Electro-Acoustic Tools
High-Level Abstractions for Audio Manipulation and Spatialisation

Richard Thomas
School of Music
University of Leeds
UK, LS2 9JT

Abstract

This paper introduces *Electro-Acoustic Tools* (EAT), a new set of abstractions for composition and diffusion that are currently under development in Pure Data Extended (Pd). EAT is accessible for preliminary educative use, but it also embraces the advanced functionality that Pd can provide for digital signal processing (DSP) effects and spatialisation. EAT has a modular architecture with instance specific MIDI and parameter preset functionality to drive open ended applications. EAT may be differentiated from pre-existing abstraction packages by its accessibility, dynamic patching, and flexible multi-point panning behaviours, which are the primary focus of the discussion here.

An earlier version of this paper was published in the conference proceedings of the 4th International Pd Convention convened 8–12 August 2011 at Bauhaus-Universität and Hochschule für Musik Franz Liszt in Weimar, Germany [1].

Keywords

Abstractions, spatialisation, dynamic, dyn~, tools.

Download EAT

You can download the latest EAT package from http://sourceforge.net/projects/eatpuredata/

Formatting Conventions

These formatting conventions are observed here:

Pd objects, externals, and abstractions are represented thusly: `[object]`.

Pd messages are represented thusly: `[message(`.

Connections between Pd objects with left-to-right data flow are represented by an intervening em dash thusly: `[message(—[object]`.

1 Introduction

There are a multitude of external and abstraction packages that are freely available for Pd. Many of them provide larger common building blocks from which complex applications may be built more quickly than through *ad hoc* solutions. Some abstractions are self-contained applications and have been provided with reduced GUIs for the end user via Pd's graph-on-parent (GOP) function. Arguably, the interfaces of existing abstractions and their accompanying documentation, if any, are often not particularly intuitive for inexperienced programmers. The EAT package aims to bridge the gap for novices to graphical programming environments and to bring original patching architectures and GUIs to Pd abstractions. Excluding the numerous lower level development abstractions, there are currently ten user friendly EAT abstractions for composition and spatialisation:

- `[EAT_Audiofile~]`
- `[EAT_Delay~]`
- `[EAT_Highpass~]`
- `[EAT_Lowpass~]`
- `[EAT_MatrixMix~]`
- `[EAT_MexicanWave~]`
- `[EAT_Mix~]`
- `[EAT_Record~]`
- `[EAT_Reverb~]`
- `[EAT_Transport]`
- `[EAT]` (shell)

1.1 Education

Excellent progress has been made of late to flatten what for many can be steep learning curve working in Pd; the tutorials of Dr Rafael Hernandez, the multi-authored Floss Manual, and Johannes Kreidler's *Loadbang* may be considered of particular

note in this regard [2, 3, 4]. Although not intended as a manual, Andy Farnell's *Designing Sound* contains an excellent introduction to programming in Pd and includes many example patches that provide some very practical solutions [5]. The official distributions of Pd and Pd-extended would benefit from a direct menu hyperlink to the Floss resource. EAT may help similarly to ingratiate Pd to novice users on a practical level, providing substantial DSP rewards for engaging with some of the program's basic conventions, including objects, control and signal connections, and arguments. EAT represents a preliminary tool for educators to teach DSP without proprietary software and provides opportunities to investigate more advanced Pd techniques at a latter stage of tuition.

1.2 Differentiating EAT

EAT was designed with generic and modular structures that are independent of a centralised abstraction. HID control via MIDI learn and control functions sets EAT apart from comparable applications; in combination with its decentralised state saving facilities, EAT provides a superior environment for real-time interfacing.

EAT's ease of integration into conventional patching structures will help to engender a familiarity with the flexibility of graphical programming in Pd and demonstrate an alternative to the sequencing and mixing styles of higher level packages (like Audacity or Ardour). Although EAT adds an extra syntactical requirement of an identity tag for instantiation, it provides the facility for multiple state saves outside of a housing abstraction – adding flexibility and efficiency over alternative applications.

1.3 Progress in Usability

Usability and accessibility will be some of the key features of the EAT abstractions. Since this project has only recently been released to the Pd community for testing (2 August 2011), it is too early to make any concrete assertions about EAT's usability. Developing accessible abstractions for EAT has required focus on areas including, but not limited to: on-screen links to help files, text help files, graphical help files using GEM to explain the GUI, consistent use of familiar font types, high contrast interfaces, simple syntax, and palatable data representation and parameter controls. Amongst its other objectives, EAT aims to make complex DSP available to novice users. These areas are key to this type of development, but the most beneficial information will be feedback from end users.

2 Novel Patching

The EAT modules provide functionality in well trodden areas for abstractions: sound file playback, DSP effects, and panning. All modules are equipped with integrated MIDI-learn and parameter preset functionality, with data written to text files and globally recalled via delocalised sends from a coordinating abstraction. EAT contributes a unique approach to abstraction design in three principal areas: accessibility, efficiency and flexibility, and multi-point panning.

Pd's GUI objects are widely used in EAT to create accessible and homogenous GOP interfaces across all modules (see Fig. 1–9, below). A limited palette of black, turquoise, grey, and coral was used to create the main GUIs for EAT. The turquoise on black colour scheme was implemented to provide a sense of familiarity to a novice audience, who are used to displays found in consumer electronics such as compact disc players and personal hi-fi systems. Coral was used occasionally because it contrasts well with the core palette of turquoise, grey, and black. Using black control targets on a light turquoise background improves visual performance and is subjectively preferable to end users [8]. The continuity and clarity of interface found in EAT is as enticing for the first time user as it is crucial in establishing conventions of functionality and user interaction [9].

EAT uses dynamic patching, discussed in-depth here, to provide efficiency and flexibility of DSP by creating multi-monophonic modules of between one and thirteen channels. This approach to abstraction design also reduces the labour required to create new multi-monophonic modules and the EAT modules may be considered easily extensible in this regard. EAT also takes advantage of a previous project conducted by this author that aspired to create a comprehensive Pd environment for basic diffusion, DSP effects, and complex panning behaviours. The key piece of development in the earlier project, a variable sine panning function, has come to prominence here with newly developed features in the EAT panning behaviours, discussed later.

3 Dynamic Object Management

Dynamic object management is broadly overlooked in most Pd documentation. It is a subject that occasionally comes to light in lists and forum activity, but has hitherto been an area lacking thorough documentation. Where audio tools are required to dynamically alter their constitution, developers have tended to shy away from using Pd in favour of other programming environments,

due to fears over stability and audio drop-outs. This paper demonstrates that, in the scenarios presented here, these fears are largely unfounded.

3.1 Why use dynamic techniques?

EAT was conceived with the intention of providing the user with multi-monophonic effects (see Fig. 10, below). The dynamic creation of objects is done during instantiation in response to abstraction creation arguments. Creating objects in Pd when audio processing is enabled is often feared to cause audio drop-outs, but experiments conducted during this project encountered no such occurrences – with the exception of abstraction creation. With large EAT modules, where several hundred abstractions are nested within, there could be a DSP drop-out of several seconds. It is expected that all modules would be loaded before commencing performance; dynamic object creation involving abstractions is not suitable for on-the-fly performances where drop-outs are unworkable.

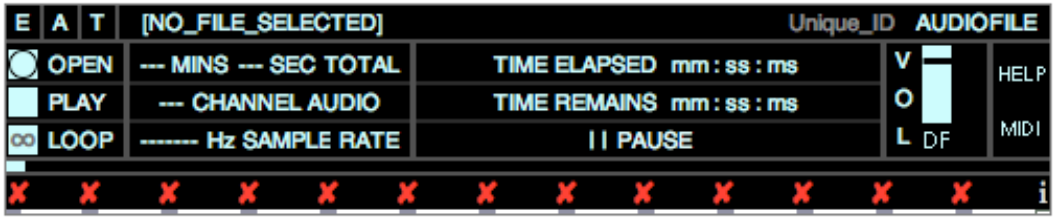

Fig. 1: EAT_Audiofile~ with no file loaded.

Fig. 2: EAT_Lowpass~ with 13 channels of 10-pole filters.

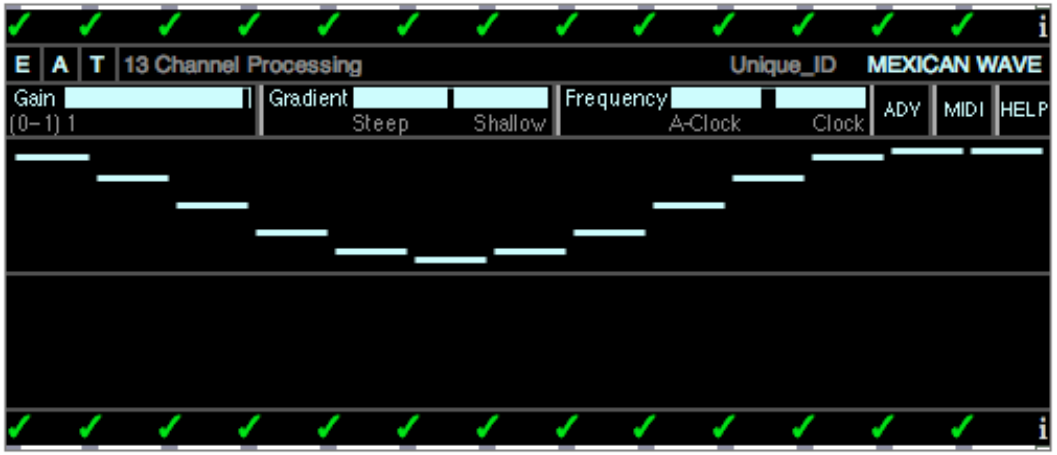

Fig. 3: EAT_Mexicanwave~.

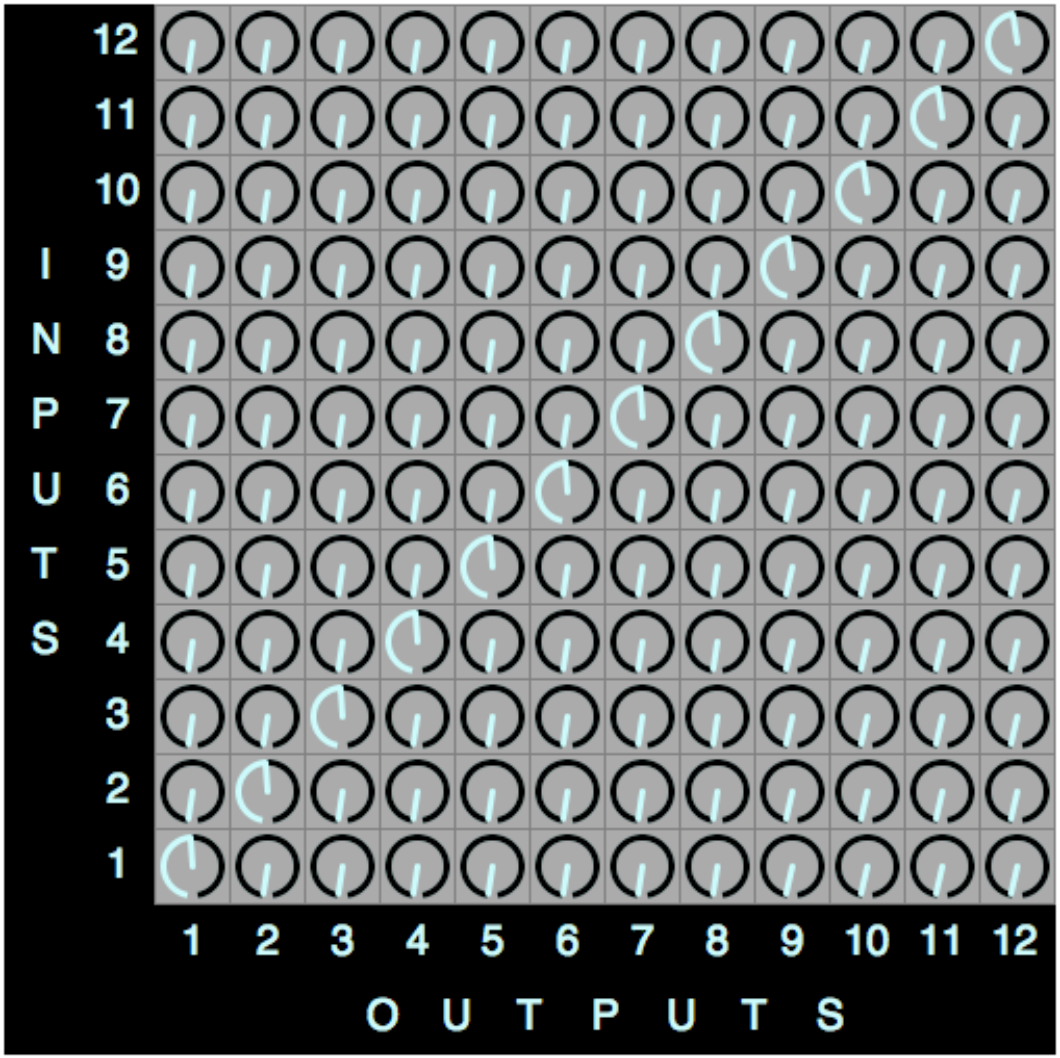

Fig. 4: EAT_Mexicanwave~ > ADV, 12x12 matrix mixer.

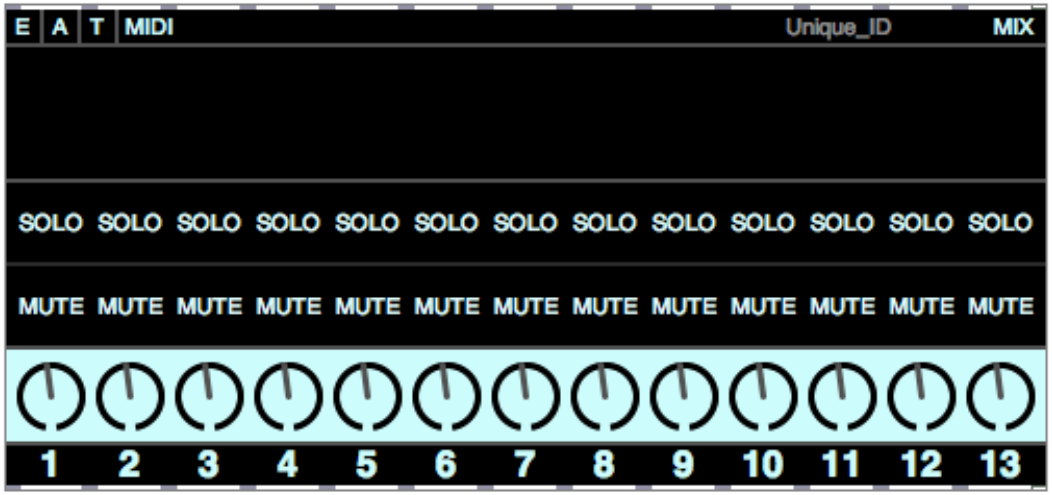
E
A
T
MIDI
Unique_ID
MIX
SOLO SOLO SOLO SOLO SOLO SOLO SOLO SOLO SOLO SOLO SOLO SOLO SOLO
MUTE MUTE MUTE MUTE MUTE MUTE MUTE MUTE MUTE MUTE MUTE MUTE MUTE
1 2 3 4 5 6 7 8 9 10 11 12 13

Fig. 5: EAT_Mix~.

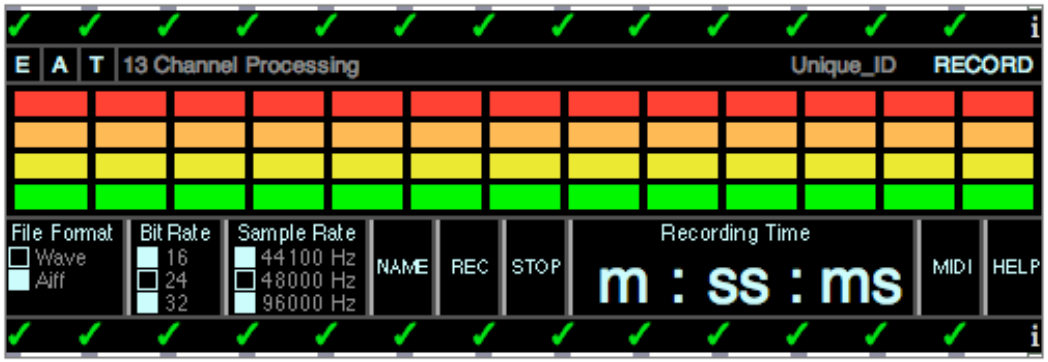

Fig. 6: EAT_Record~.

Fig. 7: EAT_Reverb~.

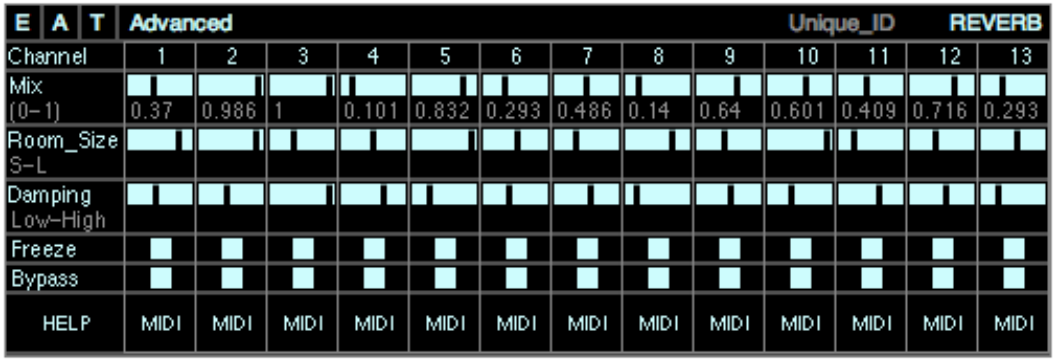

Fig. 8: EAT_Reverb~ > ADV.

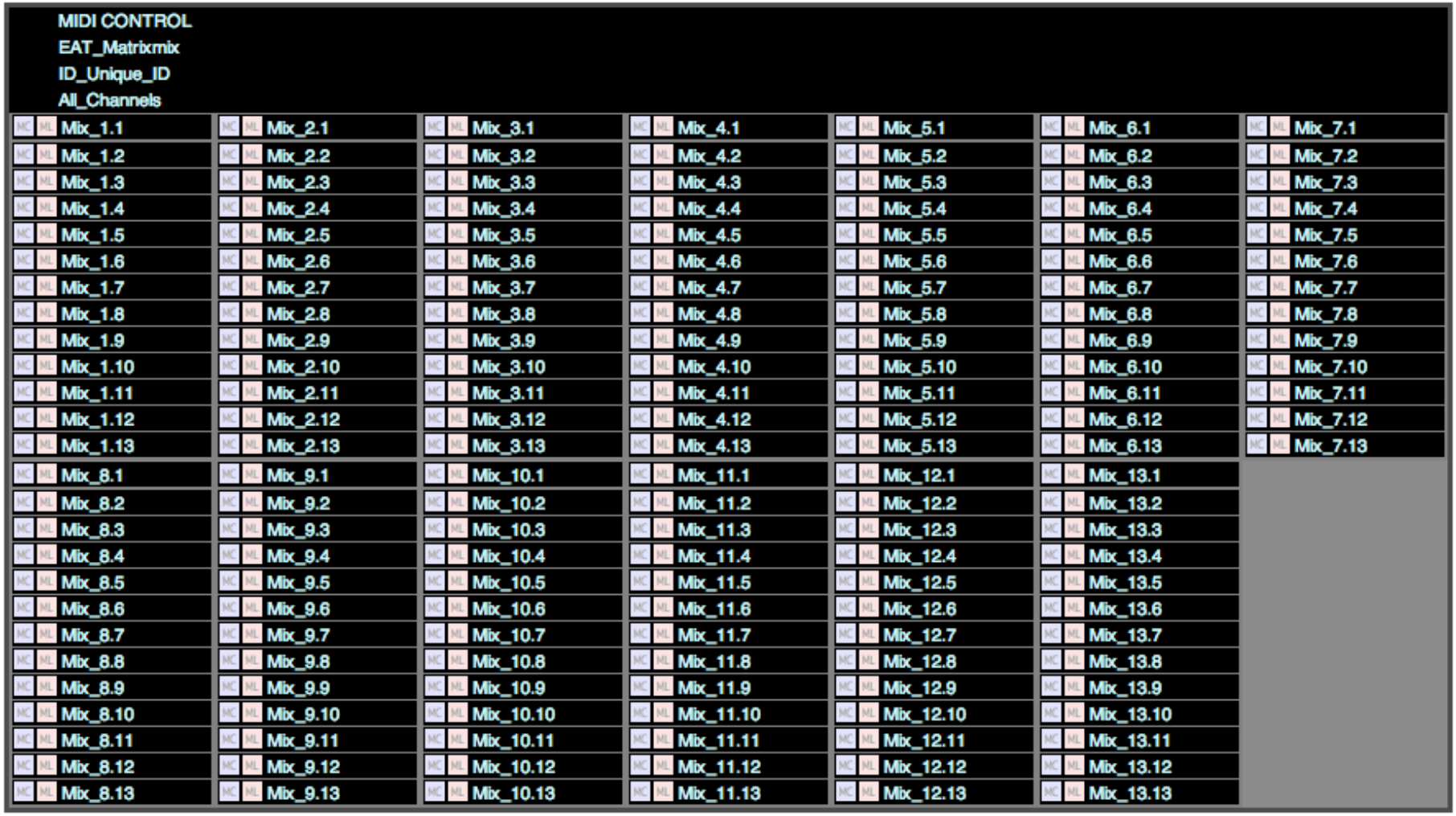

Fig. 9: MIDI Learn and Control menu in EAT_Matrixmix~

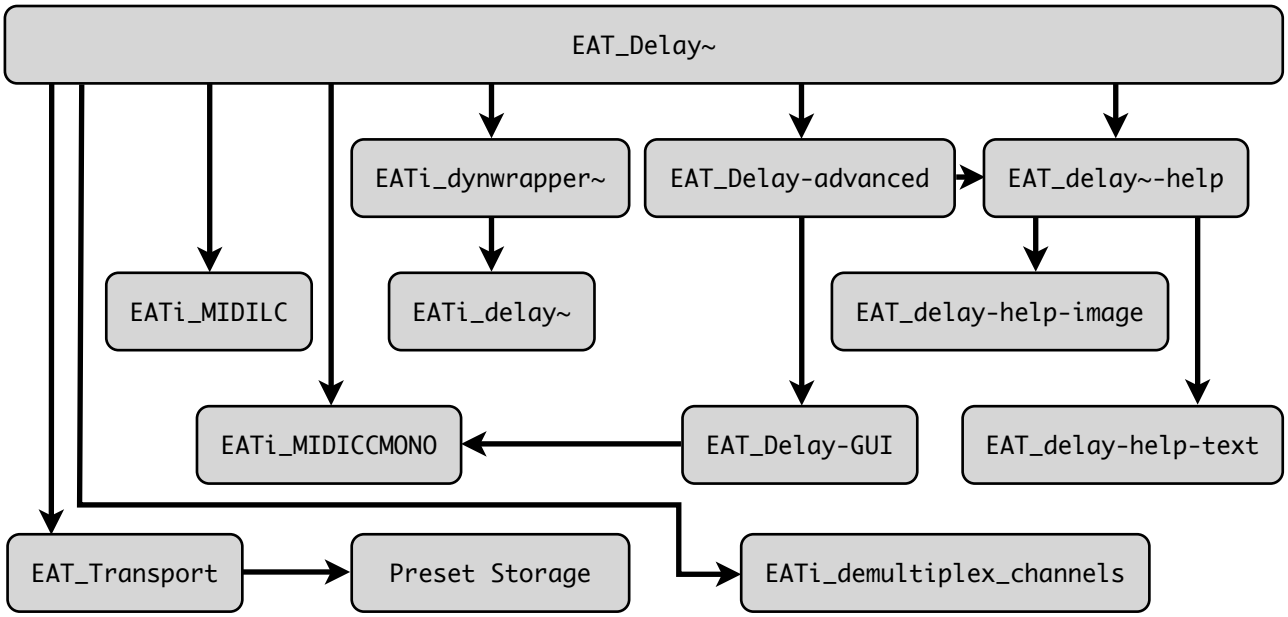

Fig. 10: An abstraction of the structure of multi-monophonic DSP effects in EAT using `[EAT_Delay~]` as the example.

3.1.1 Hardcoding

Producing hardcoded multichannel variations of every DSP effect contained in EAT would create a substantially bigger abstraction package and make the abstraction creation arguments less intuitive. The general atom format for the creation EAT modules is `[EAT_Effect~ ID_tag n-channels]`, e.g. `[EAT_Delay~ vocals 13]`.

3.1.2 `[switch~]`

Maintaining the syntax, the same end may be achieved by using the `[switch~]` object inside the mono effects patches. `[switch~]` may be toggled to turn digital signal processing on or off for that patch and any sub-patch or abstract descendants contained within. The disadvantage of this technique is that the GUI will contain obsolete control elements for disabled channels that will still draw on the computer's resources and will not always resemble the active functionality of the patch. With dynamic object management, GUI objects can be made to appear, disappear, and change their form, which is useful for modular multichannel systems like EAT; parameters of multi-monophonic EAT modules are replicated across all channels, but the maximum number of channels are not always instantiated.

3.1.3 Dollar and Hash Variables

The creation arguments of EAT modules are inherited by nested abstractions using $\$n$ arguments. EAT parameter controls use the in-built send and receive properties of Pd's GUI objects to inherit instance names and channel numbers. There is a need to clarify the nature of these arguments in different situations. In object boxes, $\$n$ arguments inherit the housing abstraction's creation arguments, with the exception of the local \$0 counter.

In GUI property sends or receives, $\$n$ arguments only work at the start of the name (e.g. \$0-Oramics); at other positions in the name, a hash symbol must be used (e.g. Oramics-#0, see Fig. 11. With $\$n > 0$ arguments, the \$ is automatically substituted for # accordingly, but it is important to be aware of the variation as a matter of course. Note that this discussion has no bearing on dollar signs used in messages, which are message time variables that have nothing to do with object arguments.[1]

3.2 Native Dynamic Patching

3.2.1 Object Creation

Objects, including abstractions, can be created by sending the appropriate message to a patcher via the special 'pd' target, which is not to be confused with the 'pd' prefix used for subpatches, e.g. `[;pd-mypatcher.pd obj 30 350`

[1] Cf. Miller Puckette's Pd manual for clarification of the difference between \$ arguments in objects and messages [10].

`metro 1000(`. This is identified as the newer, *correct* method for addressing a given patcher in namecanvas-help.pd. The older verbatim method, using a separate message and receive object, provides the same facility, e.g. `[obj 30 350 metro 1000(—[s pd-mypatcher.pd]`.

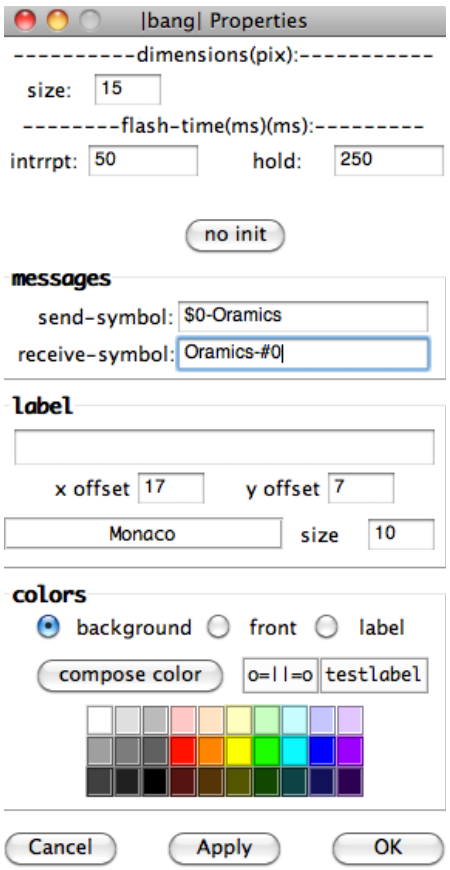

Fig. 11: GUI Properties Window.

3.2.2 `[namecanvas]`

In earlier Pd releases, this system for addressing patchers could only be achieved using the `[namecanvas]` object, which despite its inclusion in the current release has long since been declared obsolete in help-intro.pd. `[namecanvas]` allows the user to address its containing patcher via an alias provided by an argument, such as `[namecanvas Matthews]`, where the object creation message to the canvas could be `[; Matthews obj 21 4 max 2011(`. The `[namecanvas]` alias may also include the unique $0 counter.

Although antiquated, `[namecanvas]` remains a uniquely important tool in the Pd library. When there is a need to identify individual instances of abstractions—EAT requires this facility for instance specific MIDI CC and parameter preset recall and dynamic object management—`[namecanvas]` is the only solution. `[namecanvas]` must be used within the abstraction to create an alias containing the $0 counter, which prevents cross-talk between abstraction instances. There is currently no alternative method for achieving the same result via the global 'pd' target, since the canvas of an instance cannot be targeted using an abstraction's creation arguments. The recommendation must be made that the status of `[namecanvas]` as obsolete should be revised in future Pd distributions, until an alternative facility can be provided.

3.2.3 Connections

When making inlet and outlet connections, the objects are referred to sequentially according to the order in which they were created in the patch, numbering from zero. Object inlets and outlets are referred to sequentially in left-to-right order, again numbering from zero, e.g. `[; pd-mypatcher.pd connect 3 0 2 1(`, which would connect the first outlet of the fourth object created to the second inlet of the third object created.

It is plain to anticipate the difficulties one encounters with dynamic patches where more than a few objects are required; the order of object creation quickly becomes unfathomable. When one also considers the circuitous method for the removal of objects, imitating cursor selection and cutting, native dynamic object management represents an arduous prospect of trial and error. Cut and paste editing also affects the order of object creation, which does not improve the method any. A bug exists under 0.42.5-extended, when connections are made using the above method within an abstraction at initiation time, ghosts of the patch cords appear on the root canvas. One method to reduce the difficulties encountered with expansive dynamic patches is to house groups of objects within dynamically created sub-patches, but this is still far from ideal. The finest solution has been provided by Thomas Grill with his external `[dyn~]`.

3.3 `[dyn~]`

Thomas Grill's `[dyn~]` external provides a much more comfortable facility for dynamic object management. It allows the user to assign identifying (ID) tags to new Pd objects, bypassing the difficulty of object indexing by order of creation that is encountered with the native technique. `[dyn~]` is not without its own set of difficulties and idiosyncrasies, some of which might seem rather damning given its intended functionality, but these can be overcome with careful management.

3.3.1 Arguments and Message Inputs

[dyn~] takes four integer arguments: signal inlets, control message inlets, signal outputs, and control message outputs. All object creation and connection messages are sent to the left-most inlet, which conspicuously appears as a signal inlet. All objects are created within the external's self-contained canvas. The [dyn~] canvas can be opened and closed using the Boolean [vis *n*(message, which would be [vis 1(to open or [vis 0(to close.

Objects are created on the [dyn~] root canvas, denoted by the dot, using the message format [newobj . ID_tag object_name argument_1 argument_2...(, e.g. [newobj . thunderous pipe 1000(. The ID tag assigned to the object, here assigned 'thunderous', allows you to control the specific object instance using further messages sent to the far-left inlet of [dyn~]. The method of message creation is similar, using the message format [newmsg . ID_tag content content...(, e.g. [newmsg . PH flush(.

[dyn~] inlets and outlets are numbered from left-to-right, starting from zero, in the same manner as the native method. Connections are made by sending the message [conn ID_tag outlet_number ID_tag inlet_number(, e.g. [conn PH 0 thunderous 0(, which in this example would connect the first outlet of [flush(to the first inlet of [pipe 1000] on the [dyn~] root canvas. This is a much simpler and more intuitive process than the native dynamic connection technique. To delete an object the message format is [del ID_tag(. To clear all objects and connections the message is [reset(. To re-instantiate all objects and connections the message is [reload(, but this functionality does not seem to have been implemented fully at the time of writing.[2]

3.3.2 Workarounds and Patching Strategies

As mentioned in Grill's 2004 paper on [dyn~], the signal inlets do not work [11]. This would appear to sideline the external somewhat for use in dynamic audio environments. Greg Hynds suggested a workaround that uses the facility that [dyn~] provides for object creation to solve the issue [12]. If the user first creates audio receives inside [dyn~], then audio can be sent to them remotely. The minor difficulty with this workaround — when creating a set number of audio channels at instantiation as is the case with EAT — is that the user must then dynamically create sends in [dyn~]'s enclosing patcher, which must be done natively. Similarly, there is no GOP functionality for the [dyn~] canvas and therefore the GUI elements of effects for EAT had to be created separately, again using the native method.

Although abstraction arguments are inherited into the [dyn~] canvas, it is not possible to parse the $ symbol into newly created objects since this is replaced at message time. Dollar variables should be expanded before the creation message is sent to [dyn~], e.g. [bang(—[symbol $3]—[newobj . chorus.$1 my_chorus~(—[dyn~ 0 4 2 3].

3.3.3 [EATi_dynwrapper~]

The [EATi_dynwrapper~] abstraction, as the name suggests, is a wrapper for [dyn~] that aides the creation of multi-monophonic effects of up to thirteen channels (see Fig. 12). The wrapper could be adjusted to generate an infinite number of effects channels, limited only by the host's resources. A limit of thirteen channels was imposed to streamline the EAT modules' capabilities and appearance and to prevent accidental syntax errors that might hang the system. The wrapper takes three arguments in the format [EATi_dynwrapper~ mono_fx_name ID_tag *n*-channels]. Inside EAT modules, of course, the second and third arguments would be inherited from the parent using dollar variables, since the wrapper would be nested within the main abstraction that contains the GOP GUI.

There are thirteen inlets and outlets that have sends and receives dynamically created inside and outside [dyn~] and then connected depending on the wrapper's third argument (see Fig. 13). The wrapper is connected up to thirteen channels inside its parent, but it only creates the guts for the number of channels required (see the monophonic DSP effects created inside [dyn~] in Fig. 14). This permits the instantiation of a single abstraction for between one and thirteen channels depending on the arguments provided. The DSP effects are created with channel numbers and the unique ID inherited from the parent abstraction, which complete the dot delimited sends and receives inside the effects (see Fig. 15). It is not clear from Grill's paper what processes are duplicated using [dyn~] for this purpose, but preliminary testing suggests that it is an efficient solution.[3]

[2] [dyn~] version 0.1.2 tested under OS X 10.6.7.

[3] There is scope for a formal study of the effects of dynamic patching on DSP across different systems and workloads.

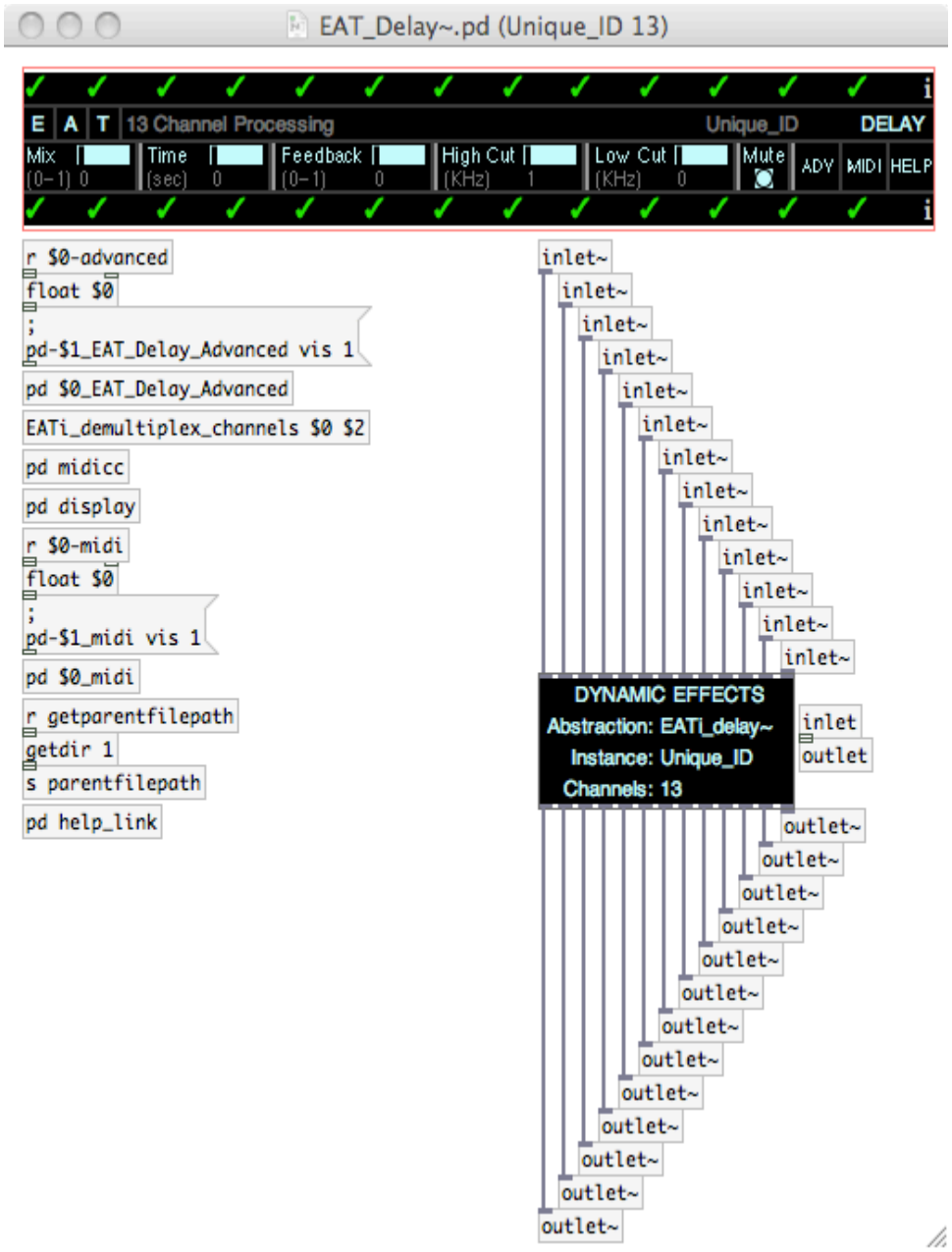

Fig. 12: [EAT_Delay~ Unique_ID 13].

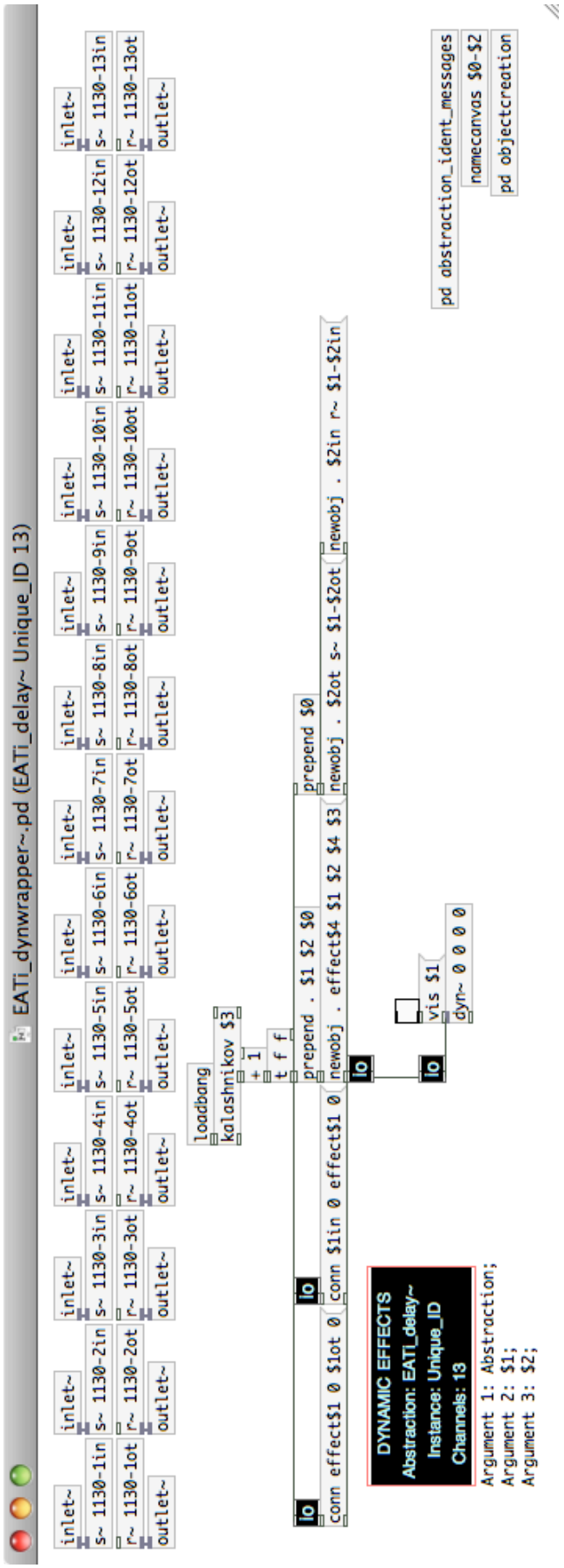

Fig. 13: `[EATi_dynwrapper~ EATi_Delay~ $1 $2]` nested within `[EAT_Delay~ Unique_ID 13]`.

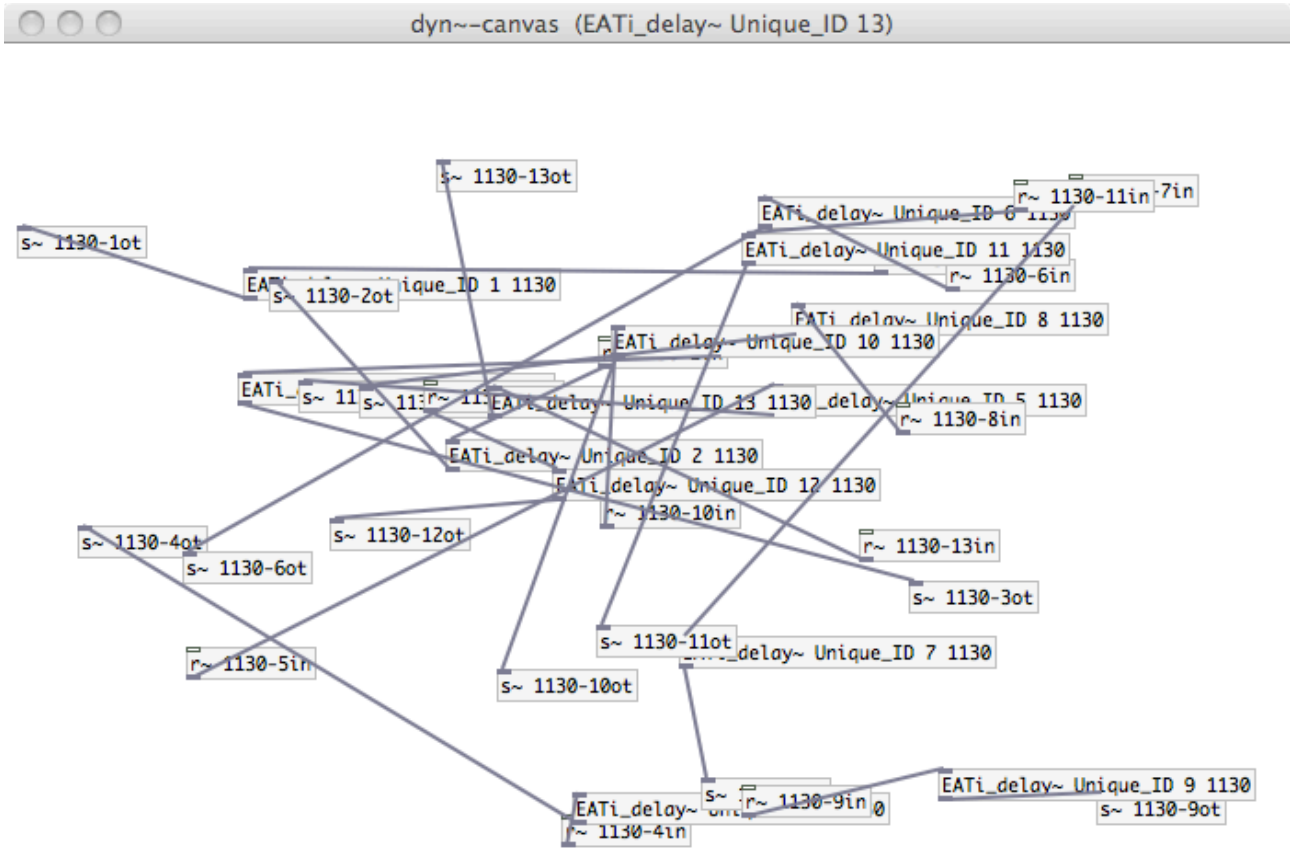

Fig. 14: [dyn~] canvas within [EATi_dynwrapper~ EATi_Delay~ $1 $2].

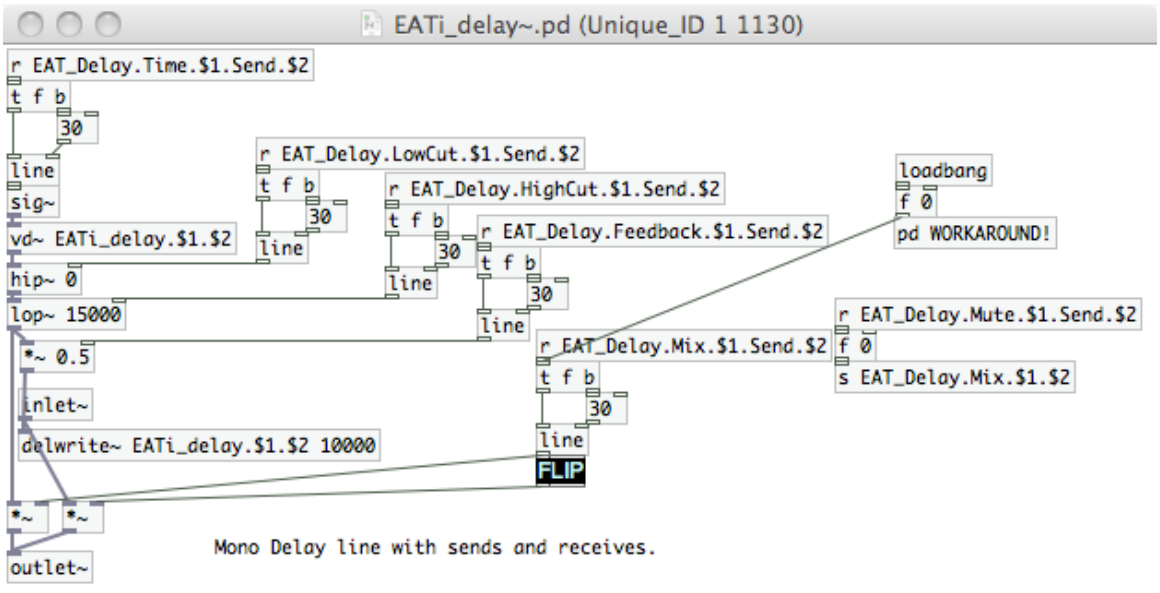

Fig. 15: [dyn~] canvas within [EATi_dynwrapper~ EATi_Delay~ $1 $2].[4]

[4]The workaround in Fig. 15 uses [loadbang] to send [float 0] to [*~], since it was discovered that [*~] would cease to function without it. The reason for this error is unknown, but the workaround is successful.

4 Multi-point Panning

`[EAT_MexicanWave~]` was the first panning module developed as part of the EAT toolkit. It takes advantage of sine waves, in a similar manner to equal power panning, in order to produce a variable window width.

4.1 Routing

In order to facilitate a dynamic input to output ratio of audio channels, an output channel number can be set by a creation argument between 1–12 (giving 2–13 channels, since channel 13 is reserved for a subwoofer signal that is always active, but not subject to the same type of gain control). The distribution of input channels to output channels could be handled externally, using the `[EAT_MatrixMix~]` module or an alternative *ad hoc* solution, but a 12x12 matrix mixer was integrated into a sub-menu with a default one-to-one input-to-output routing to provide more autonomous flexibility.

4.2 Algorithm

A complete cosine wave varies between 1 and -1 over a period of 2π. The intention of the algorithm is to reproduce a cosine wave over the available channels to create a Mexican wave panning behaviour. The cosine wave is stretched to a period equal to the required number of output channels, determined by the abstraction's third creation argument, by dividing the input to the cosine argument by the third creation argument. The output of the algorithm needs to be within a useful range of between 0 and 1, which is achieved by halving the output of the cosine (changing the range from -1 to 1 into -0.5 to 0.5) and adding 0.5 to then shift the wave into the ideal 0 to 1 range. Where y = channel gain control value, variable x = x-coordinate on wave ramping between (channel number) and (channel number+v) at a frequency of between -10Hz and 10Hz, and v = total number of channels, the following equation represents a complete implementation of a sine wave panning algorithm[5] (see Fig. 16, below): $y = \frac{1}{2}\cos(x\frac{2\pi}{v}) + 0.5$

In order to extend the functionality provided by Resound, the system developed by Dr David Moore and Dr James Mooney that inspired this module, a variable to allow a customisable sine window was inserted [13]. The cosine range needed to be altered by a positive variable (u) so that it varied sinusoidally between the lowest fixed point defined by $\frac{(u-2)}{u}$ (with a deepest trough of $-65\frac{2}{3}$ where $u = 0.03$, and maximum of 0.98 where $u = 100$) and a constant peak of 1. The output range reduction (previously $\frac{1}{2}$) becomes $\frac{1}{u}$ and the position compensation (previously +0.5) becomes $\frac{(u-1)}{u}$ to bring the maximum wave value to 1, giving: $y = \frac{1}{u}\cos(x\frac{2\pi}{v}) + \frac{(u-1)}{u}$

To prevent negative values from being distributed to input-to-output attenuators (where $u < 2$), a max function is included to ensure the output value never falls below 0. 2π is pre-calculated and the u variable factored out to give the implemented formula: $y = \max(\frac{1}{u}(\cos(x\frac{6.283185}{v}) - 1) + 1, 0)$

This formula was implemented using a tidy Pd expression, where x = \$f1, v = \$f2, and u = \$f3: `[expr max((cos($f1*(6.283185/$f2))-1)/$f3+1, 0)]` (see Fig. 17, below). This provides a sinusoidal wave that can be attenuated by varying u to achieve a subtle variation of gain between channels (see Fig. 18, below), or a sharp pan between individual channels (see Fig. 19, below). Further modules, with multiple or complex waves for instance, could be developed quickly and easily using this algorithm as a starting point.

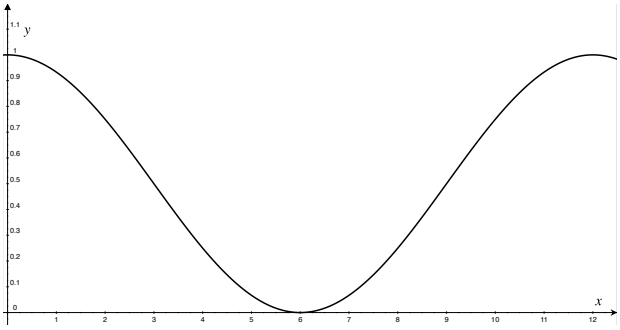

Fig. 16:
$y = \max(\frac{1}{u}(\cos(x\frac{6.283185}{v}) - 1) + 1, 0), u = 2, v = 12$

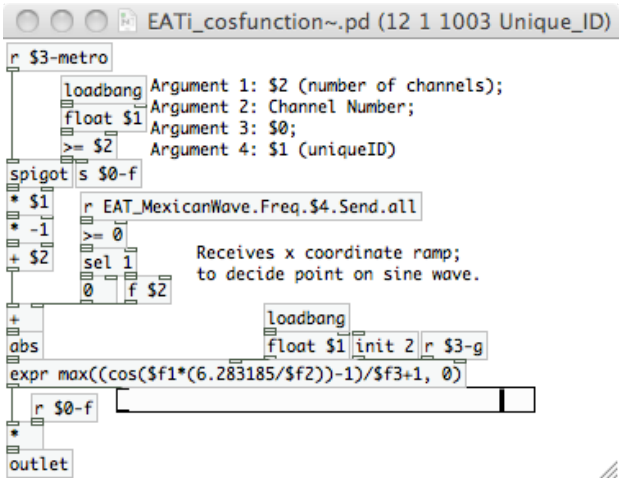

Fig. 17: `[EATi_cosfunction~ $2 1 $0 $1]` within `[EAT_Mexicanwave~ Unique_ID 12]`.

[5] When $u = 2$ the function illustrated in Fig. 1 provides the same output as the function shown here.

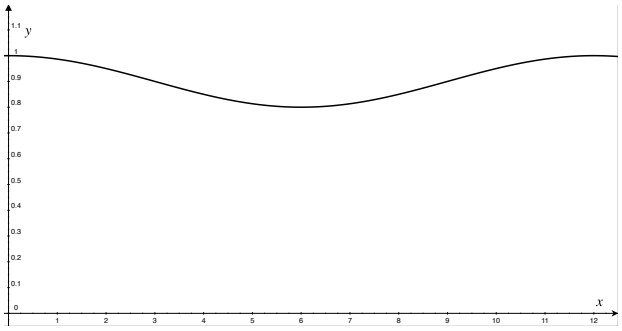

Fig. 18:
$y = \max(\frac{1}{u}(\cos(x\frac{6.283185}{v}) - 1) + 1, 0)$
$u = 10, v = 12$

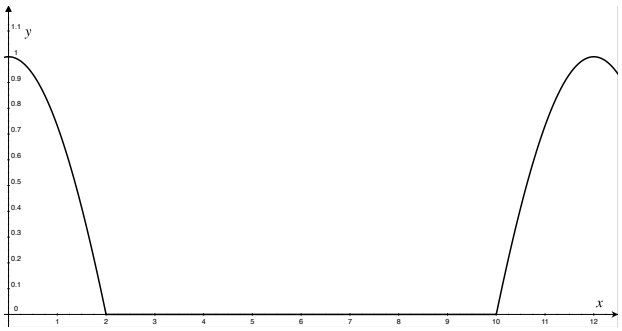

Fig. 19:
$y = \max(\frac{1}{u}(\cos(x\frac{6.283185}{v}) - 1) + 1, 0)$
$u = 0.5, v = 12$

5 Testing EAT

EAT comprises a diverse set of modular tools that all use similar syntax for instantiation and have lucid documentation in the form of two help files per module and a 'README' available to download online; there is little additional instruction for testing that would prove EAT's intended functionality, since the aim of the tools is to fulfil open-ended compositional applications. However, the step-by-step guide below will instruct some basic use of EAT. This guide assumes that you have installed Pd-extended v0.42.5 on your system successfully.

1. Download the current version of EAT for your operating system.
2. Unpack the contents to your Pd plug-ins folder: Pd-extended > Contents > Resources > extra.
3. Run Pd-extended.
4. Identify the EAT folder that contains the plug-ins in the menu under Pd > Preferences > Path.
5. Open a new patch.
6. Save it in a new folder (keeping it in a separate folder keeps all the preset files together).
7. Enable DSP.
8. Create an object box.
9. Create `[EAT_Audiofile~]` using the README arguments in the object box, e.g. `[EAT_Audiofile~ playback]`.
10. Load an audio file in `[EAT_Audiofile~]` by clicking 'OPEN' and selecting a compatible sound file (WAV is fine).
11. Create a `[dac~]` object.
12. Connect the active outlets of `[EAT_Audiofile~]` (denoted by green ticks) to the `[dac~]` object.
13. Click 'PLAY' in `[EAT_Audiofile~]`. You may want to adjust the volume using the slider on the right of the module.
14. Now create `[EAT_Delay~]` – check the README for arguments.
15. Connect it between `[EAT_Audiofile~]` and the `[dac~]` object and experiment with the parameters in the 'ADV' panel.
16. Connect and configure a MIDI controller.
17. Enter the 'MIDI' panel on the front of `[EAT_Delay~]`.
18. Click the 'ML' and 'MC' toggles on next to one of the parameters, then move an appropriate fader, knob, or switch on your MIDI controller.
19. You should then be able to control that parameter with the MIDI controller you just moved and the 'ML' toggle will have automatically cleared.
20. Set some more MIDI control parameters and alter parameter values using MIDI control as in the previous two steps.
21. Create `[EAT_Transport]` or click the link at the bottom of any EAT MIDI window. Try saving some parameter and MIDI presets.
22. Alter your MIDI control and parameter settings again. Save another preset in `[EAT_Transport]` and try toggling between them using the 'LOAD' buttons.
23. Create some other EAT modules, described in the README, and experiment by altering their positions and parameters. Try saving some more presets and explore some more MIDI control options.

6 Further Research

This project has made practical headway into the provision of accessible and efficient DSP modules in Pure Data. Although readily extensible, the modules would benefit from more homogenous architectures of nested abstractions and argument inheritance. Further attempts should be made to extract the current patcher name to customise preset file names and reduce hardcoding within send and receive names. Similarly, the help documentation, despite being reasonably comprehensive in terms of content, could benefit from restructuring according to self-modifying rules and methods. This would make the help documentation easily imitable and bring it in line with the desirable convention proposed by Mathieu Bouchard [14]. Following this core framework development, the EAT modules should be expanded using the *PdLive!* abstractions as a foundation.

The EAT modules could be adapted and optimised for inclusion in Aleš Černý's Visual Tracker application. Visual Tracker is constructed with excellent sequencing and storage architectures, but this should not become the sole vehicle for EAT. Visual Tracker is arguably not as flexible as EAT for general patching, state saving, or provision of HID control, and it would not encourage users of higher level sequencers (such as Ardour or Audacity) to engage with Pd conventions or to learn DSP techniques.

Future projects could investigate the potential of Processing to provide an efficient OSC interface for EAT. Alternatively, new native interfaces could be developed for EAT modules using advanced graphics to mask GUI objects. The discourse on dynamic patching natively and using externals would benefit from a formal study of the effects of dynamic object management on DSP consistency and CPU/GPU resources.

7 Acknowledgements

My thanks go to James Mooney and David G. Thomas to both of whom I am indebted for their generous advice and support during this project. I would also like to thank Thomas Grill, Hans-Christoph Steiner, Thomas Musil, and Miller Puckette whose work has greatly benefitted this project.

References

[1] Thomas, R., 'Electro-Acoustic Tools (EAT): High-Level Abstractions for Audio Manipulation and Spatialisation', in *Proceedings of the 4th International Pd Convention*, 2011, pp. 31–35.

[2] Hernandez, R., 'Pure Data' [Playlist], *YouTube*, 2011. Available: http://youtube.com/user/cheetomoskeeto

[3] Holzer, D. et al, *Pure Data*, 2011. Available: http://en.flossmanuals.net/pure-data

[4] Kreidler, J., *Loadbang*, trans. by Mark Barden. Hofheim: Wolke Verlag, 2009. Available: http://pd-tutorial.com

[5] Farnell, A., *Designing Sound*. Cambridge, MA: MIT, 2009. Excerpt available: http://aspress.co.uk/ds/pdf/pd_intro.pdf

[6] Recoules, B., *PdLive!*, 2011. Available: http://code.google.com/p/pdlive/

[7] Černý, A., 'VisualTracker – Modular Pd Environment for sequencing events on the timeline', in *Proceedings of the 4th International Pd Convention*, 2011, pp. 123–131.

[8] Kong-King, S. and Chin-Chiuan, L., 'Effects of Screen Type, Ambient Illumination, and Color Combination on VDT Visual Performance and Subjective Preference', in *International Journal of Industrial Ergonomics* 26(5), Elsevier, 2000, pp. 527–536.

[9] Galitz, W., *The Essential Guide to User Interface Design*, 2nd edn. New York: Wiley, 2002.

[10] Puckette, M., *Pd Documentation*, 2011. Available: http://crca.ucsd.edu/∼msp/

[11] Grill, T., 'Dyn: Dynamic Object Management', in *Proceedings of the 1st International Pd Convention*, 2004. Available: http://download.puredata.info/convention04/

[12] Hynds, G., 'Dyn∼ Object for Pure Data'. Available: http://konkanok.com/page/10/

[13] Mooney, J. and Moore, D., 'Resound: A Design-Led Approach to the Problem of Live Multi-Loudspeaker Sound Spatialisation', in *Proceedings of the RMA Annual Conference*, 2008. Available: http://james-mooney.co.uk

[14] Bouchard, M., 'Self-Modifying Help Patches', in *Proceedings of the 4th International Pd Convention*, 2011, pp. 26–30.