

Composing and Performing Electronic Music on-the-Fly with Pharo and Coypu

Domenico Cipriani¹, Sebastian Jordan Montaña², Nahuel Palumbo² and Stéphane Ducasse²

¹Pharo Association

²Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, Park Plaza, Parc scientifique de la Haute-Borne, 40 Av. Halley Bât A, 59650 Villeneuve-d'Ascq, France

Abstract

This paper introduces Coypu, a library and a dialect designed for on-the-fly music programming within the Pharo environment. Its syntax facilitates the creation of rhythmic patterns and incorporates stochastic techniques that employ randomness in composition, while providing an intuitive interface for manipulating scale and chords.

The main goal of Coypu is to provide an accessible and intuitive environment for live coding music: easy to install and to learn, yet powerful and inspiring. It is designed for newcomers and users with little programming experience.

Coypu aims to introduce people to both the world of live coding and Smalltalk in an engaging and approachable way. At the same time, Pharo's ease of defining new classes and methods, combined with its powerful collections API and version control tools, allows musicians to tailor the system to their needs.

Keywords

Pharo, on-the-fly music programming, Domain-Specific-Language, sound synthesis

1. Introduction

Live coding is a practice where the act of writing code becomes part of a live performance, often with the screen projected for a live audience [1]. It typically involves the real-time, improvisatory composition of computer-generated audiovisual material, especially music and visuals. Originating in the early 2000s, live coding has developed into a vibrant artistic and technological movement, framing code as a central expressive medium.

In this paper, we discuss the design and implementation of Coypu, a library and a dialect¹ designed for on-the-fly music programming within the Pharo environment [2]. We offer an overview of the core syntax and semantics, as well as implementation details. Additionally, we discuss its applications in musical performances and workshops, with a focus on user feedback and experience.

The development of Coypu originated from the author's intention to use Smalltalk for live coding performances. These performances were initially streamed online² during the COVID-19 pandemic using a tool developed for Symbolic Sound's Kyma system [3]. Kyma is recognised for its state-of-the-art³ synthesis engine and extensive sound design library. Although Kyma incorporates Smalltalk internally, it restricts users from defining custom classes and methods, thereby limiting the ability to develop novel programming paradigms for real-time musical programming. Furthermore, Kyma is proprietary software and depends on an external Audio Processor Unit for operation.

IWST 2025: International Workshop on Smalltalk Technologies, July 1–4, 2025, Gdansk, Poland

✉ mspgate@gmail.com (D. Cipriani); sebastian.jordan@inria.fr (S. Jordan Montaña); nahuel.palumbo@inria.fr (N. Palumbo); stephane.ducasse@inria.fr (S. Ducasse)

🌐 <https://jordanmontt.fr> (S. Jordan Montaña)

🆔 0009-0005-3023-3192 (D. Cipriani); 0000-0002-7726-8377 (S. Jordan Montaña); 0009-0001-5004-5632 (N. Palumbo); 0000-0001-6070-6599 (S. Ducasse)



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

¹In this paper we prefer to use the term *dialect* since Coypu is essentially a superset of Pharo that add methods and classes related to musical concepts and extends some of Pharo's core classes, while maintaining the same syntax and the same precedence rules

²Lucretio at UXR Zone 11.05.2020: <https://www.youtube.com/watch?v=-md6L6wdCPI&t=16385>

³Sonic State article on Kyma inventors

Pharo was chosen primarily for its extensive documentation, learning resources and its open source-ness, which were considered crucial for fostering the spread of the language within the community. Prior to Coypu, no other frameworks existed for programming music on-the-fly in any Smalltalk environment to the best of our knowledge.

In traditional musical terms, Coypu can be understood as a real-time editable score for multiple instruments. Coypu works as a client that composes live scores executable by OSC-compatible audio generators, MIDI software, external hardware, or directly within Pharo using Phausto [4, 5]. The separation between a score, which describes musical events over time, and an orchestra, which defines the synthesis algorithms or instruments, can be traced back to the MUSIC-N family of computer music languages, developed by Max Mathews at Bell Labs in the late 1950s [6].

Coypu is influenced primarily by TidalCycles [7] and Mercury [8]. TidalCycles is a widely adopted language for live coding music performances, while Mercury is a more recent platform that runs in the browser and emphasises ease of use and educational accessibility. As Coypu users initially focus on writing music creatively and quickly, with little emphasis on building or maintaining software, certain conventions or best practices in Pharo may have been disregarded.

The rest of the paper continues as follows: Section 2 describes the Performance class and why it is at the core of Coypu; Section 3 presents the performers and explains how they communicate with the audio servers; Section 4 shows Coypu, a Pharo dialect for music on-the-fly; Section 5 describes the limitations and the future work; Section 6 talks about how Coypu has been used in the last three years; and finally Section 7 concludes this paper.

2. The Performance and the Sequencers

At the core of Coypu is the Performance class. It is implemented as a singleton. The use of singletons is often criticised in the literature [9, 10] due to concerns regarding global state management, testability, and tight coupling. However, after careful consideration of these issues, we maintain our decision to prioritise the principle of economy within our dialect (see Section 4) and to favor the simplicity gained by allowing sequencers to be inserted into a single performance without requiring new instances.

Listing 1 shows an example. The code assigns the unique instance of Performance to a variable and sets the performer to an object that understands OpenSoundControl (OSC)⁴ messages. A *rumba* rhythm is assigned to a key in the Performance. The Performer interfaces with the conga instrument defined in the audio client, and the Performance is played for 16 bars.

```
p := Performance uniqueInstance.  
p performer: PerformerLocal new.  
#rumba asRhythm to: #conga.  
p playFor: 16 bars.
```

Listing 1: Performance class usage example

The Performance acts primarily as a MIDI arranger [11]. It stores tracks filled with events that are triggered by different instruments, which are rendered by the audio server (referred as Sequencers in Coypu terminology). The Performance also stores information about the frequency at which these events are triggered and is associated with a Performer subclass, responsible for generating the events for the audio server. The Performance can be played for a specified number of steps⁵ (typically defined by sending the *bars* message with an integer value), stopped, and its playback frequency can be adjusted in real-time. Additionally, the Sequencers within the Performance can be muted, soloed, or unsoloed as needed. The *playFor:* delegates the execution to the Performer subclasses [12]. When a Performance is played for a number of steps, the selected Performer advances the Performance’s playhead by incrementing its instance variable, *transportStep*. This advancement is driven by forking a Pharo process at `Processor timingPriority -1` ensuring the highest possible timing precision. A Performance

⁴wright97

⁵In Coypu a *step* is a sixteenth of a bar.

stores key-value pairs where all values must be Sequencer instances. The key for each Sequencer is stored within the Sequencer itself as *seqKey*, enabling the object to retrieve its key when needed. This *seqKey* plays a specialized role when either *PerformerSuperDirt* or *PerformerPhausto* is assigned to the *Performance*, as it selects the *external* instrument associated with the Sequencer (see Section 3).

A Sequencer instance stores the data that is read by a Performer and dispatched to the appropriate audio server. Conceptually, a Sequencer corresponds to what is commonly referred to as a track [6] in a Digital Audio Workstation (DAW) [13]. However, we chose the term Sequencer because its data structures are not constrained to uniform lengths; this flexibility allows for the composition of polyrhythmic patterns, which are often difficult to express using traditional loop-based approaches in DAWs.

A Sequencer always includes a core set of attributes: *gates*⁶, *note*, *duration* and *gate time*⁷, and *note index*⁸. Additionally, it may hold an array of extra parameters used to control sonic features such as amplitude (level), stereo positioning (panning), and timbre, depending on the nature of the associated instrument or sound source. A Sequencer can also optionally define a MIDI channel, enabling it to be played through a *PerformanceMIDI* object using a MIDI sender. For integration with the SuperDirt audio engine, the Sequencer includes a message template that determines how events are constructed and dispatched. The methods and techniques for creating Sequencer instances will be described in detail in Section 4.

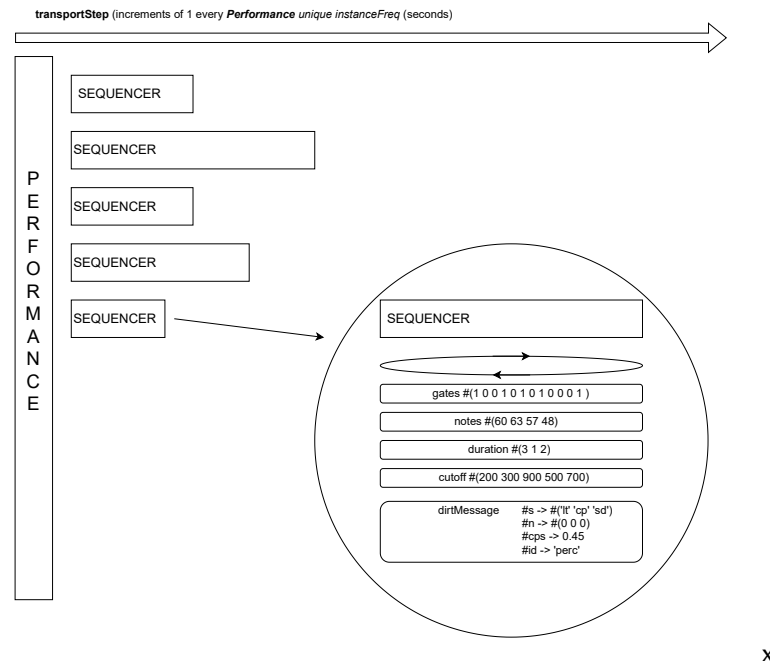


Figure 1: A Performance contains many Sequencers of different lengths. A Sequencer holds data for gates, notes, and durations, including the OSC messages to communicate with the server.

A Performance stores musical data in the form of *Sequencers*, as shown in Figure 1. In musical terms, a Sequencer can be thought of as an augmented pentagram, encompassing not only notes, rests, and durations but also information about timbres, effects, and articulations. Sequencers store values and data that the audio server uses to control the execution of synthesisers and samplers. These include MIDI note numbers, durations, velocities, and optional synthesis parameters, as well as specialised OSC messages and bundles.

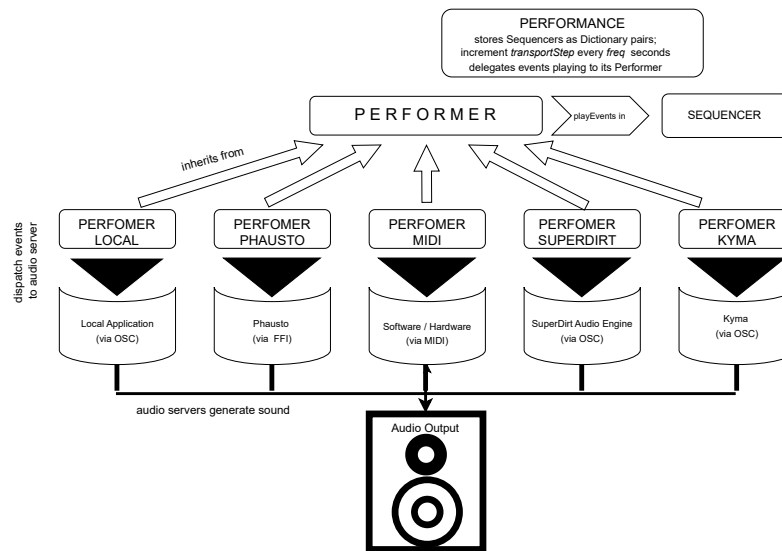
⁶A *gate* represents the on/off state of each step in a sequencer, determining whether a musical event is triggered or not.

⁷The *Duration* and *gateTime* parameters control for how long the note is sustained.

⁸The *noteIndex* parameter serves as an index that selects which instrument or audio sample to trigger at that step position.

3. The Performers

The performers enable the selection of the audio server that will render the *augmented score*⁹ contained in the Performance's sequencers¹⁰. A Performer queries the musical data stored within a Performance and communicates with the audio server, triggering note events and adjusting parameter values in real-time. An event is a control structured message that the performer sends to an audio server.



X

Figure 2: The 5 Performer subclasses send events contained in the Performance's Sequencers to a different audio server. This audio server is, in turn, responsible to generate sound.

An event contains information about the pitch, the velocity, the duration and additional parameters that the audio server will process to produce the sound.

The Performer class provides a common initialization routine and defines a template for the *playFor:* method. The responsibility for implementing the *playEventAt:in:* hook is delegated to its subclasses, thereby applying the Template Method design pattern [12].

Listing 2 shows the implementation of the method *playFor:*, which is a template method. It sets the performance tempo and iteratively processes sequencer events. It delegates the actual event handling to the hook method *playEventAt:in:*, to be implemented by subclasses.

```

Performer >> playFor: aNumberOfSteps
    "Set the performance tempo, where performance freq is the speed factor."
    self performance bpm: 60 / (performance freq * 4).
    self performance transportStep: 0.
    self performance activeProcess: ([
        aNumberOfSteps timesRepeat: [
            (Delay forSeconds: performance freq) wait.

            "Scan through the sequencers and trigger events"
            [
                self performance valuesDo: [ :seq |
                    (seq gates wrap: performance transportStep) = 1 ifTrue: [
                        self playEventAt: seq noteIndex in: seq.

                    "Increment note index after playing the event"
                ]
            ]
        ]
    ])

```

⁹The sequencers contain much more information than traditional musical scores, as they can contain synthesis parameters and instrument configurations managed by the audio engine or server.

¹⁰A Performance has a single performer slot, which holds one instance of a Performer subclass at a time.

```

        seq noteIndex: seq noteIndex + 1 ] ] ] forkAt:
        Processor timingPriority - 2.

    "Increment the transport step after scanning sequencers"
    self performance incrementTransportStep ] ] forkAt:
    Processor timingPriority - 1)

```

Listing 2: playFor: template method

We will provide a detailed discussion of the implementation of the Performer subclasses in the following subsections, outlining their similarities and exploring their differences in relation to the various audio servers with which they communicate.

3.1. PerformerLocal and PerformerSuperDirt

The PerformerLocal and PerformerSuperDirt classes dispatch events to an external application compatible with the OSC protocol via a UDP socket. The OSC protocol is designed for communication among computers, sound synthesizers, and other multimedia devices¹¹, optimized for contemporary networking technologies[14]. Coypu is not an audio synthesizer; it does not generate sound by itself. Instead, it sends events to an external audio synthesizer to be interpreted as sound. Several audio synthesis environments support OSC communication and hence can be used with Coypu. These include SuperCollider [15], Pure Data [16], Max/MSP [17], ChuckK [18], CSound [19], as well as multimedia platforms such as TouchDesigner, Processing [20], and OpenFrameworks¹².

The primary distinction between these two is that while PerformerLocal defines a syntax to which the external application must conform, PerformerSuperDirt is specifically designed to comply with the syntax of the SuperDirt audio engine used within the SuperCollider environment. To be playable with Coypu, instruments implemented in the audio server must include at least two parameters named according to the pattern *<Prefix>Gate* and *<Prefix>Note*. Any additional parameters controlling the instrument should adhere to the same naming convention, formatted as *<Prefix><ParameterName>*.

Coypu relies on the OSC package to construct packets (i.e. *messages*) and transmit them over the User Datagram Protocol (UDP). The OSC package for Pharo was originally developed and license under MIT by Markus Gaelli and then Simon Holland for Squeak.

The following method implementation illustrates how PerformerSuperDirt constructs an OSC bundle following SuperDirt specifications and sends it to SCSynth.

```

PerformerSuperDirt playEventAt: anIndex in: aSequencer
    "sends a message to SuperDirt with all the desired OSC arguments and values"

    | message dur stepDuration |
    stepDuration := Performance uniqueInstance freq asFloat.
    message := OrderedCollection new.
    message add: '/dirt/play'.
    dur := aSequencer durations asDirtArray wrap: anIndex.

    message
        add: 'delta';
        add: (stepDuration * dur) asFloat. "delta should change"
    aSequencer dirtMessage keysAndValuesDo: [ :key :value |
        message
            add: key;
            add: (value asDirtArray wrap: anIndex) ].

    (OSCBundle for: { (OSCMessage for: message) })

```

¹¹Developed in the late 1990s by Adrian Freed and Matt Wright at the Center for New Music and Audio Technologies (CNMAT), University of California, Berkeley, OSC is now widely adopted for both local- and wide-area networked media applications.

¹²<https://openframeworks.cc/about/>

```
sendToAddressString: '127.0.0.1'  
port: 57120.
```

Listing 3: The *playEventAt: in:* method constructs the OSC message that is sent to SuperCollider to play SuperDirt instruments

3.2. PerformerMIDI

The PerformerMIDI is designed to dispatch events either to external MIDI¹³ instruments or to local software applications and audio plugins via a virtual MIDI bus. To use the PerformerMIDI, first the PortMIDI library has to be downloaded. Then a MIDISender has to be created and opened with a device. When a MIDISender is opened on a device, it is automatically assigned to the corresponding slot in the PerformerMIDI class.

```
mout := MIDISender new.  
mout openWithDevice: 5.  
  
p := Performance uniqueInstance.  
p performer: PerformerMIDI new.
```

Listing 4: A new MIDISender object must be opened with a valid MIDI device to be used in a PerformerMIDI

3.3. PerformerPhausto

The PerformerPhausto component is designed to interface directly with the Phausto library and API [5] to generate sound using instruments and effects implemented within Phausto. Together, PerformerPhausto and Phausto form an integrated live coding framework for music, developed entirely within the Pharo ecosystem. This architecture effectively dissolves the traditional separation between the score and the orchestra. To use PerformerPhausto, a valid DSP must be assigned to the Performance; otherwise, an exception will be raised when the Performance starts. The events in the Sequencer are dispatched to the assigned DSP by the means of FFI calls that change the values of the parameters in the DSP assigned to the Performance. Instruments in a DSP implemented to be played with Coypu must adhere to the Coypu requirements as specified for the PerformerLocal.

4. A New Pharo Dialect for Programming Music On-the-Fly

The design of the Coypu API is guided by three core linguistic principles:

- **Iconicity:** The structure of the code should mirror the structure of the resulting music, establishing a resemblance between the form of the code and the meaning it produces. This concept¹⁴ is rooted in the linguistic notion of form-meaning similarity.
- **Economy:** Inspired by the principle of least effort [24], the API aims to minimize user input, promoting concise and efficient code that reduces the cognitive and physical demands of live coding.

¹³Musical Instrument Digital Interface is a technical standard that describes a communication protocol, digital interface, and electrical connectors used to connect a wide variety of electronic musical instruments, computers, and related audio equipment.(<https://midi.org/midi-1-0>)

¹⁴According to Haiman [21] and other typologists, such as Comrie [22], Hopper and Traugott [23], humans have a natural tendency to create linguistic signs that bear a resemblance to their meaning.

- **Semantic equivalence:** The same musical intent can be expressed in multiple syntactic forms, giving users the flexibility to choose between more economical expressions and those that prioritize clarity or iconic mapping. This implies that multiple expressions can be semantically equivalent [25].

For instance, many methods have been restructured or implemented in alternative forms to reduce the amount of typing required to modify values stored in arrays. Additionally, several multi-argument messages have been introduced to minimize the use of parentheses and cascading messages. Coypu implements various strategies for populating its sequencers with musical data, which will be described in the following subsections. If notes, durations and gateTimes are not defined at the instance creation the Sequencer will initialize with default values: a note value of **60**¹⁵, a duration equal to the minimum step interval, and a gate value of **0.8**, reflecting the behaviour of classic hardware sequencers.

Coypu can easily generate rhythmic patterns from global musical traditions by accepting an integer and a unary message specifying the rhythm name. Supported rhythms include culturally significant structures, such as Afro-Cuban clave, Middle Eastern folk meters, and other traditional percussion sequences. This educational content provides newcomers and sound artists with a deeper understanding of global musical traditions and highlights the shared roots of dance rhythms across cultures. Each method, accessible by inspecting `Rhythm` list, includes detailed ethnomusicological comments on the rhythm's geographical, historical, and social origins.

In the following subsections we illustrate different strategies to create Sequencers.

4.1. Creating Sequencers from Arrays

Sequencers can be created by sending the message `asSeq` to an array of 1s and 0s, where 1 indicates a trigger (i.e., an event is emitted) and 0 represents a rest. This is the simplest way to create a Sequencer. While it is verbose and prone to errors, it remains powerful when used to store rhythms derived from Time Unit Box notation [26], as demonstrated in the implementation of the `cumbiaClave` method.

4.2. Random generators and random walkers for notes and triggers

Coypu offers different methods to create Sequencers with random¹⁶ triggers or with random notes.

```
rand1 := 16 randomTrigs.
rand2 := 32 randomTrigsWithProbability: 65.
```

Listing 5: Two sequencers with random triggers: the first uses a uniform distribution, while the second generates triggers with a biased probability of 65%..

Listing 5 demonstrates the generation of rhythmic trigger patterns using two methods. The first line creates a sequence of 16 randomly generated triggers, representing on/off events. The second line produces a longer sequence of 32 triggers, where each trigger is activated with a 65% probability, allowing for controlled randomness in the rhythmic structure.

4.3. Euclidean rhythms and HexBeats

Euclidean rhythms are a class of rhythmic patterns that distribute a number of onsets (or beats) as evenly as possible across a given number of time steps (or pulses). The concept was introduced in a musical context by Godfried Toussaint [27], who also demonstrated that Euclidean rhythms underlie a wide range of traditional rhythms from around the world. By sending the message `euclidean` to an array of two integers, a Sequencer is created using a Euclidean rhythm generated through our Pharo implementation of the Bresenham algorithm [28]. The first element of the array specifies the number of triggers (onsets), while the second element specifies the total number of pulses.

¹⁵MIDI note number 60 corresponds to middle C

¹⁶We use the term “random” to refer to pseudo-random generation.

Hexadecimal notation has a long history in computer music, where it is used as a concise and structured way to represent values within a power-of-two range. In rhythm programming, each hexadecimal digit corresponds to four sixteenth-note subdivisions, equivalent to one beat in common time. The binary form of each digit encodes a pattern of rhythmic events, where a 1 indicates a trigger (onset) and a 0 represents a rest. For example, the digit **F** (binary 1111) results in four consecutive onsets, while **5** (binary 0101) produces an alternating pattern of onsets and rests. This notation offers a readable and expressive method for defining rhythmic sequences with minimal syntax. Sending the message `hexBeat` to a string composed of hexadecimal digits creates a *hexBeat*, a rhythmic onset pattern encoded in hexadecimal form.

4.4. A string notation system with rules based on the basics of TidalCycles' *mini notation*

Sequencers can be created using a string notation system inspired by TidalCycles' mini-notation.

```
| notes dirtNotes |
notes := '0/4 , 7 , 4/2 , 9 , 11 , 9 , 7 , 4 , 7 , 9 , 5 , 7*2 , ~'.
dirtNotes := notes asDirtNotes.
dirtNotes to: #superpiano.
'~ , 2 * 3 , ' ~ ' , 5 , 4 / 3' asDirtIndex to: #timbale
```

Listing 6: Simple example to create melodies and sound patterns with Coypu string notation

The string content must follow a specific syntax. Entries are separated by commas, with each entry representing either a rhythmic or melodic event. Numeric values correspond to notes (when the `asDirtNotes` message is sent) or to indexes (when `asDirtIndex` is used in a multisampler instrument). These numeric values implicitly generate rhythmic triggers. Rests are indicated by the tilde symbol (~) or a hyphen (-), both denoting the absence of a trigger at that step. Two operators are used to modify the temporal behavior of events: The asterisk (*) operator indicates repetition of a note across multiple consecutive steps. The slash (/) operator extends the duration of a note across a specified number of steps.

5. Limitations and Future Work

We consider Coypu to still be in its beta phase. In order to proceed toward a stable release, several current limitations need to be addressed:

- The lack of flexibility in choosing between the available linear notation and cyclical structures¹⁷ (as seen in Tidal Cycles, yet to be implemented).
- The inability to select different timing resolutions and internal rhythmic subdivisions.
- The jitter in the advancement Performance playhead¹⁸.

The first limitation may be addressed by implementing an alternative *playMode* that employs a different strategy for parsing the triggers stored within a Sequencer, while constraining the Sequencer length to a single bar. To enable the selection of various timing resolutions and rhythmic subdivisions, we propose the development of a new scheduling system inspired by Tone.js¹⁹. This system would

¹⁷Linear notation follows a sequential, approach in which musical events are programmed to occur at specific moments along a timeline (e.g., "play note 60 at step 1, then 63 at step 9"). In contrast, cyclical structures, define events by their position within a recurring loop (e.g., "play 60 and 63 alternating every half-cycle").

¹⁸Our experimental tests measured an average jitter of 1 milliseconds. This was determined by recording a downbeat rhythm at 120 beats per minute, performed by the Phausto Performer, and analyzing the timing deviations in an audio editing software.

¹⁹Tone.js is a JavaScript framework built on the Web Audio API, designed for creating interactive musical applications directly in the browser (<https://tonejs.github.io/>)

manage the scheduling of events stored in Sequencers and drive the advancement of the Performance playhead. We expect this redesigned scheduling mechanism to significantly reduce jitter. However, if the performance improvement proves insufficient, it may be necessary to fine-tune Pharo’s execution environment—particularly to lower the overhead caused by garbage collection and to minimize interruptions at yield points.

The difficulty of writing efficient tests for audio applications is well recognized²⁰ and complexity increases when testing frameworks for programming music *on-the-fly*, which involves both event dispatching and audio synthesis. A stable release also requires the development of robust testing tools for critical components such as dispatching OSC and MIDI messages, triggering Phausto events, and accurately measuring audio timing. This can be achieved by recording the audio buffer, using external software, or through a custom solution implemented in C and accessed via Foreign Function Interface (FFI). It is a significant challenge and it demands considerable time and investigation to build the necessary tools to enable precise testing and measurement.

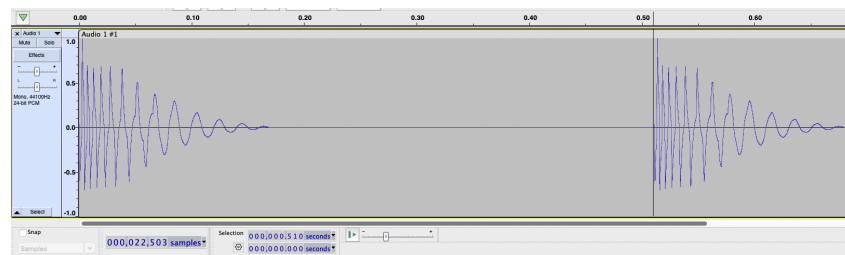


Figure 3: Time jitter measurement at 120 bpm with *Audacity* audio editing software

6. Use cases

Although technically still in its beta phase, Coypu has already been in steady use by the authors for close to three years across a range of settings—from Algoraves to music tech conferences—both as a live performance tool and as an educational platform for teaching live music programming. One of the authors performed with it live at the International Live Coding Conference 2024, hosted at System in Shanghai. It has been presented to diverse groups of users, ranging from expert live coders at the ICLC Satellite event in Düsseldorf in May 2023 to children and teenagers aged 9 to 16 at the Festival della Robotica di Pisa in May 2025. During these workshops, we focused on collecting user feedback. Participants with little or no prior coding experience appreciated Coypu’s ease of use, particularly when used in combination with TurboPhausto²¹. In contrast, experienced programmers preferred Pharo’s development tools, such as the Class Browser, Debugger, and Iceberg, along with its support for liveness and reflection. Several suggestions for future enhancements emerged during workshops and demonstrations. Among them is a plan to create a customised Playground using Bloc, a low-level UI infrastructure and framework for Pharo, aimed at offering intuitive graphical controls for both the Performance and the Performer. Another feature being explored would let users design their own interactive widgets through Toplo, enabling real-time control over Sequencer parameters in live coding scenarios.

²⁰For a detailed discussion on this topic, see Ryan Avery’s presentation at the Audio Developer Conference (ADC) 2017: Test-driven development for audio plugins.

²¹TurboPhausto is a collection of synthesisers and effects implemented in Phausto, modelled after the SuperDirt audio engine for SuperCollider, which is the default audio server for TidalCycles.

7. Conclusion

This paper presents Coypu, a library and dialect for live coding music with Pharo. The goal is to transform Pharo into a comprehensive IDE for live coding music, taking advantage of its syntax, tools, and reflectivity. The dialect's design principles of iconicity, economy, and semantic equivalence guide its approach to making musical programming easy to understand for both users and audience. We introduce its architecture built around the Performance-Sequencer paradigm, combined with multiple Performers. Our dialect's integration of diverse musical traditions, alongside modern computational techniques, creates a rich palette for creative expression. Three years of practical deployment across venues ranging from Algoraves to educational workshops have shown promising evidence of Coypu's dual role as both a performance tool and pedagogical platform. Coypu serves as a bridge between artistic and technological communities, introducing musicians and sound artists to Smalltalk. Current limitations around timing precision and cyclical structures remain areas for future development, along with the need to further spread the language within the live coding community.

Acknowledgments

We want to thank the Pharo Association for their financial support of this project. We thank the ESUG board for including our musical performance in the latest conference programs.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

- [1] A. F. Blackwell, E. Cocker, G. Cox, A. McLean, T. Magnusson, *Live Coding: a user's manual*, Mit Press, Cambridge, MA, 2022.
- [2] S. Ducasse, G. Rakic, S. Kaplar, Q. Ducasse, *Pharo 9 by Example*, Book on Demand – Keepers of the lighthouse, 2022. URL: <http://books.pharo.org>, originally written by A. Black and S. Ducasse and O. Nierstrasz and D. Pollet with D. Cassou and M. Denker.
- [3] C. Scaletti, The kyma/platypus computer music workstation, *Computer Music Journal* 13 (1989) 23–28.
- [4] D. Cipriani, N. Palumbo, S. Jordan Montaña, S. Ducasse, Phausto: fast and accessible DSP programming for sound and music creation in Pharo, in: *IWST 2024: International Workshop on Smalltalk Technologies*, Lille, France, 2024. URL: <https://hal.science/hal-04826894>.
- [5] D. Cipriani, A. Anatrini, S. Jordan Montaña, Phausto: Embedding the Faust Compiler in the Pharo World, in: *Proceedings of the International Faust Conference (IFC-24)*, Soundmit, Turin, Italy, November 21-22, 2024, Turin, Italy, 2024.
- [6] C. Roads, *The Computer Music Tutorial*, Mit Press, Cambridge, MA, 1996.
- [7] A. McLean, Making programming languages to dance to: Live coding with tidal, in: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, Madrid, Spain, 2014.
- [8] T. Hoogland, Mercury: a live coding environment focussed on quick expression for composing, performing and communicating, in: *Proceedings of the International Conference on Live Coding (ICLC)*, Madrid, Spain, 2019.
- [9] A. Hunt, D. Thomas, *The Pragmatic Programmer*, Addison Wesley, 2000.
- [10] W. J. Brown, R. C. Malveau, H. W. McCormick, III, T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley Press, 1998.
- [11] D. M. Hearn, *The MIDI Manual: A Practical Guide to MIDI in the Project Studio*, Focal Press, Oxford, UK, 2009.

- [12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [13] A. Reuter, Who let the daws out? the digital in a new generation of the digital audio workstation, *Popular Music and Society* 45 (2021) 1–16.
- [14] M. Wright, Open sound control: an enabling technology for musical networking, *Organised Sound* 10 (2005) 193–200.
- [15] S. Wilson, D. Cottle, N. Collins, The SuperCollider Book: Second Edition, 2nd ed., MIT Press, Cambridge, MA, 2015.
- [16] A. Cipriani, M. Giri, F. Bianchi, Pure Data: Electronic Music and Sound Design - Theory and Practice, volume 1, 2nd ed., Contemponet, Rome, Italy, 2016.
- [17] V. J. Manzo, Max/MSP/Jitter for Music, Oxford University Press, Oxford, England, 2016.
- [18] G. Wang, P. R. Cook, et al., Chuck: A concurrent, on-the-fly, audio programming language, in: ICMC, 2003.
- [19] V. Lazzarini, S. Yi, J. ffitich, J. Heintz, A. Cabrera, R. Wals, Csound: A Sound and Music Computing System, Springer, 2016.
- [20] C. Reas, B. Fry, Processing: A Programming Handbook for Visual Designers and Artists, MIT Press, Cambridge, MA, 2014.
- [21] J. Haiman (Ed.), Iconicity in Syntax (Proceedings of a symposium on iconicity in syntax), John Benjamins, Stanford, CA, USA, 1983.
- [22] B. Comrie, Language univesals and linguistic typology, Blackwell Publishers, Oxford, England, 1981.
- [23] P. J. Jopper, E. C. Traugott, Grammaticalization, Cambridge University Press, Cambridge, England, 1993.
- [24] G. K. Zipf, Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology, Addison-Wesley Press Inc., Cambridge 42, MA, USA, 1949.
- [25] J. Lyons, Semantics, Cambridge University Press, Cambridge, England, 1977.
- [26] J. Koetting, Analysis and notation of west african drum ensemble music, *Selected Reports in Ethnomusicology* 1 (1970) 116–146.
- [27] G. T. Toussaint, The geometry of musical rhythm: what makes a" good" rhythm good?, CRC Press, 2019.
- [28] J. Bresenham, A linear algorithm for incremental digital display of circular arcs, *Communications of the ACM* 20 (1977) 100–106.