

pnpd/nova, a new audio synthesis engine with a dataflow language

Tim BLECHMANN

Vienna, Austria

tim@klingt.org

Abstract

pnpd/nova is a new dataflow-based computer music system. Its syntax shares a common subset with max-like languages like Pd or Max/MSP, but introduces some new concepts to the dataflow language, most notably an extended and extendable message type system, data encapsulation and namespaces. Currently, it doesn't provide a graphic user interface, but contains a compiler for a text-based patcher language and a command-line interpreter, which can be controlled via OSC.

The audio synthesis engine is designed for supporting low latencies and is optimized for high performance.

Keywords

audio synthesis, computer music system, dataflow language

1 Introduction & Motivation

Max-like dataflow languages have been used in computer music systems since the 1980s. Currently Max/MSP¹ and Pd² are the most commonly used programs implementing the max paradigm[3].

Although pnpd/nova is heavily influenced by Max/MSP and Pd, it is not just a rewrite of one of these programs in C++. The dataflow language of pnpd/nova contains language features like namespaces or hierarchical object bindings, that work quite different in other max-like language and should make pnpd/nova much better suited for more complex applications.

2 Type System

2.1 Built-in Types

Like other max-like languages, pnpd/nova has a strong separation between signals and messages. The messaging happens synchronous with the audio signal, to be able to schedule

events very tightly, unless it is explicitly detached to low-priority threads by the used objects. The pnpd/nova language contains the following atom types:

bang a simple function call, the atom representation of a bang is **#None**

float a double-precision floating point number

integer an exact integer number³

symbol a reference counted, hashed string, to be used as message selector

string a string class (based on the string class from stl)

atomlist a list, containing zero or more atoms

pointer pointer to a user-defined type

2.2 Extending Types

Using the **pointer** type, it is easy to extend the type system by adding custom message types from the C++ interface. Developers of externals can define their own custom message type by simply deriving their classes from the **CustomMessageType** class. All calls to pointer methods use the same inlet handler, the dispatching is done by using the runtime type information of C++. Custom messages are passed by reference. Currently the OSC implementation is based on this feature.

3 Patcher Language

3.1 Text-Based Patcher Syntax

The text-based syntax was designed to describe the patcher language in a human-readable⁴ way. It can be used to write patches until there is a graphical patcher available. Listing 1 shows a simple 'hello world' patch.

¹<http://www.cycling74.com/products/maxmsp>

²<http://puredata.info/>

³based on the GNU Multi-Precision Library <http://www.swox.com/gmp/>

⁴in contrary to the internally used xml format

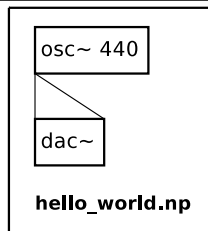
Listing 1 simple “hello world” patch

```
{
  signal = osc~<440>()
  dac~(signal[0], signal[0])
}
```

It creates a patch, containing a sine-wave oscillator with a frequency of 440 Hz with the symbolic name “signal”, that’s first outlet is connected to the first two inlets of the `dac~` class, that represents the audio output. The curly brackets `{}` define canvases, round brackets `object()` define object creations, angle brackets `object<args>()` define creation arguments and square brackets `[]` select an outlet of an object.

Connections can be defined implicitly with a comma-separated list in the round brackets as done in Listing 1 or explicitly. The construction `->` declares connections, if inlets or outlets are not specified explicitly, the first outlet is used for the connection. This is shown in Listing 2

Figure 1 hello world patch (block diagram)



Listing 2 hello “world patch” with explicit connections

```
{
  signal = osc~<440>()
  out = dac~()
  signal -> out
  signal -> out[1]
}
```

Both pieces of code represent the same block diagram, that is shown in Figure 1. The written patch files have to be converted to the internal xml file format, which is can be loaded into the interpreter.

3.2 Semantics

Data Encapsulation & Patch Lifetime

pnpd/nova patches are used as independent classes, that should be reusable in different envi-

ronments. After a patch has been created and it is inserted into the dataflow interpreter, all objects execute their loadbang functions. It is not allowed to insert information into the patch, before this has been done, otherwise it could be used before its initialization is complete, what would obviously lead to severe problems. If there is the attempt of sending information to the patch, before the loadbangs has been executed (via inlets or message busses), the message is queued and run after the loadbangs have been executed. Before a patch is destroyed, endbang functions are executed so that cleanup handlers can be called. At that time, no information is allowed to get into the patch, so calls to inlet functions and message busses will be ignored.

Object Bindings

In Max/MSP and Pd languages, a global scope is used to control the access to objects by symbolic names. If local objects have to be used per-instance, the only workaround would be to add a new unique symbol to the global scope⁵. pnpd/nova provides a mechanism to avoid that. Bindable objects are not directly bound to the symbols, but each patch contains a container for bindable objects. They can be declared explicitly on a certain scope, or implicitly on the topmost point in the hierarchy, where they are visible in every patch. A `declare` object can be used to declare global or local objects.

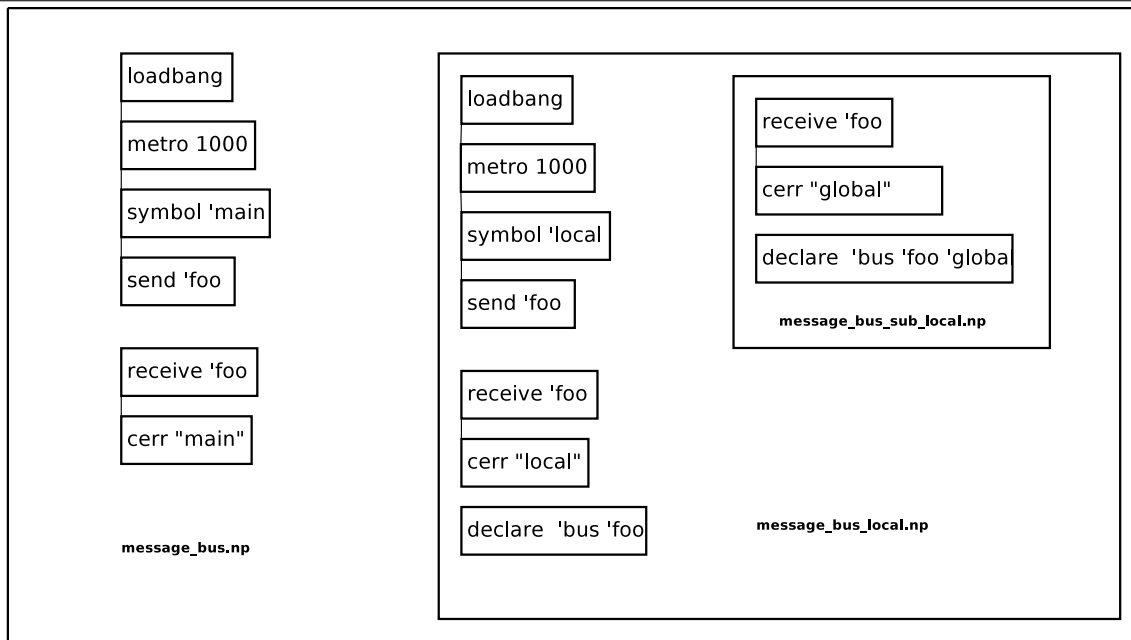
Listings 3, 4 and 5 demonstrate this. In the main patch ‘message_bus’ (Listing 3), a bus named ‘foo’ is implicitly constructed in the current scope. In the subpatch ‘message_bus_local’ (Listing 4) a local bus named ‘foo’ is explicitly declared, thus the `send` and `receive` objects of this patch are bound to the local bus. The patch ‘message_bus_sub_local’ (Listing 5), which is lower in the hierarchy than ‘message_bus_local’, declares a global bus, which is used in this scope. The global bus is located on the top of the hierarchy, where the implicitly declared bus from the top of the hierarchy is then getting rebound to. The block diagram representation is shown in Figure 2.

Namespaces

pnpd/nova features namespaces, which are inspired by the use of namespaces in python. If an object `foo.bar` is created, the interpreter searches for abstractions and externals in the

⁵in Pd this is done with `$0`, in Max/MSP with `#0` prefixes

Figure 2 message.bus patch (block diagram)



Listing 3 message_bus.np

```
{
    metro = metro<1000>(loadbang())
    send<foo>(symbol<'main'>(metro))
    cerr<"main">(receive<foo>())
    message_bus_local()
}
```

Listing 4 message_bus_local.np

```
{
    declare<'bus 'foo'>()
    metro = metro<1000>(loadbang())
    send<foo>(symbol<'local'>(metro))
    cerr<"local">(receive<foo>())
    message_bus_sub_global()
}
```

subfolder `foo` with the name `bar`, and for the external library with the name `foo`, containing the object `bar`. It will be searched relative to the path of the root patch and in user-defined search paths. If different objects are found in more than one place, loading the object fails, to avoid undefined behavior in the case of name clashes.

In future a `using` object will allow to search a certain namespace by default.

Listing 5 message_bus_sub_local.np

```
{
    declare<'bus 'foo 'global'>()
    cerr<"sub_global">(receive<foo>())
}
```

4 Some Implementation Details

pnpd/nova is completely written in C++, with the exception of the parser for the text-based patcher, which is written in python. Portaudio⁶ is used for audio i/o. Most of the internal thread communication is making use of lockfree C++ data structures. Most system operations are run in low-priority threads in order to make it possible to run pnpd/nova with as little as possible audio dropouts even during times of high cpu load.

4.1 Boost

pnpd/nova relies heavily on the boost C++ libraries⁷, as they provide very powerful, portable and stable implementations for many aspects, the C++ standard doesn't deal with. Furthermore, several boost libraries are going to be part of the future C++ standard.

It simplified the memory-management of heap-allocated objects, which is now using smart pointers in certain areas, the spirit parser frame-

⁶<http://www.portaudio.com/>

⁷<http://boost.org/>

work made it much easier to write a decent parser for the string representation of atoms. The boost graph library turn out to be a great help for representing the dsp graph and creating the dsp chain. But also for network i/o (OSC via asio), filesystem traversal, the python bindings and several other parts it simplified the code a lot.

4.2 DSP

The implementation of the dsp core is similar to Pd[2], providing means to suspend parts of the dsp graph in order to save cpu power, or to reblock and overlap specific areas, as it is required for certain applications like windowed spectral processing. As in Pd, reblocking, overlapping and muting is directly bound to the canvases, where the dsp objects are located.

But unlike Pd, it is using nested dsp chains. If a canvas contains a `switch~` or `reblock~` object, a separate dsp chain is used for this and its child canvas. Changes to the dsp graph trigger a generation of a new dsp graph in a background thread. When the new dsp chains are ready, the root chain is exchanged with the obsolete chain. Since the sorting of the dsp graph is done asynchronously, it is possible to do changes to the patches or even load patches without audio dropouts.

The ugens are not necessarily bound to graphical objects. Dsp objects can either provide the dsp function as member function or allocate a special ugen class, depending on the dsp context or the state of the dsp graph. This way, objects can implement several ugens and allocate the most efficient one.

4.3 Performance Notes

pnpd/nova is optimized for high performance, making heavy use of the sse instruction set on modern cpus of the x86 and x86_64 architectures. Although recent compilers are able to vectorize certain code, it is usually not possible to generate optimal code, e.g. because of aliasing issues or the requirement to write algorithms differently⁸. In addition to that, it is using compile-time loop unrolling for performance-critical parts, implemented using C++ template metaprogramming techniques. The results of a general-purpose benchmark⁹ against Pd-0.40 with oprofile showed 11537 samples (200000

CPU_CLK_UNHALT events per sample) compared to 27065 samples with Pd.

4.4 Portability

At the moment the only supported platform is linux. However, all dependencies are either platform independent C or C++ libraries or they support linux, osx and win32 and there are plans to provide binaries for other operating system than linux. The sse code is completely separated and plain C++ equivalents exist to assure portability to other architectures than x86. The lockfree algorithms are implemented using `atomic_ops`¹⁰, which provides a wrapper for the assembler code of the used atomic primitives for several compilers and architectures.

4.5 Extending pnpd/nova

The public C++ api can be used to write externals in C++. Externals are shared libraries that are dynamically loaded at runtime. External developers simply have to derive their external classes from the `GObj` or `GObj_dsp` base classes. The C++ api should be reasonably stable, although the binary compatibility might not be guaranteed, because pnpd/nova is using lots of header implemented inline and template functions.

Beside the C++ api, it is possible to extend pnpd/nova with python. With the `py` class it is possible to run python functions and `pyx` can load python classes, which implement messaging externals.¹¹

5 Todo List

pnpd/nova is already usable, but a lot of stuff needs to be done

- graphical user interface (possibly based on `gtkmm/gnomecanvasmm` or `PyQt4`)
- extend the object library, lots of objects are still missing
- better documentation of both the patcher language and the library
- testing, testing, testing ...

⁸more details can be found here [1]

⁹fm synth, delayline, filtered noise, sampling objects, signal i/o. testing system: pentium-m 750, 1024MB, linux 2.6.20-rt1

¹⁰http://www.hpl.hp.com/research/linux/atomic_ops/

¹¹the interface is inspired by Thomas Grill's `py/pyext` objects for Pd and Max/MSP, <http://grrrr.org/ext/py/>

6 Conclusions

Although pnpd/nova is still in an early state of development, it is already quite usable. I have been using it in concerts since the late 2006. Lots of features are still missing, especially a gui client with a graphical patcher. However, it already prove to run very efficiently on modern hardware, supporting lowest latencies. We will see, if it is able to compete with Max/MSP or Pd, since both programs have a huge user base. Nevertheless, for users of these programs, switching should be very easy. For now, the most important thing beside manpower for writing a gui, is to find some users, who are willing to do beta testing.

7 Acknowledgements

I'd like to thank Miller Puckette for Pd and for making it open source, because i learned a lot when reading the Pd sources and it inspired pnpd/nova's design and dieb13 for hosting the project at klingt.org.

References

- [1] T. Blechmann. Simd in dsp algorithmen. <https://tim.klingt.org/pnpd/Members/tim/iem.pdf>.
- [2] M. Puckette. Pure data: another integrated computer music environment. In *Proc. the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.
- [3] Miller Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.