The Somax 2 Developer’s Documentation

Rev. 0.2.0

Joakim Borg

April 26, 2024

Credits

Somax 2 is a renewed version of the Somax reactive co-improvisation paradigm by G. Assayag. Architecture, UI and code completely redesigned and written by Joakim Borg in Max and Python.

Legacy:

- Early Java prototype by Olivier Delerue: adding reactivity to OMax.
- Versions 0.1 to 1.3 by Laurent Bonnasse-Gahot: conception of the reactive memory and influence dimensions model.
- Versions 1.4 to 1.9 by Axel Chemla-Romeu-Santos: separation of the Python server and object oriented design.

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).

`repmus.ircam.fr/impro`

Contents

Chapter 1

Overview

This report outlines the implementation of the Somax 2 library and is intended for future developers and maintainers of the project. A number of reports have already been published for the Somax project, more recently [10], published in 2021, which presents the theoretical foundation of the library, [9], published in 2021, which outlines the library's software architecture and [11], which describes a number of contributions added to the library since the two former reports were published.

The above reports serve as a theoretical, high-level foundation for the project and were intended for expert users and associated researchers who would like to modify or extend the behaviour of Somax without having to understand the whole code base. In some sense, they all provide answers to the question 'what?'. In contract, this report is intended for developers who needs full understanding of the entire code base, and will provide answers to the questions 'how?' and 'why?'. This means that the focus of this report is solely the lower levels of the code; it will not describe the theoretical model of Somax — the assumption here is that the reader is already well acquainted with these reports and knows how Somax works from a theoretical point of view. The reader is also assumed to have a good knowledge of Python and MaxMSP, trivial code (i.e. code that closely follows the standard in either of the two languages without any complicated optimizations, IO operations, multiple inheritance or other aspects that may make the code difficult to understand) will not be explained. While the question 'how?' can be fully answered by simply reading through the entire code base (and should be done by any future developer regardless), reading this report should greatly reduce the amount of time needed while doing so.

Finally, since the last update on the project was released in 2022 [11], a number of alterations have been made to the architecture. This report will therefore, whenever needed, outline these alterations. More specifically, everything up to and including version 2.5.0 of Somax will be covered by this report

1.1 Files & Folders

The folder structure of the Somax 2 repository [8] can be seen in figure 1.1. The `codesign/` and `media/` folders and the `Makefile` are only relevant for the compilation step (see section 2.6).

The `max/somax/` folder contains all the MaxMSP code and is structured as a Max package. When distributing releases of Somax (i.e. with the Python code compiled into an app bundle), this folder should be the root folder of the distribution (again, see section 2.6), and while developing/using Somax directly from the GitHub repository, only this folder should be added to the File Preferences in Max (or symlinked to the `Max 8/Packages` folder).

The `python/somax/` folder is structured as a PyPI package, with all of the Python code contained in the `python/somax/somax/` subdirectory and the normal `requirements.txt` for installing all of the Python dependencies.

The procedure for installing Somax as a developer is described in the `README.md` under "Manual Installation".

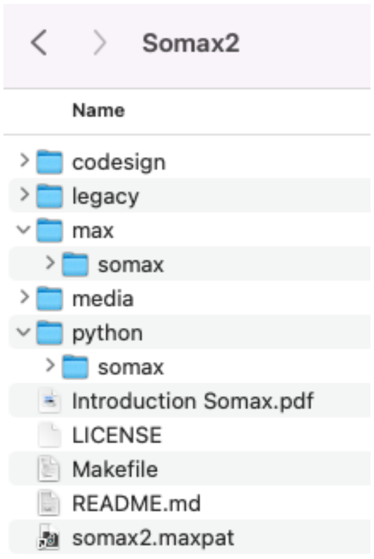

Figure 1.1: Structure of the Somax 2 repository

Chapter 2

The Python Architecture

2.1 Overview

The Python code base is the back-end of Somax and contains all of the generative aspects of the system, including modelling and classification, the implementation of the corpus and associated descriptors, scheduling and temporal behaviours, etc. The Python code base interacts with the Max front-end through OSC, where the Max front-end mainly handles real-time audio analysis and rendering, along with relevant visualization, as well as providing the user with an interface and/or GUI for interaction, but everything else is handled in Python.

A class diagram for the entire Python architecture is available in appendix A, and parts of this class diagram will be used throughout this chapter to illustrate certain parts of the architecture.

This chapter is divided into a number of sections, each outlining different aspects of the software architecture. Section 2.2 describes the entry points into the application and how the OSC communication and interaction between the core elements work, section 2.3 explains the architecture of the corpus, section 2.4 explains all aspects of the back-end related to time and scheduling, section 2.5 describes the architecture of the `Player`, i.e. the class that is responsible for the modelling of the corpus and matching influences, and finally section 2.6 will describe the current procedure of compiling the Python code into a standalone application for distribution.

2.1.1 Associated Libraries

In addition to standard dependencies (such as Librosa, Numpy, Scipy, etc., see `python/somax/requirements.txt` for a full list), there are two external libraries that were developed within the team specifically for the needs of Somax and its related DICY2 library [14], [15]. These are the MaxOSC library [4] and the GIG library [3].

The MaxOSC Library

The MaxOSC library was developed early in the 2.0 beta phase of Somax with the intention of facilitating OSC communication between Max and Python. It contains a number of classes for exposing a Python class' member functions so that they are directly callable over OSC using a Python-like syntax, as well as adding support for more complex types than the ones supported by the OSC protocol [6] (such as nested lists, hash maps, etc.)

The MaxOSC library was initially developed as two sister libraries: MaxOSC, the Python library described above, and PyOSC, a set of Max objects for facilitating the interaction in Max.[1] The latter was revisited during the work on the [3] (described below) and new externals for synchronous as well as asynchronous OSC calls to a remote, hierarchical class architecture[2], were developed, but ultimately not used in the current version of as the development of Somax took a different turn when the development of DICY2 ceased. The MaxOSC library is available on PyPI and can be installed directly through pip.

The GIG Library

The GIG library [3] was developed in 2022 with two goals in mind. Firstly, to provide a unified architecture for Somax and DICY2 so that research and improvements in either of the two would benefit the other, as well as ideally merge the two into a single framework of generative agents. Secondly, to use our experiences of working with the libraries to unify and solve a number of architectural issues with the current implementations, as well as to generalize a number of functionalities (such as OSC parsing, scheduling, classification, corpus building, etc.) and thereby provide easy-to-use classes that could be used in future projects.

Ultimately, the GIG library was not put to use in Somax as it was abandoned when the DICY2 development came to a halt, but it is referenced frequently in this report, as it provides solutions to a number of problems that still exist in the current code base of Somax, and could be used as a future reference. The rewritten version of Somax, based on the GIG library, is available on the `dev-merge-osc` branch of the Somax2 repository.

The GIG library is currently not available on PyPI, but it can be imported as a git submodule. See the `Dicy2-python` repository [2] for reference on how to use it.

2.2 Input/Output & Parsing

When using Somax with the Max front-end, the `somax_server.py` file contains the `main` function, and once launched, any other communication will be handled solely through OSC. The `main` function instantiates the `SomaxServer` class, which corresponds to the `somax.server` object in Max (see section 3.2).

The `SomaxServer`'s main role is to manage the application's `OscAgents`, where each `OscAgent` corresponds to a `somax.player` object (see section 3.3). It's also

[1]Note that the name of the python repository is `pyosc` for this reason, but only the subfolder `maxosc` is currently used in Somax

[2]See the `dev-connector-singleton` branch of the `pyosc` repository

responsible for the `Transport` object, which is the clock that synchronizes time between all the different players (this will be further described in section 2.4).

2.2.1 AsyncioOscObject

Both the `SomaxServer` and `OscAgent` inherit from the `AsyncioOscObject`, which is the class responsible for all OSC handling. This class uses an `asyncio`-based [1] UDP server for handling messages sent from the Max front-end, which means that it's able to have zero or more runtime callbacks that are processed in parallel with OSC I/O operations. Note that while `asyncio` in some sense allows us to write concurrent code, it's still effectively single-threaded, so there's no need to worry about race conditions, data races or other parallelism-related issues. This applies to all of the Somax library: while it relies on several parallelism-oriented libraries (both multiprocessing [5] and asyncio), it's designed so that no resources ever are shared and each component of the system can always be treated as if it was single-threaded.

The `AsyncioOscObject` inherits from the `Caller` class of the `maxosc` package, which automatically exposes all function of the class (and any class inheriting from it) so that it can be called directly over OSC using a python-like syntax. For example, a function like

```
def create_agent(self,
                 name: str,
                 recv_port: int,
                 send_port: int,
                 ip: str = "",
                 override: bool = False)
```

can be called over OSC using any of the following messages

```
/somax create_agent "Player1" 1234 1235 "127.0.0.1" 1
/somax create_agent name= "Player1" recv_port= 1234 send_port= 1235 override= 1
/somax create_agent name="Player1" recv_port=1234 send_port=1235 override=True
/somax create_agent "Player1" 1234 1235 override= 1
```

In other words, the syntax is similar to how a Python function normally is called but adapted to a Max environment (using spaces rather than commas, allowing spaces between argument names and values, e.g. `recv_port= 1234` rather than `recv_port=1234`, to avoid having to use the `combine` object in Max for named arguments, etc.) It's also possible to pass nested lists and/or dictionaries from Max with a Python-like syntax. e.g. for some Python function `f` that accepts nested lists, any of the following calls from max would be valid:

```
        /somax f []
        /somax f [ 1 2 3 4 ]
        /somax f [1 2 3 4]
        /somax f [ [1 2] [3 4] [ 5 6 ]]
```

Note that the OSC address is not used at all (it's `/somax` for all calls to the `SomaxServer` and the name of the agent for any call to a `OscAgent`, but both of these are redundant as each server and agent uses its own set of ports). This is because the underlying `python-osc` library does not support adding OSC addresses dynamically through

OSC calls (e.g. if we want to create an agent `agent1` at address `/somax/agent1` through the `create_agent` function called over OSC, this will simply not work using the `python-osc` library).

The `_main_loop` function is the only abstract function of the `AsyncioOscObject`. In the `SomaxServer`, this is implemented as a dynamic variable `self.loop`, which is bound either to the `__master_loop` or `__slave_loop` functions. At the moment, only the `__master_loop` is used, but the `__slave_loop` function was added in order to allow synchronizing the transport of the `SomaxServer` to an external clock, for example the Live transport (if using Somax in Max for Live device) or a transport in Somax. This is however not exposed in the front-end (i.e. the Max `somax.server` object) at the moment.

Also note that an improved version of the `AsyncioOscObject` exists in the GIG library (there called `AsyncOsc`), which has a number of subclasses for handling different types of async OSC communication.

2.2.2 SomaxServer & OscAgent

The central class of the Somax library is the `OscAgent`, which, as mentioned above, corresponds to the `somax.player` object. The `OscAgent` is initialized through the `SomaxServer`, but once initialized, the `somax.player` and `OscAgent` communicate directly through their own set of OSC ports. This class is responsible for handling all input/output to the `Player` (such as influences, output requests, setting parameters, etc.), as well as the co-ordinating time management between the `SomaxServer` and the `Player`, which is managed through the `SchedulingHandler`.

The `OscAgent` inherits from `multiprocessing.Process` class, which essentially is a way to bypass the global interpreter lock (GIL) in Python by running each `OscAgent` in its own subprocess rather than its own thread. Communication between the `OscAgent` and `SomaxServer` is handled through a `multiprocessing.Queue`, essentially a two-way pipe which passes `ProcessMessages` between the two objects. Almost all of the communication are messages related to time and scheduling and will be described in section 2.4[3].

The `SomaxServer` owns the `OscAgents` and is responsible for joining their processes once finished (e.g. when the server is shut down or if the `somax.player` is deleted in Max). In addition, the `SomaxServer` is also managing the `CorpusBuilders`, which will be launched as separate `multiprocessing.Processes` when the user is building a corpus (in order to not block the `Transport` while building corpora).

2.2.3 The Target class

Any output from the `OscAgent` and `SomaxServer` classes (i.e. messages to the front-end) are sent through the `Target` class, which is a light wrapper around the `pythonosc` UDP client (with some formatting for Max compatibility). In other words, all OSC communication is handled by the `OscAgent` and `SomaxServer` classes, so these are

[3]The only exceptions are the `PlayControl.CLEAR` and `PlayControl.TERMINATE` messages of the `ControlMessage` class, which calls `OscAgent.clear()` and `OscAgent.terminate()` functions accordingly.

the only two classes one needs to understand to fully understand all supported IO operations[4].

The `Target` will, just like the `AsyncioOscObject` return messages to a the same static OSC addresses mentioned above, and rather use a keyword for the OSC messages to the Max front-end (which is the common practice in Max). All of the keywords for the OSC messages sent from the `SomaxServer` and `OscAgent` are stored in the `ServerSendProtocol` and `PlayerSendProtocol` respectively. In other words, the entire protocol for OSC messages sent from the back-end to the front-end can be understood by reading through these two classes.[5]

2.2.4 Parsing & Exception Handling

As the `SomaxServer` and `OscAgent` classes are handling all I/O operations, they are also responsible for parsing the messages and ensuring that no invalid messages are accepted.

The Somax Python library relies heavily on type annotations, and will generally not check whether the correct type is passed; this is rather left to the developer/user to handle ("We're all consenting adults here", as the Python mantra goes). The one exception to this rule are the `SomaxServer` and `OscAgent` classes, as these handle input directly from a user in Max (through the `somax.server` and `somax.player` objects), who we cannot assume is necessarily passing the correct information, and who will need proper error messages posted in the Max console, rather than cryptic Python exceptions, when the wrong type of information is sent. Therefore, the `SomaxServer` and `OscAgent` have very strict type and bounds checking.

A number of functions in the `OscAgent` also makes frequent use of the fact that introspection is possible in Python. As many components of the Somax architecture are modular (`AbstractScaleAction`, `AbstractClassifier`, `AbstractActivityPattern`, etc.), Any class whose base class is inheriting from the `StringParsed` class can be parsed directly from a string corresponding to the class' name, assuming that they are located in the same module. For example

```
>>> AbstractScaleAction.from_string("nextstatescaleaction")
<somax.runtime.scale_actions.NextStateScaleAction object at 0x..>
```

This means that new classes can be added in the same module and be fully accessible through the OSC protocol without changing any part of the code related to the parsing. The only exception to this strategy is the `FeatureValue` class, where new features need to be added to the `somax/features/__init__.py`, as the classes to parse are spread over several modules.

Finally, it's worth noting that the `SomaxServer` is actually split in two classes: `Somax` and `SomaxServer`, and similarly the `OscAgent` in `Agent` and `OscAgent`. The

[4]There's one slight exception to this rule: the `ThreadedCorpusBuilder`. Once the process of building a corpus through the `ThreadedCorpusBuilder` has been launched, it does not communicate with the `SomaxServer`, but its return value, which is either the path of the built corpus or an error message, is sent through its own `Target`, which however is using the same ports as the `SomaxServer` and hence received directly by the `somax.server`'s `udpreceive` object.

[5]Similarly, the entire protocol for OSC messages from the front-end to the back-end can be understood by reading the function signatures of the `SomaxServer` and `OscAgent` classes.

intention with this distinction was to separate the parsing/OSC handling into a separate class, so that the `Agent` and `Somax` classes could be used directly in Python. The current separation is however not used and largely irrelevant — a proper and clean separation of the two can instead be found in the (abandoned) `dev-merge-osc` branch.

2.2.5 Parametric & Parameter

The final aspect of the I/O and parsing is the `Parametric` class. Since the configuration of the `Player` can change dynamically by adding/removing `Atoms`, `ScaleActions`, etc. it's necessary to be able to address parameters of said classes dynamically so that they can be changed from the front-end. This is handled by the `Parametric` and `Parameter` classes. The relationship between the two can be seen as a tree where `Parametric` are the branch nodes and `Parameter` are the leaf nodes, where the value of a `Parameter` can be set by addressing its position in the tree. For example, given a `Parameter` `ngram_length` of the `NGramMemorySpace` (which inherits from the `Parametric` class, where the latter is owned by an `Atom` of the player (also inheriting from the `Parametric` with the name `melodic`, the value of this parameter can be set by calling the `Player`'s `set_param` function:

```
/agent1 set_param melodic::_memory_space::_ngram_size 3
```

Note that the parameter tree needs to be re-computed when any `Parametric` (for example a `ScaleAction`) is added or removed. This is done through the `Parametric.parse_parameters` function.

2.3 The Corpus

The `Corpus` class is the material from which the `Player` generates all its content. A corpus is constructed by passing an audio file or one or multiple MIDI files to the `CorpusBuilder`'s `build` function, which will segment the file and attempt to analyze its with respect to all applicable `CorpusFeatures`. If an analysis fails (for example if the `CorpusFeature` only is designed for MIDI), it will simply ignore the given feature and move on to the next one. Understanding the procedure of building the `Corpus` shouldn't pose any problem for a developer with some experience in Python. It's also possible to build a `Corpus` by providing your own analysis to the `ManualCorpusBuilder`, which will read the analysis from a text file, see comments in the code for more info.

The `CorpusBuilder.build` will return an `AudioCorpus` or `MidiCorpus` depending on the type provided, which, when built through the `SomaxServer`, will be exported as a file. An `AudioCorpus` is exported as a `.pickle` file, while a `MidiCorpus` is exported as a gzip'ed json. Note that exporting a pickle is a far superior approach in terms of performance and file size, and the only reason why the `MidiCorpus` isn't using this format is because it hasn't been updated from its legacy json format yet.

The `Corpus` itself can be either a `AudioCorpus`, consisting of a list of `AudioCorpusEvents`, or a `MidiCorpus`, consisting of a list of `MidiCorpusEvents`, which in turn contains a list of `Notes`. In the `AudioCorpus`, each even corresponds to a segment of the audio as determined by the onset segmentation as computed by the `AudioCorpusBuilder`.

In the `MidiCorpus`, each event corresponds to all the MIDI data between two note ons[6], where the `Note._onset` and `Note._duration` attributes are stored relative to their parent `MidiCorpusEvent` (in other words, a note can have an onset of -2.5, meaning that it starts 2.5 quarter notes before the start of the given event, and therefore is held at the start of the given event). A number of convenient functions exist in the `MidiCorpusEvent` to get all the notes that starts/ends before/at/within the given event.

2.3.1 Features

Each `CorpusEvent` has a dictionary of `FeatureValues` (stored by type, e.g. `my_corpus.get_feature(MeanChroma)` will return the value of its `MeanChroma`) that was analyzed when the `Corpus` was built. There's currently a separation between `CorpusValue` and `CorpusFeature`, where the former only contains the actual value while the latter also includes the procedure to compute it from a list of `CorpusEvents` (`CorpusFeature.analyze`).[7]

The `CorpusBuilder` relies on the procedure described in section 2.2.4 for parsing all implemented `CorpusFeatures` for analysis. As the `CorpusFeatures` are split into separate modules, it's however necessary to import them in the `features/__init__.py` module to expose them to the automated parsing. So, in order to implement new `Features`:

- Implement the feature by inheriting from `CorpusFeature` (make sure to also inherit from `RuntimeFeature` to make it runtime parsable as an `FeatureInfluence`, if applicable) and implement its abstract methods
- If the feature is MIDI only or audio only, make sure to raise a `FeatureError` if it receives the wrong type of data. In this case, it will simply be ignored
- Import the feature in `features/__init__.py`

The next time a `CorpusBuilder` builds a new `Corpus`, the new feature will be available through `Corpus.get_feature`.

Note that when new features are added and the Somax library is extended with new functionalities relying on these new features, this might mean that corpora built in earlier versions of Somax may no longer work correctly. When loading a `Corpus`, from a file, the `Corpus.from_json` function uses the utility class `VersionTools` to check the version of the `Corpus` loaded corpus, and if it doesn't match the current version, it will print an explicit error. So if a new feature is added with the intention of being a mandatory part of any `Corpus`, the corpus version number, stored in the field `__version_corpus__` in `somax/__init__.py` should be incremented.

If the new features is optional, an alternative is to use the `ContentAware` class (described in section 2.3.2) to disable parts of the code that rely on the given feature.

[6]The MIDI segmentation process is described in [10].

[7]This separation was made to avoid a circular dependency between the `CorpusEvent` and the `CorpusFeature`, but a better solution to the problem is available in the `dev-merge-osc` branch.

2.3.2 ContentAware & Eligibility

In some cases, a feature may only be defined for a certain type of corpus. For example, the `VerticalDensity` feature, which corresponds to the number of MIDI notes in the given event, is undefined for an audio corpus. To still be able to use these types of features in the `Player` without crashing if a corpus of the wrong type is loaded, the `ContentAware` interface was implemented. Any class inheriting from the `ContentAware` interface will simply be bypassed when a corpus is loaded if `ContentAware._is_eligible_for` returns `False`. In practice, this is only used in a few experimental classes of the `Player`, such as `VerticalDensityScaleAction`, `OctaveBandsScaleAction`, etc. Refer to these for more info.

2.4 Scheduling

One of the most important responsibilities of the Python back-end is to manage time and scheduling. The current time, represented by a `Time` object, is managed by the `Transport` in the `SomaxServer`, and is synchronized between all `OscAgents`. The `Transport` is polled each millisecond in the `SomaxServer.callback()` function to update the `Time`, and then sends it to each existing `OscAgent` through its associated `multiprocessing.Queue` (see section 2.2.2). The `OscAgent` has its own `_main_loop` function where it receives these messages and updates time accordingly through its `Scheduler`. The current time is through the `Time` object represented both in ticks (where 1.0 ticks corresponds to one quarter note) and seconds, and has an associated tempo.

The `Transport` and `Scheduler` manage two different responsibilities: the `Transport` is the internal clock of the system, which manages how much time has elapsed since the previous callback, while the `Scheduler` is responsible for when each event should happen.

The `OscAgent` does not communicate directly with the `Scheduler`; the `OscAgent` has a `SchedulingHandler`, which has a `Scheduler`. The `SchedulingHandler` is an abstract base class with a number of different implementations, where the different implementations roughly corresponding to the `mode` attribute in the `somax.player` object, and is described in section 2.4.2.

2.4.1 ScheduledEvent

The `Scheduler` only accepts a single type of message: the abstract `ScheduledEvent` class, which has a `trigger_time`, which determines when the event should be output by the `Scheduler`. There are four main types of `ScheduledEvents`:

`TriggerEvent` When output, triggers output by the `Player` through the `OscAgent._trigger_output` function.

`RendererEvent` Contains OSC messages that will be sent directly to Max to trigger rendering of some sort (example subclasses are `MidiNoteEvent`, `AudioEvent`, `AudioOffEvent`, etc.)

`TempoEvent` Contains information about the last played `CorpusEvents` annotated tempo. If the `OscAgent` is set as the `SomaxServer`'As tempo source, it will send this message through the `multiprocessing.Queue` back to the server to set its tempo. If not, it will simply be ignored

`ContinueEvent` When output, triggers output by the `Player` through the `OscAgent._continue_output` function. The difference between `TriggerEvent` and `ContinueEvent` is described in section 2.4.4.

2.4.2 The SchedulingHandler

The `SchedulingHandler`'s main role is to receive `CorpusEvents`, convert them into the appropriate `ScheduledEvents` and schedule them accordingly through its internal `Scheduler`, which is handled by the `SchedulingHandler.add_corpus_event` function. When triggering output from a `somax.player` object in Max (e.g. through a `bang` or a `influence onset` message, which both call the `OscAgent.bang` function), this does not directly trigger output but rather schedules a `TriggerEvent` through the `SchedulingHandler.add_trigger_event` function. There are currently three different `SchedulingHandlers` with three different behaviours for `add_corpus_event` and `add_trigger_event`:

AutomaticSchedulingHandler Plays continuously by immediately scheduling a new `TriggerEvent` at the end of each `CorpusEvent` received. Any manual triggers called through the `add_trigger_event` will be ignored. This class corresponds to `@mode continuous` in the Max front-end.

ManualSchedulingHandler Schedules a single `TriggerEvent` immediately when the `add_trigger_event` is called, thereby interrupting any currently rendered events. Does not reschedule any `TriggerEvents` on `add_corpus_event`. This class corresponds to `@mode reactive` with `@cut 1` in the Max front-end.

IndirectSchedulingHandler Schedules a single `TriggerEvent` immediately when the `add_trigger_event` is called, unless it is in the middle of rendering an event, in which case it will schedule the trigger at the end of the current event. This class corresponds to `@mode reactive` with `@cut 0` in the Max front-end.

Note that while neither the `ManualSchedulingHandler` nor the `IndirectSchedulingHandler` will ever reschedule any triggers, they may still continue playing further events due to the `ContinueEvent`. These behaviours are explained in section 2.4.4.

The process of converting an `CorpusEvent` into a set of `RendererEvents` is handled by the `AudioStateHandler` (for an `AudioCorpusEvent`) and the `MidiStateHandler` (for a `MidiCorpusEvent`) classes. These classes are responsible for behaviours strictly related to the rendering, for example whether an audio renderer needs to jump in the buffer or may continue playing linearly, or in the MIDI case, to handle note offs, notes sustained over several slices, etc.

2.4.3 Synchronization & Time Stretch

Due to the separation between time management (`Transport`) and scheduling (`Scheduler`), time stretch is fairly simple to achieve, we can just apply a time stretch factor to the time received from the `Transport` and the scheduling will automatically adapt to the new, stretched tempo, assuming that we output the same information to the renderer. A naive approach would be

$$t_{\text{stretch}}[n] = t_{\text{transport}}[n] \cdot \gamma \tag{2.1}$$

where $n \in \mathbb{Z}_+$ indicate the current step in the callback, $t_{\text{stretch}}[n] \in \mathbb{R}_+$ the stretched time at step n, $t_{\text{transport}}[n] \in \mathbb{R}_+$ the `Transport`'s time at step n and $\gamma \in \mathbb{R}_+$ is the time stretch factor (i.e. 1.0 means no time stretch, 0.5 half the speed, 2.0 double the speed, etc.). Since the user may change γ over time, this would however produce discontinuities in time. To avoid this, we'll calculate the discrete integral over $t_{\text{transport}}$, i.e.

$$t_{\text{stretch}}[n] = t_{\text{stretch}}[n-1] + (t_{\text{transport}}[n] - t_{\text{transport}}[n-1]) \cdot \gamma[n] \tag{2.2}$$

where $\gamma[n]$ now indicates the current time stretch value set by the user at step n.

All of this is handled in the `SchedulingHandler`, and the time stretch factor can be set with the `SchedulingHandler.set_time_stretch_factor` function. There's however another way of achieving time stretch: by changing the server's (or more specifically, the `Transport`'s) tempo.

There are currently two modes for scheduling in Somax (super class `SchedulingMode`):

`RelativeScheduling` events are scheduled in ticks,

`AbsoluteScheduling` events are scheduled in seconds.

In relative time, which is only used by MIDI corpora, changes to the server's tempo will naturally result in changes to the playback speed, as one tick corresponds to a quarter note, the duration (in seconds) of a tick changes with the tempo. In absolute time, changes to the tempo will not change the playback speed (a second is still a second). Audio corpora only have absolute time, but may still adapt to the tempo of the server by computing the difference between its annotated tempo and the server's tempo, and adjusting the time stretch factor accordingly. The choice between the different `SchedulingModes` is not directly controlled by the user, but a result of whether the user chooses to synchronize to the server's tempo or to use a manual time stretch factor, as controlled by the `OscAgent.set_synchronize_to_global_tempo` function. In summary, we have four different cases:

Audio corpus, synchronize=False `AbsoluteScheduling` according to equation 2.2 with γ directly controlled by the user

Audio corpus, synchronize=True `AbsoluteScheduling` according to equation 2.2 with γ calculated as the quotient between the server's tempo and the last rendered `AudioCorpusEvent`

MIDI corpus, synchronize=False `AbsoluteScheduling` according to equation 2.2 with γ directly controlled by the user

MIDI corpus, synchronize=True `RelativeScheduling` without any time stretch factor, directly adapting to the server's tempo.

2.4.4 Timeout & ContinueEvent

One of the cornerstone of the Somax development process has been its modularity and encapsulation of functionality, meaning that in most cases, a certain behaviour or functionality, as controlled by the user, is limited to a single class and can therefore easily be understood by reading the code in that particular class. This approach also makes it simple to disable and/or modify the given behaviour, as the code that manages said behaviour self-contained and independent of other modules.

There's however one exception to this development principle, and that is the concept which is called "timeout". The basis for this idea is simple: when we're in a mode where output is user-triggered (i.e. when using the `ManualSchedulingHandler` or the `IndirectSchedulingHandler`), we don't want the output to immediately stop once it has finished playing an event, but we want it to continue for some time after.

In earlier versions of Somax ($\leq$2.4.0), this behaviour was achieved (for audio corpora) by simply letting the renderer continue playing the buffer linearly for N seconds after the last triggered event. While this solution was extremely simple, it came with a number of downsides: (i) there was no way to guarantee that the renderer wouldn't stop in the middle of an event, (ii) the user interface would not be able to indicate the currently played event during the timeout period[8] and (iii) playing linearly is in many cases unsatisfactory in terms of creativity.

For this reason, a new timeout behaviour was conceived to solve the issues above, i.e. (i) timeout should never stop in the middle of an event, (ii) the user interface should indicate events played during the timeout phase and (iii) it should be possible to play events in a non-linear manner (i.e. by jumping/recombining) during the timeout phase. This was implemented by adding a separate `ScheduledEvent` called `ContinueEvent`, which triggers the `OscAgent._continue_output` function (rather than the normal `OscAgent._trigger_output`, which is described in section 2.5).

Without prior knowledge, deciphering the exact control flow of the timeout behaviour throughout the many classes it applies to may be difficult, but the following summary should greatly simplify the procedure:

- Timeout only applies when using `ManualSchedulingHandler` or `IndirectSchedulingHandler`
- When `OscAgent.recombine=True`, `ContinueEvents` will trigger `Player.new_event`
- When `OscAgent.recombine=False`, `ContinueEvents` will trigger `Player.step`
- During the timeout period, the `Player` will bypass the `sparse` option of the `FallbackPeakSelector` due to the `enforce_output` parameter of the `Player.new_event` function
- When switching to a `AutomaticSchedulingHandler`, any `TriggerEvent` will be converted to a `ContinueEvent` and vice versa
- When an event is triggered by the user, the timeout period starts over

2.5 The Player

The role of the `Player` class is to select the most appropriate `CorpusEvent` at each time step when generation is called for. The `Player` has been discussed extensively

[8]A few notes on the syntax used here: By "timeout", we're referring to the point in time when the player stops, after receiving no new triggers. The "timeout period" refers to the time after the end of the triggered event during which the player continuous to play one or multiple events that are not directly triggered by the user. In other words, the sequence is the following: trigger(user)$\rightarrow$event$\rightarrow$timeout period$\rightarrow$timeout.

in [10] and [9], but a number of changes have been introduced since then. This section will give an overview of the main components and main functions of the `Player`.

The `Player` manages all of the modular components of the architecture that in turn determinate how the generation should work, and can be configured in a number of different ways (although only a single configuration is currently used in the `somax.player` object in Max). With few exceptions, the architecture of the `Player` is designed so that different behaviours are encapsulated in a single class, so that adding or removing a particular class will enable or disable a particular behaviour. These classes will be described in section 2.5.1.

In practice, to understand the full behaviour of the `Player`, it's sufficient to understand its four main functions and the classes used for passing information between these functions. These functions will be described in section 2.5.2.

2.5.1 Main Classes

The main classes managed by the `Player` are the `Atom`, `AbstractMergeAction`, `AbstractScaleAction`, `AbstractPeakSelector`, `FallbackSelector` and `TransformHandler` classes.[9]

Each `Atom` object corresponds to one layer of the generation, where each layer corresponds to one type of `Feature` of the `Corpus`. In the default `somax.player` configuration, these layers are `melodic`, which listens to pitch influences from Max, `harmonic`, which listens to chroma influences from Max, `self[melodic]` which through its `self_influenced` parameter listens to pitch influences of the previously output `CorpusEvent` through the `Player._feedback` function (which will be discussed in section 2.5.2) and `selfharmonic`, which listens to chroma influences of the previous output.

The `Atom` class manages three further objects: `Classifier`, `MemorySpace` and `ActivityPattern`. The `Classifier` is responsible for the strategy of analyzing incoming influences and converting them into discrete labels that the `MemorySpace` can use to discretely match the influences with the loaded `Corpus`, and the result is stored in the `ActivityPattern`, which is responsible for the state and peak-shifting strategy. All of this is thoroughly explained in [10], but its the classes used for communication between the objects will be briefly discussed in section 2.5.2 in terms of classes.

In earlier reports (e.g. [13], [9]), the `StreamView` class, an intermediate class between the `Player` and the `Atom`, was mentioned. Since Somax version 2.2, this class has been removed as it was never used in practice and added significant overhead in terms of performance. Now the `Player` directly manages all of its `Atoms`.

The `MergeAction` is responsible for combining all of the peaks from all `Atoms` when generating output through the `Player.new_event` function (see 2.5.2). Currently, only a single implementation of the `MergeAction` exists (`DistanceMergeAction`), which will merge peaks close to each other in time into a single peak. The algorithm is described in detail in chapter 4.2 in [10]. This procedure is necessary since

[9] In general, any class named `Abstract<ClassName>` is a modular component of Somax and has a number of implementations in the same module that represents a certain strategy for the part of the generation process that the base object corresponds to.

the peaks are stored and shifted in continuous time, and we don't want all peaks throughout the entire duration of an event to be merged into a single peak, as this would introduce a strong bias towards longer events. But it's important to be aware of, as it is one of the most expensive operations of the runtime architecture of Somax.

In the previous theoretical report [10], a distinction was made between merging peaks (chapter 4.2 in [10]) and scaling peaks (4.3 in [10]), but the software architecture did not portray this distinction, both of these roles were managed by the `MergeAction` class [9]. This has since then been changed, the `MergeAction` only manages peak merging (as described in the previous paragraph), while a new class, the `ScaleAction` manages all aspects of conditional filtering and scaling of peaks. A `ScaleAction` may change the score of a certain peak based on how well it matches a condition (e.g. if it represents the next consecutive event in the `Corpus` or how well it matches some condition regarding the energy/amplitude/duration/etc. specified by the user), but it should never remove or add any peaks. Peaks with a score of 0 will however be removed once all `ScaleActions` have been applied, so the implemented behaviour of any `ScaleActions` should be commutative in order to ensure that peaks whose score is set to 0 by any `ScaleAction` remain at 0 so that they will be removed. In summary, one could say that the `Atom` is responsible for creating peaks based on a binary condition ("does it match or not?"), while the `ScaleAction` is responsible for weighting existing peaks based on how well they match certain conditions.

The `PeakSelector` is responsible for selecting a `CorpusEvent` to output, corresponding to one of the peaks it receives. If there are no peaks (either because all peaks were filtered out by the `ScaleActions` or because no peaks existed in the `Atoms` in the first place), the `FallbackSelector` will be called to select a peak directly from the `Corpus`, if applicable.

Finally, the `TransformHandler` is responsible for managing `Transforms` (transpositions) in the architecture. When a `Transform` is added or removed by the user, this will trigger a number of behaviours in any aspect of the system related to matching (e.g. in the `NGramMemorySpace`, which keeps a history of the last N matches for each transposition).

The `TransformHandler` is also responsible for converting a `Transform` into an index, so that it can be stored and used in an optimized way in a numpy `ndarray` in the `Peaks` class. Originally, when transforms were added to Somax (which dates back to the pre-2.0 beta versions developed by Axel Chemla–Romeu-Santos), it was envisioned that transpositions would just be one type of transform, and that many other options would be possible. For this reason, the `TransformHandler` stores each `Transform` by an index rather than an absolute value (e.g. a transform of +2 semitones is not necessarily stored as the value '2', but could be any value depending on the number of transforms that have been added or removed before the +2 transform was added). The theory and application of transforms are thoroughly described in chapter 5 in [10].

2.5.2 Main Functions

The four main public functions of the `Player` are `Player.read_corpus`, `Player.influence`, `Player.new_event`, and `Player.step`. The reader should be able to gain a full understanding of the generative aspects of Somax by understanding those functions. In addition, the main classes used internally by these functions are the `AbstractInfluence`, `AbstractLabel` and `Peaks` classes.

Player.read_corpus

`Player.read_corpus` is called when a `Corpus` is loaded by the user, and will construct a model in each `Atom`'s `MemorySpace`, by in each `Atom` analyzing the `CorpusFeature` relevant for that particular layer in the `Classifier`, which converts the sequence of `CorpusFeatures` (one per `CorpusEvent` into a sequence of `Labels`, that the `MemorySpace` builds its model from.

Player.influence

The `Player.influence` is called each time the `OscAgent` receives a single influence from Max. When called, it updates the state of the targeted `Atom` (or more specifically the `Atom`'s `ActivityPattern`), where the `Peaks` are stored, but will not trigger any output.

When an `Atom` is influenced, it will first shift its stored `Peaks` based on the amount of (`Transport`'s) time that has passed since the last time it was shifted. This is followed by classifying (through its `Classifier`) the `FeatureInfluence` into a `Label` (or more specifically, one `Label` per active `Transform`), which is used by the `MemorySpace` to generate a list of `PeakEvents`, which are inserted into the currently stored `Peaks` object in the `ActivityPattern`. The `Peaks` are stored as three sequences of `np.ndarray` of equal lengths, where a particular index in one of the arrays is associated with the same index in any of the other arrays, but the entries are not sorted in any way, and multiple indices corresponding to a single `CorpusEvent` is possible. The `Peaks` are stored by time, as most `ActivityPatterns` (e.g. the `ClassicActivityPattern`) shift peaks continuously in time, so the original association to a particular `CorpusEvent` is lost at this point, as the peak is stored by its time rather than by its index.

Player.new_event & Player.step

The final set of functions are `Player.new_event` and `Player.step`, which both take the current state of the `Player` into account to select the next `CorpusEvent` that should be played.

The `Player.new_event` collects all of the `Peaks` (after shifting them based on the amount of time that has passed since the previous shift) from each `Atom` (or, again, more specifically from the `ActivityPattern` of each `Atom`), merges them through the `MergeAction.merge` function, scales them through all of its `ScaleActions`, and finally selects from the `Peaks` which corresponding `CorpusEvent` to output through the `PeakSelector`. In case no peaks exist in the `Peaks` object, the `FallbackSelector` is used to select an appropriate `CorpusEvent` from the entire `Corpus` (typically the one following the previously output event, but a number of conditions exist here).

Finally, once the `CorpusEvent` has been selected, the `Player._feedback` function will be called. This function will inform all of the `Player`'s subcomponents about which event was output so that they can update the state if necessary. This is also where all the `self_influenced` `Atoms` are influenced.

The `Player.step` function is similar to the `Player.new_event` function in terms of purpose, i.e. to play the next `CorpusEvent`, but will bypass everything related to `Peaks` and simply play the consecutive event in the `Corpus`.

2.6 Compilation

In order to facilitate the installation process for Max-oriented end users, the Python code is with each release "compiled" into a standalone application so that the user won't have to install Python to run Somax. This procedure is done through PyInstaller [7], which packages the Somax code along with the Python interpreter and all necessary dependencies into a bundle. In other words, the code is not compiled, but rather packaged into a self-contained bundle.

2.6.1 Why PyInstaller?

There are a number of libraries for packaging (or transpiling) Python code into standalone applications or libraries, for example Py2app, cx_freeze, PyOxidizer and Nuitka, to name a few. The reason why we choose to use PyInstaller was simply because at the point in time when the Somax packaging procedure was conceived, PyInstaller was the only library that supported all of the following: (1) MacOS notarization (2) the Python multiprocessing library, (3) backwards compatibility with MacOS 10.13 High Sierra (which at this point was decided to be the oldest version Somax should support).

It's however possible that this has changed over the past years, and that PyInstaller may no longer be the only viable nor the best option. The procedure outlined in this section is merely one way to package the Somax library into a standalone application that is known to work, but the reader is free to use any other procedure.

2.6.2 Procedure & Requirements

PyInstaller does not support cross-compilation, neither in terms of versions of MacOS nor Windows.[10] One of the curiosities of PyInstaller is that in terms of (major) MacOS versions, it's forward compatible but not backward compatible. It's therefore necessary to package the bundle on the oldest version that should be supported. In our case, this is currently MacOS High Sierra (10.13), but this should probably be revised in the future. The Somax bundle when packaged on High Sierra has proven to work on High Sierra (10.13), Mojave (10.14), Catalina (10.15), Big Sur (11), Monterey (12) and Ventura (13).

High Sierra does however not support notarization, as this feature was added in Catalina, so the bundle needs to be notarized on a machine running MacOS Catalina

[10]Note that only the procedure for building a MacOS .app bundle is outlined here. There's currently no official Windows release of Somax, but it is possible to run Somax on Windows with some caveats. See `https://github.com/DYCI2/Somax2/wiki/Somax2-on-Windows` for more information.

or later (recommended Big Sur or later due to significantly easier to use `notarytool` that was added in Big Sur)

The same issues with backward compatibility applies to architectures: PyInstaller bundles created on `x86_64` (Intel) will run on `arm64` (M1, M2, ...) processors, but not the other way around.

In other words, the procedure for building can be summarized as follows: (1) package the bundle using PyInstaller on an Intel machine running MacOS High Sierra, (2) notarize the bundle on any machine (Intel or ARM) running Big Sur or later. The entire procedure for building is outlined in the `Makefile` in the root folder of the Somax2 repository, but will also be described below. Note that it’s possible to to complete all of the steps on a single machine by running multiple versions of the OS in virtual environments (Parallels, VMware Fusion, VirtualBox, etc.).

Bundle Requirements

The following requirements are by no means exhaustive, but are known to work for bundling the package using PyInstaller on MacOS High Sierra (i.e. Step 1 below, all of the following steps can be done on any machine running MacOS Big Sur or later):

Host Machine: Intel x86-64

MacOS Version: High Sierra 10.13.6

Python Version: Python 3.9.5 (local `brew` installation, i.e. `brew install python@3.9`)

PyInstaller Version: PyInstaller 4.10

Dependencies: See `Somax2/python/somax/requirements-pyinstaller.txt` (i.e. `pip3 install -r requirements-pyinstaller.txt`)

Step 1: PyInstaller Bundle

The PyInstaller bundle is created by running `make pyinstaller` from the root folder on the High Sierra machine/VM. This requires the Ircam developer’s certificate (`"Developer ID Application:  INST RECHER COORD ACOUST MUSICALE"`) to be available on the machine.[11]

Step 1.1: Re-Signing the Bundle

Due to an issue with the `sklearn` package (which is a dependency of `librosa`) on High Sierra, we will have to re-sign the bundle after building it. This is done by running `make codesignature` (on any version of MacOS / any architecture). If, in the future, the minimum MacOS version supported should change to Mojave or later, this step can be ignored.

[11]The procedure of obtaining/using the Ircam developer’s certificate will not be described here, but there are a number of internal Ircam documents describing this as well as the procedure for properly signing/notarizing an application, a `dmg` and/or a `pkg`.

Step 2: Notarizing the Bundle

On a machine running MacOS Big Sur or later (on any architecture), run `make notarize` to notarize the built application. For this step to work, you will need to register your app-specific password[12] under the profile name `repmus` (or adapt the `Makefile` to match your registered app-specific password) using `xcrun notarytool store-credentials`.

Step 3: Creating the Max Package

The final step in the procedure is to create the Max package, containing the signed and notarized bundle. Normally when we launch the server from Max (e.g. through the `Start Server` button of the `somax.server.app` or the `initialize` message to the `somax.server` from the GitHub repository, this will launch the `launch_local` script in the `max/somax/misc` folder, which runs the `somax_server.py` through the local Python installation. In the Max package, we want this to point to the packaged `somax_server.app` instead. To do this, open the `somax.interpreter.maxpat` and change the `loadmess 0` into `loadmess 1`.

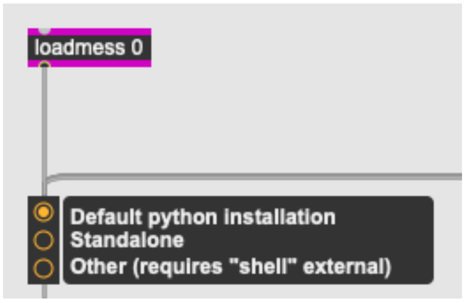
loadmess 0
Default python installation
Standalone
Other (requires "shell" external)

Figure 2.1: `somax.interpreter.maxpat`

Once this is done, run `make max-package` to create the final dmg containing the Max package. This step currently requires the `create-dmg` formulae that can be installed through `brew`. It's however not mandatory to use this, it will simply resize the installed dmg to a good format and change the background image, and all of these steps could be done manually.

Also note that should any of the steps fail, there's are two commands to clean up: `make clean` and `make clean-all`. `make clean` will remove the `build` folder and other files related ot the building step, so if for example step 3 fails, running `make clean` will reset the build to the state it was in at end of the previous step (i.e. after step 2 has succeeded). Running `make clean-all` will in addition remove the `dist` folder, which contains the signed application, and will therefore reset the state so that the build procedure must be restarted from step 1.

[12]For more info on generating an app-specific password, refer to the internal document "Développement Apple à l'Ircam".

2.6.3 PyPI Package

The `Somax2/python/somax` folder is structured as a PyPI package, but recent versions of Somax have not been published to PyPI due to lack of interest in the Python code. Should this change in the future, it's possible to create new distributions through the normal procedure of PyPI packaging, e.g. (after incrementing version, etc.):

```
python3 setup.py sdist bdist_wheel
twine check dist/*
twine upload dist/*
```

Chapter 3

The Max Architecture

3.1 Overview

The Somax Max package is responsible for two things: It serves as the front-end/API for the end user and it's responsible for handling every aspect of real-time signal processing. Everything else is handled by the Python code base.

At the core of the Max package are the `somax.server` and `somax.player` objects, both which are essentially wrappers around one pair of `udpsend-updreceive` each, through which they manage all communication with their corresponding `SomaxServer` and `OscAgent` Python objects respectively. These two objects have been designed with a single goal in mind: to behave as if they were native Max objects.

One explicit requirement for the entire project has been that as much of the code as possible should be written only in Python and Max. Since Somax is a research project, it was considered likely that future members and developers on the team mainly would have experience with Python and Max, and that it for this reason could be difficult to maintain the code in the future if a significant portion of it was written in C/C++.

For this reason, the entire Max part of the Somax code base has been written as abstractions rather than externals. While this does increase the transparency for Max developers, it comes with a number of downsides that all make the above mentioned goal more difficult, that will be discussed thoroughly in this chapter.

This chapter assumes that the reader has good knowledge of Max and will not explain basic aspects of Max that it frequently relies on (e.g. `pattr`, `bpatchers`, the distinction between `patchers` and `abstractions`, the Max threading model, etc.), but rather focus on the code base itself in order to give an overview

3.1.1 The Max Package's Structure

The Somax package consists of more than 50 abstractions, but most of them are used internally in a few core object, rather than being relevant for the end user. In this chapter, we will only focus on these core objects, as most of the abstractions can easily be understood from their corresponding contexts.

The base objects of the package are:

`somax.server` which handles the communication with the Python server

`somax.player` which handles the communication with a particular `OscAgent` on the server

`somax.audiocorpusbuilder` which is the front-end to the `CorpusBuilder` when used to create a `Corpus` from an audio file

`somax.midicorpusbuilder` which is the front-end to the `CorpusBuilder` when used to create a `Corpus` from a MIDI file

`somax.audioinfluencer` which is used to analyze a real-time signal in the same manner as the `Corpus` is analyzed when built, segmenting the signal into discrete, symbolic data that are used to influence the `Player`

`somax.midiinfluencer` same as above but for real-time MIDI streams

`somax.audiorenderer` which converts the symbolic messages received from the `OscAgent` into real-time audio data when using an audio `Corpus`

`somax.midirenderer` same as above for a for an `OscAgent` with a MIDI `Corpus`.

These objects are intended for any user who wishes to use any part of the Somax library in a modular manner. Two further variations exist for a number of those objects. `somax.<object>.ui` and `somax.<object>.app`. The former is for each object just a condensed UI suitable for using in a bpatcher, but in terms of messages and functionality identical to the core objects. The latter set of objects are application-style bpatcher objects designed for users with little or no experience in Max, where all of the functionality, including input and output, is encapsulated, with little room for the user to customize the design/behaviour of the object. The `.ui` objects that exist are `somax.player.ui`, `somax.audioinfluencer.ui` and `somax.midiinfluencer.ui`. The `.app` objects that exist are `somax.player.app`, `somax.server.app`, `somax.audioinfluencer.app` and `somax.midiinfluencer.app`. The latter set of objects will be further described in section 3.4.4, while all other sections mainly will focus on the base objects (i.e. the objects with neither an `.ui` nor `.app` extension).

3.1.2 Design Guidelines

To achieve the goal mentioned above, i.e. that all base objects should behave as if they were native Max objects, or more specifically as if they were well-designed, compiled externals, the following set of guidelines were developed:

Clear Separation Between Attributes and Messages

An attribute of an object is a setting or property that tells the object how to do its job. While this is a fundamental principle of modern Max objects, it's quite difficult to achieve for objects whose state actually is stored on a remote server, that may or may not have been started when the object is created. This will be discussed in section 3.1.3.

In addition, passing attributes to abstractions comes with a number of issues in Max, which will be discussed in section 3.1.4

Errors Should Be Explicit

Passing incorrect messages, values of the wrong type or attributes that don't exist to an object should always produce an explicit error (typically printed to the Max console), and if the message is an important part of a control flow that the user may rely on, the object should output a message or error code to indicating the error, so that the user can react accordingly. For example, if the `somax.audiocorpusbuilder` fails to build a corpus, a message indicating what went wrong will be printed in the Max console as well as the error code being output on the first outlet of the `somax.audiocorpusbuilder`, so that the user is able to react programmatically when this happens.

All Attributes Should Be Exposed to Pattr

Any attribute that the user may want to control should be exposed to the Max `pattr` system in order to allow the user to store, load and interpolate between states and configurations of the different objects.

Avoid Global State

The usage of global state (such as global `value`, `send` and `receive`, `dict` and `coll`, etc.) should be avoided at all costs and only used when absolutely necessary. By global, we mean state that is shared outside of a given abstraction, i.e. any send/receive/dict/etc. without `#0` in its name. The reason for this is simple: global state makes the code so much more difficult to read.

With that said, there are still a number of cases where global state is necessary in the Somax library. These will be discussed in section 3.1.5.

3.1.3 OSC Communication Issues

All of the communication between Max and Python is handled through (UDP-based) OSC. There are several benefits of choosing UDP-based OSC, it's efficient for real-time messaging and it's compatible with a number of different environments (most programming languages and music-oriented coding environments have actively maintained OSC modules). But most of its downsides are highly problematic for our use case: the fact that we cannot be sure whether the message was received, that there's no synchronous sending in Max, that messages may get silently dropped if the socket is full, and finally that we cannot send objects larger than a couple of kilobytes (size varies slightly between implementations).

Most of these issues are however only problematic in the initialization step, when establishing a connection between Max and Python, creating the remote representations of the `somax.player` and loading a corpus. For runtime messages, if a message should be dropped, this would in the absolute worst case result in a second or two of silence (which likely would sound like a musical pause), but in most cases, it wouldn't have any real consequences at all. For the initialization, however, there are a number of things that could go wrong. For example, if a user sends a message from Max to a remote `Player` (for example setting a certain parameter) while the remote `Player` still is in the middle of initializing and has yet to establish the UDP connection, the message sent by the user will be dropped and there's no way for the user to know that this has happened. This means that the UI in Max will indicate one value but the actual value used by the remote `Player` is another.

Now, one could claim that it's perfectly legitimate to assume that the user would understand that it's not possible to set parameters until the initialization of the remote `Player` is complete, and while this is true to some extent, it would make the objects cumbersome to use in Max, as it wouldn't be possible to rely on normal initialization / control flow mechanics in Max such as attributes, loadbangs or pattr, and the it would be important to manually keep track of the order that the messages are sent in, thus departing far from the standards of Max programming.

With this in mind, the objects that rely on communication with the remote Python server (i.e. the `somax.server` and `somax.player` objects in Max) have been designed to store its internal state inside the object as well as on the remote. This means that if the remote crashes, is restarted, or if parameters are set before the remote has been started, it's possible to just update the remote by sending the state as it has been stored in Max. This means that the objects can be used in a normal manner, where we can set any attribute at initialization independently of whether the remote is running, and then just output it after the initialization procedure has been completed.

3.1.4 Abstraction Issues

The second main issue of the Somax library is related to the requirement of writing the Max code as abstractions rather than externals. While this comes with a number of benefits, mainly in terms of transparency for the user, the major problem with abstractions in Max is that they behave differently than externals on initialization. Consider the example in figure 3.1. Assuming that the `myabstraction` object is an abstraction (which relies on `patcherargs` to pass attributes) and the `myexternal` is an external (where the attribute is a member variable), the value of the attribute `somevalue` will be different in these two objects. In the external object, `somevalue` will be set to 1 when the object is initialized, then immediately changed to 2 by the `loadbang`, as we expect. In the abstraction, the object will be initialized without a given value for `somevalue`, then the `loadbang` will set it to 2, followed by the deferlow'ed `patcherargs` setting it to 1. So in the object on the left, the value for `somevalue` will be 1 but in the object on the right, the value will be 2.

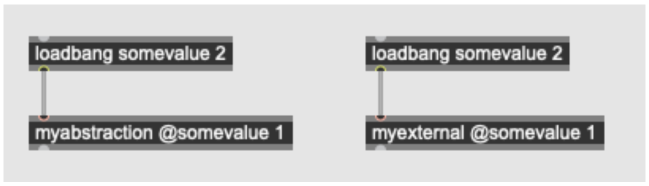

Figure 3.1: The abstraction inconsistency

Now, one could claim that this is a rather marginal case that won't be of huge importance, as you wouldn't set the attribute to one value and then immediately change it afterwards regardless. This is true for top-level objects, but becomes problematic for any nested abstraction. Consider the case in figure 3.2. Here, `myplayer.app` is an abstraction that contains `myplayer` (among many other objects, omitted for space), where we want the default value for the attribute `mode`, when used in this particular abstraction, to be `continuous`, but the user may want to change this value immediately as part of their own configuration. If `myplayer` was an external, this would work as intended, but since `myplayer` is an external, the `patcherargs` will overwrite the user settings as sent by the `loadbang`.

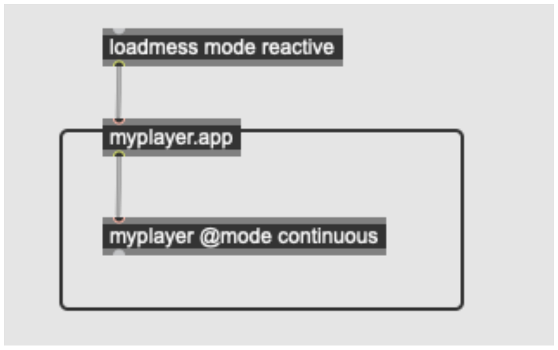

Figure 3.2: Nested abstractions

Unfortunately, there are a number of issues related to nested abstractions, such as the difficulty to pass any number of attributes from one abstraction to another (requires nested `patcherargs` where we cannot ensure the order), the difficulty to set default arguments (passing arguments from one abstraction to another with the `#1`-syntax will be replaced with `0` if the user doesn't provide any argument to the outer abstraction). Yet another issue is the fact that there's no way to fail initialization of the abstraction (as in figure 3.3) if invalid arguments are passed to the abstraction. This can be very problematic if the abstraction is manipulating some global state or performing I/O-operations. A concrete example in the Somax library would be if `somax.player` is initialized with invalid OSC ports, which unless properly handled would create the `OscAgent` on the server but then be unable to communicate with it (including deleting it). Or even worse, if its initialized with an OSC port used by another `somax.player`, which would create all sorts of issues between the two players.

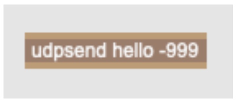

Figure 3.3: An external that failed initialization since the arguments were invalid

The workaround for most of these problems lies in the `somax.await_patcherargs` abstraction. This object is modelled on the `bach.args` object from the Bach library [12] and will store any messages sent to the abstraction and not output them until the `patcherargs` initialization is done, hence ensuring correct initialization order. Unlike the `bach.args` object, this object will also allow explicit initialization failures if the object is initialized with invalid arguments. In this case, the `somax.await_patcherargs` will continue to block any input to ensure that no I/O-operations or global state changes are done while the object is invalid. If the issues causing the initialization to fail are resolved, it's possible to reinitialize the object with the `reload` command.

3.1.5 Global State

While one of the main guidelines presented in section 3.1.2 was that global state should be avoided as much as possible, some interaction between certain objects is necessary. This could've be solved by directly connecting the objects to each other (which, in fact, was the case in early versions of Somax 2), but from a user perspective, this would've been inconvenient and difficult to understand, as it is related to internal implementation details that the user shouldn't have to be aware of. Global state is therefore used in the following cases:

Initialization of server and players

The `somax.server` and `somax.player` (or more specifically the `somax.unique_number` abstraction inside the `somax.player.core` object) objects store their corresponding OSC ports in the global `somax.recvports`. This allows us to automatically assign OSC ports when new `somax.players` are created, as well as ensure that if an object is created with manual ports that are already used elsewhere, the initialization will fail. Ports are removed from the dictionary when the `somax.player` or `somax.server` are deallocated.

The `somax.player` also stores its name in the `somax.players` dict (through the `somax.uniuqname` object) to ensure that two players with the same name cannot be created on the server.

Finally, the `somax.server` stores an internal counter of the number of `somax.server` objects that currently exists in Max, to avoid duplicate servers. This is stored in the `somax.num_servers` `value` object.

Communication between server and player

Nearly all communication between the remote `SomaxServer` and `OscAgent` (which correspond to the `somax.server` and `somax.player` objects in Max) is handled directly on the Python side, but there are a couple of cases where communication directly between the `somax.server` and `somax.player` local objects is necessary. These are:

- When the remote `OscAgent` is created. This message is sent from the `somax.player` to the `somax.server`, which sends the message to the `SomaxServer` to create the `OscAgent`.
- When the `somax.server` crashes or is terminated in a manner that doesn't allow the remote `OscAgent` to inform its local `somax.player` to stop playing.
- When the `somax.server`'s transport is started or stopped.

These messages are handled with global `send-receive` pairs on the addresses `somax.from_server` for messages from `somax.server` to `somax.player`) and `somax.to_server` for messages from `somax.player` to `somax.server`.

Communication between server and corpus builder

The `somax.midicorpusbuilder` and `somax.audiocorpusbuilder` objects do not have their own pairs of `udpsend` and `udpreceive`, but communicates through the `somax.server`. Messages from either of the local corpus builders and their remote counterparts are therefore sent through the `somax.server`, and uses the same send/receive addresses as in the above paragraph (`somax.from_server` and `somax.to_server`).

Corpus path

The `corpuspath` is the folder to which all corpus builders will export their corpora when built, as well as the folder where all `somax.players` by default will search for corpora and associated audio files. The corpus path is fully managed by the `somax.corpuspath` abstraction, which internally uses the `value` object with name `somax.corpuspath` to store the corpus path, as well as `send-receive` address `somax.corpuspath_update`, which triggers an update in all `somax.corpuspath` objects when the path is updated in one of them.

Communication between .app objects

Since some of the explicit requirements for the `somax.<object>.app` objects (described in section 3.4.4) are that they should be self contained, support fully dynamic routing of influences between objects and be duplicatable without having to leave presentation mode (i.e. without having to add or remove any patch cords), global sends and receives is a must here.

Influence messages between .app objects are are sent through the `somax.source` object (contained in any `.app` object that has an influencer) using an address specific for that particular source with the format `source.<name>` (where `<name>` is the name of the object, e.g. `Player2` or `AudioInfluencer`). These names are also stored in the dictionary `somax.sources`.

Finally, the `somax.routing` send/receive-address is used whenever a source is added or removed to update any related UI object, for example to remove the source from all associated `umenus` if the source is deleted.

3.2 The Server Object

The `somax.server` object manages all communication with the `SomaxServer` class in Python, and is responsible for launching the Python application/script as well as managing its lifetime. From a development point of view, the `somax.server` object has five different phases: (1) object initialization, (2) remote initialization, (3) runtime, (4) remote termination, (5) object deletion. Note that this is different from what the status outlet (second outlet of the `somax.server` object) shows: this outlet shows the different states as a result from the different phases. Figure 3.4 illustrates the relation between the phases and the states of the object. The state is continuously polled by the `somax.serverstatus` abstraction, which continuously sends requests over OSC to check whether the remote server is responding and update its status.

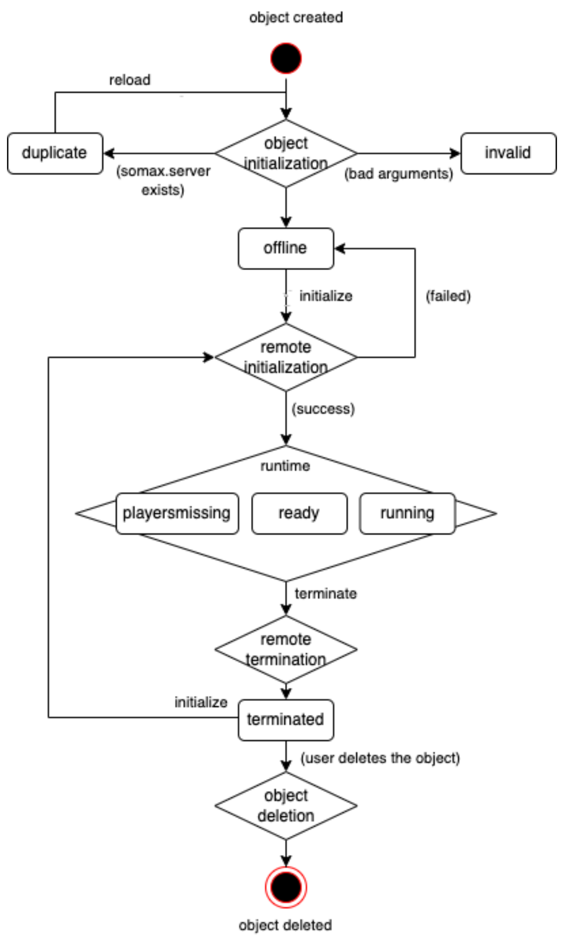

Figure 3.4: State transition diagram for the `somax.server` object. The rounded rectangular objects represent the states as shown on the second outlet of the object and arrows indicate name of messages sent to the `somax.server` object (parenthesized text represent cases, not messages). Note that the diamond blocks do not necessarily represent conditions (for example the "remote termination" phase cannot meaningfully fail) they are only used to illustrate the different phases/transitions described in this text.

Object Initialization

When the `somax.server` object is created, the strategy described in section 3.1.4 is used to handle `patcherargs` accordingly. There are three possible outcomes here:

- The initialization fails because invalid arguments are provided (for example a port that is already in use[1]). The state will be `invalid` and the object will ignore all messages.
- The initialization fails because another `somax.server` already exists. The state will be `duplicate` and the object will ignore all messages except the `reload` message.
- The initialization succeeds. The state will be `offline` and accept any messages.

The reason why it's important for this step to be able to fail is because `somax.player` sends their `create_agent` messages to the `somax.to_server` address. If we have two initialized `somax.server` objects, this means that we will receive it twice and therefore create two duplicate remote agents, causing all sorts of issues. The same issue applies to all messages in general: even if we send a message directly on the inlet of one of two duplicate `somax.server` objects, we will only send a single message through the `udpsend`, but any response from the remote server will be received on both of the `udpreceive`, and hence risk causing a number of issues. The duplication status is handled through the internal `somax.num_servers value` object mentioned in section 3.1.5, which is incremented when the object is created, independently of whether initialization succeeded or not. The ports used by the object are also stored in the `somax.recvports` dictionary.

Sending `reload` to a `somax.server` object will trigger the procedure described under "Object Deletion" immediately followed by the "Object Initialization"

All `loadbang`/`loadmess` messages are handled when the object is created (technically before the `patcherargs`, but this will only set the default values for a number of parameters and settings, it will not trigger any output, and the order that the `loadbang`s are executed in is (and should always be) irrelevant.

Remote Initialization

When the `initialize` message is sent to the `somax.server` (internally or externally – the `autoinitialize` attribute will trigger the `initialize` message internally once object initialization is complete), the remote server script/application will be launched through the `somax.interpreter`, specifically by using the `;max launchbrowser` command, which opens the provided file in the default application. Once the server has been properly launched, the `SomaxServer` will immediately send the message `initialized` over OSC, which will be output on the first outlet of the `somax.server` object, as well as send its currently stored value for all relevant attributes (currently only `tempo` and `active`).

[1]Note that while it's possible to initialize the server with custom ports and/or ip, it will not be possible to launch the server through the user interface in this case. To do this, the server must be manually launched from a terminal.

Runtime

The runtime phase, from when the connection between the local somax.server and the remote SomaxServer has been established (represented by the states playersmissing, ready and running) is where the user likely spends 99% of the time, but is from a development point of view the phase that requires the least amount of work. During this phase, any message sent to the somax.server object will simply be sent directly to the server (but stateful messages, such as tempo and active will also stored locally, in case the server is restarted).

During this phase, the somax.serverstatus will continuously poll the remote server, by sending a list of the names of all the local somax.player objects that exist in Max. The remote server will respond with two values, one boolean indicating whether all of these objects also exist on the remote (if not, the status will be playersmissing) and whether the server's transport is active (status: running or not (status: ready). Typically, the remote server should always be able to respond to these requests within a second, as all of its heavy operations are computed in separate threads, but in some cases (for example when running "Test Segmentation" from the somax.audiocorpusbuilder), the server will change its status to working, indicating that it didn't receive any response. If the server doesn't receive any response for 25 seconds, the somax.serverstatus will assume that the server has crashed and change the status to terminated (not represented in figure 3.4).

Remote Termination

When an explicit terminate message is sent by the user, the remote server will be shut down and trigger the terminated message before closing. It will before shutting down delete all of its remote OscAgents and thereby trigger terminated messages to all corresponding somax.players. The somax.serverstatus will receive the terminated message and change its state to terminated. Note that this state is internally identical to offline.

Object Deletion

When the somax.server object deleted, the freebang is triggered, which decrements the somax.num_servers value is decremented and its ports are deallocated from the somax.recvports dictionary. If the autoterminate attribute is enabled (which is the default case), the remote server will also be shut down as if terminate was explicitly called[2].

3.3 The Player Object

The somax.player corresponds to a single OscAgent instance on the remote. Since this object corresponds to a remote object, it will have the same five phases as the

[2]Note that this is not represented in figure 3.4, but the object can obviously be deleted from any of the states above, so a direct transition from any state to the deletion phase is possible and non-problematic)

somax.server (object initialization, remote initialization, runtime, remote termination and object deletion), but only the first two are complicated in terms of implementations. From a user point of view, we're only concerned with one thing: is the object initialized on the remote server or not. This is why the status outlet (second outlet) of the somax.player is much more simple than that of the somax.server.

The somax.player is divided into two abstractions, the somax.player and the somax.player.core. The somax.player.core corresponds to a generic class that is mandatory for any player, while the somax.player object is the implementation of a specific player consisting of four Atoms, a specific set of ScaleActions and loads of exposed parameters. It would however be possible to use the somax.player.core to implement a different configuration of a player, for example an OMax-like player with only a single, self-influenced Atom, or a highly complex player with 8 different layers corresponding to different musical dimensions. In terms of roles, the somax.player.core is responsible for the initialization and termination of the remote Player instance (the former is managed through local sends/receives to the somax.server) as well as OSC-communication with the remote OscAgent once it has been created, while the somax.player is responsible for the initialization of the Players subcomponents (Atom, ScaleAction, PeakSelector, etc.) as well as storing and managing the state of all of the remote parameters.

Object Initialization

When a somax.player is created, there are two main steps: initialization of the somax.player.core and initialization of the somax.player. The initialization of the somax.player.core is similar to the procedure described for the somax.server. The object takes four arguments: the name of the player, OSC send port, OSC receive port and ip address of the remote player.

If no arguments are provided, it will automatically be assigned a unique name (using #0-syntax) and the first two available ports that aren't in use by any other Somax object (through the somax.recvports dictionary). In this case, initialization of the somax.player.core cannot fail. If the user provides a name manually, this name will be changed to <name>_#0 if another somax.player with the same name already exists. If the user provides manual ports and those ports are already in use by another somax.player or somax.server, the object initialization will fail (through somax.await_patcherargs, as explained in section 3.1.4). If the initialization succeeds, the name and ports will be stored in their corresponding so that no other player may use them while the object exists.

The somax.player is responsible for all remote parameters, which are handled as attributes in Max. When created, all attributes are set to their default values through a loadbang/loadmess, but does not trigger any output[3]. After that, the patcherargs is used to parse any attributes provided by the user. Note that initialization of the somax.player cannot fail, only its contained somax.player.core may fail.

[3]Note that all parameters are accessible through pattr, so while the message set $1 typically doesn't trigger any output from the object itself, the internal pattr will still be updated, and this triggers output on the third outlet through the pattstorage object

Remote Initialization

When the `initialize` message is sent to the `somax.player` (internally or externally – the `autoinitialize` attribute will trigger the `initialize` message internally once object initialization is complete), the remote initialization will begin. This process is a bit more complicated than the `somax.server`'s initialization, as the remote `Player` contains a number of subcomponents (`Atoms`, `ScaleActions`, etc.), which in turn have a number of parameters, and we can obviously not set the value of a particular component's parameters before the component has been created. The remote initialization procedure is as follows:

(i) the `somax.player.core` will send the `create_agent` message to any existing `somax.server` through the `somax.to_server` address, which will send it to the remote server over OSC (this step will obviously fail if no `somax.server` exists or if the remote `SomaxServer` isn't initialized), creating and starting the `OscAgent`, which sends the message `initialized` over OSC immediately after it has been started. This message is received by the `somax.player.core` but consumed by the `somax.player` (in other words, it will not be output through the outlet of the `somax.player`) to trigger (iii)

(ii) When the `somax.player` receives the `initialized` message, it will send OSC messages to the remote `OscAgent` to instantiate all of its `Atoms`. For each `Atom` created, the remote will respond with a list of the names of all created `Atoms`. When this list has the length of four (i.e. when the last of the four `Atoms` used in the default configuration is instantiated) it will trigger step (iii)

(iii) Once all `Atoms` are instantiated, the `somax.player` will send the values of all of its stored parameters to the remote `OscAgent` in no specific order. If two parameters depend on each other, the order of initialization is handled locally in the `somax.player` (which is the case for some experimental parameters, for example taboo, where parameter `tabooenable` also triggers output of `tabooduration`). Once all parameter messages have been sent, the `somax.player` will output the `initialized` message and finally load any corpus, if the user has provided one before instantiation.

Note that the goal of this instantiation procedure is to achieve what was described in section 3.1.3. That is, to allow the user to set the parameters through normal means (attributes, loadbangs, etc.) at any time, without having to wait for the server/player to be online before doing this. It will ensure that the state is preserved if the player is terminated and reinitialized. In effect, we're allowing the user to build their own UI around the `somax.player` object, where the user always can be sure that the state of their UI will, as long as its value has been sent to the `somax.player`, always be the actual value of the parameter on the remote.

Runtime

Similarly to section 3.2, the runtime phase, which starts when the remote `OscAgent` is fully initialized (when the `somax.player` outputs the `initialized` message) is the most important phase for the user, but simple in terms of the `somax.player` object implementation. During this phase, the `somax.player`'s main role is to convert incoming Max messages (i.e. converting flat case messages into OSC function calls on

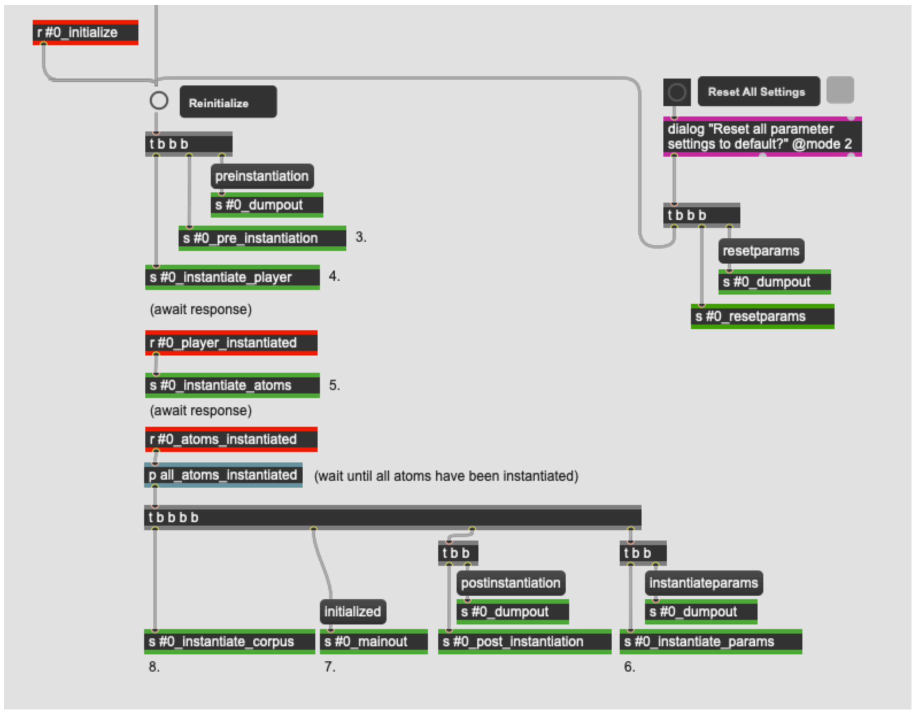

Figure 3.5: The initialization code in `somax.player`

the form `<func_name> <param1>= <value1> <param2>= <value2> ...` into corresponding Python messages.

There's a separation between `messages` on one hand and `attributes` on the other (which are clearly separated in the maxhelp of the `somax.player`). Messages are converted to the appropriate Python syntax and sent directly to the server without storing them. Attributes, on the other hand, generally revolve around some stateful object in Max (`toggle`, `number`, `value`, `slider`, etc.) and are typically set through the `pattrhub` object, which also triggers output from the targeted object and therefore updates the remote as well. There are also a couple of macro parameters (`mode`, `weights`, `transpositions`, etc.). These typically bypass pattr in order to accept and/or output messages with a better format. For example, the `matrixctrl` controlling transpositions will in Max output a 36-element list of `<row> <col> <value> <row> <col> <value>`, which is unnecessarily difficult to understand and use for a one-row fixed size `matrixctrl`, and is therefore converted into a 12-element indexmap before being output.

Remote Termination & Object Deletion

For the `somax.player`, both of these are trivial. When the remote is terminated (through an explicit `terminate` message sent by the user through the `freebang` when the object is deleted), the `somax.player.core` will simply send the `terminate` message to the remote `OscAgent`, which terminates it and deletes it from the `SomaxServer`. When the object is deleted, the `somax.player.core` will deallocate its name and ports from their corresponding dictionaries (through the `somax.uniquename` and `somax.uniquenumber` abstractions in which they are stored).

3.4 Peripherals

In the whole Somax architecture, 99% of the development work revolves around the `somax.player` and `somax.server` objects and their associated Python code. Those objects are also the most difficult to get a good overview of, as it's not possible to understand exactly what's going on by reading the Python and Max code separately. It's also difficult to step through the code with a debugger, as there's no joint debugger for both Max and Python (it's obviously possible to step through individual function calls to Python, but that won't help with the overview. It's exactly for this reason that this document is written – to provide the developer with the overview that otherwise is difficult to achieve.

Apart from the `somax.server` and `somax.player` object, there are however six more base objects of the Somax Max library (`somax.audioinfluencer`, `somax.midiinfluencer`, `somax.audiorenderer`, `somax.midirenderer`, `somax.audiocorpusbuilder` and `somax.midicorpusbuilder`, a number of UI-based utility objects (e.g. `somax.pan2`, `somax.midioutput`, `somax.audioinput`, etc.), a number of compact UI-based versions of the base objects (`somax.<object>.ui`), application-style ready-to use objects for users with little or no experience with Max programming (`somax.<object>.app`) as well as over 50 "package-private" abstractions, i.e. abstractions that are reused in several places in the Somax package but typically not relevant to the end user.

These objects are however typically small and self-contained in a more traditional sense, closely following the Max standard for abstractions, where all inputs and output are well-documented with no side effects or complex dependencies. An experienced Max developer shouldn't have to spend more than a minute or two to grasp the overview of any base, ui or app object apart from the `somax.server` or `somax.player`. Still, in the following sections, a couple of notes will be given on each of these objects for completeness.

3.4.1 Influencers

The `somax.audioinfluencer` and `somax.midiinfluencer` objects are responsible for segmenting and analyzing real-time audio / MIDI streams into discrete influences for the `Player.influence` function described in section 2.5.2. This analysis procedure is more or less identical to the procedure for segmenting and analyzing the corpus (see section 2.3), but with fewer classifiers (only onset, pitch and chroma) and slightly different analysis algorithms (`bonk` for onset, `OMax.Yin+` for pitch and `ircamdescriptor` for chroma in the audio case, and identical procedures as in the `CorpusBuilder` for MIDI).

3.4.2 Renderers

The `somax.audiorenderer` and `somax.midirenderer` converts the `RenderingMessages` that the `OscAgent` sends each time it plays a new slice into real-time audio / MIDi streams to output.

The MIDI renderer's implementation doesn't need any explanation, it's basically just a pass-through with some convenience functions to handle flushing in the case where the server crashes (or other cases of incorrect termination of a `somax.player`).

The audio renderer utilizes N parallel `groove`$\sim$ (through `mc.groove`$\sim$) objects rendering the same `buffer`$\sim$, in which the audio file corresponding to the corpus is loaded, with crossfades between them occurring each time an `audio event` message is received. The `somax.audiorenderer` will ignore the end time of the event and continue playing the buffer linearly until it receives a new `audio event` message or an `audio audio_off`, in which case it will stop playing. If the `AudioStateHandler` (see section 2.4.2) determines that an event is a direct continuation of the previous event, it will rather output a `audio continuation` message, in which case no crossfade is necessary and the renderer will simply continue playing the buffer linearly. It will however use the time stretch information even for `audio continuation` events, in case the user has changed the tempo since the last event was received.

This implementation is by no means perfect, all parallel N `groove`$\sim$ objects are playing all the time (even when they are muted) with possibly different time stretch and pitch shift factors, resulting in a high computational cost, which unfortunately is necessary since it typically takes 1-10 ms for a `groove`$\sim$ to start playing once stopped. Also, if N or more events are received within the threshold of the `release` parameter (default value: 150 ms), this is at risk of causing clipping issues in the audio. A typical renderer for granular or concatenative synthesis (e.g. `mubu.concat`$\sim$) would solve this issue, but currently there are no such renderers supporting per-grain time stretch / pitch shift. So unless an external is developed for this particular purpose, or `mubu.concat`$\sim$ adds support for this, the current rendering strategy is likely the best one available.

3.4.3 Corpus Builders

The `somax.audiocorpusbuilder` and `somax.midicorpusbuilder` are UI objects designed for building the corpus through the `somax.server`. They are essentially abstractions with a single purpose: to format and send the `build_corpus` command to the `SomaxServer`. The corpus builders do not have their own OSC addresses, communication is done internally through the `somax.to_server` (on which it sends the `build_corpus` command from the `somax.server`) and `somax.from_server` (on which it receives status updates while building as well as the path to the built corpus when complete).

3.4.4 UI & App Objects

As mentioned in section 3.1.1, there are two additional variations on each of the base objects: `somax.<object>.ui` and `somax.<object>.app`. The `.ui` objects are just convenient wrappers around the base objects that can be used as bpatchers with a condensed UI.

The `.app` objects encapsulate these `somax.<object>.ui` objects along with abstractions for audio / MIDI input and/or output when relevant (e.g. the `somax.audioinfluencer.app` has both an audio input abstraction, a `somax.audioinfluencer.app` and an audio output and an audio output abstraction). The `.app` objects also manage wireless routing of influences from influencers and any `somax.player.app` (which has influencers) to any other `somax.player.app` through the `somax.player.routing` abstraction. Internally, this is handled through the `source.<name>` addresses described in section 3.1.5.

Chapter 4

Real-time Recording

Recording a corpus in real-time is a special case that is difficult to categorize as either a part of the Max architecture or the Python architecture, as it so heavily relies on both. Unlike most behaviours described in chapters 2 and 3, real-time recording requires constant back and forth communication, where the Max architecture records audio into a buffer, sends information to the remote player, which validates it and stores a new corpus event, and returns this information before the next event can be recorded.

In-depth description of real-time audio recording

4.1 Latency Correction

As described in the previous sections , there are two main mechanisms involved when recording audio using the `somax.audiorecord` object: the underlying `record` object, which records the incoming audio into a `buffer`, and the `somax.audioinfluencer`, which provides the information about segmentation and annotations of the content in the buffer.

not described yet

The problem is that these two are not perfectly aligned. For pitch-based segmentation (`OMax.Yin+`), the latency $\gamma \in \mathbb{R}$ (in ms) between the true onset of a signal and the analyzed onset is expected to be $2T + w$, where $T \in \mathbb{R}$ is the period of the minimum frequency (as specified by the user) and $w \in \mathbb{R}$ is the length of the window used by the algorithm (again, as specified by the user). For onset-based segmentation (`bonk`), the latency is a non-zero constant.

Consider the example in figure 4.1. The upper image illustrates an audio signal with four onsets recorded into a buffer (the box corresponds to the whole buffer, but data has only been recorded in the first half), with red lines drawn where the onsets occur. The blue line indicates the end of the currently recorded segment. The lower image illustrates the onsets as they were recorded. There are a number of things to note here.

First of all, there cannot be any gaps in the buffer that don't correspond to a event. For this reason, the onset of event $\mathcal{E}_{i+1}$ should always correspond to the sum

"As should have been described earlier"

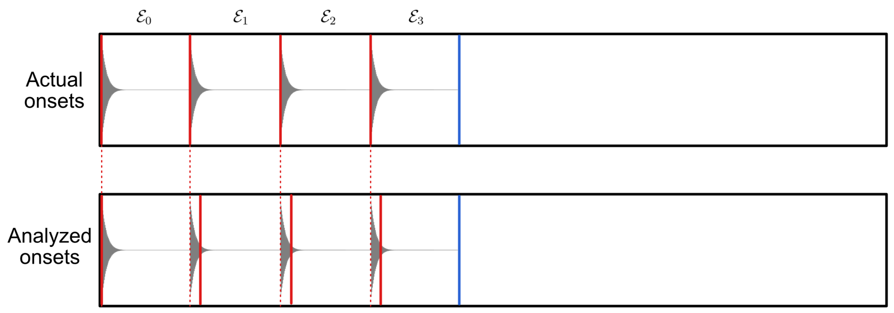

Figure 4.1: latency of analysis in relation to the audio buffer

of the onset and duration of event $\mathcal{E}_i$, or more formally

$$o_{i+i} = o_i + d_i$$

where $o_i \in \mathbb{R}$ denotes the onset time of event $\mathcal{E}_i$, and $d_i \in \mathbb{R}$ denotes the duration of said event. In practice, this means that event $\mathcal{E}_i$ is not recorded until onset o_{i+1} is registered.

This should already have been explained

Secondly, the first onset in each recorded segment will always be perfectly aligned with the buffer. This follows from the first statement; as there cannot be any gaps in the buffer, the first detected onset (prior to latency correction, see below) will be considered as the end of the first recorded event.

Thirdly, the final segment will only be recorded if the `appendend` option is enabled. If this option is enabled, the last event will be recorded with a duration corresponding to the exact position of the end (blue line in figure 4.1. If this option is disabled, the last recorded segment will in the image above be $\mathcal{E}_2$.

This should also have been described already

Appendix A

Class Diagram

Here, a class diagram for the entire Python architecture is presented. A high resolution image is also available in the GitHub repository[1]. Note that final classes that extend a particular abstract class are not represented in this diagram (e.g. the `NextStateScaleAction` or any of the other 12 scale actions that implement the `AbstractScaleAction` interface), as this would clutter the diagram and significantly reduce the readability. Non-final classes that extend an abstract class are still present as they are important from an architectural perspective (e.g. `RendererEvent` which extends the abstract `ScheduledEvent`).

Finally, certain aspects of the architecture have been omitted as they currently do not serve any meaningful architectural purpose but are simply present for legacy reasons (e.g. the distinction between the classes `Agent` and `OscAgent`, where the former has been omitted since it no longer serves any purpose), or stateless/static classes that serve little purpose from an architectural point of view (e.g. enum classes).

[1] `https://github.com/repmus-docs/SomaxDocs/tree/master/UML`

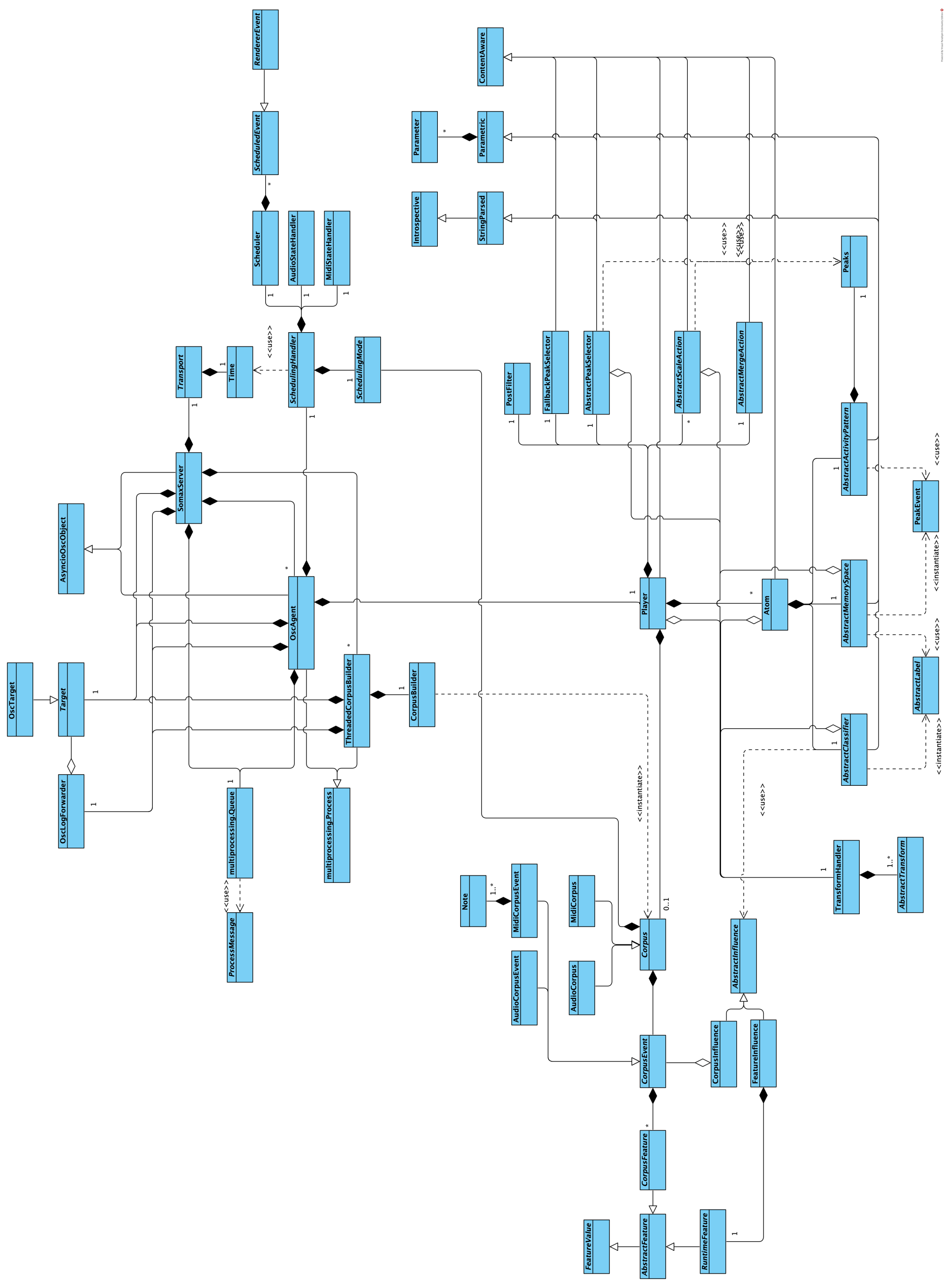

Figure A.1: Class Diagram for the Python architecture. Note: a high resolution image is available in the GitHub repository.

Bibliography

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

[2] Dicy2-python. `https://github.com/DYCI2/Dicy2-python`. Accessed: 2023-03-16.

[3] Generative interactive agents library (GIG). `https://github.com/DYCI2/gig`. Accessed: 2023-03-16.

[4] Maxosc/pyosc. `https://github.com/jobor019/pyosc`. Accessed: 2023-03-16.

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

[6] Open Sound Control. `https://opensoundcontrol.stanford.edu/`. Accessed: 2022-02-01.

[7] PyInstaller. `https://pyinstaller.org/en/stable/`. Accessed: 2022-03-20.

[8] Somax2. `https://github.com/DYCI2/Somax2`. Accessed: 2023-03-13.

[9] The Somax Software Architecture Rev. 0.2.0. Ircam-STMS, Technical Report, 2021.

[10] The Somax Theoretical Model Rev. 0.2.0. Ircam-STMS, Technical Report, 2021.

[11] Mid-term Update on the Somax System. Ircam-STMS, Technical Report, 2022.

[12] Andrea Agostini and Daniele Ghisi. Bach: An environment for computer-aided composition in max. In ICMC, 2012.

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

[14] Jérôme Nika. Guiding human-computer music improvisation: introducing authoring and control with temporal scenarios. PhD thesis, Paris 6, 2016.

[15] Jérôme Nika, Augustin Muller, Joakim Borg, Gérard Assayag, and Matthew Ostrowski. Dicy2 for Max. Ircam UMR STMS 9912, 2022.