Pure riddle

Krzysztof Czaja
Chopin Academy of Music
Warsaw, Poland
czaja@chopin.edu.pl

ABSTRACT

A small set of extensions to the PureData architecture and to its external API is proposed, which will facilitate the creation of audio feature detectors (pitch followers, beat trackers, onset detectors, etc.) as Pd patches. An experimental implementation is discussed, and two application examples are demonstrated.

1. INTRODUCTION

Most audio feature detectors used in modular environments are defined as black boxes. Various implementations of pitch followers, beat trackers, onset detectors, etc. are available in Pd, however none has been coded in Pd.

The advantage of black-box implementations is that they are usually better optimized and easier to port between different modular environments. The advantage of white-box implementations, apart from their accessibility for study to a wider public, is that they can be made more flexible, easier to adjust to changing needs, and their parts may be reused in different contexts. Although white-box implementations are typically less optimized, the overall performance of an application patch may in some cases be better tuned by reusing common parts of white-box detectors, than by putting several black-box detectors to work on the same signal, since computation effort is quite likely to be wasted in the latter case.

Unfortunately, defining feature detectors in terms of simple, generic building blocks is hard. Doing so usually requires different computation block sizes to coexist in a single patching window or even in signal I/O of a single object. Typical vector sizes are other than power-of-two.

One solution to this problem is moving all the nonstandard signal communication between objects to asynchronous domain. This is further discussed in section 6. A synchronous solution is the main topic of this paper.

2. APPROACH

At the core of such approach is purely synchronous flow of data of arbitrary size. In order to achieve this, Pd has to be extended one way or another. Such an extension, the *riddle extension*, is presented below primarily as a proposal to slightly modify the internal implementation of Pd and to add a small number of calls to its external API. On the other hand, this extension has already been implemented in the form of a fully usable external library — *riddle library*. The implementation of riddle library is likely to be much different from any internal version that would ever be developed, but there are very few differences between the API provided by riddle library and the API proposed for riddle extension. Therefore, in this paper both versions of the API are referred to by the same term, *riddle API*. Finally, a *riddle object* is any object that depends on riddle API.

All riddle objects in a window maintain a constant rate of calculations. This is the nominal rate for that window, i.e. derived from a parent window or set by a `block~` or a `switch~` object. However, the connections between riddle objects may carry variable amount of data. The proposal distinguishes two cases

- a single vector of constant size;
- a collection of dynamically-sized vectors.

Both cases are supported by the library version, provided data size never exceeds the nominal block size of the window.

The vector size, in the first case, is determined during the creation of the dsp chain, and does not change until the next time dsp chain is recalculated. The second case requires that the actual data is being passed accompanied with extra elements determining vector sizes, while the data layout may only be defined during dsp chain creation.

In this scenario, destination objects have to be informed about the kind of data they are receiving. Riddle library provides a way for source objects to announce what they are going to transmit, and a way for destination objects to obtain that information. It also provides a way to assert that destination objects receive what they want, and to disable them in case of failure.

All announcements, queries and assertions are performed during dsp chain recalculation phase.

3. API

The proposed extension allows riddle objects to selectively *redeclare* some or all of their signal outputs. In order to do so, a class should define a special method, `dspblock`, which is to be called by the dsp graph sorting procedure[1], prior to the regular call of the object's `dsp` method.

The task of the `dspblock` routine is to process the *signal resolution slots* which contain riddle declarations, i.e. vector sizes or data layout of the object's signal ports: inlets, outlets, and remote connections, where the latter may be direct or buffered through arrays, delay lines, etc. Specifically, the `dspblock` method is only invoked when there is any change of vector size or data layout in input slots, or of the block size or sampling rate of the containing patch

[1]The name of this procedure in the current Pd sources is `ugen_done_graph`.

(at least, that is the case in the current version of riddle library). The routine should read the input slots, and fill in the output slots. When it is done, the graph sorting procedure validates and transfers vector sizes and data layout from current object's output slots to the input slots of all locally and remotely connected objects. If there is any serious failure during this process, the object is disabled and its signal outlets are muted. Otherwise, the usual `dsp` method is invoked.

The main source of failures are incompatible connections between objects. Any possible incompatibility should be checked for explicitly in `dspblock` method of a destination object — provided the destination is actually a riddle object. If a connection is from a redeclared outlet to a signal inlet of a regular Pd object, i.e. an object for which the `dspblock` method has not been defined, the possible failure is determined implicitly by the dsp chain sorting procedure depending on

- the "strictness flag" which the source object may set for any of its outlets: a strict connection to a non-riddle object disables the source;
- the class of destination object: for example, a connection to `print~` is always accepted, and a connection to `send~` or `throw~` is always rejected[2].

Generally, connections from riddle objects to non-riddle objects are safe, because all blocks of signal data that are used in the same patch window are allocated the same amount of memory space, regardless of their interpretation as regular vectors or according to a riddle declaration[3].

Riddle library accepts connections to subpatch inlets and to `outlet~` objects, and recursively traverses such connections through to the final destinations. The internal implementation, however, should accept only those connections that do not involve reblocking. Doing so in riddle library would probably be the most tricky part of the implementation, and it was not attempted.

Pd symbols are used to specify data layout in riddle declarations. They are called *patterns* and have to be in the following *normal form*: each element of a pattern is either, the next lower-case letter not used so far in the pattern, or an upper-case letter, such that the corresponding lower-case letter is included in the preceding part of the pattern. The starting letter `'a'` may be omitted. Any pattern that does not conform to these rules is rejected. In a pattern, upper-case letters represent vectors, and lower-case letters represent vector sizes.

The proposed extension to the Pd API consists of ten calls, which are designed to be used in the `dspblock` routine:

```
int riddle_getsourceblock(t_object *, int siginno)
t_symbol *riddle_getsourcelayout(t_object *, int siginno,
                                 int *maxblkp)
int riddle_getsourceflags(t_object *, int siginno)
void riddle_setoutblock(t_object *, int sigoutno, int newblk)
void riddle_setoutlayout(t_object *, int sigoutno,
                         t_symbol *pattern, int maxblk)
void riddle_setoutflags(t_object *, int sigoutno, int flags)
int riddle_checksourceblock(t_object *, int siginno, int reqblk)
int riddle_checksourcelayout(t_object *, int siginno,
                             t_symbol *reqpattern, int *maxblkp)
int riddle_isdisabled(t_object *)
void riddle_disable(t_object *)
```

[2]This restriction may be lifted in future library versions.

[3]Or, in case of the internal implementation, at least the amount required by the nominal block size of the window.

The calls: `riddle_disable`, `riddle_getsourceblock`, `riddle_getsourcelayout`, and `riddle_getsourceflags` may be used in the object's `dsp` routine as well as in the `dspblock` routine.

A riddle object may declare that the total data size never exceeds a certain value, by passing a positive `maxblk` argument to the `riddle_setoutlayout` call. The value of that argument is truncated to the nominal block size of a window, which is also the default value used when the argument's value is zero. The `riddle_setoutblock` call performs the same truncation. This is a serious limitation of the library version of riddle extension, which should not by imposed by the internal version.

A support for remote connections should be an intrinsic part of riddle extension. The corresponding API calls are not listed in this paper. Some further details are discussed in section 4.

4. IMPLEMENTATION

The experimental implementation takes form of an external library[4]. The library consists of two parts. The *riddle core* is a set of routines emulating the proposed extensions to the object class definition and dsp graph sorting mechanism. A collection of externals, each linked to the riddle core, contains various reusable building blocks aimed at feature detection applications. In an attempt to minimize confusion, the names of all riddle externals start with the prefix `rd.`, as in `rd.erb~`. The best way to use the library is to load a "hook external", `rd`, during the Pd startup phase, e.g. by using the option `-lib rd`. Loading specific riddle externals on demand is also possible.

Neither the dsp graph sorting procedure, nor the Pd object definition cannot be modified by the external library. Therefore, riddle library provides three wrapping routines that replace the usual object constructor, destructor, and `dsp` methods. A wrapping routine does the extra work common to all riddle objects, and calls the class-specific routine (the `dsp` method actually calls two routines: `dspblock` and `dsp`). The class-specific routines are to be implemented in the usual way. They have to be registered in the class setup routine, and the special call `riddle_setup` should be used instead of the usual `class_new` for that purpose. This call should be removed from the internal implementation.

An important peculiarity of riddle library, which is likely to disappear in the internal implementation, is the way in which riddle declarations are propagated to input slots of destination objects. For reasons that are beyond the scope of this paper it may be safely achieved only by sending special `"_reblock"` messages to destination objects. The `_reblock` method is implicitly declared for all riddle classes by the `riddle_setup` routine.

Riddle library introduces a common infrastructure for remote communication between riddle objects[5]. It may be used to build both, unbuffered connections similar to standard `send~`/`receive~` networks, and buffered connections in the form of arrays, delay lines, etc. The main entity used in both cases is *riddle buffer*, which may be owned by a single riddle object. A riddle buffer is indirectly accessible to other

[4]It may be downloaded from the `miXed/riddle` directory of the SourceForge repository for Pd externals.

[5]This part of the library is still very experimental.

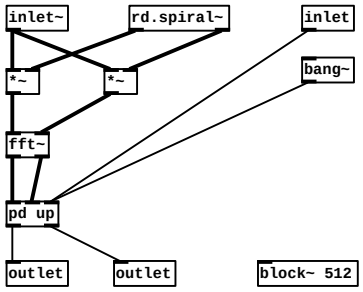
riddle objects through separate, albeit identically labelled riddle buffers. For each label, there is at most one riddle buffer with writing permissions to buffered data, and several readers (riddle library does not support accumulating buffers yet). Current implementation maintains compatibility between interconnected riddle buffers, and keeps track of their lifetime.

For each riddle buffer there is a separate *remote slot* assigned to an owning riddle object. Remote slots are processed similarly to the local input and output slots of riddle objects.

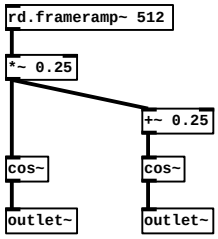
All remote connections potentially introduce the writing/reading order ambiguity. This is particularly unfortunate when a connection is redeclared, since a remote slot has to be filled by the source before it may be queried by the destination. If any destination riddle object precedes its remote source in the dsp chain, the current solution schedules a second pass of the dsp chain creation.

5. EXAMPLES

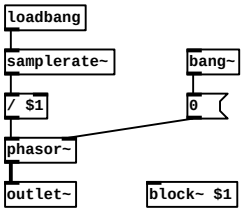
Two simple application examples show how a white-box feature detector may be built as a synchronous network. They might also help to gather some insight into performance differences between white-box and black-box implementations. The first example is Miller Puckette's `fiddle~` pitch tracker [4] dissected into small pieces:



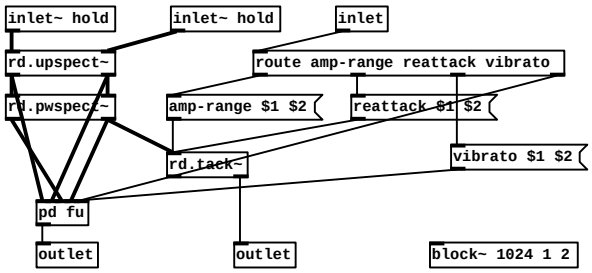
where `rd.spiral~` is an abstraction:



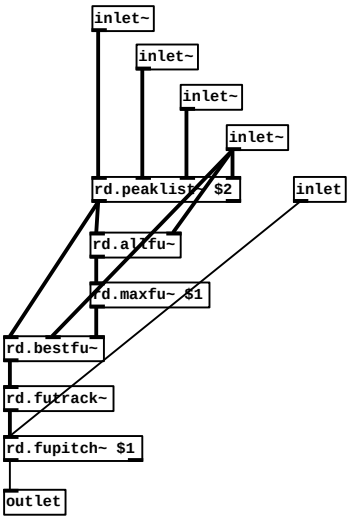
and so is `rd.frameramp~`:



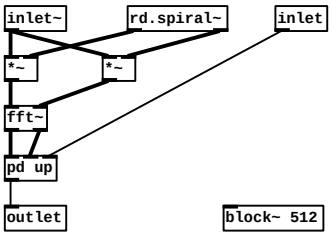
The `up` subpatch is a network of riddle objects:



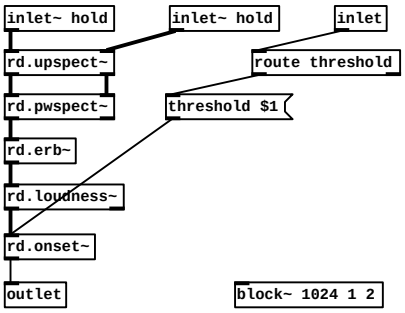
and so is the `fu` subpatch:



The second example is based on the Klapuri-Collins onset detector [1]. The main window is quite similar to the one from the previous example:



There are similarities in the `up` subpatch too:



It is obvious, that these two detectors may easily be combined into a couple, so that when they are set to work on a common input signal, most of the calculation effort will be shared.

6. OTHER APPROACHES

A different approach to white-box implementation of feature detectors is based on performing a signal-to-message conversion at an early stage, and keeping the main body of calculations in the asynchronous domain. Many such "event-driven" solutions are possible. The simplest one is to directly pack signal vectors into messages, similarly to the way the `iemmatrix` library does it with matrices. This solution suffers from the obvious performance penalty.

There are also many reference-based ways of passing data around in patching environments, including the Pd native data structures [2]. Unfortunately, Pd data facility is not yet ready for the task (some of the reasons are presented in [3]).

FTM [5] is particularly interesting as a basis for development of white-box feature detectors, especially, if extended with the Gabor library [6]. The main focus of Gabor library is granular synthesis and transformation, where the natural computation rates are variable: random, pitch-synchronous, etc. Nonetheless, effectively synchronous implementations of analysis algorithms may be driven by Gabor's constant-rate signal-to-event transformers, like the `gbr.slice~` module.

No Pd version of FTM exists yet. Porting has been initiated, but it is still not clear how smoothly would FTM eventually fit into the Pd message system and its atom typing rules.

7. CONCLUSIONS

Riddle extension has a clearly defined goal of supporting the implementation of white-box, purely synchronous signal feature detectors. The price of achieving this goal is performance downgrading in relation to corresponding black-box implementations, when compared one-to-one in simple test patches. From a set of very informal performance measurements that were carried out so far it follows, that the black-box `fiddle~` requires approximately 75% of calculation time spent in running its white-box version.

Event-driven solutions are certainly more general and flexible than any purely synchronous solution can be. They tend to be much more complex as well. Nonetheless, it is a bad idea to argue about coding style without taking into account the requirements of a particular application. It is impossible to discuss personal habits and taste. The best way, instead, is to provide several options, and a purely synchronous one should be among them.

8. REFERENCES

[1] N. Collins. A comparison of sound onset detection algorithms with emphasis on psychoacoustically motivated detection functions. In *118th Convention of Audio Engineering Society*, Barcelona, Spain, May 2005.

[2] M. S. Puckette. Using pd as a score language. In *International Computer Music Conference (ICMC)*, pages 184–187, Göteborg, Sweden, 2002.

[3] M. S. Puckette. A divide between 'compositional' and 'performative' aspects of pd. In *First Internation Pd Convention*, Graz, Austria, 2004.

[4] M. S. Puckette and T. Apel. Real-time audio analysis tools for pd and msp. In *International Computer Music Conference (ICMC)*, pages 109–112, 1998.

[5] N. Schnell, R. Borghesi, D. Schwarz, F. Bevilacqua, and R. Müller. Ftm — complex data structures for max. In *International Computer Music Conference (ICMC)*, Barcelona, Spain, September 2005.

[6] N. Schnell and D. Schwarz. Gabor, multi-representation real-time analysis/synthesis. In *COST-G6 Conference on Digital Audio Effects (DAFx)*, pages 122–126, Madrid, Spain, September 2005.