Reflection in Pure Data

IOhannes m zmölnig
Institute of electronic Music and Acoustics,
Inffeldgasse 10,
8010 Graz,
Austria,
zmoelnig@iem.at

Abstract

One of the more prominent realtime computer music languages on linux is *Pure Data*. While *Pure data* provides ample constructs for signal domain specific programming, it has only limited capabilities for metaprogramming. In this paper we present *iemguts*, a collection of objects that add (amongst other things) reflection capabilities to Pure Data.

Keywords

Metaprogramming, Pure data, agents

1 Introduction

Pure Data is a programming language for realtime signal processing and computer music. While it can be regarded as a domain specific language to accomplish the task of handling and manipulating streams of numbers, an important aspect is the twodimensional diagrammatic representation of algorithms. As Mathieu Bouchard has put it: "The Diagram is the Program".[1]

Due to the representational nature of Pd, it is well suited for direct presentation to the audience [1], e.g. by means of code projection during Live Coding performances: the graphical environment offers access for a lay audience by means of reading (or at least: something to look at).

If a graphical patch cannot be read as source code (due to lack of programming knowledge), what remains is a set of labeled rectangles, which are interconnected by lines. (The interconnections becomes obvious in Live Coding performances, when connected objects are moved around with the connecting lines sticking to the inlets respective outlets of the rectangles).

Assuming that an audience can understand, that the "rectangles" are supposed to represent "entities that do something" (e.g. processes), one can describe the entire environment as an *agent*-based system.

[1]This is the motto of the Pd-addicted *#dataflow* IRC-channel at FreeNode

An *agent* as descibed in [2] is "situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future." Usually Pd-objects do not meet this criterion.

Nevertheless, it does not take so much, to make a Pd-object aware of its environment in order to be able to react on it. Unfortunately, plain Pd offers only limited functionality to accomplish even this.

What is needed to add "self-awareness" to a programm (e.g. a Pd-patch) is called *reflection* in computer sciences. According to Demers and Malenfant, computational reflection is "the process of reasoning about and/or acting upon oneself" [3], with this "oneself" being the program.

This is closely related to what is called *dynamic patching* within the Pd-community, generating (parts of) Pd-patches from within Pd.

Metaprogramming/Reflection offers ways to inspect the current state of a patch (from an interpreters point of view) as well as it can ease the process of writing patches, e.g. by allowing to use "macros", that automatically create or insert snippets of code without having to manually patch them.

In section 2 of this paper we will have a closer look at what plain Pd offers for this kind of programming, and what are the shortcomings of the available mechanisms. We will then present *iemguts*, a library that expands the powers of Reflections, in section 3.

2 Metaprogramming in Pd

Reflection can be split into two different parts: *self-examination* and *self-modification*. While Pd has some powerful abilities to modify the running patch (with a focus on *adding* to the patch rather than *changing* existing parts of it), there are only a few objects that allow a patch to examine its own runtime behaviour: [realtime] (and the related [cputime]). This object measures te time elapsed between two [bang(s, and can therefore be used for profiling cpu-hungry objects (see Fig.1).

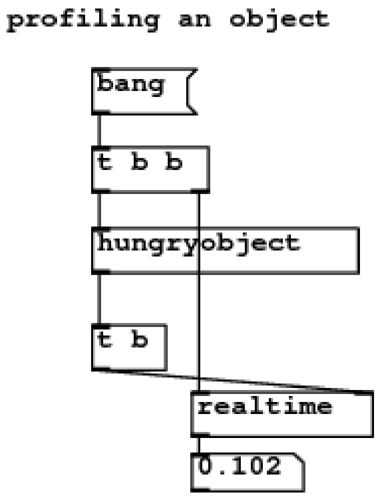

Figure 1: Profiling the [hungryobject] which takes approx. 0.1ms to execute

Due to the realtime nature of Pd, the main application for this kind of self-examination lies probably within the realms of debugging and optimizing the patch manually, rather than changing the behaviour of the patch during execution.

2.1 Dynamic patching

A more powerful kind of meta-programming is available via "dynamic patching".

Pd itself builds up patches (e.g. on loading), by sending messages to the current canvas. E.g. the code snippet in Listing 1 is taken from a Pd patch containing a [bang(message connected to a [print] object.

Listing 1: Pd-patch excerpt

```
#X msg 53 92 bang;
#X obj 53 113 print;
#X connect 0 0 1 0;
```

On loading, Pd will send each line of the patch-file to the current patch (which is implictely bound to the receive-label #X). It is thus equivalent to the patch shown in Fig.2, where the messages get sent to the subpatch that is implicitely bound to the receive-label pd-sub.

A good introduction to dynamic patching is given in the *pd-msg* tutorial [4]

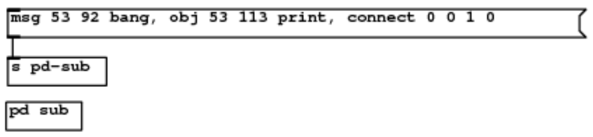

Figure 2: Simple self-creating patch

2.2 The woes of metaprogramming in Pd

Apart from being able to create patches on the fly, dynamic patching can also simulate user-interaction, by sending special messages for mouse-movements and keyboard events to a canvas.

However, this has some severe drawbacks: for one thing, such things can only be done on visible (that is: opened) canvases. Additionally, programmatically fuddling with the user-interaction interferes heavily with normal (non-programmatical) user-interaction, esp. when the former does not happen in zero logical time but in a scheduled way. E.g. the mouse-pointer will move eratically, which makes live-patching unseemly hard.

Another major drawback is, that messages like connect work on patch-local object indices, rather than on object-labels. E.g. "connect 0 0 2 0" means "connect the 1st outlet of the 1st object to the 1st inlet of the 3rd object".

This only works if the indices are known beforehand. As soon as the user starts creating their own objects in the canvas in concurrency to the dynamic patching system, the indices become unknown to the system, thus breaking connections.

3 iemguts: Design and conventions

iemguts is a library designed to expose internal (on the C-code) information and methods at the patch-level. For instance it helps to keep track of dynamically changing object indices.

In a first implementation, abstractions are given the possibility to find out information about themselves, rather than about other objects living in the same instance of Pd, following a $self-like convention.

The reasoning behind this is, that if only abstractions are considered, it is sufficient to have some sort of self-awareness. Information about neighbouring abstractions can be acquired by implementing a *query-response* system that allows absractions to exchange information about themselves.

The scope of reflection is therefore limited to the abstraction, or the canvas containing the abstraction (the canvas being the "environment" of the abstraction, which might be of interest to the object). In order to be able to encapsulate refelction-logic into "sub-abstractions" (abstractions within the reflected abstraction), the scope can also be shifted up towards the top-level patch.

By *canvas* we mean what is represented visually as a "window", no matter whether this window is an abstraction, a sub-patch or either of them contained in one of the two.

To sum up, the scope of reflection is always restricted to a certain depth within the patch hierarchy, and to the specified canvas (with other canvases at the same depth being out of scope).

A few objects do not comply with this convention, having a global scope. This is due to the global property of the content they access. For instance, since Pd loads classes into a global scope, an object that queries the existance of a class within Pd operates on the same scope level.

With respect to (semi-)automatically interaction it was important to allow to write agent-like objects that do not need a supervisor in order to interact.

3.1 Space-awareness

Since Pd is a graphical language, the most important property of an object is probably its position within a canvas. This property is usually not used within Pd's syntax, with position having no meaning to the structure of a patch (with the noteable exception of `[inlet]`s and `[outlet]`s). Nevertheless it is one of the most obvious things when merely looking at a patch. In order to be able to use this property, the object `[canvasposition]` will return the current position of the object within its containing canvas.

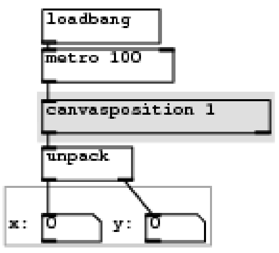

Figure 3: querying the position of the abstraction within its canvas 10 times per second

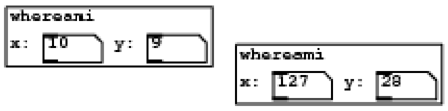

Figure 4: GOP-abstraction of Fig.3 in action

The user can thus change the behaviour of an object by dragging it around the canvas. (For a simple example see Fig.3 and 4)

For systems that want to use more autonomous agents, it is not only possible to *get* the current position but also to *set* it, effectively allowing the object to move around by itself.

3.2 Modes of interaction

Objects can interact in several ways: either by using explicit *connect*ions, or by connecting to an (invisible) bus transporting messages via `[send]`/`[receive]`.

In Live Coding contexts, the send/receive method can be suboptimal, as it can easily be overlooked, especially when considering objects that can automatically start communicating with each other if some condition is met (e.g. the two objects are within a certain minimal distance). In this case, the explicit communication via connections is more readable.

However, in order to allow connection management by a patch itself, it has to gain some knowledge about the objects to connect: their object indices and which iolets are actually available.

`[canvasindex]` will return the object's own index within its parent patch, while `[canvasconnections]` will give information about the available iolets.

Since we always need two objects in order to successfully connect, this information is not fully sufficient. The proposed solution is to use a send/receive based query-system in order to find out these properties of a partner object.[2]

Once connections have been made, `[canvasconnections]` can also be used to query which (other) object connects to a certain iolet.

3.3 Sharing a local namespace

Pd does not have a local namespace for labels, like send/receive-names. However, it provides

[2]One could see this as an implicit *subconscious* communication between the two objects.

a mechanism for creating pseudo-local namespaces, using the unique (per abstraction) variable `$0`. Hierarchies of patches can share this unique value by passing it as an argument. In a Live Coding situation this is sub-optimal since the performer has to remember to add the `$0` to all relevant objects, and – more important – the audience (that is probably not literate with respect to Pure data) has to decode yet another awkward language construct in order to understand better what is going on. Therefore several people have implemented objects that can access the value of `$0` of the parent canvas without having to explicitly pass as an argument.

For the sake of simplicity (and to avoid dependencies), *iemguts* includes its own object `[canvasdollarzero]` which accomplishes this task.

3.4 Reproduction

A simple form of cloning an object in Pd can be considered creating an object of the same class (that is: name) and with the same state (that is: arguments).

However, more often than not the internal state of an object will be modified at runtime (through external messages): in this case the given arguments do not reflect the internal state any more. Creating another instance of the class with the same given arguments will re-create the object in its initial state rather than its current state.

In order to make cloning of objects use the current state, one can utilize Pd's way of object duplication. Whenever an object get's dumped to a file (e.g. when the patch is saved), or copied to the clipboard (when the user pressed `Ctrl/Apple-C`) or when an object get's duplicated (by pressing `Ctrl/Apple-D`), Pd will query the object for a "save-string". If we can dynamically modify this save-string, we can use it to reflect the current state of the object, automatically making copies of the object to be clones (at the time of duplication).

This functionality is provided by `[canvasargs]`.

A similar object is `[canvasname]` which can be used to both query and modify an abstraction's name. In conjunction with patch-saving this allows to build a simplistic version-control system for patches: whenever an abstraction is duplicated a new branch is created from the current state.

3.5 Self destruction

While Pd has built-in capabilities for creating patches, it completely lacks of programmatical ways to destroy parts of these patches. The only way to remove something from a patch is to completely `clear` a canvas, which is rather coarse in terms of granularity. An especially annoying bug within the implementation of Pd will make the interpreter crash if an objects triggers the clearance of a canvas that contains this very object.

To avoid such crashes and to allow an abstraction to safely remove itself from the patch, the `[canvasdelete]` object can be used. Banging the `[canvasdelete]` object will simply delete the containing object, allowing for objects that have only a limited time of existance.

3.6 Anybody here?

A more theoretical object is `[classtest]`, an object that tests whether Pd knows about a certain class. E.g. it can be used to test whether "xyz" names an objectclass in Pd, simply by sending the symbol "xyz" to the `[classtest]`-object: it will output "1" if the given symbol names a registered class and "0" otherwise.

This information can then be used e.g. to dynamically build up a patch depending on whether a certain (external) object is present or not.

This object is noteably different from the ones introduced before, as it is not concerned with the state of the abstraction it is living in, but rather with the overall state of the Pd-interpreter.

4 Future works

The current implementation is focused on self-awareness of objects, and does not give access to other objects. While this is no problem when dealing only with abstractions (where reflection capabilities can easily be added using the *iemguts*-objects), it does not allow the same level of access to unmodifiable objects (e.g. internals, like `[metro]`.

It is thus planned to add a second class of objects that work exclusively on properties of other objects within a canvas, referenced by their object index. For this to work properly an additional object for finding an object within the canvas by its class.

5 Conclusions

We presented *iemguts*, a library for Pure data that adds the possibility of metaprogramming techniques like reflection to abstractions. This can be used to enhance the patching workflow by implementing patching-helpers like macros or to create self-aware intelligent agents like systems, that interact with each other in ways currently not possible.

The main focus during the development of objects has been on Live Coding environments, wherein this library has been successfully used for various performances.

References

[1] IOhannes m zmölnig and Gerhard Eckel. Live coding: an overview. In *Proc. of the ICMC*, Kopenhagen, 2007.

[2] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*. Springer, 1996.

[3] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proc. of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, page 29–38, 1995.

[4] Damien Henry. pd-msg tutorial, 2002.