Model-View-Controller separation in Max using Jamoma

Trond Lossius
BEK
trond.lossius@bek.no

Theo de la Hogue
GMEA
theod@gmea.net

Pascal Baltazar
L'Arboretum
pascal@baltazars.org

Tim Place
74objects LLC
tim@74objects.com

Nathan Wolek
Stetson University
nwolek@stetson.edu

Julien Rabin
GMEA
julien.rabin@gmail.com

ABSTRACT

The Model-View-Controller (MVC) software architecture pattern separates these three program components, and is well-suited for interactive applications where flexible human-computer interfaces are required. Separating data presentation from the underlying process enables multiple views of the same model, customised views, synchronisation between views, as well as views that can be dynamically loaded, bound to a model, and then disposed. Jamoma 0.6 enables MVC separation in Cycling'74 Max through custom externals and patching guidelines for developers. Models and views can then be nested for a hierarchal structuring of services. A local preset system is available in all models, along with namespace and services that can be inspected and queried application-wide. This system can be used to manage cues with modular, stringent and transparent handling of priorities. It can also be expanded for inter-application exchange, enabling the distribution of models and views over a network using OSC and Minuit. While this paper demonstrates key principles via simple patchers, a more elaborate demonstration of MVC separation in Max is provided in [1].

1. INTRODUCTION

1.1 Concept of Model-View-Controller separation

Model-View-Controller (MVC) is an architecture pattern for developing interactive computer applications that breaks the application's design into three distinct elements [2]. A *model* represents a collection of data together with the methods necessary to process these data. The *view* provides an interface for monitoring and interacting with the model. The *controller* is the link between the model and view, and negotiates information between them. MVC enforces a clear separation between processes, their states, and how these are being represented to the user. This separation results in each concept being expressed in just one place, which in turn makes the code easier to write and maintain. The architecture also makes it possible to have multiple views for the same model. In this way, views can be customised and adapted dynamically based on the needs of the user at any one time, without these changes affecting the model itself. Furthermore, this separation permits the developer to completely overhaul the look and feel of the application simply by reworking the views, without requiring any changes to the models.

While MVC separation is common within many programming domains such as web applications development, it is far less common in interactive computer music platforms such as the Cycling'74 Max environment [1]. In applications for real-time creative work, the model could be a synthesis or audio process algorithm, while the view provides the graphical user interface (GUI) for the algorithm. Alternatively, the view could also be an interface with hardware controllers, or an interface for inter-application communication using e.g., OpenSoundControl (OSC) [3].

Prior to Max 5, the programming of user interfaces tended to render the logical flow of a patcher's underlying algorithm undecipherable due to dense overlaying of objects and patch cords. In versions 5 and later, a patcher can be represented in two ways: *Edit Mode* and *Presentation Mode*. Typically, a user of the program will employ Edit Mode during initial development, organising a patcher's layout with an emphasis on the logical structure of the processing algorithm. Presentation Mode is then used to reorganise the layout into a more intuitive GUI for interaction, displaying only select objects of relevance and hiding patch cords. Still a tight connection remains between the algorithm represented within a given patcher and its interface, as they are typically coded together in the same patcher window. The interface can not easily be substituted or altered without touching the underlying algorithm when both are contained within a single patcher. MVC separation would be improved by instead storing the algorithm as a patcher that is separated from its interface, so that several different views could be developed for interaction with a single underlying model.

1.2 Jamoma project to date

Jamoma began as a system for developing high-level modules in the Max environment. It addressed concerns about sharing and exchanging Max patchers in a modular system, and leveraged this structured environment to provide an effective, efficient, and powerful means of automating

[1] http://www.cycling74.com. All URLs in this article were last accessed July 14th 2014.

and controlling patchers [4]. The source code is currently maintained as an open-source C++ framework, named *Jamoma Core* [2], consolidating several frameworks [5, 6, 7], and a number of implementations, the most complete being Jamoma for Max [3]. The most recent stable version of Jamoma for Max, version 0.5.7, was released early 2013 [4]. The upcoming version 0.6 of Jamoma implements improved model-view-controller separation, and depends on a new Modular framework within Jamoma Core [7]. This paper will not discuss the design of the C++ code in further details, but rather present how Jamoma 0.6 enables model-view-controller separation in Max, how models and views can be designed, and discuss benefits of this approach with respect to custom and alternative interfaces, namespace exploration, monitoring of changes in the patchers, management of mappings and states, and inter-application communication.

2. MVC IN MAX USING JAMOMA

In the following description, key terminology will be introduced using *italics*, the name of Max externals will be **boldface**, and object arguments and attributes, as well as messages communicated to and from objects, will be denoted using `monospace`.

Jamoma 0.6 for Max employs an object-oriented programming approach along side a client-server architecture in order to provide MVC separation. The *model* is a Max patcher that wraps a high-level entity such as a media generator or signal processor with an accompanying set of functional services. The *view* is implemented as a separate Max patcher to provide an interface for monitoring and controlling the services of one or many models, most commonly by means of a GUI. Considered in terms of object-oriented programming, the model and view are both classes, but their purposes differ. Considered in terms of client-server architecture, the model is a *server*, while the view is a *client* that *binds* to the server.

2.1 Setting up a model

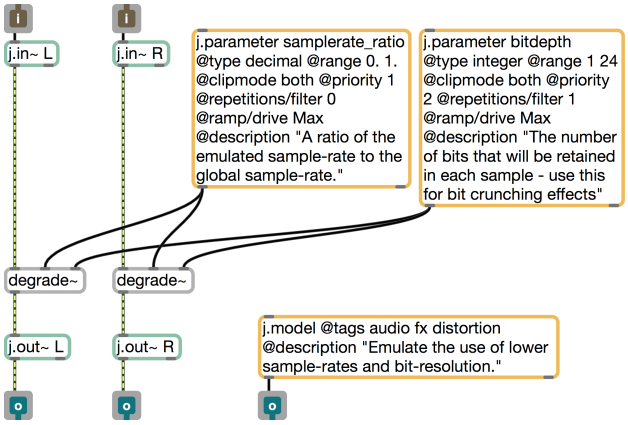

Figure 1. A Jamoma model.

[2] https://github.com/jamoma/JamomaCore/
[3] https://github.com/jamoma/JamomaMax/
[4] http://jamoma.org/download/

Figure 1 is an example of a simple Jamoma model. This stereo audio effect wraps a pair of **degrade~** objects, reducing bit depth and sample rate in order to distort the signal. A Max patcher is declared as a *model* if it contains a **j.model** object, with the attributes of this object primarily used to document the model. Specific uses of the `@tags` attribute will be explored in section 3.4. Within a model, **j.parameter**, **j.message** and **j.return** objects declare various *services* and specify the properties of these services as attributes.

j.parameter defines a *property* of the model. In Max, the associated value for this property can be easily set and queried. The *state* of the model is the ensemble of the values of all of its parameters.

j.message defines a *method* of the model. In Max, this results in a message that can be sent to the model to induce a specific action. Users should be aware that any argument(s) of such a message are not stored as part of the state of the model.

j.return is used to return control information to Max. For example, if a **metro** object were wrapped as a model, the user could enlist **j.return** to output information at each tick.

For the model in figure 1, two parameters are declared as services of the model, `samplerate_ratio` and `bitdepth`, but the model has no message or return objects. For each of the parameters, a number of attributes can be specified, as detailed in [8, 9], including the *type* of data they can hold and process, the *range* of values accepted, what to do when receiving values outside that range, whether it is possible to *ramp* to new values over time, and whether *repetitions* in the value will be filtered to save processing resources or not. The output from these parameters connects to the relevant inputs of the two **degrade~** objects they are to control.

This model receives audio signals to be processed, and returns the processed signals. The **j.in~** objects declare the signal inlets, while the **j.out~** objects declares the signal outlets. These two objects introduce a number of additional services for adjusting output gain, mixing dry and wet signals, muting or bypassing the model's audio processing, and remotely sending and receiving the incoming and processed audio signals. These additional services are collectively referred to as *audio amenities*. The argument to the **j.in~** and **j.out~** objects identifies the channel and informs the model what inlets and outlets are to be associated for mixing and bypassing. The benefits of these services will be discussed further in section 2.2, when a view is designed for this model.

Although this example demonstrates a stereo audio effect, model design is not limited to audio algorithms. Any patcher can be wrapped into a model, including those that process control data, audio, Jamoma AudioGraph multichannel audio signals [6] and/or video signals in the form of Jitter matrices, Jitter OpenGL textures, or other kinds of Jitter OpenGL data. In the same way that **j.in~** and **j.out~** provides common services of relevance to audio processing, dedicated objects are available for control rate data including video (**j.in** and **j.out**) and AudioGraph multichannel signals (**j.in=** and **j.out=**). Each of these offer amenities relevant to the stated data or signal type.

2.2 Designing a view

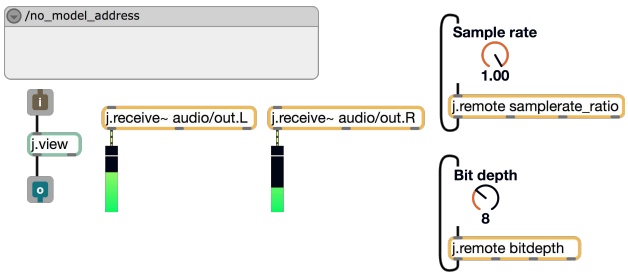

Figure 2. A Jamoma view in Edit mode.

A Max patcher is declared as a *view* if it contains a **j.view** external. In a similar way to how **j.parameter**, **j.message** and **j.return** declares services in a model, the **j.remote**, **j.send**, **j.receive**, **j.send~** and **j.receive~** objects can be used in views to bind to corresponding services within the model.

j.remote allows the view to both receive notification about updates from the model and send value changes to the model. It is the most common way to bind to a **j.parameter** within a model.

j.send instead communicates changing values to the model from the view, but does not inform the view about changes in the model. This will often be used when binding to a **j.message** within a model.

j.receive notifies the view about changing values from the model, but does not provide any means to control the model. Typically, this object will be used to bind to a **j.return** within a model.

j.send~ and **j.receive~** are audio rate counterparts of **j.send** and **j.receive**.

Figure 2 serves to illustrate uses of the above objects by presenting the default view for the audio effect discussed in section 2.1 with the patcher in Edit Mode. Two instances of **j.remote** bind to the corresponding **j.parameter** objects in the model, and can be connected to GUI objects within the view patcher so that the values of the parameters can be monitored and changed. Two instances of **j.receive~** bind to the signals sent by the pair of **j.out~** objects in the model, and provide a means for monitoring levels of the processed audio signals from the model. **j.ui** provides a default visual background for the view's GUI, while also offering access to additional services related to preset handling, as well as audio and/or video amenities, depending on the content of the associated model. The purpose of these will become apparent as we start using the model and view together in section 2.4. While **j.ui** provides convenient access to a number of services, it is not mandatory to include this object; developers are free to design the look and feel of a view in any way they want.

By default, views open in Presentation Mode, hiding the Jamoma controller objects and organising the layout of user interface elements, or *widgets*, into a functional interface. Some examples of views can be seen in figure 3.

2.3 Controllers

Jamoma does not require Max developers to develop a third patcher to act as a controller for models and views. Instead, controller responsibilities are mostly integrated into the various **j.*** externals presented and discussed throughout this paper.

When the Max application starts, the **j.loader** external initiates the Jamoma environment and sets up a global node directory for organising all future nodes into a tree structure [7]. Models and views are dynamically added to and removed from this directory structure as they are created or disposed off, ensuring global awareness of all the nodes that are available at any given time. The Jamoma environment enables behind-the-scenes communication between the **j.model** object and the various services in the model, between **j.view** and the various subscribers in the view, and between subscribers and the services that they bind to. It also enables behind-the-scenes communication of audio signals between pairs of **j.in~** and **j.out~** objects in audio effects models needed for dry/wet mixing and bypassing, as part of the audio amenities. Finally, Jamoma enables the various advanced abilities for querying, monitoring and controlling models that will be described in section 3.

2.4 Using models and views

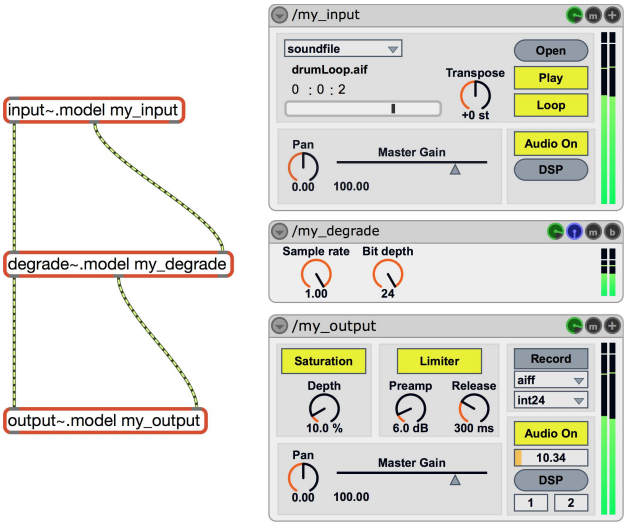

Figure 3. Three Jamoma models for audio processing, and their associated views.

Figure 3 is a simple Max patcher illustrating the use of three models and their corresponding views. Models and views are stored as separate files, and the convention is to use a `.model` suffix in the filename of models, and a `.view` suffix in view filenames. This patcher contains three audio models: `input~.model`, `degrade~.model` and `output~.model`. The input model wraps common sources of audio signals such as file playback, input from the sound card, and test signals. In figure 3, it is currently being used to play a sound file from disk. The degrade model was previously discussed in section 2.1. The output model offers saturation, limiting, stereo balance and gain adjustment as master effects before passing the signal to the system's audio outputs. Additionally, it provides the ability to record the audio signal to disk. Arguments to the three models provide each of them with an identifying name: `my_input`, `my_degrade`

and `my_output`. These names must be unique, so that views can unambiguously address a specific model instance.

The patcher also contains three views, each of them instantiated as a **bpatcher**. Each view needs to know which model instance it should bind to, something that can be set in two ways. In this patcher it is set statically as the first argument to the **bpatcher** in its inspector. **j.view** will grab this argument at load time. Alternatively it can be set dynamically by changing the value of **j.view**'s `model:address` attribute to the address of the chosen model. We will see examples of this later, in section 3.5. Once the view has been associated with a specific model instance, all instances of **j.remote**, etc., in the view will monitor and allow access to the corresponding services (**j.parameter**, **j.message** or **j.return**) of the model.

Comparing the look of the **j.ui** object of `degrade~.view` in figure 2 to the one in figure 3, four additional widgets have appeared in the upper right corner. When the view binds to the model, it is informed by the aforementioned pairs of **j.in~** and **j.out~** objects that the model offers specific audio amenities for output gain adjustment, mixing, bypassing and muting. **j.view** then instructs **j.ui** to provide additional widgets for accessing these services. **j.ui** can provide widgets for similar amenities when a view is binding to Jamoma AudioGraph multichannel audio models or video processing models. The upper left widget of all **j.ui** objects opens a pop-up menu offering access to a number of services for preset storing, recalling and interpolation, and querying of the current state of the associated model, collectively known as the *preset amenities*. The same pop-up menu also offers access to documentation for the model. There is an additional pop-up menu available under the name of the bound model, which will give a list of all services for this particular model, and allow users to see and edit all of their attributes in a pop-up window.

In figure 3, the models and views have been located side by side for simplicity, but they can just as well be located in different patchers or subpatchers. Readers familiar with complex, real-time interactive applications can imagine a collection of all models together in one patcher, with a corresponding collection of views in a separate patcher. This hopefully provides a glimpse of the benefits that MVC separation can bring to patcher organization in Max.

It should be noted that the **j.model** or **j.view** objects are mutually exclusive, and therefore cannot be located within the same patcher. This restriction is in place to enforce separation of a model and a view. However, a model may have additional **j.model** objects present in subpatchers, resulting in *nested* models, as discussed in section 3.1. In the same way, views can be nested using additional **j.view** objects within subpatchers.

3. MVC AND PATCHER MANAGEMENT

Implementing Max patchers with MVC separation using Jamoma helps address a number of real life problems experienced when working on larger applications and artistic projects in Max that might otherwise require advance programming skills. Separating models, controllers and views has proven to be useful in many other programming contexts, but has previously been difficult to achieve in Max because of challenges in implementing the controller layer. Because Jamoma provides a ready-made controller layer for Max, it frees the developer to concentrate on the development of models and views.

3.1 Nested models

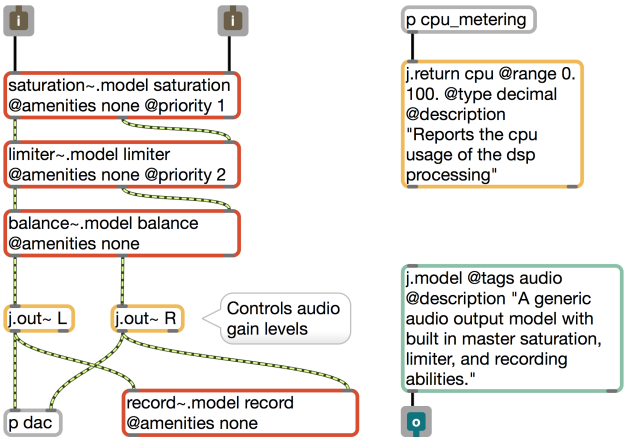

Figure 4. The output model embeds several other models, indicated as objects with red border.

One of the earliest motivations for making the transition to MVC in Jamoma was the ability to have nested models with a hierarchal structuring of services. The inner workings of the `output~.model` provides a good example of why this is useful. As mentioned in section 2.4, the output model provides saturation, limiting and stereo balance as master effects, in addition to audio recording capabilities. Figure 4 reveals how the output model provides these effects and capabilities via several nested models. Each of these models has functionalities that are useful outside of the output model, and rather than having duplicate implementations of effects such as the limiter, it is more DRY (Don't Repeat Yourself) [10] to create a `limiter~.model` and then embed this in other models as needed.

During our earlier discussion of the degrade model in section 2.1, the concept of audio amenities was introduced. By default, the limiter model would have these generic audio services for controlling gain, mix, bypassing, muting and level metering. It would also have services associated with presets. However, when used as a nested model, it might make sense to avoid duplication of the parent model's services and deactivate these services within the limiter model. To facilitate this, the `@amenities` attribute can be used to enable or disable these amenities using the keywords `all` or `none`, or more selectively specifying specific services, such as `audio` or `preset`. When present, this attribute is retrieved by the **j.model** object within the embedded model, and configures services accordingly.

Nesting models leads to well-structured and descriptive namespaces for the parameters in the model, as illustrated by the namespace of parameters in the output model listed in figure 5.

```
audio/gain
audio/mute
balance/mode
balance/position
balance/shape
dac/channel.L
dac/channel.R
limiter/active
limiter/dcblocker/active
limiter/lookahead
limiter/mode
limiter/postamp
limiter/preamp
limiter/release
limiter/threshold
record/file/type
record/samptype
saturation/active
saturation/depth
saturation/mode
saturation/preamp
```

Figure 5. Output model parameter namespace.

3.2 Management of priorities

When initialising a model or recalling a preset, proper configuration of the model is often dependent on ensuring that parameters are updated in a particular order. For example, when describing the loudspeaker layout for multichannel sound reproduction, the number of speakers needs to be set before the positions of the speakers. This can be administered using the `@priority` attribute of **j.parameter**.

In larger models, the nested sub-models might need to be configured in a prioritised sequence as well. Although it is not really needed for this particular model, figure 4 illustrates how a priority attribute can be set for the embedded models. Priorities are sorted according to a pre-order depth-first recursive traversal of the node tree [11]. The four embedded models of the output model reside at the same level of the node tree structure, as illustrated in figure 5, and since the saturation model has first priority, it will be the first to be set. The various parameters of the saturation model will be set according to how priorities have been specified internally in the model. Next the limiter will be set, and finally all the nodes that do not have any priority, such as the balance and record models.

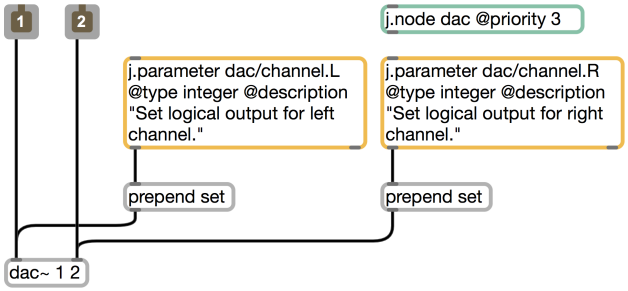

Figure 6. Setting priority of a node using **j.node**.

When creating model parameters, it is possible to design the namespace to use intermediate nodes without having to set them up as models of their own. Figure 6 shows the `dac` subpatcher of the output model, containing two parameters labeled `dac/channel.L` and `dac/channel.R`. In this case, the intermediate node `dac` is not declared anywhere as a model; instead, it is implicitly added to the namespace tree when the service is registered. If the priority of these two parameters should be raised in comparison with other nodes in the output model, it needs to happen at the node level where `dac` resides, rather than the sub-branch where the two nodes `channel.L` and `channel.R` reside. In order to accomplish this, a **j.node** object is introduced, raising the priority of the `DAC` and enforcing a priority for these two parameters that is after the saturation and limiter models, but before the stereo balance and recording models. **j.node** can be thought of as a lightweight cousin of **j.model**, providing control over a few select properties at a node level, but without turning that node into a full-fledged model.

3.3 Object instances

There are many situations where it is advantageous to use a specific model multiple times within a patcher, whether it be for parallel filters in an equalizing filter bank, or sources and speakers in spatial scene descriptions. Whenever a given model is used multiple times within the Max environment, there are several ways that Jamoma supports accessing these distinct *instances* of the model, starting with the namespace. Whereas the *forward slash* separator allows users to discriminate between successive nodal hierarchical levels, the *dot* separator is used to discriminate between different instances of the same class residing at the same hierarchical level. Instance identifiers can be provided as a numeric or symbolic suffix, depending on what best describes their scope. The addresses of the dac parameters `dac/channel.L` and `dac/channel.R` in figures 5 and 6 illustrate this, setting what physical output channel on the sound card the left and right channel signals of the model are supposed to send audio to. This method for dealing with instance naming represents a key difference between namespaces in Jamoma and OSC addresses [12]. OSC addresses such as `/channel/L` offer no way to discriminate between the functional description (`channel`) and the part used to describe the instance (`L`). By comparison, the dot separator in (`/channel.L`) makes this explicit for humans and computers alike.

Objects instances, be they models or services, can be created manually by providing the instance identifier as a suffix to the address argument. The binding mechanism of Jamoma requires that all nodes have unique names, therefore two models can not share the same address. Whenever a conflict arises, either by opening the same patcher twice or by duplicating a model in a patcher, Jamoma posts a warning to the Max window and temporarily solves the issue by introducing unique instance suffixes such as `my_output.1` and `my_output.2`. If this happens simultaneously for models and views, Jamoma will attempt to keep correct bindings between associated models in views while introducing unique identifiers for both.

It is also possible to create arrays of models dynamically by loading the patcher into a **poly~** object. In this case, the **poly~** instance number will be adopted by Jamoma to identify the model instance. Arrays of services can be created with the **j.parameter_array**, **j.message_array** and **j.return_array** externals. For example, parameters controlling the as-

signment of eight audio channels could be configured easily using `j.parameter_array channel.[8]`. It is then possible to create implicit nodes for each instance in this array of parameters. In the case of our eight audio channels, we could add parameters to control the speaker position on each channel using `j.parameter_array channel.[8]/position`.

3.4 Namespace management

The address of a service functions in a manner similar to file and folder paths within a POSIX-compliant terminal shell. Addresses with a leading forward slash are considered *absolute*, and start from the root of the local application. The absolute address of one of the parameters of the degrade model in figure 3 is `/my_degrade/bitdepth`. Absolute addresses are OSC-compliant [3]. Addresses with no leading forward slash are considered *relative* to the context of the current objects.

j.model sets up a local namespace context within the model, and if a parameter or some other service is provided with a relative address, the address will be relative to the address of the model instance. When creating a `my_degrade` model instance in figure 3, the address of the `bitdepth` parameter from figure 1 becomes `/my_degrade/bitdepth`. The same holds true for views. If subscribing objects within the view are provided with a relative address, it will be relative to the address that **j.view** binds to. When the degrade view seen in figure 2 binds to the `/my_degrade` model as seen in figure 3, the `bitdepth` **j.remote** object within the view will bind to `/my_degrade/bitdepth`.

When using absolute addresses for **j.remote**, **j.send** or **j.receive**, it is possible to design views that bind to services in several separate models. Using absolute addresses, it is also possible to use **j.remote**, **j.send** and **j.receive** outside of patchers strictly defined as views by the presence of a **j.view** object as described in section 2.2. The same applies to **j.send~** and **j.receive~**. The address to bind to can be set dynamically by setting the `@address` attribute of subscribing objects, as demonstrated in figure 7.

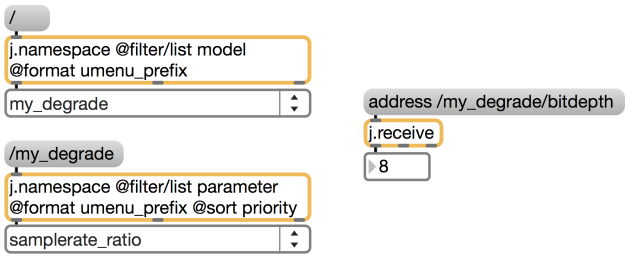

Figure 7. Querying of namespace for models and model parameter, and dynamically setting what service to bind to.

The namespace of a Jamoma application can be explored in several ways. Using **j.namespace**, the node tree namespace can be explored recursively node by node, or it can be explored using a flexible set of filters. Some pre-set filters are available in order to filter nodes based on the type of Max object they represent, such as **j.model**, **j.parameter**, **j.message** and **j.return**. It is also possible, with a dedicated syntax, to filter nodes based on what type of object they represent within the node tree (e.g., model, service, preset, or signal inlet), or on the value of their attributes. For instance the arguments of the **j.model** tag attribute in figure 1 identifies this class of models as as a distortion audio effect, and it will show up in namespace queries for nodes tagged with each of these keywords. Figure 7 illustrates how **j.namespace** can be used to build a menu of models, and also a menu of parameters for the `my_degrade` model. Combined with the ability to dynamically set what address subscribers bind to, this provides incredible flexibility for manipulating the mappings between parameters during development or performance.

While **j.namespace** narrows in on a few select nodes at a time, **j.modular** provides a bird's-eye perspective, reporting the whole namespace of the local application. This can be exported as an XML file, or used to announce the services of the local environment for use in query-based inter-application communication with other OSC-compliant or Jamoma-based applications. An example of inter-application communication will be provided in section 3.6

At the time of this writing, only a limited subset of the address pattern matches described in the OSC specification [13] have been implemented in Jamoma. Asterix can be used as a wildcard between forward slashes (`/*/`) to select all nodes at a certain level, and can even be used to select all instances of a certain kind, such as using `/channel.*/` to access all eight audio channels from the description in section 3.3. Because this is implemented within the controller infrastructure, it is a powerful feature for monitoring parameters. For example, if we designed a `my_midi_in` model to receive and parse incoming MIDI messages from an external device on all 16 channels, we could easily monitor all incoming continuous controller messages on all channels using `j.receive /my_midi_in/channel.*/cc.*`.

j.remote_array provides access to an array of values, either as a list of all values for all instances when set to the `@format array`, or individually for each instance when set to `@format single`. **j.remote_array** is not dependent on how the services of the node tree have been created in the patchers. For instance, if we create eight instances of the `input~.model` as introduced in section 2.4 and named `my_input.1`, `my_input.2`, etc., the volume of all models can be accessed using a single `j.remote_array /my_input.[8]-/audio/gain` object.

3.5 Multiple and dynamic views

It is important to note the direction of dependencies between a model and view: The view depends on the model, but the model does not depend on the view [14]. One implication of this direction is that a variety of views can be created for one and the same model. For a computer musician working with interactive systems as part of an extended performance instrument, the interface could be adapted to specific needs of a given situation. In rehearsal situations, a lot of details about the model may need to be exposed, while only a few services are relevant during the actual performance. It is also possible to load and dispose of views during the course of a performance, so that the

exposed interface dynamically updates to present only the information relevant to a specific moment in performance.

Another benefit, when combined with observer pattern implementation, is that it is possible to have several simultaneous presentations of the same parameter [14]. Section 2.2.3 of [1] provides an example of how several views towards the same audio filter can co-exist. One view displays the filter parameters as numerical values, while the other view provides a graphical interface displaying the resulting frequency response, using **filtergraph~**. The two subscribers do not interact directly, but a change in one will update the state of the model, and then be reflected by an update of displayed value in the other subscriber.

3.6 State management and inter-application communication

Managing states is a critical feature of a multimedia system. In Jamoma, there are two systems for state management: *presets* and *cues*. Both use the same underlying mechanism, with some important differences. The preset system is an embedded utility in **j.model**, and is local to the model (and its potential sub-models). Presets can be saved as text files and shared amongst several instances of the same model, and such presets are used for the default initialisation of models. The **j.cue** object stores and recalls states as well, but operates application-wide. It can store states for the entire system, or more selectively for designated parts of the namespace. Different cues stored in the same **j.cue** object might address different subsets of the namespace. **j.namespace** can be used with **j.cue** to provide an interface for selecting what subset of the namespace to use when storing new cues. Cues and presets both respects priorities as discussed in section 3.2.

The **j.modular** object facilitates inter-application communication using multiple protocols via a set of plugins within a generic framework [15]. This provides a way for new protocols to be added as they are developed, as well as existing protocols to be maintained within a consistent interface. At the time of this writing, two protocols are implemented as plugins and available within **j.modular**; *OSC* [3] and *Minuit*. The Jamoma implementation of *OSC* can be used to pass OSC messages between the two applications, and makes it possible to mirror a remote application after loading its namespace from an XML file. **j.remote** objects can then bind to parameters in the remote server application and create networked client views in Max for models residing in the other application. *Minuit* is a protocol that enhances OSC to allow querying of namespaces and the value of specific nodes within remote applications, as well as subscriptions to remote services [5] . Minuit also enables remote control of Jamoma model parameters, so that a remote application can provide networked views of local models within a Jamoma application. These abilities are used by the *i-score* application, an interactive intermedia sequencer. i-score can query and visualise the namespace tree of one or several Jamoma applications, and then remotely create snapshots or automate any of their services. The snapshots of application state can be further arranged as structural events in time in the i-score sequencer in order to author time-based multi-media interactive scenarios [16].

[5] https://github.com/Minuit/minuit

4. CONCLUSION AND FUTURE DIRECTIONS

In interactive applications, user interfaces are especially prone to frequent change requests. The ability to maintain a flexible interface design and respond to such change requests can be hindered if the user interface is tightly interwoven with the underlying processing algorithm. The Model-View-Controller architecture pattern divides such applications into distinct parts, separating interface development from other parts of the program's development and making it easier to remain flexible.

[14] argues that the most important separation is that between view and model, as they are fundamentally about different concerns. This point is just as valid in real-time music applications as it is in business applications: The development of the view should focus exclusively on UI layout and mechanics. In contrast, the development of a model should focus on processing signals, media and control data. Depending on context, users may want to see the same underlying model information presented in different ways. Separating model and view enables the development of multiple views that all use the same model. A key point of this separation is the direction of the dependencies; the view depends on the model, but the model does not depend on the view. Due to the implementation of a observer pattern, each view is updated whenever the associated model changes, so that several synchronised views can co-exist [17]. Views become 'pluggable', and can be substituted one for another dynamically at run-time.

Jamoma 0.6 introduces a number of externals that together enable MVC separation in the Cycling'74 Max environment. The network communication features of the controller layer in Jamoma even enables inter-application distribution of responsibilities, where views on one or more clients interact with models hosted at a server. This can be used for networked, distributed and collaborative performances. Designing models and views as patchers is straight-forward, and results in tangible improvements to the overall process of designing interfaces within Max.

This paper has provided several brief examples that serve to illustrate these improvements. For a more extended demonstration of MVC separation in Max using Jamoma, please see [1]. In that demonstration, an equaliser model is created with an array of filter bands of variable size. Next, a series of views are developed for displaying frequency response of the equaliser, and for interacting with a single filter band. Finally, all of the above is combined to provide a compound view that demonstrate the use of several parallel views towards the same service, dynamic binding of views to services, and the use of nested views. As such, the examples demonstrates many of the functionalities, possibilities and benefits that MVC separation offers.

MVC separation is also beneficial with respect to automated testing. The development of Jamoma relies heavily on such testing during each C++ build, as well as implementation testing in Max [18]. Non-visual objects are usu-

ally easier to test than visual ones, and separating model and view allows to test model logics without having to script GUI updates [14]. In the future, we anticipate that MVC separation will make it easier to develop model and intra-model integration tests.

The need to refactor Jamoma for MVC separation has been motivated by its developers own artistic and research practises. During the alpha development of Jamoma 0.6, it has been used for a number of large artistic projects in France and Norway, including works developed at GMEA, BEK, by The Baltazars and a new stage production currently in development by Verdensteatret.

Jamoma 0.6 is scheduled for release during the summer of 2014. It requires Max 6.1 or higher, and is distributed as a downloadable package [6] . Externals are currently available for Mac OSX only, however we invite assistance from experienced Windows developers to help in making it available for this platform.

Acknowledgments

The initial idea to refactor Jamoma to facilitate MVC separation emerged at a workshop hosted by iMAL in Brussels in 2007, and has been further refined in later workshops hosted by GMEA, BEK and fourMs lab, University of Oslo. Development has been supported by the French National Research Agency via the research projects Virage 2008-2010 and OSSIA 2012-2015, Hordaland County Council, Arts Council Norway and l'Arboretum. We would like to express our gratitude towards fellow Jamoma developers and all other artists, developers and researchers that we have consulted with in the process.

5. REFERENCES

[1] T. Lossius, N. Wolek, T. de la Hogue, and P. Baltazar, "Demo: Using Jamama's MVC features to design an audio effect interface," in *Proc. of the joint ICMC SMC 2014 Conference*, 2014.

[2] T. Reenskaug, "Models - views - controllers," Technical Note, Xerox Parc, Tech. Rep., 1979. [Online]. Available: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[3] M. Wright and A. Freed, "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers," in *Proc. of the 1997 International Computer Music Conference*, Thessaloniki, Greece, 1997, pp. 101–104.

[4] T. Place and T. Lossius, "Jamoma: A modular standard for structuring patches in Max," in *Proc. of the 2006 International Computer Music Conference*, New Orleans, US, 2006, pp. 143–146.

[5] T. Place, T. Lossius, and N. Peters, "A flexible and dynamic C++ framework and library for digital audio signal processing," in *Proc. of the 2010 International Computer Music Conference*, New York, US, 2010, pp. 157–164.

[6] ——, "The Jamoma audio graph layer," in *Proc. of the 13th International Conference on Digital Audio Effects*, Graz, Austria, 2010.

[7] T. De La Hogue, J. Rabin, and L. Garnier, "Jamoma Modular: une librairie C++ dediee au developpement d'applications modulaires pour la creation," in *Proc. of the 17es Journées d'Informatique Musicale*, Saint-Etienne, France, 2011.

[8] T. Place, T. Lossius, A. R. Jensenius, and N. Peters, "Flexible control of composite parameters in Max/MSP," in *Proc. of the 2008 International Computer Music Conference*, T. I. C. M. Association, Ed., Belfast, Northern Ireland, 2008, pp. 233–236.

[9] T. Place, T. Lossius, A. R. Jensenius, N. Peters, and P. Baltazar, "Addressing Classes by Differentiating Values and Properties in OSC," in *Proc. of the 2008 International Conference on New Interfaces for Musical Expression*, Genova, Italy, 2008, pp. 181–184.

[10] A. Hunt and D. Thomas, *The Pragmatic Programmer*. Addison-Wesley, 1999.

[11] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[12] A. Schmeder, A. Freed, and D. Wessel, "Best practices for Open Sound Control," in *Proc. of the Linux Audio Conference 2010*, 2010, pp. 131–139.

[13] M. Wright, "The open sound control 1.0 specification. version 1.0." CNMAT, Tech. Rep., 2002. [Online]. Available: http://opensoundcontrol.org/spec-1_0

[14] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[15] A. Allombert, R. Marczak, M. Desainte-Catherine, P. Baltazar, and L. Garnier, "Virage : Designing an interactive intermedia sequencer from users requirements and theoretical background,," in *Proc. of the 2010 International Computer Music Conference*, New York, US, 2010.

[16] P. Baltazar, T. de la Hogue, and M. Desainte-Catherine, "i-score, an interactive sequencer for the intermedia arts," in *Submitted to the joint ICMC/SMC conference*, Athens, Greece, 2014.

[17] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture. A system of patterns. Volume 1*. John Wiley & Sons, Inc., 1996.

[18] N. Peters, T. Lossius, and T. Place, "An automated testing suite for computer music environments." in *Proc. of the 9th Sound and Music Computing Conference*, Copenhagen, Denmark, 2012.

[6] http://www.jamoma.org/download/