct sp (vid vars)))

;LINEAR
(defmethod g-linear (sp coeffs vars rel-type (v fixnum))
    "Post␣the␣linear␣relation␣coeffs*vars␣rel-type␣v."
    (val-linear sp coeffs (vid vars) rel-type v))

(defmethod g-linear (sp coeffs vars rel-type (v int-var))
    (var-linear sp coeffs (vid vars) rel-type (vid v)))

;ARITHMETICS
(defmethod g-abs (sp (v1 int-var) (v2 int-var))
    "Post␣the␣constraints␣that␣v2␣=␣|v1|."
    (ge-abs sp (vid v1) (vid v2)))

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

(defmethod g-mod (sp (v1 int-var) (v2 int-var) (v3 int-var))
    "Post␣the␣constraints␣that␣v3␣=␣v1%v2."
    (var-mod sp (vid v1) (vid v2) (vid v3)))

(defmethod g-divmod (sp (v1 int-var) (v2 int-var) (v3 int-var) (v4 int-var))
    "Post␣the␣constraints␣that␣v3␣=␣v1/v2␣and␣v4␣=␣v1%v2."
    (ge-divmod sp (vid v1) (vid v2) (vid v3) (vid v4)))

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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

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

;NUMBER OF VALUES
(defmethod g-nvalues (sp vars rel-type (v int-var))
    "Post␣the␣constraints␣that␣v␣is␣the␣number␣of␣distinct␣values␣in␣vars."
    (nvalues sp (vid vars) rel-type (vid v)))
```

```
;HAMILTONIAN PATH/CIRCUIT
(defmethod g-circuit (sp costs vars1 vars2 v)
    "Post␣the␣constraint␣that␣values␣of␣vars1␣are␣the␣edges␣of␣an␣hamiltonian␣circuit␣in
␣␣␣␣the␣graph␣formed␣by␣the␣n␣variables␣in␣vars1,␣vars2␣are␣the␣costs␣of␣these␣edges␣described
␣␣␣␣by␣costs,␣and␣v␣is␣the␣total␣cost␣of␣the␣circuit,␣i.e.␣sum(vars2)."
    (hcircuit sp costs (vid vars1) (vid vars2) (vid v)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for bool constraints ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;OP
(defmethod g-op (sp (v1 bool-var) bool-op (v2 bool-var) (v3 fixnum))
    "Post␣the␣constraints␣that␣v1␣bool-op␣v2␣=␣v3."
    (val-bool-op sp (vid v1) bool-op (vid v2) v3))

(defmethod g-op (sp (v1 bool-var) bool-op (v2 bool-var) (v3 bool-var))
    (var-bool-op sp (vid v1) bool-op (vid v2) (vid v3)))

;REL
(defmethod g-rel (sp (v1 bool-var) rel-type (v2 fixnum))
    "Post␣the␣constraints␣that␣v1␣rel-type␣v2."
    (val-bool-rel sp (vid v1) rel-type v2))

(defmethod g-rel (sp (v1 bool-var) rel-type (v2 bool-var))
    (var-bool-rel sp (vid v1) rel-type (vid v2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for exploration ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod g-branch (sp (v int-var) var-strat val-strat)
    "Post␣a␣branching␣on␣v␣with␣strategies␣var-strat␣and␣val-strat."
    (branch sp (list (vid v)) var-strat val-strat))

(defmethod g-branch (sp (v bool-var) var-strat val-strat)
    (branch-b sp (list (vid v)) var-strat val-strat))

(defmethod g-branch (sp (v list) var-strat val-strat)
    (if (typep (car v) 'int-var)
        (branch sp (vid v) var-strat val-strat)
        (branch-b sp (vid v) var-strat val-strat)))

;cost
(defmethod g-cost (sp (v int-var))
    "Defines␣that␣v␣is␣the␣cost␣of␣sp."
    (set-cost sp (vid v)))

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

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

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

(defmethod search-engine (sp &optional (bab nil))
    "Creates␣a␣new␣search␣engine␣(dfs␣or␣bab)."
    (if bab
        (make-instance 'BAB-engine :bab (bab-engine-low sp))
        (make-instance 'DFS-engine :dfs (dfs-engine-low sp))))

;solution exist?
```

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

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

(defmethod search-next ((se DFS-engine))
    (sol? (dfs-next (dfs se))))

(defmethod search-next ((se null))
    nil)

;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for solutions ;
;;;;;;;;;;;;;;;;;;;;;;;;;

;values
(defmethod g-values (sp (v int-var))
    "Get␣the␣values␣assigned␣to␣v."
    (get-value sp (vid v)))

(defmethod g-values (sp (v list))
    (get-values sp (vid v)))

(defmethod g-values ((sp null) v)
    nil)

;print
(defmethod g-print (sp (v int-var))
    "Print␣v."
    (print-vars sp (list (vid v))))

(defmethod g-print (sp (v list))
    (print-vars sp (vid v)))

(defmethod g-print ((sp null) v)
    nil)
```

Appendix C

Experiments on the model

This appendix shows the experiments conducted to choose the model. It contains the code used to model the problem described in 4.2 as well as the results in a tabular.

C.1 Experiments

This section shows the source code of the different tests on models in C++ using Gecode.

C.1.1 Explicit position and duration

This model explicitly declares positions and durations as integer variables.

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

class ExplicitPOSDRT : public Space {
protected:
    IntVarArray drt;
    IntVarArray pos;
public:
    /* Constructor for the Space
     *
     * <dmin> and <dmax> are the min and max number of events.
     * <N> and <D> form the time signature of the output rhythm.
     * <smallest> is shortest authorized duration, e.g. 4 for a
     * fouth note (1/4)
     */
    ExplicitPOSDRT(int dmin, int dmax, int N, int D, int smallest) {
        //GENERAL
        int duration = N * smallest / D;
        drt = IntVarArray(*this, dmax, -duration, duration);
```

```
        pos = IntVarArray(*this, dmax, 0, duration);
        IntVarArgs _drt(*this, dmax, 0, duration);

        //there is at least dmin notes
        count(*this, drt, 0, IRT_LQ, dmax-dmin);

        //sum of notes durations is the total duration
        for(int i = 0; i < dmax; i++) {
            abs(*this, drt[i], _drt[i]);
            rel(*this, (pos[i] == duration) >> (drt[i] == 0));
        }
        rel(*this, sum(_drt) == duration);

        //positions general csts
        for(int i = 1; i < dmax; i++) {
            rel(*this, pos[i] == (pos[i-1] + _drt[i-1]));
        }
        rel(*this, pos[0], IRT_EQ, 0);

        //PROBLEM SPECIFIC
        count(*this, pos, 3*smallest/D, IRT_GQ, 1);
        count(*this, pos, 9*smallest/D, IRT_GQ, 1);
        for(int i = 1; i < dmax; i++) {
            rel(*this, (pos[i] == 3*smallest/D) >> (drt[i] == 6));
            rel(*this, (pos[i] == 9*smallest/D) >> (drt[i] == 6));
        }

        //branching
        branch(*this, drt, INT_VAR_SIZE_MIN(), INT_VAL_MED());
        branch(*this, pos, INT_VAR_SIZE_MIN(), INT_VAL_MED());

    }

    ExplicitPOSDRT(ExplicitPOSDRT& s) : Space(s) {
        drt.update(*this, s.drt);
        pos.update(*this, s.pos);
    }

    virtual Space* copy(void) {
        return new ExplicitPOSDRT(*this);
    }

    // print solution
    void print(void) const {
        ...
    }

    //get rhythmic sequence
    vector<int> get_rhythm(void) const {
        vector<int> vs;
        for(int i = 0; i < drt.size(); i++) {
            int v = drt[i].val();
            if(v != 0) vs.push_back(v);
        }
        return vs;
    }
};
```

C.1.2 Explicit duration only

This model explicitly declares durations as integer variables and deduces the positions: $pos_i = \sum_{j=0}^{i-1} drt_i$.

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

/* Returns the <n> first elements of the argument array
 * <iva>.
 */
IntVarArgs sub_array(int n, IntVarArgs iva) {
    IntVarArgs r(n);
    for(int i = 0; i < n; i++) {
        r[i] = iva[i];
    }
    return r;
}

class ExplicitDRT : public Space {
protected:
    IntVarArray drt;
public:

    /* Constructor for the Space
     *
     * <dmin> and <dmax> are the min and max number of events.
     * <N> and <D> form the time signature of the output rhythm.
     * <smallest> is shortest authorized duration, e.g. 4 for a
     * fouth note (1/4)
     */
    ExplicitDRT(int dmin, int dmax, int N, int D, int smallest) {
        //GENERAL
        int duration = N * smallest / D;
        drt = IntVarArray(*this, dmax, -duration, duration);
        IntVarArgs _drt(*this, dmax, 0, duration);

        //there is at least dmin notes
        count(*this, drt, 0, IRT_LQ, dmax-dmin);

        //sum of notes durations is the total duration
        for(int i = 0; i < dmax; i++) {
            abs(*this, drt[i], _drt[i]);
        }
        rel(*this, sum(_drt) == duration);

        //the position of event i = sum(0, i-1)(|drt|)
        IntVarArgs pos(*this, dmax, 0, duration);
        rel(*this, pos[0], IRT_EQ, 0);
        for(int i = 1; i < dmax; i++) {
            rel(*this, sum(sub_array(i, _drt)) == pos[i]);
            rel(*this, (pos[i] == duration) >> (drt[i] == 0));
        }

        //PROBLEM SPECIFIC
        for(int i = 1; i < dmax; i++) {
            rel(*this, (pos[i] == 3*smallest/D) >> (drt[i] == 6));
            rel(*this, (pos[i] == 9*smallest/D) >> (drt[i] == 6));
        }
        count(*this, pos, 3*smallest/D, IRT_EQ, 1);
        count(*this, pos, 9*smallest/D, IRT_EQ, 1);


        //branching
        branch(*this, drt, INT_VAR_SIZE_MIN(), INT_VAL_MED());

    }

    ExplicitDRT(ExplicitDRT& s) : Space(s) {
        drt.update(*this, s.drt);
    }
```

```
    virtual Space* copy(void) {
        return new ExplicitDRT(*this);
    }

    // print solution
    void print(void) const {
        ...
    }

    //get rhythmic sequence
    vector<int> get_rhythm(void) const {
        vector<int> vs;
        for(int i = 0; i < drt.size(); i++) {
            int v = drt[i].val();
            if(v != 0) vs.push_back(v);
        }
        return vs;
    }
};
```

C.1.3 Explicit position only

This model explicitly declares positions as integer variables and deduces the duration: $drt_{i-1} = pos_i - pos_{i-1}$.

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

class ExplicitPOS : public Space {
protected:
    IntVarArray pos;
    BoolVarArray rests;
    int duration; //needed in order to construct the output sequence
public:
    /* Constructor for the Space
    *
    * <dmin> and <dmax> are the min and max number of events.
    * <N> and <D> form the time signature of the output rhythm.
    * <smallest> is shortest authorized duration, e.g. 4 for a
    * fouth note (1/4)
    */
    ExplicitPOS(int dmin, int dmax, int N, int D, int smallest) {
        //GENERAL
        duration = N * smallest / D;
        rests = BoolVarArray(*this, dmax, 0, 1);
        pos = IntVarArray(*this, dmax, 0, duration);

        //there is at least dmin notes
        IntVarArgs n_ex(*this, dmax-1, 0, duration-1);
        for(int i = 1; i < dmax; i++) {
            rel(*this, n_ex[i-1] == pos[i] - pos[i-1]);
            rel(*this, (pos[i] == duration) >> (n_ex[i-1] == 0));
        }
        count(*this, n_ex, 0, IRT_LQ, dmax-dmin);

        //positions general csts
        for(int i = 1; i < dmax; i++) {
            rel(*this, pos[i] >= pos[i-1]);
        }
        rel(*this, pos[0], IRT_EQ, 0);
```

```
        //PROBLEM SPECIFIC
        count(*this, pos, 3*smallest/D, IRT_GQ, 1);
        count(*this, pos, 9*smallest/D, IRT_GQ, 1);
        for(int i = 2; i < dmax; i++) {
            rel(*this, (pos[i-1] == 3*smallest/D) >>
                ((pos[i] == pos[i-1] + 6) && (!rests[i-1])));
            rel(*this, (pos[i-1] == 9*smallest/D) >>
                ((pos[i] == pos[i-1] + 6) && (!rests[i-1])));
        }
        rel(*this, (pos[dmax-1] == 9*smallest/D) >> (!rests[dmax-1]));

        //branching
        branch(*this, pos, INT_VAR_SIZE_MIN(), INT_VAL_MED());
        branch(*this, rests, BOOL_VAR_NONE(), BOOL_VAL_MIN());
    }

    ExplicitPOS(ExplicitPOS& s) : Space(s) {
        rests.update(*this, s.rests);
        pos.update(*this, s.pos);

        duration = s.duration;
    }

    virtual Space* copy(void) {
        return new ExplicitPOS(*this);
    }

    // print solution
    void print(void) const {
        ...
    }

    //get rhythmic sequence
    vector<int> get_rhythm(void) const {
        vector<int> vs;
        for(int i = 0; i < pos.size()-1; i++) {
            if(pos[i+1].val() > pos[i].val()) {
                if(rests[i].val())
                    vs.push_back(pos[i].val() - pos[i+1].val());
                else
                    vs.push_back(pos[i+1].val() - pos[i].val());
            }
        }
        if(duration > pos[pos.size()-1].val()) {
            if(rests[pos.size()-1].val())
                vs.push_back(pos[pos.size()-1].val() - duration);
            else
                vs.push_back(duration - pos[pos.size()-1].val());
        }

        return vs;
    }
};
```

Explicit position and duration	Explicit duration only	Explicit position only
Distinct solutions		
108080	108080	108080
Equivalent solutions		
30800	30800	68960
Time $[ms]$		
539	403	163
530	404	164
529	405	162
529	404	163
531	403	162
529	404	162
530	404	163
529	404	163
532	405	163
529	405	163
544	407	163
534	408	163
535	407	163
535	407	164
537	407	162
534	407	162
535	407	162
534	407	163
535	407	163
535	407	163
Average time $[ms]$		
533.25	405.6	162.8

Table C.1: Results of the experiment on the rhythm model

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**