Revista Vórtex | Vortex Music Journal | ISSN 2317–9937 | http://vortex.unespar.edu.br/
https://doi.org/10.33871/vortex.2026.14.12468

ORIGINAL PAPER

dddd: Closing the Abstraction Gap in Pure Data for Computer-Aided Composition

Charles K. Neimog

University of São Paulo, Escola de Comunicações e Artes (ECA) | São Paulo, Brazil

Resumo: Compositores pensam em acordes, gestos, ritmo/tempo e formas, não em redes de pack e unpack. Este artigo argumenta que a principal contribuição de *dddd* é devolver tempo ao pensamento musical no Pure Data (Pd). A biblioteca introduz uma camada de abstração em Lua que permite manipular estruturas hierárquicas como entidades musicais de primeira classe, reduzindo a complexidade acidental descrita por Brooks (1987). O modelo é demonstrado por duas bibliotecas: *pd-bhack*, voltada à Composição Assistida por Computador (CAC) e à notação simbólica inspirada no OpenMusic, e *pd-upic*, dedicada a representações gráficas baseadas no sistema UPIC de Iannis Xenakis. Embora aproxime o Pd de ambientes como Max (BACH) e OpenMusic, *dddd* preserva seu modelo nativo e pode ser compilado para a Web via *pd4web*, ampliando seu potencial para aplicações interativas, educacionais e de pesquisa.

Palavras-chave: Composição Assistida por Computador, Pure Data, Bach Library.

Abstract: Composers think in terms of chords, gestures, time/rhythmic, and forms—not networks of *pack* and *unpack* objects. This article argues that the primary contribution of *dddd* is to return time to musical thinking within Pure Data (Pd). The library introduces a Lua-based abstraction layer that enables hierarchical structures to be manipulated as first-class musical entities, reducing the accidental complexity described by Brooks (1987). The approach is demonstrated through two libraries: pd-bhack, designed for Computer-Assisted Composition (CAC) and symbolic notation inspired by OpenMusic, and pd-upic, dedicated to graphical representations based on Iannis Xenakis's UPIC system. While bringing Pd closer to environments such as Max (through BACH) and OpenMusic, *dddd* preserves Pd's native programming model and can be compiled for the Web via pd4web, expanding its potential for interactive, educational, and research applications.

Keywords: Computer-Aided Composition, Pure Data, Bach Library.

Received on July 14, 2026. Accepted on October 30, 2026. Available online: October 30, 2026. Editors: Dr. Felipe de Almeida Ribeiro; Dr. Fabio Guilherme Poletto. Creative Commons CC-BY: https://creativecommons.org/licenses/by/4.0/

Composers think in chords, harmonic progressions, time/rhythm, graphical trajectories, and formal sections; they do not think in floats. Yet in Pure Data (Pd), many computer-aided composition workflows force composers to translate these musical objects into atomic messages, flat lists, and explicit routing networks before any musical transformation can occur. The result is not simply a technical inconvenience, but a cognitive displacement: time that could be spent reasoning about harmony, rhythm, form, or gesture is spent maintaining the machinery required to represent them. This article argues that *dddd* can help return time to musical thought by closing the gap between the objects composers imagine and the low-level messages Pd natively transmits.

dddd (Dictionary-Driven Data Design), whose name playfully references the *llll* (Lisp-like linked list) data structure of the BACH library for Max (Agostini; Ghisi, 2012), addresses this limitation by providing structured data abstractions for Pure Data (Pd). Instead of exchanging flat lists of values, Pd objects can communicate through hierarchical structures representing entities such as chords, rhythm trees, and graphical forms. In conceptual terms, *dddd* provides a foundation for Abstract Data Types (ADTs) in Pd: musical objects can be manipulated according to their meaning rather than how their constituent values are represented.

Implemented in Lua through the *pdlua* binding (Gräf; Schoen; Wesch, 2025), each *dddd* object is represented internally using recursive Lua tables. Unlike languages that distinguish between arrays and dictionaries, Lua provides a single table data type that supports both sequential, integer-indexed elements and associative key–value mappings. Tables can also contain other tables, allowing arbitrarily nested data structures. This flexibility makes them suitable for representing musical entities that combine ordered collections with named attributes.

Communication between Pd objects nevertheless remains compatible with Pd's native message-passing model through lightweight identifiers. In a conventional patch, a musical object such as a chord usually has to be decomposed into separate lists or message streams for pitches, velocities, durations, articulations, noteheads, and other symbolic attributes. These components must then be routed, synchronized, unpacked, and reconstructed whenever the chord is transmitted or processed. With *dddd*, a Pd object can instead emit a compact message such as *dddd* <*RANDOM_ID*>, from which receiving objects transparently recover the complete representation.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

The chord remains a single computational entity throughout the patch; specialized operations such as [*chord.get pitches*] or [*chord.get duration*] can access individual attributes without requiring the composer to split and later reassemble the underlying musical data.

In this respect, Pd moves closer to the mechanisms found in computer-aided composition environments such as OpenMusic (Assayag et al., 1999) and BACH (Agostini; Ghisi, 2012). Those systems allow composers to manipulate symbolic musical objects directly, whereas Pd's control domain remains centered on atomic messages. *dddd* does not replace Pd's dataflow paradigm; it extends that paradigm to musical structures while preserving the language's deterministic execution model.

Several projects have enriched Pd's compositional capabilities through scripting environments, interoperability bridges, or specialized notation libraries, including *py4pd*, *om-pd* (Neimog; De Souza, 2022), Jaime E. Oliver's [*notes_lib*], and components of the ELSE library. They expand Pd's functionality, but generally address specific compositional tasks or rely on external environments, leaving the underlying representation of complex musical data unchanged. *dddd* is intended instead as common infrastructure for multiple compositional libraries developed entirely within the Pd ecosystem using Lua language.

Two independent implementations provide the practical tests developed below. *pd-bhack* offers symbolic notation, rhythm trees, harmonic structures, and notation-aware playback inspired by the OpenMusic ecosystem. *pd-upic* explores graphical representations based on Iannis Xenakis' UPIC system. Their contrasting domains test whether the same infrastructure can support distinct compositional paradigms without leaving Pd's visual programming model. Since the implementation is entirely in Lua, it can also be deployed through *pd4web* (Neimog; Coelho De Souza, 2025) extending these uses from desktop applications to web environments, including smartphones – a practical advantage for sound installations.

The source code of *dddd* is publicly available on GitHub (Neimog, 2026a), along with developer documentation (Neimog, 2026b).

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

1. The Data Representation Problem in Pd

Pd was designed primarily for real-time digital signal processing and synthesis. Its execution model and native data representation reflect this objective: information is exchanged via a small set of atomic message types including floats, symbols, and flat lists. This minimalistic design provides deterministic and efficient execution, making Pd particularly effective for audio-rate and control-rate computation. However, the same design that benefits real-time processing also limits the representation of complex information.

Formal compositional thought operates on hierarchical rather than atomic structures. A chord is not merely a sequence of pitches, nor is a rhythmic measure simply a list of durations. Musical entities encapsulate multiple interrelated attributes – including pitches, durations, dynamics, articulations, noteheads, and structural relationships – that together define a single conceptual object. When these hierarchical structures must be represented using only flat lists, musicians are forced to encode semantic relationships manually instead of expressing them directly.

Brooks (1987) describes this phenomenon as *accidental complexity*: complexity introduced by the implementation, programming language, or computing environment rather than by the inherent requirements of the problem itself. A useful analogy is numerical notation. Performing arithmetic with Roman numerals is considerably more difficult than with Arabic numerals, not because addition is inherently difficult, but because Roman numerals lack the positional structure required for efficient computation. The additional effort originates from the representation rather than from the mathematical operation.

Pd's native message types produce the same difficulty when used to represent complex musical structures. A rhythm tree, chord, or graphical object must be decomposed into flat sequences of values and repeatedly reconstructed throughout a patch. Composers consequently spend substantial time building networks of [*pack*], [*unpack*], [*route*], [*trigger*], and related objects simply to preserve relationships among data. Each additional routing network diverts attention from musical work – listening, testing harmonic alternatives, shaping form, or refining the behavior of a system – and toward implementation details. In Brooks' terms, the accidental complexity of the implementation begins to overshadow the essential complexity of the musical problem. For *dddd*, reducing it is as

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

much a compositional motivation as a technical one.

Liskov and Zilles (1974) argue that robust software systems are built around Abstract Data Types (ADTs), which separate an object's interface from its underlying representation. From a compositional perspective, a chord should be defined by operations such as transposition, inversion, filtering, or serialization – not by the way its constituent values are stored or transmitted within Pd.

Implementation problems are appealing because they are concrete and usually admit well-defined solutions, yet solving one frequently exposes another, creating an effectively endless sequence of technical tasks. Composition, by contrast, is not a problem with a single correct solution. It requires sustained engagement with open-ended musical questions whose answers emerge through exploration rather than debugging. Interrupting that process to solve implementation problems does more than consume time: when the composer eventually returns to the musical problem, the original line of thought may have been lost. New musical ideas often emerge instead, bringing different technical requirements and leading the work in a different direction. The design of *dddd* was ultimately guided by my own compositional perspective: at certain stages of the compositional process, data representation should become largely invisible so that the composer's attention remains focused on musical thought rather than on the mechanics of its implementation.

2. The dddd Implementation: Solving the Representation Problem

dddd stores musical data internally as recursive Lua tables, but encapsulates the implementation from the graphical patch environment. Composers can therefore work through high-level musical operations, while developers retain access to low-level programming and data-structure design when extending the framework. Musical objects may also be serialized as textual expressions using a strict bracketed syntax inspired by Lisp. A dedicated parser reconstructs these expressions as recursive Lua tables, so complex structures can be created, inspected, and exchanged directly within Pd patches. The distinction between external representation and internal implementation creates a useful separation within the workflow: everyday compositional work proceeds with chords, rhythm trees, or formal structures, without the repeated decomposition and reconstruction demanded by Pd's native flat messages.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

Transporting these hierarchical structures between graphical objects while preserving Pd's dataflow semantics requires a deterministic communication mechanism. Rather than transmitting the complete data structure through patch cords, *dddd* employs a temporary routing-by-reference strategy. Whenever an object emits a *dddd* instance, the corresponding hierarchical structure is registered in a global Lua registry associated with the current Pd instance and indexed by a randomly generated twelve-character hexadecimal identifier. Only this lightweight identifier – for example, *dddd <d330a16cda70>* – is propagated through the patch cord, while the actual data remains stored internally in a Lua table.

Although the architecture relies on references, its observable behavior preserves Pd's pass-by-value semantics. Upon receiving an identifier, an object immediately resolves the corresponding registry entry and reconstructs an independent copy of the associated structure. This behavior relies on Pd's depth-first synchronous execution model. Once the synchronous execution chain initiated by the originating object finishes, the corresponding registry entry is removed, allowing Lua's garbage collector to reclaim the memory. Coupling object lifetime to Pd's deterministic execution order provides automatic memory management without requiring the programmer to allocate or deallocate these structures explicitly.

This mechanism – creating an independent copy for each receiving object – is essential to Pd's message-passing semantics. In a graphical patch, a single outlet is often connected to multiple objects. If every receiver operated on the same underlying data structure, a destructive transformation performed by one object – such as transposing a chord, filtering a collection, or inverting a rhythmic sequence – would modify the data subsequently processed by all other connected objects. Such side effects would violate the deterministic behavior expected from Pd patches.

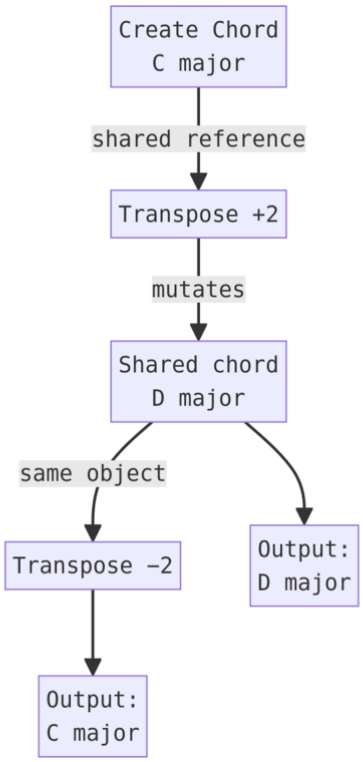
Consider an example. Suppose one object generates a C-major chord, and, as a composer, you want to create a symmetrical texture by producing versions transposed two semitones above and two semitones below the original. To accomplish this, the chord generator is connected to two transpose objects: one configured with +2 and the other with −2. If both transpose objects receive references to the same underlying chord, the order in which messages are dispatched becomes significant. Assume the +2 transpose receives the message first. It transforms the shared chord from C major to D major. When the −2 transpose is subsequently invoked, it no longer receives the original C-major

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

chord; instead, it operates on the already modified D-major chord, producing C major rather than B♭ major. Consequently, instead of obtaining the expected pair of B♭ major and D major, the patch produces C major and D major.

By giving each receiving object its own copy of the data, every transformation is applied independently to the original message, making the result independent of connection order and preserving the deterministic semantics of the patch.

FIGURE 1 – In this figure, it is shown the problem generated by passing the values of Pd object by reference and not by value. In this case, the composer wants to create D, F#, A and Bb, D, F from a C, E, G chord. Nevertheless, what is created (without copying) is C, E, G and D, F#, A.[1]



The strategy costs more than shared-reference techniques such as copy-on-write, since every transmission requires a recursive deep copy. I chose this trade-off because preserving Pd's pass-by-value semantics mattered more than minimizing allocations or CPU use. Eliminating shared mutable state maintains deterministic behavior and prevents upstream memory mutations, both valuable where compositional results must be reproducible. The operations should occur exclusively at the

[1] All figures and tables were created by the author.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

control rate, where the overhead has generally proved acceptable. For more demanding applications, *pd-lua* also supports LuaJIT, whose just-in-time compilation substantially accelerates recursive table manipulation compared with the standard interpreted Lua runtime.

Communication among *dddd* Pd objects depends on a shared global registry, so the framework must maintain a single runtime instance. For this reason, *dddd* is distributed exclusively as an integrated component of the *xlab* library. Independent copies embedded in third-party libraries would create incompatible registries and prevent reliable data exchange. Centralizing the implementation within *xlab* keeps inter-object communication consistent across concurrent compositional projects. The framework requires Pure Data 0.57 or later, or PlugData 0.9.4 or later.

2.1. Relationship to bach's llll Data Structure

The design of *dddd* is closely related to the *llll* data structure introduced by the BACH library for Max (Agostini; Ghisi, 2012). Both systems address the limitations of flat messages in visual dataflow environments by introducing hierarchical representations capable of handling complex musical information. They also adopt similar communication strategies: instead of transmitting complete structures through patch cords, objects exchange lightweight identifiers associated with internally stored data. Both systems preserve pass-by-value semantics, allowing receiving objects to manipulate their data independently. In this respect, *dddd* does not propose a fundamentally new approach to hierarchical message transmission, but rather adapts a comparable principle to the Pure Data environment.

The principal difference lies in their internal data representations. Bach implements *llll* as hierarchical linked lists written in C/C++, emphasizing sequential collections that can be nested to arbitrary depths. Although associative relationships can be represented by treating the first element of a sublist as a key, retrieving a value requires traversing the corresponding list (Agostini; Ghisi, 2012). In contrast, *dddd* uses recursive Lua tables, which natively combine sequential indexing and associative key–value mappings. This allows structures to contain both ordered elements and named attributes without requiring separate representations. For example, a chord can contain an ordered collection of notes alongside attributes describing its duration, articulation, or other properties. The

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

distinction is therefore not the ability to represent hierarchical information, which both systems provide, but the organization and accessibility of that information.

Another important distinction concerns the implementation language. Bach relies primarily on C/C++ for its underlying data structures and computationally demanding operations, providing direct control over memory and performance. *dddd* instead uses Lua through the *pdlua* binding. This choice prioritizes development flexibility and extensibility: recursive structures can be constructed and manipulated using Lua's native data types, while new domain-specific objects can be implemented without developing or compiling additional C externals. This is particularly relevant to the intended role of *dddd* as a common infrastructure for independent compositional libraries. As Agostini and Ghisi (2015) acknowledge, implementing complex operations directly in C can present challenges for developers, particularly composers with limited programming experience.

These advantages involve computational trade-offs. Lua tables and automatic garbage collection simplify memory management but introduce runtime overhead compared with carefully optimized native data structures. Furthermore, *dddd* currently performs recursive deep copies when transmitting data between objects, potentially increasing memory allocation and processing costs. Although LuaJIT can mitigate some of these costs, *dddd* is not intended to outperform C-based implementations in computationally intensive applications. Its design instead favors a balance between expressive data representation, ease of extension, and compatibility with Pd's execution model. An additional practical consequence is integration with *pd4web*, allowing the same Lua-based framework and its associated compositional libraries to operate in WebAssembly-based browser environments.

Consequently, *dddd* should not be understood as a replacement for BACH or as an attempt to reproduce its extensive compositional functionality. Rather, it provides Pd with a comparable foundation for hierarchical data manipulation while adopting a different internal representation and development model. Musical structures such as chords, rhythm trees, and graphical objects are defined by libraries built upon *dddd*, whereas the framework itself remains responsible for the general mechanisms of data construction, transmission, serialization, and runtime type validation.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

3. Architecture of the dddd Framework

Symbolic object models developed in computer-aided composition environments such as OpenMusic strongly influenced the design of *dddd*. Reproducing those environments was never the aim. The narrower task is to provide infrastructure on which independent compositional libraries can be implemented directly within Pd: constructing and manipulating recursive data, transmitting it through Pd's dataflow architecture, and supporting domain-specific abstractions.

A *dddd* instance is a hierarchical data object constructed and managed through the *dddd* framework. Its internal representation consists of a recursive Lua table, optionally associated with a semantic type that identifies its intended interpretation. A Lua table alone does not necessarily constitute a *dddd* instance; the distinction lies in its integration with the framework's construction, type management, and communication mechanisms.

Internally, every *dddd* object is represented as a hierarchical Lua table. The framework provides facilities for constructing these structures from Pd messages or existing Lua tables, serializing and parsing nested representations, transmitting structured objects through the dataflow protocol, and inspecting their contents during development. Table 1 summarizes the principal operations provided by the framework.

TABLE 1 – Functional organization of the principal operations provided by the dddd framework.

Category	Representative Operations	Purpose
Construction	*new*, *new_from_atoms*, *new_from_table*	Construct hierarchical *dddd* objects from Pd atoms or existing Lua tables.
Serialization	*to_table*, *parse_list*	Convert between textual nested representations and the internal hierarchical structure.
Communication	*Output*, *new_from_id*, *get_dddd_from_id*	Transmit structured objects through Pd's dataflow and reconstruct them on the receiving side.
Runtime Type System	*set_type*, *get_type*, *assert_type*	Associate semantic types with objects and validate compatibility during execution.
Inspection	*print*, *get_table*	Inspect and debug the internal representation of *dddd* objects.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

For example, *dddd* provides the mechanisms required to store, transmit, and validate a chord object, but does not define the musical properties of a chord or implement operations such as transposition. These responsibilities belong to *pd-bhack*. Similarly, *pd-upic* defines graphical structures and their musical interpretation, while relying on the same *dddd* infrastructure for data management and communication.

Early versions of *dddd* represented recursive data without any notion of type. That was sufficient for storing arbitrary information, but a receiver could not distinguish a chord from a rhythm tree or a graphical object. Incompatible connections produced no informative diagnostics, and debugging became increasingly difficult as the libraries grew.

The resulting lightweight runtime type system is based on three operations: *set_type*, *get_type*, and *assert_type*. It does not enforce static typing. Libraries instead associate type annotations with their data and validate them during execution, detecting incompatible connections while retaining the flexibility needed by independent library authors.

For example, the *pd-upic* library defines a domain-specific type named *SvgObject*. If such an object is accidentally routed to an object from *pd-bhack* that expects symbolic notation rather than graphical data, the receiving object can invoke *assert_type* to verify the expected type and immediately report a descriptive error. Because the type system is intentionally lightweight, individual libraries remain free to define their own semantic object types while sharing the same underlying *dddd* infrastructure.

Extensibility follows directly from the communication protocol. Any Lua object can participate in *dddd* dataflow by implementing an *in_X_dddd* method, where X denotes the inlet number. Providing this interface makes the object compatible with other *dddd*-aware libraries, so independently developed compositional tools can exchange data without further adaptation.

4. Evaluating dddd through Two Compositional Libraries

The two libraries place different demands on *dddd*. *pd-bhack* requires rhythm trees, chords, microtonal pitch representations, and notation-aware playback; *pd-upic* requires graphical objects, attribute-based routing, and recursive parent-child relationships. The first evaluates symbolic

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

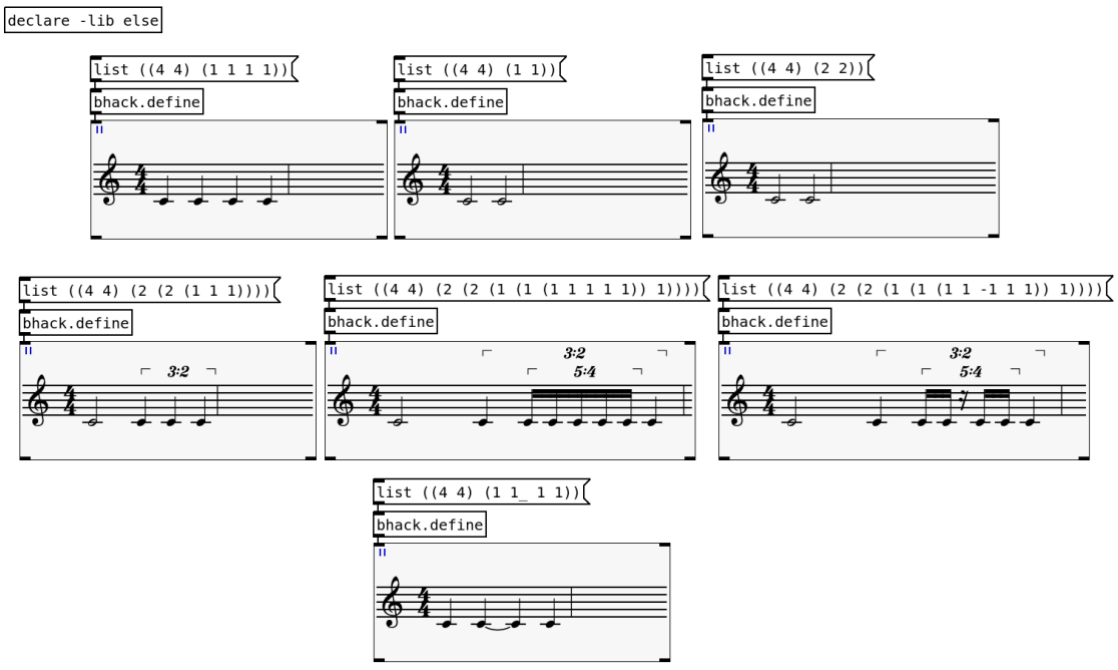
organization, the other spatial and graphical organization.

4.1. Symbolic Notation with pd-bhack

pd-bhack is a library for symbolic notation and computer-aided composition in Pd. It is not presented here as a complete notation environment, but as a demanding use of the *dddd* model. A voice object capable of rendering pitched measures is encoded not as a flat sequence of pitches and durations, but as a structure containing measures, time signatures, chords, notes, articulations, noteheads, stems, and playback attributes.

One of the central models implemented in pd-bhack is the rhythm tree. Rhythm trees were developed in the context of PatchWork and OpenMusic as a way of representing rhythm as a nested hierarchy of timespans. In pd-bhack, this model is adapted to Pd through *dddd*: each measure is represented as a structured object whose first component defines the time signature and whose second component defines the rhythmic content. For example, the expression ((4 4) (1 1 1 1)) represents a 4/4 measure containing four quarter notes, while nested structures such as ((4 4) (2 (2 (1 1 1)))) represent subdivisions such as tuplets, as shown in Figure 2.

FIGURE 2 – Different rhythm-tree inputs and their corresponding rendering in musical notation.



https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

Crucially, the rhythmic organization remains computationally intact rather than being flattened into unrelated values. A tuplet, rest, or nested subdivision can be inspected, transformed, rendered, or transmitted without manual reconstruction through Pd routing objects. Such operations preserve both the musical relationships and the environment's dataflow model.

The same abstraction principle is used for symbolic pitch and notation attributes. In `pd-bhack`, a chord may contain several notes, while each note can carry independent information such as pitch name, MIDI value, accidental glyph, notehead, articulation, or stem behavior. This makes it possible to separate symbolic notation from playback representation. For example, the same pitch object may store a written spelling, an uncommon accidental, and a MIDI value as related but distinct attributes. This separation is especially relevant for microtonal notation, where *pd-bhack* supports conventional 12-EDO, quarter-tone 24-EDO, and 96-EDO accidentals (following new OpenMusic notations).

Notation-aware playback follows the same design. Conventional MIDI playback reduces a score to numerical events such as pitch, velocity, and duration. *pd-bhack*, by contrast, can emit noteheads, articulations, stems, rhythmic values, and pitch spellings during playback. Routing these attributes independently inside Pd allows written notation to control synthesis, sample selection, or other compositional processes. Playback thus exposes the score's content to the patch instead of merely reproducing MIDI notes.

4.1.1. Symbolic Playback for Computational Music Research

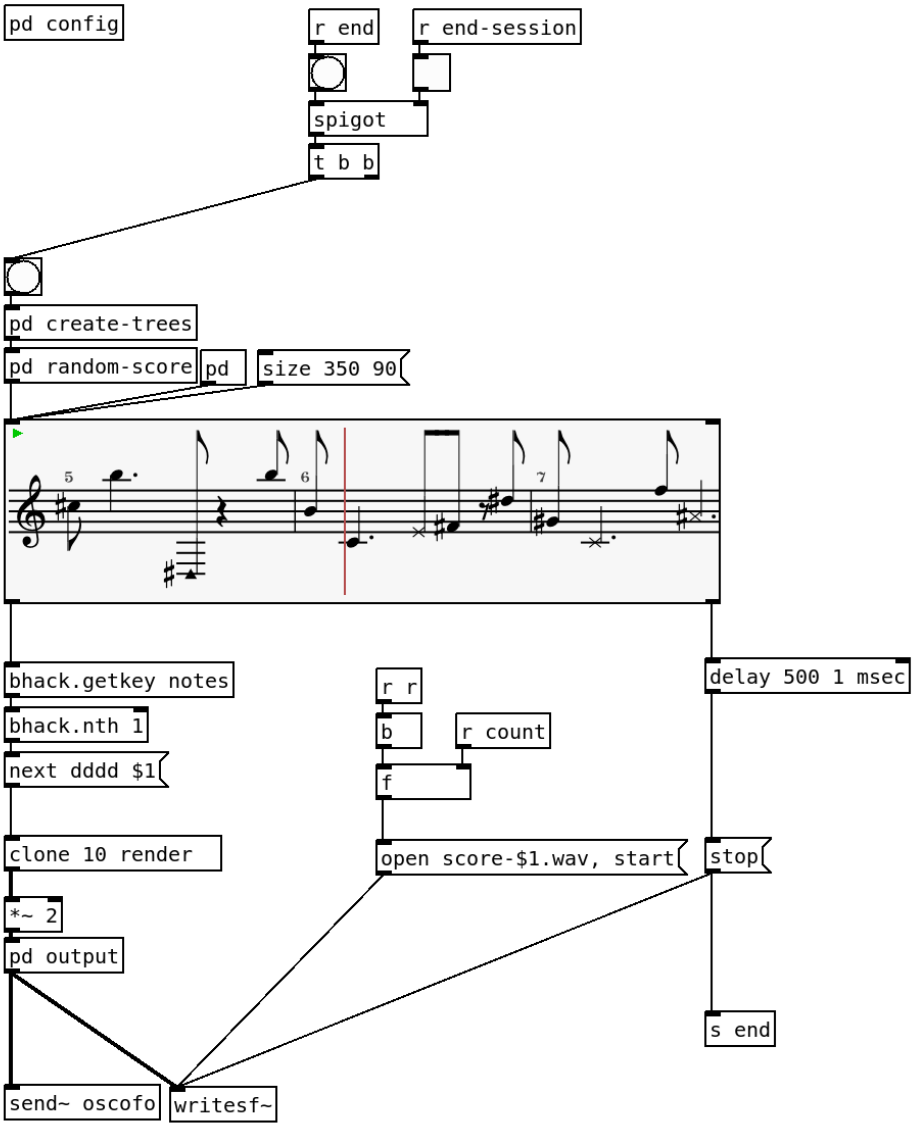
Symbolic playback also has uses beyond composition. Because its events preserve notation-level information rather than reducing the score to MIDI messages, *pd-bhack* can generate datasets containing pitches, rhythmic values, noteheads, articulations, stems, accidentals, and other notation attributes. Machine-learning systems in computational music research often require precisely this kind of annotated symbolic data for training, validation, and benchmarking.

Unlike conventional MIDI, which represents performance events primarily, symbolic playback retains the relationship between written notation and its musical meaning. Pd patches can thereby generate symbolic *corpora* automatically, including extended notation and microtonal accidentals

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

rarely available in existing datasets. In Figure 3, pd-bhack generates synthetic musical material that can be rendered and used to evaluate score-following systems such as OpenScofo (note that the different noteheads are just possible because of the encapsulation provided by *dddd*).

FIGURE 3 – Automatically generated symbolic score produced by pd-bhack and rendered for score-following.



Although originally developed for score-following research, the workflow has broader uses. A shared notation representation can serve composition, symbolic analysis, dataset generation, and machine-learning workflows without separate internal data models.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

Taken together, these uses of *pd-bhack* test more than storage. Rhythm trees require nested temporal representation; chords and notes require nested symbolic entities; microtonal spellings require notation to remain distinct from playback; and notation-aware events must become active parameters in a Pd patch. Dataset generation then carries the same representation into research beyond composition. No intermediate conversion to unrelated atoms is needed.

4.2. Graphical Composition with pd-upic

pd-upic approaches the same data model from graphical composition, drawing on the UPIC system developed by Iannis Xenakis. Whereas the original UPIC relied on dedicated hardware, pd-upic uses Inkscape and SVG as its graphical interface. Spatial and graphical structures impose requirements quite different from those of symbolic notation.

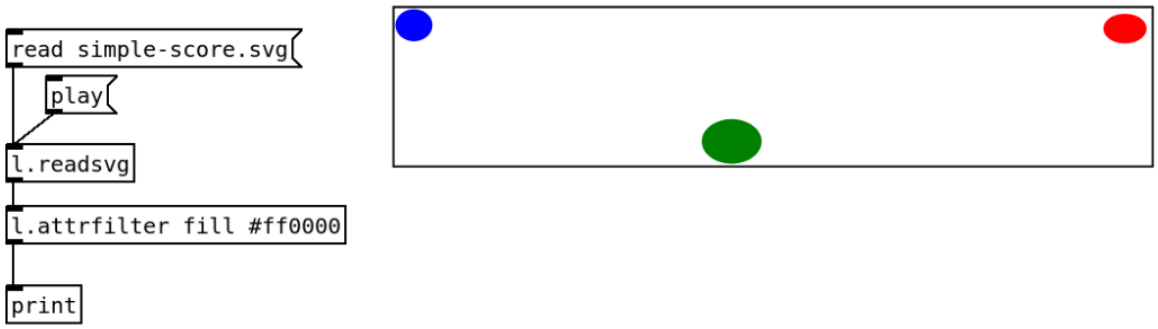
The central object in pd-upic is *u.read.svg*. This object reads an SVG file containing graphical elements and interprets selected regions as temporal containers. Each valid container is defined by two custom properties: duration and onset. These properties determine the temporal position and duration of the container, while its width in pixels is used to compute the relative timing of the graphical elements located inside it. Rectangles, circles, lines, and other SVG primitives can therefore be interpreted not only as visual objects, but also as time-based musical events.

Once the SVG file has been parsed, [*u.read.svg*] generates an independent *dddd* structure for each graphical element. During playback, triggered by a play message, the object advances an internal clock with millisecond resolution. Whenever the current time corresponds to the onset of a stored graphical element, the corresponding *dddd* object is emitted into the Pd patch. At that point, the graphical element is no longer treated as a flat list of coordinates, but as a structured object containing attributes such as position, size, color, stroke, fill, parent container, onset, and duration.

The resulting representation supports attribute-based processing inside Pd. A graphical element may, for example, be filtered by its visual properties before reaching a synthesis or processing patch. In Figure 4, *l.attr.filter* forwards only objects whose fill attribute equals *#ff0000*, so the print object receives only the final red circle. The filter itself is elementary; what matters is that it operates on a graphical object rather than on a manually indexed list of values.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

FIGURE 4 – Attribute-based filtering in pd-upic: *u.read.svg* emits graphical elements as dddd objects, while l.attr.filter forwards only objects whose fill attribute matches #ff0000.



Here, the same infrastructure used for symbolic and temporal relationships in *pd-bhack* carries graphical and temporal ones. In either domain, Pd patches operate on relevant attributes without first translating the compositional material into low-level lists.

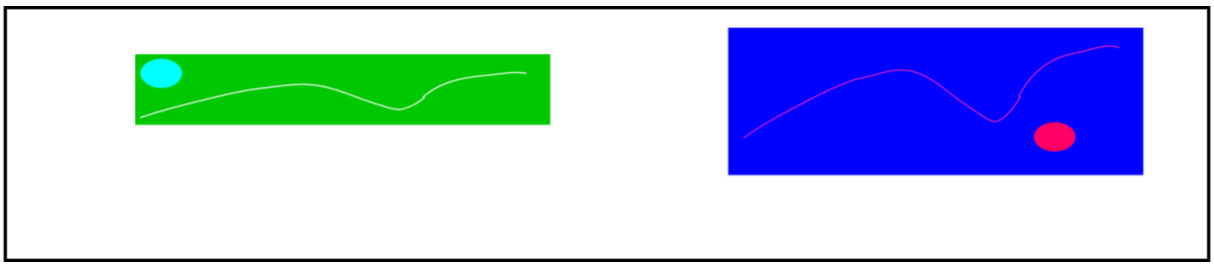
4.2.1. Parent-Child Relationships in Graphical Objects

The first work composed with *pd-upic* exposed an important limitation: graphical elements could initially be associated with only one set of synthesis or processing parameters. Similar materials appearing in different structural contexts were therefore difficult to differentiate. The same type of line or ellipse, for example, might need to trigger different spatialization, filtering, or reverberation processes depending on the larger region in which it appears.

I therefore extended *pd-upic* with recursive parent-child relationships between graphical objects. When *u.read.svg* detects elements within another valid temporal container, it does not insert them directly into the main timeline. They are stored as children of the parent object. In Figure 5, only the green and blue rectangles are scheduled in the main timeline; the line and ellipse remain attached to their respective parents until explicitly retrieved.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

FIGURE 5 – Hierarchical graphical objects in pd-upic: only parent containers are scheduled in the main timeline, while nested elements remain attached as children until retrieved by u.play.children.



The nested elements can later be accessed through *u.play.children*, which retrieves and schedules the children associated with a parent graphical object. Before this happens, the parent object may be filtered, routed, or transformed according to its own attributes. As a result, identical graphical primitives can lead to different musical outcomes depending on the parent structure in which they occur. A line inside a green rectangle, for instance, may be sent to one synthesis chain, while a visually similar line inside a blue rectangle may be sent to another.

Recursive child objects preserve graphical relationships directly inside Pd despite its largely flat native data model. Equivalent behavior could be implemented with explicit pointer-based structures in C or C++, as in some graphical programming libraries, but that choice would expose library developers to low-level memory management. *dddd* retains the same relationships while remaining compatible with Pd's dataflow semantics.

pd-upic extends the argument beyond symbolic notation. Visual attributes become active computational parameters, and parent-child context can alter musical behavior: a primitive acquires different consequences according to where it occurs. Graphical objects, temporal containers, and recursive relationships can thus share Pd's dataflow without being reduced to flat messages.

5. Web Deployment through pd4web

Implementing *dddd* entirely in Lua also removes a restriction to desktop Pd. Through *pd4web*, Lua-based objects can be compiled to WebAssembly and executed directly in modern browsers; the same *dddd* abstractions used in desktop patches operate without modification on the web.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

Portability changes how these tools can circulate. Interactive applications developed with *dddd* require neither a Pd installation nor additional external libraries from their users. Algorithmic composition systems, educational tools, and research prototypes can be shared through a standard browser while retaining the data structures and programming model of their desktop counterparts. Recent projects developed with *pd4web* already use these technologies to generate harmonic progressions and other symbolic musical structures in the browser, mainly for Music Perception classes.

Practical applications of this ecosystem include *NIS* (Neimog, 2026c), a musical work in which *dddd* is employed to support harmonic transformations between chords. These transformations generate intermediate chords through linear interpolation of pitch values, while gradually adjusting the number of notes between the initial and target chords. Educational applications include interactive music reading exercises (Neimog, 2026d) and exercises for identifying musical modes (Neimog, 2026e). These projects illustrate how browser-based musical applications can support both artistic experimentation and music education, making computational approaches to musical structures accessible beyond conventional desktop environments.

WebAssembly applications may also be deployed as Progressive Web Applications (PWAs), allowing one code base to operate on desktop computers, tablets, and smartphones. Music education, interactive installations, browser-based compositional environments, and research-software dissemination all benefit from this reach. The underlying programming model remains consistent across native and web implementations.

6. Comparative Positioning

Among established computer-aided composition systems, native integration with Pd and web deployment are rarely found together. BACH provides powerful symbolic and recursive structures for CAC, but is tied to Max. OpenMusic offers a mature visual model for symbolic composition, but operates as a standalone environment rather than a Pd-native extension. Other bridges exchange data between Pd and external systems while generally retaining a separation between the sound patch and its compositional representation.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

With *dddd*, Pd itself hosts high-level compositional objects. This matters because Pd is already widely used for synthesis, interaction, installation work, pedagogy, and performance systems. Symbolic and graphical structures moving through their own dataflow reduce the need to divide a project between a CAC environment and a separate sound-processing environment. Lua and pd4web further allow the same work to become a browser-based compositional tool, educational instrument, or distributable research prototype.

The point is not to reproduce every feature of BACH or OpenMusic. The contribution is infrastructural: general support for CAC within Pd, while retaining the portability and openness valued in experimental music practice. Its particular position is Pd-native like an external library, yet deployable on the web like a contemporary interactive application.

7. Limitations

The abstraction proposed here is not without cost. The pass-by-value behavior of *dddd* depends on recursively copying hierarchical Lua tables when objects are transmitted between receivers. This preserves Pd's deterministic message semantics and avoids shared mutable state, but it increases memory allocation and can impose overhead for very large structures, dense control-rate activity, or patches that duplicate many complex objects in parallel. In practice, this overhead is acceptable for the symbolic and graphical workflows discussed in this article, but it should not be confused with a zero-cost abstraction.

Performance also depends on the Lua runtime available in the target environment. LuaJIT can substantially improve recursive table manipulation and parsing, but its availability is an implementation dependency rather than a purely musical feature. Systems that rely on the standard interpreted Lua runtime may require more careful profiling, especially when large symbolic datasets or deeply nested graphical structures are generated in real time.

There is also a usability limitation. Although composers using *dddd* do not need to manipulate Lua tables directly, developers of new *dddd*-based objects must understand the Lua layer, the runtime type conventions, and the object communication protocol. For Pd users who are comfortable with visual patching but unfamiliar with Lua, debugging may be more complex than debugging a

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

conventional Pd abstraction. Error reporting through the lightweight type system helps, but it does not eliminate the need for Lua literacy when developing new libraries.

Finally, the current distribution strategy couples *dddd* to the *xlab* library in order to guarantee a single shared runtime registry. This avoids incompatible independent registries and makes inter-object communication reliable, but it also means that the framework's adoption is tied to ELSE's installation, release cycle, and architectural decisions. This coupling is defensible for consistency, but it remains an important dependency for future development and community maintenance.

8. Final Considerations

Pd should not remain a first-class environment for signals while remaining a second-class environment for computer-aided composition. Composers work with chords, rhythm trees, gestures, and formal structures; carrying these objects should not require unpacking them into floats and reconstructing them through routing networks. *dddd* is a practical step toward that goal, building upon the work done with *pd-lua*.

Across *pd-bhack* and *pd-upic*, high-level compositional entities remain transmissible inside Pd without replacing its dataflow model. Patches can describe musical relationships directly through high-level data abstractions. The practical gain is not only cleaner patching but also a redistribution of compositional attention: less time constructing mechanisms for representing musical objects and more time exploring the musical ideas those objects make possible.

The next stage of the work is necessarily communal. *dddd* will become significant only if composers, Pd developers, teachers, and researchers build beyond the examples presented here: harmonic tools, rhythm systems, notation interfaces, graphical environments, analysis objects, pedagogical applications, and browser-based CAC instruments. Its Lua foundation and *pd4web* compatibility make such expansion technically plausible and easy. Success depends on whether the Pd community treats high-level data representation as an important musical infrastructure.

dddd is an invitation to make Pd a first-class CAC platform, not merely a signal-processing tool with compositional workarounds attached. The architectural question ultimately concerns compositional time: how much of it can remain with musical thought.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

REFERENCES

AGOSTINI, Andrea; GHISI, Daniele. Bach: an environment for computer-aided composition in Max. In: **International Computer Music Conference**, 2012, Ljubljana. Proceedings [...]. [S. l.: s. n.], 2012. p. [inserir].

ASSAYAG, Gérard; RUEDA, Camilo; LAURSON, Mikael; AGON, Carlos; DELERUE, Olivier. Computer-assisted composition at IRCAM: from PatchWork to OpenMusic. **Computer Music Journal**, [S. l.], v. 23, n. 3, p. 59-72, 1999.

BROOKS JR., Frederick P. No silver bullet: essence and accidents of software engineering. **Computer**, [S. l.], v. 20, n. 4, p. 10-19, Apr. 1987. DOI: 10.1109/MC.1987.1663532.

GRÄF, Albert; SCHOEN, Timothy; WESCH, Benjamin. Pd-lua signals and graphics. In: **LINUX AUDIO CONFERENCE**, 19., 2025, Lyon, France. Proceedings [...]. Lyon: [s. n.], 2025. Available at: https://hal.science/hal-05096052.

NEIMOG, Charles Klippel. **dddd: getting started, examples, and API reference**. In: NEIMOG, Charles Klippel. pd-dddd. [S. l.], 2026. Available at: https://charlesneimog.github.io/pd-dddd/. Accessed on: Oct. 30, 2026.

NEIMOG, Charles Klippel. **pd-dddd: a data abstraction framework for Pure Data (Pd) implemented via pdlua**. [S. l.]: GitHub, 2026. Available at:https://github.com/charlesneimog/pd-dddd. Accessed on: Oct. 30, 2026.

NEIMOG, Charles Klippel. **NIS**. [S. l.], [2026]. Available at: https://charlesneimog.github.io/Nis/. Accessed on: Oct. 30, 2026.

NEIMOG, Charles Klippel. **Leitura 1**. [S. l.], [2026]. Available in: https://charlesneimog.github.io/percepcao/leitura/leitura-1/. Accessed on: Oct. 30, 2026.

NEIMOG, Charles Klippel. **Ouvido modal: exercício de percepção de modos e sequências musicais**. [S. l.], [2026]. Available at: https://charlesneimog.github.io/percepcao/modos/ex1/. Accessed on: Oct. 30, 2026.

NEIMOG, Charles Klippel; SOUZA, Rodolfo Coelho de. Enabling interactive music performance through web browsers for non-programmers. In: **BRAZILIAN SYMPOSIUM ON COMPUTER MUSIC**, 19., 2025, Campinas. Proceedings [...]. Campinas: Sociedade Brasileira de Computação, 2025. p.

https://creativecommons.org/licenses/by/4.0/ | https://doi.org/10.33871/vortex.2026.14.12468

NEIMOG, Charles Klippel; SOUZA, Rodolfo Coelho de. OM-pd: API between PureData and OM-sharp. In: **Congresso da Anppom**, 32., 2022, Natal. Proceedings [...]. Natal: Associação Nacional de Pesquisa e Pós-Graduação em Música, 2022. Available in: https://anppom.org.br/anais/anaiscongresso_anppom_2022/papers/1165/public/1165-5675-1-PB.pdf.

ABOUT THE AUTHOR

Charles K. Neimog is a composer, professor at State University of Paraná, and postdoctoral researcher at the University of São Paulo, where he earned his PhD in Music. He develops open-source tools including *pd4web*, *OpenScofo*, and *py4pd*, with a focus on mobile phones for music creation and education. His work connects computer music, composition, and philosophy of technology, particularly the thought of Bernard Stiegler. ORCID: https://orcid.org/0000-0002-9780-0041. E-mail: charlesneimog@outlook.com.

RESEARCH DATA AVAILABILITY

- Research data is available in a repository.

STATEMENT ON THE USE OF AI

ChatGPT (OpenAI) was used to review manuscript's English, identify unnatural phrasing, and verify translations. All suggestions were carefully reviewed by the author.

https://creativecommons.org/licenses/by/4.0/