Self-replication: how to do `more` using `less`

Krzysztof Czaja
The Fryderyk Chopin University of Music
Okólnik 2
Warsaw, Poland
czaja@chopin.edu.pl

Abstract

Multi-instantiation is the process of taking a single declaration of an object, or a patch, and creating several instances of it, *replicating* the structure, while possibly varying the initial state. An important application, among many others, is supporting the implementation of polyphonic instruments.

There are two distinct design options: the process of replication may be initiated and controlled either from the outside, or from the inside of the replicated patch. The latter possibility is explored in this paper in an attempt to advocate for *self-replication* as a conceptually simple, yet quite generic and powerful mechanism.

Keywords

Multi-instantiation, self-replication, polyphony.

1 The `poly~`, its clones, and the `more` to come

One of the few remaining PureData vices, which still compromise its many virtues [1], is the lack of implicit multi-instantiation mechanism. For example, according to the common expectation, any implementation of a sound synthesis algorithm should qualify for seamless abstraction into a polyphonic instrument. Pd has not yet fulfilled this expectation, and a path of improvement is proposed in this paper.

A canonical example of a working solution exists in the Max/MSP world. A specially programmed abstraction may be replicated by passing its name as a creation argument to an object `poly~` [2]. The interface used for controlling the instances of an abstraction is quite complex, and MIDI-centric. The replicated patch should contain `poly~`-specific variants of inlets and outlets, which handle interfacing with the outside patch. The communication between instances and the object `poly~` itself is passed through an object `thispoly~` contained in the replicated patch.

The number of instances is initially specified by an argument to the object `poly~`, and it may be changed at any time by sending the message `voices` to this object. Directed at specific instances are ints, floats, the bang, and lists. The message `target` is used for redirection.

Selective parametrization of instances is not supported — all instances are created with the same set of parameter values, which are specified at the end of an argument list given to the object `poly~`. All data shared by the instances have to be stored outside of the replicated patch. In-place editing of the patch is not possible, although viewing its contents is.

At least two attempts to imitate selected features of `poly~` in a Pd external have failed. Although performing quite well, they turned out to be too limited to be useful in practice. The only `poly~`-like implicit replication mechanism available today, is quite convoluted and volatile, unfortunately. It is based on clever dynamic patching tricks, which misuse a set of messages originally designed for the manual editing of patches — a temporary solution [3].

If a similar attempt is to be successful, some changes to Pd core will be necessary. These changes need not be guided, however, by the goal of borrowing an existing design. They are, instead, an opportunity to extend the internal mechanism of instantiating Pd objects and patches.

Instantiation-time evaluation is an integral part of Pd, although currently, it is limited to a simple form of parametrization: arguments of an object, before being passed to the constructor, are first evaluated in the context of parameters of the containing patch. The Pd core evaluates these arguments using literal values of patch parameters, and taking literal values of the result of textual substitution. The patch parameters, in turn, are the values given as arguments to an object which instantiated the patch.

One can imagine several ways of extending the user-guided computation performed during the in-

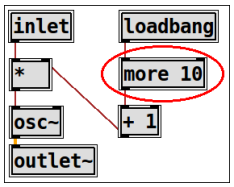
stantiation phase. Design for metaprogramming, however, is a much broader topic than this paper's focus. The scope of this proposal is limited to the process of instantiating replicated copies of a patch — its embedded instances. The idea is to provide two special objects, `more` and `less`, each having a very simple interface designed to control the way embedded instances are created, and how they operate. A prototype implementation is being developed in a branch of a clone of the Pd vanilla repository [4].

2 Motivating examples

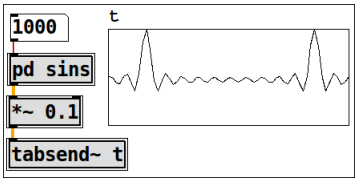
From a user's perspective, the task of introducing implicit multi-instantiation into a project is threefold. First, the user creates an object, an abstraction or a subpatch, and prepares it for replication. Then, the user has to trigger, somehow, the transformation of the object into an array of embedded instances. The last part is interfacing the instances with the project as a whole. A few simplified examples are presented in this section in order to give a taste of working with embedded instances.

2.1 Ten `sins`

The subpatch `sins` shown below contains the object `more 10`, which requests for nine more instances of the subpatch.



Seen from the outside, this subpatch behaves like an ordinary signal generator. It produces a harmonic series of ten sinusoidal partials, where the fundamental frequency is controlled by messages sent to the object's only inlet:



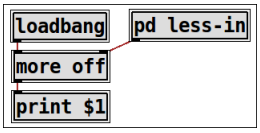
What happens under the hood, however, is that the constructor of the object `more` sends a replication request to its enclosing subpatch. Later on, when the instantiation of the `sins` subpatch finalizes, the Pd core will fulfill the request and create nine extra *embedded instances* of the subpatch. These instances will share inlets and outlets with the main, *surface instance* created, as usual, for the `pd sins` object. The signal outlet of `pd sins` will output the sum of signal streams coming from all ten instances.

The extra instances cannot be opened into a window for editing or viewing of their contents. They are embedded one after the other, forming the *embedding chain*. The position of an instance in the embedding chain, the *embedding index*, may be queried: the object `more`, when banged, will output the value of embedding index of its containing instance, which equals 0 for the surface instance, 1 for the first embedded instance, etc.

2.2 Seven `vices`

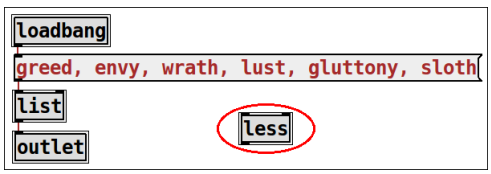
The next example introduces the `more`'s companion object — `less`. The interaction of these two objects facilitates the dynamic creation and reconfiguration of embedded instances. Moreover, the object `less` is used for redirecting specific messages to specific instances — after specializing the object `more` for detachment of embedded instances from surface input.

This time, the replicated patch will be an abstraction, instead of a subpatch. The following definition can be saved in a file `vices.pd`, and instantiated as an object `vices pride`.



The argument `off` given to the object `more` is a request for the *detached mode* of embedding. Since there is no numerical argument, the number of embedded instances is not specified statically — they will be dynamically requested, instead.

The subpatch `less-in` contains the object `less`, without arguments and not connected:



The purpose of this object is to declare, that the subpatch `less-in` is to be *shared* between all instances of the abstraction `vices`. In other words, this is a *singleton* subpatch — the `loadbang` fires only once, and the right inlet of the object `more` receives just the six symbols, no more.

In general, if a replicated patch contains an immediate subpatch, which in turn contains an object `less`, then the subpatch is shared. (It is not necessarily a global singleton, however, because the replicated patch may belong to another patch, also replicated, thus forming another level of replication hierarchy.)

The `loadbang`s of shared patches *fire early*. This timing is an important feature, because the shared patch should be fully initialized before instantiation of the enclosing patch ends, and before any actual embedding starts.

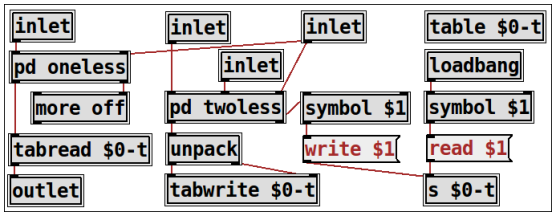
In the abstraction shown above, the six symbols sent to the right inlet of the object `more` are the replication requests provided in place of the missing numerical argument of the object. These requests schedule the creation of six embedded instances. Each symbol is substituted for the first parameter of a consecutive instance.

If the abstraction is created as an object `vices pride`, the resulting printout will read: `pride 0`, followed by `greed 1`, `envy 2`, etc., down to `sloth 6`.

2.3 Keeping `vices` under control

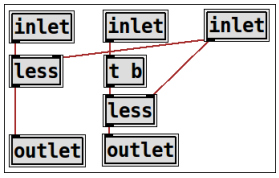
The initially very simple example described above will now be extended into a version illustrating the mechanism of *redirection* of messages to specific instances. It also shows the usefulness of dynamic replication requests. (This does not mean that the abstraction is already useful as such; it is just a skeleton, which may be further developed.)

Each instance of the abstraction `vices` contains its own localized array created by the object `table`. These arrays may hold, for example: presets, values specific to sections of a piece, envelopes of synthesized partials, spectral profiles, etc. The abstraction handles reading from or writing to arrays, and loading or saving them to disk:



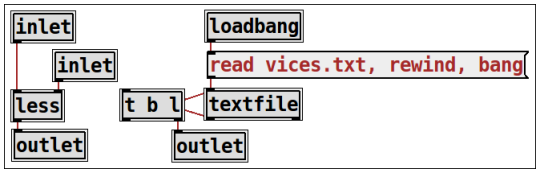
The left inlet of the abstraction passes control messages on their way to the `tabread` object, and the second inlet does the same for the `tabwrite`. The third inlet triggers file saving, and the last one redirects control to specific instances.

It is only the surface instance which originally receives all writing and saving requests, due to the detached mode of embedding. The requests must then be redirected to specific instances, which is the task of the subpatch `twoless`:



A float message sent to the right inlet of the object `less` specifies the embedding index of a target instance. Any message sent to the left inlet of the object `less` is redirected to the current target.

Similarly, the subpatch `oneless` redirects reading requests:



This subpatch contains also the initialization part, which controls self-replication of the parent abstraction. Instead of hard-coding replication requests in a message box, the list of requests is loaded from a text file.

3 Detailed exposition

This paper advocates for nothing more, than extending Pd with two internal object classes, `more` and `less`. The interface of the two objects is deliberately designed to be as simple as possible. As a consequence, this exposition will be kept short.

3.1 The `more` inside

The purpose of the object class `more` is to initiate and control self-replication of a patch it is part of, which may be a toplevel patch, a subpatch, or an abstraction instance containing the object. The object `more` has always two inlets, one outlet, and accepts a single number and/or a single symbol as optional arguments.

The numerical argument N is a request for *no less* than $N-1$ embedded instances. Thus, if there is more than one object `more` in a patch, all numerical arguments will be ignored, except the largest one. The largest N is not necessarily the final number of created instances, since additional instances may be requested dynamically.

Symbolic arguments are reserved for the control of *embedding modes*. Currently, the only valid value is `off`, which is a request for the *detached mode* of embedding. Thus, if a patch contains objects `more`, and any of these objects is given the argument `off`, the embedded instances of the patch will have message inlets disabled.

The left inlet of `more` accepts any message. The object first converts the received message to a list, if necessary, then precedes it by the embedding index, and sends the result through the single outlet. This behaviour is strictly equivalent to passing that same message to an object `list prepend` *`id`*, where *id* is the embedding index. In particular, sending a `bang` to an object `more` forces the object to output the embedding index as a float message.

The right inlet is used for dynamic self-replication and reconfiguration. Each data message (i.e. one of: bang, float, symbol, and list messages), when sent to the right inlet during instantiation of a patch, is a request for one more embedded instance. The message arguments are passed to the requested instance as its list of creation arguments.

3.2 The `less` of early banging

The object class `less` is introduced for two reasons. First, it declares that its enclosing subpatch is shared between all instances of the parent patch — the one directly containing the shared subpatch. Second, it manages redirection of messages to specific instances of the replicated parent patch.

The redirection works as if the shared subpatch migrated from one instance of its enclosing patch to another. The procedure may be compared to rebinding a variable to a different value. The instance currently containing the shared subpatch is initially a free variable. This variable is bound to the source instance of any message arriving to any inlet of the subpatch. The binding occurs at the time when an inlet receives a message, i.e. at the point of calling one of the inlet's methods. The variable is unbound as soon as the message has been processed, i.e. before the method returns.

The currently bound instance is the destination of messages passed to any outlet of the subpatch. Calling the subpatch via an internal timer, or remotely, through a receive, results in the variable remaining unbound, and the messages thus triggered will not leave the subpatch — unless remotely, or after redirection.

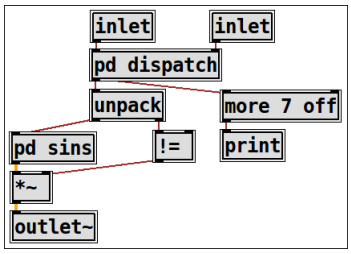
Performing a redirection may thus be understood as rebinding the variable to another instance. The new target instance is determined by the last object `less` that the message has been piped through. The right inlet of the object sets the target, and the left inlet accepts any message, which is passed, unchanged, to the object's single outlet.

The object `less` takes a single optional float argument, which initializes the target instance.

The insertion of an object `less` in a subpatch has a side effect of "early firing", which means, that if such a subpatch contains any loadbang objects, they fire *before* the replication of a parent patch starts — not after, as is otherwise the case.

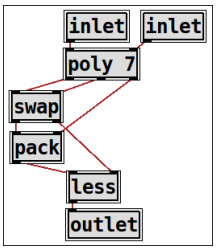
3.3 Seven voices

To conclude the exposition, let's consider the prototypical application of self-replicating patches — managing of polyphony. As an example, a two-level hierarchy of embedded instances will be created, which may serve as a skeleton of an additive synthesizer. The outer patch defines a voice, and contains an object `more`, which requests for seven instances:



At the inner level, there is the subpatch `sins`, which was described in the section 2.1. Since that subpatch replicates itself into ten instances, and there are seven voices, the total number of created partials is 70.

The input to the outer patch are note-like messages of pitch and "velocity". The outer patch passes them to its another subpatch, `dispatch`, which "allocates" voices in the usual manner.



This dispatching scheme is a clear example of redirection: the messages sent originally to the surface

instance of the outer patch are redirected to a chosen instance taken from a pool of voices.

As a natural extension, this simplistic skeleton of an instrument may be merged with the abstraction `vices` described in the section 2.3, where the arrays would define envelope profiles of separate partials, or groups of partials.

4 Implementation

The prototype implementation plugs into the Pd core without breaking its monolithic, single-threaded DSP graph design. The DSP part is somewhat convoluted due to optimizations, which already exist in the `d_ugen` and `g_io` code ("borrowing" of signal vectors). The actual embedding is performed during execution of the function `canvas_restore`. Handling of message inlets is straightforward, and the overhead is one extra test required by a potential need for passing a message to embedded inlets. The changes to message outlets induce the same overhead, although some extra code in the constructor has to be executed for shared subpatches in preparation for redirection. (The costs of message handling may be avoided by implicitly specializing the class of inlets and outlets, whenever a patch is to be replicated.)

The delicate part of the current implementation is the way a singleton subpatch is transformed into a "stub" in embedded instances. During replication of a patch, its immediate subpatches are scanned for instances of the object `less`. If such a subpatch is found, all its contents is deleted, except the inlets, the outlets and the `less` itself.

5 That's it, more or less...

There are some controversial issues originating from the proposed design. The most urgent one is the need to fully understand the consequences of choosing self-replication over a sub-replicating, `poly~`-like solution. Another example of an open question is the difference in treatment of surface instance and embedded instances. Any design decision, however, is to be approved, improved or rejected only in confrontation with real-world projects, and these are yet to be created.

References

[1] M.S. Puckette. *The Theory and Technique of Electronic Music*. World Scientific Press, Singapore, 2007.

[2] `poly~` *Reference*, in: *Max 5 Help and Documentation*. `http://cycling74.com/docs/max5/refpages/msp-ref/poly~.html`.

[3] *PureData repository*. `http://pure-data.svn.sourceforge.net`.

[4] *Branch 'moreless' of the PureData repository*. `http://github.com/k7f/PureData`.