Music Processing Suite Documentation

User Guide

Version 1.12.0

26 August 2022

Table of Contents

Introduction

Music Processing Suite (MPS) is a software system for creating, visualizing, transforming, analyzing and generating musical compositions using advanced symbolic music representations. MPS is based on domain-specific models containing individual representations of musical contexts such as meter, tempo, rhythms, pitches, scales, harmonies, loudness, lyrics and more. Based on these models, the following functionality is provided:

- Composition language for intuitive, redundancy-free music modeling and notation
- Tools for creating scores and lead sheets
- Transformation infrastructure for conversions between various music representation formats such as MIDI, MusicXML, LilyPond, PDF, CSV, SuperCollider and more
- Context-sensitive music search functionality
- Analysis infrastructure for statistical music analysis and visualization
- Algorithmic composition based on evolutionary algorithms
- Modern Eclipse-based graphical user interface

Refer to the Quickstart Tutorial for a quick introduction of the features and visit www.musicprocessing.net for more information.

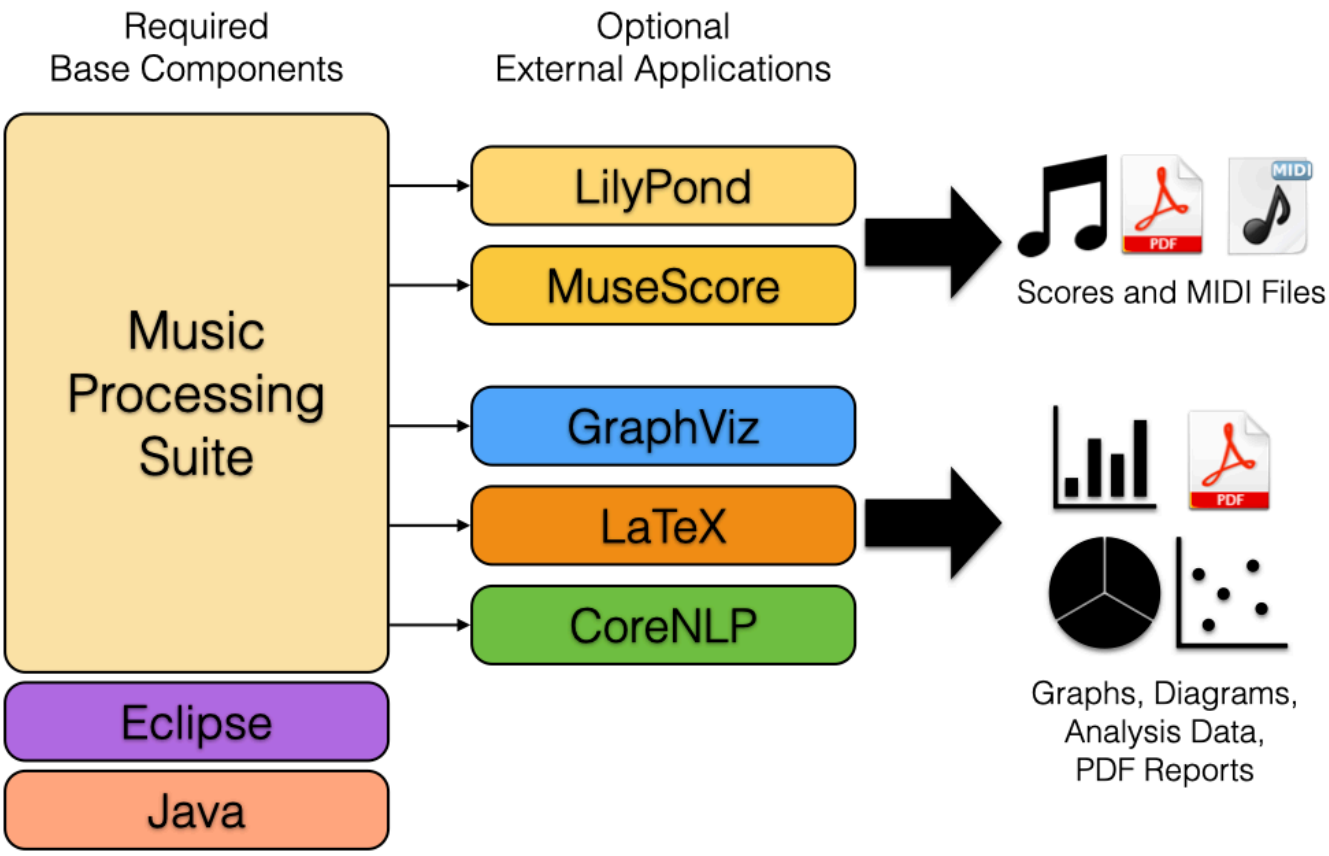
MPS runs on the Eclipse Platform. This document covers the installation of all required software components and many usage scenarios of MPS. The documentation is structured as follows:

This document also contains a Troubleshooting section with solutions for known issues.

Installation

This chapter contains instructions on how Music Processing Suite (MPS) and optional additional applications are installed.

Overview



MPS runs on the Eclipse platform, which requires a Java runtime environment. In order to use basic features of MPS, only Java and the stand-alone MPS application (which includes Eclipse) are required. Depending on your applications and use cases, additional external applications can be utilized by MPS for enhanced results and additional output formats.

Requirements

Music Processing Suite runs on Windows, Linux and Mac OS X. To run MPS, a Java Runtime Environment (JRE) or Java Development Kit (JDK) is required in version 17 or higher.

Quick Installation Guide

The following instructions help you to set up MPS as quickly as possible. Please refer to the detailed instructions below if any of the instructions are unclear.

Required software:

- Download a Java JDK (at least Java 17) from here and install it
- Download MPS here and extract it

The following instructions are for **Mac users only**:

- Move `MPS.app` to your `Applications` folder.
- Open a terminal (search for *Terminal* in spotlight).
- Copy the following command into the terminal and hit enter: `sudo xattr -rd com.apple.quarantine /Applications/MPS.app`
- Enter your Mac user password (it will not be displayed for security reasons) and hit enter
- You should now be able to start `MPS.app` like every other Mac application

Depending on what you want to do with MPS, you might consider installing the following additional applications:

- If you want to compose music using the MPS composition language and generate scores and/or MIDI files for your compositions, install either LilyPond or MuseScore:
  - Download LilyPond from lilypond.org, install it and configure the installation path under Preferences → LilyPond → Compiler
  - Download MuseScore from musescore.org and install it
- If you want to generate graphs of composition models or musical analysis results, download GraphViz from graphviz.org and install it. Note for Mac users: native installation packages might not be available, but it is possible to install GraphViz via MacPorts or Homebrew.
- If you want to generate PDF reports containing musical analysis results, charts, diagrams and tables, download LaTeX from latex-project.org and install it
- If you want to perform sentiment analysis of lyrics using Stanford CoreNLP, download it from stanfordnlp.github.io and extract it to a directory of your choice. Configure the directory path under Preferences → Music Processing Suite → Analysis → Sentiment Analysis.

Java Installation

Recommended Java distributions compatible with MPS can be downloaded from Azul. These are open source Java distributions which have less license restrictions than the Java packages provided by Oracle. Download and install a JDK for your operating system. The minimum version is Java 17.

How to Check if Java is Installed

If you don’t know whether Java is already installed or which version is installed, open a terminal/ command line on your operating system and enter

```
java -version
```

and hit enter. If java is installed, the first line of the output shows the Java version. For example, this is the first output line for a Java 17 JDK of the Zulu distribution:

```
openjdk version "17.0.3" 2022-04-19 LTS
```

Music Processing Suite Installation

Music Processing Suite can be installed as stand-alone software or into an existing Eclipse installation. If you install MPS for the first time or are not sure what to do, go for the stand-alone installation. If you are already familiar with Eclipse and have a recent version installed, you can consider installing MPS into your existing Eclipse instance.

Stand-alone Installation

Stand-alone versions of MPS can be downloaded at www.musicprocessing.net. Download the appropriate version for your operating system and processor architecture. Unpack the downloaded archive file and start MPS by starting the respective executable file (`MPS.exe` for Windows, `MPS` for Linux and `MPS.app` for Mac).

Important note for Mac users: If you have trouble running MPS on Mac because of security issues or OS X says the app is „damaged", execute the following command in a terminal (to open a terminal, simply search for *Terminal* in spotlight) to unlock the MPS application:

```
sudo xattr -rd com.apple.quarantine /Applications/MPS.app
```

Refer to this post for more details.

Installation Into an Existing Eclipse Instance via Update Site

Please skip this section if you already installed the stand-alone version in the previous step. If you already have a version of Eclipse installed, the MPS plugins can installed into your existing Eclipse installation. In your existing Eclipse instance, navigate to **Help** → **Install New Software** and enter the following update site URL:

```
https://updates.musicprocessing.net/
```

Select the Music Processing Suite Feature and click through the installation wizard. After the installation, Eclipse will be restarted and MPS will be ready to use.

Optional Components

The following software components are optional and are used for the following purposes:

- **LilyPond** and/or **MuseScore** to generate musical scores, lead sheets and MIDI files
- **Graphviz** for visualizing composition models and graph-based music analysis results
- **LaTeX** for generating PDF reports containing musical analysis results, charts, diagrams and tables
- **CoreNLP** for advanced sentiment analysis of lyrics

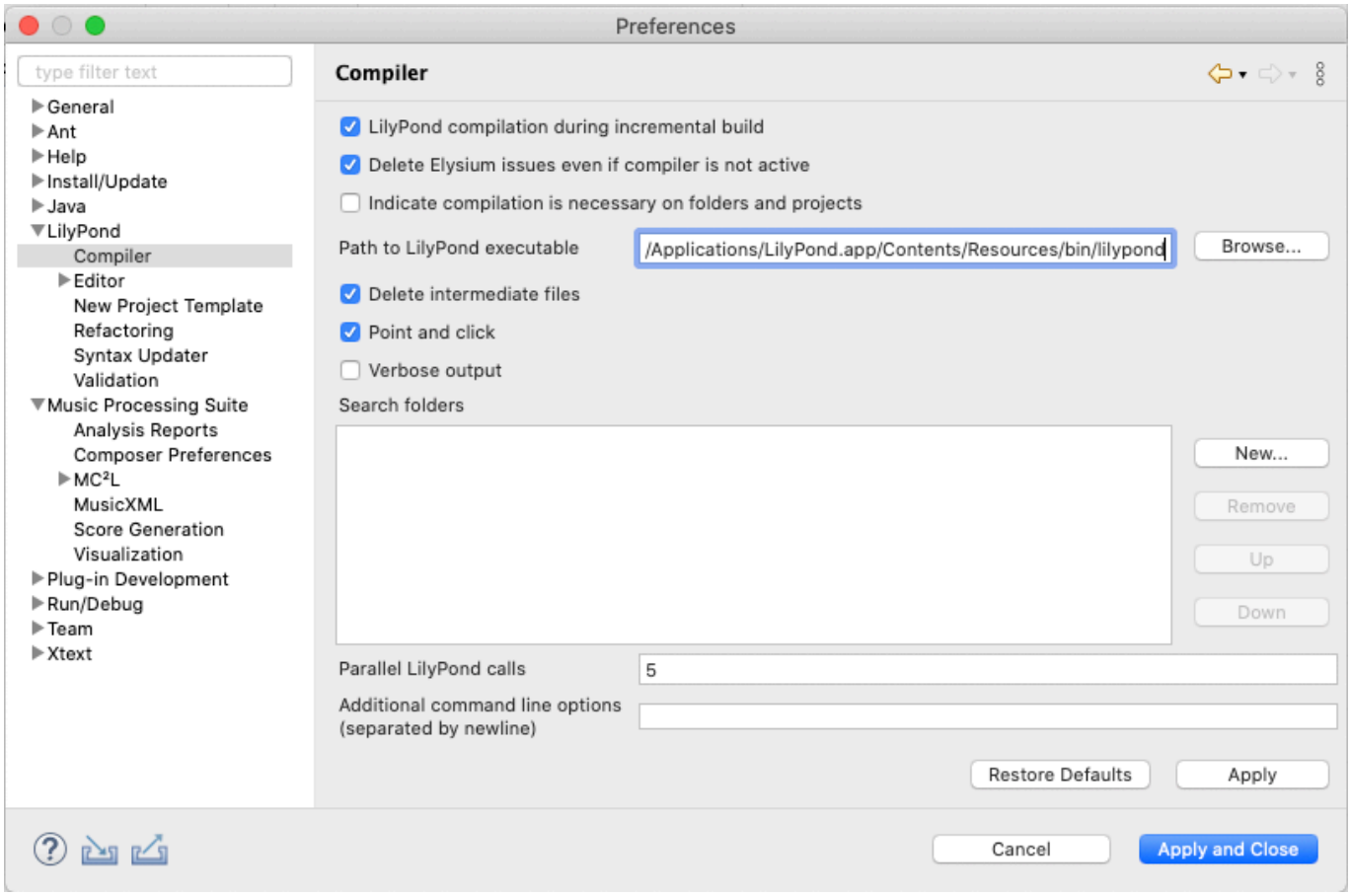
LilyPond Installation

Download LilyPond for your operating system from lilypond.org and install as described on the website.

LilyPond Configuration

To connect LilyPond to MPS, follow these steps:

1. Open the MPS Preferences
   1. On Windows and Linux: In the menu bar, choose Window → Preferences
   2. On Mac: In the menu bar, choose Music Processing Suite (application menu) → Preferences
2. Navigate to LilyPond → Compiler
3. Choose the location of the `lilypond` executable, e.g. `/Applications/LilyPond.app/Contents/Resources/bin/lilypond`.



MuseScore Installation

Download MuseScore for your operating system from musescore.org and install as described on the website.

MuseScore Configuration

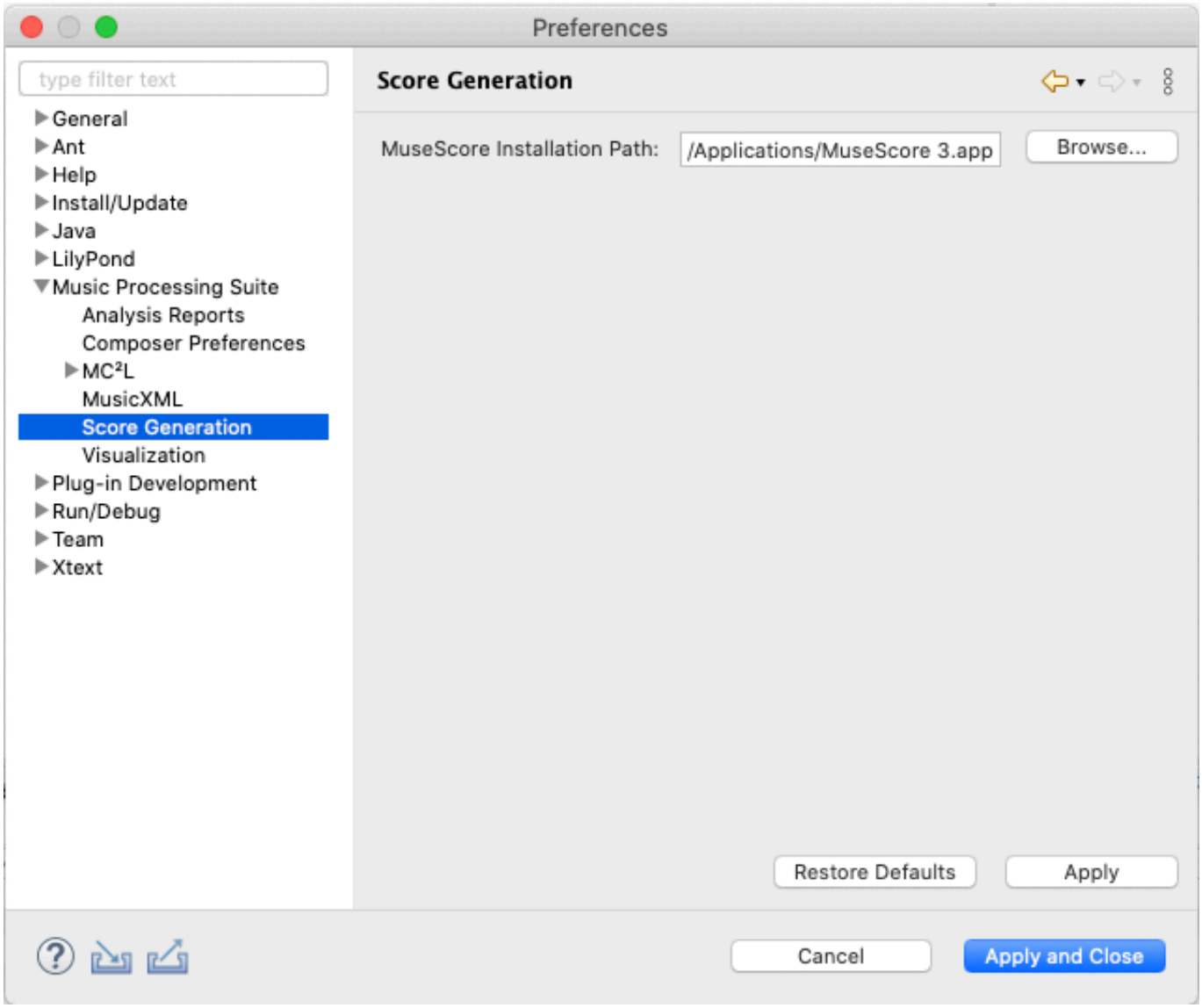
MPS will automatically search for MuseScore in the following locations depending on your operating system:

Operating System	Default Search Paths	Executable Names

Linux	`/usr/local/bin /usr/bin`	`mscore3 musescore3 mscore`
Mac OS X	`/Applications/MuseScore 3.app /usr/local/bin /usr/bin`	`mscore mscore3 musescore3`
Windows	`C:\Program*\MuseScore*`	`MuseScore3.exe`

If MuseScore cannot be located automatically or if you have installed MuseScore elsewhere, the path can be configured as follows:

1. Open the MPS Preferences
   1. On Windows and Linux: In the menu bar, choose Window → Preferences
   2. On Mac: In the menu bar, choose Music Processing Suite (application menu) → Preferences
2. Navigate to Music Processing Suite → Score Generation
3. Choose the location of the MuseScore installation, e.g. `/Applications/MuseScore 3.app`



Graphviz Installation

GraphViz is required if you want to visualize composition models in the form of context trees and/or display graph-based music analysis results.
The application can be downloaded from graphviz.org.

Optionally, install the program `graphviz-gui` if you require an additional viewer to display `.dot` files directly. You don't need `graphviz-gui` if you are comfortable with displaying composition models using an PDF viewer.

Note for Mac users: native installation packages might not be available, but it is possible to install GraphViz via MacPorts or Homebrew. The following commands can be entered into a terminal to install Homebrew and GraphViz:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/HEAD/install.sh)"
```

```
brew install graphviz
```

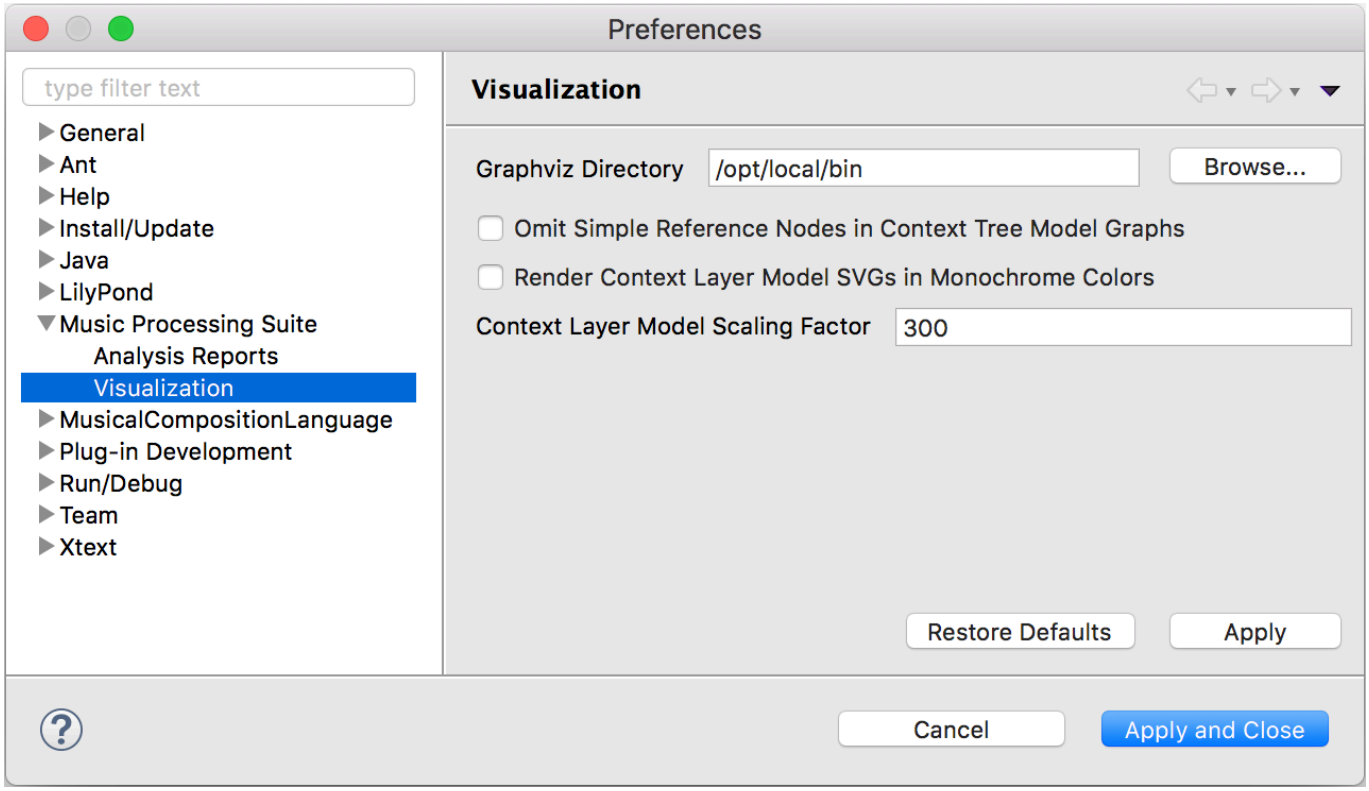
Graphviz Configuration

MPS will automatically search for GraphViz in the following locations depending on your operating system:

Operating System	Default Search Paths	Executable Name
Linux	`/usr/local/bin /usr/bin`	`dot`
Mac OS X	`/opt/local/bin /usr/local/bin /usr/bin`	`dot`
Windows	`C:\Program*\Graphviz*\bin`	`dot.exe`

If GraphViz cannot be located automatically or if you have installed GraphViz elsewhere, the path can be configured as follows:

1. Open the MPS Preferences
   1. On Windows and Linux: In the menu bar, choose Window → Preferences
   2. On Mac: In the menu bar, choose Music Processing Suite (application menu) → Preferences
2. Navigate to Music Processing Suite → Visualization
3. Choose the directory containing the Graphviz `dot` executable, e.g. `/opt/local/bin`.



LaTeX Installation

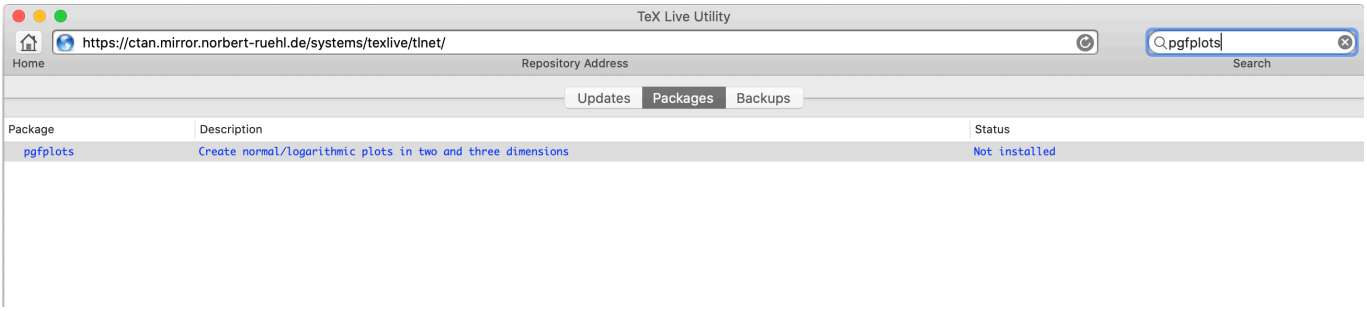
LaTeX is only required if you intend to analyze music with MPS and want to export statistical analysis results and graphs in the form of PDF reports. Visit latex-project.org to get a LaTeX distribution for your operating system. Refer to the following table for recommendations for your operating system.

Operating System	Recommended Distribution	Remarks
Windows	MiKTeX	It is recommended to enable the feature to install missing packages on-the-fly.
Linux	TeX Live	Custom LaTeX packages might be available for your specific Linux distribution.
Mac OS X	BasicTeX	BasicTeX is a minimal and small TeX distribution (about 80 MB), compared to the full MacTeX distribution which is over 4 GB.

Make sure that the following packages are installed in your LaTeX system using the TeX Live Utility. The following packages have to be installed manually when using a BasicTeX distribution on Mac OS X:

```
pgfplots
collcell
adjustbox
collectbox
```

When using BasicTeX on Mac OS X, download the TeX Live Utility here. Save it in your *Applications* folder and open it (if Mac OS X complains about security issues, right-click and open it or allow access in the security section of your system preferences). Possibly an update will be performed after the first start.



Once the tool is opened, switch to the *Packages* tab. Then enter packages to be installed in the search field on the top right. Right-click on the package and choose *Install* to perform the installation. Repeat this step for all packages listed above.

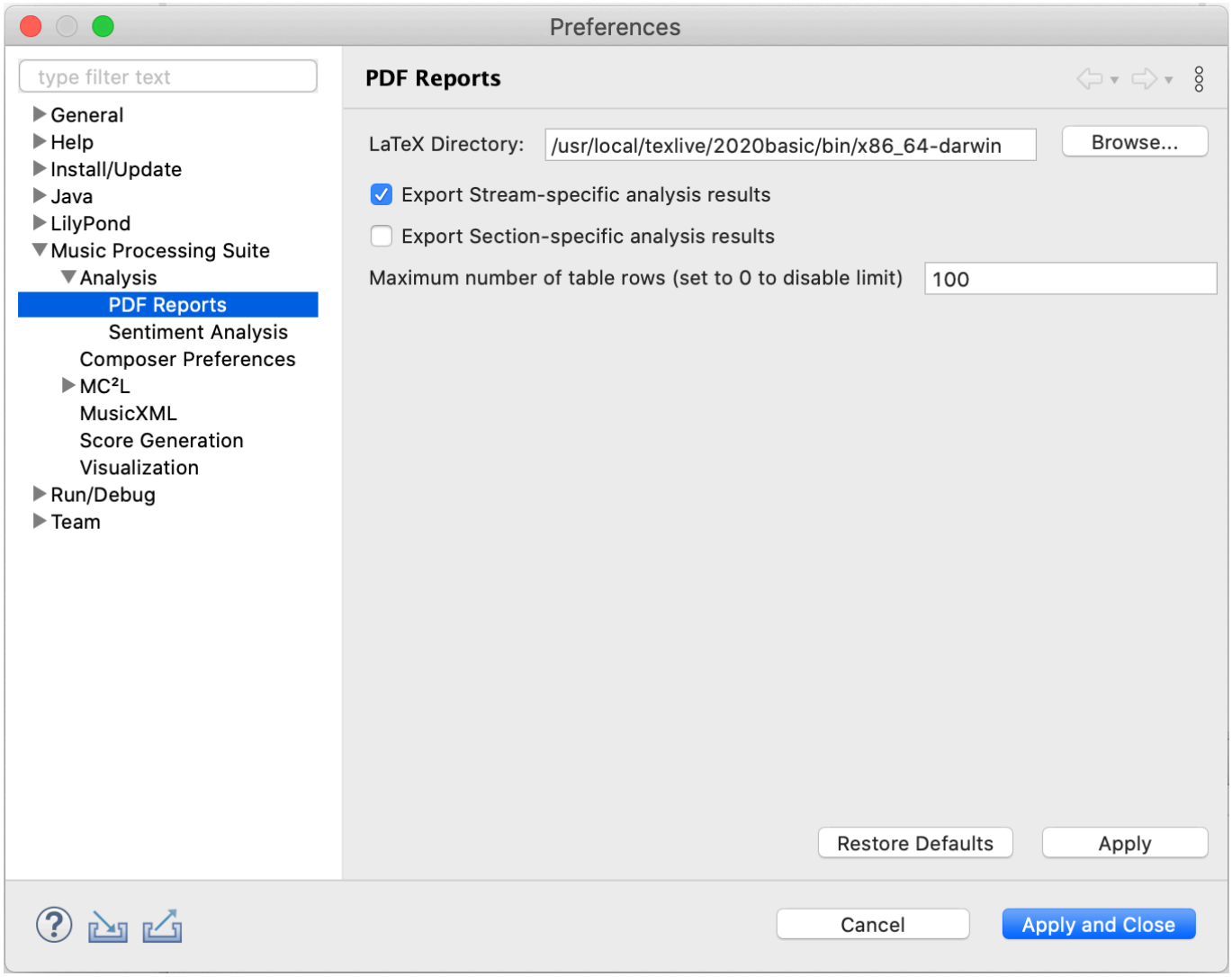
LaTeX Configuration

MPS will automatically search for LaTeX in the following locations depending on your operating system:

Operating System	Default Search Paths	Executable Name
Linux	`/usr/local/texlive /usr/share/texlive /usr/local/bin /usr/bin`	`lualatex`
Mac OS X	`/usr/local/texlive /usr/share/texlive /opt/local/bin /usr/local/bin /usr/bin`	`lualatex`
Windows	`C:\Program*\MiKTeX*`	`lualatex.exe`

If LaTeX cannot be located automatically or if you have installed LaTeX elsewhere, the path can be configured as follows:

1. Open the MPS Preferences
   1. On Windows and Linux: In the menu bar, choose Window → Preferences
   2. On Mac: In the menu bar, choose Music Processing Suite (application menu) → Preferences
2. Go to Music Processing Suite → Analysis → PDF Reports
3. Choose the location of the directory containing the LaTeX executables, e.g. `/usr/local/texlive/2020basic/bin/x86_64-darwin`.



CoreNLP Installation

CoreNLP is an advanced natural language processing framework developed by the Stanford NLP group. If a CoreNLP server is installed and running, it can be utilized by MPS for the analysis of sentiment polarities in lyrics.

To install CoreNLP, download it from stanfordnlp.github.io and extract it to a directory of your choice.

CoreNLP Configuration

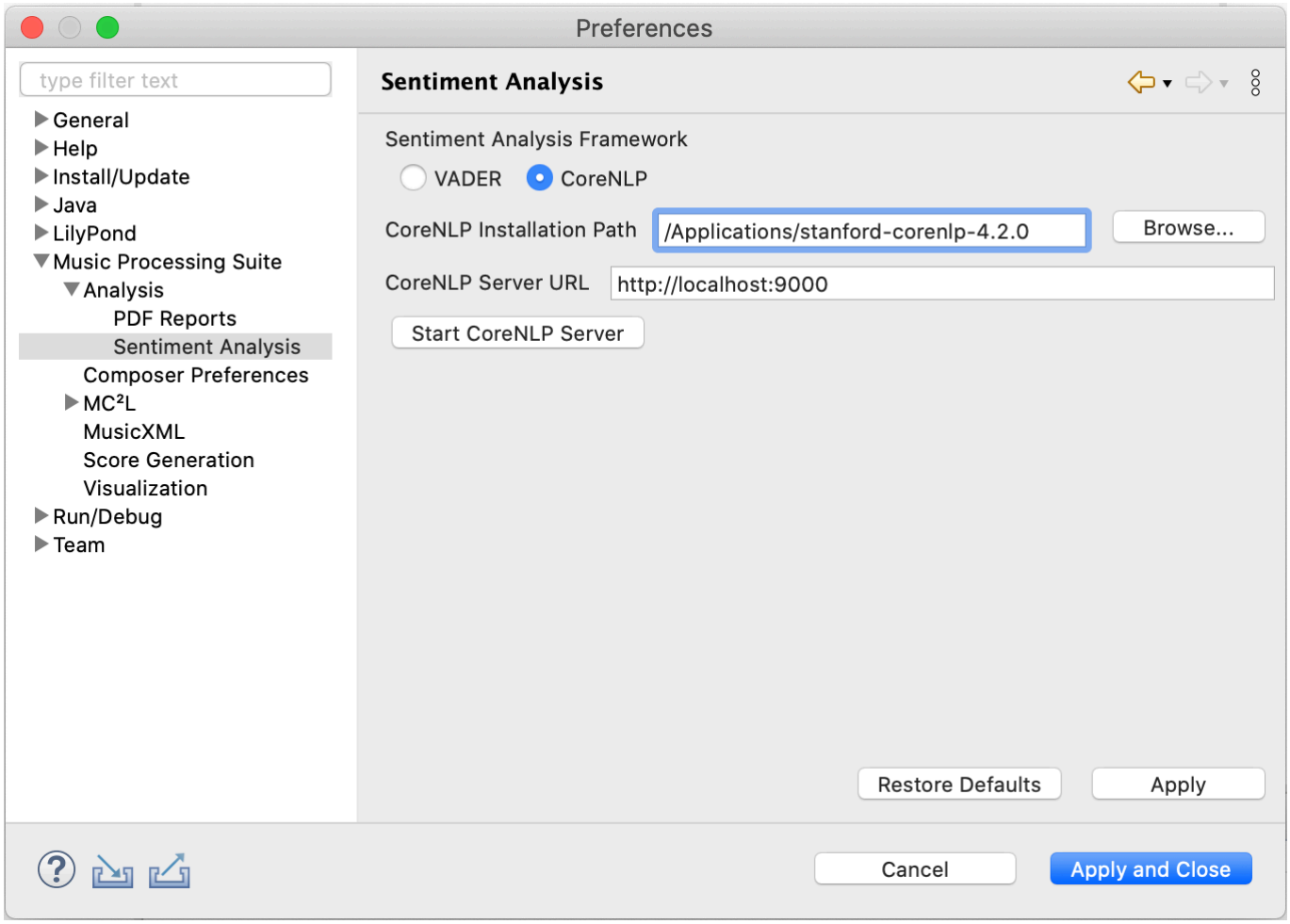
MPS will automatically search for CoreNLP in the following locations depending on your operating system:

Operating System	Default Search Paths	JAR File Name
Linux	`/opt/stanford-corenlp-*`	`stanford-corenlp-*.jar`
Mac OS X	`/Applications/stanford-corenlp-*`	`stanford-corenlp-*.jar`

Windows	`C:\Program*\stanford-corenlp-*`	`stanford-corenlp-*.jar`

If LaTeX cannot be located automatically or if you have installed LaTeX elsewhere, the path can be configured as follows:

1. Open the MPS Preferences
   1. On Windows and Linux: In the menu bar, choose Window → Preferences
   2. On Mac: In the menu bar, choose Music Processing Suite (application menu) → Preferences
2. Go to Music Processing Suite → Analysis → Sentiment Analysis
3. Select *CoreNLP* as sentiment analysis framework
4. Choose the location of the directory containing CoreNLP, e.g. `/Applications/stanford-corenlp-4.2.0`.



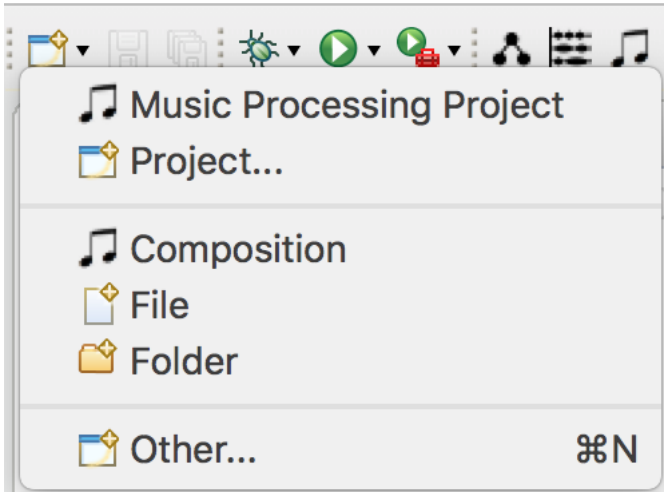
Quickstart Tutorial

This chapter provides a quick tutorial to become acquainted with the most important features of MPS. Refer to the remaining chapters for more detailed descriptions and explanations.

Creating Projects

Click the button

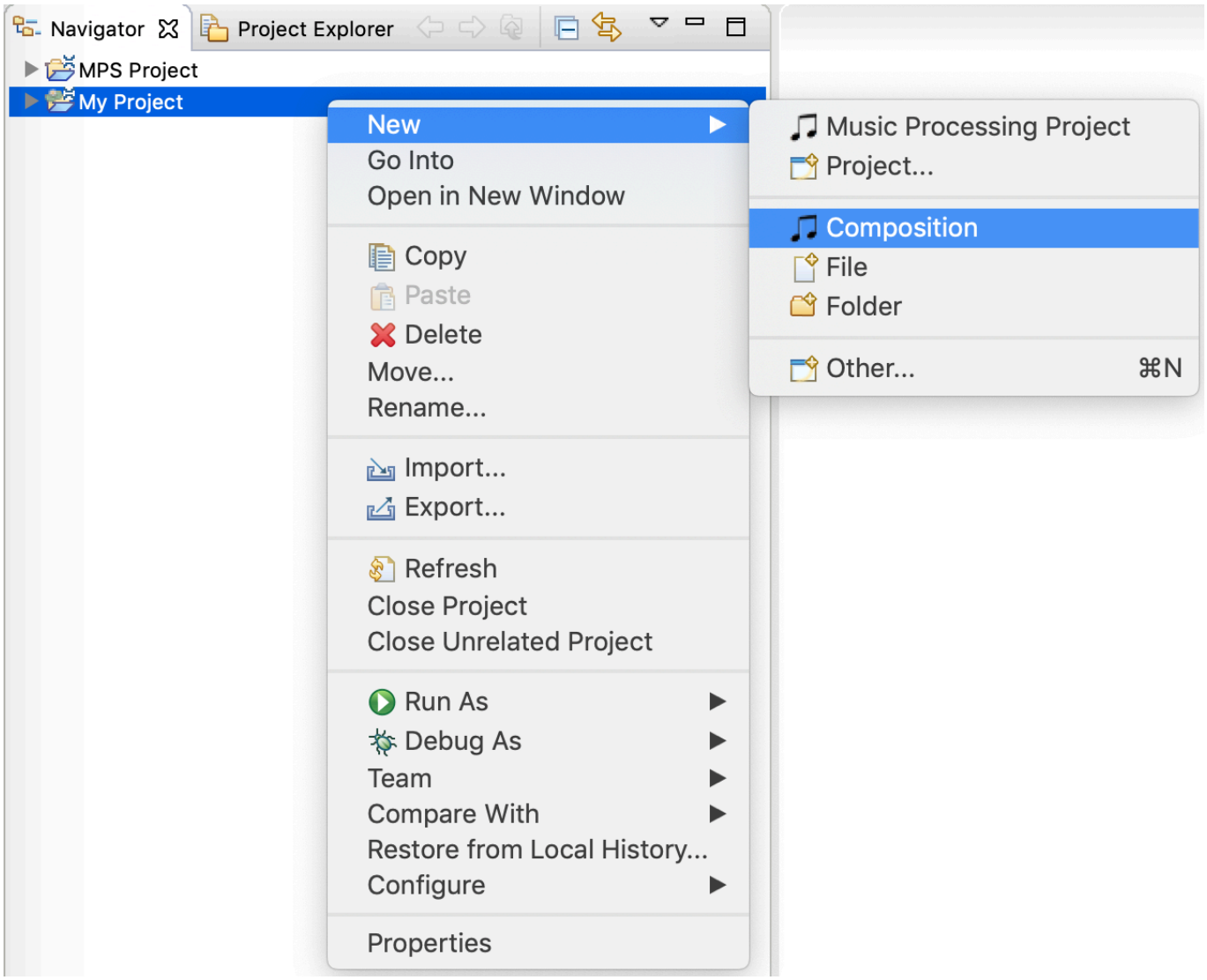
to open a menu for creating new projects and files:



Choose *Music Processing Project* to create a new project configured for MPS and enter a name for your project. Click *Finish* to complete the project creation.

Creating Compositions

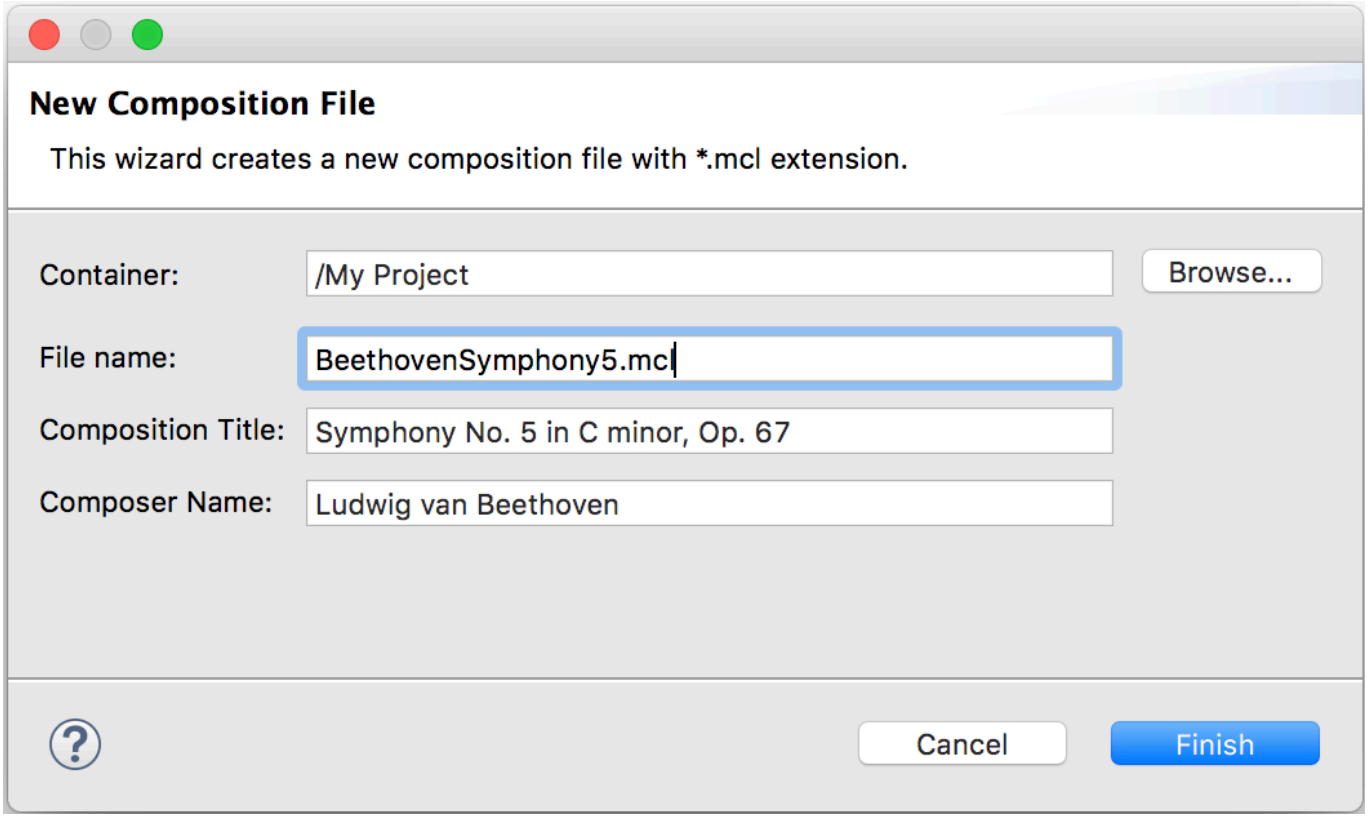
To create a new composition, right-click on a project or folder and choose *New* → *Composition*.



Alternatively, the menu introduced in the previous section can be used, which can be accessed using the button

.

The following wizard is opened:



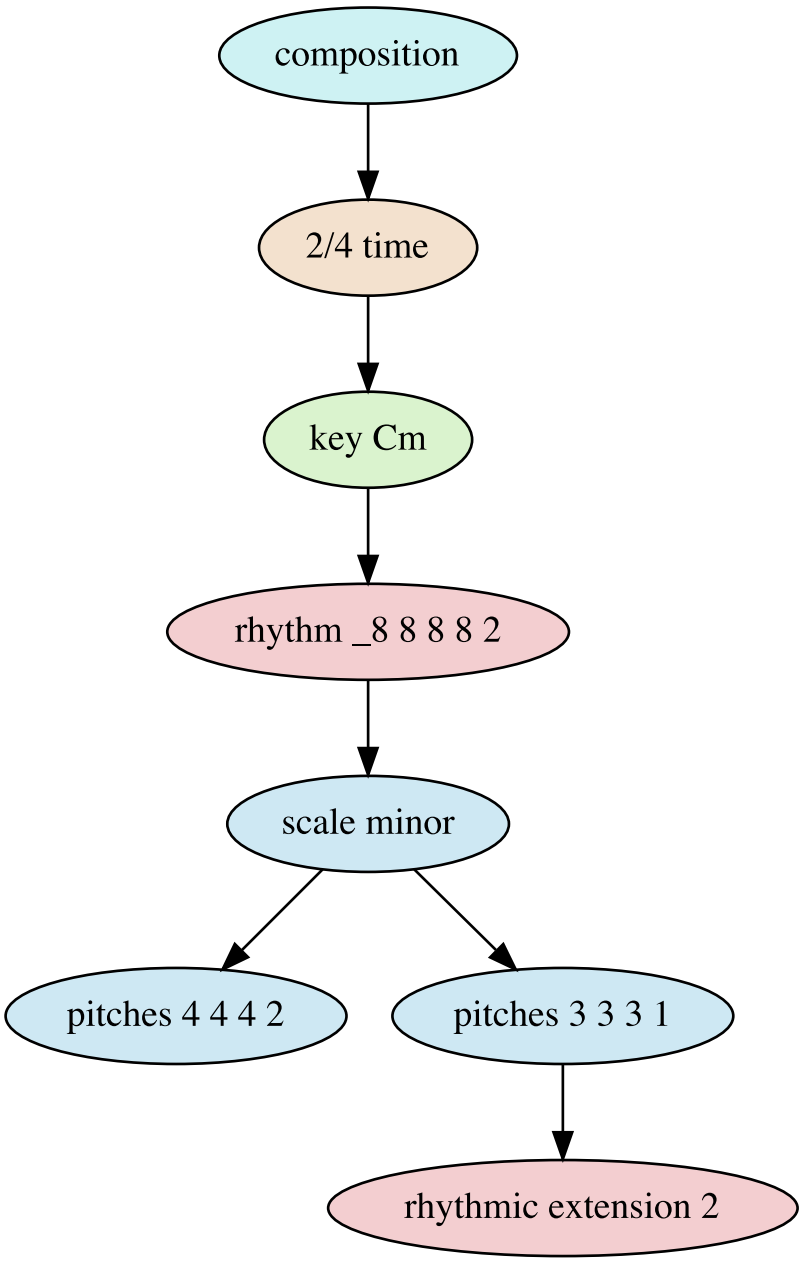
Enter the file name and optionally provide a title and the composer name. Click *Finish* to create your first composition. The composition editor will open in the center of the application, containing the specified metadata and an empty composition tree. Add the following code to add some musical content:

```
composition
{
    time 2/4, key Cm
    {
        rhythm _8 8 8 8 2
        {
            pitches 4 4 4 2
            pitches 3 3 3 1, rhythmicExtension 2
        }
    }
}
```

Visualizing Context Tree Composition Models

To view a graphical representation of the composition model, click the button

to render a tree graph using GraphViz. Note that GraphViz must be installed in configured to use this feature as described in the Installation chapter. The following graph results:



Visualizing Context Layer Composition Models

For a time-based representation of the resulting musical context layers, so called context layer models can be generated. Click on the button

to create an SVG file containing the layer model. The file will be opened with the default

application registered in your operating system to open scalable vector graphics files, e.g. a web browser. The corresponding context layer model looks like this:

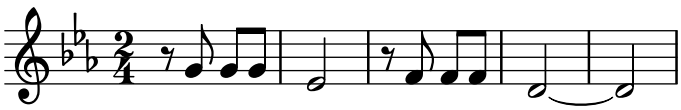
Stream 1	Meter (5)	2/4 time			2/4 time	2/4 time				2/4 time	2/4 time
	Key (1)	Cm									
	Harmony (1)	Cm									
	Harmonic Rhythm (2)	1				1.					
	Rhythm (10)	_8	8	8	8	2	_8	8	8	8	1
	Scale (1)	minor									
	Degrees (10)		4	4	4	2		3	3	3	1
	Pitches (10)		G4	G4	G4	Eb4		F4	F4	F4	D4
	Loudness (1)	loudness mf									
	Time (Measures)	1				2	3				4	5
	Time (Absolute)	0					1					2

Creating Scores and Lead Sheets

To create a score representation of a composition, click the button
♫
in the toolbar. If the score is created for the first time, a dialog with score generation options will be shown. After configuring the score options, click *Apply* and then *Run*.

MPS will compile a LilyPond file of the piece. The Elysium Eclipse plug-in will take care of compiling this file to a corresponding PDF and MIDI file. Note that LilyPond must be installed to use this feature as described in section LilyPond Installation.

The score resulting from the example looks like this:



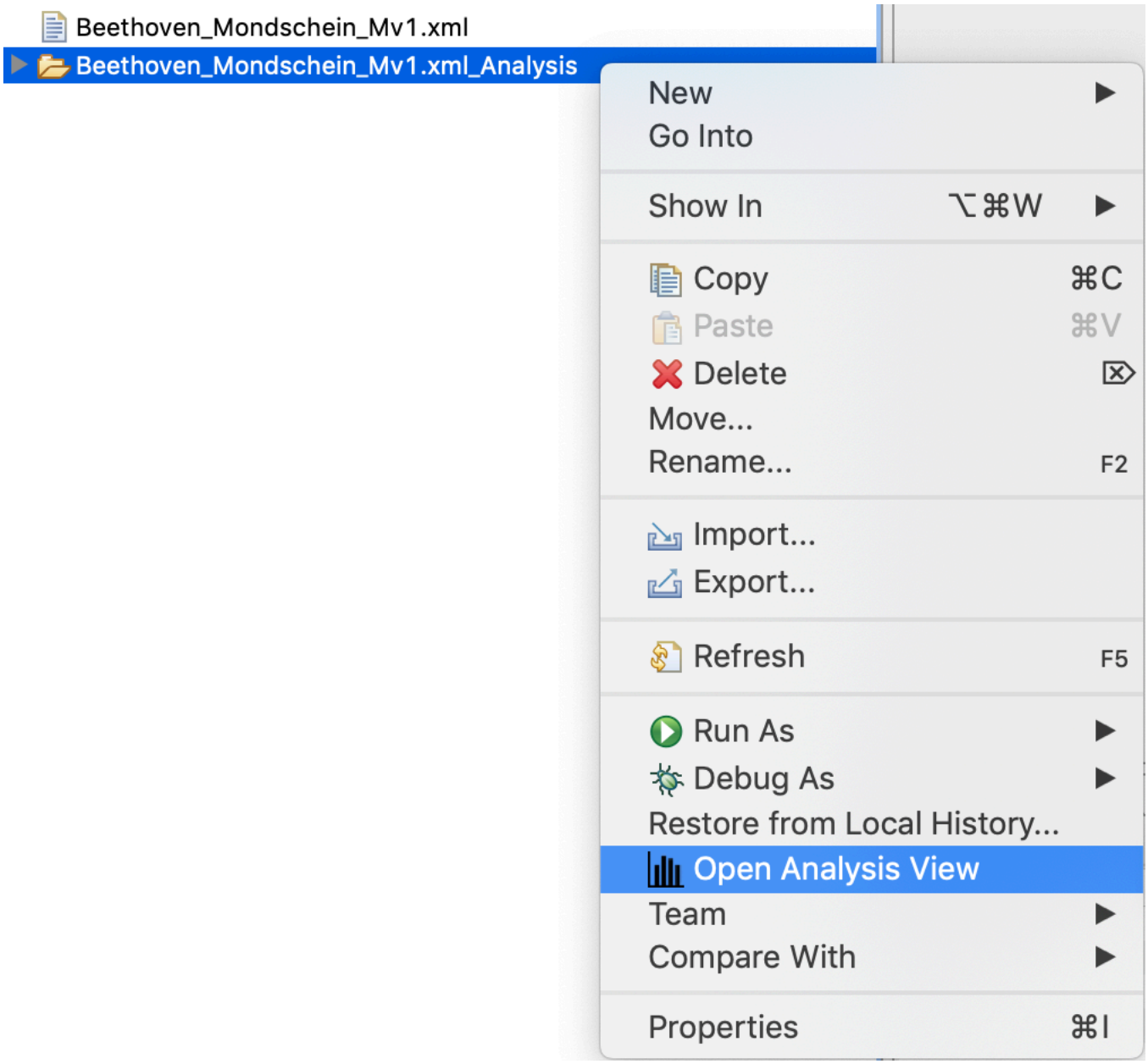
Options to configure the score generation process are documented here.

Analyzing Compositions

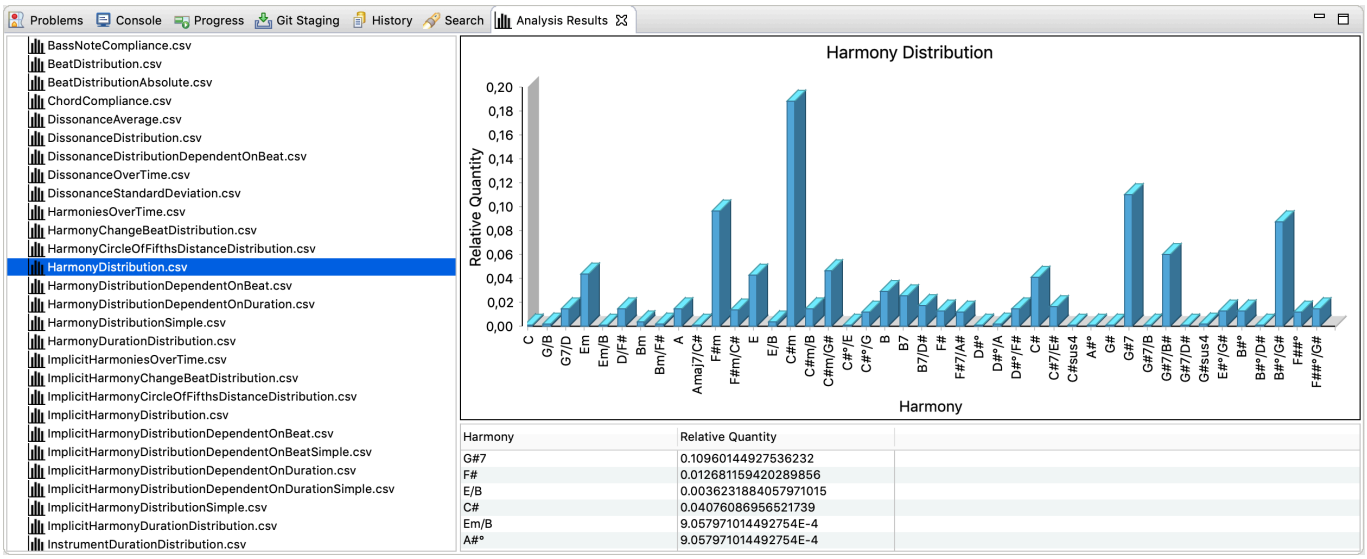
To perform musical analyses, select a file or folder to analyze and click the button

in the toolbar. If a file or folder is analyzed for the first time, a dialog with several analysis options opens. Either leave the configuration as it is or make adjustments as required, then click *Apply* and *Run*.

A new folder containing analysis results will be created next to the selected input resource. A number of CSV files and DOT graph files will be created in that folder. To explore and visualize the results, MPS provides a dedicated view, which can be opened by right-clicking the created analysis folder and selecting *Open Analysis View*:



The view is populated with data whenever a folder containing analysis results is selected.

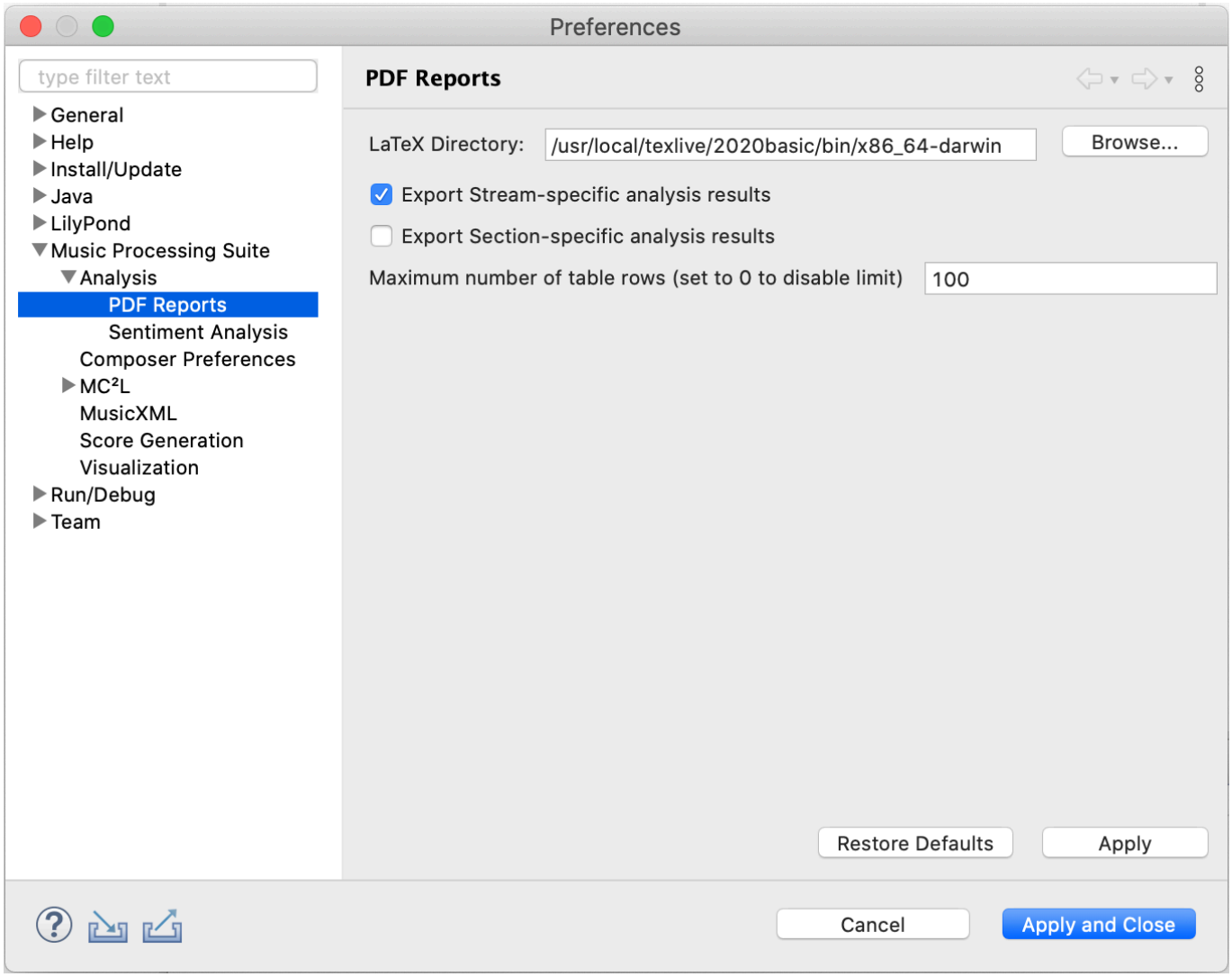


To create a nice PDF report visualizing the data, select the analysis output folder and click the button

in the toolbar. This will create a LaTeX file which will be compiled to a PDF file if a LaTeX environment is installed and configured as explained in section LaTeX Installation. The PDF file will contain statistical plots, diagrams and progression graphs. For a harmonic progression graph example, refer to section Analysis.

Options for analysis report generation are available here:

1. Open the MPS Preferences
    1. On Windows and Linux: In the menu bar, choose Window → Preferences
    2. On Mac: In the menu bar, choose Music Processing Suite (application menu) → Preferences
2. Go to Music Processing Suite → Analysis Reports

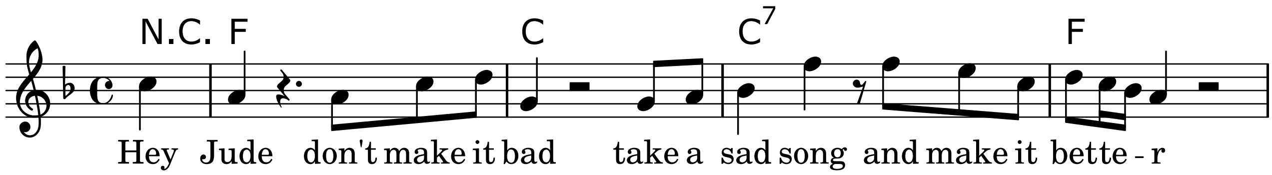


Context Layer Models

Most computer systems processing music data operate on sequential, time-based models that describe music as (one or multiple) sequences of notes and rests. This seems to be an adequate representation considering that the common form of music notation are scores, which primarily contain sequences of notes, rests and instructions how these are to be played. While sequential representations are suitable for many computer-aided music processing scenarios and also convenient for performing musicians, these are not sufficient for describing all aspects and relations of a musical composition from a composer's point of view.

Introductory Example

The following example demonstrates a model of the first four measures of the well-known Beatles song *Hey Jude*. The vocal part looks like this:



The corresponding context layer model is depicted below:

Stream 1	Instrument (1)	vocals																				
	Meter (5)	4/4 time	4/4 time					4/4 time				4/4 time						4/4 time				
	Tempo (1)	tempo 72																				
	Key (1)	F																				
	Harmony (5)		F					C				C7						F				
	Harmonic Rhythm (5)		1					1				1						1				
	Scale (1)	major																				
	Rhythm (21)	4	4	_4.	8	8	8	4	_2	8	8	4	4	_8	8	8	8	8	16	16	4	_2
	Degrees (18)	4	2		2	4	5	1		1	2	3	7		7	6	4	5	4	3	2	
	Pitches (18)	C5	A		A	C5	D5	G		G	A	Bb	F5		F5	E5	C5	D5	C5	Bb	A	
	Loudness (1)	loudness mf																				
	Lyrics (18)	Hey	Jude		don't	make	it	bad		take	a	sad	song		and	make	it	bet-	te-	-	r	
	Score Label (1)	Verse																				
	Time (Measures)	1	2					3				4						5				
	Time (Absolute)	-1 / 4	0					1				2						3				

Model Structure

Context layer models contain one or multiple *streams* which are comparable to voices or parts in scores. Each staff in a score is represented by at least one stream. A single part is divided into multiple streams if it in turn contains multiple voices (e.g. a fugue in four voices might be notated in two piano staves, however the context layer model representation will contain four streams).

Streams contain individual layers for various musical context dimensions, which are explained in the following sections. The layers in turn contain time-dependent context elements, each of which have a start time and a duration. Technical details of time representation are set out in section Time Model.

Instrumentation Context

The instrument layer specifies which instrument is used at which point of time in a stream. In the context layer model in the previous example, this context never changes and indicates a vocal part.

Metric Contexts

The meter layer provides the metric context of a musical stream. It contains time signatures, the start time and duration of which correspond to measures. Note that pieces may commence with an anacrusis (also known as pickup or upbeat) which implies a shortened initial measure. The measure numbers are shown in a timeline at the bottom of the context layer model visualization.

The current tempo is determined by an individual context layer. It usually contains elements specifying a constant tempo, as shown in the previous example. However, the tempo layer also supports gradual tempo changes to model *accelerando* and *decelerando*.

Harmonic Contexts

The current key context of streams is given by a correspondent context layer, which can change in the course of the composition.

Another harmonic context is given by context harmonies, which usually change more frequently than the key. In the previous example the key remains constant, while the context harmonies change in each measure.

Rhythmic Contexts

Rhythm is one of the most crucial dimensions of music. This is also reflected in context layer models: rhythm context layers are obligatory for each stream. The rhythmic dimension defines the durations and proportions of the notes or sound events produced by the stream.

Another rhythmic dimension is the harmonic rhythm, which specifies the durational proportions of context harmonies.

Pitch Contexts

The context layer model in the previous example also contains context layers regarding pitches, namely `Scale`, `Degrees` and `Pitches`.

Often pitches are derived from a contextually suitable scale, on which pitches can be addressed using scale degrees. In the example, pitches are derived from the F major scale (which in turn matches the key context) using zero-based scale degrees. For example, the degree 0 will resolve to F, 1 to G, 2 to A, 3 to Bb and so on.

The resulting absolute note names (including the octave) are visible in the `Pitches` context layer. If no octave is specified, the middle octave, which is encoded as octave with number 4 according to scientific pitch notation, is implied (see section Pitches for more details).

Loudness Contexts

Another musical context layer represents the progress of loudness throughout musical streams. It contains elements with static loudness instructions such as *piano* or *forte*. Gradual loudness progressions are also supported to model *crescendo* and *decrescendo*.

Another loudness-related context layer, which is not covered in the previous example, accommodates dynamic accents such as *sforzando* (notated as *sfz*, *sf* or *fz*), *sforzando* followed immediately by *piano* (*sfp*), *rinforzando* (*rfz*) or *fortepiano*, forte followed immediately by piano (*fp*).

Lyrics

Vocal streams can contain lyrics as an individual context. Using this layer, syllables can be assigned to individual notes as shown in the previous example.

Labels

Another context can be supplied in the form of labels for individuals parts of a composition. These could be, for example: *Verse*, *Chorus*, *Bridge*, *Solo* for popular music, or *Exposition*, *Development* and *Recapitulation* for a piece based on a sonata form.

Custom Contexts

MPS provides a number of default context layer types, most of which have been discussed in the previous sections. Yet, the number of layers is not fixed and the model was designed to be extensible in order to accommodate new context layers. For example, new context dimensions for fingering instructions, the spatial position of a musical stream or the emotional character of certain sections could be added. Refer to section Custom Contexts for detailed instructions on how to add custom context layers.

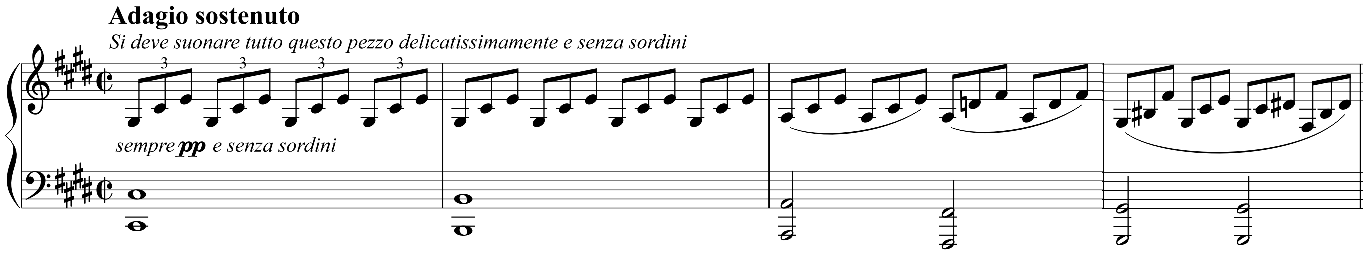
Time Model

Each context layer model has an internal timeline which by definition starts at `t=0` at the beginning of the first full measure. The earliest point of time can become negative in the case of anacruses at the beginning of the piece, as shown in the previous example, which starts at `t=-1/4` due to the upbeat.

Because the accuracy of floating point numbers is not sufficient for representing decimal fractions in all cases, MPS uses fractions (which internally store an integer numerator and denominator separately) for all points of time and durations.

Parallel Streams

To demonstrate the combination of multiple parts, a context layer model of the first four measures of Ludwig van Beethoven's *Piano Sonata No. 14 in C# minor* is presented. Compare the original score with the context layer model depicted below:



<table>
<tr><td rowspan="9">Stream 1</td><td>Instrument (1)</td><td colspan="48">piano</td></tr>
<tr><td>Meter (4)</td><td colspan="12">2/2 time</td><td colspan="12">2/2 time</td><td colspan="12">2/2 time</td><td colspan="12">2/2 time</td></tr>
<tr><td>Tonal Center (1)</td><td colspan="48">C#m</td></tr>
<tr><td>Harmony (8)</td><td colspan="12">C#m</td><td colspan="12">C#m/B</td><td colspan="6">A</td><td colspan="6">D/F#</td><td colspan="3">G#7</td><td colspan="3">C#m/G#</td><td colspan="3">G#sus4</td><td colspan="3">G#7</td></tr>
<tr><td>Harmonic Rhythm (8)</td><td colspan="12">1</td><td colspan="12">1</td><td colspan="6">2</td><td colspan="6">2</td><td colspan="3">4</td><td colspan="3">4</td><td colspan="3">4</td><td colspan="3">4</td></tr>
<tr><td>Scale (1)</td><td colspan="48">minor</td></tr>
<tr><td>Rhythm (48)</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td><td>12</td></tr>
<tr><td>Pitches (48)</td><td>G#3</td><td>C#</td><td>E</td><td>G#3</td><td>C#</td><td>E</td><td>G#3</td><td>C#</td><td>E</td><td>G#3</td><td>C#</td><td>E</td><td>G#3</td><td>C#</td><td>E</td><td>G#3</td><td>C#</td><td>E</td><td>G#3</td><td>C#</td><td>E</td><td>G#3</td><td>C#</td><td>E</td><td>A3</td><td>C#</td><td>E</td><td>A3</td><td>C#</td><td>E</td><td>A3</td><td>D</td><td>F#</td><td>A3</td><td>D</td><td>F#</td><td>G#3</td><td>B#3</td><td>F#</td><td>G#3</td><td>C#</td><td>E</td><td>G#3</td><td>C#</td><td>D#</td><td>F#3</td><td>B#3</td><td>D#</td></tr>
<tr><td>Loudness (1)</td><td colspan="48">loudness pp</td></tr>
<tr><td rowspan="9">Stream 2</td><td>Instrument (1)</td><td colspan="48">piano</td></tr>
<tr><td>Meter (4)</td><td colspan="12">2/2 time</td><td colspan="12">2/2 time</td><td colspan="12">2/2 time</td><td colspan="12">2/2 time</td></tr>
<tr><td>Tonal Center (1)</td><td colspan="48">C#m</td></tr>
<tr><td>Harmony (8)</td><td colspan="12">C#m</td><td colspan="12">C#m/B</td><td colspan="6">A</td><td colspan="6">D/F#</td><td colspan="3">G#7</td><td colspan="3">C#m/G#</td><td colspan="3">G#sus4</td><td colspan="3">G#7</td></tr>
<tr><td>Harmonic Rhythm (8)</td><td colspan="12">1</td><td colspan="12">1</td><td colspan="6">2</td><td colspan="6">2</td><td colspan="3">4</td><td colspan="3">4</td><td colspan="3">4</td><td colspan="3">4</td></tr>
<tr><td>Scale (1)</td><td colspan="48">minor</td></tr>
<tr><td>Rhythm (6)</td><td colspan="12">1</td><td colspan="12">1</td><td colspan="6">2</td><td colspan="6">2</td><td colspan="6">2</td><td colspan="6">2</td></tr>
<tr><td>Pitches (6)</td><td colspan="12">[C#3, C#2]</td><td colspan="12">[B2, B1]</td><td colspan="6">[A2, A1]</td><td colspan="6">[F#2, F#1]</td><td colspan="6">[G#2, G#1]</td><td colspan="6">[G#2, G#1]</td></tr>
<tr><td>Loudness (1)</td><td colspan="48">loudness pp</td></tr>
<tr><td></td><td>Time (Measures)</td><td colspan="12">1</td><td colspan="12">2</td><td colspan="12">3</td><td colspan="12">4</td></tr>
</table>

The model contains two streams, namely one for the arpeggios in the right hand and an individual stream for the arpeggio-based accompaniment of the left hand. Note that both streams contain the same information on instrument, meter, tempo key, harmony, harmonic rhythm, scale and loudness layers. However, the streams contain individual rhythm, degree and pitch layers.

The fact that streams can either share common information or specify individual information can be utilized to represent arbitrary musical constellations between the streams. For example, for multi-tonal compositions concurrent streams could contain different harmonic and tonal contexts. For compositions which do not rely on tonality, all context layers relating to tonality can be removed. In this way, the proposed model is suitable for the representation of a number of musical concepts and constellations, which can not in all cases be made visible in musical scores.

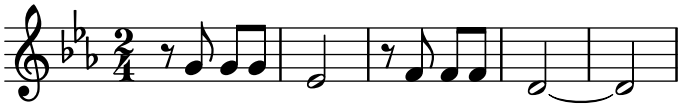
Composition Language and Context Tree Models

The composition model in MPS represents musical pieces by means of a tree-based structure containing musical context information. Besides musical contexts, the model contains so called *context modifiers*, *context generators* and *control structures*. All aforementioned elements are explained in the following sections.

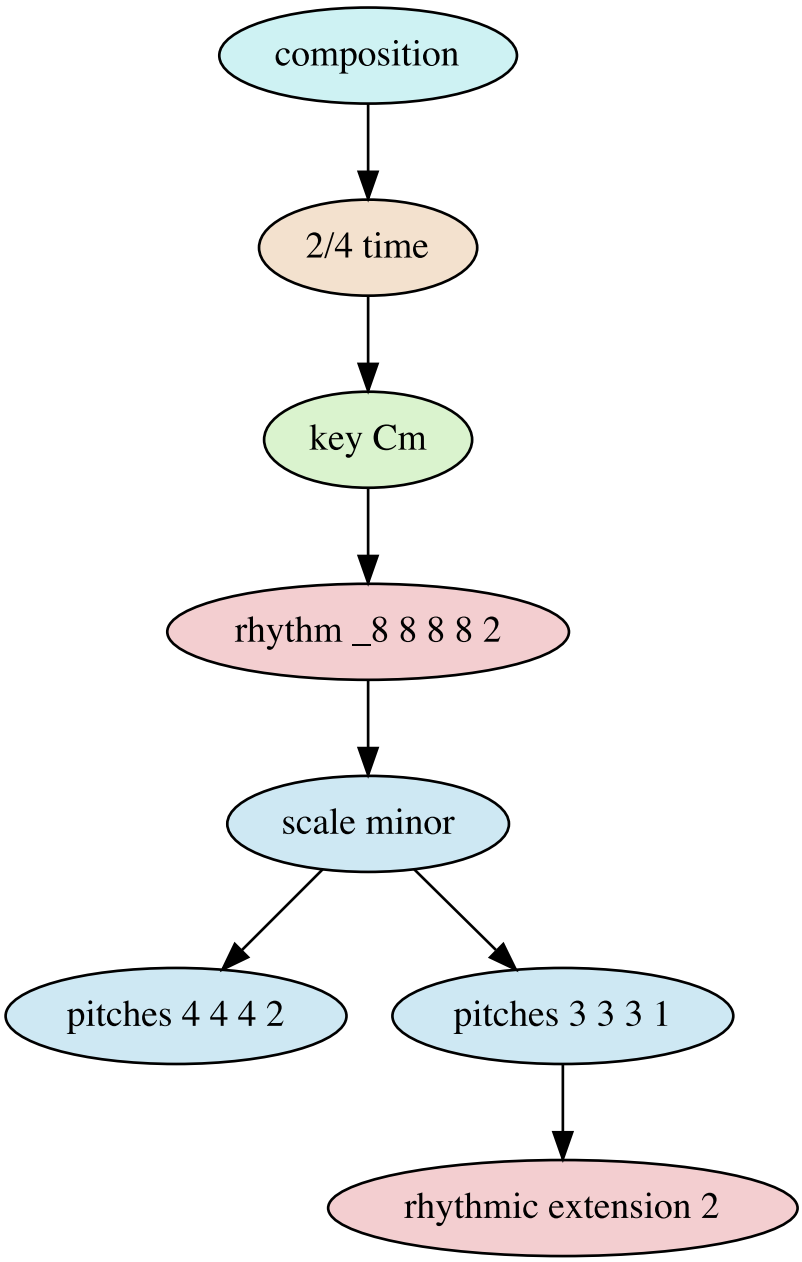
The model was developed in conjunction with a comprehensible domain-specific composition language, in which music can easily be notated. It is demonstrated in the following sections, how the composition language can be used to create composition models. The models can later be transformed to scores, lead sheets and a number of other representations, as explained in chapter Music Transformation and Visualization.

Introductory Example

As a first example, a model of Ludwig van Beethoven's world-famous *Symphony No. 5 in C Minor Op. 67* motif is presented. The score looks like this:



Consider the correspondent context model:



Each composition model defines a root node labeled `composition`. It contains a tree of context objects. In the previously shown model, the key of the composition (namely *C minor*) is defined by means of a `key` context. Below that, a metric context (a `2/4 time` signature) is defined. On the next layer, a `rhythm` is specified representing the famous rhythm of the motif, namely an eighth rest followed by three eighth notes and a half note. The syntax used to describe this rhythm is part of the composition language, which is also introduced in this chapter. Refer to section Rhythms for more details.

The pitches to be played are specified in terms of zero-based degrees on the minor `scale`, meaning that the number `0` represents the root note C, `1` the note D, `2` the note Eb and so on. Note that the context tree diverges below the `scale` node into two separate branches. This is interpreted as follows: First, all contexts between the `composition` and the `scale` node are combined with the left branch (namely the node `pitches 4 4 4 2`), and sequentially combined with the right branch (namely the node `pitches 3 3 3 1` and the `rhythmic extension`).

Note that both combined context sets contain the same rhythm, but different pitches. This is a pattern which is very frequently used in musical compositions: the same musical context (in this case a rhythm) is combined with a set of other musical contexts of another type (in this case pitches). The left branch effectively represents the first two measures of the composition. In this case, the pitches evaluate three times to G and once to Eb. In the right branch, which represents the rest of the motif, the pitches evaluate three times to F and once to D. The D, however, is rhythmically different from the Eb in the second measure, for its duration is two half notes instead of only one. This is only a minor modification compared to the original rhythm. In the composition model, it is not required to define a new rhythm. Instead, only the modification of the current rhythm must be specified, which is done with a so called *context modifier* named *rhythmic extension*, which doubles the duration of the last half note.

The context model can also be represented in terms of a simple text file in the corresponding domain-specific language. Compare the following syntactical representation with the previously introduced graphical model.

```
composition
{
    time 2/4, key Cm
    {
        rhythm _8 8 8 8 2
        {
            scale minor
            {
                pitches 4 4 4 2
                pitches 3 3 3 1
                {
                    rhythmicExtension duration 2
                }
            }
        }
    }
}
```

Key Concepts

The preceding example demonstrates a few key aspects of the model:

- Compositions can be expressed as combinations of musical aspects or contexts in various constellations.

- These contexts can be represented in form of a tree. When combining different tree branches sequentially, the tree can be interpreted as a musical composition.
- The tree structure is suitable for representing music in a compact form avoiding redundant information. In the example, the `rhythm` context is used twice but must be specified only once.
- If a musical structure (such as a rhythm) is modified in the course of a composition, the modification process can be specified rather than defining a new rhythm instance.

These concepts, among other mechanisms, are further elaborated in the following sections.

Hierarchical Structures

Musical compositions are usually to some extent organized and perceived in hierarchically arranged units. Compositions can generally have multiple hierarchical levels of organization.

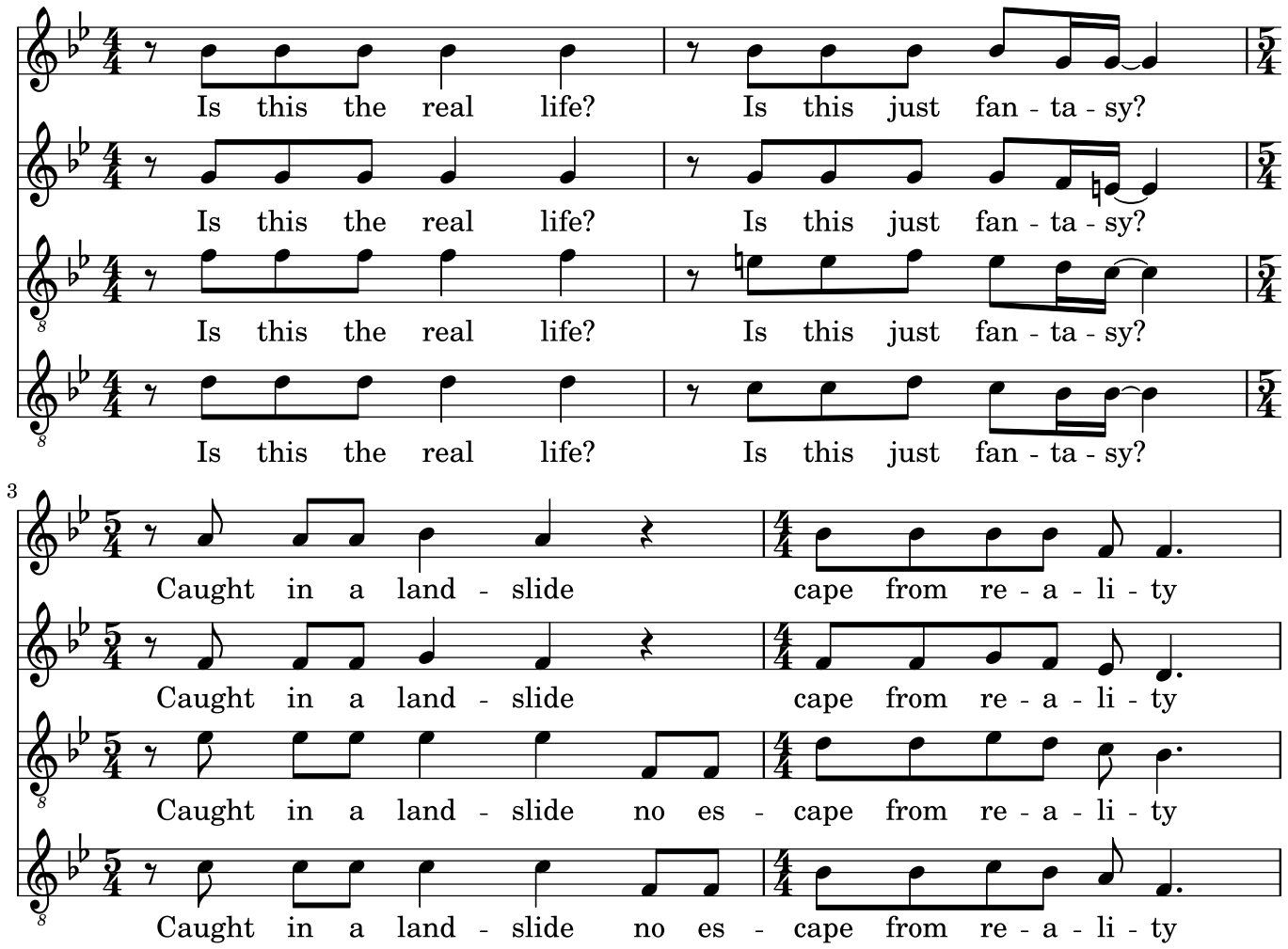
The hierarchical nature of context tree models is used for multiple purposes:

- Utilizing hierarchical structures is useful for **logical and graphical grouping** and for improving the clarity and readability of musical context tree models. The number of hierarchy levels in MPS context models is not limited, which allows to model musical context trees with arbitrary complexity.
- The hierarchy level of contexts in the tree models is also used to express the **scope** of the corresponding contexts. Generally, the higher a context is located in the tree, the more global is its impact on the musical composition.
- Furthermore, **hierarchical relations between individual contexts** can be modeled. For example, pitches can be put in a local harmonic context such as a chord or harmony, which in turn can be related to a local and/or global key.

Inheritance

A very effective way to avoid redundant information is to harness a technique commonly used in object-oriented software development called *inheritance*. It involves defining hierarchical dependencies between object types in order to utilize already existing properties and/or functionality from another object type.

In MPS, the principle of inheritance is applied to musical context tree models. This is illustrated using a musical example. Consider the following score of the beginning of Queen's *Bohemian Rhapsody*:



The score contains a some redundant information. For example, the parts are arranged homorhythmically, i.e. the rhythms of all four parts are exactly identical except for the end of the third measure. Also, the lyrics for all parts are exactly identical. In traditional scores, the composer or arranger has no other choice but to write the same rhythms and syllables all over again. In MPS context tree models however, the rhythm and the lyrics have to be specified only once and can be reused using various techniques. One of these techniques is inheritance, which is demonstrated in the following context tree model representing the first two measures of the piece:

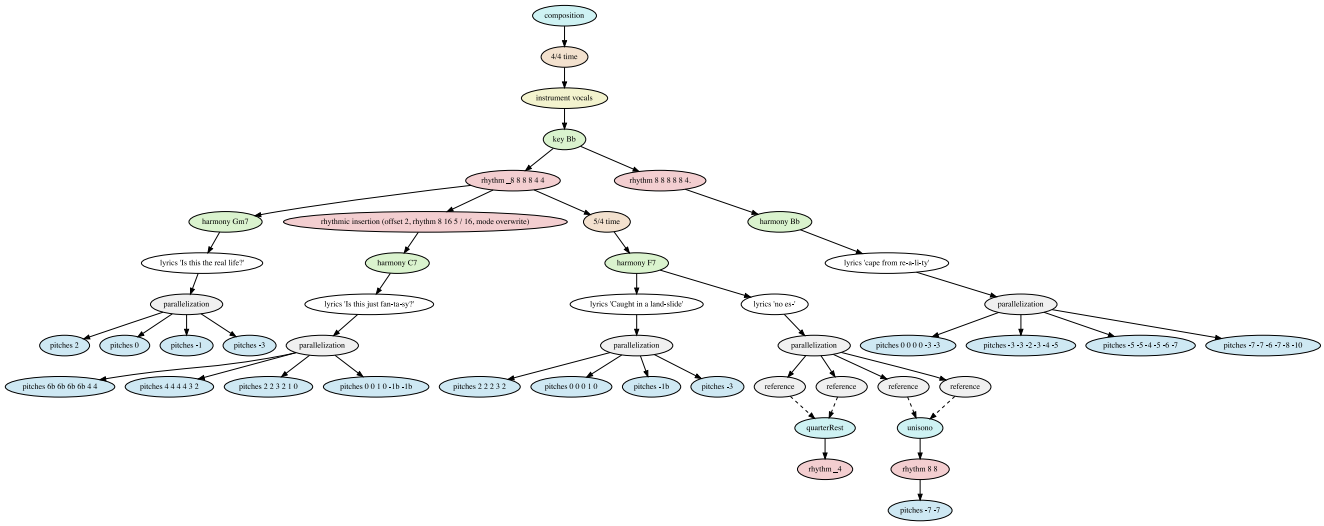
The inheritance hierarchy is made visible by arrows and by the positions of the context nodes. Arrows are interpreted as „all inherited contexts are passed on to the node in direction of the arrow". Inheriting nodes will normally drawn on the next hierarchy level which implies a lower position in the graph visualization. In this way, the instrument (vocals), the key (B flat major), the rhythm, the context harmony (G minor seventh), and the lyrics („Is this the real life?") are aggregated and passed on to the left `parallelization` node. It has four child nodes, which produce the four individual vocal parts of the first measure. They have different pitches, but have all the previously enumerated contexts in common. Using inheritance, all common contexts have to be specified only once, which is a major advantage of context tree models.

The same technique is used in the second measure, which inherits common instrument, key, base rhythm, context harmony and lyrics contexts. Note that further optimization methods are used in the model, which are explained in the following sections.

Polymorphism

Another model concept inspired by object-oriented programming is *polymorphism*, which allows to override (and also to extend) particular parts of inherited functionality. In context tree models,

this concept can be used to overwrite contextual information. To elaborate, another context tree model of *Bohemian Rhapsody* is shown in the following model, this time containing context information of the first four measures of the piece.

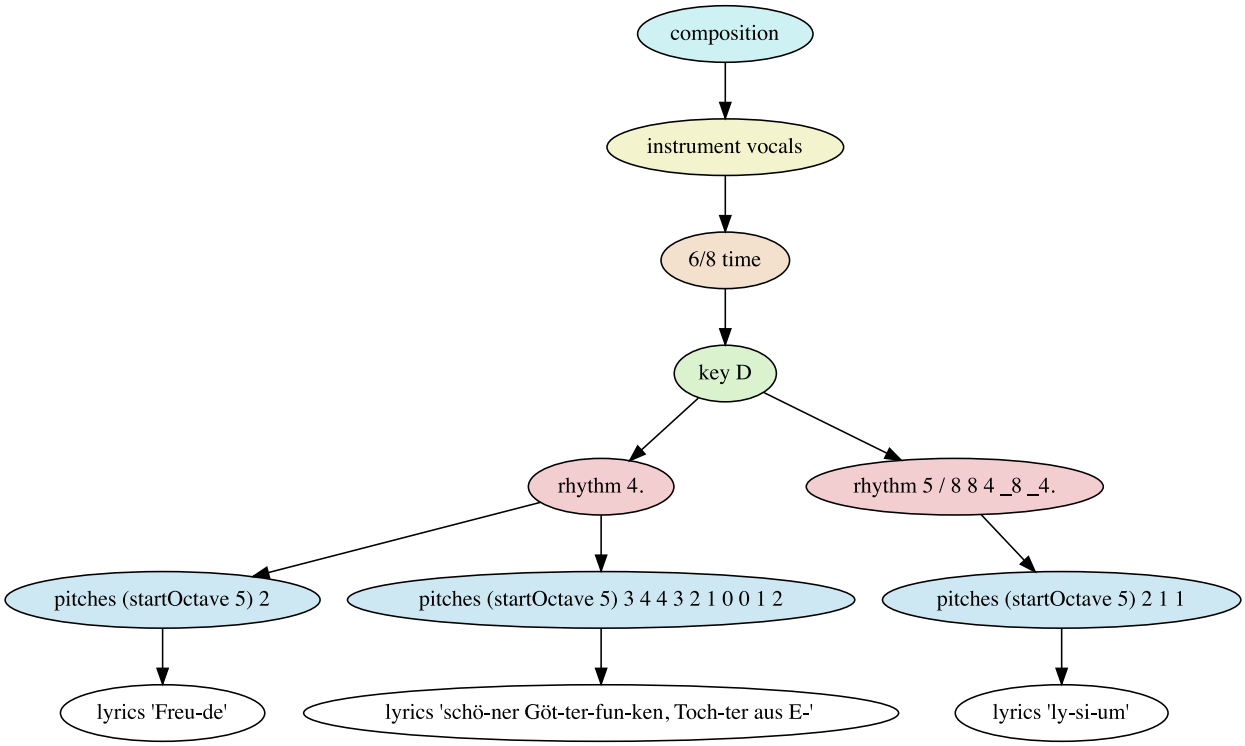


The time signature change in the third measure is modeled using a polymorphic construction. In the model, the `5/4` time signature context is positioned on a lower hierarchy level than the `4/4` time context at the top of the tree. The metric context `5/4` effectively overrides the `4/4` context temporarily (namely for one measure). After the subtree of the `5/4` measure is processed, the main `4/4` time signature becomes operative again. This technique can be applied to any musical context. For instance, temporary changes regarding meter, tempo, instruments, rhythms, pitches and harmonic contexts can be modeled.

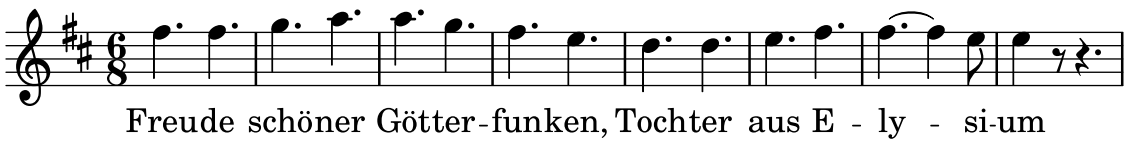
Note that this representation has additional value compared to a purely sequential representation. In the context tree model, it is directly visible that the `4/4` meter is of higher importance in the composition than the `5/4` meter. In fact, it becomes apparent that the `5/4` meter is only used in terms of a temporary „excursus” from the standard meter of the piece and is used in a subsidiary manner.

Auto Expansion

When combining contexts defining musical sequences (e.g. rhythms, pitches or lyric syllables), these sequences do not necessarily need to have the same length. If the number of available rhythm notes, pitches and syllables does not match, the system will automatically apply a so called *auto expansion*. The consequence is that shorter sequences will automatically be repeated until the longest sequence is consumed completely. Consider the following example:



It results in the following score:



The language representation looks like this:

```
composition
{
    instrument vocals, time 6/8, key D
    {
        rhythm 4.
        {
            pitches(startOctave 5) 2
            {
                lyrics "Freu-de"
            }
            pitches(startOctave 5) 3 4 4 3 2 1 0 0 1 2
            {
                lyrics "schö-ner Göt-ter-fun-ken, Toch-ter aus E-"
            }
```

```
        }
        rhythm 5/8 8 4 _8 _4.
        {
            pitches(startOctave 5) 2 1 1
            {
                lyrics "ly-si-um"
            }
        }
    }
}
```

In the previous example, musical sequences of different lengths are combined. In particular, in the leftmost subtree combines the rhythm `4.` (i.e. a dotted quarter note) with a pitch on the third scale degree (zero-based, i.e. 2) and the two lyric syllables `Freu-de`. While the rhythm and the pitch sequence only contain one element, the lyric sequence contains two syllables. The system automatically wraps and repeats the rhythm and the pitches until both syllables are processed. In sum, this results in two dotted quarter notes, both with the same pitch, but with different syllables.

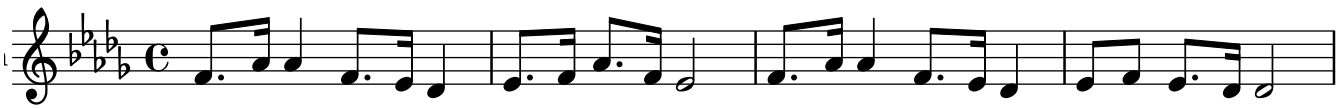
Auto expansion was also used in the previous *Bohemian Rhapsody* example, in which the rhythm `_8 8 8 8 4 4` is combined with the lyrics „Is this the real life?" and multiple pitch contexts for individual parts, namely `pitches 2`, `pitches 0`, `pitches -1` and `pitches -3`. The rhythm contains six rhythmic notes, the first of which is a rest, leaving five assignable notes for syllables and pitches. The lyrics contain exactly five matching syllables. The pitch contexts, however, contain only one pitch each. Therefore, the pitches are repeated until the rhythm and the lyrics are consumed completely.

Using auto expansion, redundant musical sequences can be represented in an effective way, providing yet a useful compression method for context tree models.

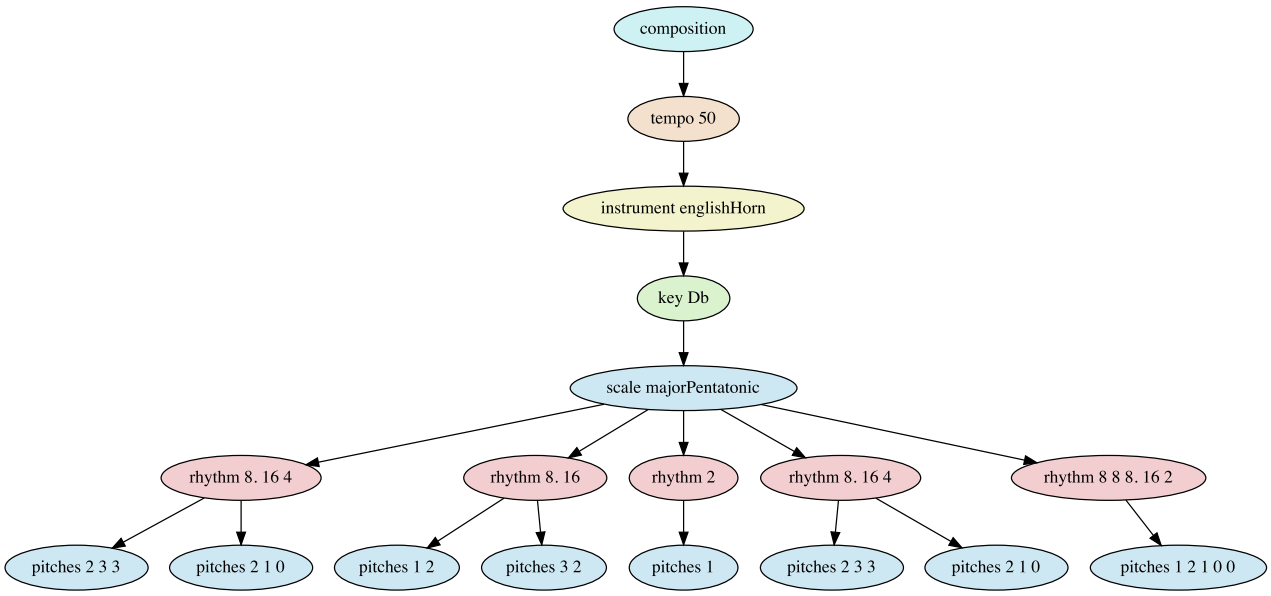
Modularization using Fragments

Another technique to avoid redundant information in context tree models is modularization. To this means, arbitrary subtrees can be extracted into so called *fragments*. These are named subtrees which can be referenced from other places in the model. If subtrees occur multiple times in a model, they only have to be defined once in a fragment in order to be referenced at any place they are required.

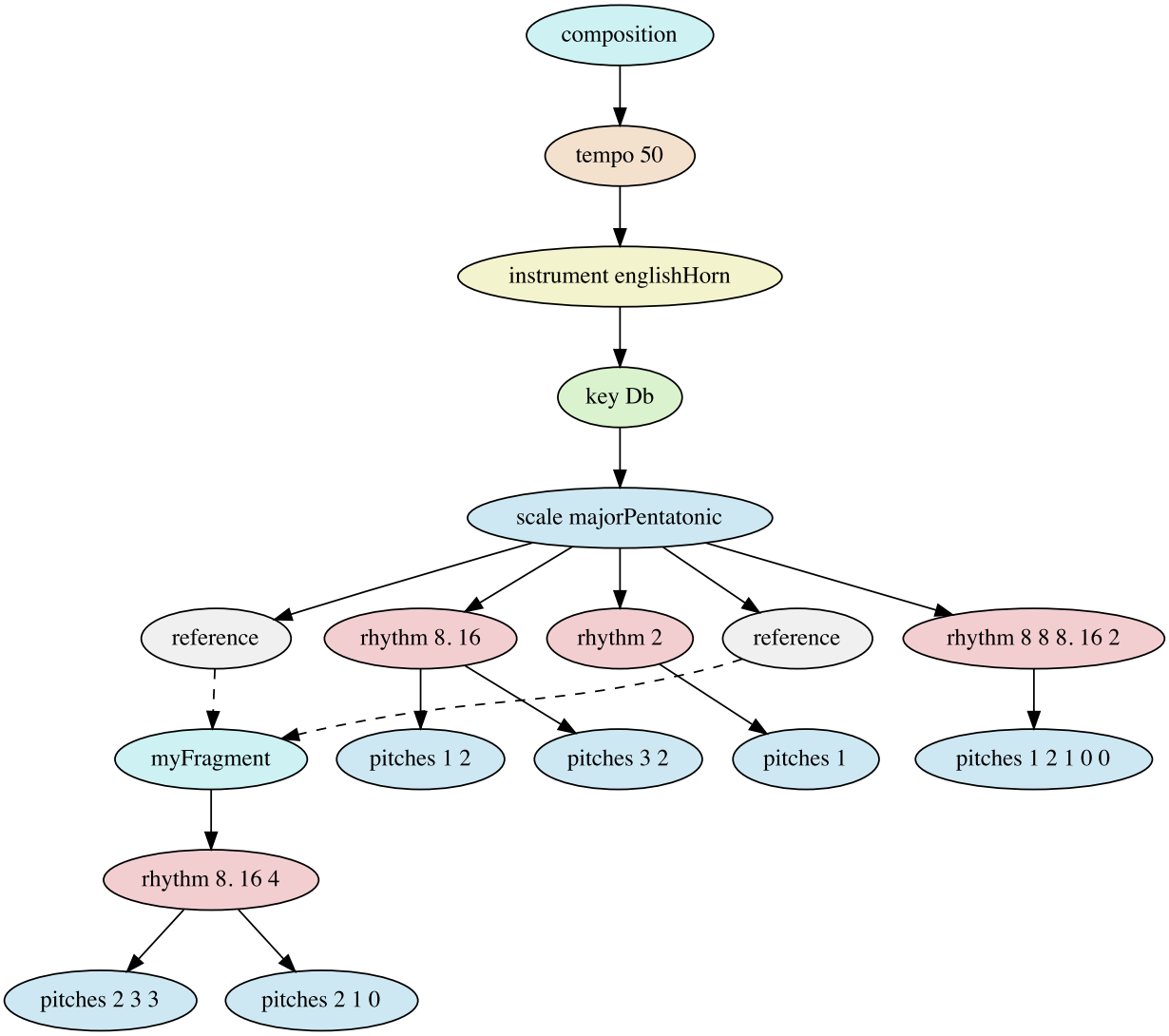
As an example, consider the English horn theme of Antonin Dvorak's *Symphony No. 9 in E minor, „From the New World", Op. 95, B. 178*:



One possible context model for this score looks like this:



However, this model can be further optimized, as it contains some redundant information. Compare measures 1 and 3, which are exactly identical. The corresponding subtrees, i.e. the subtrees originating at the rhythms `8. 16 4`, can be extracted to a fragment and referenced twice:



Contexts

Rhythms

Rhythm is one of the most central aspects in music. In the composition model, rhythms are represented as an individual context dimension and can be expressed using the corresponding domain-specific language. The syntax is very simple, yet powerful. Consider the following example:

```
rhythm _8 8 8 8 2
```

It yields the motif rhythm of Beethoven's famous Symphony No. 5 in C Minor, Op. 67:

Of course, more complex rhythms can be defined using the language. Refer to the following table for a detailed explanation of note and rest duration syntax variants.

Syntax	Example	Result	Description
n (Integer Literal)	`2 4 8 16`		Integer literals are interpreted as reciprocal duration, e.g. 4 represents a quarter note, 8 an eighth note etc.
_ (Underscore Prefix)	`_2 _4 _8 _16`		Prefix to indicate that the following duration is to be interpreted as a rest duration.
n! (Integer Literal with ! Suffix)	`2!`		Integer literals followed by an exclamation mark are **not** interpreted as reciprocal duration but as literal duration, e.g. `2!` specifies a duration of two whole notes.
. (Dot Suffix)	`8. 16 8.. 32`		Dots are used as suffixes to extend the preceding note or rest duration with a factor of `1.5`. Multiple dots can be used in a row.
n/m (Fraction with Integer Numerator and Denominator)	`5/4`		Fractional note or rest duration, normally used if the duration can not be expressed as a canonical duration using simple fractions of two and dots.
~ (Tilde Suffix)	`1~ 4`		Suffix used to indicate that the current note is rhythmically tied to the following note.
(n/m: <durations>) (Tuplet)	`(3/2: 8 8 8)`		Specifies a tuplet in which n notes are played in the original

duration of `m` notes. The adjacent example produces an eighth triplet. To compute the resulting durations, the original durations have to be multiplied with the fraction `m/n`. For example, in the case of the triplet, the note durations are multiplied with `2/3`, yielding durations of `1/8 * 2/3 = 1/12` for each of the eighth triplet notes.

Examples

Syntax	Resulting Rhythm	Description
`_8 8 8 8 2`		Ludwig van Beethoven, *Symphony No. 5 in C Minor Op. 67*, Motif Rhythm
`4. 8 8 8 _4`		George Frideric Handel, *Hallelujah Chorus from Messiah, HWV 56*, Motif Rhythm
`2 4 4 4. 16 16 4 _4`		Wolfgang Amadeus Mozart, *Piano Sonata No. 16 in C major, K. 545*, Opening Theme Rhythm
`8 8 8 _8 8 8 _8 8 _8 8 8 _8`		Steve Reich, *Clapping Music*, Rhythmic Motif
`_8 8 8 8 8. 32 32 8 8 8~ 16 16 16 16 16`		J. S. Bach, *Fugue in C Major, BWV 846*, Subject
`4 _8 (3/2: 16 16 16) 4 _8 (3/2: 16 16 16) 4`		Wolfgang Amadeus Mozart, *Symphony No. 41 in C major, K. 551*, Opening Theme Rhythm

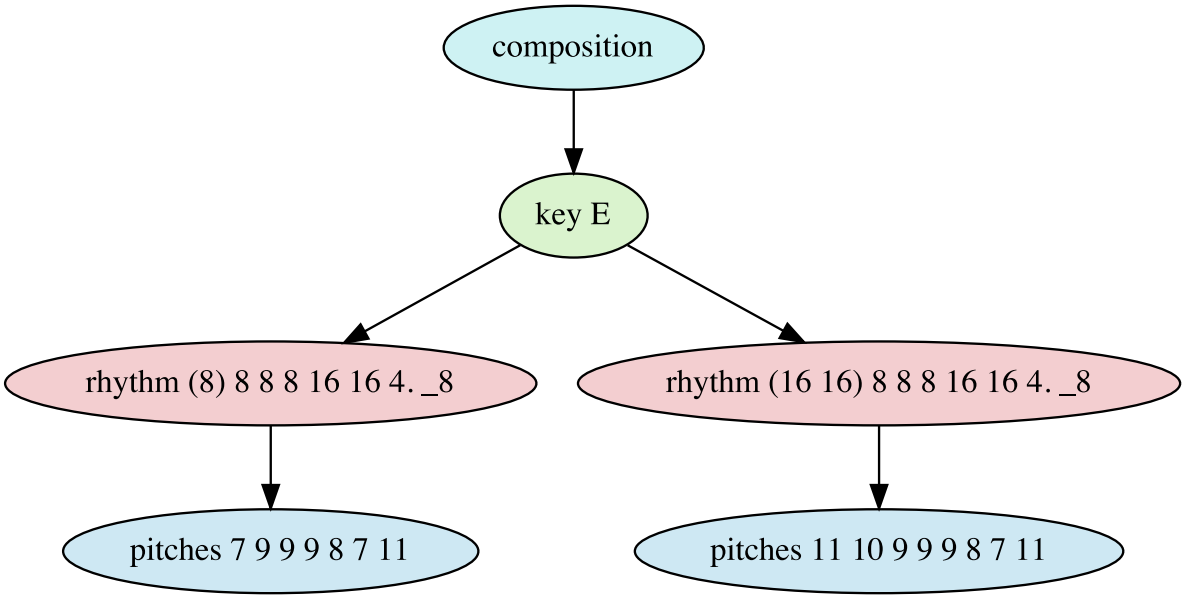
Anacruses

Some musical phrases do not start directly on a metrically strong beat, but are preceded by one or more notes, which are referred to as *anacrusis*, also known as *pickup* or *upbeat*. Often this happens at the very beginning of a piece, yet also phrases in the middle of compositions can be initiated using pickup beats.

To indicate anacruses in MPS, the pickup beats simply are enclosed in parentheses. For example, the following code specifies a rhythm with an eighth note pickup beat:

```
rhythm (8) 8 8 8 16 16 4. _8
```

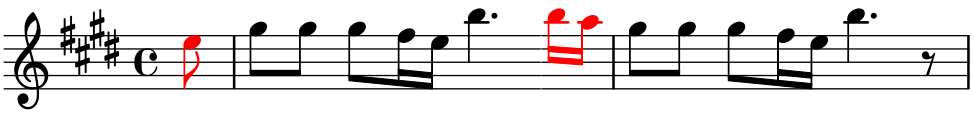
Consider the following model of Vivaldi's *Concerto No. 1 in E major, Op. 8, RV 269* known as *Spring* from the *Four Seasons*, in which anacruses at the beginning and in the middle of a phrase are specified.



The equivalent syntactic representation of the model is:

```
composition
{
    key E
    {
        rhythm (8) 8 8 8 16 16 4. _8
        {
            pitches 7 9 9 9 8 7 11
        }
        rhythm (16 16) 8 8 8 16 16 4. _8
        {
            pitches 11 10 9 9 9 8 7 11
        }
    }
}
```

The result of this model is the following score (with anacruses at the beginning and before the second full measure marked in red):

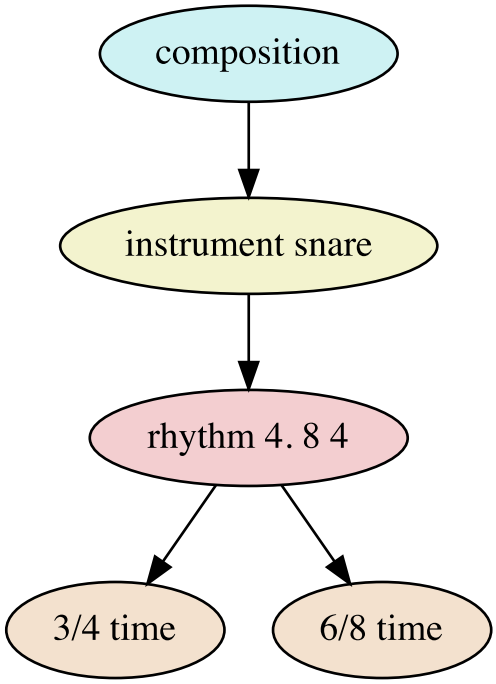


Time Signatures

Time signatures are defined using the `time` keyword in combination with a fraction, for example:

```
time 3/4
```

Depending on the metric context, the very same rhythm can have different musical meanings. This is illustrated in the following example:

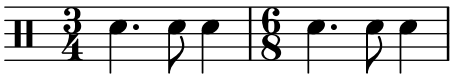


The corresponding source code looks like this:

```
composition
{
    instrument snare
    {
```

```
        rhythm 4. 8 4
        {
            time 3/4
            time 6/8
        }
    }
}
```

When compiling the model, the following score results:



It demonstrates that the very same rhythm can have different musical meanings depending on the metric context.

Tempo

Tempo is an individual context dimension which can be changed independently from time signatures. The tempo is specified in beats per minute (BPM). Example:

```
tempo 100
```

By default, the BPM specification defines the temporal distance of quarter notes. It is also possible to define other note durations to which the BPM specification relates. To specify the tempo for eighth notes, for example, the following syntax is used:

```
tempo 80 noteDuration 8
```

The note duration syntax is the same as described in section Rhythms.

It is also possible to define gradual tempo changes, as demonstrated in the following example:

```
tempo 80 -> 110 noteDuration 8
```

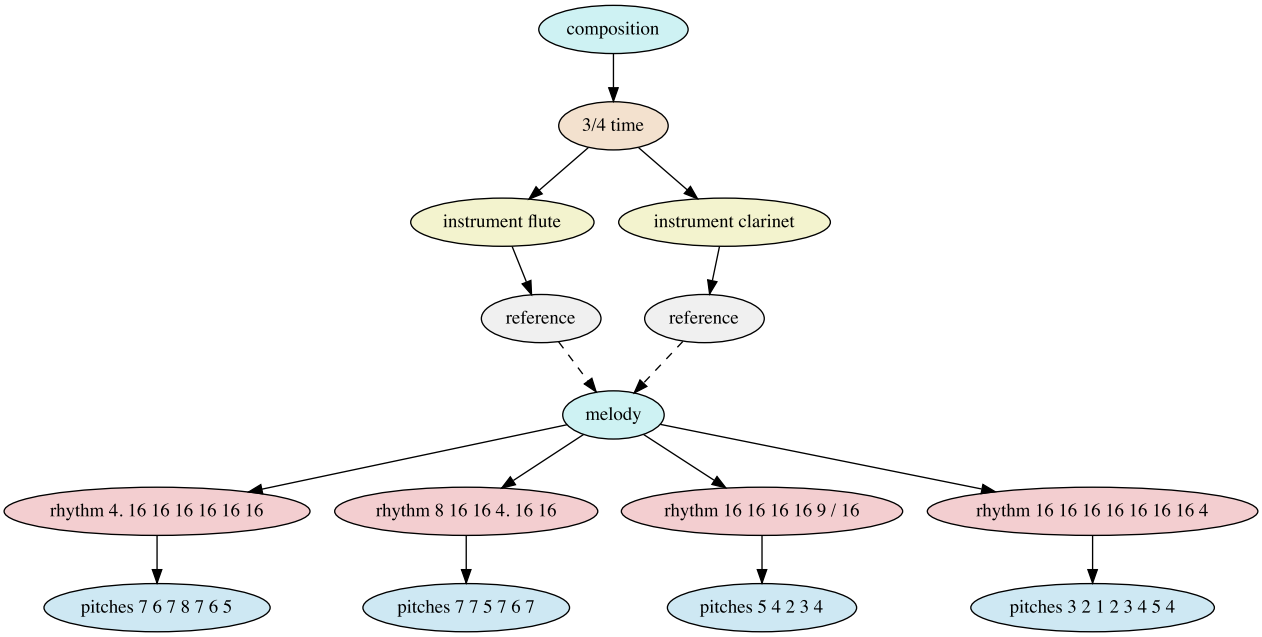
Instruments

The `instrument` context defines by which instrument the musical material in the respective part of the model is played. Syntactically, this context is defined by the `instrument` keyword followed by an instrument identifier, such as:

```
instrument guitar
```

Refer to the following section Available Instruments for a complete list of predefined instruments.

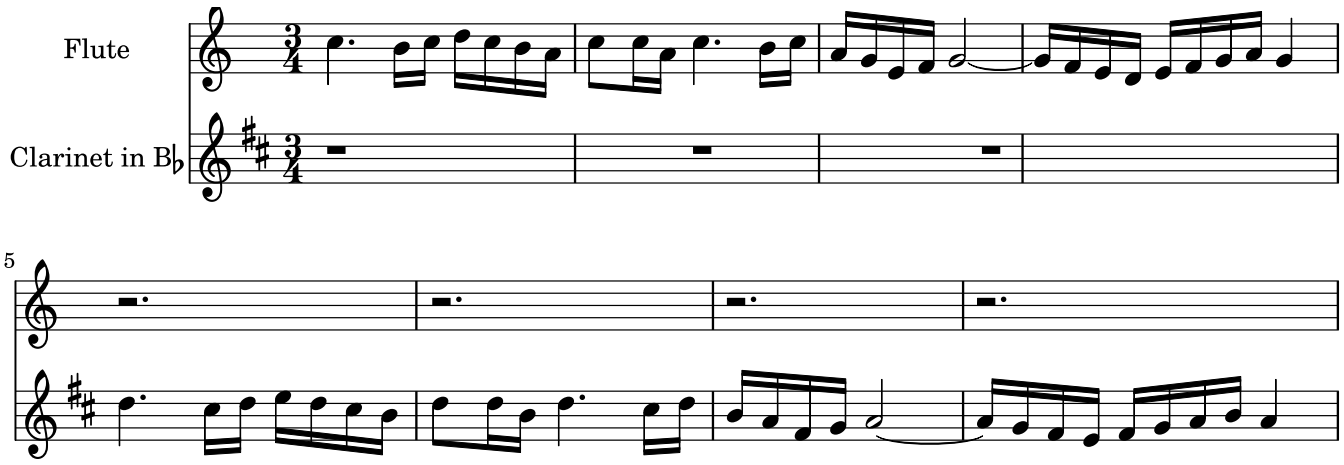
The following context model represents an excerpt of the famous *Boléro* by Maurice Ravel, in which a part of the melody is sequentially played by the flute and the clarinet:



Syntactically, this can be expressed as:

```
composition
{
    time 3/4
    {
        instrument flute
        {
            fragmentRef melody
        }
        instrument clarinet
        {
            fragmentRef melody
        }
    }
}

fragment melody
{
    rhythm 4. 16 16 16 16 16 16, pitches 7 6 7 8 7 6 5
    rhythm 8 16 16 4. 16 16, pitches 7 7 5 7 6 7
    rhythm 16 16 16 16 9/16, pitches 5 4 2 3 4
    rhythm 16 16 16 16 16 16 16 4, pitches 3 2 1 2 3 4 5 4
}
```

The score looks like this. Note that the clarinet is notated in B flat.



For a version of this model in which the melody is played simultaneously, refer to section Parallelizations.

Available Instruments

The MPS library contains a number of predefined instruments, which are listed and described in the following sections.

Instruments with Variable Pitches

The following instruments are generally playable in different pitches depending on their compass.

Identifier	Name	Description
`accordion`	Accordion	
`acousticBass`	Acoustic Bass	Transposing Instrument: Sounds one octave lower than notated, using bass clef for notation by default
`acousticGuitar`	Acoustic Guitar	Transposing Instrument: Sounds one octave higher than notated
`acousticSteelGuitar`	Acoustic Steel Guitar	Transposing Instrument: Sounds one octave higher than notated
`altoSax`	Alto Saxophone	
`altoSaxInEb`	Alto Saxophone in Eb	Transposing Instrument: Sounds a major sixth lower than notated
`banjo`	Banjo	
`bass`	Bass Guitar	Transposing Instrument: Sounds one octave lower than

		notated, using bass clef for notation by default
`bassClarinet`	Bass Clarinet	
`bassClarinetInBb`	Bass Clarinet in Bb	Transposing Instrument: Sounds a major ninth lower than notated
`bassoon`	Bassoon	
`bassPicked`	Picked Bass Guitar	Transposing Instrument: Sounds one octave lower than notated, using bass clef for notation by default
`baritoneSax`	Baritone Saxophone	
`baritoneSaxInEb`	Baritone Saxophone in Eb	Transposing Instrument: Sounds a major thirteenth lower than notated
`celesta`	Celesta	Transposing Instrument: Sounds one octave higher than notated
`cello`	Cello	
`clarinet`	Clarinet	
`clarinetInA`	Clarinet in A	Transposing Instrument: Sounds a minor third lower than notated
`clarinetInBb`	Clarinet in Bb	Transposing Instrument: Sounds a major second lower than notated
`clarinetInEb`	Clarinet in Eb	Transposing Instrument: Sounds a minor third higher than notated
`contrabassoon`	Contrabassoon	Transposing Instrument: Sounds one octave lower than notated, using bass clef for notation by default
`doubleBass`	Double Bass	Transposing Instrument: Sounds one octave lower than notated, using bass clef for notation by default
`drawbarOrgan`	Drawbar Organ	
`electricGuitar`	Electric Guitar	Transposing Instrument: Sounds one octave higher than notated

`electricGuitarDistorted`	Distorted Electric Guitar	Transposing Instrument: Sounds one octave higher than notated
`electricGuitarJazz`	Electric Jazz Guitar	Transposing Instrument: Sounds one octave higher than notated
`electricGuitarMuted`	Muted Electric Guitar	Transposing Instrument: Sounds one octave higher than notated
`electricGuitarOverdriven`	Overdriven Electric Guitar	Transposing Instrument: Sounds one octave higher than notated
`electricPiano`	Electric Piano	
`englishHorn`	English Horn	
`englishHornInF`	English Horn in F	Transposing Instrument: Sounds a perfect fifth lower than notated
`flute`	Flute	
`frenshHorn`	Frensh Horn	
`glockenspiel`	Glockenspiel	Transposing Instrument: Sounds two octaves higher than notated
`harmonica`	Harmonica	
`harp`	Orchestral Harp	
`harpsichord`	Harpsichord	
`horn`	Horn	Synonymously used for Frensh Horn
`hornInF`	Horn in F	Transposing Instrument: Sounds a perfect fifth lower than notated
`oboe`	Oboe	
`organ`	Church Organ	
`pad`	Pad (New Age)	
`pad2`	Pad (Warm)	
`panFlute`	Pan Flute	
`percussiveOrgan`	Percussive Organ	
`piano`	Piano	Used as default if no instrument is specified

piccolo	Piccolo	Transposing Instrument: Sounds one octave higher than notated
reedOrgan	Reed Organ	
recorder	Soprano Recorder	Transposing Instrument: Sounds one octave higher than notated
recorderAlto	Alto Recorder	
recorderBass	Bass Recorder	Transposing Instrument: Sounds one octave higher than notated, using bass clef for notation by default
recorderContrabass	Contrabass Recorder	Notated using bass clef by default
recorderGarklein	Garklein Recorder	Transposing Instrument: Sounds two octaves higher than notated
recorderGreatBass	Great Bass Recorder	Transposing Instrument: Sounds one octave higher than notated, using bass clef for notation by default
recorderSopranino	Sopranino Recorder	Transposing Instrument: Sounds one octave higher than notated
recorderSubGreatBass	Sub-Great Bass Recorder	Transposing Instrument: Sounds one octave lower than notated, using bass clef for notation by default
recorderSubContrabass	Sub-Contrabass Recorder	Transposing Instrument: Sounds one octave lower than notated, using bass clef for notation by default
recorderTenor	Tenor Recorder	
rockOrgan	Rock Organ	
sitar	Sitar	
sopranoSax	Soprano Saxophone	
tenorSax	Tenor Saxophone	
tenorSaxInBb	Tenor Saxophone in Bb	Transposing Instrument: Sounds a major ninth lower than notated
timpani	Timpani	

`trombone`	Trombone	
`trumpet`	Trumpet	
`trumpetInD`	Trumpet in D	Transposing Instrument: Sounds a major second higher than notated
`trumpetInBb`	Trumpet in Bb	Transposing Instrument: Sounds a major second lower than notated
`trumpetMuted`	Muted Trumpet	
`tuba`	Tuba	
`vibraphone`	Vibraphone	
`viola`	Viola	
`violin`	Violin	
`vocals`	Vocals	
`xylophone`	Xylophone	

Untuned Percussion Instruments

The following instruments can generally not be played in different pitches:

Identifier	Name	Description
`agogoHigh`	High Agogo	
`agogoLow`	Low Agogo	
`bassDrum`	Bass Drum	
`bassDrum2`	Bass Drum 2	Alternative Bass Drum
`bongoHigh`	High Bongo	
`bongoLow`	Low Bongo	
`cabasa`	Cabasa	
`china`	China Cymbal	
`claves`	Claves	
`congaHigh`	High Conga	
`congaLow`	Low Conga	
`congaHighMuted`	Muted High Conga	
`cowbell`	Cowbell	
`crash`	Crash Cymbal	
`crash2`	Crash Cymbal 2	
`cuica`	Cuica	
`cuicaMuted`	Muted Cuica	
`guiroShort`	Short Guiro	

`guiroLong`	Long Guiro	
`handClaps`	Hand Claps	
`hiHatClosed`	Closed Hi-Hat	
`hiHatPedal`	Pedal Hi-Hat	Hi-Hat played via pedal
`hiHatOpen`	Open Hi-Hat	
`maracas`	Maracas	
`ride`	Ride Cymbal	
`ride2`	Ride Cymbal 2	
`rideBell`	Ride Cymbal Bell	
`sideStick`	Side Stick	
`snare`	Snare Drum	
`snareElectric`	Electric Snare Drum	
`splash`	Splash Cymbal	
`tambourine`	Tambourine	
`timbaleHigh`	High Timbale	
`timbaleLow`	Low Timbale	
`tomHigh`	High Tom	
`tomHighMid`	High-Mid Tom	
`tomLowMid`	Low-Mid Tom	
`tomLow`	Low Tom	
`tomFloorHigh`	High Floor Tom	
`tomFloorLow`	Low Floor Tom	
`triangle`	Triangle	
`triangleMuted`	Muted Triangle	
`vibraslap`	Vibraslap	
`whistleShort`	Short Whistle	
`whistleLong`	Long Whistle	
`woodBlockHigh`	High Wood Block	
`woodBlockLow`	Low Wood Block	

Instrument Definitions

If additional instruments are required, users are able to define custom instruments by providing instrument definitions. Consider the following definition of an acoustic bass guitar:

```
instrumentDef acousticBass
{
    pitchRange [23..67]
    maxSimultaneousNotes 4
```

```
    scoreLabel "Bass"
    lilyPondInstrumentName "acoustic bass"
    defaultClef bass
    defaultOctave 2
}
```

The `instrumentDef` keyword is followed by an instrument identifier, which is used to reference the instrument definition in instrument contexts. For example, the acoustic bass can be referenced using the following syntax:

```
instrument acousticBass
```

Enclosed in curly braces, optional instrument parameters follow. Refer to the following table for descriptions of the individual parameters.

Parameter	Description
`type`	Either `percussion` in case of percussion instruments or `synth` for synthesizers used for electronic / electroacoustic music. Omit this parameter to create an instrument of `default` type which is playable in different pitches.
`pitchRange`	Specifies the compass of the instrument in terms of MIDI notes in the syntax `[lowest note..highest note]`.
`maxSimultaneousNotes`	Specifies the maximum number of notes which can be played simultaneously.
`scoreLabel`	Name of the instrument which is displayed at the beginning of staves in scores.
`lilyPondInstrumentName`	Instrument name used for assigning a MIDI instrument when exporting LilyPond scores. See LilyPond documentation.
`defaultClef`	Default clef to use in scores. Currently supported clef names are: `treble`, `alto`, `tenor` and `bass`.
`defaultOctave`	Default MIDI octave to use if none is specified in composition models.

Pitches

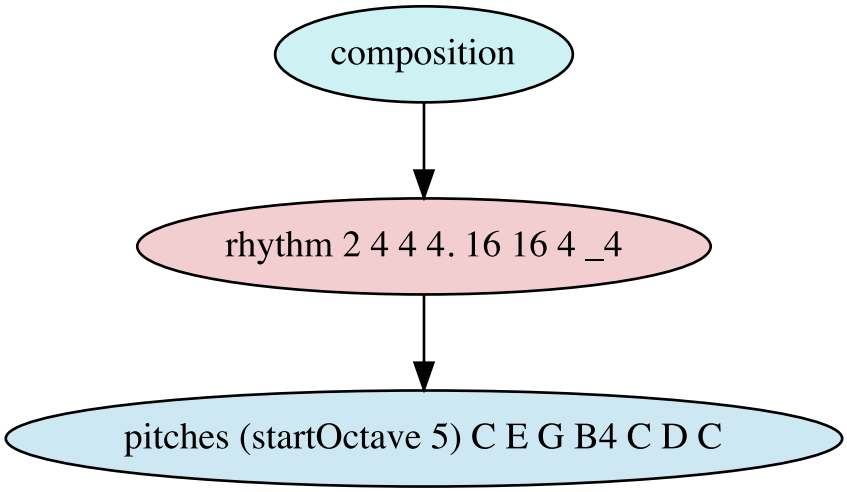
MPS supports multiple types of pitch specifications. One possibility is to specify absolute pitches and octave numbers such as `Ab5`. Refer to the following table for a specification of octave numbers:

MIDI Note Numbers	Octave Number	Octave Name
0-11	-1	Double Contra
12-23	0	Sub Contra

24-35	1	Contra
36-47	2	Great
48-59	3	Small
60-71	4	One-line
72-83	5	Two-line
84-95	6	Three-line
96-107	7	Four-line
108-119	8	Five-line
120-127	9	Six-line

MIDI note numbers were specifically not chosen as pitch unit, since enharmonic differentiations are not possible. For instance, the pitch names `G#` and `Ab` correspond to same key on piano (assuming the same octave number is specified), but have different musical meanings relating to the harmonic context (see section Harmonic Contexts for more details). For this reason, harmonically significant pitch names are used. Alternatively, pitches may be given in terms of degrees on a scale, which is elaborated in section Scales.

The first two measures of W.A. Mozart's *Piano Sonata No. 16 in C major, K. 545*, also known as *Sonata Facile*, are used as an example for pitch specifications using pitch names and octave numbers. Consider the following model:

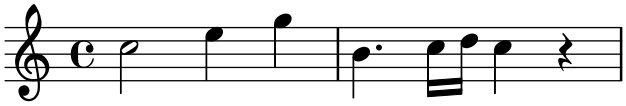


The correspondent syntactical representation is:

```
composition
{
```

```
    rhythm 2 4 4 4. 16 16 4 _4
    {
        pitches (startOctave 5) C E G B_4 C D C
    }
}
```

The resulting score is:



Various syntax alternatives for pitch specifications are listed in the following table:

Syntax	**Description**
`<note name>`	Used for specifying pitches explicitly, e.g. `D`, `C#` or `Eb`.
`<integer number>`	Used for pitch specifications based on scale degrees. Refer to section Scales for more details.
# (suffix)	Raises the previously specified pitch or scale degree by one semitone.
b (suffix)	Lowers the previously specified pitch or scale degree by one semitone.
`[<pitches>]`	Square brackets are used to specify chords. For example, a D major chord can be written as `[D F# A]`.
<code>@</code> (prefix)	Indicates the usage of an expression to dynamically compute a pitch or scale degree. For example, the expression <code>@getRootNote()</code> evaluates to the root note of the current context harmony. Refer to section Expressions for more details.

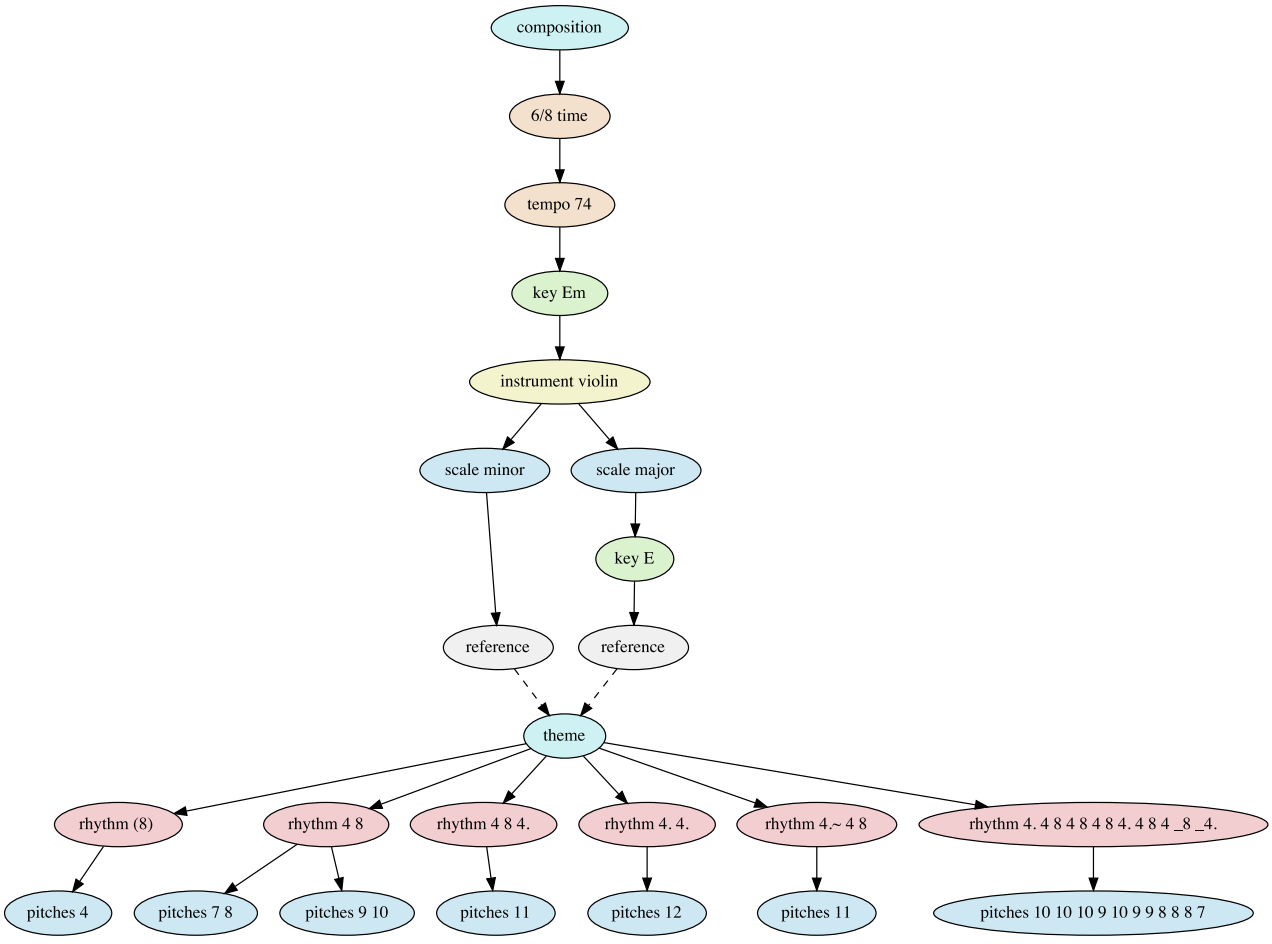
Additional parameters may be used when specifying pitches, which are explained in the following table. If these parameters are used, they have to be syntactically enclosed in parentheses before pitches or scale degrees are specified, as demonstrated in the previous listing with the `startOctave` parameter.

Parameter	**Description**
`startOctave`	Specifies the octave to use if no octaves are defined explicitly.
`findNearestOctave`	If set to `true`, the system will change the octave automatically if it implies a smaller semitone distance to the previous note. Example: in the pitch sequence `A C` the

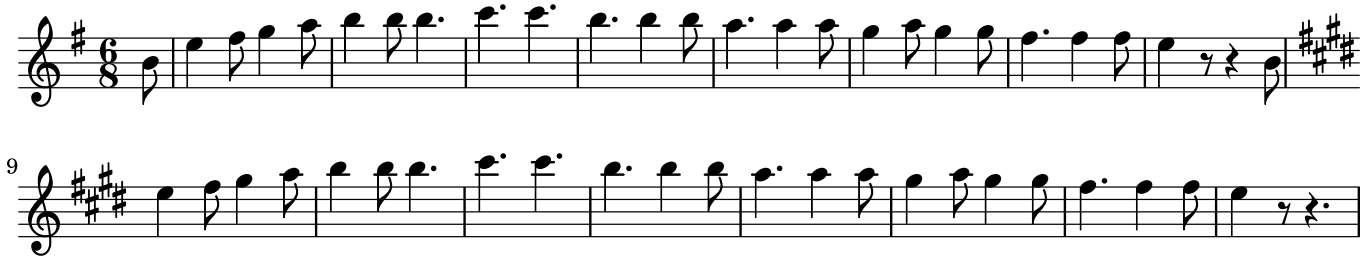
	system would start in the default octave yielding `A4`. With `findNearestOctave` enabled, the next pitch would be `C5` because it has a smaller distance to `A4` than `C4`.
`relative to`	Specifies which harmonic context is to be used to determine the context scale and its tonic. Possible values are `key` and `harmony`. Refer to sections Scales and Harmonic Contexts for more details.

Scales

An alternative to specifying absolute pitches is referring to pitches in terms of scale degrees. Consider the following example, which shows a context tree model of Bed#ich Smetana's *Moldau Theme*. The pitches in the model are defined in terms of zero-based scale degrees. The theme is referenced twice in the model: once from a `minor` scale context and once from a `major` scale context.



The compilation of the model results in the following score:



The model can be represented syntactically as follows:

```
composition
{
    time 6/8, key Em, instrument violin
    {
        scale minor
        {
            fragmentRef theme
        }
        scale major, key E
        {
            fragmentRef theme
        }
    }
}

fragment theme
{
    rhythm (8)
    {
        pitches 4
    }
    rhythm 4 8
    {
        pitches 7 8
        pitches 9 10
    }
    rhythm 4 8 4.
    {
        pitches 11
    }
    rhythm 4. 4.
    {
        pitches 12
    }
    rhythm 4. ~ 4 8
    {
        pitches 11
```

```
    }
    rhythm 4. 4 8 4 8 4 8 4. 4 8 4 _8 _4.
    {
        pitches 10 10 10 9 10 9 9 8 8 8 7
    }
}
```

Using scale degrees instead of absolute note names has several advantages:

1. Scale degrees are syntactically easier and shorter to write.
2. Thinking in terms of scale degrees is often more adequate regarding music theory and reflects the way most composers and musicians think about pitches.
3. Scale degrees can be easily projected onto another scale. In other words, the same degrees can be used in another scale context, which allows interesting musical variations.

This is also the case for the *Vltava* model, in which the theme is presented in two scale contexts, namely a minor and a major version.

Note that the scale contexts used in the previous example are optional, because a default scale context is derived from the current key context automatically. In the left branch, the current key context is *Em* (E minor) which results in a matching minor scale context by default. In the right branch, the harmonic context is *E: (E major) and therefore the default scale is _major*. Refer to section Harmonic Contexts for more details.

Scale Definitions

MPS provides a number of built-in scales, which are listed in the following table:

Name	**Identifier**	**Degrees in Semitones**
Major	`major`	`0 2 4 5 7 9 11`
Ionian	`ionian`	`0 2 4 5 7 9 11`
Minor	`minor`	`0 2 3 5 7 8 10`
Aeolian	`aeolian`	`0 2 3 5 7 8 10`
Blues	`blues`	`0 3 5 6 7 10`
Chromatic	`chromatic`	`0 1 2 3 4 5 6 7 8 9 10 11`
Diminished	`diminished`	`0 1 3 4 6 7 9 10`
Dorian	`dorian`	`0 2 3 5 7 9 10`
Harmonic Major	`harmonicMajor`	`0 2 4 5 7 8 11`
Harmonic Minor	`harmonicMinor`	`0 2 3 5 7 8 11`
Locrian	`locrian`	`0 1 3 5 6 8 10`
Lydian	`lydian`	`0 2 4 6 7 9 11`
Major Pentatonic	`majorPentatonic`	`0 2 4 7 9`
Minor Pentatonic	`minorPentatonic`	`0 3 5 7 10`

Melodic Major	`melodicMajor`	`0 2 4 5 7 8 10`
Melodic Minor	`melodicMinor`	`0 2 3 5 7 9 11`
Mixolydian	`mixolydian`	`0 2 4 5 7 9 10`
Phrygian	`phrygian`	`0 1 3 5 7 8 10`
Whole-tone	`whole`	`0 2 4 6 8 10`

If additional scales are required, users are able to define custom scales using scale definitions in the header section of composition files. Here is an example definition for the dorian scale:

```
scaleDef dorian
{
    degrees 0 2 3 5 7 9 10
}
```

Loudness

To account for the loudness dimension of music, MPS supports both static loudness contexts and gradual loudness contexts. The latter are used to model *crescendo* and *decrescendo*.

Static loudness specifications are syntactically described with the `loudness` keyword followed by a single loudness instruction such as

```
loudness ff
```

Refer to the following table for a enumeration of possible loudness specifications and mappings to common loudness units.

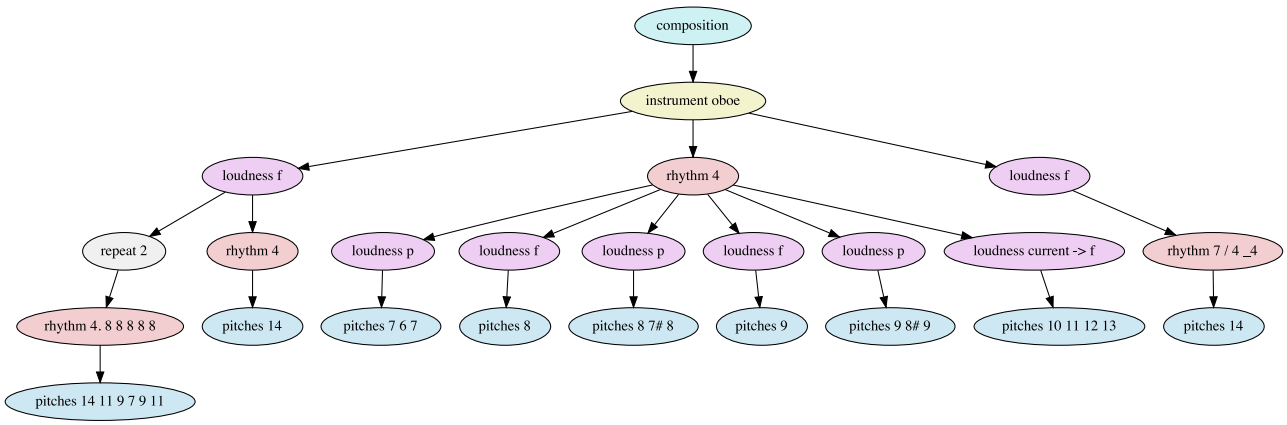
Name	Literal	MIDI Velocity	Amplitude	Approximated Sound Pressure Level in dB(SPL)
pppppp	pppppp	4	0.03	3.78
ppppp	ppppp	8	0.06	7.56
pppp	pppp	16	0.13	15.12
pianopianissimo	ppp	28	0.22	26.46
pianissimo	pp	40	0.31	37.80
piano	p	52	0.41	49.13
mezzopiano	mp	64	0.50	60.47
mezzoforte	mf	76	0.60	71.81
forte	f	88	0.69	83.15
fortissimo	ff	100	0.79	94.49
fortefortissimo	fff	112	0.88	105.83
ffff	ffff	120	0.94	113.39
fffff	fffff	124	0.98	117.17

ffffff	ffffff	127	1.00	120.00

Gradual loudness specifications (i.e. *crescendo* and *decrescendo*) contain two loudness instructions delimited by the token `->` such as

```
loudness p -> f
```

For gradual loudness instructions, the special loudness instruction `current` may be used, which refers to the last loudness level specified in the composition. This is also demonstrated in the following context tree model of W.A. Mozart's *Concerto for Flute, Harp, and Orchestra in C major, K. 299/297c*.

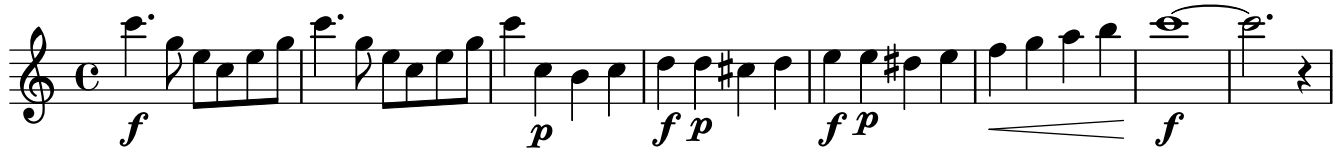


The language representation of this model looks like this:

```
composition
{
    instrument oboe
    {
        loudness f
        {
            repeat 2
            {
                rhythm 4. 8 8 8 8 8
                {
                    pitches 14 11 9 7 9 11
                }
            }
            rhythm 4
            {
                pitches 14
            }
        }
        rhythm 4
        {
```

```
            loudness p
            {
                pitches 7 6 7
            }
            loudness f
            {
                pitches 8
            }
            loudness p
            {
                pitches 8 7# 8
            }
            loudness f
            {
                pitches 9
            }
            loudness p
            {
                pitches 9 8# 9
            }
            loudness current -> f
            {
                pitches 10 11 12 13
            }
        }
        loudness f
        {
            rhythm 7/4 _4
            {
                pitches 14
            }
        }
    }
}
```

This is the resulting score. Note specifically the *crescendo* resulting from a gradual loudness context in the sixth measure:



Harmonic Contexts

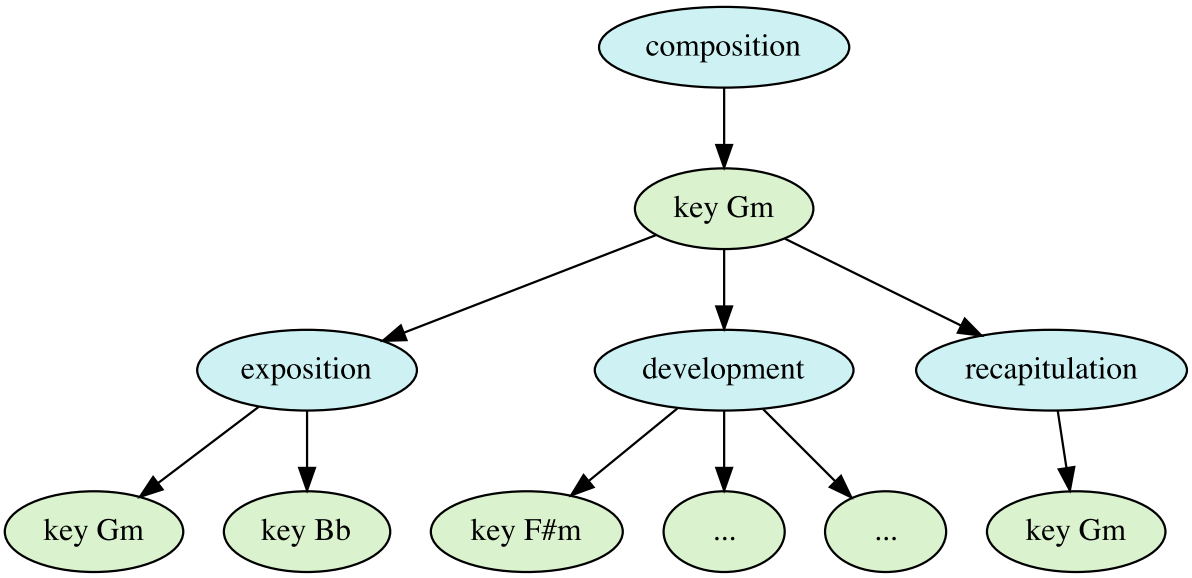
Harmonic contexts are especially important in western tonal music, in which pitches in compositions are usually organized in reference to specific keys. Matching scales and functions

of specific chords can be derived depending on the key context. MPS supports explicit specifications of harmonic contexts including hierarchically arranged keys and contextual harmonies.

Keys

Keys serve as musical „landmarks” in tonal compositions. While simple pieces might only define one key, more complex compositions might incorporate temporary key changes (modulations) or even key changes for whole sections or parts of the piece, for instance compositions geared to the sonata form. Modulations and key changes can be modeled elegantly in MPS using hierarchical arrangements (as discussed in section Hierarchical Structures and Polymorphism). In this way, the scope of the specified keys can be controlled using an arbitrary number of logical levels.

An example is provided in the following figure, which contains a schematic hierarchical arrangement of keys used in the first movement of Mozart's *Symphony No. 40 in G minor, K. 550*. The global key of this movement is *G minor*. Themes are presented in the exposition in *G minor* and its relative major key *Bb major*. In the development, Mozart modulates through a number of keys starting with *F# minor*. The recapitulation concludes in the global key *G minor*.



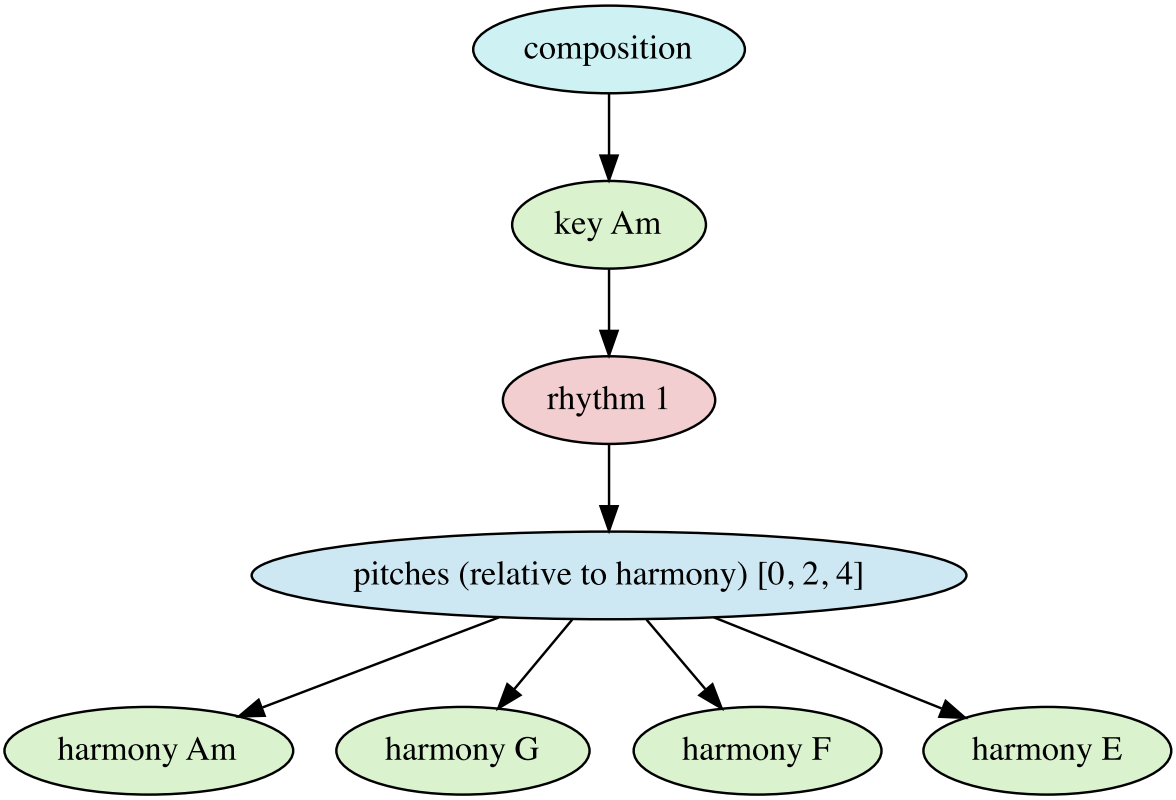
Syntactically, keys are defined by referring to the root note name (for instance `G` or `D#`) and the optional suffix `m` indicating a minor harmony (e.g. `Am` or `Bbm`).

Harmonies

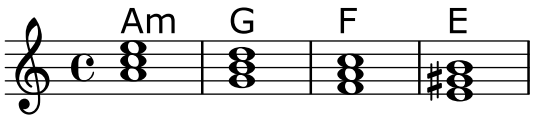
While keys provide a global harmonic context in tonal compositions, harmonic progressions provide local harmonic transitions. These can be expressed implicitly by specifying

simultaneously sounding notes or in an explicit way, for example in the style of lead sheets (as shown in this example).

The following context tree model defines a harmonic progression consisting of four local harmonies. These are hierarchically embedded in the global key context *A minor*.



The resulting score is shown below:



Syntactically, this can be written as:

```
composition
{
    key Am
    {
        rhythm 1
```

```
        {
            pitches(relative to harmony) [0 2 4]
            {
                harmony Am
                harmony G
                harmony F
                harmony E
            }
        }
    }
}
```

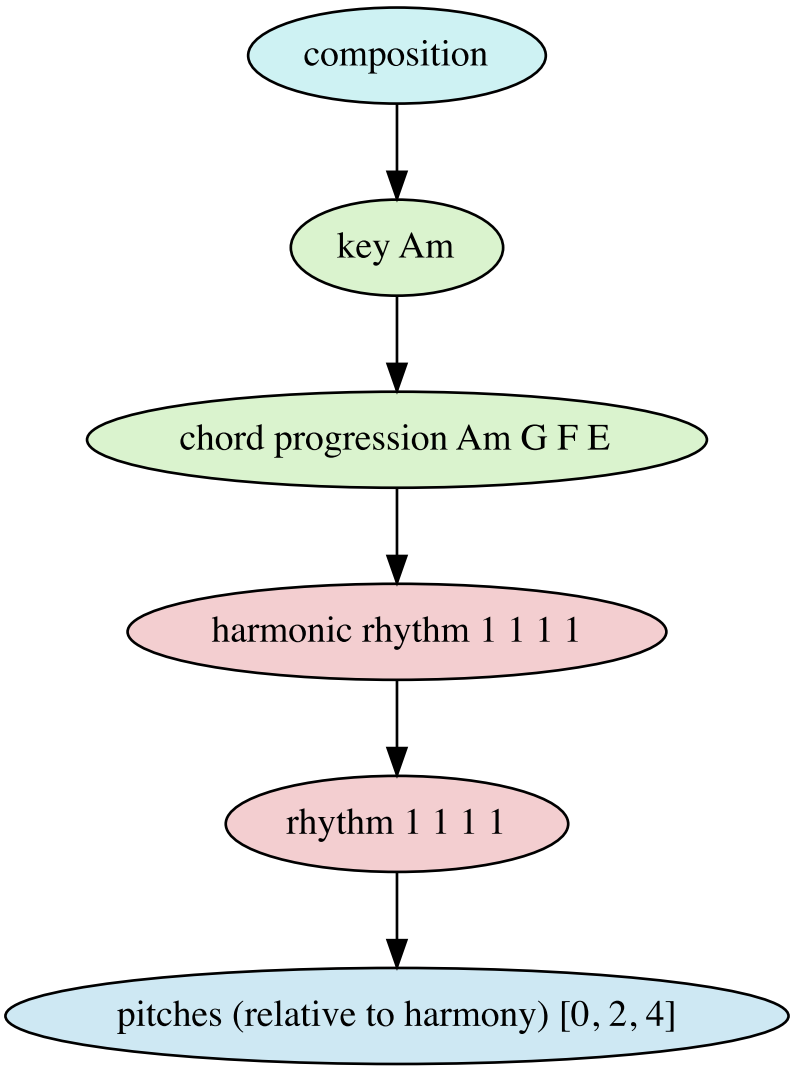
The complexity of harmonies is not limited to major and minor chords. MPS supports additional notes and harmony specifications as specified in the following table:

Syntax	Description
`<integer number>`	Additional harmony note relative to the root note, expressed in terms of scale degrees. For example, `F7` translates to a F major chord with added minor seventh.
`#` or `b` (prefix)	Optional prefix for additional harmony notes to indicate a semitone correction upwards or downwards, respectively.
`maj7`	Adds a major seventh relative to the root note.
`m7`	Indicates a minor chord with a minor seventh.
`sus2`	Suspended second chord in which a perfect second is added and the third is omitted.
`sus4`	Suspended second chord containing a perfect fourth but no third.
`°`	Diminished chord
`+`	Augmented chord
`power`	Power chord containing only the root and the fifth. Frequently used in rock and metal genres.

Note that these additions can be combined, for instance `A7sus4` defines a harmony with the notes A, D, E and G. Refer to section Harmonic Modifiers for more examples demonstrating harmony additions.

Harmonic Progressions

In certain cases it is convenient to specify a harmonic sequence as a whole. In MPS, this is possible using the `harmonicProgression` keyword in combination with a `harmonicRhythm` instruction defining the duration of each harmony in the progression. This is demonstrated in the following context tree model:



It results in an equivalent score as the previous example in section Harmonies.

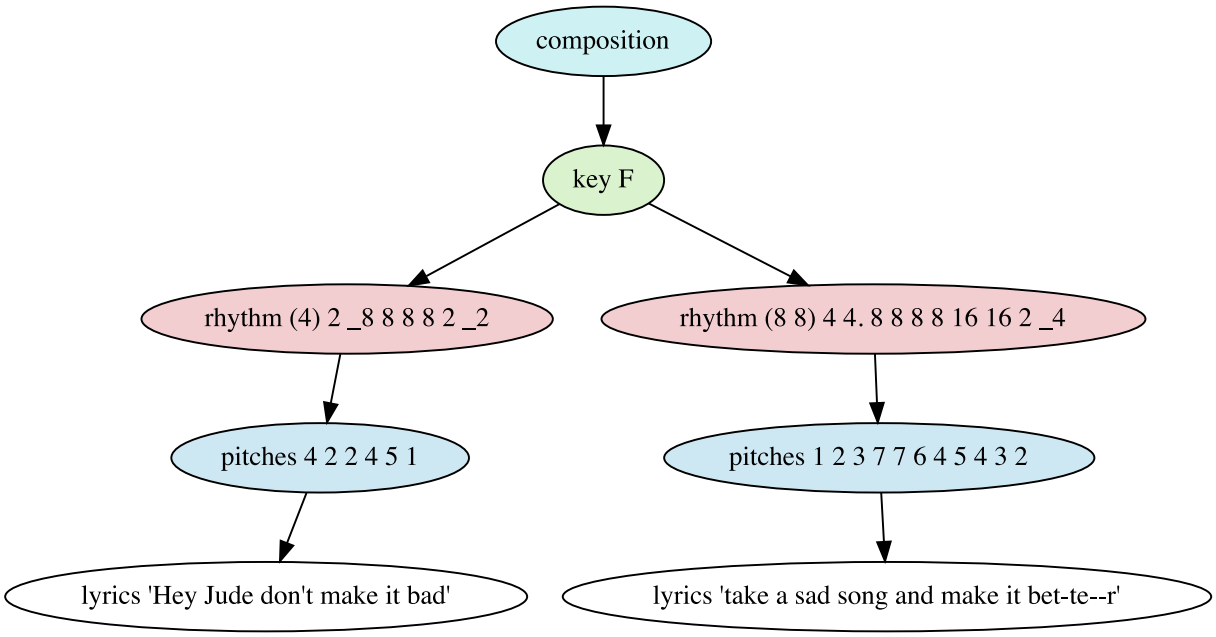
The following code contains the corresponding language representation:

```
composition
{
    key Am
    {
        harmonicProgression Am G F E, harmonicRhythm 1 1 1 1
        {
            rhythm 1 1 1 1
            {
                pitches(relative to harmony) [0 2 4]
```

```
            }
        }
    }
}
```

Lyrics

In vocal music, sung notes are normally associated with syllables, which is considered as a separate context dimension in the MPS model. Syllables are provided using a simple word-based syntax. To distribute syllables of a word onto multiple notes, hyphens (-) may be used. Syllable assignments for specific notes can be skipped using underscores (_). As an example, the first measures of the song *Hey Jude* by the Beatles is used. The context tree model looks like this:

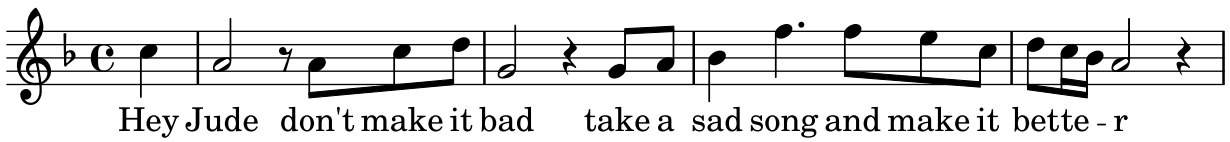


Is can be represented with the following syntax:

```
composition
{
    key F
    {
        rhythm (4) 2 _8 8 8 8 2 _2
        {
            pitches 4 2 2 4 5 1
            {
                lyrics "Hey Jude don't make it bad"
            }
```

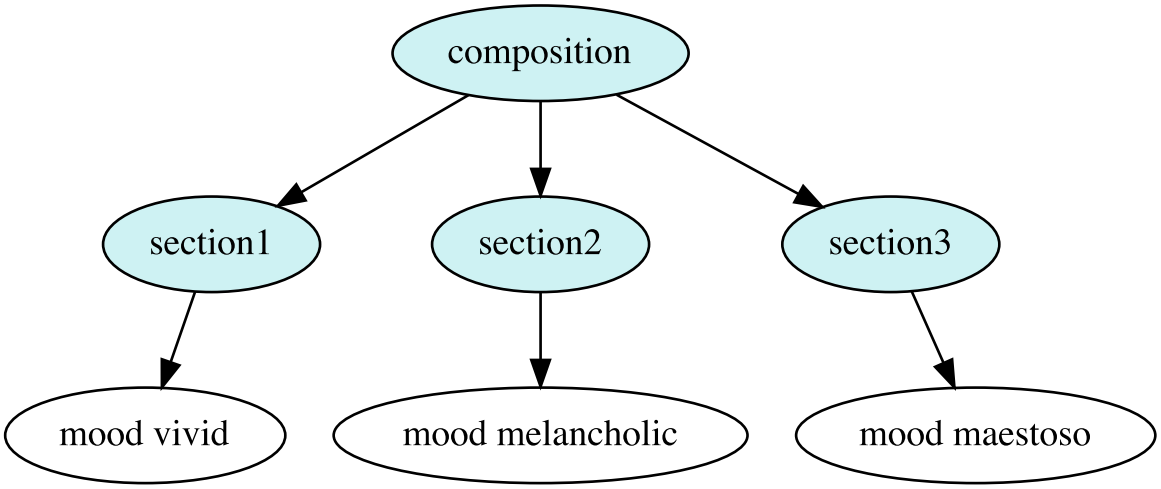
```
        }
        rhythm (8 8) 4 4. 8 8 8 8 16 16 2 _4
        {
            pitches 1 2 3 7 7 6 4 5 4 3 2
            {
                lyrics "take a sad song and make it bet-te--r"
            }
        }
    }
}
```

It results in the following score:



Custom Contexts

MPS offers the feature to create arbitrary custom contexts. An example is shown in the following model:



The context tree model contains three sections which individual moods are described by means of custom context nodes. Custom contexts are syntactically defined by the keyword `customContext`, followed by a context identifier (in this case `mood`) and a string literal containing the value for the context. Refer to the following listing for the corresponding language representation:

```
composition
{
    fragment section1
    {
        customContext mood "vivid"
    }

    fragment section2
    {
        customContext mood "melancholic"
    }

    fragment section3
    {
        customContext mood "maestoso"
    }

}
```

Custom contexts are visually represented as separate layers in models. Scores generated from models containing custom contexts will contain textual annotations such as ``Mood: vivid'' at the top of the relevant staves.

Context Modifiers

Frequently, already introduced musical material is slightly changed and shaped in the course of compositions. In these cases, no fundamentally new ideas are introduced, but existing ones are modified. To account for this, so called context modifiers allow to adjust already existing musical material. Their functionality is explained in the following subsections.

By default, modifiers are applied to the next matching context above the modifier node. If the modifier should also be applied to nodes beneath it, add the keyword `recursive` after the modifier specification.

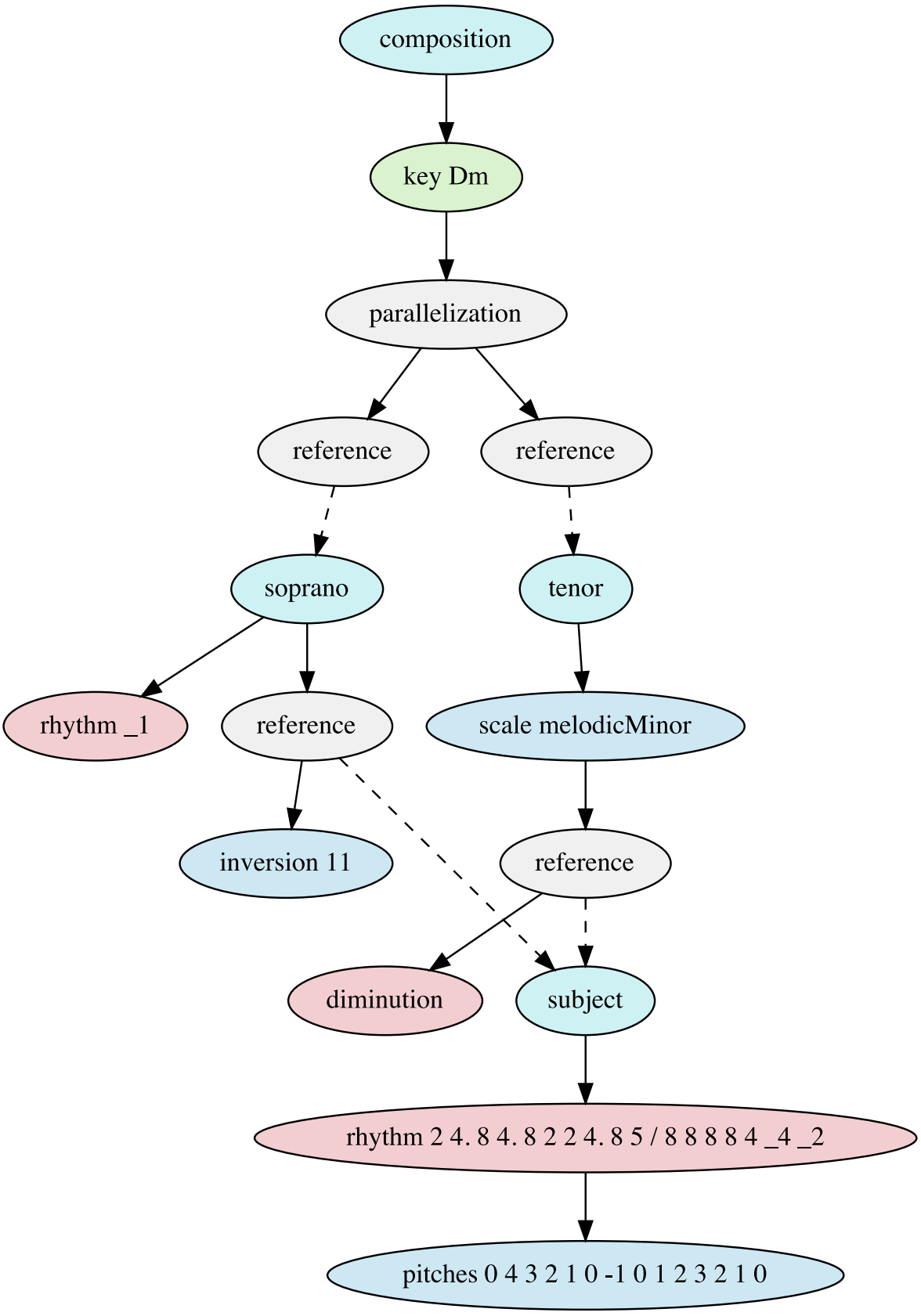
Rhythmic Modifiers

Rhythmic context modifiers have the purpose of manipulating existing rhythmic contexts in a musical composition.

Augmentations and Diminutions

Rhythmic augmentation involves prolonging the note lengths of a given rhythm by multiplying the original lengths with a constant factor, typically 2. However, other scale factors are possible. A rhythmic diminution is considered as the opposite of a rhythmic augmentation, i.e. the note lengths are not extended but shortened by a constant factor.

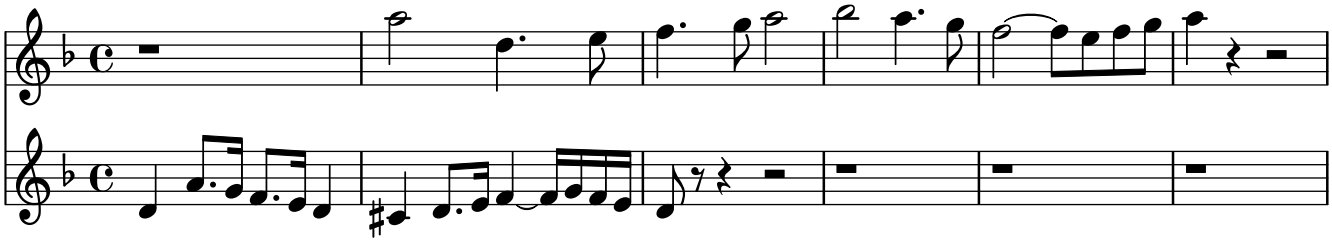
The following example demonstrates a model of a subject being transformed using diminution and inversion. It can be found in J.S. Bach’s *Die Kunst der Fuge, BWV 1080, Contrapunctus VII*.



The language representation looks as follows:

```
composition
{
    key Dm
    {
        parallel
        {
            fragmentRef soprano
            fragmentRef tenor
        }
    }
}

fragment soprano
{
    rhythm _1
    inversion 11
    {
        fragmentRef subject
    }
}

fragment tenor
{
    diminution, scale melodicMinor
    {
        fragmentRef subject
    }
}

fragment subject
{
    rhythm 2 4. 8 4. 8 2 2 4. 8 5/8 8 8 8 4 _4 _2
    {
        pitches 0 4 3 2 1 0 -1 0 1 2 3 2 1 0
    }
}
```

The following results from this model:



Rhythmic Extensions

Rhythmic modifiers are used to extend the duration of the last note or rest in a rhythm. This modifier was already demonstrated in the context tree model for Beethoven's *Symphony No. 5 in C Minor, Op. 67* in section Introductory Example.

Syntactically, rhythmic extensions are specified using the keyword `rhythmicExtension`, followed by a note duration as explained in section Rhythms. If the note duration is positive, the rhythm is extended. If the note duration is negative, the rhythm is shortened by the absolute value of the given negative duration.

Rhythmic Adjustments

Rhythmic adjustment modifiers allow to modify the rhythm in the current context at the beginning and at the end. The modifications are specified by means of two durations for the beginning and the end of the rhythm, respectively. It is possible to specify both or only one of the parameters. Refer to the following table for detailed parameter descriptions.

Parameter	Description
`startDelta`	Specifies how the rhythm is modified at the beginning. If `startDelta` is positive, the rhythm will start from the given time, effectively shortening the rhythm by `startDelta`. If `startDelta` is negative, the first note or rest of the rhythm will be extended.
`endDelta`	Specifies a duration for the adjustment of the end of the rhythm. If `endDelta` is positive, the rhythm is extended; if `endDelta` is negative, the rhythm is shortened. The behaviour is identical with the `rhythmicExtension` modifier introduced in section Rhythmic Extensions.

Rhythmic Insertions

This modifier inserts a rhythm into the contextually present rhythm. This can either happen in an additive manner, whereupon existing notes and rests are shifted to the right, or in a destructive manner, whereupon existing elements are overwritten.

A rhythmic insertion was already demonstrated in Queen's *Bohemian Rhapsody* in section Inheritance. Refer to the score in this section and compare the rhythms in the first and the

second measure, which both start off with three eighth notes, but continue differently. In the model this is expressed using a rhythmic insertion. It is used in the right subtree, which represents the specifics of the second measure. The rhythm `8 16 5/16` is inserted into the basic rhythm `_8 8 8 8 4 4` at offset 2, i.e. after the duration of a half note, effectively replacing the two quarter notes with the specified rhythm. The following table contains explanations for all parameters of this modifier.

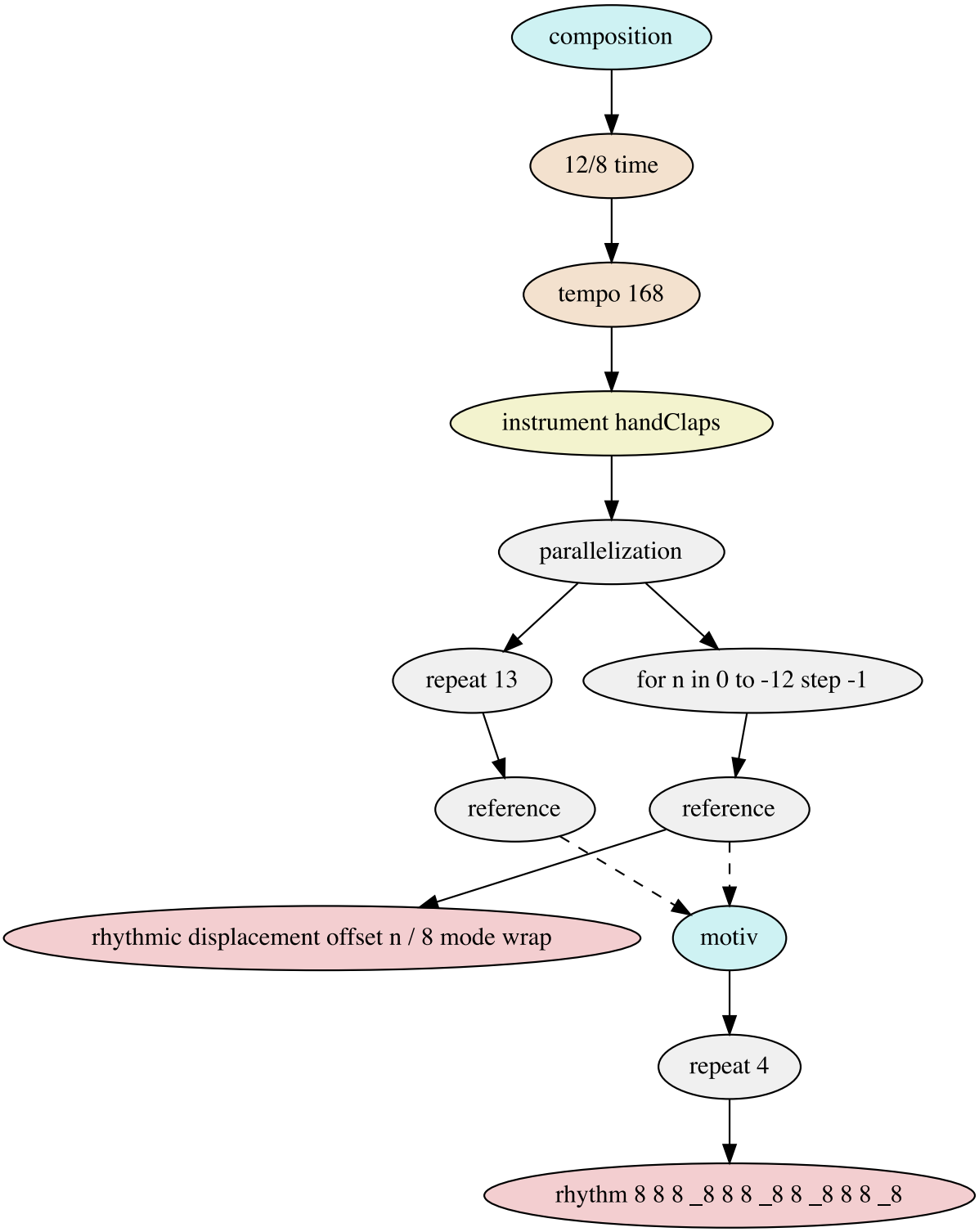
Parameter	Description
`offset`	Specifies after which duration the insertion should be applied to the rhythm.
`rhythm`	Defines the rhythm to be inserted in the syntax introduced in section Rhythms.
`mode`	Either `insert` to shift existing notes and rests after the insertion to the right or `overwrite` to overwrite existing elements.

Rhythmic Displacements

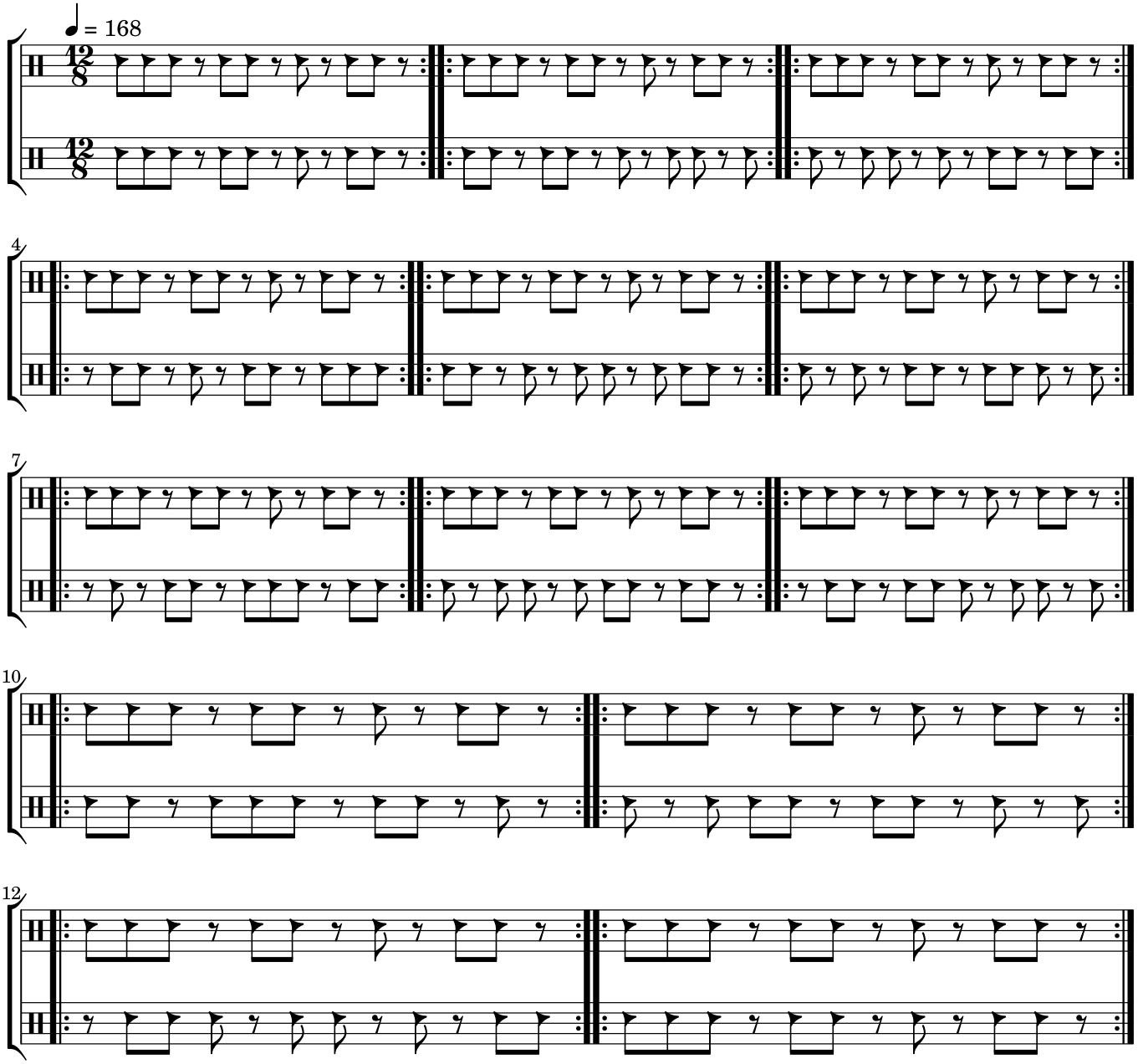
Rhythmic displacement modifiers are used to translate existing rhythms by moving them to the right or to the left in itself. The modifier takes a note duration offset and a mode specification as parameters, which is explained in detail in the following table.

Parameter	Description
`offset`	Defines the rhythm translation offset. For positive durations, the rhythm is shifted to the right, for negative durations to the left.
`mode`	In `discard` mode, notes moved over the rhythm's boundary are removed. In `wrap` mode, the notes are appended to the other end of the rhythm.

As an example, consider Steve Reich's composition *Clapping Music*, in which a rhythmic motif is repeatedly performed by two players. For the second player, the rhythm is iteratively shifted and wrapped, resulting in twelve rhythmic variations. The following context tree model contains a repeatedly applied rhythmic displacement modifier:



The following score results:



The syntactical representation of the model follows:

```
composition
{
    time 12/8, tempo 168
    {
        instrument handClaps
        {
            parallel
            {
                repeat 13
                {
```

```
                    fragmentRef motiv
                }
                for n in 0 to -12 step -1
                {
                    fragmentRef motiv
                    {
                        rhythmicDisplacement mode wrap offset n/8
                    }
                }
            }
        }
    }
}

fragment motiv
{
    repeat 4
    {
        rhythm 8 8 8 _8 8 8 _8 8 _8 8 8 _8
    }
}
```

Pitch Modifiers

Pitch modifiers are used for manipulating contexts in the musical pitch dimension.

Transpositions

Transpositions have the effect of modifying contextually available pitches. The modifier can be applied in three modes in order to support semitone-based transpositions, scale-based transpositions and octave translations. All parameters are explained in the following table:

Parameter	Description
`mode`	Defines the unit of the `interval` expression. Three modes are available: `absolute` for semitone-based transpositions, `inScale` to perform transpositions of scale degrees and `octaves` for octave translations. If the parameter is not specified, the default `absolute` will be used.
`interval`	Expression which must be interpretable as an integer number. The unit of this number is defined by the `mode` parameter.

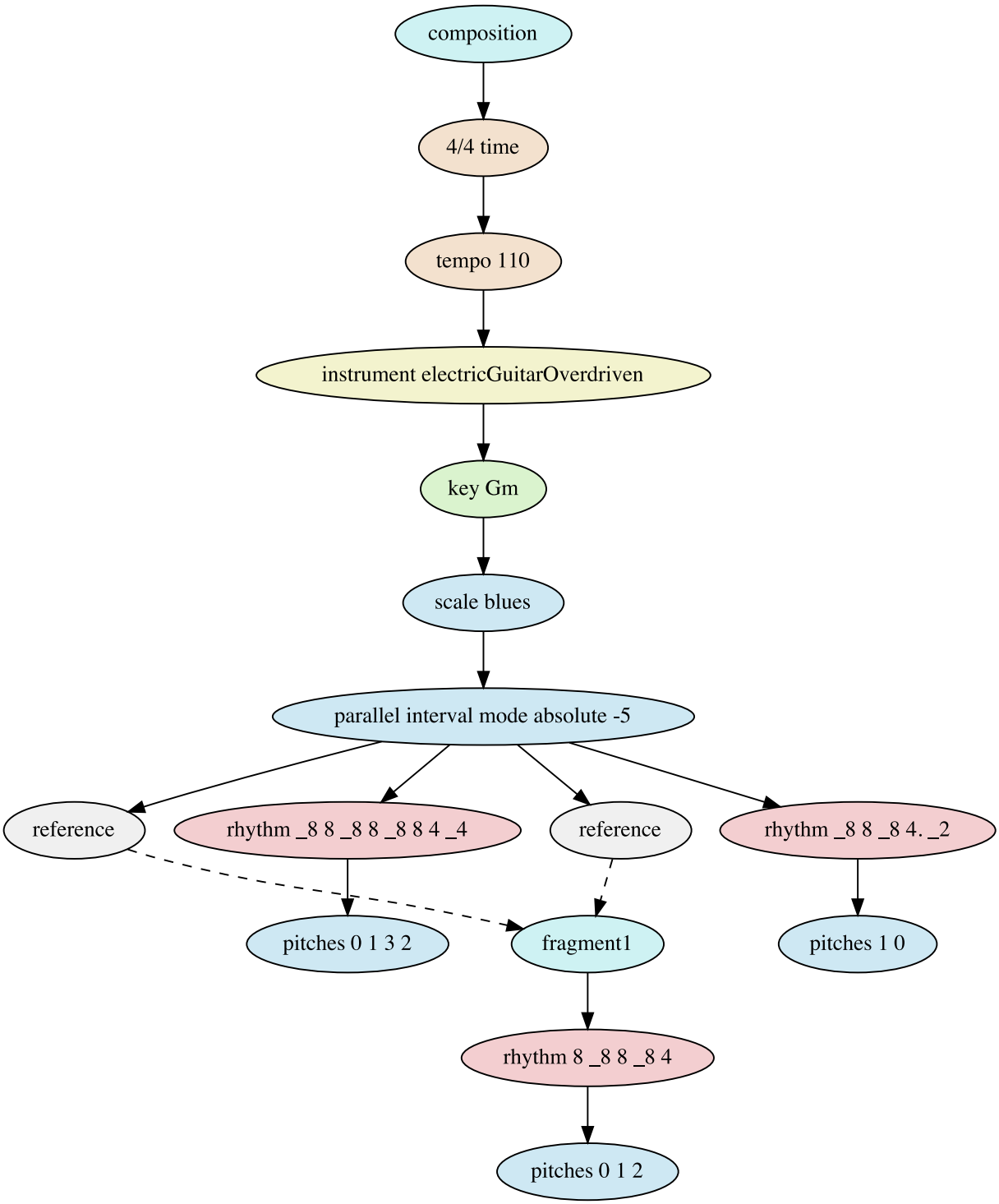
Refer to section Sequences for an example demonstrating various transposition techniques.

Inversions

Inversions were already demonstrated in section Augmentations and Diminutions in conjunction with a diminution using J.S. Bach's *Die Kunst der Fuge, BWV 1080, Contrapunctus VII* as an example.

Parallel Intervals

Parallel interval modifiers add simultaneously audible pitches in a specific interval to existing pitches. The intervals can be specified in terms of semitones, scale degrees or octaves. As an example, a context tree model of the guitar intro of Deep Purple's *Smoke on the Water* is demonstrated:

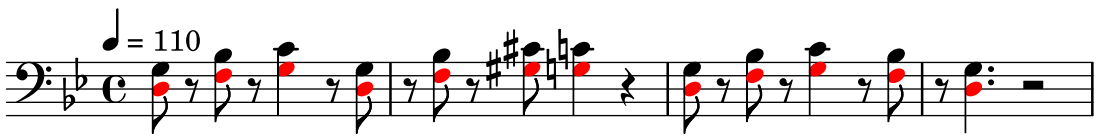


The language representation of this model is:

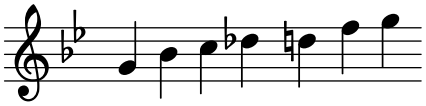
```
composition
{
```

```
    time 4/4, tempo 110
    {
        instrument electricGuitarOverdriven
        {
            key Gm
            {
                scale blues
                {
                    parallelInterval mode absolute -5 recursive
                    {
                        fragmentRef fragment1

                        rhythm _8 8 _8 8 _8 8 4 _4
                        {
                            pitches 0 1 3 2
                        }

                        fragmentRef fragment1

                        rhythm _8 8 _8 4. _2
                        {
                            pitches 1 0
                        }
                    }
                }
            }
        }
    }
}

fragment fragment1
{
    rhythm 8 _8 8 _8 4
    {
        pitches 0 1 2
    }
}
```

The model results in the following score:



The main melodic motif is notated in terms of degrees on the minor blues scale, which consists of the minor pentatonic scale with an added „blue note” between the 3rd and 4th scale degree:



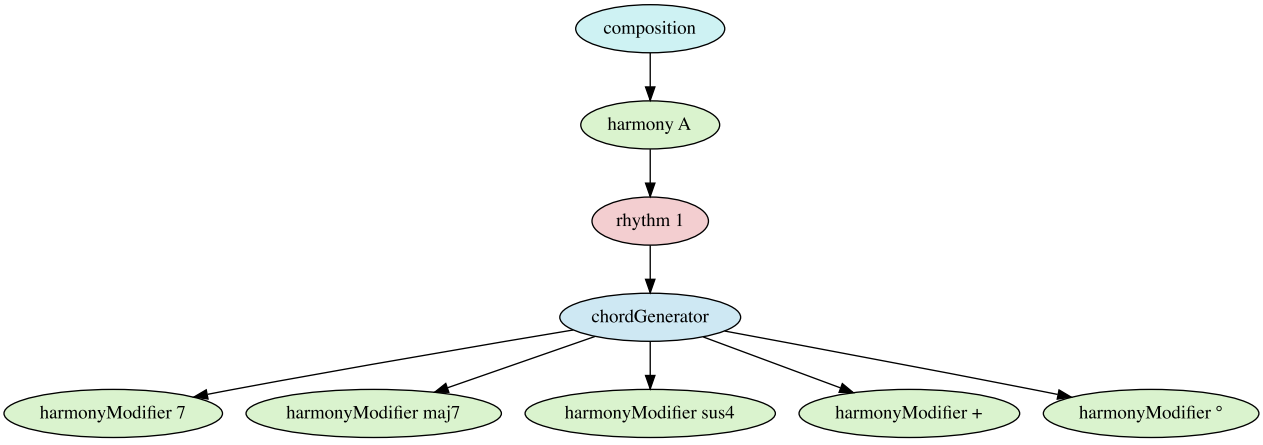
The upper notes of the famous *Smoke on the Water* riff can be specified in terms of scale degrees on the G minor blues scale. When analyzing the distance between the notes, it becomes apparent that the lower notes have a constant distance to the upper notes, namely five semitones or a perfect fourth. It is therefore convenient to specify this circumstance rather than specifying each lower pitch manually. Refer to the following table for a detailed description of parallel interval modifier parameters.

Parameter	Description
`mode`	Specifies the interval unit. Available modes are `absolute` (in semitones), `inScale` (for scale-specific parallel intervals) and `octaves`.
`interval`	Expression to define the parallel interval. The expression must be interpretable as integer number. See section Expressions for more details.

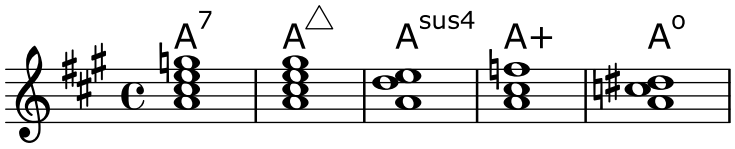
Note that the first and third measure are exactly identical, which is why the individual musical contexts of these measures were extracted to a fragment and referenced twice, as already described in section Fragments.

Harmonic Modifiers

Harmonic modifiers are used to extend or alter contextually accessible harmonies. In the following context tree model, various harmony modifications of the base harmony A major are demonstrated:



The resulting chords of the modifications are: A major, A⁷, A^{maj7}, A augmented and A diminished. Compare the model with the resulting score:



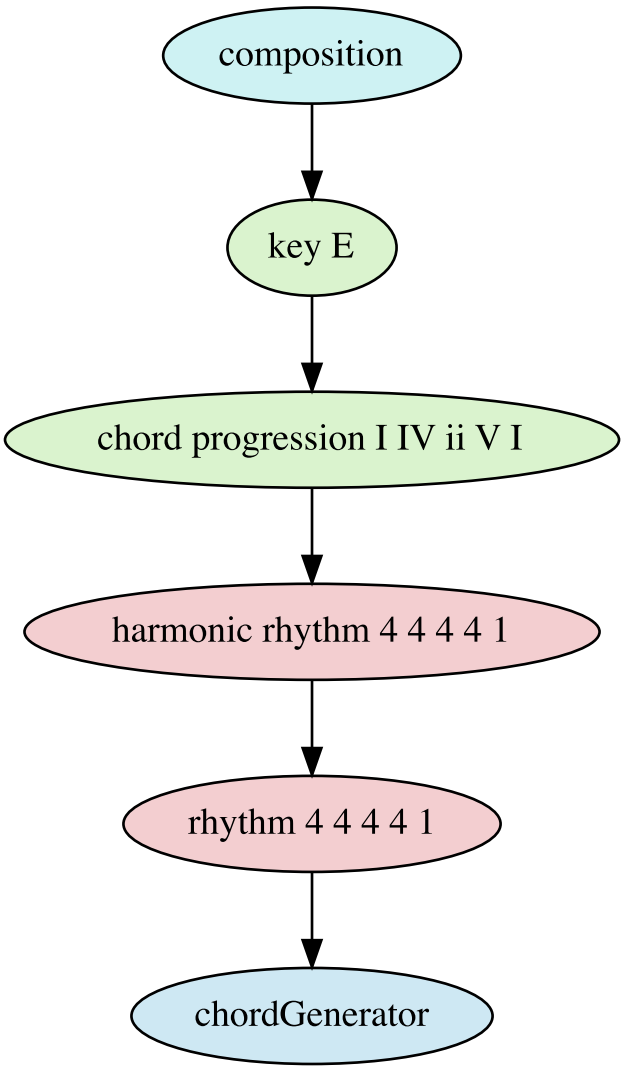
Refer to section Chord Generators for details on the `chordGenerator`.

Context Generators

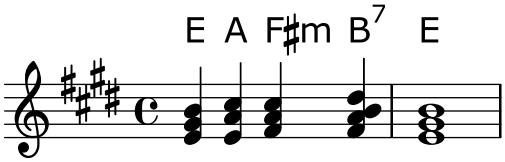
The purpose of context generators is to create new contexts based on already existing contexts. For example, pitch contexts can be built based on harmonic contexts, as explained in the following sections.

Chord Generators

Chord generators create pitch contexts representing specific chord inversions for contextually available harmonies. Refer to the following model for an example, in which an abstract chord progression is defined using Roman numerals.



Concrete chord inversions are derived using a chord generator, resulting in the following score:



In the language, this model can be expressed as follows:

```
composition {
    key E
    {
```

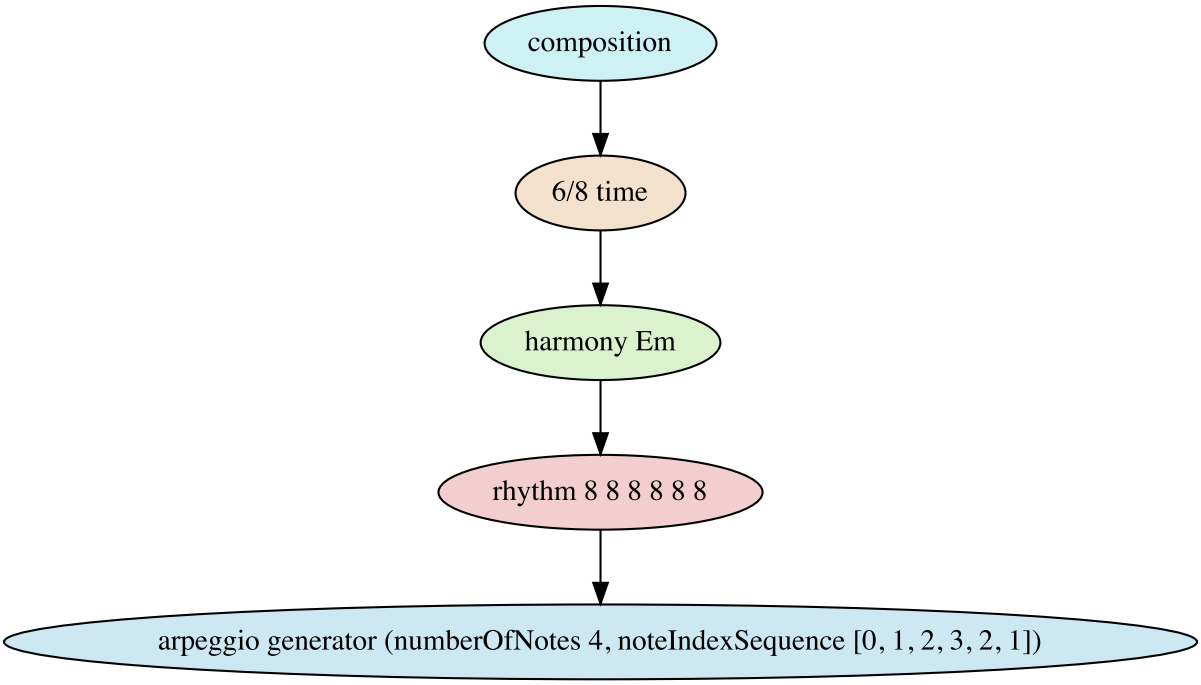
```
        harmonicProgression I IV ii V7 I
        {
            harmonicRhythm 4 4 4 4 1
            {
                rhythm 4 4 4 4 1
                {
                    chordGenerator
                }
            }
        }
    }
}
```

Chord generators can be flexibly configured for various musical applications. All possible parameters are described in the following table:

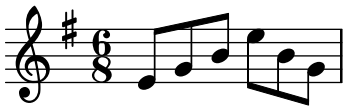
Parameter	Description
`startOctave`	Defines the octave in which the lowest note of the first chord is generated.
`startInversion`	Specifies the default inversion of this chord generator. 0 corresponds to the root position, 1 to the first inversion etc.
`numberOfNotes`	Defines how many notes are generated for each chord. If this parameter is not specified, the minimum number of notes to express a harmony adequately are used. For example, three notes are used for major or minor chords but four notes for a dominant seventh chord.
`includeBassNote`	If set to `true`, the bass note (which in some cases can be different from the root note) is included in chords.
`findNearestInversion`	If set to `true`, the system will minimize the distance between successive chords. In other words, inversions with a minimum aggregated semitone distance to the previous chord will be used.

Arpeggio Generators

Arpeggio generators are specialized chord generators which allow to distribute individual notes of generated chords sequentially in time. A simple example is demonstrated in the following context tree model:



The resulting score is:



The corresponding language representation is:

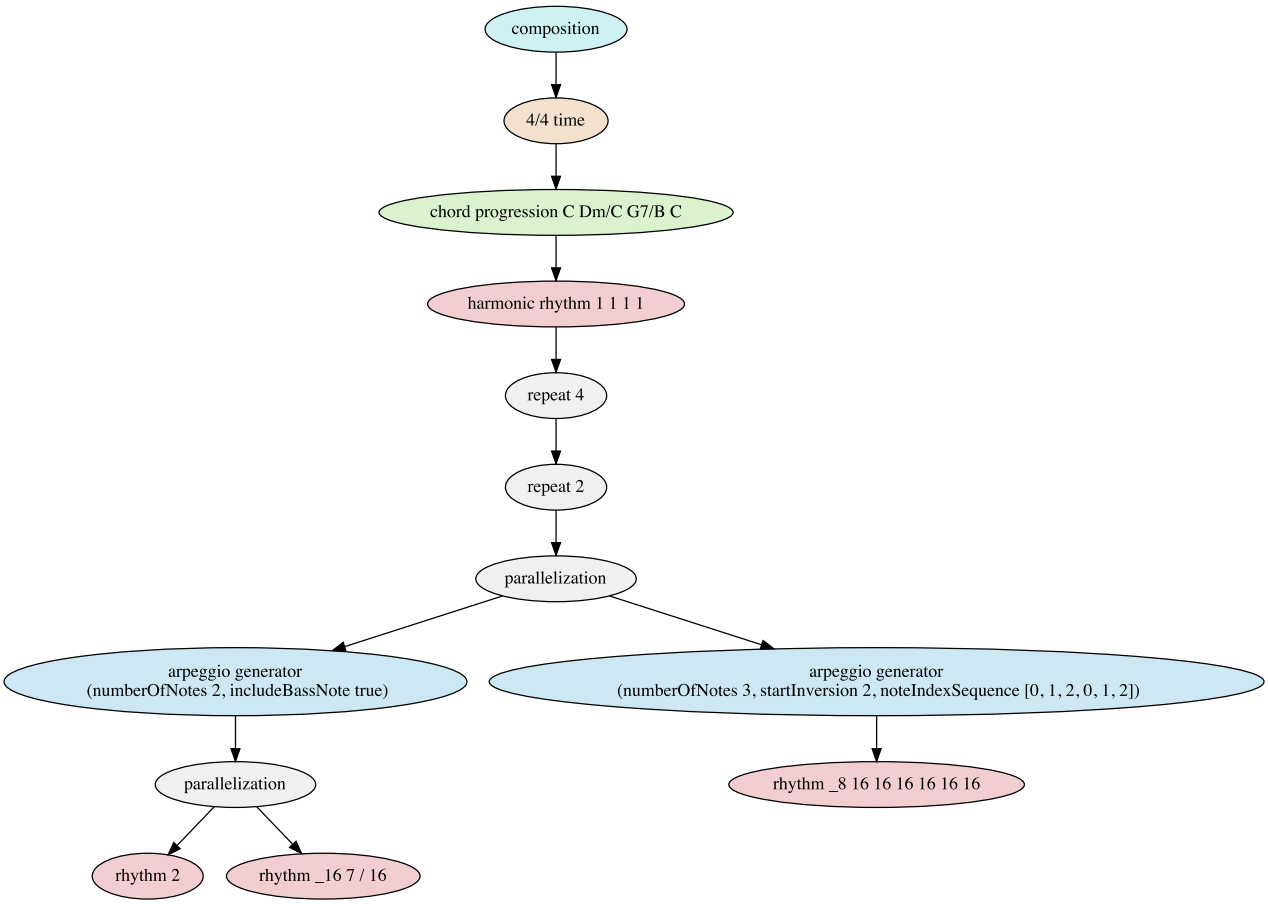
```
composition
{
  time 6/8, harmony Em
  {
    rhythm 8 8 8 8 8 8
    {
      arpeggioGenerator (numberOfNotes 4 noteIndexSequence 0 1 2 3 2
 1)
    }
  }
}
```

Internally, arpeggio generators determine concrete chord inversions just like chord generators. Therefore, all parameters of chord generators (see section Chord Generators) can be applied to arpeggio generators. However, instead of generating simultaneously played notes, arpeggio generators produce sequentially played notes in a contextually available rhythm. For this

purpose, the generator sequentially chooses notes from the current chord. By default, notes are chosen in ascending order and this sequence is wrapped if more notes are required. For example, for a D minor chord (D-F#-A) and a rhythm with four notes, the resulting arpeggio sequence would be D-F#-A-D.

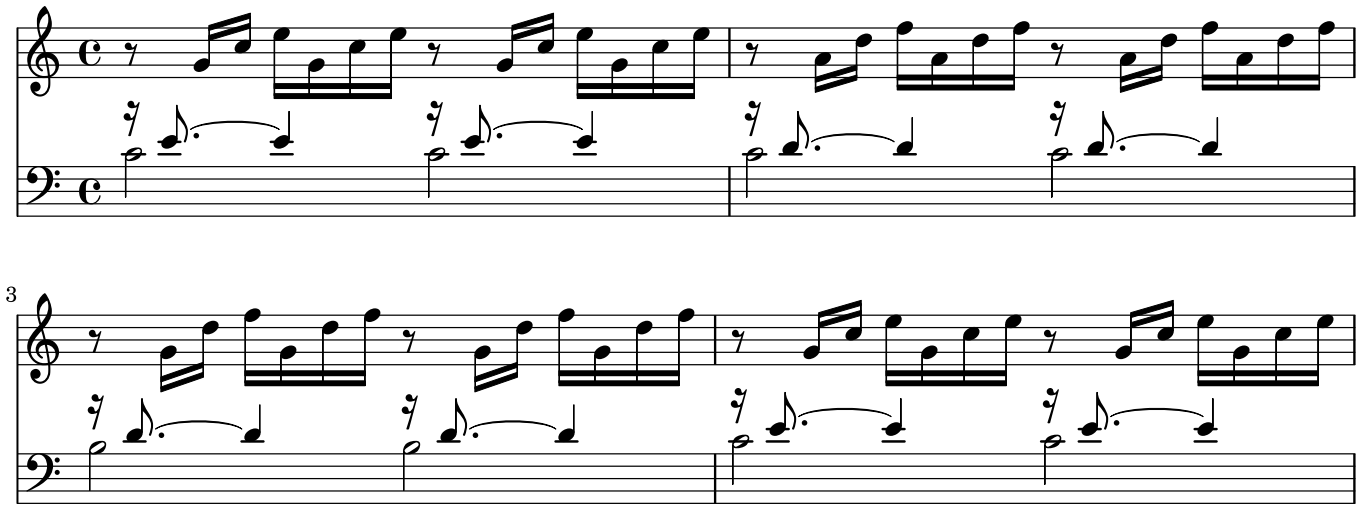
The sequence of the selected notes can be influenced with the so called *note index sequence*. Each note in the chord is assigned a zero-based index (e.g. for the above mentioned example the indices would be: D $\Rightarrow$ 0, F# $\Rightarrow$ 1, A $\Rightarrow$ 2). To produce descending instead of ascending arpeggios, the default note index sequence `0 1 2` could be changed to `2 1 0`. In the previous example, the note index sequence `0 1 2 3 2 1` is used which results in an alternating ascending and descending arpeggio.

A more complex example is demonstrated in the following model:



The model produces the first four measures of J.S. Bach's well-known *Prelude in C Major, BWV 846*. Two separate arpeggio generators are used to generate independent arpeggios for the left and the right hand. An advanced feature is used in the third chord (used in the third measure). The harmony is specified as G⁷ with B in the bass. Additionally, a so called *note exclusion* with the syntax `-B` is specified. It instructs the compiler to skip the relevant note during the chord inversion computing process. As can be seen in measure 3 in the

following score, the note B is not present in the arpeggio. To account for this, specific notes can be excluded from the chord generation process.



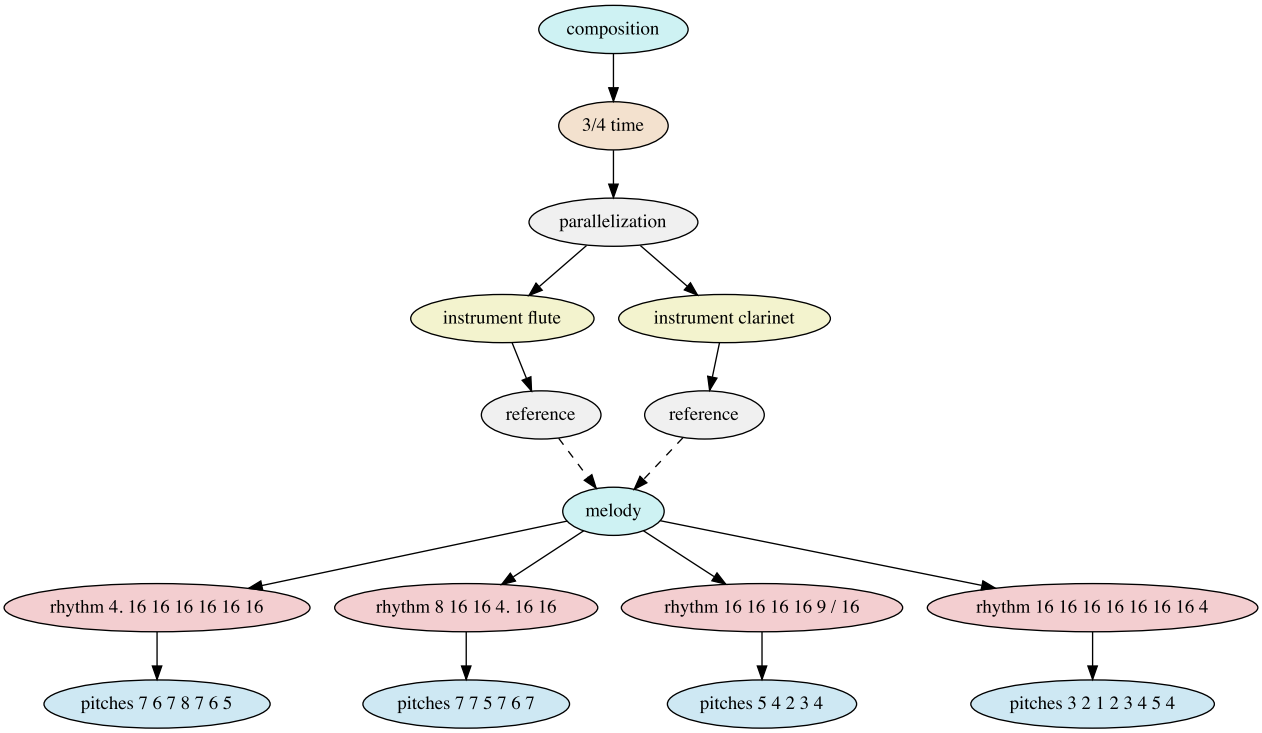
Control Structures

Control structures can be utilized to dynamically reuse contexts in context tree models with the help of loops, iterative modifications and other algorithmic constructs, which are explained in detail in this section.

Parallelizations

Parallelizations are used to indicate that tree branches below are not to be evaluated sequentially, but in parallel. This results in individual musical streams resulting in multiple parts or voices being played simultaneously.

As an example, a parallel version of an already introduced context tree model is shown. The following model uses a *parallelization* node to purpose the melody being played simultaneously by flute and clarinet:

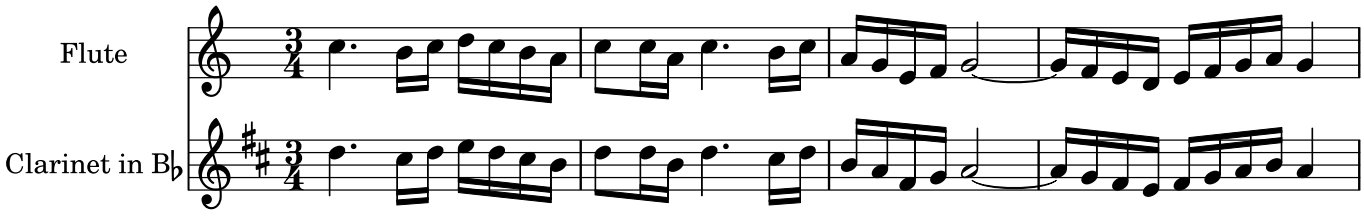


This is syntactically accomplished with the `parallel` keyword:

```
composition
{
    time 3/4
    {
        parallel
        {
            instrument flute
            {
                fragmentRef melody
            }
            instrument clarinet
            {
                fragmentRef melody
            }
        }
    }
}

fragment melody
{
    rhythm 4. 16 16 16 16 16 16, pitches 7 6 7 8 7 6 5
    rhythm 8 16 16 4. 16 16, pitches 7 7 5 7 6 7
```

```
    rhythm 16 16 16 16 9/16, pitches 5 4 2 3 4
    rhythm 16 16 16 16 16 16 16 4, pitches 3 2 1 2 3 4 5 4
}
```

The resulting score is:

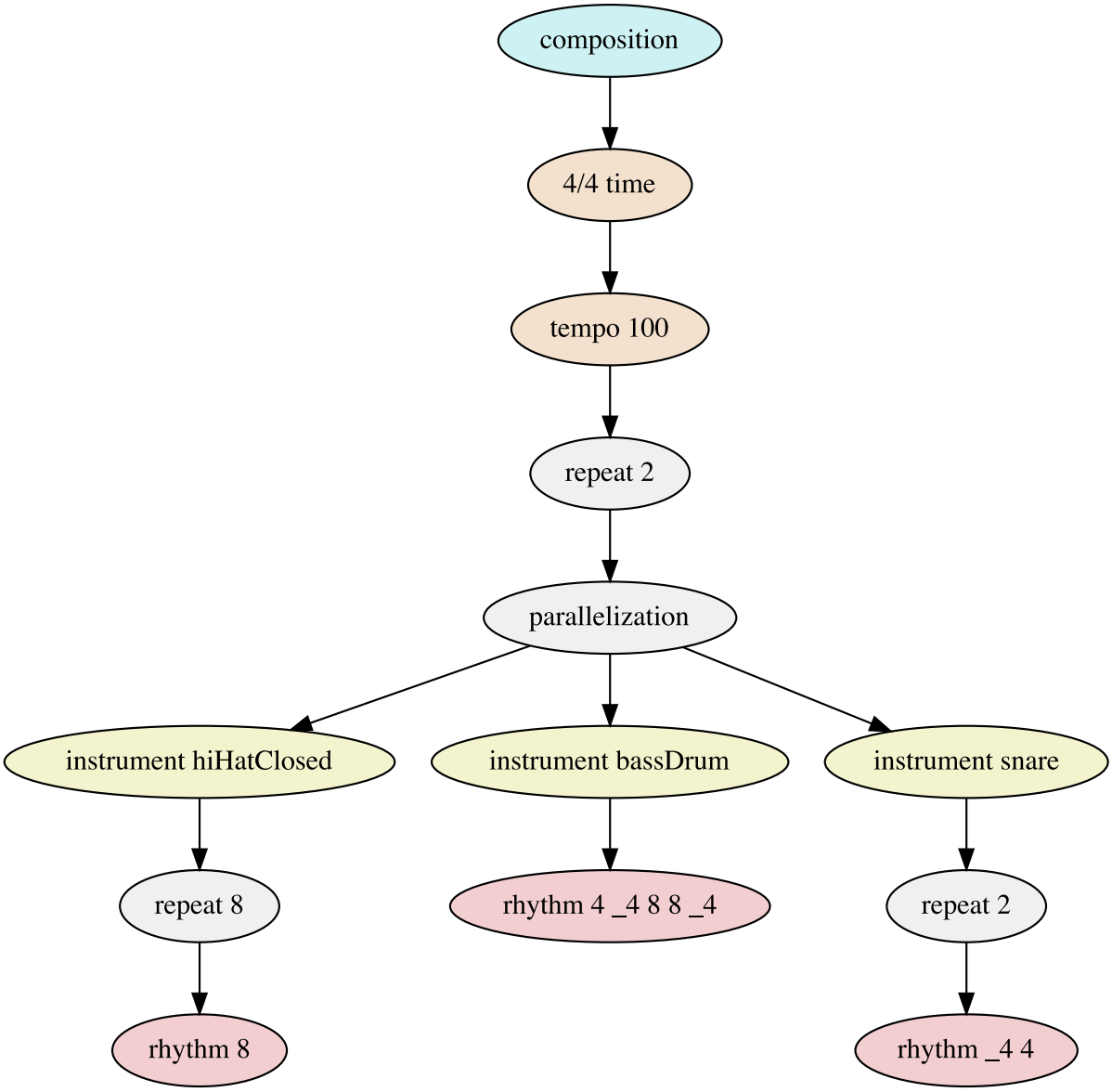


Compare with the model already presented in section Instruments, which results in sequentially played melodies.

Repetitions

Repetition is a frequently utilized technique in music composition and is applied in a variety of forms. A common form of repetitions is known from musical scores, in which repeat signs indicate that a section of the score is to be played again (as an example, refer to section Rhythmic Displacements).

In MPS, arbitrary subtrees of contexts can be repeated, which can be applied to single contexts or combinations of musical contexts. Furthermore, repetitions can be nested hierarchically. This is demonstrated using a context tree model of a simple drum groove:

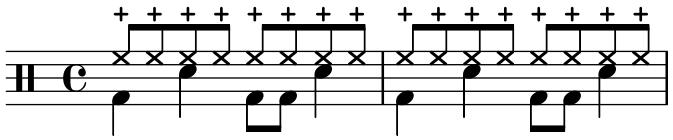


The corresponding language representation is:

```
composition
{
    time 4/4, tempo 100
    {
        repeat 2
        {
            parallel
            {
                instrument hiHatClosed
```

```
                {
                    repeat 8
                    {
                        rhythm 8
                    }
                }
                instrument snare
                {
                    repeat 2
                    {
                        rhythm _4 4
                    }
                }
                instrument bassDrum
                {
                    rhythm 4 _4 8 8 _4
                }
            }
        }
    }
}
```

The model produces the following score:

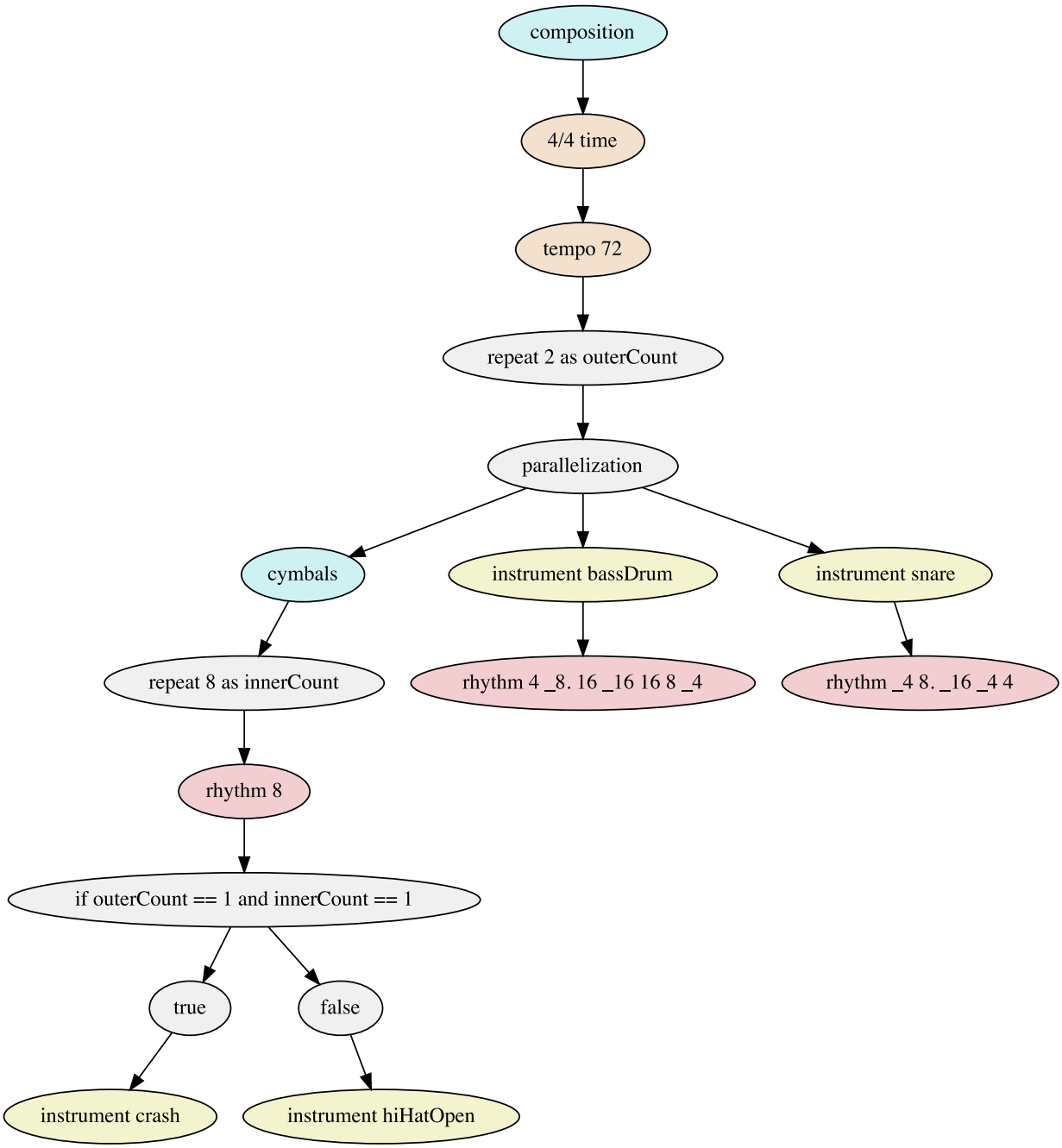


The model contains nested control structures to repeat context subtrees. The outer structure (`repeat 2`) repeats the whole measure produced by the subtree below the `parallel` element. It produces musical material for closed hi-hats, bass drum and snare. A nested repetition resulting in 8 eights notes is specified for the hi-hats. Also, the snare drum repeats the rhythmic pattern of a quarter rest followed by a quarter note (`rhythm _4 4`) twice, which is also expressed as a nested repetition. In this manner, repetitions of musical context subtrees can be hierarchically nested in arbitrary complexity.

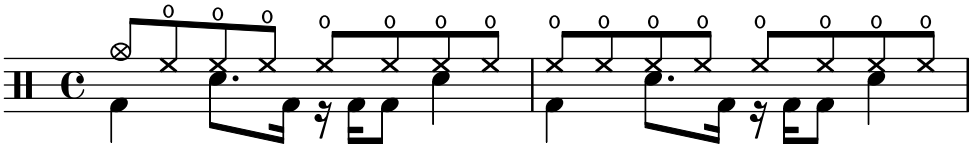
The repeat count can be bound to a variable, which can be utilized to introduce conditional contexts. This technique is demonstrated in the following section.

Conditions

Condition nodes can be used to define conditional contexts. Therefore, an expression is defined which is evaluated to a boolean expression, yielding either `true` or `false`. Depending on the result, a different context tree branch is used. This is illustrated in the following context tree model, which produces the drum introduction of Coldplay's *In My Place*.



The resulting drum part is:



Syntactically, this model can be expressed as:

```
composition
{
    time 4/4, tempo 72
    {
        repeat 2 as outerCount
        {
            parallel
            {
                fragment cymbals
                {
                    repeat 8 as innerCount
                    {
                        rhythm 8
                        {
                            if outerCount == 1 and innerCount == 1
                            {
                                instrument crash
                            }
                            else
                            {
                                instrument hiHatOpen
                            }
                        }
                    }
                }

                instrument bassDrum
                {
                    rhythm 4 _8. 16 _16 16 8 _4
                }

                instrument snare
                {
                    rhythm _4 8. _16 _4 4
                }
            }
        }
    }
}
```

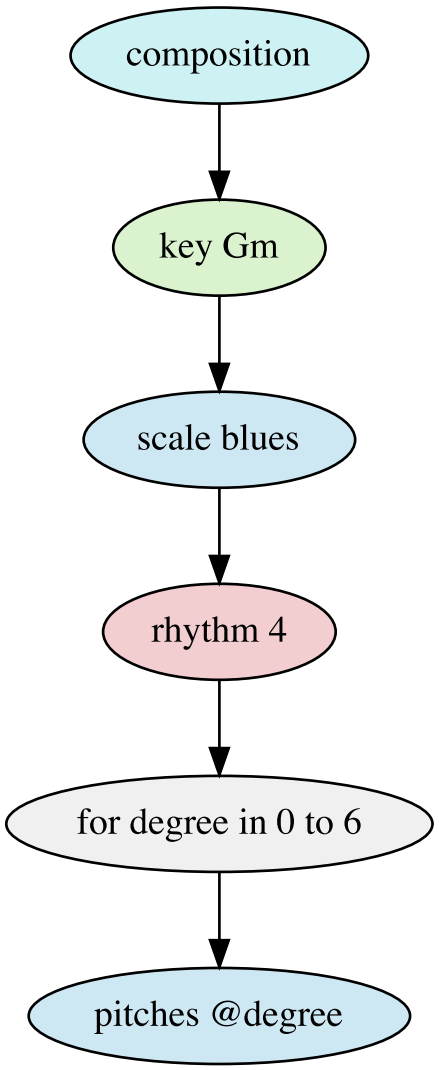
The contexts for the cymbals are specified conditionally in this context model. A condition based on the current repetition counts of an outer and an inner `repeat` control structure is specified. It evaluates to true if both the outer and inner repetition count is 1. If this is the case, a crash cymbal is used as instrument context. In all other cases, the open hi-hats are played. In the

two measures shown in in the drum part, it can be seen that the condition evaluates to `true` only in the first measure on the first beat, on which a crash cymbal is played. On all other beats, especially on the first beat in the second measure, an open hi-hat is played because the outer repetition count evaluates to 2 in the second measure.

Condition expressions can be based on arbitrary variables defined in any context nodes which are hierarchically placed above the current condition node. Notably, results of function calls can be used to create dynamically modeled compositions using conditional contexts. Refer to section Function Calls for more details.

Iterations

Iterations are used to create loops in which musical material is iteratively modified. The control structure resembles `for` loops in general purpose programming languages. Iterations define a control variable which typically changes its value in every loop iteration. The following model and the corresponding code demonstrate an iteration producing a G minor blues scale, which was already introduced in section Parallel Intervals.



The language representation of this model is:

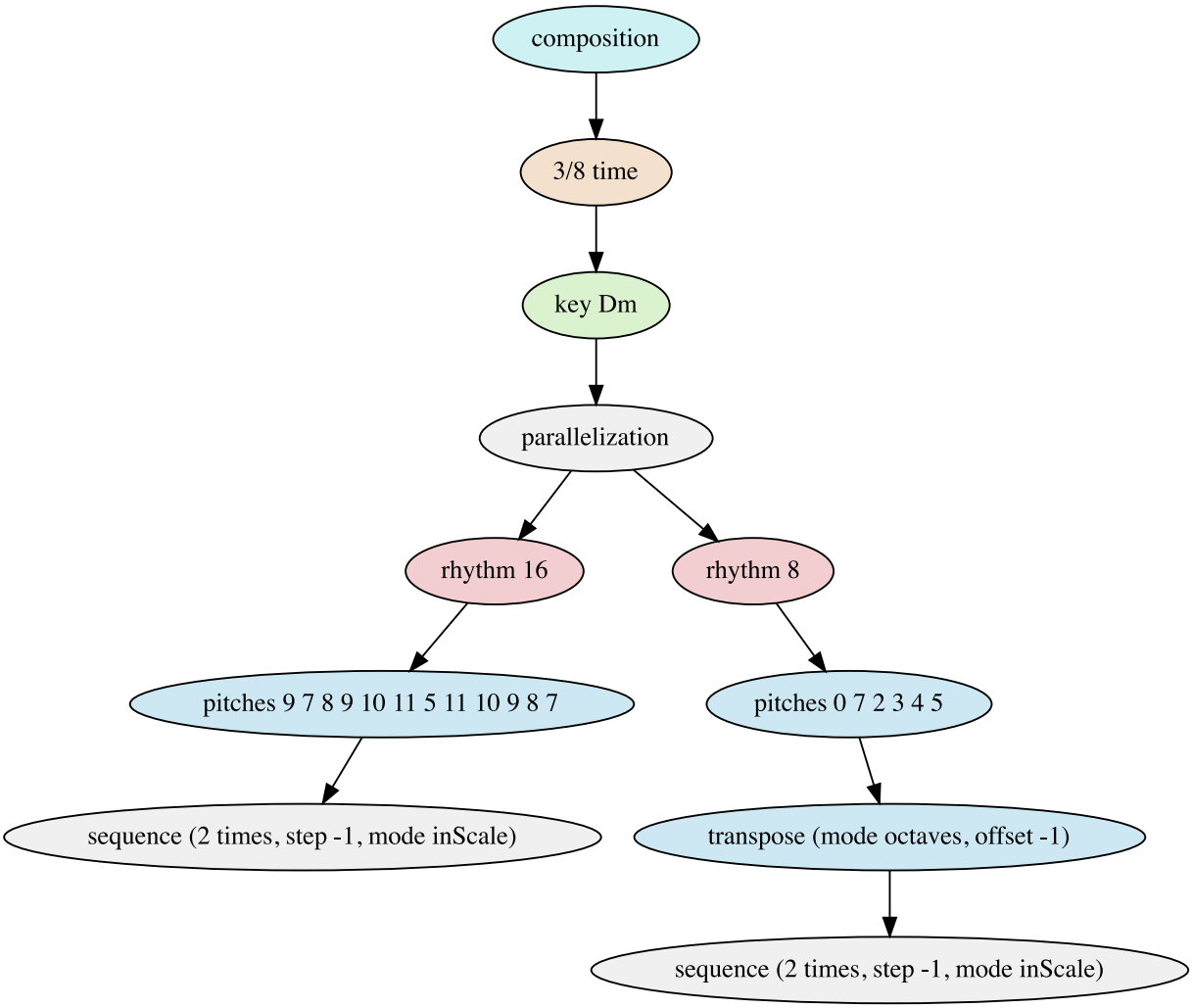
```
composition
{
    key Gm
    {
        scale blues
        {
            rhythm 4
            {
                for degree in 0 to 6
                {
```

```
                    pitches @degree
                }
            }
        }
    }
}
```

Also refer to section Rhythmic Displacements, in which a rhythmic pattern is iteratively displaced using a corresponding control structure and a suitable rhythmic modifier.

Sequences

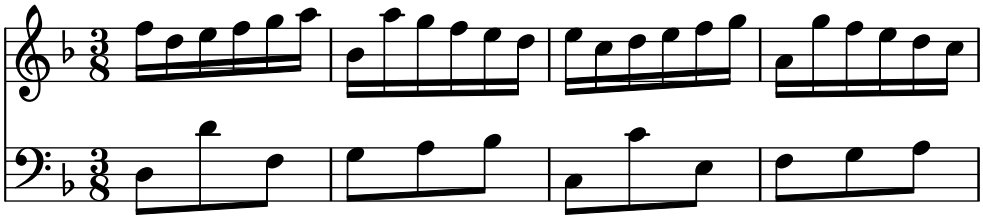
MPS provides a separate control structure for melodic sequences. Technically, melodic sequences are translated to an iteration with nested transpositions. The following context tree model represents a sequence from J.S. Bach's *Invention No. 4 in D minor, BWV 775*:



The model can be syntactically represented as follows:

```
composition
{
    time 3/8, key Dm
    {
        parallel
        {
            rhythm 16
            {
                pitches 9 7 8 9 10 11 5 11 10 9 8 7
                {
                    sequence 2 times step -1 mode inScale
                }
            }
            rhythm 8
            {
                pitches 0 7 2 3 4 5, transpose mode octaves -1
                {
                    sequence 2 times step -1 mode inScale
                }
            }
        }
    }
}
```

The model produces the following score:



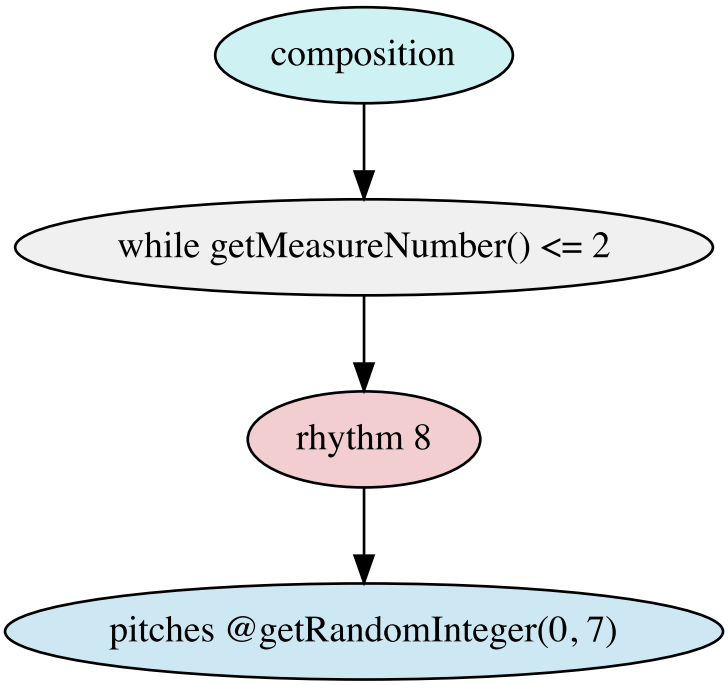
Sequence control structures are applied both to the right hand and the left hand voice. Both sequence control structures are applied twice (`2 times`). In the first iteration, the specified pitches are adopted without modification. In the second iteration, the pitches are transposed one step down. Consequently, the scale degrees of both voices are diatonically transposed down in parallel. Refer to the following table for detailed parameter descriptions.

Parameter	Description
`times`	Specifies how often the sequence is repeated.
`step`	Defines the offset of the iteratively applied transposition. The unit of this expression is defined by the `mode` parameter.

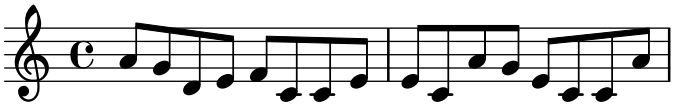
`mode`	Defines the unit of the `interval` expression. Three modes are available: `absolute` for semitone-based transpositions, `inScale` to perform transpositions of scale degrees and `octaves` for octave translations. If the parameter is not specified, the default `absolute` will be used.

While-Loops

The contents of while-loops are applied as long as a specified condition is fulfilled. An example is demonstrated in the following model:



A possible result is shown below:



The syntax representation of this model is:

```
composition
```

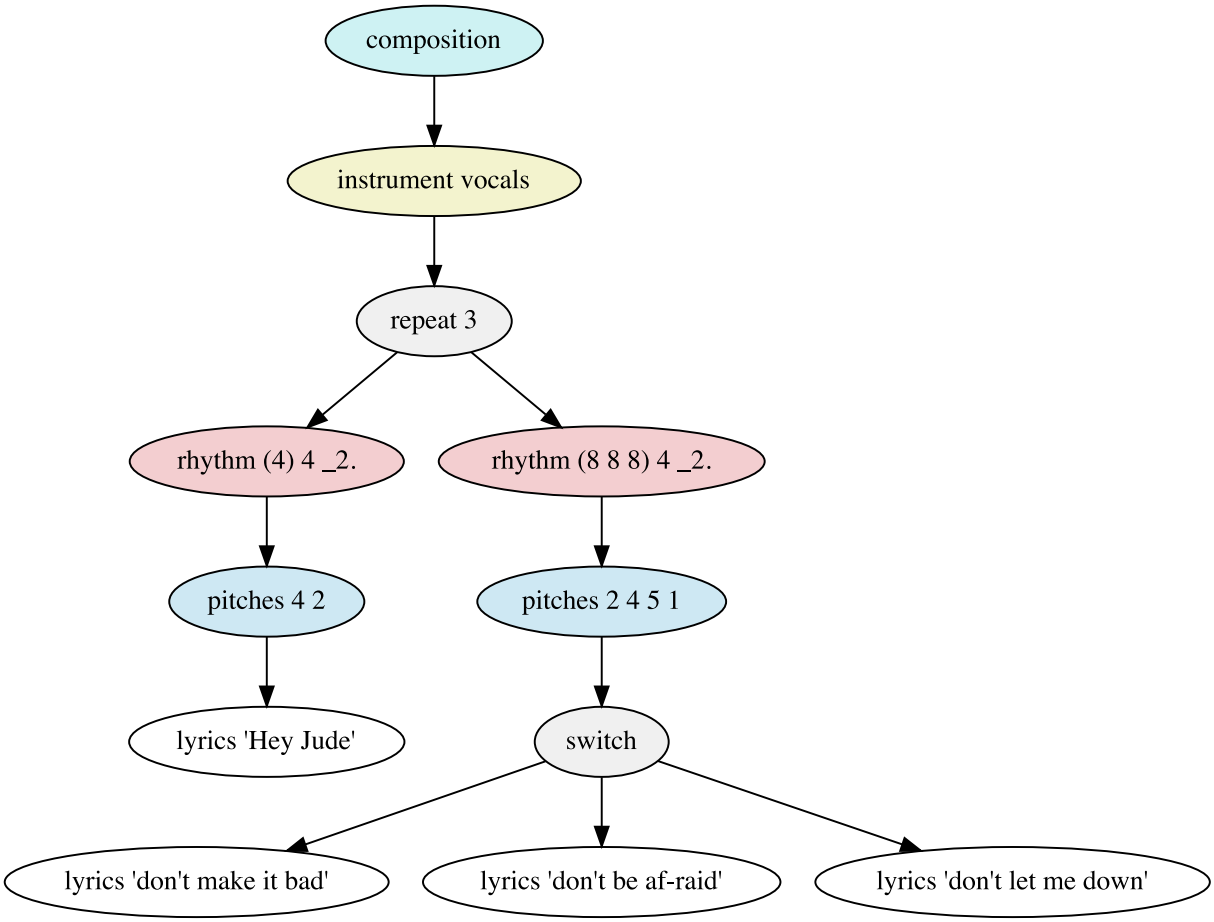
```
{
    while getMeasureNumber() <= 2
    {
        rhythm 8
        {
            pitches @getRandomInteger(0, 7)
        }
    }
}
```

The loop is applied while the measure number is less than or equal to 2 (i.e. in the first two measures). The current measure number can be retrieved using the function `getMeasureNumber()`. Pitches are chosen randomly using another function call to `getRandomInteger()`. Refer to section Function Calls for more details.

Switches

This control structure selects and processes only one of the specified child tree branches for each invocation. If the structure is encountered again (e.g. due to a repeat), the next child branch is processed. If no more child branches are available, processing continues from the first child branch again.

An example is provided in the following context tree model, in which the same melody is repeated three times. The switch control structure applies three different lyrics contexts for each loop iteration. Consequently, each time the switch is encountered, other lyrics are produced in the right subtree.

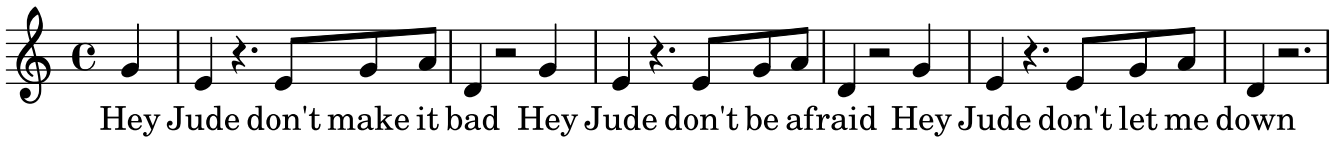


The corresponding language representation is:

```
composition
{
    instrument vocals
    {
        repeat 3
        {
            rhythm (4) 4 _2.
            {
                pitches 4 2
                {
                    lyrics "Hey Jude"
                }
            }
            rhythm (8 8 8) 4 _2.
            {
                pitches 2 4 5 1
```

```
                {
                    switch
                    {
                        lyrics "don't make it bad"
                        lyrics "don't be af-raid"
                        lyrics "don't let me down"
                    }
                }
            }
        }
    }
}
```

The model produces the following score:



It is also possible to define other non-consecutive processing sequences. This is done by specifying a so called *child index sequence*, as shown below:

```
switch childIndexSequence 0 0 1
```

The previously specified switch will process the first child branch twice, followed by the second child branch. If invoked again, processing will start over at the beginning of the custom sequence.

Expressions

Expressions are used to represent dynamically computable parameters in context tree models. These are especially useful for algorithmic composition, in which certain musical parameters are computed based on mathematical rules. MPS uses a custom expression language supporting logical and arithmetic expressions with variables and function calls.

Literals

A basic unit of information in the expression language is given in the form of literals. Refer to the following table containing a summary of available literal types.

Type	Description	Internal Type
`boolean`	Boolean value. Permitted literals are `true` and `false`.	`boolean`
`integer`	Integer number with optional negative sign, such as `42`, `-23` or `0`.	`int`

`float`	Floating point number with optional negative sign, such as `3.1415` or `-2.1`.	`double`
`fraction`	Fraction represented by an integer numerator and integer denominator, for instance `1/4`. Arithmetic divisions automatically result in a fraction if both operands are integer numbers.	Fraction
`string`	Represents a sequence of zero or more characters encoded in UTF-16.	`java.lang.String`

Operators

The system supports boolean operators, comparison operators and arithmetic operators. The operators are listed in the following table ordered by operator priority, from highest to lowest precedence.

Operator	Description
`!`	Unary boolean negation operator. For example, `!true` evaluates to `false`.
`-`	Unary arithmetic negation. For example, `-(2+1)` evaluates to `-3`.
`*`	Arithmetic multiplication
`/`	Arithmetic division. Results in a fraction if both operands are integer numbers.
`%`	Modulo operator
`+`	Arithmetic addition. May also be used to concatenate strings.
`-`	Arithmetic subtraction
`==`	Evaluates to `true` if the left operand is equal to the right operand.
`!=`	Evaluates to `true` if the left operand is not equal to the right operand.
`<`	Evaluates to `true` if the left operand is less than the right operand.
`>`	Evaluates to `true` if the left operand is greater than the right operand.
`<=`	Evaluates to `true` if the left operand is equal to or less than the right operand.

`>=`	Evaluates to `true` if the left operand is equal to or greater than the right operand.
`and`	Boolean *and* operator. Result of the expression is `true` if and only if both operands evaluate to `true`.
`or`	Boolean *or* operator. Result of the expression is `true` if at least one of the operands evaluates to `true`.

Parentheses may be used for custom operator prioritization, for example:

```
(2 + 3) * 4
```

In the previous expression, the term `2+3` is evaluated first and the result is multiplied with `4`. If no parentheses would be used, `3*4` would be evaluated first due to higher precedence of the multiplication operator.

Type Conversions

Expressions are dynamically casted if required. For example, to evaluate the following expression, several dynamic type casts are applied.

```
1 + 0.7 > 3/4 and !(n % 2)
```

To sum `1 + 0.7`, `1` is implicitly converted to a floating point number. To evaluate the comparison `1.7 > 3/4`, 1.7 is automatically converted to the fraction `17/10`. The result of the left-hand comparison `17/10 > 3/4` yields `true`. The modulo operation on the right hand side results in the remainder of n being divided by 2. The remainder is wrapped in a boolean negation. This implies that the remainder must implicitly be cast to a boolean expression. It evaluates to `false` if the remainder is equal to zero and to `true` otherwise. The boolean result of this implicit cast is negated and then used as right operand for the *and* conjunction. The right hand side of the *and*-operator can also be read as: „if n is dividable by 2”. Refer to following table for an overview of implicit type conversion rules.

Type 1	**Type 2**	**Resulting Type**
`boolean`	`integer`	`integer`
`boolean`	`float`	`float`
`boolean`	`fraction`	`fraction`
`boolean`	`string`	`string`
`integer`	`float`	`float`
`integer`	`fraction`	`fraction`
`integer`	`string`	`string`
`float`	`fraction`	`fraction`
`float`	`string`	`string`
`fraction`	`string`	`string`

The following table specifies the applied transformations. Source types are listed on the left, target types are listed in the columns on top.

	boolean	**integer**	**float**	**fraction**	**string**
boolean	-	`false` ⇒ 0, `true` ⇒ 1	`false` ⇒ 0.0, `true` ⇒ 1.0	`false` ⇒ `0/1`, `true` ⇒ `1/1`	`false` ⇒ „false”, `true` ⇒ "true"
integer	`false` if equal to `0`, `true` otherwise	-	as specified by doubleValue	`n/1`	as specified by valueOf
float	`false` if equal to `0.0`, `true` otherwise	Nearest integer below the value of the floating point number	-	Nearest computable fraction as specified by Fraction	as specified by valueOf
fraction	`false` if fraction is equal to `0/1`, `true` otherwise	Whole number part of the fraction as specified by intValue	as specified by doubleValue	-	as specified by toString
string	`false` if string is empty, `true` otherwise	as specified by parseInt	as specified by parseDouble	Supported if string contains two integer numbers separated by a slash (`/`) or a single integer number	-

Function Calls

Functions are used to dynamically retrieve musical context information. They are evaluated during the compilation process (see section Rendering Context Layer Model Representations). The returned values depend on the given parameters, the stream context and the temporal context in which they are invoked. Refer to the following table for an overview of available functions.

Signature	**Return Type**	**Description**
`chordsAvailable()`	`boolean`	Returns `true` if context harmonies are available in the current context, `false` otherwise.
`getRootNote()`	`NoteReference`	Returns the root note of the current context harmony.
`getBassNote()`	`NoteReference`	Returns the bass note of the current context harmony,

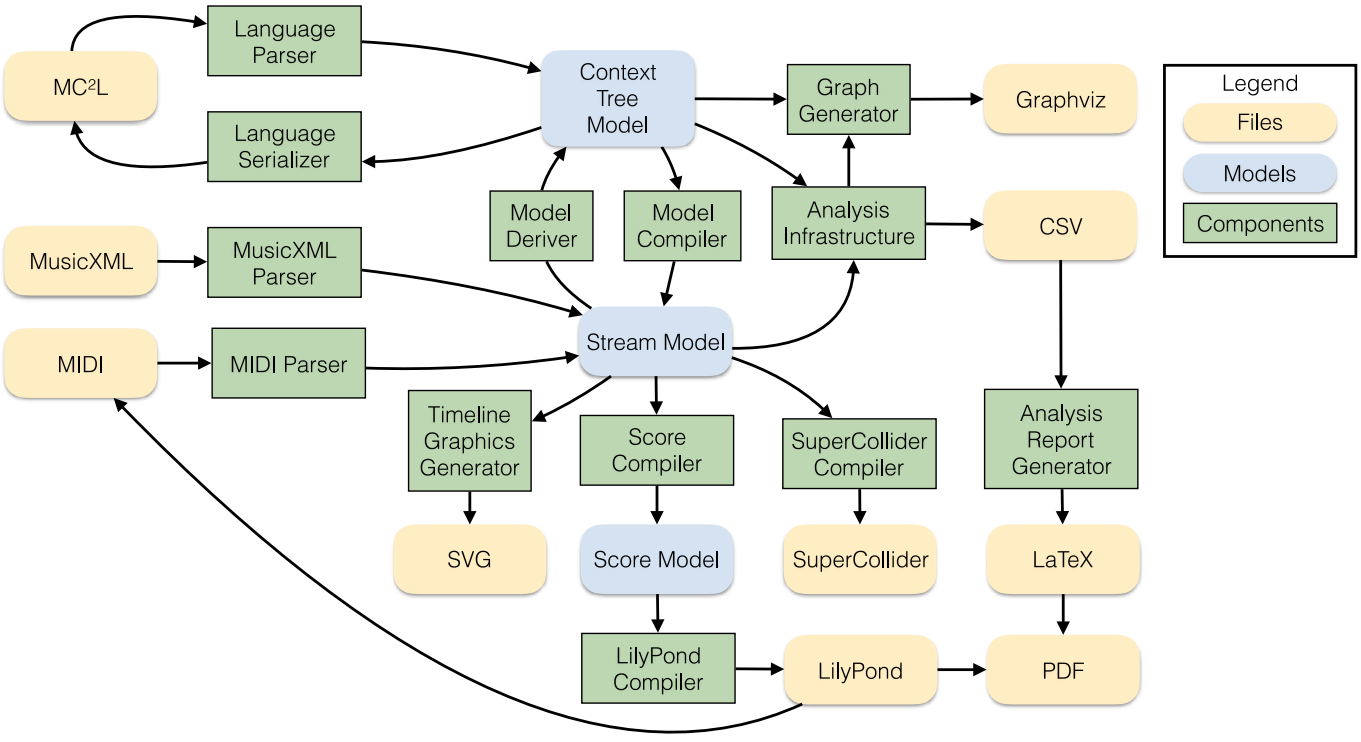
		which can in some cases be different from the root note.
`getRandomBoolean()`	`boolean`	Returns a random boolean value, i.e. `true` or `false`.
`getRandomInteger(min, max)`	`integer`	Returns a random integer value between `min` (inclusive) and `max` (exclusive).
`getRandomDouble(min, max)`	`double`	Returns a random double value between `min` (inclusive) and `max` (exclusive).
`getTime()`	`fraction`	Returns the current time in the current stream in terms of note duration (e.g. after a quarter note, the elapsed time is `1/4`). Refer to section Time Model for more details.
`getTimeSignature()`	`TimeSignature`	Returns the current time signature.
`isInFragmentContext(string)`	`boolean`	Returns `true` if the current context stack contains the fragment with the given name, `false` otherwise.

Music Transformation and Visualization

Music Processing Suite offers a transformation infrastructure to convert a number of music representation formats, including:

- `MIDI` files
- `MusicXML` files
- `MC3L` files, text files containing composition tree models specified using the domain-specific MPS composition language
- In-memory context tree composition models
- In-memory context layer composition models
- `DOT` representations of context tree models
- `SVG` representations of context layer models
- `LilyPond` score markup
- `PDF` scores
- `SuperCollider` code
- Music analysis data in the form of `CSV` files
- Music analysis reports in the form of `PDF` files

Refer to the following diagram for an overview:



Most of the transformations are covered in this chapter. For transformations regarding music analysis, refer to the corresponding chapter.

Rendering Context Tree Model Visualizations

Visual representations of context tree models can be generated by clicking the button

in the toolbar at the top of the application. The action is applicable to `.mcl` files, which must be opened in an editor or selected before the button is clicked.

The resulting graph is stored as a `.dot` file with the file name `<source file>_TreeModel.svg`, which can be processed with Graphviz. If you want to open the resulting graphs, make sure Graphviz is installed and configured as described in section Graphviz Installation.

For rendering options, see section Visualization Options.

Rendering Context Layer Model Representations

To transform a musical composition to a context layer model representation, simply select a source file and click the button

in the toolbar at the top of the application.

Files with the following extensions can be converted to a context layer model:

- MIDI files with `.mid` or `.midi` extension
- `.xml` files with MusicXML content and `.mxl` files (compressed MusicXML)
- Files with `.mcl` extension containing context tree composition models in the domain-specific language of MPS

In case of `.mcl` files, the conversion can also be invoked if the file is currently opened in an editor which is currently focused.

As a result, an `.svg` file with the name `<source file>_StreamModel.svg` will be created next to the source file. It will automatically be opened using a suitable application or editor. To configure a custom application to open `.svg` files, refer to the official Eclipse documentation.

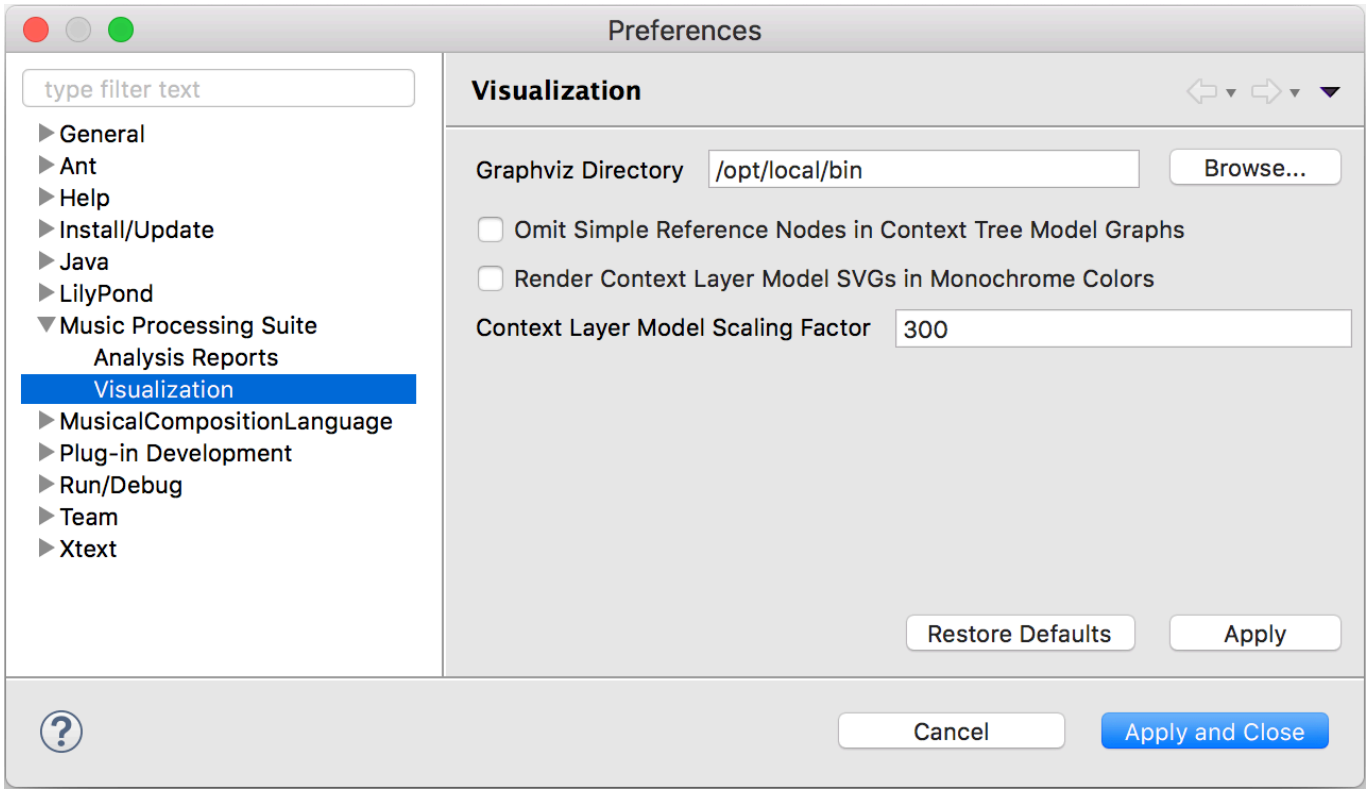
Refer to the following section for rendering options.

Visualization Options

To adjust visualization options for context layer models and context layer models, follow these steps:

1. Open the MPS Preferences
   1. On Windows and Linux: In the menu bar, choose Window → Preferences
   2. On Mac: In the menu bar, choose Music Processing Suite (application menu) → Preferences
2. Navigate to Music Processing Suite → Visualization

The following preference page will be shown.



The following visualization options are available:

Parameter	Description
Omit Simple Reference Nodes in Context Tree Model Graphs	By default, references are visualized by means of a reference node and a dotted line to the referenced node. If this option is activated, only the dotted line will be displayed. This is only possible if the reference node does not contain additional child nodes.
Render Context Layer Model SVGs in Monochrome Colors	Uses gray shades for stream model visualizations.
Context Layer Model Scaling Factor	Defines the number of horizontal graphical units used to represent a whole note. Increase this value if labels overlap their container bounds in stream model SVG visualizations.

Transforming Compositions to Scores

MPS supports the generation of musical scores in `.pdf` and `.midi` format for various input formats. The music notation software LilyPond is used for this purpose. Before using this feature, make sure that LilyPond is installed and configured as described in section LilyPond Installation. Score generation is triggered using the button

♫
in the toolbar at the top of the application.

The following input file types are supported:

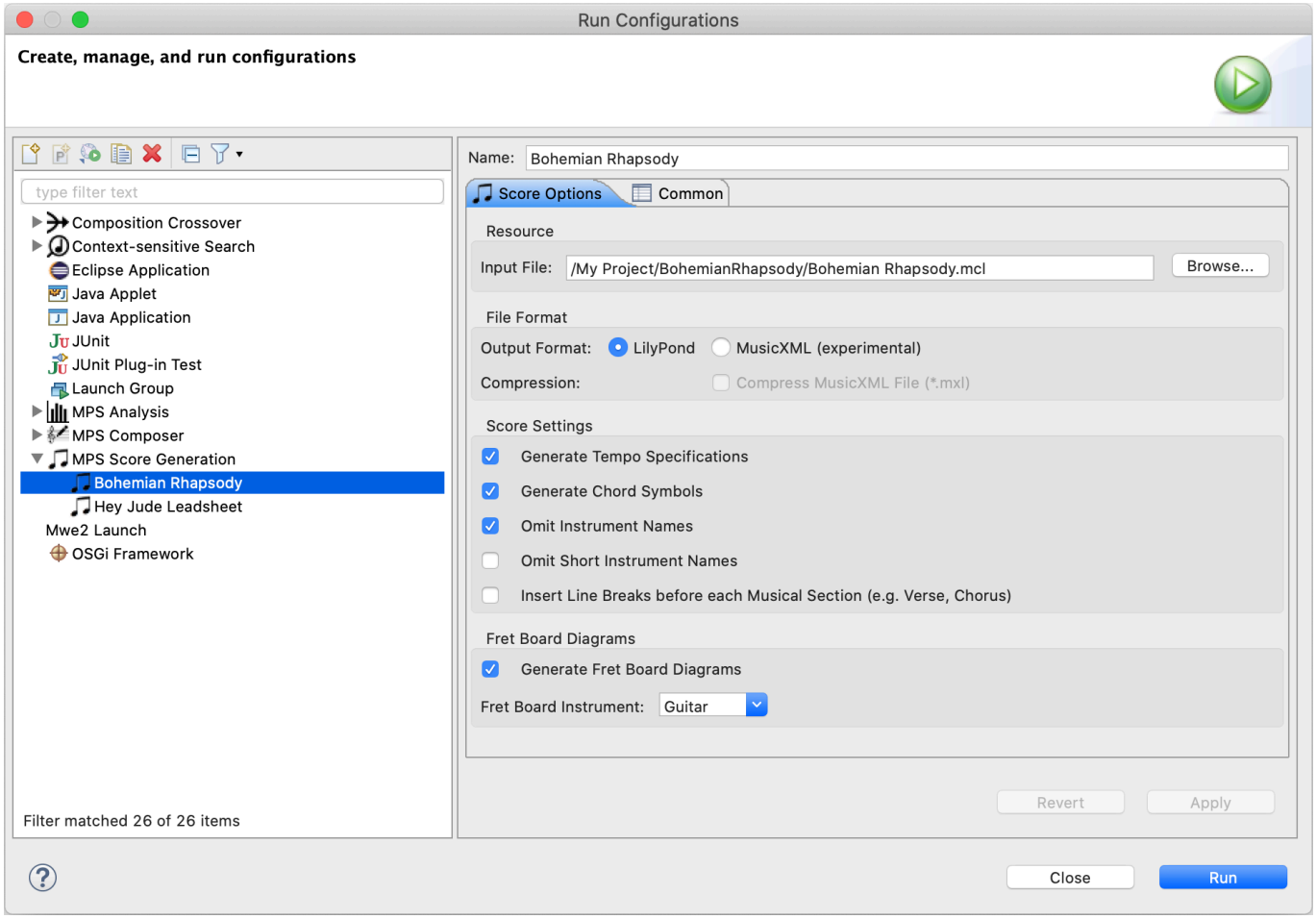
- MIDI files with `.mid` or `.midi` extension
- `.xml` files with MusicXML content and `.mxl` files (compressed MusicXML)
- Files with `.mcl` extension containing context tree composition models in the domain-specific language of MPS

In case of `.mcl` files, the conversion can also be invoked if the file is currently opened in an editor which is currently focused.

The system will internally convert the input file to a context layer model and derive a score representation of that model. Subsequently, the score representation will be exported in the form of a LilyPond file with the file name pattern `<source file base name>_Score.ly`. The Eclipse plugin Elysium, which is automatically installed with MPS, will take care of the compilation of the LilyPond file, resulting in `<source file base name>_Score.pdf` and `<source file base name>_Score.midi` files of the score.

Score Generation Options

When generating a score for the first time, a launch configuration is automatically generated and opened. Launch configurations allow the customization of the score generation for each source file individually. The launch configuration dialog looks like this:



After making the desired adjustments, click the *Apply* button and then click *Run* to generate the score.

To change launch configurations later, click the black arrow next to the button

in the toolbar and click *Run Configurations...*. Open the category *MPS Score Generation* and select the desired launch configuration.

The following options are available:

Parameter	**Description**
Output Format	Available output formats are LilyPond (`.ly` files) and MusicXML (`.musicxml` or `.mxl` files, depending on whether compression is enabled).
Generate Tempo Specifications	Indicates whether tempo specifications are generated at the beginning of the piece and when the tempo changes.
Generate Chord Symbols	If activated, chord symbols are generated above the staves.

Omit Instrument Names	Omits instrument names on the left hand side of the staves. Also removes indention at the beginning of the first staff or staff group.
Omit Short Instrument Names	Short instrument names are usually instrument abbreviations printed at the left hand side of staves after the very first staff or staff group, for example *Cl.* instead of *Clarinet*. Check this box to omit these.
Insert Line Breaks before each Musical Section	If set, labeled sections (e.g. Verse, Chorus) start in a new staff block.
Generate Fret Board Diagrams	If activated, fret board diagrams for the selected instrument are rendered above the staves illustrating the fingering of the respective chords.

Score launch configurations can be reused, edited and deleted at any time. Refer to section Launch Configurations for further details.

Transforming Compositions to SuperCollider

To convert a composition into executable code for the sound synthesis language SuperCollider, simply click the button

in the toolbar at the top of the application. A file named `<source file base name>.scd` will be created as a result, containing the code. It can be opened in SuperCollider or directly executed from MPS, as explained in the following section.

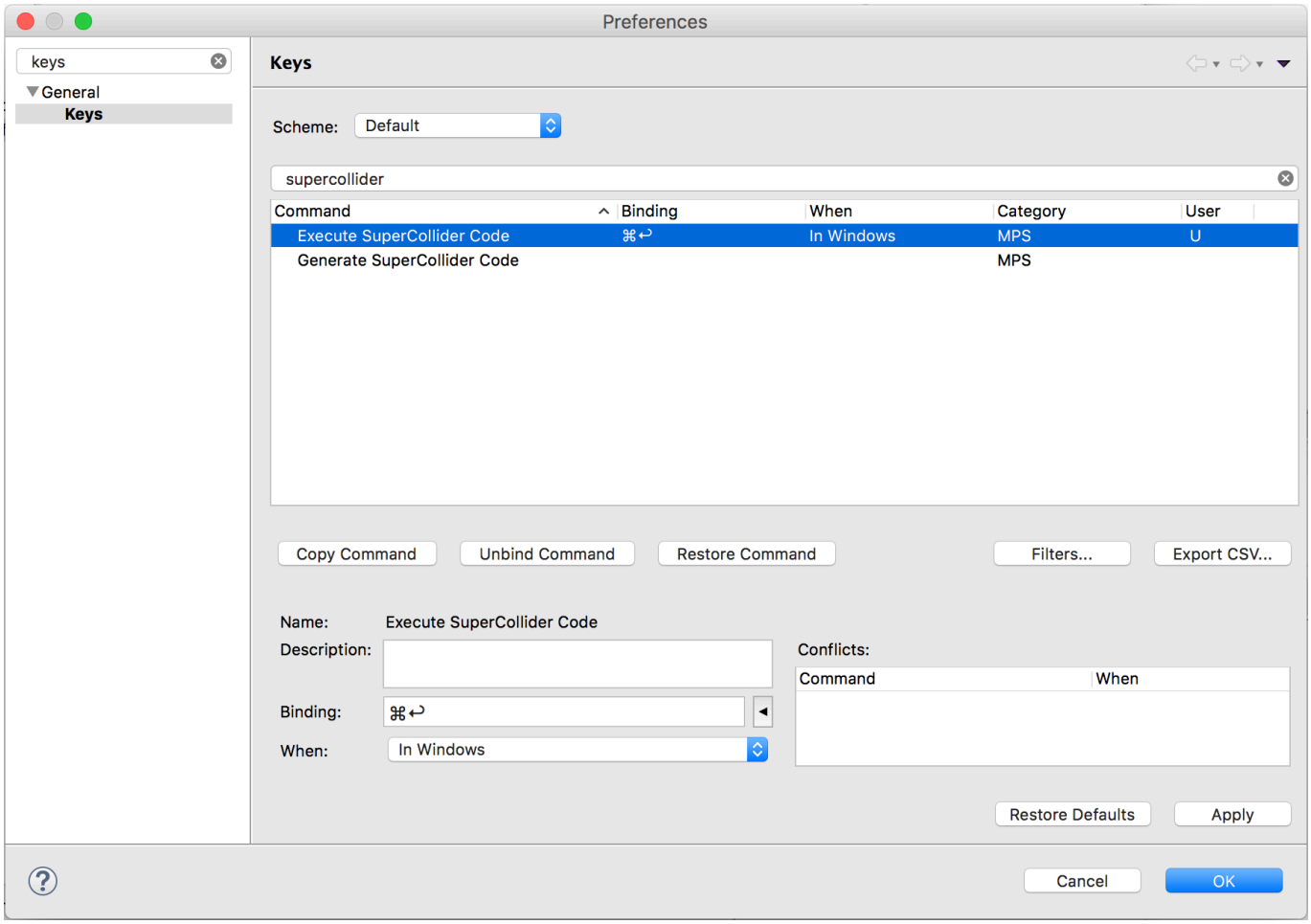
Executing SuperCollider Code

The generated SuperCollider code can directly be executed in a running SuperCollider instance. If you want to use this feature, a corresponding extension has to be installed in your SuperCollider `Extensions` directory. To install the MPS extension, copy the file `MusicProcessingSuiteInterface.scd` located in the directory `SuperCollider` of your installation root directory (which is wrapped into a directory named `Contents/Eclipse` in the `MPS.app` container on Mac OS X) to your SuperCollider `Extensions` directory. Refer to http://doc.sccode.org/Guides/UsingExtensions.html for help locating the `Extensions` directory.

Once the extension is installed (and SuperCollider restarted if already running), the resulting SuperCollider code can directly be executed by clicking the button

in the toolbar.

It is also possible to perform code execution with a key binding. It can be defined under *Preferences* → *General* → *Keys*. Search for the action *Execute SuperCollider Code* and define your own key binding. The recommended scope is *In Windows*. The configuration is shown in the following screenshot:



Deriving Context Tree Models

Context tree models can automatically be derived for existing compositions.
To invoke this functionality, select an input file and click the button

in the toolbar.

Supported input file types are:

- MIDI files with `.mid` or `.midi` extension
- `.xml` files with MusicXML content and `.mxl` files (compressed MusicXML)
- Also `.mcl` files (which already contain context tree models) can be given as input, because it can be interesting to see whether the algorithm is able to construct an even more compact model representation of a given context tree.

The resulting context tree model will be stored under the file name `<source file base name>_DerivedModel.mcl` next to the input file.

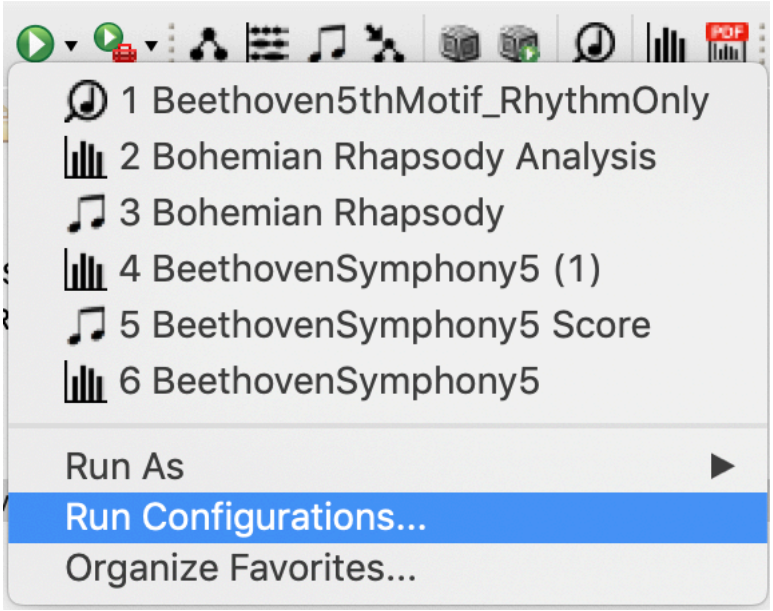
Launch Configurations

Eclipse stores configuration parameters for runnable processes in so called *Launch Configurations*. MPS creates default launch configurations for the following processes automatically:

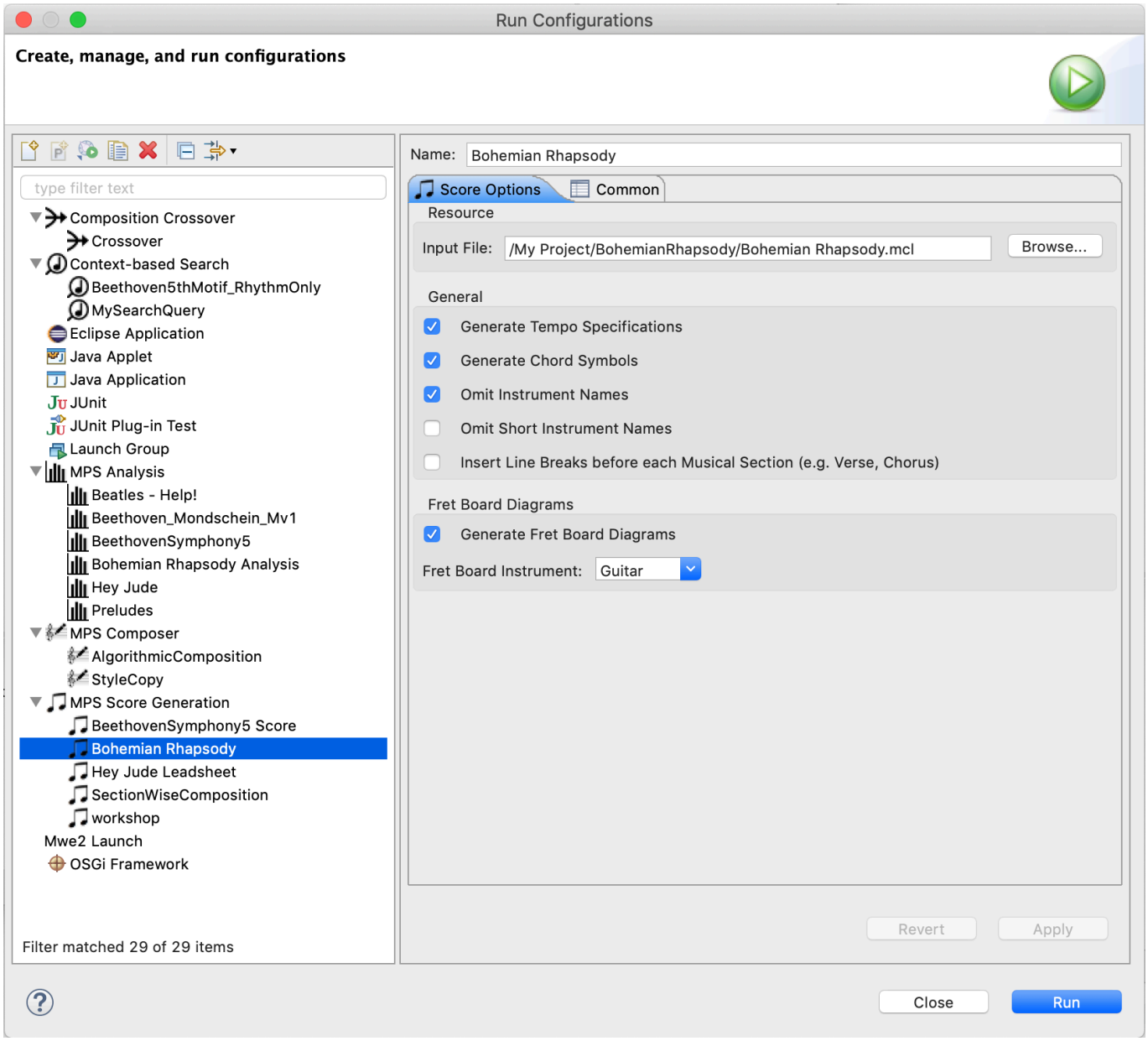
- Score Generation
- Context-sensitive Search
- Music Analysis
- Algorithmic Composition
- Composition Crossover

These launch configurations can be reused, edited and deleted at any time. To show available launch configurations, click the black arrow next to the button

in the toolbar and click *Run Configurations...*.



In the following dialog, all launch configurations are displayed grouped by launch types. All launch configurations can be run, edited and deleted. In the example below, a score generation launch configuration is shown.



Context-sensitive Search

MPS provides advanced context-sensitive search functionality to find musical material in a corpus of compositions. Musical search queries can be expressed in the domain-specific language introduced in chapter Composition Language and Context Tree Models. Conceptually, search queries are formulated in terms of a compositional fragment, which is transformed to a corresponding context layer model search pattern (see chapter Context Layer Models for more details) and matched against context layer model representations of the pieces in the corpus.

Formulating Search Queries

To search for the rhythm of Beethoven's 5th Symphony motif, for instance, the search query can be formulated as:

```
composition
{
    rhythm _8 8 8 8 2
}
```

To search for a combination of this rhythm with the scale degrees `4 4 4 2` in the key *C minor*, the query syntax could be:

```
composition
{
    key Cm, rhythm _8 8 8 8 2, pitches 4 4 4 2
}
```

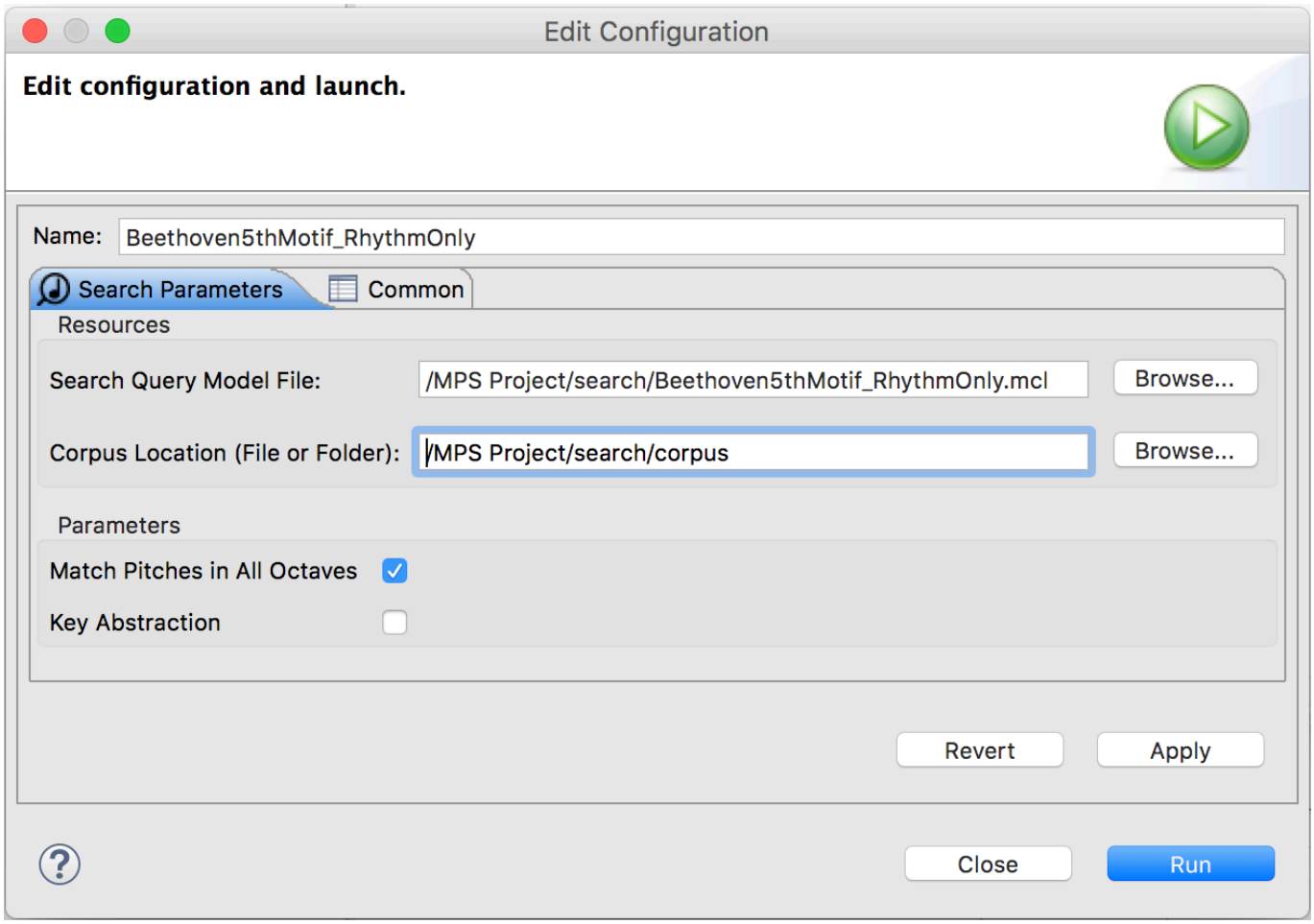
Performing Context-sensitive Search

After formulating a search query, the search process can be invoked by clicking the button

in the toolbar. The search query file must either be opened in the active editor or be selected in a navigator-based view. Alternatively, the search process can be initiated by invoking the context menu with a right click and choosing *Run As* $\rightarrow$ *Context-sensitive Search Query*.

Search Configuration

When a search query is executed for the first time, a launch configuration is automatically created and a dialog is opened in which further search parameters can be set:



In this dialog, the corpus location, i.e. a file or folder to be searched, must be specified. Refer to the following table for descriptions of other search parameters:

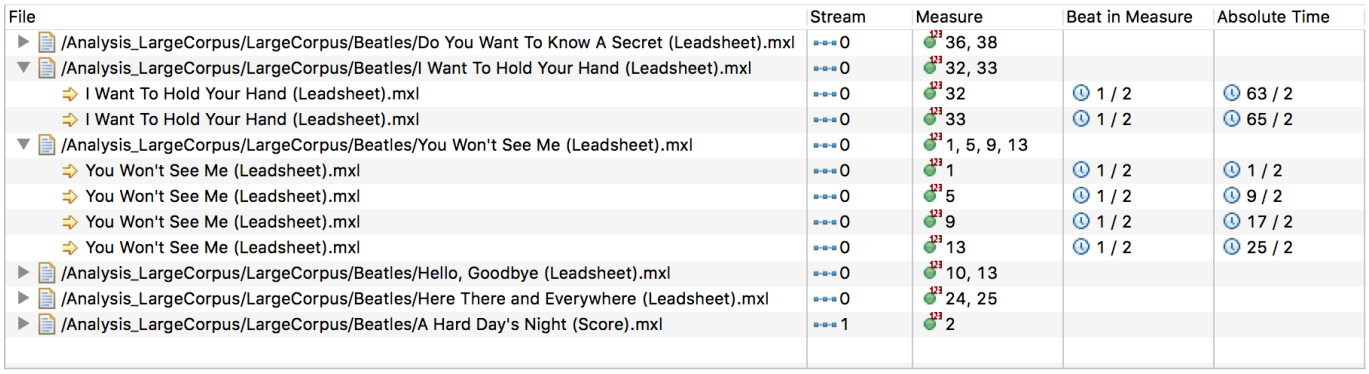
Parameter	Description
Match pitches in all octaves	Enables abstraction for octaves. For example, if the search query contains pitches in octave 4 (which is the middle octave), the same pitches will also be matched in other octaves.
Key Abstraction	This option will enable key context matching only based on the minor/major flag. For example, if the query contains a context specifying the key *C minor*, matches will be found in all minor keys.

Search launch configurations can be reused, edited and deleted at any time. Refer to section Launch Configurations for further details.

Search Result Presentation

Search results will be displayed in the *Search* view, which is located at the bottom of the MPS IDE by default. Search results are presented hierarchically in a table. For each search result,

the workspace-relative file path, stream numbers, measures, beats (i.e. zero-based point of time in the corresponding measures) and absolute time is displayed as shown in the following screenshot:



File	Stream	Measure	Beat in Measure	Absolute Time
/Analysis_LargeCorpus/LargeCorpus/Beatles/Do You Want To Know A Secret (Leadsheet).mxl	0	36, 38		
/Analysis_LargeCorpus/LargeCorpus/Beatles/I Want To Hold Your Hand (Leadsheet).mxl	0	32, 33		
I Want To Hold Your Hand (Leadsheet).mxl	0	32	1 / 2	63 / 2
I Want To Hold Your Hand (Leadsheet).mxl	0	33	1 / 2	65 / 2
/Analysis_LargeCorpus/LargeCorpus/Beatles/You Won't See Me (Leadsheet).mxl	0	1, 5, 9, 13		
You Won't See Me (Leadsheet).mxl	0	1	1 / 2	1 / 2
You Won't See Me (Leadsheet).mxl	0	5	1 / 2	9 / 2
You Won't See Me (Leadsheet).mxl	0	9	1 / 2	17 / 2
You Won't See Me (Leadsheet).mxl	0	13	1 / 2	25 / 2
/Analysis_LargeCorpus/LargeCorpus/Beatles/Hello, Goodbye (Leadsheet).mxl	0	10, 13		
/Analysis_LargeCorpus/LargeCorpus/Beatles/Here There and Everywhere (Leadsheet).mxl	0	24, 25		
/Analysis_LargeCorpus/LargeCorpus/Beatles/A Hard Day's Night (Score).mxl	1	2		

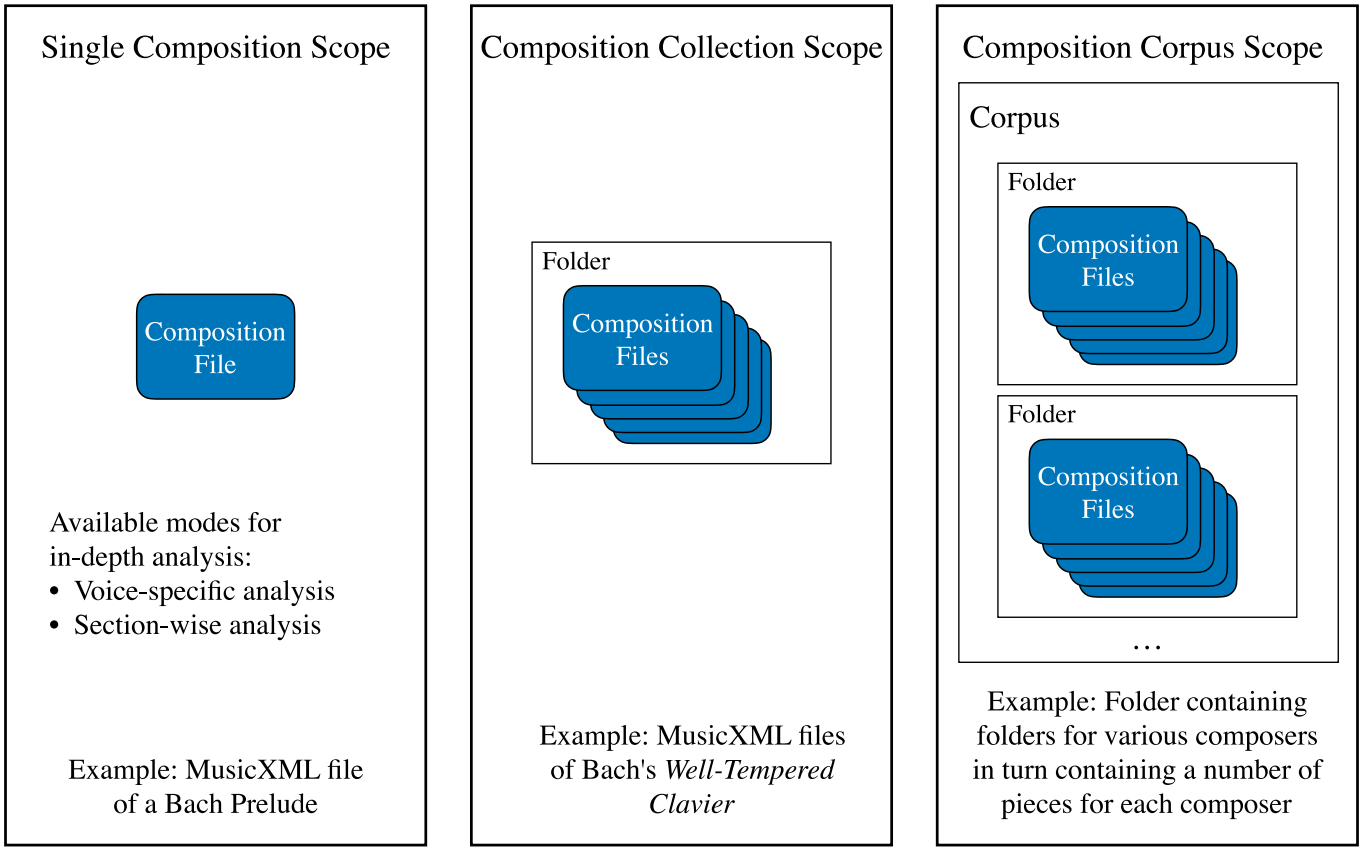
Music Analysis

An extensive music analysis framework was developed as part of MPS, which facilitates statistical and musicological analyses.

There are a number of motivations for the development of the MPS analysis tool:

- Due to the design of the underlying context-based model, new analysis methods can be developed which benefit from the available context information. Not only explicit information, but also implicit and contextual data can be taken into account and relations between individual context layers can be explored easily.
- Once certain analysis algorithms are developed, they can be applied not only to one musical piece, but also easily to hundreds or thousands of pieces. In this way, interesting results can be obtained by using the advantages of computers. In this way, human researches can be disburdened from laborious manual work.
- By using the processing capabilities of computers, statistical commonalities and distinctive features of musical compositions can be explored.
- MPS does not require the knowledge of programming skills. Users only have to provide `MIDI` and/or `MusicXML` files. The system produces output files in CSV format which can be opened with any regular spreadsheet application.
- MPS also generates comprehensible graphical plots and charts from the analysis results, which can be viewed directly in MPS or can be exported in the form of PDF analysis reports.
- By interpreting musical analysis results, insights can be gained into characteristic properties of individual pieces, composers and styles.

Analysis Scopes

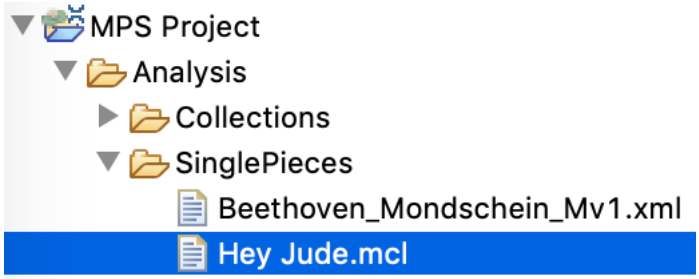


To analyze musical compositions, one or more files containing symbolic music data in `MIDI` or `MusicXML` format have to be provided. The analysis and export process is configurable. Different analysis scopes are possible. Either a single piece, a collection of pieces or a corpus of pieces can be analyzed.

The first musical analysis scope to be introduced is the single piece scope. It is used for gaining insights about a specific musical composition by performing either a global analysis or a voice-specific analysis of the piece. It is also possible to combine the aforementioned analysis modes. Furthermore musically meaningful sections can manually be marked in the score in order to perform section-wise analysis, which is optionally combinable with voice-wise analysis.

Collection analysis is suitable for comparing multiple musical pieces among each other. For this purpose, analysis results are shown next to each other in combined representations. Corpus analysis goes even one more step further, allowing to compare multiple collections of compositions, e.g. folders containing multiple compositions of individual composers.

Analyzing Music



To start analyzing music, simply select a file or folder and click the button

in the toolbar. Another possibility is invoking the context menu by using a right click and choosing *Run As* → *MPS Analysis Run*. If a file or folder is analyzed for the first time, a corresponding launch configuration will be opened. Adjust the configuration as desired and then click *Apply* and/or *Run* to start the analysis process. See section Configuring Music Analysis for more details.

The progress of the analysis process will be shown in the *Progress* view, which is located at the bottom of the application by default.

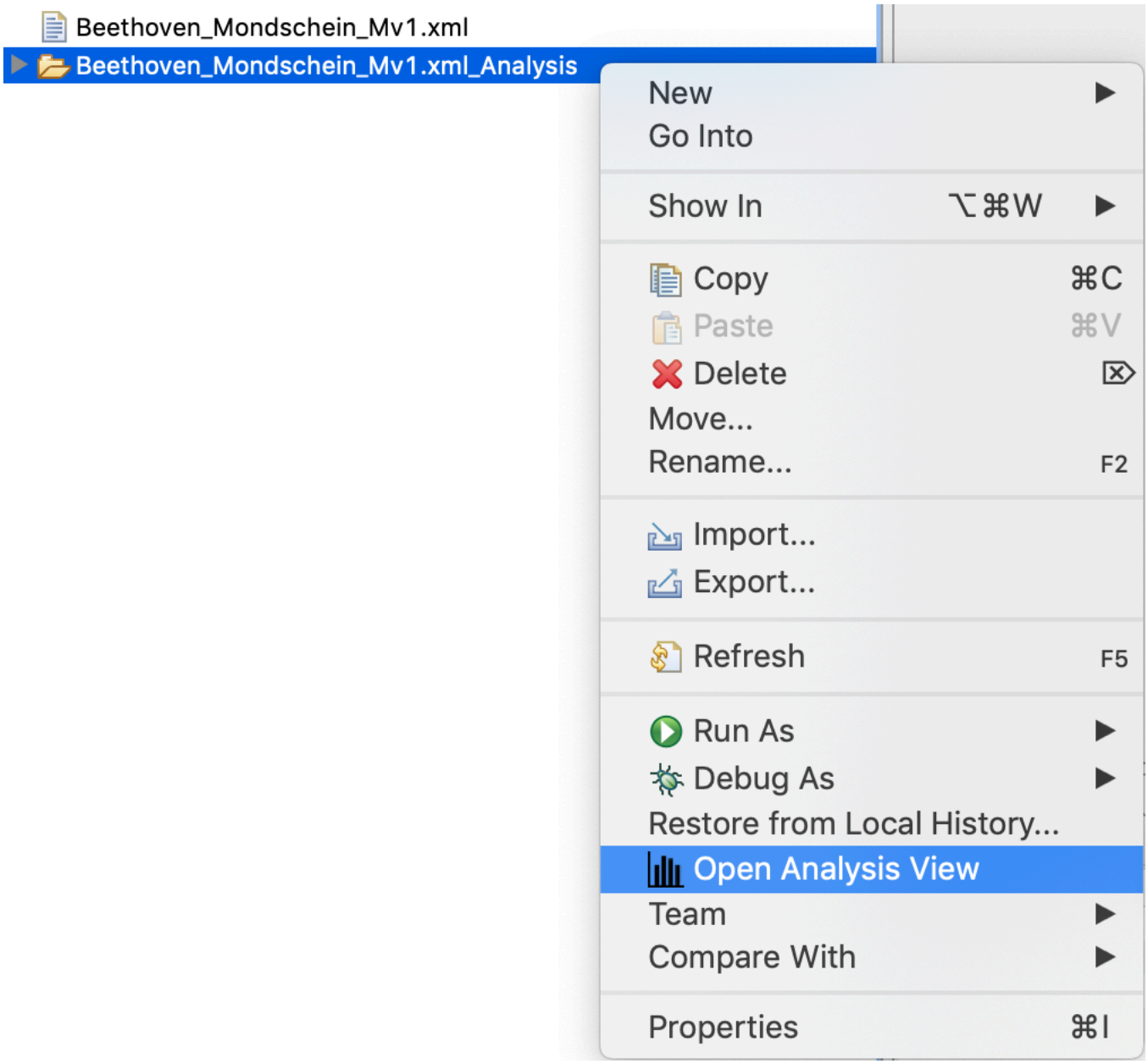
Folders with the name schema `<analyzed file or folder>_Analysis` will be created for each analyzed file or folder. The results will be stored in these folders in the form of CSV files. These contain column names in the first line/row and analysis result values in the following lines/rows.

Exploring Analysis Results

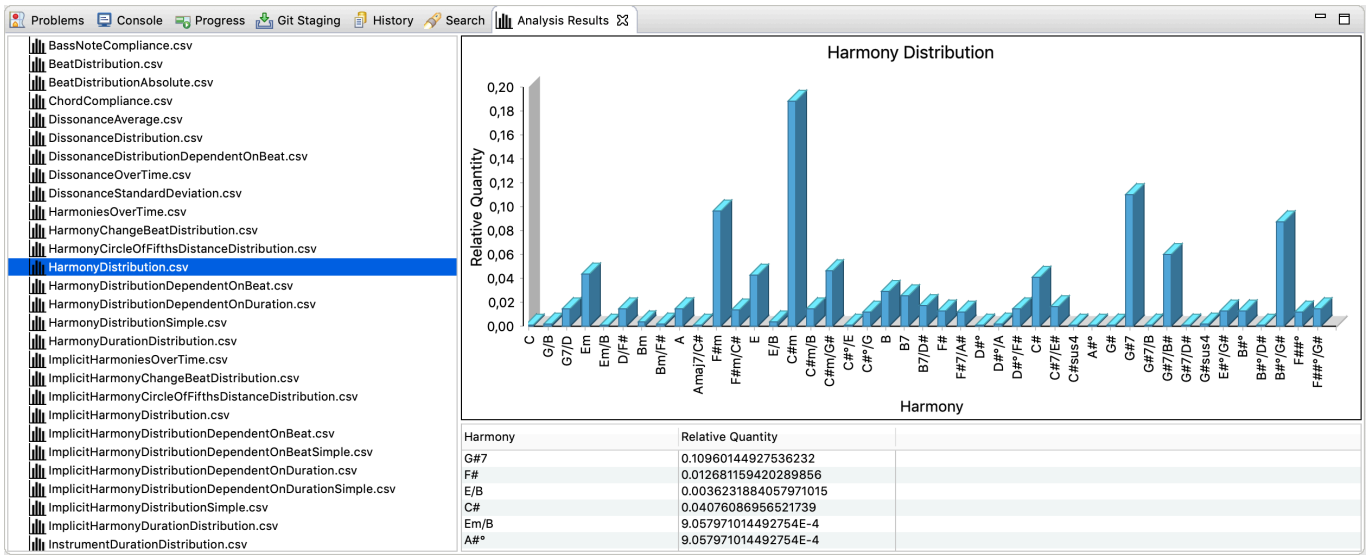
There are multiple options for viewing and exploring analysis results. One option is to view the raw analysis data directly or in a spreadsheet editor. To view raw analysis data, follow these steps:

- To view a CSV file in raw text format, right-click on a CSV file and choose *Open With* → *Text Editor*.
- To view a CSV file in a spreadsheet editor such as Excel, OpenOffice or Numbers, select *Open With* → *System Editor*. This will open the application which is associated with the `.csv` file extension in your operating system.

A more comfortable option is to use the Analysis Result Browser provided by MPS. To open it, right-click an analysis folder and and choose *Open Analysis View* from the context menu:



The analysis result view will be shown. To initialize the view with data, select an arbitrary folder containing analysis results. Once the view is associated with a data set, CSV files can be selected on the left hand side of the view. MPS will display the raw CSV data and a corresponding plot or graph depending on the content of the selected CSV file.



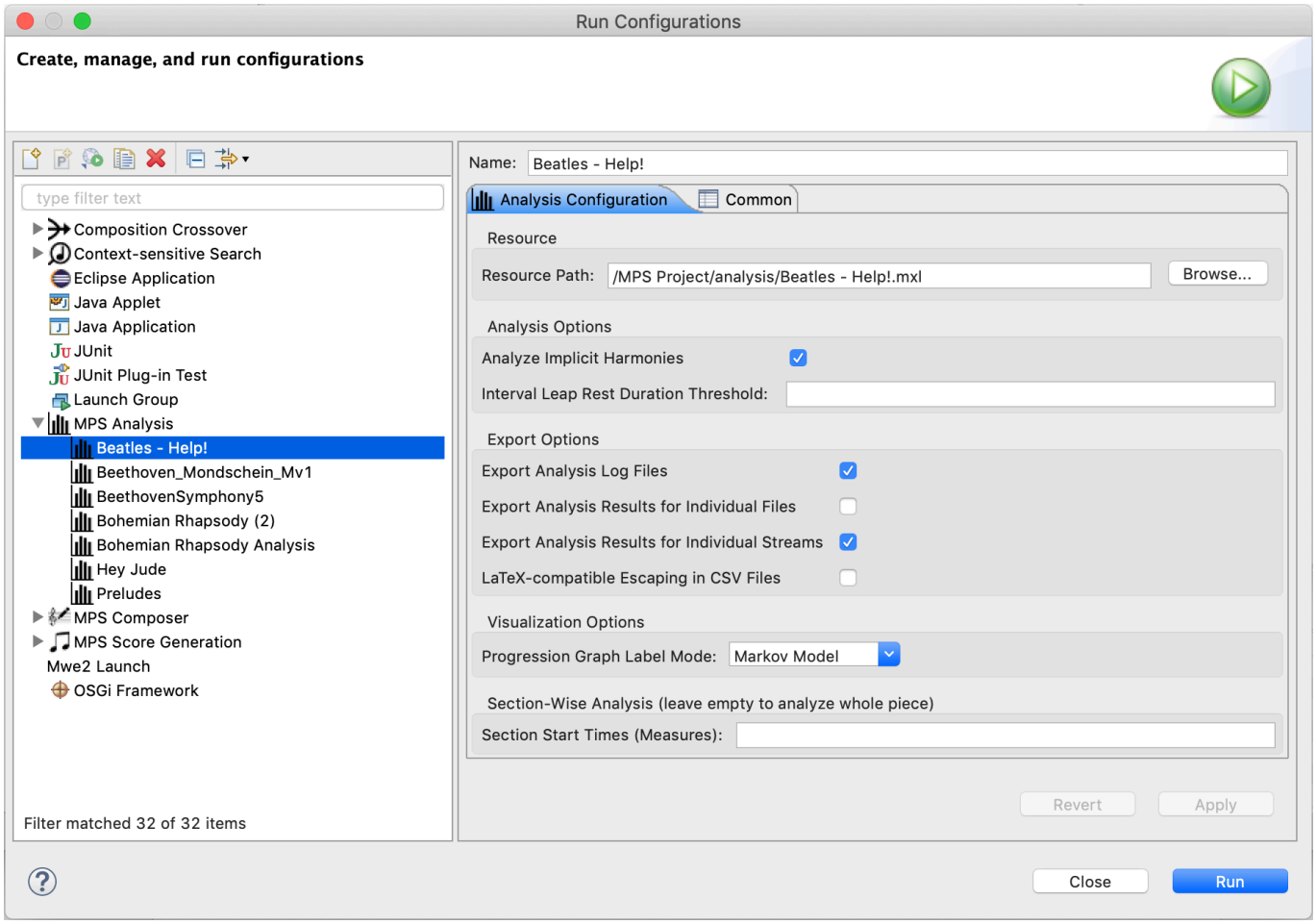
The generated charts can be exported as file. To export a chart, right-click on the graphic and select *Export Chart to File*. After specifying the destination, the file will be generated. Currently, the following image formats are supported:

- BMP
- JPEG
- PNG

Configuring Music Analysis

The analysis process can be configured by adjusting a so called run configuration.
A run configuration is automatically created when a file or folder is initially analyzed.
To change run configurations later, click the black arrow next to the button

in the toolbar and click *Run Configurations...*. The following window will open:



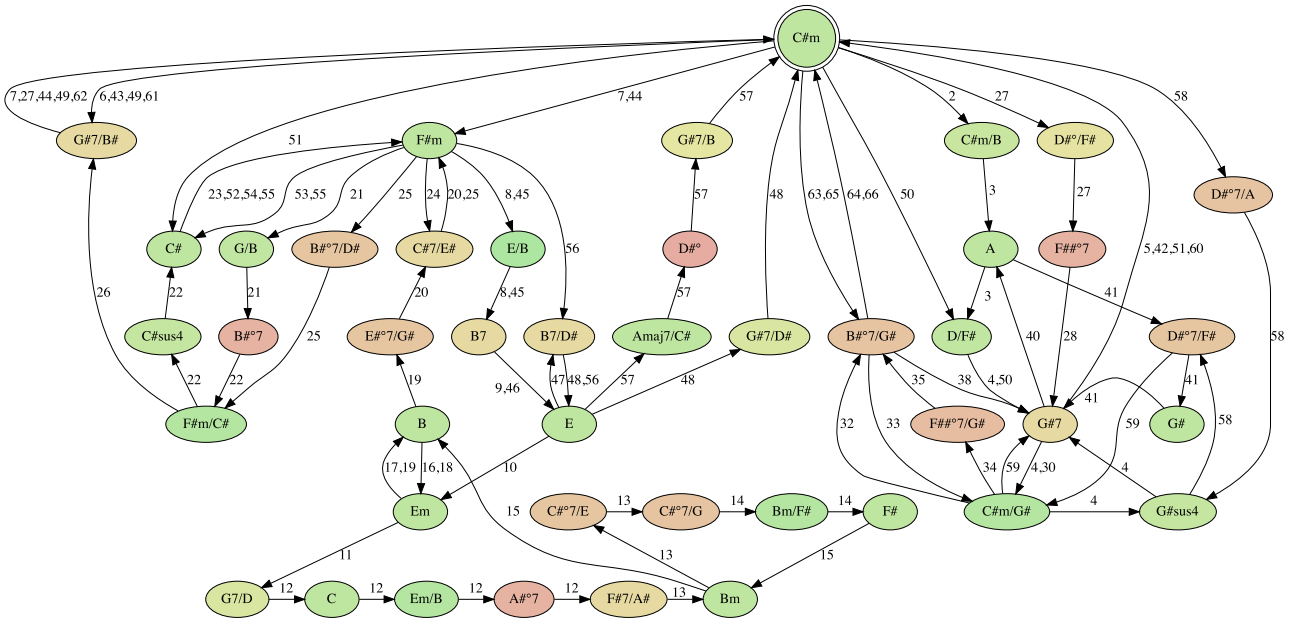
You can either search for an existing run configuration to modify or create a new configuration by clicking the button

.

The following parameters can be set:

Parameter	Description
Resource Path	Workspace-relative path to a file or folder to be analyzed.
Analyze Implicit Harmonies	Attempt to derive harmonies by analyzing simultaneously audible pitches in the piece.
Interval Leap Rest Duration Threshold	Specifies the impact of rests between notes for the interval leap detection algorithm. If the value is empty, every rest resets the algorithm, i.e. no interval leaps are detected across rests. In case the value is `0`, interval leaps are detected across all rests. If the value is any other positive fraction (e.g. `1/4`), the algorithm is reset in case rests longer than the given fraction are detected.

Export Analysis Log Files	If enabled, detailed log files are exported.
Export Analysis Results for Individual Files	If multiple files are analyzed (collection or corpus scope) and this option is enabled, detailed analysis results for each file are exported.
Export Analysis Results for Individual Streams	Indicates whether individual analysis results for each stream (voice) should be exported.
Progression Graph Label Mode	Configures how edges in progression graphs (e.g. rhythmic progressions, pitch progressions or harmonic progressions) should be labeled. *Occurrence Count* displays how often a progression was encountered. *Measure Numbers* shows the measure number in which the corresponding transition was detected. *Markov Model* displays the markov model probabilities for each transition. As an example, a harmonic progression graph is shown below with measure number labels. Refer to section Generating Progression Graphs for more details.
Section Start Times	Provide a comma-separated list of measure numbers if a single piece should be analyzed section-wise. The first measure does not need to be supplied. Example: `5,17,33`. If this field is left empty, the piece is considered to have one large section.



The figure shows the harmonic progression graph of Ludwig van Beethoven's *Piano Sonata No. 14 in C# minor, first movement*. The numbers specify the measures in which the corresponding chord change was detected. The colors of the chords encode the consonance or dissonance of the relevant chord (green corresponds to consonant and red to dissonant). The graph reveals which harmonic progressions are only used once and which are used multiple times. The latter are easily identifiable due to comma-separated enumerations of measure numbers in which the corresponding transitions occur.

Analysis launch configurations can be reused, edited and deleted at any time. Refer to section Launch Configurations for further details.

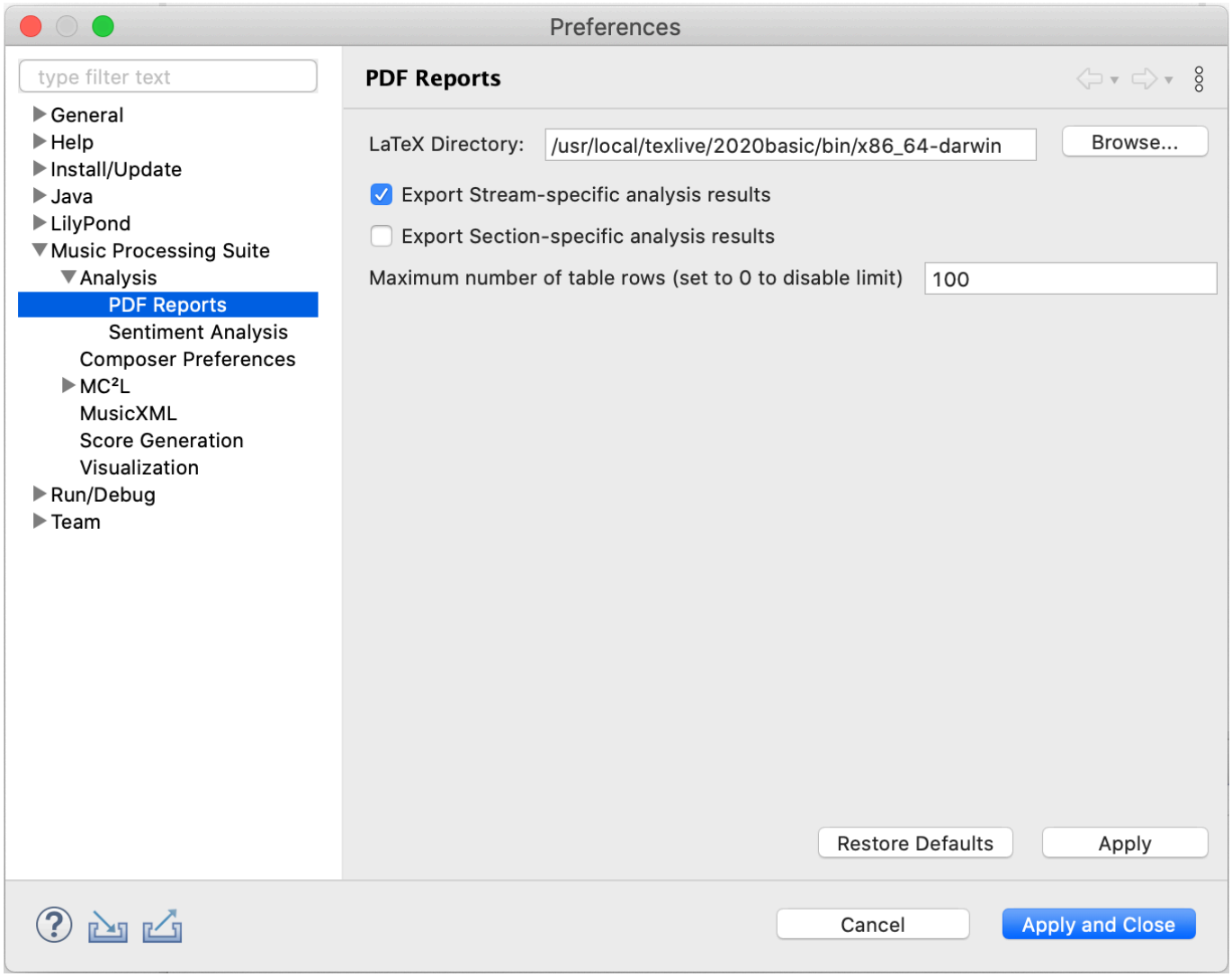
Generating Analysis PDF Reports

Analysis results can also be exported in the form of PDF reports. To invoke a report generation, simply select an analysis data folder of your choice and click the button 
in the toolbar. Note that you must have a LaTeX environment installed to render analysis reports, which is described in sections LaTeX Installation and LaTeX Configuration in the installation chapter. The PDF will be generated in a directory next the the analysis folder with the name `<Analysis Folder Name>_Report`.

Analysis Report PDF Settings

To configure the contents of analysis report PDFs, navigate the the corresponding preference page:

1. Open the MPS Preferences
   1. On Windows and Linux: In the menu bar, choose Window → Preferences
   2. On Mac: In the menu bar, choose Music Processing Suite (application menu) → Preferences
2. Navigate to Music Processing Suite → Analysis Reports



The following settings are available:

Parameter	Description
LaTeX Directory	Path to a LaTeX installation containing the LaTeX executables.
Export Stream-specific Results	Configures whether the PDF report should contain analysis results for individual streams.
Export Stream-specific Results	If set, the PDF report will contain individual analysis results for each section. This only works if sections were defined and a section-specific analysis was performed.
Maximum number of table rows	Sets the maximum number of table rows to be displayed in analysis report PDFs. The default is 100. If set to 0, all rows are printed.

Generating Progression Graphs

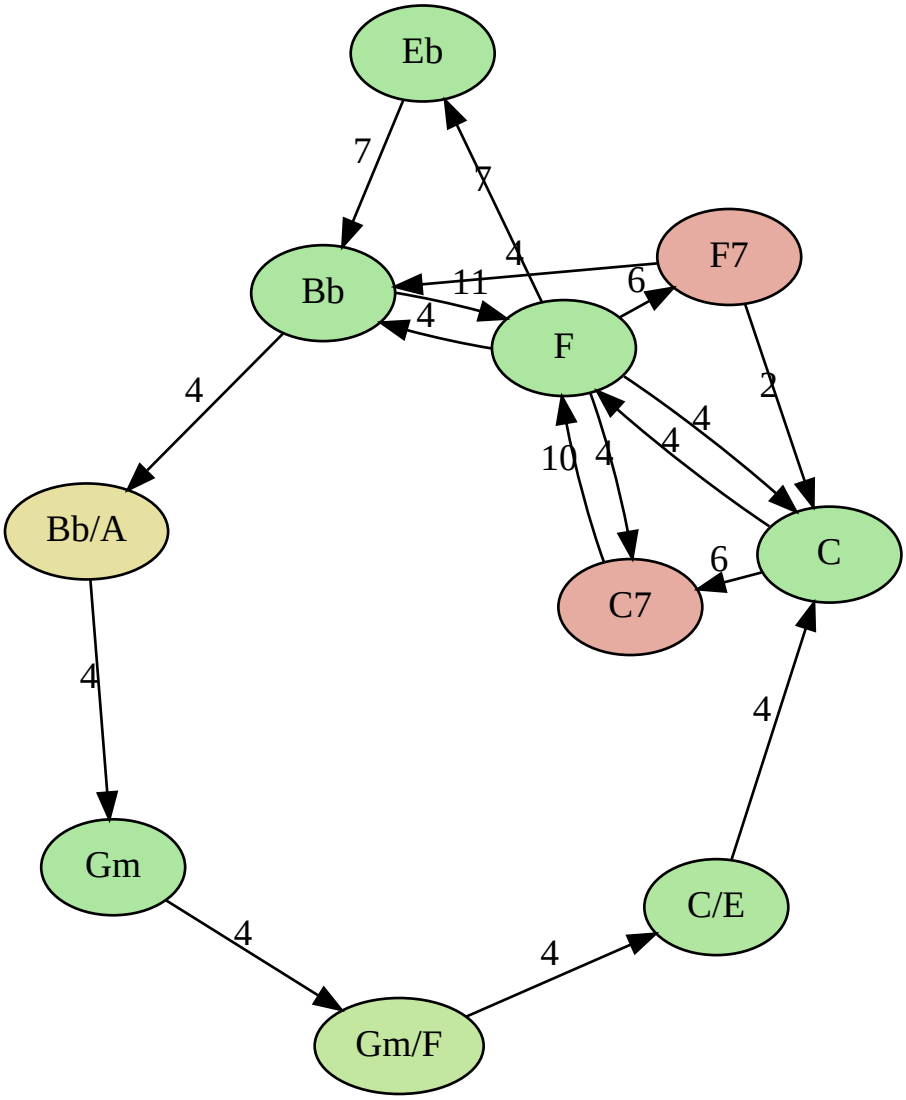
When analyzing music with MPS, progression graphs of the following musical aspects are generated:

- Harmonic Progressions
- Rhythmic Progressions
- Pitch Progressions
- Lyric Progressions

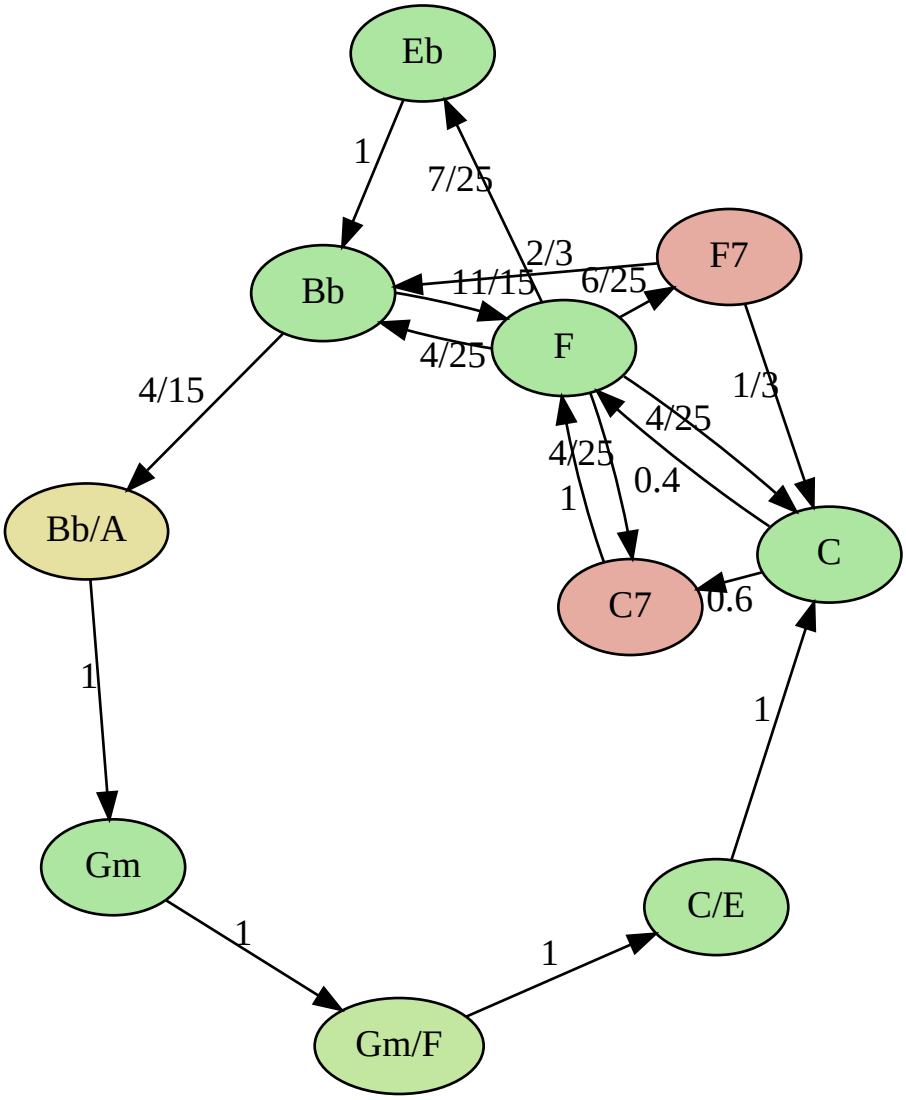
Depending on the setting *Progression Graph Label Mode* described in the previous section, the edges in the resulting graphs will either have

- Occurrence Counts
- Measure Numbers or
- Markov Model Probabilities

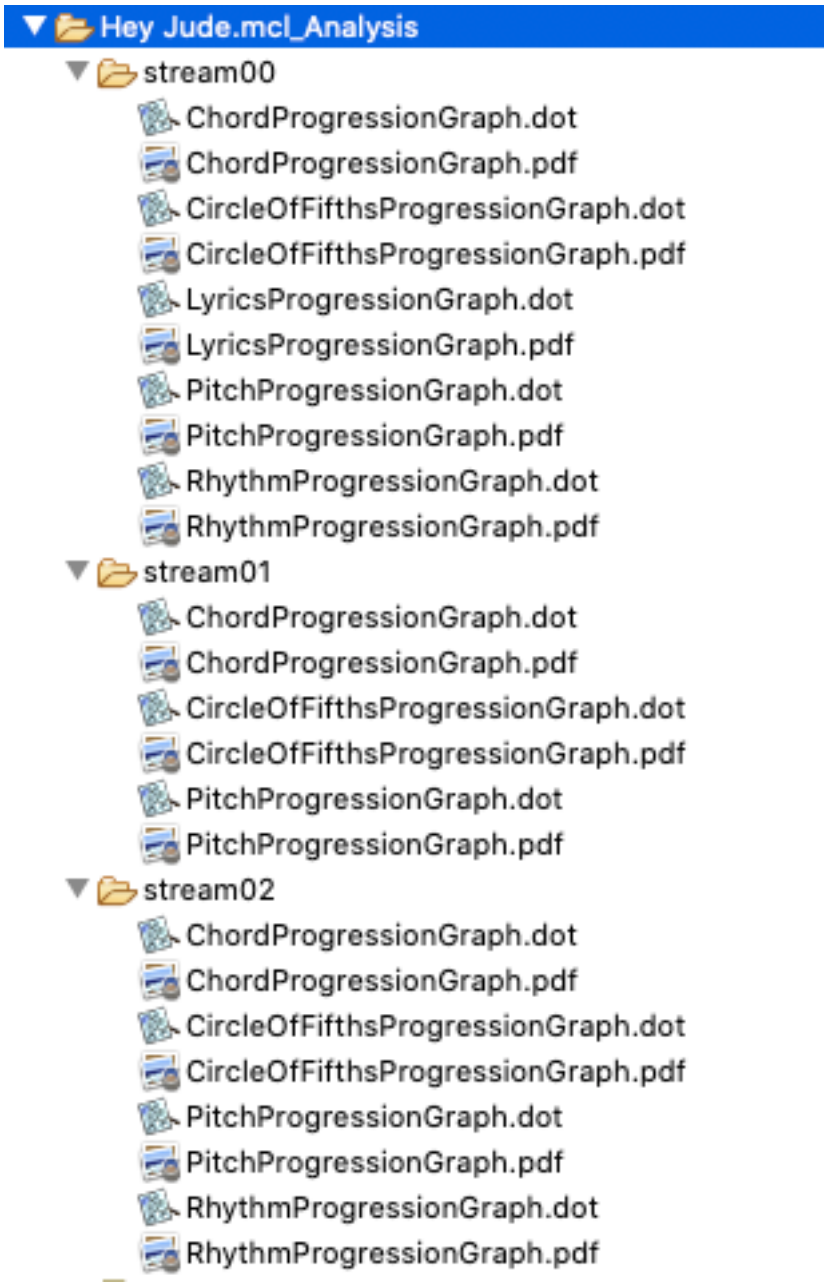
as labels. An example with measure numbers was already presented in the previous section. Here is a harmonic progression graph of *Hey Jude* by the Beatles with occurrence count labels visualizing how often the corresponding transition was found in the piece:



And this is the same harmonic progression graph as a Markov model with probability labels:



The graphs are exported for each stream separately and can be found in the respective subfolders:



Corresponding PDF files will be generated automatically if GraphViz is installed and configured as described in GraphViz Installation.

If the graphs should be included in analysis report PDF files, the option *Export Stream-specific Analysis Results* needs to be activated in the preferences as shown in the previous section.

Analysis Features

The following table lists all 117 musical features which can be analyzed with MPS.

Feature	Type	Exported Data
Total Number of Notes	Number	Total Number of Notes
Total Number of Rests	Number	Total Number of Rests

Instrument Duration Distribution	Relative Distribution	Instrument, Relative Quantity
Note Density	Percentage	Note Density
Note Duration Average	Average	Note Duration Average
Note Duration Standard Deviation	Standard Deviation	Note Duration Standard Deviation
Note Duration Distribution	Relative Distribution	Note Duration, Relative Quantity
Note Duration Distribution Dependent on Beat	Relative Distribution	Beat, Note Duration, Relative Quantity
Note Duration Distribution Dependent on Beat (Absolute)	Absolute Distribution	Beat, Note Duration, Absolute Quantity
Rest Duration Average	Average	Rest Duration Average
Rest Duration Standard Deviation	Standard Deviation	Rest Duration Standard Deviation
Rest Duration Distribution	Relative Distribution	Rest Duration, Relative Quantity
Rest Duration Distribution Dependent on Beat	Relative Distribution	Beat, Rest Duration, Relative Quantity
Beat Distribution	Relative Distribution	Beat, Relative Quantity
Beat Distribution	Absolute Distribution	Beat, Absolute Quantity
Time Signature Distribution	Relative Distribution	Time Signature, Relative Quantity
Loudness over Time	Values Over Time	Time, Loudness
Loudness over Time	Values Over Time	Measure, Loudness
Loudness Distribution	Relative Distribution	Loudness, Relative Quantity
Minimum Pitch	String	Minimum Pitch
Maximum Pitch	String	Maximum Pitch
Minimum Pitch over Time	Values Over Time	Time, Minimum Pitch
Minimum Pitch over Time	Values Over Time	Measure, Minimum Pitch
Maximum Pitch over Time	Values Over Time	Time, Maximum Pitch
Maximum Pitch over Time	Values Over Time	Measure, Maximum Pitch
Pitch Distribution	Relative Distribution	Pitch, Relative Quantity
Piano Roll	Values Over Time	Time, Pitch, Duration
Piano Roll	Values Over Time	Measure, Pitch, Duration
Interval Leap Average	Average	Interval Leap Average
Interval Leap Standard Deviation	Standard Deviation	Interval Leap Standard Deviation

Interval Leap Distribution	Relative Distribution	Interval Leap, Relative Quantity
Interval Leap Distribution Dependent on Beat	Relative Distribution	Beat, Interval Leap, Relative Quantity
Interval Leaps over Time	Values Over Time	Time, Interval Leap, Absolute Quantity
Interval Leaps over Time	Values Over Time	Measure, Interval Leap, Absolute Quantity
Number of Simultaneously Audible Notes Average	Average	Number of Simultaneously Audible Notes Average
Number of Simultaneously Audible Notes Standard Deviation	Standard Deviation	Number of Simultaneously Audible Notes Standard Deviation
Number of Simultaneously Audible Notes Distribution	Relative Distribution	Number of Simultaneously Audible Notes, Relative Quantity
Pitch Combination Distribution	Relative Distribution	Pitch Combination, Relative Quantity
Pitch Combination Distribution (Absolute)	Absolute Distribution	Pitch Combination, Absolute Quantity
Pitch Combination Duration Distribution	Relative Distribution	Pitch Combination, Relative Quantity
Pitch Combination Duration Distribution (Absolute)	Absolute Distribution	Pitch Combination, Duration
Simultaneously Audible Interval Distribution	Relative Distribution	Interval, Relative Quantity
Simultaneously Audible Interval Distribution Dependent on Beat	Relative Distribution	Beat, Interval, Relative Quantity
Dissonance Average	Average	Dissonance Average
Dissonance Standard Deviation	Standard Deviation	Dissonance Standard Deviation
Dissonance Distribution	Relative Distribution	Dissonance, Relative Quantity
Dissonance Distribution Dependent on Beat	Relative Distribution	Beat, Dissonance, Relative Quantity
Dissonance over Time	Values Over Time	Time, Dissonance
Dissonance over Time	Values Over Time	Measure, Dissonance
Keys over Time	Values Over Time	Time, Key
Keys over Time	Values Over Time	Measure, Key
Key Distribution	Relative Distribution	Key, Relative Quantity

Circle of Fifths Distance Distribution of Keys	Relative Distribution	Circle of Fifths Distance, Relative Quantity
Harmonies over Time	Values Over Time	Time, Harmony
Harmonies over Time	Values Over Time	Measure, Harmony
Harmony Distribution	Relative Distribution	Harmony, Relative Quantity
Harmony Distribution (Simplified)	Relative Distribution	Harmony, Relative Quantity
Chord Inversion Distribution	Relative Distribution	Inversion, Relative Quantity
Harmony Change Beat Distribution	Relative Distribution	Beat, Relative Quantity
Harmony Duration Distribution	Relative Distribution	Harmony Duration, Relative Quantity
Circle of Fifths Distance Distribution of Harmonies	Relative Distribution	Circle of Fifths Distance, Relative Quantity
Harmony Distribution Dependent on Beat	Relative Distribution	Beat, Harmony, Relative Quantity
Harmony Distribution Dependent on Duration	Relative Distribution	Harmony Duration, Harmony, Relative Quantity
Implicit Harmonies over Time	Values Over Time	Time, Implicit Harmony, Inversion, Pitch Combination
Implicit Harmonies over Time (Simplified)	Values Over Time	Time, Implicit Harmony, Inversion, Pitch Combination
Implicit Harmonies over Time	Values Over Time	Measure, Implicit Harmony, Inversion, Pitch Combination
Implicit Harmonies over Time (Simplified)	Values Over Time	Measure, Implicit Harmony, Inversion, Pitch Combination
Implicit Harmony Distribution	Relative Distribution	Implicit Harmony, Relative Quantity
Implicit Harmony Distribution (Simplified)	Relative Distribution	Implicit Harmony, Relative Quantity
Harmony Change Beat Distribution (Simplified)	Relative Distribution	Beat, Relative Quantity
Implicit Harmony Duration Distribution	Relative Distribution	Harmony Duration, Relative Quantity
Circle of Fifths Distance Distribution of Implicit Harmonies	Relative Distribution	Circle of Fifths Distance, Relative Quantity
Implicit Harmony Distribution Dependent on Beat	Relative Distribution	Beat, Implicit Harmony, Relative Quantity

Implicit Harmony Distribution Dependent on Beat (Simplified)	Relative Distribution	Beat, Implicit Harmony, Relative Quantity
Implicit Harmony Distribution Dependent on Duration	Relative Distribution	Harmony Duration, Implicit Harmony, Relative Quantity
Implicit Harmony Distribution Dependent on Duration (Simplified)	Relative Distribution	Harmony Duration, Implicit Harmony, Relative Quantity
Harmony Analysis over Time	Values Over Time	Time, Key, Harmony, Roman Numeral (Classical), Roman Numeral (Pop/Jazz)
Harmony Analysis over Time	Values Over Time	Measure, Key, Harmony, Roman Numeral (Classical), Roman Numeral (Pop/Jazz)
Chord Compliance	Percentage	Chord Compliance
Root Note Compliance	Percentage	Root Note Compliance
Bass Note Compliance	Percentage	Bass Note Compliance
Scale Compliance Relative to Key	Percentage	Scale Compliance Relative to Key
Scale Compliance Relative to Harmony	Percentage	Scale Compliance Relative to Harmony
Syllable Distribution	Relative Distribution	Syllable, Relative Quantity
Words over Time	Values Over Time	Start Time, End Time, Word
Words over Time	Values Over Time	Start Measure, End Measure, Word
Word Distribution	Relative Distribution	Word, Relative Quantity
Sentence Parts over Time	Values Over Time	Start Time, End Time, Sentence Part
Sentence Parts over Time	Values Over Time	Start Measure, End Measure, Sentence Part
Sentences over Time	Values Over Time	Start Time, End Time, Sentence
Sentences over Time	Values Over Time	Start Measure, End Measure, Sentence
Sentence Sentiment Polarity over Time	Values Over Time	Start Time, End Time, Sentence, Compound Polarity, Very Negative Polarity, Negative Polarity, Neutral Polarity, Positive Polarity, Very Positive Polarity

Sentence Sentiment Polarity over Time	Values Over Time	Start Measure, End Measure, Sentence, Compound Polarity, Very Negative Polarity, Negative Polarity, Neutral Polarity, Positive Polarity, Very Positive Polarity
Aggregated Note Duration Distribution	Aggregated Relative Distribution	Note Duration, Relative Quantity
Aggregated Rest Duration Distribution	Aggregated Relative Distribution	Rest Duration, Relative Quantity
Aggregated Note Duration Distribution Dependent on Beat	Aggregated Relative Distribution	Beat, Note Duration, Relative Quantity
Aggregated Beat Distribution	Aggregated Relative Distribution	Beat, Relative Quantity
Aggregated Pitch Distribution	Aggregated Relative Distribution	Pitch, Relative Quantity
Aggregated Interval Leap Distribution	Aggregated Relative Distribution	Interval Leap, Relative Quantity
Aggregated Interval Leap Distribution Dependent on Beat	Aggregated Relative Distribution	Beat, Interval Leap, Relative Quantity
Aggregated Dissonance Distribution	Aggregated Relative Distribution	Dissonance, Relative Quantity
Aggregated Dissonance Distribution Dependent on Beat	Aggregated Relative Distribution	Beat, Dissonance, Relative Quantity
Aggregated Simultaneously Audible Interval Distribution	Aggregated Relative Distribution	Interval, Relative Quantity
Aggregated Simultaneously Audible Interval Distribution Dependent on Beat	Aggregated Relative Distribution	Beat, Interval, Relative Quantity
Aggregated Time Signature Distribution	Aggregated Relative Distribution	Time Signature, Relative Quantity
Aggregated Key Distribution	Aggregated Relative Distribution	Key, Relative Quantity
Aggregated Harmony Distribution	Aggregated Relative Distribution	Harmony, Relative Quantity
Aggregated Simplified Harmony Distribution	Aggregated Relative Distribution	Harmony, Relative Quantity

Aggregated Implicit Harmony Distribution	Aggregated Relative Distribution	Implicit Harmony, Relative Quantity
Aggregated Simplified Implicit Harmony Distribution	Aggregated Relative Distribution	Implicit Harmony, Relative Quantity
Aggregated Harmony Duration Distribution	Aggregated Relative Distribution	Harmony Duration, Relative Quantity
Aggregated Harmony Change Beat Distribution	Aggregated Relative Distribution	Beat, Relative Quantity
Aggregated Harmony Distribution Dependent on Beat	Aggregated Relative Distribution	Beat, Harmony, Relative Quantity
Aggregated Harmony Distribution Dependent on Duration	Aggregated Relative Distribution	Duration, Harmony, Relative Quantity
Aggregated Circle of Fifths Distance Distribution of Keys	Aggregated Relative Distribution	Circle of Fifths Distance, Relative Quantity
Aggregated Circle of Fifths Distance Distribution of Harmonies	Aggregated Relative Distribution	Circle of Fifths Distance, Relative Quantity
Aggregated Circle of Fifths Distance Distribution of Implicit Harmonies	Aggregated Relative Distribution	Circle of Fifths Distance, Relative Quantity

Algorithmic Composition

MPS features an algorithmic composition system which generates music based on statistical criteria.

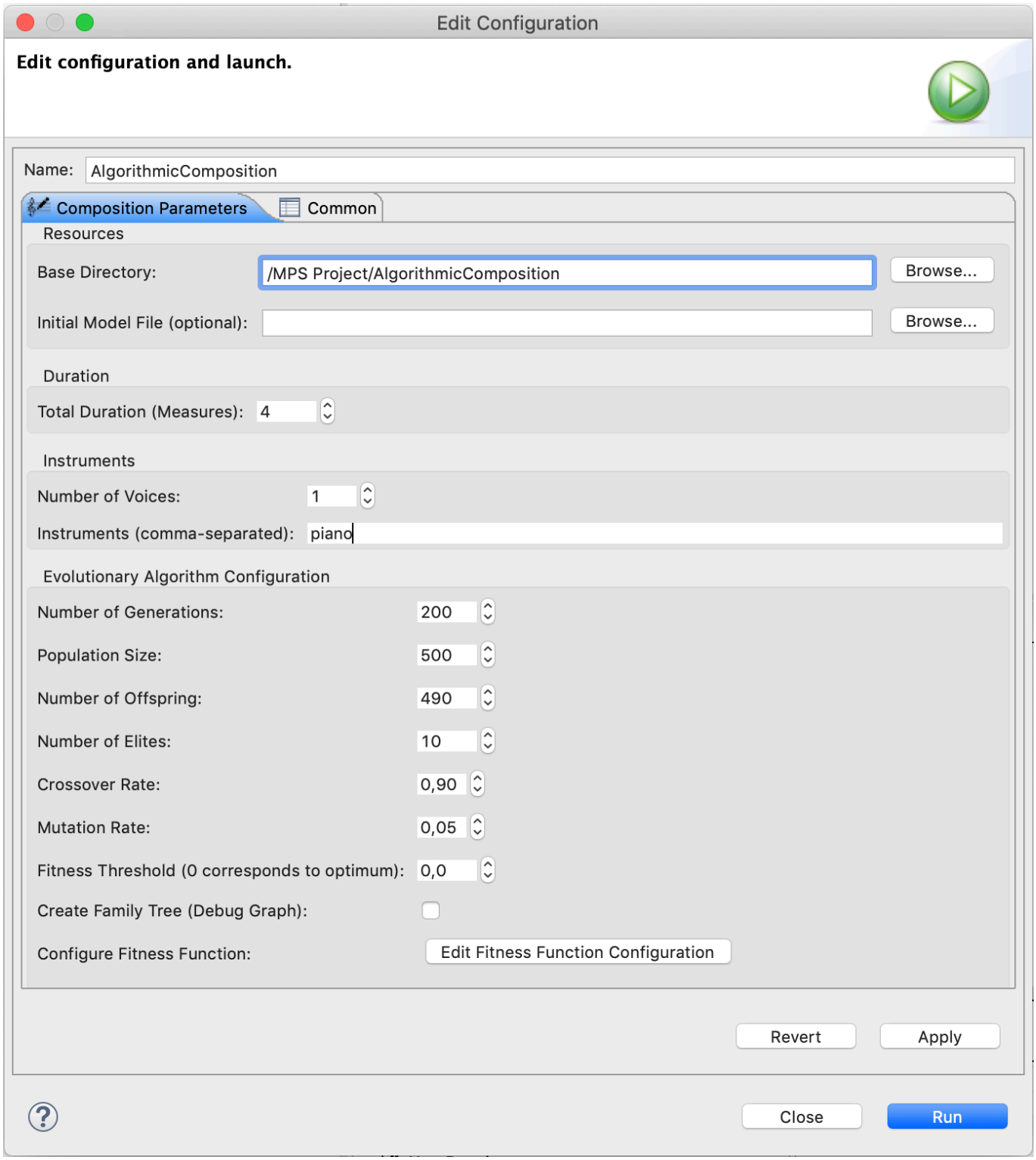
Generating Compositions

To start generating a composition, it is recommended to create a new folder by right-clicking on a project and selecting *New* → *Folder*.

Next, select the created folder and click on the algorithmic composition button:


Algorithmic Composition Launch Configurations

A launch configuration will be created and opened automatically:



Here you can enter the desired number of measures, number of voices and the instruments that should be used in the composition. If you don't specify any instruments here, instruments will be selected randomly.

Furthermore, parameters for the evolutionary algorithm can be adjusted. It is recommended to use a high crossover rate and a low mutation rate in order to prevent the algorithm from

converging too early on local optima (as opposed to global optima, which are preferable). Adjusting the number of generations, the population size or the number of offspring will influence the time the algorithm takes to run. However, if these values are set too low, the evolutionary algorithm might not be able to satisfy all optimization criteria.

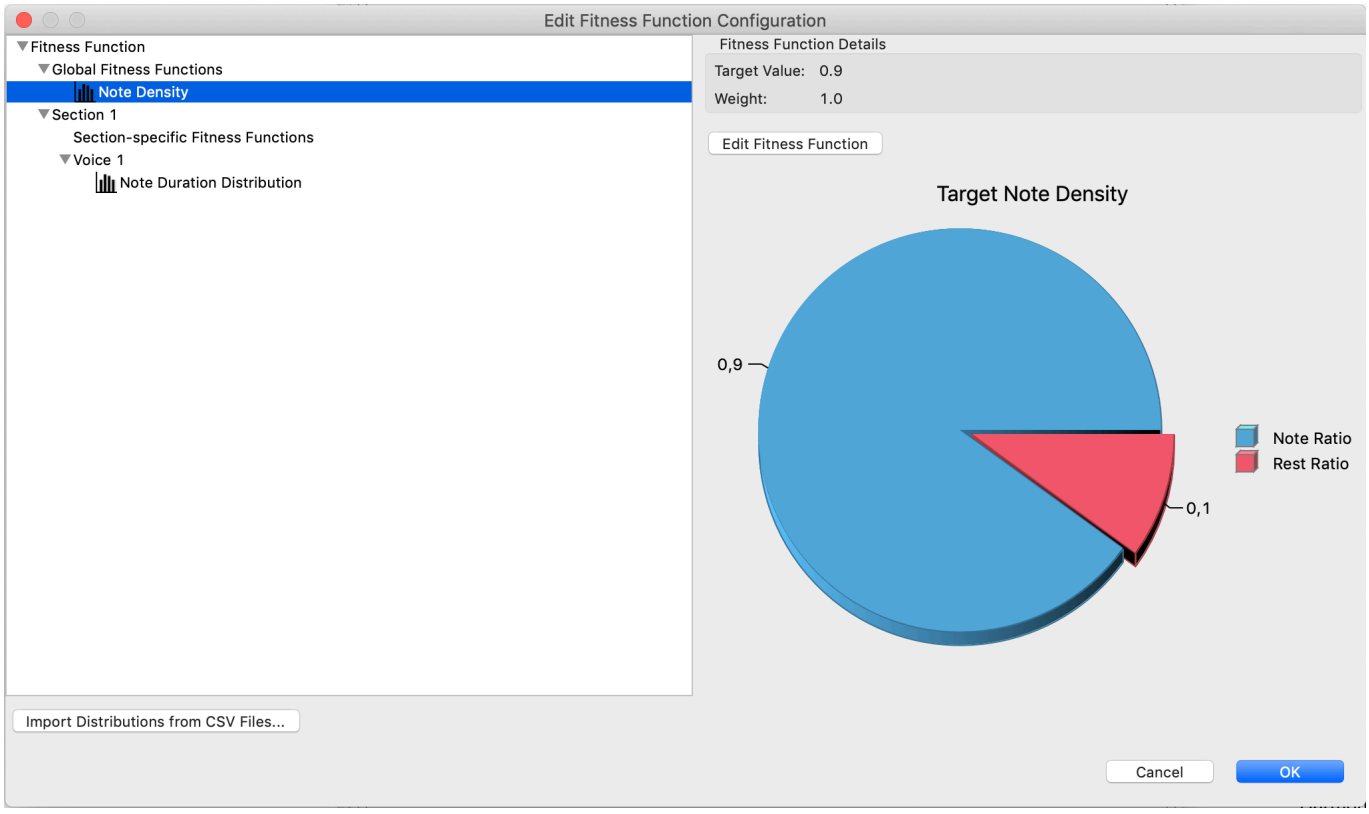
Composer launch configurations can be reused, edited and deleted at any time. Refer to section Launch Configurations for further details.

Fitness Function Configuration

The statistical criteria to be fulfilled are specified by means of a so called fitness function. The fitness function can contain statistical criteria on a global level and multiple subsidiary levels. Each musical piece to be generated is divided into an arbitrary number of sections, which in turn contain individual voices. It is possible to specify musical criteria for the whole piece (global level), for individual sections (section level) and for each voice in each section (voice or stream level).

Note that the fitness function configuration must be performed at least once. Otherwise, the evolutionary algorithm can not be run.

Open the fitness function configuration dialog by clicking the button *Open Fitness Function Configuration* in the algorithmic composition launch configuration:



The user interface allows to add, edit and remove fitness functions on the global level, section level and voice/stream level. The fitness function must contain at least one section and one

voice per section. In the example above, there is a global fitness function aiming for an overall note density of 90%, as well as a section with one voice in which the distribution of note durations is optimized.

The fitness function configuration will be stored in a file named `FitnessFunctionConfiguration.ff` in the specified base directory for the generated composition.

Creating Fitness Functions by Importing Analysis Results

The fitness function configuration dialog offers a button labeled *Import Distributions from CSV files* which allows to construct fitness functions from MPS analysis results. Generating compositions that have imported statistical values as target fitness function will effectively result in a style copy of the analyzed compositions. Note that not all statistical features can automatically be converted into a fitness function. Refer to chapter Music Analysis for more details on analysis features.

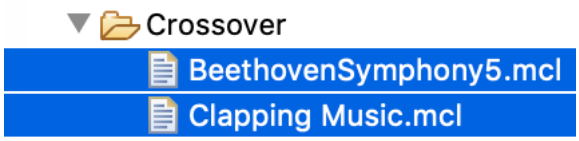
Generating Compositions Algorithmically

After configuring an algorithmic composition run, click the *Run* button in the launch configuration dialog. Progress will be reported in the corresponding *Progress* view, which is located at the bottom of the application by default. Depending on your configuration, evolutionary algorithm runs might take quite a long time. The results will be saved in a subdirectory named *ComposerResults* in the specified base directory for the generated composition. The resulting composition file is named *SectionWiseComposition.mcl* and can be converted into a score and/or MIDI file as described in section Transforming Compositions to Scores.

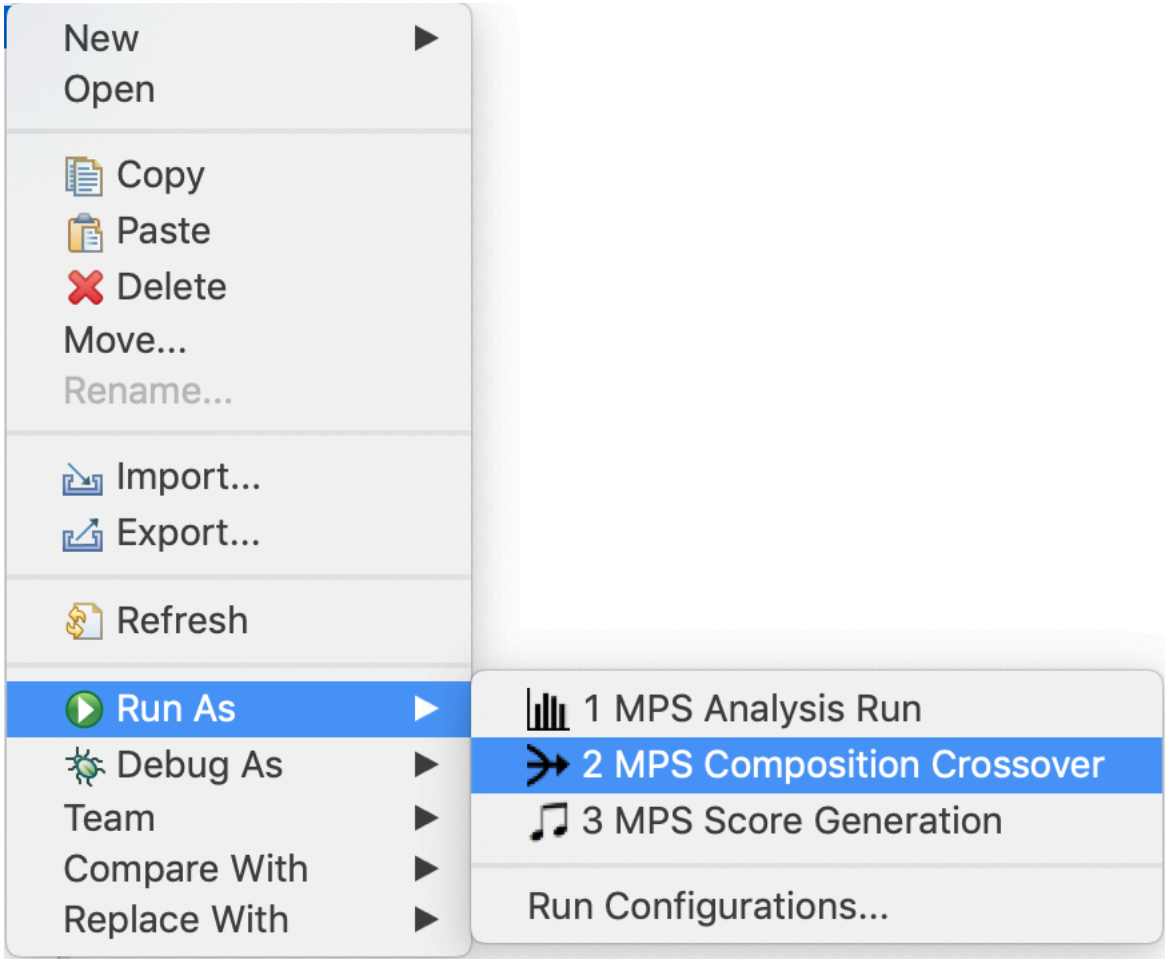
Composition Crossover

Composition crossover combines musical material of an arbitrary number of compositions to a new composition. This is achieved by recombining context tree composition models using the evolutionary algorithm described before. The algorithm is configured with a very high crossover rate (close or equal to 100%) and uses an automatically generated fitness function which optimizes the resulting context tree model to contain about the same ratio of tree nodes from each input composition.

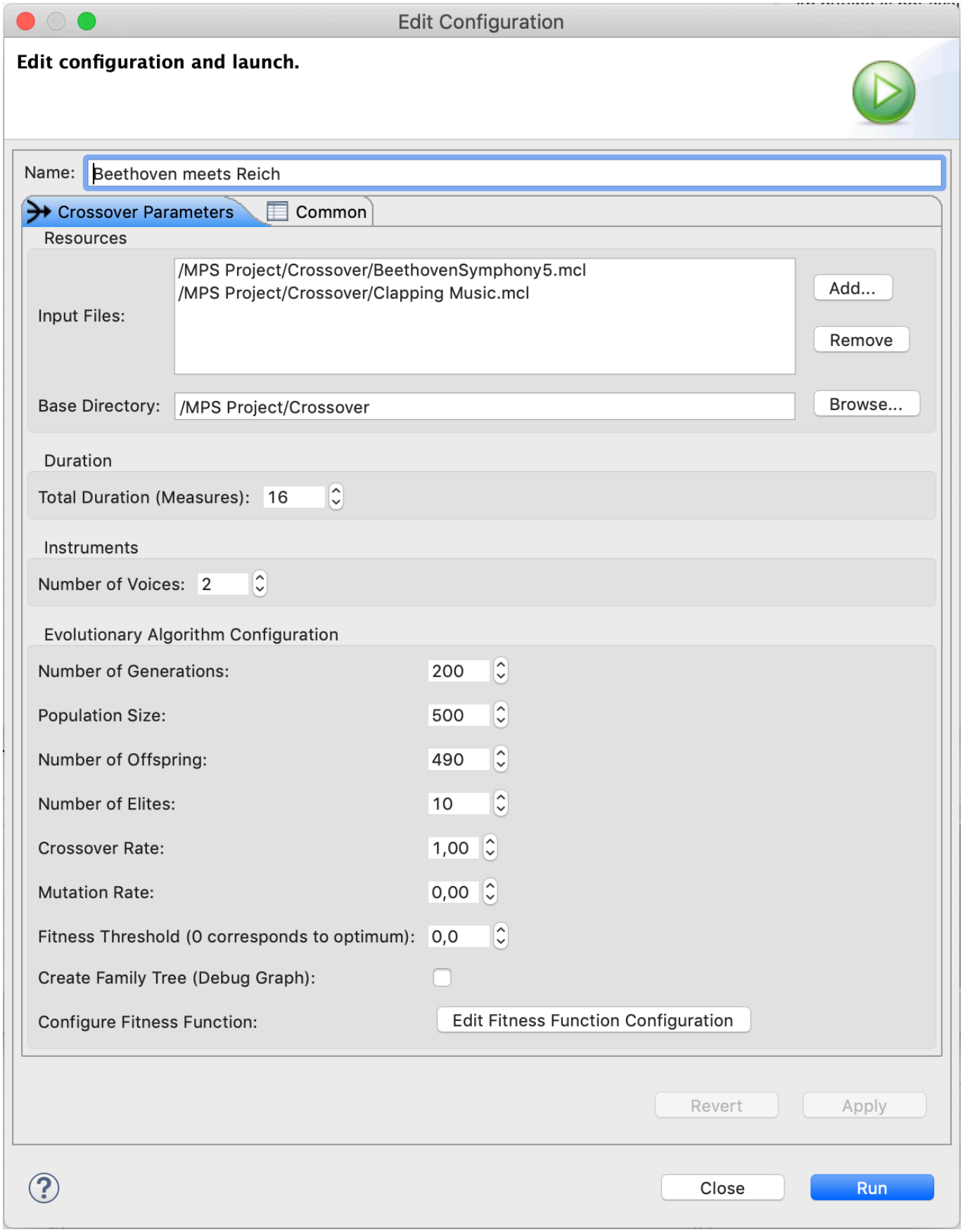
To start a composition crossover run, select at least two compositions in your workspace:



Open the context menu with a right click and select *Run As* → *MPS Composition Crossover*:



A launch configuration will automatically be created and the following dialog appears:



Specify the desired number of measures and voices. It is recommended to use a very high crossover rate and a very low mutation rate. Adjusting the number of generations, the population size and the number of offspring influences the time the algorithm takes to run. However, if these values are set too low, the evolutionary algorithm might not be able to satisfy all optimization criteria.

Crossover launch configurations can be reused, edited and deleted at any time. Refer to section Launch Configurations for further details.

Troubleshooting

This section contains solutions to known issues.

Analysis Report Generation Fails with Fatal Error

If PDF generation fails with the following error:

```
Fatal error occurred, no output PDF file produced
```

In general, search for the **first** error in the console output to pinpoint the problem.

Possible fixes:

- Check whether all required LaTeX packages are installed as described in section LaTeX Installation
- Check whether the `AnalysisReport.tex` file contains unescaped special characters. If this is the case, please report this on the MPS mailing list. For example, in LaTeX the following replacements must be performed:
  - `_` → `\_`
  - `#` → `\#`
  - `°` → `$^\circ$` or `\textdegree`