The Somax 2 Software Architecture

Rev. 0.2.0

Joakim Borg

April 16, 2021

Contents

Somax 2 is a new version of the Somax reactive co-improvisation paradigm by. G. Assayag, with a completely redesigned architecture and UI written by Joakim Borg.

Legacy Somax versions by Laurent Bonnasse-Gahot and Axel Chemla-Romeu-Santos.

The Somax 2 project is part of the ANR project MERCI (Mixed Musical Reality with Creative Instruments) and the ERC project REACH (Raising Co-creativity in Cyber-Human Musicianship).

PI : Gérard Assayag
Music Representation Team
IRCAM STMS Lab (CNRS, Sorbonne University, Ministry of Culture).

Chapter 1

Overview

This report documents the underlying software architecture of the Somax environment. It is written with the intention to give a clear overview of the Somax architecture for anyone who in some way wishes to extend, contribute, maintain or in some other way modify the current architecture, as well for any reader who simply wishes to dissect parts of or the whole architecture to get a better understanding of its implemented solutions. It is recommended to read [3] before reading this report, in which the theoretical model of Somax is described, but it's possible to read them in any order, as there are plenty of cross-references to relevant sections in both of these reports.

Chapter 2 in this text summarizes the Somax model from a user perspective and is intended to give an overview of the system for anyone with limited practical experience of the Somax system. Chapter 3 describes the back-end of the architecture, written in Python, and outlines the main modules and the relationship between them, as well as important classes that can be extended to implement new behaviour to the system. Chapter 4 gives an overview of the front-end of the system, written in Max, describing the core Max objects of the system. Finally, the user-oriented documentation and tutorials of Somax, which are written in and integrated into Max and due to the visual format difficult to parse into text, are included in the appendix as screenshots.

Do note that this is an early revision of the document and should only be seen as an outline of the architecture. It should contain any information necessary to add new classifiers, models, filters and other modular aspects of the system to the code, but it will not describe every aspect of the architecture nor does it go very deep into implementation details - for that it will be necessary to read through the entire code base. Still, it's intended to serve as a good starting point for such a process, and reading this document should significantly speed up this endeavour.

Chapter 2

An Introduction to Somax

Somax is an interactive system which improvises around a musical material, aiming to provide a stylistically coherent improvisation while in real-time listening to and adapting to input from a musician. The system is trained on some musical material selected by the user, from which it constructs a corpus that will serve as a basis for the improvisation. The corpus and the output of the system is currently based on MIDI, but it is able to listen and adapt to both audio and MIDI input from a musician.

The main idea is that Somax should serve as a co-creative agent in the improvisational process, where the system after some initial tuning is able to listen and adapt to the musician in a self-sufficient manner. Of course, the input doesn't have to come from a live musician; any type of audio and/or MIDI input works, be it from an audio file, score editor, synthesizer or DAW. Somax also allows detailed parametric controls of its output and can even be played as an instrument in its own right. Also, the system isn't necessarily limited to a single agent or a single input source - it is possible to create an entire ensemble of agents where the user in detail can control how the agents listen to various input sources as well as to each other.

The goal of this text is to provide a brief introduction to Somax and provide the reader the with fundamental knowledge about how its interaction model works, which in turn should serve as a basis for making informed choices when tuning and interacting with the system. For a hands-on introduction to Somax and its user interface, also see the interactive tutorials that can be found in the same folder as this document.

2.1 The Corpus and the Navigation Model

As previously mentioned, the Somax system generates its improvisation material based on an external set of musical material, the «corpus». This corpus can be constructed from one or multiple MIDI files freely chosen by the user. In contrast to many other generative approaches, the system does not construct a model that eventually is independent of the material that was used to train it. Rather, the model is constructed directly on top of the original data and provides a way to navigate through it in a non-linear manner. One way of seeing this is to consider that some

fine-grained aspects of the musical stream are somehow too complex to be modeled, but will be preserved – to a certain extent – when rereading this musical material.

When presented with some MIDI material, the first step is to segment the musical stream into discrete units or «slices», which are vertical (polyphonic) segments of the original midi file and where the duration of a slice is the distance between two note onsets.[1] Each slice is analyzed and classified with regards to a number of musical features related to its harmony, individual pitches, dynamics, etc., and these features will serve as the main basis for constructing the navigation model. This process is in a way similar to that of concatenative synthesis, where an audio signal is segmented into meaningful units, analyzed and recombined with respect to the analysis, but in this case (at least for now) based on MIDI data.

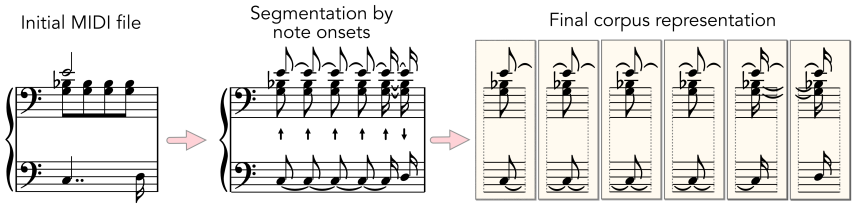

Figure 2.1: Constructing a corpus by segmenting midi data into «slices».

From the sequence of slices, the navigation model is constructed through the detection of common patterns within the musical material. Or more specifically, the procedure of detecting common patterns is repeated for each feature in the analysis, effectively resulting in a multilayer representation where each layer roughly corresponds to one feature, i.e., one layer for harmony, one for pitch, etc.

When a musician interacts with the system, a similar process of segmentation and multilayer analysis is computed on the input stream, and at each point in time the result of this process is matched to that of the navigation model, generating activations, or «peaks», at certain points in the corpus where the input corresponds to the model. Each peak corresponds to a point in the memory, so the entire set of peaks can effectively be seen linearly as a one-dimensional representation over the corpus' time axis (see figures 2.4 and 2.6 below for examples). The peaks in each layer are merged and scaled all according to how the system has been tuned, and finally the output slice is selected based on the distribution of the peaks.

The generated output of this process is an co-improvisation that recombines existing material in a way that's coherent with the sequential logic and statistical properties of the original material while at the same time adapting in real-time to match the input from the musician. One benefit of this procedure compared to many other approaches of statistical machine learning is that the system can generate long im-

[1] In recorded corpora (or any type of non-quantized MIDI) it is rare for any two notes that are perceived as simultaneous to be exactly simultaneous. Since one goal of Somax is to be able to maintain and reproduce the original timings within slices as recorded, notes with almost simultaneous onsets will still be grouped together in a single slice but with their internal timing offset preserved.

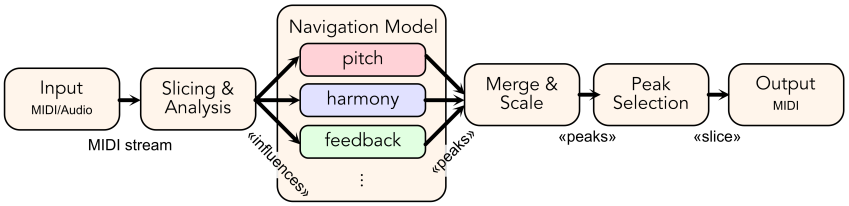

Figure 2.2: An overview of the steps through which the system generates its output at each given point in time.

provisations from a small musical corpus, allowing the user detailed control over the style of the output or even specifically compose pieces of material intended to serve as a Somax corpora. Of course, it has to be recognized that this method is also one of the downsides with the model. The process of improvisation is to some extent reduced to a smart cut and pasting of pre-existing material, which is a very simplistic modelling of what improvisation is as a human skill. Still, the output is through the process of attempting to balance the internal logic of the corpus with the external logic of the musician often providing a mix of coherency and unexpectedness in a way that convincingly gives the impression of an active agent in the improvisational process.

2.2 Interaction

When interacting with Somax, there are three main concepts that are important to understand: «slices», «influences» and «peaks». A slice, as previously mentioned, is a short segment of the corpus and serves as the smallest building block of the output of the system. The slice can be manipulated to some extent (transposed, filtered with regards to voices/channels, etc.) but will always maintain most fundamental properties of the original corpus.

An «influence» is in a way conceptually very similar to a slice, but with a vastly different purpose. When Somax listens to a musician, this musical stream is segmented and analyzed with respect to its musical parameters similarly to how the corpus was constructed, but with a slightly different set of methods to be able to operate in real-time. The result of this process are discrete chunks of multilayer data or «influences», which the system uses to be able to compare the input to the corpus, where the main purpose of the influence is to act as the guide that determines the output of the system. The concept of an influence may initially seem like an implementation detail, but will become increasingly important for more complex configurations with multiple agents and/or multiple input sources. The main takeaway is that the system cannot listen directly to a musical input stream, but will need to translate it into influences, and that the process of tuning the listener can be a very important factor for the quality of the co-improvisation.

Finally, a «peak» is, again, a point in the corpus where the input corresponds to

the model, or simply a match between an incoming influence and a corresponding slice that would serve as an output candidate. Each peak has a height, corresponding to a probability (or viability) of that particular slice as an output candidate. Unlike influences (which are visible in the interface) and slices (which are correlated to the audible output of the system), peaks are never interacted with directly, they're only part of the internal state of the system, but perhaps the most vital part. Each peak effectively corresponds to a slice in the corpus that could serve as an output at the current point in time, given the latest influence. Having a reasonable number of peaks is thus vital for the quality of the output, since having no peaks means that the output has not taken the musician's influences into account, and on the opposite side, in most cases a large number of peaks indicate that the matching is imprecise.

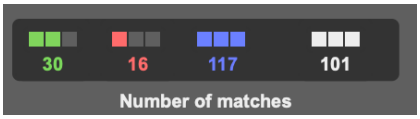

Figure 2.3: While the user doesn't interact directly with the peaks, they are still indicated in the user interface. Here, the colors green, red, blue and white correspond to the number of peaks in the feedback layer (more on this later), pitch layer, harmonic layer and total number of peaks after merge respectively.

To put the concept of peaks in context, let's dive a bit deeper in how the system works. While the musician is playing, Somax is at each detected onset segmenting the input into influences, carrying information about the pitch, harmony, etc. of what the musician currently is playing. This process is carried out by agents of the system called «influencers». This information is routed to a «player», which handles the entire process of matching and generating output. The influence is classified in multiple layers by the player, as briefly mentioned, where each layer corresponds to one musical dimension (e.g. harmony, melody) of the influence. In each layer, a model of the corpus with respect to the particular layer's musical dimension exists, and upon receiving an influence, the model will look for sequences in the corpus that match the sequence of most recent influences from the input, and in each of those places generate peaks.

The system is also simulating a type of short-term memory inside this model by not immediately discarding peaks from previous influences, but rather shifting them along the time axis of the corpus and decaying their height corresponding to the amount of time that has passed, followed by merging them with the new set of peaks. This means that sequences continuously matching several consecutive influences will be more highly prioritized over others, as is illustrated in figure 2.4. Finally, the peaks from all layers will be merged together into a single set of peaks which the system will use to probabilistically determine which slice is the best output candidate[2]. The result of this multilayer peak merging process is an output that will not just strictly match the harmony and pitch of the influence but rather improvise around the most recent history of influences with regards to the corpus, often (de-

[2]actually, in addition to this, there are a number of parameters that scale the height of the peaks individually with regard to a number of other musical parameters of choice, but this is thoroughly documented in the help files in max and will not be discussed here

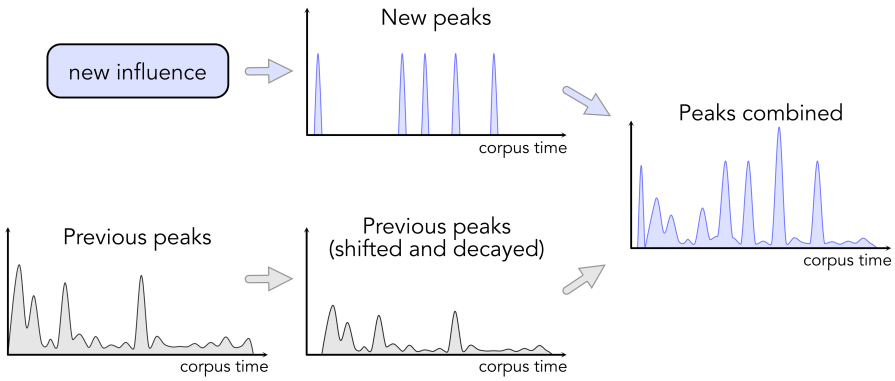

Figure 2.4: The process of shifting and decaying previous peaks in a single layer upon receiving new influences (the process of matching the incoming influence to the corpus has been omitted for clarity).

pending on how the parameters tuned) selecting peaks matching both the harmony and/or melody of the input but with an ability and agency to act more freely with regards to its history. In addition, there's also a layer which listens to the output of the system, a feedback layer, that can be used to balance the player's consistency with the input with its continuity with its own performance. The balance between the different layers as well as control over the decay time of old peaks, length of sequences to match in the memories, etc. are all available in the user interface as displayed in figure 2.5.

If the concept of peaks isn't perfectly clear to you after reading this – don't worry! Go to the tutorial and start experimenting with the system while keeping one thing in mind: if the number of peaks is continuously zero or continuously too high[3], this is likely an indicator that the system is working poorly and should be retuned. If not – you're probably doing quite well.

Another important aspect of the interaction with Somax is its relation to time. According to the user's preference, each player can be assigned to either operate continuously in time as an autonomous agent, maintaining the pulse and exact within-slice timings of the original corpus (while possibly adapting to the tempo and/or phase of the input), or operate reactively, generating output synchronously as requested by the input. In the continuous case, this means that the player improvises freely over time while still taking the influences of the musician into account, while in the reactive case, it synchronizes strictly (note-by-note) with the input. Of course, the player is in the latter mode not strictly limited to the input from where it receives its influences, but could be connected to a third source of some sort, for example any type of step-sequencer or other generative approaches, thus giving the user multiple options for controlling the temporal domain of the system.

[3]exactly how large "too high" is varies with the type of layer and context, but larger than 10% of the total number of slices in the corpus with no transpositions active could serve as a reference of "too high" that is valid for most layers and contexts

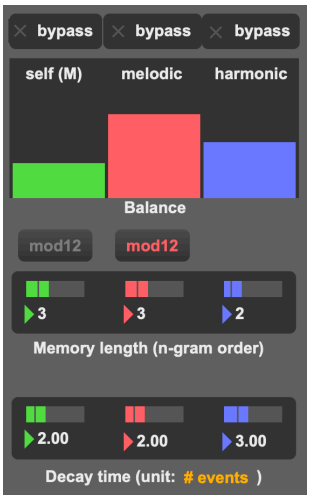

Figure 2.5: User interface to control the balance between the dimensions, length of matching sequences for each dimension as well as decay time of peaks.

2.3 Constructing the Corpus

By default, constructing a corpus is as simple as dragging a MIDI file to the «Corpus Builder» window in the user interface, from which Somax will build an internal representation with slices, as explained, along with annotations attached to relevant dimensions. There are however a number of things to consider here. If the input to a player and its internal corpus were made from similar source of materials, an ideal response of the player would tend to simply replicate the input. A great deal of variations or new musical situations will however arise from the discrepancies between the input and the corpus, and from the different mappings that the user can set for defining the musical dimensions considered by the player. As we are mostly talking about MIDI content here (players and MidiInfluencers) this is nothing else than a mapping between the MIDI channels and the melodic and harmonic dimensions.

This mapping occurs three-fold. Firstly, a player will have to know what subset (what MIDI channels) of its content (its corpus) maps to its internal melodic or harmonic dimensions, also called its «listening dimensions». Secondly, a source of influence (from an external MIDI input - a «MidiInfluencer» - or from the output of another player) will have to know how to map parts of its MIDI content to the influence's melodic and harmonic dimensions (these influence dimensions will be matched with the receptive player's «listening» dimensions). Finally the player must decide what part of its content is to be effectively played. The user will be able to control these three mappings in order to set precise interaction schemes.

For example, when creating a corpus from a polyphonic MIDI file (e.g. a string quartet with channels one to four assigned to the different instruments), the user could want the system to map notes from channel one (e.g. the lead violin) to the player's melodic listening dimension and notes from channels two to four (e.g. the remaining instruments) to its harmonic listening dimension. When a player, loaded

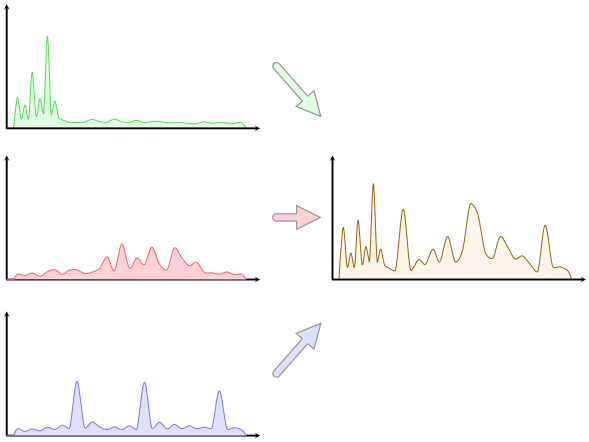

Figure 2.6: Three layers of peaks corresponding to different musical dimensions such as pitch, harmony, feedback, etc., being merged into one set of peaks before the final scaling and peak selection. Here, all three layers are weighted equally, but it is possible to balance the contribution from each of the layers.

with the said corpus, reacts to an incoming influence, it will continuously try to match the melodic and harmonic dimensions coming from the influence to is listening melodic and harmonic dimensions as mapped from the corpus. So, for example, if this player specifies channels two to four as output channels and listens to the melodic influence of a monophonic input from a musician, this means that the system will attempt to find slices where the content of channel one (e.g. the lead violin) corresponds to the input of the musician, while only outputting the content of channel two to four (e.g. the rest of the quartet). In this case, it would effectively generate an accompaniment to the input of the musician.

For a more complex input to the system (e.g. multiple musicians, a multichannel MIDI file, etc.) it is in the same manner possible to set the mapping between the input's MIDI channels and the melodic and harmonic dimensions used to generate influences to the system. This can be very useful in systems with multiple players that are listening to and influencing each other. Returning to the example with the string quartet corpus, we could create a system with two players: player P1, generating its improvisation from the melodic part of the corpus (e.g. the lead violin's part) and another player P2, generating its improvisation from the harmonic part of the same corpus (e.g. the rest of the quartet). We could now have P1 reactive to the melodic influence from a live musician; P2 could react to P1 by generating a harmonic texture coherent with P1's directions; P2 could also influence P1's progression choices on a harmonic basis, thus competing with the external musician's influence.

As shown in the above example, it is possible to create highly sophisticated networks of players influencing each other with detailed control over each player's listening and output dimensions. However, In simple cases (e.g. a single player with a

corpus constructed from a piano MIDI file consisting of one or two channels) it is not necessary to focus on this particular aspect of the system - the default settings[4] will suffice. While the "ideal" output in this case, as mentioned in the beginning of this section, would be an identical replication of the input, the discrepancies between the input and the corpus will almost always ensure that the output is much more dynamic than a simple replication. Still, being aware of these intricacies will, once you've familiarized yourself with Somax, be very helpful for configuring the system for specific situations and improving the quality of the output.

[4] By default, all channels are mapped to all dimensions in all players and influencers. In this case, the whole musical content will influence both the harmonic dimension as well as the melodic (by default, the highest note registered will feed the melodic dimension although you can specify otherwise, e.g. bass line, etc.)

Chapter 3

The Somax Back-end

This chapter presents the realization of the theoretical framework described in [3]. In particular, an overview of the back-end of the architecture, which is written in Python, as well as the implementations of certain algorithms of relevance to the model is presented here. This chapter describes the architecture that has been in use since version 2.0 of Somax, which due to the latest[1] version 2.3 contains some slight amendments that will be presented in chapter 4, which describes the updated front-end, as well as a multithreaded solution for the architecture described in this chapter. Regardless, what's presented in this chapter is still relevant for version 2.3 and necessary to understand before reading chapter 4.

3.1 Overview

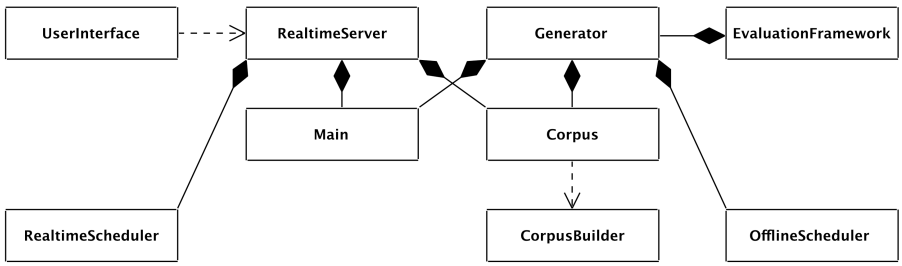

Figure 3.1: Module diagram over the main modules in the system and the relationship between them.

Figure 3.1 shows the different modules of the system and how they relate to each other. There are two main branches in this figure, one stemming from the `RealtimeServer` module, corresponding to the real-time (i.e. human-machine improvisation) framework, and one stemming from the `Generator` module, corresponding to the offline

[1] as of 2021-04-12.

(i.e. composition-oriented) framework. Both of them share the `Corpus` module (and its related `CorpusBuilder`), which handles the construction of corpora and will be described in section 3.2, and the `Main` module, which handles all the internal (runtime) logic of the system and will be described in section 3.3. The `RealtimeServer` and its related `UserInterface` module will not be specifically described in this chapter, as they were thoroughly described in [7], and the updates to the user interface since then will be presented in chapter 4. The logic of the `RealtimeScheduler` module, which handles the runtime scheduling of events over time, has however been significantly updated and will be presented in section 3.4.2.

In the branch stemming from the `Generator` module, the `Generator` itself will be described in section 3.4.4 along with its `OfflineScheduler`, which handles scheduling as an offline process and will be described in section 3.4.3. Note that this branch has not been actively maintained since 2020-06-30 and has thus not updated with regards to the changes implemented in chapter 4.

3.2 Corpus Builder

The purpose of the `CorpusBuilder` module is to construct the core of the system, the `Corpus`, from MIDI and/or audio files. It's an offline (as opposed to real-time) system that can be accessed both through a command-line build script as well as through the real-time user interface.

Another purpose is to achieve the format-agnostic behaviour of the runtime system as was described in [3], i.e. to ensure that there are no differences in how the runtime system handles corpora built from audio files in comparison to corpora built on MIDI files. However, as we will see in section 3.2.2, this behaviour isn't fully implemented in the current version of Somax[2]. For this reason, most of the behaviour described in this chapter will focus mainly on the MIDI implementation.

When building a corpus from MIDI files, the first step is to create a `NoteMatrix` class, which essentially is a matrix where each row correspond to a single MIDI note and each column correspond to pitch, velocity, channel, relative onset time (measured in ticks/beats since start of file), absolute onset time (measured in milliseconds since start of time), relative duration (measured in ticks), absolute duration (measured in milliseconds) and tempo. The rows are sorted by their relative onsets, i.e. their occurrences in time. This format is similar to the format used in [8] with the addition of tempo. The `NoteMatrix` class also stores any MIDI control changes and meta messages, as well as information regarding the original MIDI tracks, to allow reconstruction of the original MIDI file.

PITCH	VELOCITY	CHANNEL	ONSET (tick)	ONSET (msec)	DURATION (tick)	DURATION (msec)	TEMPO

Figure 3.2: Columns of the `MidiMatrix` class.

[2] As of 2021-04-12.

3.2.1 Slicing

Once the `NoteMatrix` is constructed, the next step is to determine the temporal boundaries for each slice, i.e. slicing. This is done differently for audio and MIDI as described in [3] - per beat for audio and per note for MIDI. In the case of MIDI file(s), to achieve the behaviour outlined in chapter 2.1, algorithm 1 is used. In this algorithm, the `NoteMatrix` is denoted $\mathcal{N}$, where $\mathcal{N}_i$ denotes the note at row i and `.absolute_onset`, `.relative_onset`, etc. denote the columns corresponding to indices in figure 3.2. The algorithm is iterating over all MIDI notes and creating a new slice for any note whose onset is more than $\varepsilon \in \mathbb{Z}_+$ millseconds from the previous note, otherwise appending the note to the current slice. The parameter ε is very important, as it will ensure that notes occurring sufficiently close (for example appoggiaturas and other articulation) are treated as part of the same slice, which both will be important for analysis later as well as maintaining the original rhythmic feel of the file without quantization. Another important aspect of the algorithm is line 14, which results in that any note in the previous slice that overlaps into the new slice will be added as well, and thus have an onset $\mathcal{N}_i$`.relative_onset` that may be earlier than the slice onset d_u. This will play an important role when scheduling slices, which is be described in section 3.4.1. The result of the slicing procedure is the `Corpus` class with all parameters set apart from its `Traits`.

Algorithm 1 Slicing a `NoteMatrix` $\mathcal{N}$ into a `Corpus` $\mathcal{C}$

1: $u = 0$
2: $\mathcal{C} = \{\}$
3: $\theta_u^{(\mathcal{N})} = \mathcal{N}_0$
4: $t_u^{(\mathcal{C})} = \mathcal{N}_0$`.relative_onset`
5: $\zeta_u = \mathcal{N}_0$`.tempo`
6: $\tau_u = \mathcal{N}_0$`.absolute_onset`
7: **for** $i = 1$ to $|\mathcal{N}| - 1$ **do**
8: **if** $\mathcal{N}_i$`.absolute_onset` $> \tau_u + \varepsilon$ **then**
9: $d_u = \mathcal{N}_i$`.relative_onset` $- t_u^{(\mathcal{C})}$
10: $\delta_u = \mathcal{N}_i$`.absolute_onset` $- \tau_u$
11: $\mathcal{S}_u^{(\mathcal{C})} = \left\{ t_u^{(\mathcal{C})}, d_u, \zeta_u, \tau_u, \delta_u, \theta_u^{(\mathcal{N})} \right\}$
12: $\mathcal{C} = \mathcal{C} \cup \left\{ \mathcal{S}_u^{(\mathcal{C})} \right\}$
13: $u = u + 1$
14: $\theta_u^{(\mathcal{N})} = \left\{ n \mid n \in \theta_{u-1}^{(\mathcal{N})},\ n$`.relative_onset` $+ n$`.relative_duration` $> d_{u-1} \right\}$
15: $t_u^{(\mathcal{C})} = \mathcal{N}_i$`.relative_onset`
16: $\zeta_u = \mathcal{N}_i$`.tempo`
17: $\tau_u = \mathcal{N}_i$`.absoulute_onset`
18: **else**
19: $\theta_u^{(\mathcal{N})} = \theta_u^{(\mathcal{N})} \cup \{\mathcal{N}_i\}$
20: **end if**
21: **end for**

3.2.2 Trait Analysis

In parallel with the slicing, two lowpass-filtered pseudo-spectrogram are computed from the `NoteMatrix` in the `Spectrogram` class, one for the foreground (melodic) channels and one for the background (harmonic) channels, which are both specified by the user. If no channels are specified, the foreground and background spectrogram will be computed from the entire set of channels and thus be identical. The procedure for computing the pseudo-spectrogram is the same as in [7], with the addition of the filter being interchangeable to allow different types of filtering, as well as no filtering at all. From these spectrogram, two pseudo-chromagram are computed in the `Chromagram` class, again one for the foreground channels and one for the background channels. For audio, these are computed directly from the audio data.

Once the `Spectrogram` and the `Chromagram` have been computed and the slicing procedure to create the `Corpus` is completed, the trait analysis begins. The trait analysis is dynamic, which means that it will import any class in the code base extending the `AbstractTrait` stereotype (see figure 3.3) and call the `analyze` function on each slice in the corpus.

```
class AbstractTrait(ABC):
    @classmethod
    @abstractmethod
    def analyze(cls, event: CorpusEvent, audio_data: np.ndarray,
                fg_spectrogram: Spectrogram, bg_spectrogram: Spectrogram,
                fg_chromagram: Chromagram, bg_chromagram: Chromagram,
                **kwargs):
        pass
```

Figure 3.3: The `AbstractTrait` stereotype, which is used to analyze all traits.

In the current state of the system, the following traits have been implemented:

Notes $\theta^{(\mathcal{N})}$ The midi notes contained in the current slice, as defined in algorithm 1. Note that at the moment, no corresponding values exist for audio data, which in other words means that the model isn't truly format-agnostic yet. Ideally, this could be solved by estimating these values with a polyphonic f0-estimator, for example [9].

Top Note $\theta^{(P_T)} \in \mathbb{Z}_{[0,127]}$ This value is simply the note number of the highest note in each slice, i.e.

$$\theta_u^{(P_T)} = \left\{ \mathcal{N}_i\texttt{.pitch} \mid \mathcal{N}_i \in \theta_u^{(\mathcal{N})} \wedge \mathcal{N}_j \in \theta_u^{(\mathcal{N})} : \mathcal{N}_i\texttt{.pitch} \geq \mathcal{N}_j\texttt{.pitch} \right\}. \quad (3.1)$$

Onset Chroma $\boldsymbol{\theta}^{(\boldsymbol{C})} \in \mathbb{R}^{12\times 2}$, which is the column in the `Chromagram` class (both foreground and background) at the index corresponding to the absolute onset of the slice, i.e.

$$\boldsymbol{\theta}_u^{(\boldsymbol{C})} = \begin{bmatrix} \boldsymbol{C}^{(\text{fg})}_{:,\tau_u} & \boldsymbol{C}^{(\text{bg})}_{:,\tau_u} \end{bmatrix} \quad (3.2)$$

where $\boldsymbol{C}$ denotes the `Chromagram` class as constructed in the previous step.

3.2.3 The Corpus

With the trait analysis completed, the `Corpus` class is finalized. Once again, it's important to emphasize the two purposes of the `Corpus` class: to (a) abstract the data of the audio/midi file(s) into high-level data that can be used for classification and (b) to create a format-agnostic object. The latter means that from this point, all raw MIDI and audio data as well as the spectrogram and chromagram will be thrown away. For MIDI data, this isn't a problem, as the `Notes` trait $\theta^{(\mathcal{N})}$ contains all the MIDI data - in fact, the `Corpus` and `NoteMatrix` are interchangeable, thus allowing re-export of the `Corpus` back to MIDI. For audio data, only a reference to the original file will be kept, thus the raw audio data corresponding each slice $\mathcal{S}_u^{(\mathcal{C})}$ can be reproduced by its absolute onset τ_u and its absolute duration δ_u. This compact data format also allows exporting the corpus as a JSON-file for quickly reloading previously built corpora.

Finally, while the `CorpusBuilder` module is the main way to construct a `Corpus`, it's not the only way. As we will see in section 3.3, a `Corpus` can be constructed from another `Corpus` during a real-time performance, and as we will see in section 3.4, it can also be generated from other corpora offline.

3.3 Runtime Architecture

The runtime architecture handles all the influencing and output generation, it's basically the core of the system. While most of it already has been explained in [7], some key aspects have been left out due to the article's condensed format, as well as some critical changes made since the article was written. For this reason, the design of the architecture and the relation between the algorithms described in [3] and the components of the system will be reiterated in the following section.

3.3.1 The Runtime System's Components

The architecture of the system can be seen in figure 3.4. At the root of the system is the `Player` class, through which all interaction with the system occurs. At the opposite end, at what could be considered the core of the system, is the `Atom`, where each `Atom` corresponds to one of the $r = 1, \ldots, R$ layers described in [3]. The `Atom` contains one `Classifier` instance, corresponding to a classifier $\Theta^{(r)}$, one `MemorySpace` instance, corresponding to a model $\mathcal{M}^{(r)}$ and one `ActivityPattern` instance, which handles storing, shifting, decaying and concatenation of peaks $\boldsymbol{P}^{(r)}$ as described in [3].

Inbetween the `Player` and the `Atoms` is the `StreamView` class. Each `Player` contains any number of `StreamViews`, which in turn is a recursive structure containing any number of `StreamViews` and any number of `Atoms`, effectively forming a tree structure where the `Player` correspond to the root of the tree, each `StreamView` correspond to a branch and each `Atom` correspond to a leaf of the tree. Each `Atom` and `StreamView` is assigned a (by the user controlled) weight $\alpha^{(r)}$ and at each branch in the tree, merging and user-controlled filtering Γ are performed by the `MergeAction`

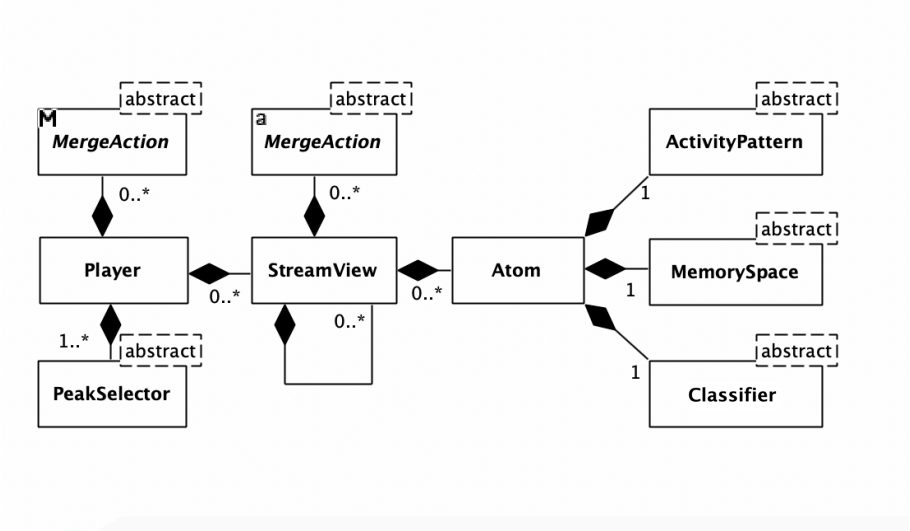

Figure 3.4: Simplified class diagram over the main components of the runtime architecture.

class. Finally, once all peaks have been merged up through the tree to the `Player` class, a final set of (user-defined) `MergeActions` are performed, and from that set of peaks, the output is selected by the `PeakSelector` class.

There are three main states in the runtime architecture, each roughly corresponding to the three first chapters of [3]: initialization (i.e. constructing the corpus), influence and output. In the initialization state, which is only performed once, the user defines the runtime architecture, i.e. the tree structure, which `Classifiers` (and `MemorySpaces` and `ActivityPatterns`) to use in each layer, as well as the `MergeActions` to use at each branch. This is also where the `Corpus` is built (or loaded from a previously built `Corpus`), clustered, classified and modelled in each of the `Atoms`. Note that while this step is mandatory for the initialization, the clustering, classification and modelling can be recomputed with different parameters in each of the `Atoms` while the system is running.

The two other processes, influence and output, takes turns continuously while the system is running and operate in opposite directions, where the influence state flows from the `Player` through the architecture, computing each of the steps defined in the chapter «Influence» of [3], ending in each of the `ActivityPatterns` where the generated peaks are stored, while the output process gathers all the generated peaks in each of the `ActivityPatterns` and merge them towards the `Player`, generating the output according to the steps in the chapter «Generate» of [3].

3.3.2 Modularity and Dynamicity

Similarly to the `AbstractTrait` stereotype defined in section 3.2.2, each of the components labelled with the `abstract` keyword in figure 3.4 can be substituted using each class' corresponding stereotype. For the `MemorySpace`, `ActivityPattern` and

```
class Parameter(HasParameterDict):
     def __init__(self, default_value: Ranged, min_value: Ranged,
                  max_value: Ranged, type_str: str,
                  description: str, setter: Optional[Callable]):
         # ...
```

Figure 3.5: The constructor for the `Parameter` class.

```
class AbstractClassifier(ABC):
   @abstractmethod
   def cluster(self, corpus: Corpus) -> None:
       pass

   @abstractmethod
   def classify_corpus(self, corpus: Corpus) -> List[AbstractLabel]:
       pass

   @abstractmethod
   def classify_influence(self, influence: AbstractInfluence) -> AbstractLabel:
       pass
```

Figure 3.6: Stereotype for implementing a `Classifier`.

`PeakSelector` classes, this behaviour is simply for future use and their stereotypes will not be discussed in detail. In this work, the stereotypes for the `Classifier` class and the `MergeAction` class are of greater importance, as a number of each have been implemented and will be described in section 3.3.3 and section 3.3.4 respectively.

Finally, among the novelties added to the system are the `Parametric` and `Parameter` classes (these are not displayed in figure 3.4, but all of the classes in the figure extends the `Parametric` class). From a software engineering perspective, addressing a parameter somewhere inside dynamic tree can be difficult, especially when communicating with an external client over a string-based protocol. The purpose of the `Parametric` class is to expose any `Parameter` to the user interface. In practice, this means that any class that extends the `Parametric` class can declare any of its user-controlled parameters as a `Parameter` class, and it will be immediately available in the user interface with a name, type, range, description and optional setter function. The constructor for the `Parameter class` can be seen in figure 3.5.

3.3.3 Clustering and Classification: the `Classifier` class

Clustering and classification is handled by the `Classifier` class. Each classifier is implemented by extending the `AbstractClassifier` stereotype shown in figure 3.6, implementing the functions `cluster`, `classify_corpus` and `classify_influence`. In practice, not all `Classifiers` rely on the `Corpus` for clustering - in fact some `Classifiers` don't implement the `cluster` function at all.

In the current state of the system, these classifiers have been implemented:

Top Note Classifier $\Theta^{(P_T)}$, which, as the trait $\theta^{(P_T)}$ already is a discrete parameter, simply is an identity classifier, defined so that

$$l = \Theta^{(P_T)}\left(\theta^{(P_T)} \mid \mathcal{C}\right) = \theta^{(P_T)}, \quad l \in \mathbb{Z}_{[0,127]}, \tag{3.3}$$

in other words, a `Classifier` without any clustering and thus independent of $\mathcal{C}$.

Pitch Class Classifier $\Theta^{(P_{12})}$, defined so that

$$l = \Theta^{(P_{12})}\left(\theta^{(P_T)} \mid \mathcal{C}\right) = \theta^{(P_T)} \bmod 12, \quad l \in \mathbb{Z}_{[0,11]}. \tag{3.4}$$

Again, a `Classifier` without any clustering and thus independent of $\mathcal{C}$.

SOM Chroma Classifier $\Theta^{(\boldsymbol{C}_{\text{SOM}})}$ A classifier of onset chroma vectors based on the original Somax implementation as defined in [6]. The clustering was computed using a self-organizing map on a matrix $\boldsymbol{X}$ of 3600 chroma vectors, i.e. $\boldsymbol{X} \in \mathbb{R}^{3600\times 12}$, returning a set of labels $\boldsymbol{l}^{(\boldsymbol{X})} \in \mathbb{Z}^{3600}_{[0,121]}$. The origin of these 3600 chroma vectors, as well as the exact parameters for the self-organizing map has unfortunately been lost, but this classifier will serve as an important base case when comparing different chroma classifiers.

As the self-organizing map itself can't be used for classifying corpora or influences, this classifier will simply select the label of the row in $\boldsymbol{X}$ minimizing the distance to the chroma vector $\boldsymbol{\theta}^{(\boldsymbol{C})} \in \mathbb{R}^{12}$ to classify, i.e.

$$l = \Theta^{(\boldsymbol{C}_{\text{SOM}})}\left(\theta^{(\boldsymbol{C})} \mid \mathcal{C}\right) = l_i^{(\boldsymbol{X})} \tag{3.5}$$

where

$$i = \underset{\boldsymbol{x}\in\boldsymbol{X}}{\operatorname{argmin}} \|\boldsymbol{x} - \boldsymbol{\theta}^{(\boldsymbol{C})}\|_2 \tag{3.6}$$

and $l_i^{(\boldsymbol{X})}$ denotes the label in $\boldsymbol{l}^{(\boldsymbol{X})}$ at index i.

Absolute GMM Chroma Classifier $\Theta^{(\boldsymbol{C}_{\text{AGMM}})}$ A classifier of onset chroma vectors based on a Gaussian Mixture Model clustering. The classifier uses the same matrix $\boldsymbol{X} \in \mathbb{R}^{3600\times 12}$ as the SOM Chroma Classifier for clustering, but with a user-defined number of clusters K. The initial clustering $\Theta_{i=0}^{(\boldsymbol{C}_{|\text{GMM}|})}$ is computed using K-means [4] with K clusters and the EM-algorithm iterated for (user-defined) I iterations. The classification is defined as x

$$l = \Theta_I^{(\boldsymbol{C}_{\text{AGMM}})}\left(\boldsymbol{\theta}^{(\boldsymbol{C})} \mid \mathcal{C}\right) = \underset{k\in 1\ldots K}{\operatorname{argmax}}\, p\left(C_I^{(k)} \mid \boldsymbol{\theta}^{(\boldsymbol{C})}\right) \tag{3.7}$$

where $C_I^{(k)}$ denotes cluster k after I iterations. Compared to the SOM Chroma Classifier, the main benefit with this is the variable number of clusters. Having a variable number of matches means that the precision of the classifier can be adjusted (where a higher number of clusters would mean a higher precision) at the cost of number of matches in the corpus (where a high number of clusters in most cases will result in less matches). This means that each performance can be parametrically tuned with regards to how well the corpus matches the input.

Relative GMM Chroma Classifier $\Theta^{(\boldsymbol{C}_{\text{RGMM}})}$ This classifier is identical to the Absolute GMM Chroma Classifier, but uses the data in corpus $\mathcal{C}$ to construct the matrix $\boldsymbol{X}$, and thus uses $\mathcal{C}$ for both clustering and initial classification. In other words, we have

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{\theta}_1^{(\boldsymbol{C})} \\ \vdots \\ \boldsymbol{\theta}_U^{(\boldsymbol{C})} \end{bmatrix}, \tag{3.8}$$

$\boldsymbol{X} \in \mathbb{R}^{U \times 12}$. This is the first classifier where the clustering is input dependent. Compared to the Absolute GMM Chroma Classifier, having a clustering dependent on the corpus can potentially result in very poor matches if the corpus is harmonically dissimilar to the input, as the classification algorithm will simply select the match with the highest probability (which then may be very low). But on the other hand, if the corpus and the input are harmonically similar, the precision in the matches may be much higher, even with a low number of classes, thus (ideally) resulting in a high number of matches with high precision.

3.3.4 Fuzzy Filtering: the `MergeAction` class

The scaling of individual peaks with regards to parameters of the peaks or their related slices $\mathcal{S}_u^{(\mathcal{C})}$ is handled by the `MergeAction` class, which only requires implementation of the `merge` function. The stereotype for this class is shown in figure 3.7, and there are currently two such fuzzy filters that the system makes use of:

Phase Modulation $\Gamma^{(\phi)}$ which scales the peaks with regards to their current phase/-position in the beat so that peaks occurring at phase close to the current phase of the output time $t^{(\mathcal{Y})}$ are emphasized and vice versa,

$$\Gamma^{(\phi)}\left(\boldsymbol{p}_i\right) = \begin{bmatrix} t_i^{(\mathcal{C})} & \phi_i y_i \end{bmatrix}^T \quad \forall \boldsymbol{p}_i \in \boldsymbol{P}_w \tag{3.9}$$

where

$$\phi_i = \exp\left[\cos\left(2\pi\left(t_w^{(\mathcal{Y})} - t_i^{(\mathcal{C})}\right)\right) - 1\right], \quad \phi \in \mathbb{R} \tag{3.10}$$

Next State Modulation $\Gamma^{(+)}$ which scales peaks close in time to the previously output slice $\mathcal{S}_{w-1}^{(\mathcal{Y})}$ by a constant α, i.e. for some $\varepsilon \in \mathbb{R}$

$$\Gamma^{(+)}(\boldsymbol{p}_i) = \begin{cases} \begin{bmatrix} t_i^{(\mathcal{C})} & \alpha y_i \end{bmatrix}^T & \text{if } \left|t_i^{(\mathcal{C})} - t_{w-1}^{(\mathcal{C})}\right| < \varepsilon \\ \boldsymbol{p}_i & \text{otherwise} \end{cases} \quad \forall \boldsymbol{p}_i \in \boldsymbol{P}_w. \tag{3.11}$$

3.4 Scheduling and the `Generator` Module

While the internal algorithms of the system has been thoroughly described by now, it has not yet been presented in context as a key aspect is missing - how input and

```
class AbstractMergeAction(Parametric):
        @abstractmethod
        def merge(self, peaks: Peaks, time: float,
                  history: ImprovisationMemory,
                  corpus: Corpus = None, **kwargs) -> Peaks:
            pass
```

Figure 3.7: Stereotype for implementing a `MergeAction`.

output is handled over time, i.e. scheduling. The following sections describe the `Scheduler` module, which determines how influences and triggers are scheduled to generate actual MIDI/audio output and the modes that the scheduler operate under. Section 3.4.4 describes the `Generator` module, which is using the scheduler to generate new corpora offline.

3.4.1 Scheduling

The main role of the `Scheduler` is to handle triggers to appropriately queue and output slices $\mathcal{S}_w^{(\mathcal{Y})}$ as they unfold over time, similar to a timeline in a DAW but where the events in the timeline are continuously generated by the system itself. The `Scheduler` has a running tick $t^{(\mathcal{Q})}$, a tempo $\zeta^{(\mathcal{Q})}$ and a queue of scheduled events $\mathcal{E}$, where each event has timestamp $t_w^{(\mathcal{Y})}$ and a predefined behaviour upon triggering, which depends on the event type. An event will be triggered when its tick $t_w^{(\mathcal{Y})}$ is greater than or equal to the scheduler tick $t^{(\mathcal{Q})}$. There are currently seven types of scheduled events:

`TempoEvent`: Sets the tempo of the scheduler to its value when triggered.

`MidiEvent`: Outputs a stored MIDI note on or note off message when triggered.

`AudioEvent`: Outputs an interval $[\tau_{\text{start}}, \tau_{\text{end}}]$ (in milliseconds) in the audio file to play over a duration determined by a tempo factor f_ζ, defined as

$$f_\zeta = \frac{\zeta_w^{(\mathcal{Y})}}{\zeta^{(\mathcal{Q})}} \tag{3.12}$$

where $\zeta_w^{(\mathcal{Y})}$ denotes the tempo of the audio event's corresponding slice $\mathcal{S}_w^{(\mathcal{Y})}$.

`CorpusEvent`: Outputs a slice $\mathcal{S}_w^{(\mathcal{Y})}$ when triggered. As we will see in sections 3.4.2 and 3.4.3, this behaviour seems to overlaps with the behaviour of MIDI and audio events, but they are never used in combination.

`InfluenceEvent`: Calls the influence process for a given `Player` with its stored value.

`TriggerEvent`: Corresponding to an event in the trigger stream $\mathcal{Y}$, which when triggered calls the generate process. The output of the generate process is sent back to the scheduler and queued, either as a `CorpusEvent` or as a `MidiEvent`/`AudioEvent`, depending on the type of scheduler, as we will see in the following two sections. In practice, this means that all `MidiEvents`, `AudioEvents` and `CorpusEvents` are queued only through a `TriggerEvent`.

There are two different ways to add `TriggerEvents` to the scheduler, which in turn depends on the scheduler's mode, which may be either `Automatic` or `Manual`. The `Manual` mode means that `TriggerEvents` are added manually, which in practice means that they are added by the system after every influence call. This is useful to create a note-by-note interaction between the system and the input. The `Automatic` mode means that new `TriggerEvents` are automatically queued after a duration corresponding to the generated output slice $\mathcal{S}_w^{(\mathcal{Y})}$, i.e. for a trigger $\mathcal{Y}_i$, a new trigger $\mathcal{Y}_{i+1}$ is added at $t_{i+1}^{(\mathcal{Y})}$ defined as

$$t_{i+1}^{(\mathcal{Y})} = t_i^{(\mathcal{Y})} + d_w^{(\mathcal{Y})}, \tag{3.13}$$

where $d_w^{(\mathcal{Y})}$ denotes the duration of the generated slice. Note that the retriggering uses the time of the trigger $t^{(\mathcal{Y})}$ rather than the time of the scheduler $t^{(\mathcal{Q})}$ when queueing new triggers to avoid drifting.

In practice, the scheduler is divided into two different classes, the `RealTimeScheduler`, which will be described in section 3.4.2, and the `OfflineScheduler`, which will be described in section 3.4.3. As we will see, these two have very little in common apart from the handling of `TempoEvents` and `TriggerEvents`.

3.4.2 Real-time Scheduling

The role of the `RealTimeScheduler` is in many ways similar to the of the audio thread in an audio plugin. It's behaviour is that of a high-priority thread that's continuously polled at a millisecond interval, at each poll i updating its tick so that

$$t_i^{(\mathcal{Q})} = t_{i-1}^{(\mathcal{Q})} + \Delta\tau_i \frac{\zeta^{(\mathcal{Q})}}{60}, \tag{3.14}$$

where $\Delta\tau_i$ denotes the number of milliseconds that have passed since the last poll, and triggering any event w whose tick $t_w^{(\mathcal{Y})} \geq t_i^{(\mathcal{Q})}$.

When the system is used as a real-time framework, the scheduler is based on the `asyncio` Python module, where influencing and setting parameters, as well as queueing new `TriggerEvents` and `TempoEvents`, is handled by a different thread corresponding to the ui thread in an audio plugin. The `asyncio` module is however not truly multithreaded, but rather handles ui calls in-between polling scheduler. These ui calls are blocking the thread and completes its operation before the next poll is called, hence eliminating any risk of tearing. While such a solution would not be acceptable for a real audio thread as the ui calls may delay the audio thread up to a few milliseconds, it's not a problem when handling an event-based stream as the delays incurred by this process generally are too small to be perceivable.[3]

Once a `TriggerEvent` has generated an output slice $\mathcal{S}_w^{(\mathcal{Y})}$, the real-time scheduler will extract its content as an `AudioEvent` for audio corpora or as a set of `MidiEvents` for MIDI corpora. For MIDI notes $\mathcal{N}_w$, great care must be taken when determining note ons and note offs, since we according to the slicing procedure would add a

[3] Do note that this behaviour is a simplification of the scheduling process occurring in each `Player`. As will be seen in chapter 4, Somax is using an actual multithreaded solution, but the behaviour of each individual thread is still in accordance with the description in this chapter.

single note to multiple slices. In practice, we generate note ons at $t_w^{(\mathcal{Y})}$ for any note $n_i \in \mathcal{N}_w^{(\text{on})}$, where the latter is defined as

$$\mathcal{N}_w^{(\text{on})} = \mathcal{N}_w^{(\mathcal{Y})} \setminus \mathcal{N}_{w-1}^{(\text{from})} \tag{3.15}$$

and note offs at $t_w^{(\mathcal{W})} + n_j\texttt{.duration}$ for any note $n_j \in \mathcal{N}_w^{(\text{off})}$, where the latter is defined as

$$\mathcal{N}_w^{(\text{off})} = \left(\mathcal{N}_w^{(\mathcal{Y})} \setminus \mathcal{N}_w^{(\text{from})}\right) \cup \left(\mathcal{N}_{w-1}^{(\text{from})} \setminus \mathcal{N}_w^{(\text{to})}\right) \tag{3.16}$$

where

$$\mathcal{N}_w^{(\text{to})} = \left\{ n \mid n \in \mathcal{N}_w^{(\mathcal{Y})} \colon n\texttt{.onset} < t_w^{(\mathcal{Y})} \right\} \tag{3.17}$$

and

$$\mathcal{N}_w^{(\text{from})} = \left\{ n \mid n \in \mathcal{N}_w^{(\mathcal{Y})} \colon n\texttt{.onset} + n\texttt{.duration} > t_w^{(\mathcal{Y})} + d_w^{(\mathcal{Y})} \right\}. \tag{3.18}$$

When using the system in real-time, the `RealTimeScheduler` doesn't handle `InfluenceEvents` or `CorpusEvents`. The former are handled directly by the ui thread and the latter are converted to `MidiEvents` or `AudioEvents`.

3.4.3 Offline Scheduling

The `OfflineScheduler` is unlike the `RealTimeScheduler` designed for a single thread, where any operation stems from the scheduler itself while running. It is also not continuously polled, but iterating over all the events $\mathcal{E}$ in the scheduler in order (where the iterator is being updated after each cycle to allow re-queueing of `TriggerEvents`) until the queue is empty. At each step i in the iteration, the tick $t_i^{(\mathcal{Q})}$ is updated so that

$$t_i^{(\mathcal{Q})} = \min_{t^{(\mathcal{Y})} \in \mathcal{E}} t^{(\mathcal{Y})} \tag{3.19}$$

where once again all events w whose tick $t_w^{(\mathcal{Y})} \geq t_i^{(\mathcal{Q})}$ are triggered in order, sorted by tick position as the first axis and type by the second, to ensure that `InfluenceEvents` are triggered before `TriggerEvents`, should they occur simultaneously.

Unlike the `RealTimeScheduler`, the `OfflineScheduler` will not handle `MidiEvents` or `AudioEvents` at all - it will output the slice $\mathcal{S}_w^{(\mathcal{Y})}$ corresponding to the `CorpusEvent` directly, effectively producing a new `Corpus`. But since the `Corpus` class is interchangeable with its MIDI and/or audio data, the generated result could easily be converted to a MIDI/audio file.

3.4.4 The `Generator` Module

The `Generator` is a separate module completely detached from the MaxMSP environment and the real-time system, and is designed around the `OfflineScheduler` to quickly generate new corpora. Similarly to section 3.3, the first steps when creating a `Generator` are to define and initialize the architecture and load a `Corpus`,

```
class Generator(ABC):
    def __init__(self, source_corpus: Corpus, influence_corpus: Corpus,
                 use_optimization: bool, gather_peak_statistics: bool,
                 name: Optional[str], **kwargs):
        # ...

    def run(self) -> Tuple[Corpus, Optional[PeaksStatistics]]:
        # ...

    @abstractmethod
    def initialize(self, **kwargs) -> None:
        pass
```

Figure 3.8: The signature of the constructor and `run` function as well as the `initialize` stereotype of the `Generator` class

which we from here on will call the source corpus. The `Generator` itself is an abstract class where the definition of the architecture is done by extending the class and implementing the `initialize` function, as can be seen in figure 3.8. But as the figure shows, there `Generator` requires two corpora in the constructor and returns a third corpus from the `run` function. These three correspond to the three main processes of Somax: a source corpus $\mathcal{C}$, an influence corpus $\mathcal{K}$ (instead of an influence stream as was defined it in [3]) and an output corpus $\mathcal{O} = \left\{\mathcal{S}_1^{(\mathcal{Y})}, \dots \mathcal{S}_W^{(\mathcal{Y})}\right\}$, constructed from the output slices $\mathcal{S}_w^{(\mathcal{Y})}$. In other words, it will build the architecture and source corpus $\mathcal{C}$ as usual, but it will also build an influence corpus $\mathcal{K}$ using the same procedure. Each of the slices $\mathcal{S}_v^{(\mathcal{K})}$ of the latter are then queued as `InfluenceEvents` in the `OfflineScheduler` at their corresponding ticks $t_v^{(\mathcal{K})}$, together with a `TempoEvent` constructed from the slice's tempo $\zeta_v^{(\mathcal{K})}$ and (if the mode is set to `Manual`) a `TriggerEvent`.

When the `run` function is called, the `Generator` will iterate over all events in the `OfflineScheduler` for as long as there are `InfluenceEvents` left in the queue and then stop, effectively producing a corpus $\mathcal{O}$ with the same duration (and hopefully same traits) as the influence corpus $\mathcal{K}$, using the slices of the source corpus $\mathcal{C}$.

The `Generator` module also allows the user to gather statistics about the peaks at each step in the iteration, which can be useful for evaluating the usefulness of the architecture from a performer's perspective.

Chapter 4

The Somax Front-end

The user interface of Somax, which was discussed briefly in [7], is implemented in the Max programming language. It was originally designed as a thin client, where all of the computation is handled on the Python server (apart from the real-time signal processing required for audio signals used as influences). As most readers likely are aware of, Max is a visual programming language where the default means of programming is by connecting objects using patch cords. In most cases, the readability of a Max program is determined by how easy it is to follow the cords throughout the program. The Somax user interface was originally designed with this in mind to promote readability on both micro and macro levels of the program, but is from version 2.3 using wireless communication (`send` and `receive`) between objects on a macro level. While this approach to some extent obscures the readability (or at least the global signal flow) of the system, the benefits are manifold. First of all, the architecture becomes easier for the user to extend - adding new players can be done with a single keypress - and objects can dynamically select which other objects to interact with without having to modify the architecture. This new architecture and its implications will be presented in section 4.1.

Another purpose of this redesign is to make the system integrable into Ableton Live. Somax can as a system be described as a «function» that reads one or multiple audio and/or MIDI streams and outputs one or multiple MIDI streams. For compatibility with Live, this has to be split into several smaller objects based on Live's syntax of «instruments» (function that reads one MIDI stream and outputs one audio stream), «audio effects» (function that reads one audio stream and outputs one audio stream) and «MIDI effects» (function that reads one MIDI stream and outputs one MIDI stream). Using a wireless architecture, this goal is possible to achieve for Somax. Do note that the wireless paradigm presented in this report is just a prerequisite for the Max for Live version of Somax; the latter it is still work in progress and will be presented in a future report.

To accommodate these changes, the Python back-end has been updated to drastically increase the performance when using multiple players. This will be presented in section 4.2.

4.1 The Wireless Max Architecture

The Max architecture currently consists of four main objects: `AudioInfluencer`, `MidiInfluencer`, `Player` and `Server`. The `AudioInfluencer` and `MidiInfluencer` read a continuous stream of audio or MIDI data respectively and perform the slicing and trait analysis steps described in [3]. In the user interface, the data resulting from this process is called an `influence`, which is routed to a player (and its Python back-end model) to compute the following steps in the influencing process. The `Player` object is essentially a client for the corresponding `Player` class in the Python back-end, which handles all of the runtime architecture described in section 3.3. Finally, the `Server` object handles communication with the `Server` class in the Python back-end, which is the root of the entire system, handling all players as well as the transport (i.e. the master clock of the event scheduling). So far, this is identical to the architecture presented in [7], apart from the fact that that there is no routing object in the middle, which is now rather handled wirelessly and will be described further down in this section.

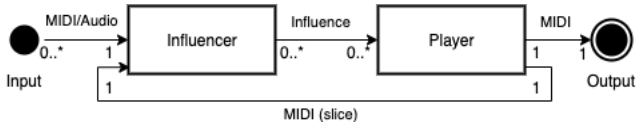

Figure 4.1: Generalized interaction model of the Somax user interface.

Figure 4.1 depicts the interaction model of the user interface on a large-scale level. Each input is sent into an influencer, which computes discrete influences from the data and forwards it to one or multiple players. An influencer may send influences to multiple players and a player may receive influences from multiple influences (hence, a many-to-many relationship). Each player generates an output MIDI stream based on said influences, which is also sent into an influencer to allow further influencing of other players with the generated output.

A simplified diagram over the entire wireless system can be seen in figure 4.2. Here we see that the only objects that do not have corresponding objects in the Python back-end are the influencers. Each influencer is given a name by the user, which the system ensures is unique, and this name will serve as the address on which the influencer sends its influences to players.

The `Player` object in Max was originally designed as a thin client around the `Player` class in the back-end, but has with this update been given two additional roles: routing and re-influencing. The routing module of the player (whose user interface is depicted in figure 4.3) lists all available influencers and players and receives influences based on what the user selects. The player also contains a `PlayerInfluencer`, which is a simple variation on the `MidiInfluencer` that also accepts certain metadata from the back-end to optimize performance and reduce latency, and give the user more detailed control when routing influences between players.[1] When initialized, the `Player` is assigned a unique (user-controlled) name as well as two unique OSC ports - one for receiving one for sending messages to the

[1] More details on advanced routing with the `PlayerInfluencer` can be found in [**?**].

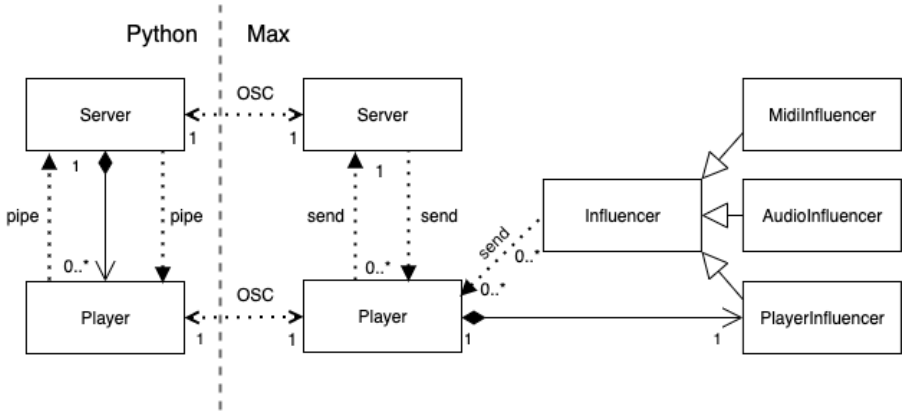

Figure 4.2: Interaction model for the wireless architecture. Dotted arrow lines denote some sort of "wireless" communication between objects while filled arrow lines denote their traditional UML relations (composition, generalization) and corresponding cardinality.

corresponding `Player` class. An instantiation message is sent to the `Server` object, which creates and owns the `Player` in the back-end, but once communication is established, the `Player` object and `Player` class communicate influences, MIDI output and parameter changes directly over said OSC addresses.

Figure 4.3: The routing module of each player. Here, all influencers and other players are listed, and the user can select whether to listen to pitch influences (P), onsets (O) and/or chroma influences (C) from that particular source. As chroma influences are continuous, it's also possible to mix chromas from different sources. The point of segmentation is determined by the chroma onset (CO).

Finally, the `Server` object, which directly corresponds to the `Server` class (in Python) and the root of the entire system, is also initialized with unique receive and send OSC ports as well as a unique name. In addition, it also contains the beat tracker module [5], a front-end module for the corpus builder , as well as a number

of utilities for managing and recording the MIDI output. The role of the `Server` class will be described in detail in section 4.2.

4.1.1 Additional Updates to the User Interface

In addition to the changes related to the wireless architecture, the user interface of each individual component has been redesigned from scratch. A Model-View-Controller design pattern has been applied to ensure that no inconsistencies exist between the Python back-end (model) and the Max user interface (view and controller). Each object has been remodelled so that multiple views/controllers may exist for the same data - one compact view, listing only the most vital parameters that are necessary to give the performer an overview while interacting with the system, and one full view where all parameters are available. This update also includes detailed (user-oriented) documentation of each object as well as a tutorial.

4.2 The Parallelized Python Architecture

Due to the changes presented in the previous section, the architecture of the Python back-end, originally implemented in a concurrent (but single core) manner, had to be updated to a parallel multicore solution. As shown in [3], a generate-influence cycle for a single player can in extreme cases take 30 milliseconds or more. In the case of a single player, this means 30 milliseconds of latency for the player, which of course is not ideal, but it will not break the perception of pulse assuming that the latency stays fairly constant. This is however not the case for multiple players operating on a single core. For example, if all three players receives their influence/onset messages simultaneously, the delay between sending the message and the output of the last player can be up to 90 milliseconds, while if influence/onset messages overlap perfectly, the delay for each player is only 30 milliseconds. This creates a latency interval for each player varying between 30-90 ms under even load (in this particular case, increasing linearly with each added player), effectively breaking all perception of pulse in the generated content.

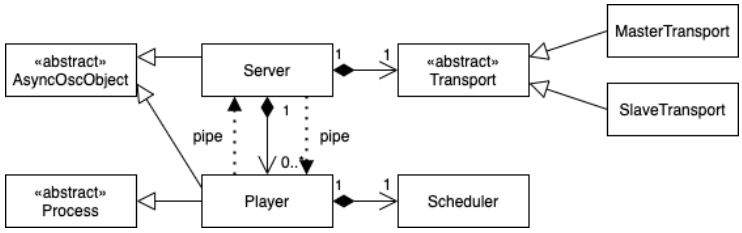

Figure 4.4: Simplified class diagram over the new parallelized architecture.

In the new architecture, shown in figure 4.4, which combines the concurrent strategies of the previous implementation with parallelization, each `Player` runs in its own `Process` [2], thereby not impacting each other in terms of performance, as long as there are fewer players running than free cores on the system. The `Server` is running two coroutines using the `asyncio` module [1], where one coroutine

continuously receives messages from the Max `Server` object over OSC and the other coroutine continuously updates the time of the `Transport` class and forwards this to each `Player`. Do note that there are two different `Transport`s available in figure 4.4 - the `MasterTransport` and the `SlaveTransport`. In the former case, the time (measured in ticks $t \in \mathbb{R}_+$ as in previous sections) is updated with each callback so that at update $i = 0, 1, \ldots,$

$$t_i = t_{i-1} + \Delta T \cdot \frac{\zeta_{i-1}}{60}, \tag{4.1}$$

where $\Delta T \in \mathbb{R}_+$ denotes the time (in seconds) elapsed since the last callback and ζ_{i-1} denotes the tempo of the scheduler at the previous time step. In the case of the `SlaveScheduler`, the idea is that the time is updated from an external source, for example the master clock of Max or the current time from Ableton Live, i.e.

$$t_i = t_i^{(\text{ext})}. \tag{4.2}$$

These messages are sent over OSC through the `Server` object (in Max) and the second coroutine of the `Server` class is therefore unused in this case. The time t_i is at each update i sent to all `Player`s of the `Server` using a pipeline.

The `Player` class is also running two coroutines; one coroutine continuously receiving messages from the Max (parameter updates, influences and onsets) and one coroutine continuously receiving message from the pipeline connected to the `Server`. As in the earlier architecture, these messages are queued through the `Scheduler`, which was described in chapter 2.1, with the only exception that each `Player` now has its own `Scheduler`, instead of a shared one for all `Player`s. Also, if the user has set a specific `Player` as the «Tempo Master», i.e. the source from which the `Transport` should receive its tempo, this data is returned to the `Server` through another pipeline and updated accordingly, e.g. for a tempo $\zeta_w^{(\mathcal{C})}$ returned by the `Player` after update t_i, the tempo of the `Transport` is updated so that

$$\zeta_i = \zeta_w^{(\mathcal{C})}. \tag{4.3}$$

Bibliography

[1] asyncio — Asynchronous I/O. https://docs.python.org/3/library/asyncio.html. Accessed: 2021-03-30.

[2] multiprocessing - Process-based parallelism. https://docs.python.org/3/library/multiprocessing.html. Accessed: 2021-03-30.

[3] The Somax Theoretical Model.

[4] Christopher M Bishop. Pattern recognition and machine learning. springer, 2006.

[5] Laurent Bonnasse-Gahot. Donner à OMax le sens du rythme: vers une improvisation plus riche avec la machine. École des Hautes Études en sciences sociales, Tech. Rep, 2010.

[6] Laurent Bonnasse-Gahot. An update on the SOMax project. Ircam-STMS, Tech. Rep, 2014.

[7] Joakim Borg. Somax 2: A Real-time Framework for Human-Machine Improvisation. Internal Report - Aalborg University Copenhagen, 2019.

[8] Tuomas Eerola and Petri Toiviainen. Midi toolbox: Matlab tools for music research. Department of Music, University of Jyvaskyla, 2004.

[9] Justin Salamon and Emilia Gómez. Melody extraction from polyphonic music signals using pitch contour characteristics. IEEE Transactions on Audio, Speech, and Language Processing, 20(6):1759–1770, 2012.