Modality

Marije Baalman
STEIM, nescivi
Amsterdam
marije@nescivi.nl

Till Bovermann
Berlin University of
the Arts, Berlin
t.bovermann@udk-berlin.de

Alberto de Campo
Berlin University of
the Arts, Berlin
decampo@udk-berlin.de

Miguel Negrão
Sonic Arts Research
Centre, Belfast
miguel.negrao@friendlyvirus.org

ABSTRACT

The Modality Project explores the idea of highly modal performance instruments i.e., setups where a small set of controllers can be used to play a wide variety of sound processes by changing control constellations on the fly.

The Modality Toolkit is a SuperCollider library which simplifies the creation of such instruments. To this end, a common code interface, MKtl, is used to connect controllers from various sources and protocols. Currently, HID and MIDI are supported; GUI-based interfaces can be created on the fly from interface descriptions. Detailed use cases demonstrate the concepts of working with modality practically in code.

This paper gives an overview on the concept of modality as seen by a group of sound artists and researchers, and describes one interdisciplinary approach to creating a toolkit written for and by electronic musicians.

1. INTRODUCTION

The *Modality* project [1] was initiated by Jeff Carey and Bjørnar Habbestad, who, after several years of collaboration, realised that they, despite playing completely different setups, both had the need to easily switch between functionalities within performance. While both had custom implementations of this behaviour, it appeared to be not flexible enough. Especially extending their setup felt cumbersome, and original ideas got lost over the hassle of implementation of mapping rules.

Out of these observations arose the idea to gather a group of experts in sound and music computing (and specifically from the SuperCollider community) which eventually formed the ModalityTeam. Starting with five people at the first meeting, more people became involved. The group currently consists of 12 members.

The intention of this paper is two-fold: Firstly, after an introduction on the concept of modality and related work (Section 2), it gives insight on work in an interdisciplinary team of loose collaboration. It is driven mostly by a shared interest in, on the one hand, sound, music, and performance practice, and on the other hand software design and development (Section 3). Secondly, it reports on the outcomes in both conceptual and concrete implementation details (Section 5 to 6). The paper concludes with a reflection on the work done over the course of the last 5 years.

[1] http://modality.bek.no

Figure 1. Impressions of the Modality meeting (left), workshop (center), and concert (right) 2014 at STEIM, Amsterdam.

2. THE MODALITY WAY

The Modality project is dedicated to modal interaction with synthesis processes for physical control in performance and musical practice. The name *Modality* arose from the idea to investigate the creation and extensive use of modal interfaces. One particular strength of such modal interfaces is that they allow fast changes and therefore a broader variety for sonic discovery. This can be of benefit when, for example, improvising with musicians playing acoustic instruments. Out of this arouse the question on how HCI interfaces can be conceptualised and with a small set of physical controls assigned to a relatively large function set. We contend that integration of such on-the-fly remapping features helps to create flexible instruments that are powerful yet interesting and therefore rewarding to play and listen to.

The primary product of the Modality project is the *Modality Toolkit*, a software library that facilitates (a) access to hardware and software controllers, (b) flexible routing of control messages to generative processes, and (c) recording, filtering and further processing of controller signals. The ModalityTeam, an international and transdisciplinary group of people that see themselves as users and developers for SC meets at regular intervals to work on the library, discuss issues around music making, and perform in self-organised concerts.

Modality, however, can also be understood as a social descriptor for the ModalityTeam. The fact that a number of programmers and artists from different (music)cultures and nationalitirecoges meet up on a more or less regular basis does not necessarily imply that they share the same understanding, let alone opinion. It turned out that themes as fundamental to the Modality project as "performance practice", "control strategies" and even "software paradigms" were highly ambiguous and interpreted in different ways. Further, it turned out to be a learning process to not only listen to other people's opinions but to also take them into account during software design and implementation.

As a third interpretation level, the term Modality influences the structure of the meetings. Reflecting the divergence between participants, most of the meetings consisted of a broad spectrum of activities, namely (a) developer phases in which the Modality Toolkit was implemented, (b) public workshops disseminating knowledge about the Modality Toolkit, and (c) concerts in which participants performed with their custom instruments.

2.1 Related work

The Modality Toolkit stands in the tradition of a line of related systems, dedicated to control data flow and filtering. Particularly, it is informed by systems like *OSCulator* [1], STEIM's *junXion* [2], the *Digital Orchestra Kit* [3] and SC's own *multiton pattern* implementations.

OSCulator Osculator is an OS X GUI based software aimed at connecting devices and routing messages between them. It supports multiple protocols such as MIDI, HID, OSC or TUIO and is capable of creating complex responses to incoming events, including scaling values, splitting events, merging events, storing values for later use, enabling or disabling actions and toggling global presets.

junXion is a "[...] data routing application that can process [hardware] 'sensors' [...] using conditional processing and remapping" [2]. It is a stand-alone program to be put in the middle between the control input layer and the synthesis layer. The roots of its development lay in the advanced sensor input and data manipulation features of pioneering live sampling software LiSa [2]. [2]

In JunXion, data flow is organised in patches with an *input-action-output*-logic. Inputs can come from as many as eight different types of data sources. The actions process that data by means of user-definable behaviours such as switching or toggling but also differentiation, or complex activity measurement and based on conditional statements incorporate other incoming data. Output can be generated and sent in various formats to listening programs.

Digital Orchestra Toolkit [4] was created as part of the *Digital Orchestra project* around "[...] a number of paradigms for the design, creation and performance of digital musical instruments in the context of a long-term interdisciplinary, collaborative environment. Issues related to mapping strategies, notation, the relationship of physical and musical gestures, robustness, responsiveness, and haptic feedback arose during the course of the project."[5]. The toolkit consists of a number of Max/MSP objects implementing data acquisition and processing for various hardware devices and protocols.

Multiton design patterns in SC SuperCollider has flexible proxy objects for tasks, patterns, sound processes, and functions, which allow replacing the proxy's object while using it. (Modality follows these, e.g. in the `MKtl(<name>)` access scheme.) Named variants of these classes, like `Tdef`, `Pdef`, `Ndef`, or `MIDIdef`, `OSCdef` follow the multiton pattern by creating named instances only, and keeping them in a global dictionary. Calling the constructor e.g., `Ndef(\a)`, returns an existing instance by that name or, if not found create it. Supplying a second argument, `Ndef(\a, { LFSaw.ar })`, replaces the proxy's current object with the new one given. This is very useful in live coding situations, where remembering name-function pairs is much easier than doing full variable administration by hand.

3. THE MODALITY MEETINGS

To illustrate the Modality way as described in Section 1, this section reports on the outcomes and discussions within the four modality meetings held so far.

October 2010, BEK, Bergen Initiated by Jeff Carey and Bjørnar Habbestad, several experts and sound artists met to discuss shared ideas about modal control in performance and rehearsal situations. The attendees soon agreed that easy access and outlining of modal control structures is of great interest for all. First sketches for uniform access were made based on the then already existing JITMIDIKtl quark [3], creating a more uniform access scheme to controllers in the Ktl quark.

May 2011, STEIM, Amsterdam Discussions revealed the need for users to abstract from hardware dependencies, and being able to do flexible routings and filtering incoming data. A new SC quark was initiated and the group started implementing two sets of functionalities:

`MKtl` objects were intended to connect MIDI and HID hardware devices. They stored capabilities of each device in a configuration file. Instead of assigning functions to hardware-specifics, we considered controllers as a combination of *controller elements*, which were given human-readable short names for semantically simple access (e.g., `'sl1'` instead of MIDI channel 0 cc 14). This scheme was considered extensible for OpenSoundControl, serial ports, and other hardware interfaces, to have a uniform workflow, abstracted away from the actual backend.

[2] As of today, LiSa's sampling engine is not being further developed, as many software synthesizer are available to replace its functionality. Similarly, STEIM's groundbreaking sensor and interfacing technologies have become readily available through a host of affordable controllers and DIY-kits, e.g., those based around the Arduino platform.

[3] a quark is an extension library in SuperCollider parlance.

`MDispatch` objects allowed creating *calculation units*; abstract filters that render output from a given input, like the conversion of a button press (*on*, *off*) to one trigger event (*now*), or the calculation of slider speed. Many templates for commonly used functionality were created.

A stumbling block at the time was an OS-dependant (and, due to changes in the API of Apple's HID toolkit, on the OSX side non-functional) HID implementation in SC.[4] The project's state was presented at the SC.symposium 2012 in London, UK [6].

November 2013, BEK, Bergen After a hiatus of almost 2 years, this meeting focused on practical steps. It particularly took a while to get back to a productive working environment. The introduction of an issue tracker to define and discuss development goals (in combination with git as a repository for code development) helped to get on track again.

In this light, example use cases of different levels of complexity were noted down in order to define the demands (and limits) of the Modality toolkit (see Section 4). It also turned out to help understand, how complex the user-written code to implement the use case would be and therefore get insights on how the toolkit has to be adjusted to facilitate this.

Further, unified functionality to the input layers were added: *Explorer* classes for MIDI and HID listen to incoming messages from a source and generate initial data to help writing description files. Hierarchical ordering of elements within these files was introduced to allow the representation of semantical grouping.

Proposals for related useful concepts, such as FRP and Influx, were explored (see Section 6.5).

April 2014, STEIM, Amsterdam Many aspects of the input side were unified and simplified further, thus nearing completion of the input layer.

Description file handling was improved in many ways, and GUIs could be initiated for missing devices. Mapping strategies were simplified toward a unified style with e.g., `SoftSet` and `RelSet` (see Section 6.3).

While in the meetings before, writing documentation was mostly postponed until it was too late, in this meeting, documentation and examples were written in dedicated sessions, and use cases were sketched in text and implemented in various coding style variants.

Finally, the OS unification of the HID interface implementation was fixed, pending full tests.

4. EXAMPLES / USE CASES

We created a number of simple to medium-complex uses cases, which serve both as examples for modality concepts, and as test cases that show how simply they can be implemented in different coding styles.

4.1 Switching operation mode

This example illustrates a situation where there are multiple global modes of operation. Depending on which mode the system is in the physical controls perform entirely different actions in the system. This is similar to how computer keyboards perform different actions depending on which modifier keys (shift, ctrl, alt) are pressed.

Consider 16 buttons in a 4×4 grid. The first 3 rows contain *memory* buttons, in the last row the first 3 are *play* buttons and the last one is the *shift* button. Sound sources can be copied from the play buttons to the memory buttons.

Play slots Each play button is assigned one fixed adsr enveloped sound sources with a single parameter. Depressing a play button turns the sound on, releasing it causes the sound to decay.

Memory slots Each memory button can be assigned from one to all three sound sources associated with the play buttons. Depressing the button activates all assigned sound sources simultaneously.

Slider When a sound source is active, the slider controls one of the synthesis parameters. There is a pickup mechanism in place such that the slider only causes the parameter to change once it is close enough to current value to avoid jumps.

Shift Button Pressing the shift button causes the system to go into *copy* mode. When in copy mode, up to three of the play buttons can be pressed followed by one of the memory buttons. This will copy the sources of the selected play slots together with their current values for the parameter into the selected memory slot. If the shift key is released mid-way no assignment takes place. Copying into an already assigned memory slot replaces the existing sources and parameter values.

The number of different operation modes could be easily extended by having multiple modifier buttons with different combinations of them setting the system to different modes of operation.

4.2 Exchanging actions

In this example a certain number of control elements are assigned to an equal number of synth parameters. Upon pressing a button the system enters *remap* mode: it waits for movement from two different elements and then switches the parameters that they control amongst themselves. This simple example illustrates the need that often arises in a performance of freeing a finger or hand, by moving the action that it is controlling at that moment to another physical control (or just disconnecting it momentarily), so that the hand or finger is now free to control some other parameter which at that point in the performance has become more important.

[4] As of April 2014, this has been solved in SuperCollider 3.7 with a new cross-platform HID implementation.

5. ISLANDS, BRIDGES, UNIFORM SCHEMES

As many home towns of modality members are harbour cities, Islands and Bridges were chosen as a mental model for conceiving and understanding highly modal instruments. Islands are software objects or processes which represent sources (input devices, control-generating processes), destinations (output processes for sound, visuals), combinations of these, such as transformers (which, like destinations, process incoming control information, and send the results on like sources). Islands should be as self-sufficient as possible, and show uniform behavior to allow simple on-the-fly changes of connections with bridges.

Bridges typically are made by user code that connects islands; conventional digital instruments then contain a fixed collection of islands and one constellation of bridges between them. Modal performance instruments achieve their modes by switching between different combinations of bridges, adding some, removing others, adjusting settings. Modality aims to make writing and configuring bridges as simple as possible.

In other words, Modality shifts the instrument metaphor from linear chains of command to flexible networks of communication (or further on, of mutual influence).

The uniform communication schemes recommended by Modality are largely based on existing conventions in SC, and extend them with only few new methods. Thus many SC quarks dealing with interface devices or data processing are useful sources for more islands. Beside the modality-toolkit, the team actively works on other modality-relevant quarks. These are e.g., *SenseWorld* supporting sensor devices, *Manta* accessing an OSC controller, *FPLib* containing FRP (see Section 6.5.2), *VariousMixedThings* containing Influx (see Section 6.5.3), *UnitLib* [7], *wslib* mostly GUI-related niceties, *KeyPlayer* contains KtlLoop, and *DMX* output to light systems.

5.1 The uniform input device scheme (MIDI, HID, OSC, GUI, Serial)

Input devices (such as the `MKtl` class) or other control sources follow a scheme: They have rich descriptions, with simple short human-readable element names, which are hierarchically ordered where applicable. One can access each element by name or hierarchical indexes. Each element can either have a single action, or one can add and remove multiple actions individually by identity or name.

Every `MKtl` can be substituted by a Graphical User Interface derived from the corresponding description file. When operated, it acts identically to the physical device.

Explorers simplify adding new devices or sources: One activates every possible controller action at least once to collect specimens of every possible message type. Then an Explorer can make a description file template from this, and the user adds the final touches by giving them simple, short and clear element names, and organising their hierarchical order. This is implemented fully for MIDI and HID, with other protocols to follow.

Transformer islands expect control input from sources, and know how to create control output for destinations. E.g. an `Influx` is a transformer which accepts m bipolar parameters from a source, and converts them to n process parameters for a destination with a matrix of weights.

5.2 Proposed uniform destination schema

Modality-compatible sources have containers for configurable actions for sending messages to destinations, and they know how to convert their controller ranges to be unipolar (interval $[0, 1]$).

Modality-compatible destinations also follow existing SC schemes: They respond to set messages for control values; they remember current parameter values, and they often have specifications for control parameters (i.e., range, warp, step size, etc). Most objects that are active processes respond to .play, .stop, .pause and resume messages.

Requiring destinations to know their parameters specs and current states allows more flexible control in several ways: The `setUni` method can be used to set a param from the controller side's unipolar value; keeping the `Spec` with the destination process is semantically simpler to argue for, and multiple control sources will immediately use changed specs if they belong to the object. The `RelSet` class method can be used to nudge a parameter relative to its current value. `SoftSet` class methods can be used to take over a parameter only when the physical controller is close enough to its value, or when the physical controller knows the object's previous value well enough (which is the case when it has set it to that value). If specs are kept with the object, they can easily be adjusted there (e.g., for zooming into a subset of the full range), keeping the controller element side code simpler by sending unipolar values, and letting the object provide the spec:

```
{ arg el; dest.setUni(\amp, el.value) }
```

Finally, conforming to the SC convention of `play/stop`, `pause/resume` allows very simple de/activation when switching newly to or away from a process.

6. SPECIFICATION, DESIGN AND IMPLEMENTATION OF THE MODALITY TOOLKIT

The following specification of the Modality toolkit conforms to the island/bridges/unification scheme introduced in Section 5 and respects the use case described in Section 4.

6.1 Specification

The Modality toolkit [8] aims to facilitate

- data acquisition from commercially available controllers (e.g, HID and MIDI) by providing a common software interface,
- processing of control data streams,
- sending control data to these controllers (e.g., fader positions, LED states),
- graphical feedback of the current state in the form of a GUI of connected to the device, as well as replacing a controller with a GUI substitute, and finally

- mapping the output of these data streams to input parameters of sound engines.

Specific attention is given to the concept of modal control: the ability to change the mapping on-the-fly from one control element to another, possibly located on another device or to change assigned functionality of one control element based on the state of another.

6.2 Implementation

The Modality Toolkit is implemented as a set of classes for the SuperCollider language [9]. The control elements of devices are accessed through the `MKtl` class. A control element is a part of a controller that either generates and/or accepts a one-dimensional stream of events. Each `MKtl` object consists of elements such as sliders, knobs, buttons or encoders. It is possible to assign actions to such elements that are evaluated every time the value of that element gets updated. Elements are instances of `MKtlElement` and are kept in a tree-like data structure of nested arrays and dictionaries which represent the spatial grouping of control elements in the physical controller.

The `elementDescription` variable of MKtlElement contains a dictionary with information about that element such as its type (e.g., button) and control spec for scaling incoming values. This dictionary can be used to extract multiple elements from the data structure by filtering using a conditional expression, for instance retrieving all elements of type `slider`.

Using the multiton pattern described in Section 2.1, each `MKtl` has a name, and only one `MKtl` is active with that name at any given time. MKtl's can be retrieved from a global dictionary by name, using the `MKtl('name')` syntax. The system keeps a global set of auto-generated names for all the controllers that have description files. These short names are auto-generated from the name of the device plus a number starting from zero indexing multiple identical devices (e.g., `'nnkn0'` from `'nanoKONTROL'`). If a user tries to fetch an MKtl with one of the auto-generated names and it is not yet created the system will look for the corresponding device and if it is found an MKtl is created from the description file and connected to the device by creating MIDI or HID responders. This feature means the user can initialize an MKtl for a given device using always the same single line of code.

```
k = MKtl('nnkn0');
```

Actions are added to elements by setting the MKtlElements' `action` to a function. It is also possible to add and remove multiple unnamed actions to the same element using `FunctionList` or named functions which can also be re-ordered using `FuncChain`.

```
~el = MKtl('nnkn0')
.elements[\sl][0];

//add action
~el.action = { |e|
  var freq = e.value
   .linlin(0.0,1.0,300,3000);
  x.set(\freq, freq)
};

//remove action
~el.action.action = nil
```

Elements with output capabilities can also send values back to the device, this is done using the `value_` method of `MKtlElement`:

```
MKtl('bcr20000')
.elements[\kn][0][0]
.value_(0.3)
```

6.3 Unifications of interface implementations

The Modality Toolkit works uniformly across multiple protocols. The base class MKtl provides the generic functionality and the children classes (`HIDMKtl`, `MIDIMktl`, `OSCMKtl`, etc.) implement the specific back-end for each protocol. Since the interface for using Modality is defined in `MKtl` and `MKtlElement`, which are protocol agnostic, the syntax and semantics remain uniform across all protocols. The incoming values from the device are normalized by the `MKtl` to be in the interval $[0, 1]$ and outgoing values are expected to be in the same $[0, 1]$ interval and then scaled to the range used by the specific protocol. This facilitates switching between devices that use different protocols while keeping the event logic unaltered.

MKtl also has the useful feature of automatically creating a GUI representation of a known device from its description file. If the user tries to instantiate an MKtl with an auto-generated name corresponding to a known device, but the device is not currently available, an `MKtlGui` will be automatically created instead. This makes it trivial to exchange a physical controller for a GUI representation without having to change any code at all.

6.4 Description files

In order to use a device within the Modality Toolkit context, a *device description* file is needed that characterises each control element and its semantical position in relation to other elements. It is implemented using one text file per device containing a dictionary with the fields

protocol currently, HID and MIDI are implemented. Note that only one protocol per device is allowed,

device the name of the device as provided by the operating system,

description a dictionary with a tree structure composed of nested dictionaries and arrays. The value at each leaf of the tree is an element dictionary with key-value pairs describing the element at hand. An element dictionary contains technical specifications of the element, namely identification information (e.g., for MIDI, the MIDI number and channel), the physical type of control (button, slider, etc.) and a `ControlSpec` that specifies how to convert the incoming values to the range $[0, 1]$.

As an example, the element dictionary for a button of a MIDI device would look like this:

```
\rew: (
        \midiMsgType: \cc,
        \type: \button,
        \midiChan: 0,
        \midiNum: 47,
        \spec: \midiBut,
        \mode: \push
)
```

Elements which are physically (or virtually) grouped on the device such as with pages, rows or columns are grouped together in the description file using arrays. For instance, the third button on the second row of page 4 of a Korg NanoKONTROL can be accessed with the following code, assuming zero-based numbering:

```
MKtl('nnkn0').elements[\sl][3][1][2]
```

The hierarchical grouping of elements also facilitates bulk addressing of elements by traversing the hierarchy starting at the desired node. For instance, it is easy to programatically add actions to multiple elements:

```
MKtl('nnkn0').elements[\sl]
.do{ |xs, page|
 xs.do{ |xs, row|
  xs.do{ |element, column|
   element.action =
    {[page, row, column].postln}
  }
 }
}
```

New devices can be added to the toolkit easily, all that is needed is to write the corresponding device description file. If a user tries to access a device for which there is still no description file available, the toolkit guides the user in the process of creating the description file. More specifically, a description file can be generated for HID devices using the `HIDExplorer` class, which collects information provided by the low-level HID stack. For the less self-documenting range of devices connected via MIDI, the user is asked by the `MIDIexplorer` class to operate all available physical controls. The captured data stream is then used to generate a description file. As a last step in both cases, the user supplies suitable labels and orders the elements hierarchically according to their physical placement on the device.

6.5 Modality related projects and quarks

6.5.1 MDispatch

The MDispatch class was an initial attempt at creating self-contained event logic units. `MDispatch` has similar structure as `MKtl` (both inheriting from `MAbstractKtl`). A dispatch has output elements (`MDispatchOut` class) where actions can be added similarly to `MKtlElement`. It also has inputs which are updated by registering callbacks on the elements of other `MAbstractKtl`s. When an event is received from a source the input element and its value are saved and a list of state processing functions is run sequentially. These state processing functions have access to all the internal state of the dispatch which includes the source element which caused the update, all the values of the output elements and any other state variables defined for auxiliary calculations. An MDispatch can be either created from a template containing predefined functionality or explicitly defined. The process for explicitly creating an MDispatch is to specify outputs, define state processing functions and finally connect the dispatch to sources of events. In order to facilitate this process it is possible to first connect to a source and just copy the output names of the source to the output names of dispatch, in fact mirroring the same elements. This makes it straightforward to create path-through processors which take the value from each output element of a source `MAbstractKtl` and send a processed version through an element with same key. Below is the code for creating a dispatch that only outputs when incoming values are increasing and takes values from a midi device:

```
k = MKtl('nnkn0');
d = MDispatch.make(\up, k);
```

At the time when MDispatch was created several templates were written for tasks such as for soft paging, getting velocity values or filtering events.

MDispatch was an interesting experiment that allowed for some degree of re-usability of event logic code, nevertheless for varied reasons it did not gain wide adoption amongst users of the Modality toolkit and its development is currently paused.

6.5.2 FRP

The traditional method of dealing with incoming events is through callback functions. Functional Reactive Programming, or FRP, is an alternative paradigm for programming dynamic and reactive systems using first-class composable abstractions. The two main abstractions are event streams (sequences of discrete-time event occurrences) and behaviours or signals (time-varying values). Most of the original work on FRP was done on the Haskell programming language[5] [10, 11, 12, 13].

The FRP paradigm seemed promising for the construction of musical digital instruments. An FRP network could determine how events from physical controllers affect sound processes. To explore this possibility a set of classes for doing FRP in SuperCollider, part of the FPLib library [14], was created based on *reactive-web* [15] and *reactive-banana* [16].

FP-Lib has the same interface as *reactive-banana* for defining the event network: outputs are defined in terms of inputs using *combinators* (pure functions) applied to the signals or event streams in order to construct an *event graph*. To get events into the event graph the system has to register with external sources, the inputs (MIDI,HID,OSC,timers),

[5] Haskell is a modern, pure, lazy, statically typed functional programming language.

and to have any effect on the outside world it must perform actions based on the outputs of the event graph. The *event graph* together with *inputs* and *outputs* form an *event network* which can be compiled and activated and deactivated, respectively. The most important transformations when using combinators are:

- transforming event streams into signals and vice-versa.
- Merging event streams.
- Filtering events streams.
- Maintaining state that can be affected by event streams carrying state altering functions.
- Merging n signals using an n-ary function.
- Applying a time-varying function (stored in an signal) to an event stream allowing for recursive graphs.
- Dynamic event switching: changing the event graph based on an event occurrence.

Since all the functions used to construct the graph should be pure, it is possible to abstract a subset of the graph into a single function and be confident that the result will be identical due to referential transparency. Also, external sources connected to inputs and actions performed on outputs can be exchanged without changing the event graph. This facilitates building and testing a personal library of event logic functions that can be re-used for different instruments or different parts of the same instrument. Several use cases put forward by the Modality Team have been implemented using FPLib and so far the system as shown itself capable of creating complex event graphs to be used in digital instruments.

6.5.3 Influx - lose control, gain influence

The Influx concept starts from three practical and aesthetic assumptions: Mapping may be the most flexible part of a NIME; detailed mental models of the instrument may detract from listening while playing; and generally, surprise may be desirable for audiences and performers alike.

In Modality terms, Influx and its variants are transformer islands: `Influx` maps m named numerical input values to n output values by creating bipolar weights that determine how much each input value influences which output value. In the simplest case, each controller parameter will influence every sound process parameter by a different weight e.g., a random amount. These weights can be gradually entangled by randomising, or disentangled by blending toward a known set of weights. This approach allows heuristic exploration of mappings one would never make by hand, and gently forces players to really listen to how the instrument sounds when they play.

`InfluxSpread` can send these control values to multiple destinations. `InfluxMix` can receive “influence values” from multiple sources and can determine how much influence it accepts from which source.

`ProxyPreset` allows storing, blending and crossfading between settings of a process e.g., keeping traces of an ongoing performance which can be re-used as musical material. An Influx can use such a preset as a reference point as the center of its parameter space. This allows playing relative to a known setting, where, for example, zooming allows very subtle explorations of shadings within a known sweet spot in parameter space.

`EventLoop` can record any control data as events with key-value pairs, such as parameter names and values, event time, and other named values describing the event. This allows capturing algorithmically generated streams, performance data from input devices, and many others. One can modify playback by time-scaling, segment selection, playback direction and gradual scrambling of local event order; one can also go back in the history of recorded loops.

The control data variant `KtlLoop` also allows on-the-fly rescaling of numerical control data. The gesture can be scaled to larger or smaller ranges, and shifted by offsets. All these modifications can quickly be accessed in performance, and the opportunities they create are quite distinct from audio loops. In performance, a KtlLoop can replace a live input stream (e.g., realtime-acquired HID data), then the loop can be reshaped while playing. It allows polyphonic layering by letting a loop continue and having it auto-mutate, so each repetition is slowly shifting.

These heuristics may lead both to finding non-obvious but interesting mapping strategies which can be built into more traditionally well-controlled instruments, and to new concepts for playing single-person instruments with a flexible degree of familiarity or surprise, or multi-player/instrument ensembles based on networks of influence. In effect, it allows musicians to relinquish some control and gain influence in exchange.

6.5.4 SenseWorld DataNetwork

The SenseWorld DataNetwork was initially developed for easy data exchange with other programs [17], but within SuperCollider can also be used as a central “datahub”. Within this framework a single data stream is regarded as a *DataSlot*; multiple data streams that for some reason belong together (e.g., the data comes from the same device, or are datastreams of a similar type) are organised as a *DataNode*. The framework provides methods to query the current value of a node or slot, to set functions to be performed on the data, whenever new data comes in, or put the data automatically on a bus on the SuperCollider server (audio engine), where it can be used directly in synthesis processes, or Unit Generators can be used to process the data further. The framework also provides various methods for common data processing methods, such as calculating the mean, variation, gating, range checking, smoothing, etc. The result of each data processing unit is made available again on the *DataNetwork*, and can be used in the same way as unprocessed data. The framework also comes with a GUI that allows visualisation of the data, as well as controls for switching on printing the data to the post window, enabling recording of the data, or creating a bus on the server.

Data from devices accessed with Modality can easily be used as input to the DataNetwork and thus form a DataNode, and then further used in that framework.

7. CONCLUSIONS

We find Modality an interesting approach toward creating more fluidly playable performance setups for electronic music. By providing rich knowledge about known controllers and easy ways to collect and add this information for new ones, one can create very short controller setup code. By providing uniform access for many different controller protocols and semantic names for all elements, one can substitute a specific controller for another (or a stand-in GUI) rather quickly. In addition, the flexibility provided in connecting multiple sets of actions with one or more controllers allows creating setups which support more modal concepts of playing, using a small set of controls for many different combinations of processes within a single performance.

We hope that Modality contributes to making the creation of more complex performance setups accessible for more musicians; as more setups get realised with Modality, we will learn whether they also remain flexible to extension and change when reaching higher levels of complexity.

Acknowledgments

The Modality team is (in alphabetical order): Marije Baalman, Tim Blechmann, Till Bovermann, Alberto de Campo, Jeff Carey, Bjørnar Habbestad, Dominik Hildebrand Marques Lopes, Amelie Hinrichsen, Robert van Heumen, Hannes Hoelzl, Miguel Negrão, and Wouter Snoei.

Associated organisations are (in alphabetical order): BEK, the project *Design, Development and Dissemination of New Musical Instruments* of UdK Berlin/TU Berlin, supported by the Einstein Foundation, nescivi, and STEIM.

The Modality meetings have been funded by Bergen Kommune, Nordisk Kulturfond and Creative Industries Fund NL.

Authors of the paper are listed in alphabetical order.

8. REFERENCES

[1] C. Troillard, “Osculator user’s manual,” Wildora, Tech. Rep., 2012.

[2] STEIM, “junXion | STEIM,” online, 2014. [Online]. Available: http://steim.org/product/junxion/

[3] J. Malloch, S. Sinclair, and M. M. Wanderley, “A network-based framework for collaborative development and performance of digital musical instruments,” in *Computer Music Modeling and Retrieval. Sense of Sounds*. Springer, 2008, pp. 401–425.

[4] IDMIL, “Digital Orchestra Toolkit | IDMIL, McGill,” online, 2014. [Online]. Available: http://idmil.org/software/digital_orchestra_toolbox

[5] S. Ferguson and M. M. Wanderley, “The mcgill digital orchestra: Interdisciplinarity in digital musical instrument design,” in *Proceedings of the International Conference on Interdisciplinary Musicology (CIM), Paris, France*. CIM, 2009, pp. 70–71.

[6] “SC.symposium 2012.” [Online]. Available: http://www.sc2012.org.uk/conference/

[7] “Unit-lib.” [Online]. Available: https://github.com/GameOfLife/Unit-Lib

[8] “Modality Toolkit.” [Online]. Available: https://github.com/ModalityTeam/Modality-toolkit

[9] J. McCartney, “Rethinking the computer music language: SuperCollider,” *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, Dec. 2002.

[10] C. Elliott and P. Hudak, “Functional reactive animation,” in *ACM SIGPLAN Notices*, vol. 32, 1997, pp. 263–273.

[11] C. M. Elliott, “Push-pull functional reactive programming,” in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, 2009, pp. 25–36.

[12] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, “Arrows, robots, and functional reactive programming,” *Advanced Functional Programming*, pp. 1949–1949, 2003.

[13] A. Courtney, H. Nilsson, and J. Peterson, “The yampa arcade,” in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. ACM, 2003, pp. 7–18.

[14] “FPLib.” [Online]. Available: https://github.com/miguel-negrao/FPLib

[15] “Reactive Web.” [Online]. Available: http://scalareactive.org

[16] “Reactive-banana.” [Online]. Available: http://www.haskell.org/haskellwiki/Reactive-banana

[17] M. A. J. Baalman, V. De Belleval, C. L. Salter, J. Malloch, J. Thibodeau, and M. M. Wanderley, “Sense/stage - low cost, open source wireless sensor and data sharing infrastructure for live performance and interactive realtime environments.” in *Proceedings of the International Computer Music Conference (ICMC) 2010, New York City / Stony Brook, NY, USA, June 1-5, 2010*. ICMA, 2010.