# Rage Against The Glue: Beyond Run-Time Media Frameworks with Modern C++

Jean-Michaël Celerier

# Rage Against The Glue:
# Beyond Run-Time Media Frameworks with Modern C++

**Jean-Michaël Celerier**
celtera.dev
contact@celtera.dev

## ABSTRACT

*We identify a set of issues with the current abstraction systems for media objects and introduce a methodology to solve these issues, associated with a sample implementation in the Avendish library. This methodology is based on the limited reflection features available in recent C++ versions, unlike the existing systems which are overwhelmingly based on class-based inheritance or other run-time affordances of the language.*

*We propose using a simple subset of the C++ object model to define media processor's metadata and interface declaratively: this subset can be reflected in order to generate the binding code to various plug-in interfaces such as VST, the Max or Python API, and automatically generate UI code or OSC APIs at compile-time.*

*Unlike existing systems and frameworks, our proposed method has the advantage of being non-intrusive. The media processors do not need to inherit from existing base classes or be part of a framework: they can be written without even having to include any specific header.*

## 1. INTRODUCTION

We aim to devise the most adequate abstraction for defining and implementing *media processors* (**MP**) in C++: real-time or offline audio effects and instruments, message-based objects for use in patching systems such as Max/MSP and PureData or in object-oriented programming languages such as Python, visual processing objects for machine-learning pipelines. Depending on the host environment (**HE**) or API used, those are called *nodes*, *unit generators / ugens*, *objects*, *boxes*, *plug-ins*...

Using reflection to extract information from the implementation of a MP has multiple advantages over existing systems. First, the object implementation does not need to depend on an existing library. It is possible with this method to write an audio processor with controls, metadata such as ranges and names, which does not include any header or inherit from any class. The code only needs to implement the right types, in a structural sub-typing sense. This is future-proof: today, if an existing API is abandoned by its developers, algorithms written against this API have to migrate to another, as seen for the complicated VST2 to VST3 transition. In our case, the code does not depend on such an API ; the processor code never interacts directly with the binding code. It will be possible *a posteriori* to write a binding to newer APIs without requiring any code change. Second, there is no run-time cost: the binding process can be done entirely at compile-time unlike existing solutions such as Juce[1]. Third, multiple implementation strategies can easily be tried. One could compare atomic variables for controls and message-passing through lock-free queues for the usual problem of communication between an audio processor and a user interface.

This paper will cover the issues with existing binding systems for MPs, introduce the C++ features which enables defining and reflecting MPs in a declarative fashion, give an overview of the proposed way of writing MPs and introduce a few additional possibilities of this system.



### 1.1 Open issues in media processor APIs

We cover succinctly in this section various undesirable properties of the current APIs and systems for implementing MPs in C++.

#### *1.1.1 The quadratic glue problem*

This is also known as the $M \times N$ problem [2]. In the field of intermedia creation, it manifests itself as $M$ MPs (distortion, chorus, OSC message sender, pitch detection, MIDI arpeggiator…) and $N$ HEs (Max/MSP, PureData, ossia score [3], vvvv, OpenFrameworks, Python, Unity3D, Unreal Engine, Essentia, VAMP, PiPo, digital audio workstations (DAWs)…). It is meaningful for almost every such algorithm to be present in almost every such environment. This requires writing $M \times N$ glue code between the MP algorithms and the HE data structures.

Abstractions exist to allow MPs to support multiple environments. Writing against the VST API enables an audio algorithm to work in many DAWs. Writing against the well-known Juce API allows code to be abstracted over multiple DAW APIs (VST, AAX, LV2, etc.), but there are two costs to this: the runtime performance cost due to going through run-time abstractions and parameter systems, and the limited expressive power: parameters must conform to the API defined by Juce, which is biased towards audio processors. Juce does not allow the creation of a Max object that processes messages or a Python class with multiple methods for instance. Other *meta-plugin-APIs*, such as iPlug2[4] and DPF [1] , behave similarly.

[1] https://github.com/DISTRHO/DPF

### 1.1.2 Example of the problem

Consider the task of declaring a user-facing control for a MP: the cut-off frequency of a low-pass filter. Examples taken from various extensible software environments are shown in Table 1: most of them additionally require to inherit from some base class.

In all these examples, the low-pass code can be implemented with a similar expression once the parameters' values have been retrieved from the parameter system. Yet, it is necessary to write as many low-pass implementations as there are environments. In addition, APIs which abstract over HEs, such as iPlug2, Jamoma, and Juce, may store some meta-data at run-time even for APIs which would be unable to leverage them, which wastes memory. This causes *frameworkization*: a MP author must ask *"Am I writing against JUCE, iPlug2, DPF, VST3 directly ?"*. Likewise, a MP implemented in Bespoke, cannot be trivially ported to VCVRack even when a large amount of algorithms is theoretically portable to both.

Other approaches rely on custom pre-processors using C++ comments as in RATL [5] or generation of MPs defined in another language, such as in UGG [6]. In the first case, the approach lacks the static checking opportunities offered by the C++ type system, and require the meta-processor to be able to parse enough C++ to understand the comments [2] . In the second case, the interoperability with separate libraries is harder, as the generator only supports a predefined set of operations ; mainly arithmetic on numbers. In contrast, the proposed approach only depends on native C++ constructs: everything is done through language features, no separate system is involved.

### 1.1.3 Parameter addressing and safety

The existing abstractions generally do not offer convincing safety mechanisms, or coerce everything into the lowest common denominator, such as single or double-precision floating-point value. This is caused by run-time parameter abstractions, evidenced in Table 1: when retrieving the value of a parameter, it is not possible to simply access a value of the correct type, which is hidden by the abstraction. Instead, the developer of the MP has to query the parameter system for this value. This querying mechanism can be done in various ways:

- Index-based queries: PureData, Max/MSP, VST2. Error-prone: the developer has to remember to which numeric index a parameter is assigned and either switch on it or go fetch it in an array. Adding a new parameter at the beginning may require the developer to recompute all the indices in the MP codebase.
- String-based queries: `getParameter("freq")`. Used in Essentia. Error-prone: since string content is not part of the type-system, the compiler cannot warn that `"Freq"` has been used instead of `"freq"` which leads to hard-to-debug issues.
- Enum-based queries: a layer of safety above indices, which are replaced by enumerations which can be named: `enum Param { kCutoff, kGain }`. Used in VCVRack, iPlug2 for instance.

A better technique, found in Bespoke, is to separate the parameter abstraction and its value, by passing a pointer to the value to the parameter abstraction, which will be tasked of querying and updating it. This way, the processing code can directly use the parameter variable without going through the abstraction layer required by the metadata system.

### 1.1.4 Cache optimality, abstraction cost

Another issue directly related to this parameter metadata system, is the locality of the storage of parameters. In systems such as Juce, the parameter values are stored within objects, which can be dynamically allocated: in terms of memory layout, the controls may be far apart in memory. This means that instead of needing to fetch one cache line if eight parameters were laid out contiguously, the CPU may have to perform eight different memory fetches, assuming a 64-byte cache line and a MP with eight double-precision parameters. This can have a performance impact especially at smaller buffer sizes where the parameters may have to be fetched again at a high frequency.

Additionally, all the present abstractions' overhead is per instance, even when the metadata of each parameter is known to be constant across all instances of a given MP. That is, when loading a bank of 64 filters, the description of the filter's parameters will be duplicated at least 64 times in memory. A better system ought to store a single copy of the metadata for the whole program, and not per MP instance, to facilitate large-scale systems with thousands of objects.

Finally, these systems are less amenable to compiler optimizations, due to the indirection barrier between parameters. In the cases where one would perform a compile-time combination of multiple MPs, such as building a synthesizer out of an oscillator, a filter, and an amplification stage, the compiler is less likely to be able to optimize computations across modules.

### 1.1.5 A note on other languages

We do not cover non-C++ systems and DSLs such as Faust[12], SOUL [5] , Vult [6] …. They are mainly concerned about defining DSPs. In contrast, using a general-purpose language allows a wider range of MPs to be designed: event-driven message processors, image processors, network communication systems, etc. The system should allow the user to implement a 2D image filter that would be useable in drawing software as well as in Max/MSP, without having to write glue code.

Staying in C++ enables the algorithm authors to benefit from the ecosystem of libraries and APIs. Most of the systems mentioned in Table 1 are implemented in C++, and most of the libraries for performing media-related processing or data communication offer a C or C++ API: interacting with hardware such as LeapMotion, Kinect, Wiimote, etc. Systems based on separate languages generally need a small layer of C or C++ glue to implement e.g. OSC communication: one cannot easily write OSC network code directly in a Faust, SOUL or Vult object as these language do not expose [7] socket primitives. This implies the existence of bindings akin to Faust architecture files [13]: in our case, as C++ code which reflects MPs.

[2] Consider for instance `printf("this␣\"co␣//␣de\"");` as an example which requires non-trivial work to parse correctly

[5] `https://soul.dev`

[6] `https://www.vult-dsp.com`

[7] As far as the author could find.

| System | Code example: declaring a new parameter |
|---|---|
| Bespoke[7] | `new FloatSlider(this, "freq", 5, 4, 120, 15, &mFreq, 20, 2000)` |
| Essentia[8] | `declareParameter("cutoff", "frequency␣[Hz]", "(0,inf)", 1500.)` |
| iPlug2[4] | `GetParam(kFreq)->InitDouble("Freq", 0., 0., 100.0, 0.01, "Hz")` |
| PiPo[9] | `PiPoScalarAttr<double> factor{this, "freq", "Frequency", false, 1.0}` |
| Jamoma[10] | `addAttributeWithSetter(freq, kTypeFloat64)`<br>`addAttributeProperty(freq, range, TTValue(2.0, sr*0.475))`<br>`addAttributeProperty(freq, description, TT("Cutoff"))` |
| Jamoma2 | `Parameter<double, Limit::None<double>, NativeUnit::Hz> frequency`<br>`= { this, "frequency", 1000.0, ... };` |
| Juce[1] | `addParameter(freq = new juce::AudioParameterFloat("freq", "Freq", 0.0f, 1.0f, 0.5f))` |
| LV2[11] | Declared in a Turtle file defining the processor metadata |
| Max / PureData | No parameter type, run-time mechanism generally based on switch/case |
| VCVRack [3] | `configParam(FREQ_PARAM, -54.f, 54.f, 0.f, "Frequency", "␣Hz", ...)` |
| VST2 [4] | No parameter type, run-time mechanism based on queries from the host |
| VST3 | `parameters.addParameter(new ScaledParameter ("Freq", "Hz", 0, 1000, ...))` |

**Table 1**: Declaring a frequency parameter with metadata such as minimum and maximum value, unit, textual description, etc. in various environments.

### 1.2 Our goals

We aim to provide a C++ methodology and system to alleviate the aforementioned issues:

- End the *frameworkization*: it should be possible to implement a MP solely through language primitives, without requiring specific types coming from a third-party library. This improves composability, optimizability, and compatibility: a MP which does not use any library but is implemented simply as a C++ struct with appropriately-named members is trivial to compile and run on even very small embedded targets (such as microcontrollers), uncommon targets (WebAssembly, eBPF) or to port to a new host, unlike a MP written with Juce.
- Zero-cost design with a strict "do not pay for what you do not use" policy: unused metadata should be able to be entirely optimized-out by the compiler. The system should allow inter-procedural optimizations among MPs to the limit of what C++ compilers are able to perform. Binding to an existing plug-in API should have no run-time over-head over implementing a plug-in directly against this API in the most efficient way.
- Safe design where parameters are defined in a non-redundant, integrated way, without per-instance cost for the metadata: what we want is the processing code to be able to refer to simple C++ variables directly for accessing parameters, so that the C++ language can type-check them to reduce the opportunity for run-time errors especially when adding and removing parameters in MPs. These variables should at the same time carry the metadata (description, range, unit...), without having to duplicate them for each instance of the MP. We do not want in particular a system where one would have to repeat the declaration of a parameter or split it in multiple places (by defining for instance enumerations). We do not want to require the use macros either due to their notorious error-proneness.
- Extensible design: an HE with specific port types, such as geometry data for 3D software, should be able to add support for them.

## 2. AVENDISH: LIBRARY DESIGN PROCESS

The goals exposed in section 1.2 are mostly achievable thanks to various features added to the C++ programming language over its last evolutions. Even if the language does not support an explicit "reflection" feature as in Java, Python and others, we can combine enough existing features of the language to approximate reflection to a point which is enough for implementing MPs succeeding in these goals.

An example implementation, which is able to target VST, VST3, Max/MSP, PureData, Python, create an OSC API and create a Qt GUI has been developed by the author and is available on GitHub [8] under a free license.

An important point of this implementation is that MPs can be written without having any explicit dependency on the library: technically, even existing code can be reinterpreted as being an Avendish-compatible MP, without requiring manual intervention. Since some tasks are verbose, a helper library nonetheless provides valid implementation of common use-cases. However, the author encourages organizations to develop their own helper libraries with the coding style and conventions they are most used to.

### 2.1 The C++ subset we are using

Note that by subset, we do not mean that users of the API are restricted from using parts of the C++ language for their implementation ; only that this work does not imply that every C++ construct is used. For instance, dynamic polymorphism through inheritance is not processed in any way as it does not add any meaningful information for what we wish to do. Most run-time environments do not even have a way to make use of it: a polymorphic VST control does not make sense.

The main restrictions are that reflected types must be *aggregates* in C++ parlance. Simply put, types which do not have explicit constructors, private or protected members, or virtual functions. Every `struct` example throughout this document is an *aggregate*. Templates cannot easily be

[8] Available at `https://github.com/celtera/avendish`. A complete documentation is provided, and explains how to author a MP from scratch: `https://celtera.github.io/avendish`.

mapped either as they exist at a level of abstraction above the object model we are working with ; one cannot reflect a template, only a particular instantiation of it.

### 2.2 Reflection features in modern C++

We present to the reader the features introduced in recent C++ standards which we are able to use to implement the MP bindings.

#### *2.2.1 Reflection on struct members with Boost.PFR*

The core technique making this project possible is implemented within the `Boost.PFR`[9] library. It enables compile-time iteration of the members of an aggregate. The structure `Controls` below is an example of aggregate: `Boost.PFR` permits iteration and subsequent reflection of the members `a` , `b`, `c` in order. The implementation leverages destructuring, a C++17 feature.

```
struct Controls {
  std::string a;
  float b;
  int c;
};
```

#### *2.2.2 Querying existence of members*

Thanks to C++20 concepts[15], it becomes possible to query the following information at compile-time: "Does the type T have a member variable named `foo`?". The `has_a_foo()` function will return a compile-time computed boolean value, true if it is passed as template argument a type with a member variable `foo`, false if not. `requires` yields a boolean: whether T satisfies the specified requirements:

```
template<typename T>
consteval bool has_a_foo()
{ return requires (T t) { t.foo; }; }
```

#### *2.2.3 Querying metadata on members*

It is possible to perform queries on the type of such variables, such as: "Does the type T have a member variable `bar` of type float?".

This is done with concepts and requires-expressions:

```
template<typename T>
consteval bool has_a_float_bar() {
  return requires (T t)
    { { t.bar } -> std::same_as<float>; };
}
```

#### *2.2.4 Reacting to existence of members*

It is possible to implement different behaviours depending on those cases, thanks to the `if constexpr` C++17 feature. Unlike `if`, the condition is guaranteed to be resolved at compile-time and thus bears no cost and does not exist at run-time ; for each template instantiation, either the condition will be true or not true ; each instantiation will only have one of the branches compiled in (or none at all).

[9] `github.com/boostorg/pfr`. A prototype which leverages the variadic destructuring feature: `auto [... members] = a_structure;` proposed in standard paper P1061 [14] allows discarding the Boost dependency entirely. This feature however only works in development versions of C++ compilers so far.

```
template<typename T>
void print_foo_or_bar() {
  if constexpr(has_a_foo<T>())
    print("{}", t.foo);
  else if constexpr(has_a_float_bar<T>())
    print("{}", t.bar);
}
```

This code compiles if `T` is an integer, a character array, or even the following type:

```
struct empty_type { };
```

The implication for our needs is that it becomes possible to reflect object properties at compile-time, without any run-time cost.

#### *2.2.5 A wide API contract*

It is possible to support various variants and coding styles for a given concept. The techniques presented here can be used to make the first part of Jon Postel's statement: "Be liberal in what you accept, and conservative in what you send" applicable to C++ API design. Consider an user-facing description metadata for an object, various organizations may use distinct coding styles:

```
struct object_of_org_A {
  auto desc() { return "Equalizer"; }
};

struct object_of_org_B {
  static void getDesc(std::string& storage)
  { storage = "Equalizer"; }
};
```

Both implementations transmit the same information. We can on the library side write the following function:

```
template<typename T>
string get_description(const T& t) {
  if constexpr(requires { t.desc(); }) {
    return t.desc();
  }
  else if constexpr(requires (string s)
                      { t.getDesc(s); })
  {
    string ret; t.getDesc(ret); return ret;
  } // etc..
}
```

This way both possibilities will be treated in an equally optimal way. Not all cases can be accounted for ; as evidenced in Table 1, it is impossible to get a community of our scale to agree on a single interface. Instead, we should consider making binding libraries which try to accept as many systems and ways of implementing MPs as possible, especially when extending those independent binding libraries is a task as simple as adding a branch to a compile-time `if`.

This approach accommodates both for future MP APIs when they will inevitably come up, and for various ways of designing the actual MPs, and do both these things without having to pay a run-time abstraction cost either in CPU usage or memory, simply through writing or extending independent binding code.

### 2.3 Application to our work

The outlined reflection mechanism allows the author of a binding to check at compile-time whether an object is, for instance:

- Something that will only process messages.
- Something that is a monophonic, per-sample audio processor (and thus can be duplicated to multiple channels), for instance because the structure has a member function of the form: `float operator()(float input)`.
- Something that is a polyphonic audio processor, for instance because the structure has a member function of the form: `void operator()(float** in, float** out, int frames)`, or adequate input / output ports.
- Something that is a video filter, because it has a texture input port and a texture output port.

Defining an ontology for ports is a work-in-progress for which the author believes a collaboration across the computer music community would be fruitful. During the library development process, many different shapes of ports were tried. Here, `my_port` is categorized as a polyphonic audio port, and `another_port` as a sample-based audio port.

```
struct {                    struct {
  doubles** frames;           float sample;
  int channels;             } another_port;
} my_port;
```

Binding code will adapt these recognized forms to what HEs expect. One can write a per-sample audio processor: it is converted to a per-block VST plug-in automatically. The binding code will run for every frame and every channel of the input buffers. Conversely, a per-block definition can be mapped to a per-sample system such as VCVRack, simply by passing a sample count of 1 and allocating 1-sized arrays as automatic storage (i.e. on the stack). This way, an algorithm can be expressed in the way that is the most natural for it, while retaining portability to all existing environments.

A self-contained example of a simple distortion processor is available: `https://github.com/celtera/avendish/blob/main/examples/Raw/Minimal.hpp`. Various other examples showcase the supported features.

The Avendish library is currently able to reflect on the following features of a class:

- Inputs, outputs: either as ports or in simple cases as arguments to an `operator()` function. Recognized ports value types can be `int`, `bool`, `float`, `string` and enumerations, sample-based, channel-based or array-of-channel-based audio ports, MIDI ports ; RGBA texture ports.
- Methods, free functions and lambda functions can be used for messages in Max/MSP and PureData. Their arguments are automatically reflected: `void foo(int f, std::string v)` will display an error in Max and Pd consoles if the messages sent to the object are incompatible with the types. A similar feature is available for initialization of Max/MSP or PureData objects. No explicit type conversion is ever required.
- A preset system to declare built-in presets of input controls is available.

### 2.4 Defining controls

The idea is to make the control meta-data part of the type system. As shown earlier, most systems store the metadata relative to a parameter in the same object than the parameter itself which is bad for cache behaviour and memory use. That is, if our parameter is a float, we want:

```
sizeof(TypeOfParameter) == sizeof(float)
```

without a single wasted or repeated byte. Thus metadatas such as range, unit or descriptions, cannot just be stored as member variable. We are able to leverage our system to put this information as static methods instead: this means that the storage will be allocated not once per instance, but once for the whole program. Static data members are technically possible, but incur additional limitations: there cannot be a static variable in an unnamed class. Functions and methods however are “free” in a storage sense ; besides, it is possible to define structures inside methods which allows to simplify the code. A control, which combines static metadata and per-instance value, can be implemented with:

```
struct {
  struct range
  { float min = -1, max = 1, init = 0.; };
  float value;
};
```

### 2.5 Simplifying controls

If we have many similar controls (some synthesizers can have thousands), we may want to simplify the way they are defined.

Before C++20 this could be done with macros which would expand to code such as in section 2.4, but those come with well-known issues. In C++20, we can instead make them part of the type system:

```
struct range {
  float min{}, max{}, init{};
};

template<range Def>
struct float_control {
  consteval auto range() { return Def; }
  float value;
};

float_control<range{.min = 0, .max = 10}>
  control_1;
```

In all the above cases, we can easily create new control types, with the range metadata embedded in the type itself. This enables performing static, compile-time checks on the control ranges.

## 3. APPLICATIONS AND FUTURE WORK

### 3.1 Thread-safety

MP communication between e.g. processing and GUI threads can be made automatically thread-safe, for instance through atomic variables [10] and lock-free queues. Given a struct, one can synthesize a tuple of equivalent atomic variables at compile-time. This is implemented for the VST2 binding:

```
struct inputs {
  struct { float value; } control1;
  struct { int value; } control2;
};
using controls = to_atomics<inputs>;
// controls will be this type:
// tuple<atomic<float>, atomic<int>>;
```

### 3.2 Large-scale replication of processes with structure-of-arrays transformation

A programming technique originating in video game development consists in laying out each individual data member of a large set of entities (filters, synthesizer voices, etc.) contiguously. This has the benefit of adapting to modern SIMD CPUs, which are able to process up to 16 floating point values with a single instruction. The reflection abilities described here allow to generate such arrays from the structures defined. This enables synthesizers with a high level of parallelism across voices, large filter banks, etc.

### 3.3 Amenability of such processes to automatic compiler optimizations

MPs defined as such can combined simply by wiring the calls manually. Everything is just plain members and member functions: there is no indirection. Given basic nodes implementing fundamental operations ($+$, $\times$), we define a simple dataflow. Testing with recent compilers showed that they can vectorize calls across objects [11] : the entire computation, defined as a set of interacting C++ structures, is recognized as being equivalent to a single vector instruction: the proposed system enables inter-procedural compiler optimization.

## 4. CONCLUSIONS

An empirical observation which underpins this work is that class-based inheritance works for code reuse and interoperability at the organisation level, not at the ecosystem level. Every HE implements custom APIs for defining MPs. VCVRack uses sample-based audio processors, Max/MSP can use either event-based message-passing, and block-based monophonic or polyphonic signal processing, etc. These different needs means that polymorphic interfaces are incompatible ; yet we show that it is now possible to write a system where the canonical, dependency-free version of a MP can be authored, and then bound automatically to various environments with the same run-time cost than if this was done optimally by hand.

The techniques presented here have a wide reach: the author has successfully applied them to the declarative definition of GPU pipelines and user interfaces.

[10] Atomic variables are safe for concurrent access by different threads.

[11] `https://gcc.godbolt.org/z/dKT5cbvcr`

## 5. REFERENCES


[1] Martin Robinson. *Getting started with JUCE*. Packt Publishing Ltd, 2013.

[2] Mike Loukides. “Thinking About Glue”. In: *O’Reilly Radar*. 2021.

[3] Jean-Michaël Celerier et al. “OSSIA: Towards a Unified Interface for Scoring Time and Interaction”. In: *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR)*. Paris, France, 2015.

[4] Oliver Larkin, Alex Harker, and Jari Kleimola. “iPlug 2: Desktop Plug-in Framework Meets Web Audio Modules”. In: *Proceedings of the 4th Web Audio Conference*. Berlin, Germany, 2018.

[5] Karl MacMillan, Michael Droettboom, and Ichiro Fujinaga. “A System to Port Unit Generators Between Audio DSP Systems.” In: *Proceedings of the International Computer Music Conference (ICMC)*. Havana, Cuba, 2001.

[6] Roger B Dannenberg. “UGG: a Unit Generator Generator”. In: *Proceedings of the International Computer Music Conference (ICMC)*. Daegu, South Korea, 2018.

[7] Benedict Gaster and Ryan Challinor. “Bespoke Anywhere”. In: *Proceedings of the New Interfaces for Musical Expression Conference (NIME)*. Shanghai, China, 2021.

[8] Dmitry Bogdanov et al. “Essentia: An audio analysis library for music information retrieval”. In: *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR)*. Curitiba, Brazil, 2013.

[9] Norbert Schnell et al. “PiPo, A Plugin Interface for Afferent Data Stream Processing Modules”. In: *International Symposium on Music Information Retrieval (ISMIR)*. Suzhou, China, 2017.

[10] Timothy Place, Trond Lossius, and Nils Peters. “The Jamoma Audio Graph Layer”. In: *Proceedings of the International Conference on Digital Audio Effects (DAFx)*. Graz, Austria, 2010.

[11] David Robillard. “LV2 atoms: A data model for real-time audio plugins”. In: *Proceedings of the Linux Audio Conference (LAC)*. Karlsruhe, Germany, 2014.

[12] Yann Orlarey, Dominique Fober, and Stéphane Letz. “Faust: an efficient functional approach to DSP programming”. In: *New Computational Paradigms for Computer Music*. Paris, France, 2007.

[13] Dominique Fober, Yann Orlarey, and Stéphane Letz. “FAUST Architectures Design and OSC Support.” In: *Proceedings of the International Conference on Digital Audio Effects (DAFx)*. Paris, France, 2011.

[14] Barry Rezvin and Jonathan Wakely. “Structured Bindings can introduce a Pack”. In: *ISO/IEC WG21 – 2022/04 mailing*. 2022.

[15] ISO/IEC. *C++20 standard*. 2020.