MOZ'LIB & PWFORMAX : REWIRING CAC HERITAGE

Julien Vincenot

PhD (defended 2024)
Department of Music
Harvard University
mail@julienvincenot.com

Örjan Sandred

Professor of Composition
Desautels Faculty of Music
University of Manitoba
sandred@umanitoba.ca

Juan S. Vassallo

PhD (defended 2025)
The Grieg Academy
University of Bergen
personal@juanvassallo.com

ABSTRACT

This paper surveys ten years of development on *MOZ'Lib*, an open-source package for Max built around the *bach* ecosystem for computer-aided composition (CAC). Originally a set of pedagogical, entry-level modules for beginners of CAC, it features now a more advanced framework, *PWforMax*, which allows musicians to explore the rich heritage of Lisp-based CAC within Max. The package provides a variety of paths for teaching music creation, experimentation, but also to maintain cultural continuity in digital creativity. After presenting MOZ'Lib's foundations and knowledge base — rooted in the PatchWork family of CAC environments — the discussion focuses on rule-based generative systems. The paper also reflects on the place of artificial intelligence in computer-aided composition practices today, and advocates for a diversity of AI paradigms in computer music. Two constraints solvers, *PWConstraints* and *Cluster-Engine*, whose interfaces have been largely reimagined for Max, are examined in detail. In the last sections, authors discuss some design choices made over the years — in particular to bridge the conceptual and technical gaps between Lisp-based CAC and Max — as well as ongoing and future developments.

1. INTRODUCTION

MOZ'Lib (or simply MOZ) is a Max package developed by Julien Vincenot since 2015. It can be considered a non-official satellite of the *bach* ecosystem[1], created by composers Andrea Agostini and Daniele Ghisi in 2010. Originally not meant for public release, MOZ'Lib has been completely refactored in its current form and fully documented in English as part of Vincenot's PhD research in composition at Harvard University, and freely distributed since 2020[2]. The current version is compatible with all macOS environments (version 10.12 and later, Intel and Silicon), as well as Windows (version 10 and later[3]), thanks to the help of Matteo Marson and Juan S. Vassallo.

Both a beginner-friendly option, and a rather advanced, specialized tool, MOZ'Lib inherits the philosophy of previous computer-aided composition (or CAC) environments : first and foremost, the PatchWork family (including OpenMusic and PWGL), initiated by Mikael Laurson, and of course *bach* itself. Although it is clearly a part of its core, the package's main goal is not to offer a comprehensive catalogue of pre-existing techniques — a form of "canonical" culture of algorithmic music, which can be intimidating or overly prescriptive to newcomers. Instead it hopes to foster a broader, exploratory and holistic approach, focusing on a flexibility of practices (plural), multiplicity of representations, expressivity and extensibility, but also the preservation or continuation of past practices.

Today this package can be divided into two distinct halves, with significantly different purposes : the original MOZ'Lib itself, a set of pedagogical tools designed to explore music writing, creation and computer programming ; and its expert counterpart *PWforMax*[4], which allows to generate and run Lisp code in Max, and thus to work with codebases inherited from the PW family. Among various areas of interest, PWforMax gives access to a selection of rule-based generative systems (i.e. music constraint solvers), that offer uniquely expressive possibilities. As such, it represents fertile ground for anyone looking to experiment with alternate AI directions in music today.

2. CAC FOR AGES 9 TO 99

The original set of pedagogical MOZ modules was commissioned by the *Ariane#* project and Conservatory of Montbéliard (France). It resulted from lengthy

[1] cf. Agostini & Ghisi, 2015

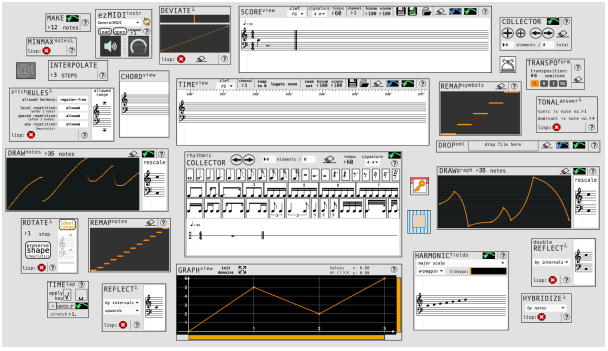
[2] MOZ'Lib is distributed under the GNU General Public License version 3 (GPLv3) and regularly updated.
cf. github.com/JulienVincenot/MOZLib/

[3] Currently, Windows users are required to perform a list of operations manually, including the installation of WSL (Windows Subsystem for Linux) in order to use Lisp functionalities in Max. An alternate solution is being examined, in collaboration with Paulo Raposo, using the Chocolatey package manager instead of WSL to handle SBCL.

[4] The name PWforMax was chosen as a homage to the historic contributions made by Mikael Laurson with PatchWork then PWGL, and particularly his pioneering work on music constraints. The tools are truly missed, and MOZ'Lib is a modest attempt to recapture some of their unique capabilities.

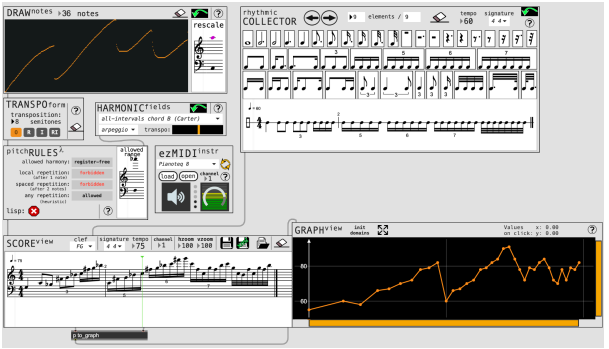
discussions with composer and pedagogist Gaja Maffezzoli, who specialized early in combining music creation and pedagogy for children, first in Florence's Tempo Reale center, then in Montbéliard. Its first goal was to introduce young musicians in training to music theory and instrumental composition with the help of digital tools. The library had to be accessible, appealing, and most importantly, highly responsive. PWGL, which was already endangered at the time[5], was initially considered to develop these tools, but *bach* was eventually chosen as the most promising option.

Instead of proposing a set of rigid, single-purpose tools (typically non-editable Max apps), another choice made early on was to build on the great flexibility of both Max and *bach* to propose a sort of simplified environment, combining ideas like a chain of plug-ins in a DAW. The solution was inspired by the BEAP and Vizzie modules introduced in 2014 by Max 7 : a library of bpatchers, i.e. large boxes with embedded user interfaces, that could be connected to each other like regular Max objects to build more complex scenarios with limited programming knowledge.



To this day, the main library includes more than 30 modules to generate, transform, display and listen to notated music materials, that can be easily drag-and-dropped[6] into a new patch. Between modules, patch cords usually carry simplified (flattened) lists of pitches (or midicents), rhythm (represented as rational numbers), purely symbolic data or other parameters[7].

This almost "baby CAC" approach meant facilitating access to *bach* features for everyone, in a way similar to *cage*[8], trying to reduce as much as possible the distance between idea and result. Considering their inherent complexity and steep learning curve for beginners, a strong emphasis was put on all the typical "end-of-branch" modules, i.e. so-called MOZ Editors. First and foremost, CHORDview, SCOREview and TIMEview (all inspired from similar objects in PWGL and OM), allow to quickly obtain some score representation from a simple list of pitches and/or durations. GRAPHview can instantly display multiple tracks of numerical data (musical or non-musical) as 2D break-point functions[9]. MOZ Editors also accommodate more experimental modules such as METAview, which makes use of automatic patch scripting to combine metered and non-metered notations, a feature unavailable in *bach*. Beyond the intended audience of total beginners of CAC, many of these modules have proven more and more useful in a variety of situations, to accelerate workflow when testing ideas, to display motifs or formal schemes, to quickly control a result as notation or play it with a vst~ instrument.



[5] A sequel to the original PatchWork, created in 1986 by Mikael Laurson and developed with several collaborators at IRCAM, PWGL was developed since 2002 by Laurson together with Mika Kuuskankare and Vesa Norilo, later joined by Kilian Sprotte. First released in 2007, it remained officially in "beta" form until its final known version (RC19, build 342). Despite a dedicated community, PWGL began to wither in 2016. This decline may be attributed to multiple factors, including the lack of a timely 64-bit port, a refusal to open its source code (most likely for a lack of trust following OpenMusic's succession to PatchWork), difficult portability because of a rather unaffordable Lisp implementation (LispWorks), and eventually a lack of funding and institutional support. These challenges made PWGL unusable beyond macOS 10.14.6, and the official website quietly disappeared sometime in 2020. cf. archive.org for the missing www2.siba.fi/PWGL/

[6] In the Max toolbar, a dedicated menu gives access to the palette of MOZ modules (or snippets) arranged in different categories :

• Generators — modules to create simple music entities from scratch by drawing, recording or building up sequences with predetermined options (rhythm cells, harmonic fields, etc.)

• Editors — modules to display music entities according to various modes of representation : flat melodic profile or list of chords (inspired by PWGL's chord editor), sequencer-like chronometric notation (based on *bach.roll*), traditional metric notation (based on *bach.score*) or more abstract (break-point function or graph)

• Transformers — a selection of CAC techniques operating mostly on pitches — transposition, retrograde/inversion, mirroring, distortion, interpolation, non-linear remapping, rotation (including heuristic preservation of shape), basic constraints, etc.

• Utilities — in/out modules and smaller tools.

[7] Many of these modules were in fact inspired by Vincenot's personal experience and workflow in PWGL, re-actualized in *bach*. But unlike PatchWork and its successors, where the programming logic was so-called bottom-up or "demand-driven", MOZ'Lib inherits the same "data-driven" approach as Max and *bach* where data follows the tree of the patch from top to bottom. This logic is in fact quite unnatural to users familiar of previous, Lisp-based CAC environments. Trying to combine to two styles of interaction resulted in some debatable design choices like the addition of "callback" buttons to certain modules, forcing data to traverse the patch again and update the whole tree, for instance when a new module is introduced in the chain.

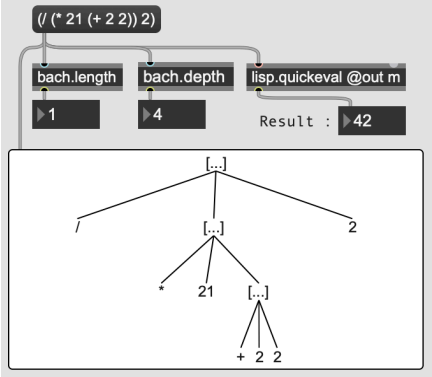
[8] This package was the first addition to *bach* by its creators, followed by *dada* and *ears*. It offers a collection of composition tools — all abstractions that demonstrate the possibilities of *bach* objects.

[9] In our view, this module is just as crucial as the notation modules —if not more so— because it encourages a sense of abstraction in musical processes (in particular for evaluating perceptible directionality) which can benefit both composition and creative analysis.

Among the early batch of pedagogical modules, MOZ Transformers quickly became the trickiest part of the project. One of Vincenot's first goals was to emulate certain PWGL tools by Jacopo Baboni Schilingi, who introduced him to CAC in 2007. Many of these were in fact rather complex Lisp programs, impossible to recreate using only Max or *bach* objects. This was the initial impulse to develop a system that could transplant techniques from the PatchWork family into Max.

3. *PWFORMAX* : RE-OPENING THE PARENTHESIS

The first prototype which would eventually become PWforMax was the result of a collective effort during the 2015 meeting of the international PRISMA research group[10], hosted at IRCAM, notably by Örjan Sandred, Hans Tutschku and Johannes Kretz. It relies on a simple mechanism : a piece of Lisp code (or s-expression) is initially produced in Max, then evaluated outside of Max by SBCL[11] via shell commands[12]. After the evaluation is completed, the result is returned back to Max with usually imperceptible latency, then further developed or displayed with regular *bach* interfaces.

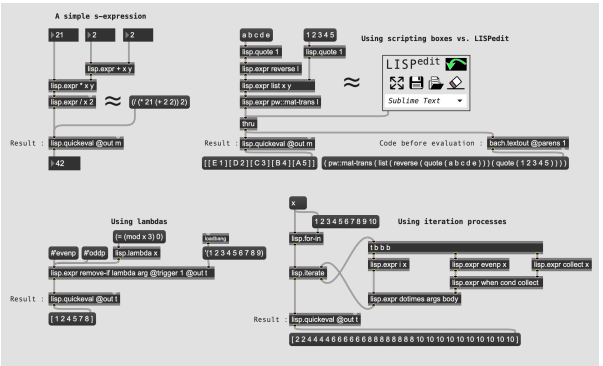


In the earliest versions, standard Max messages were used to generate the Lisp code dynamically, in combination with the *sprintf* object. But this approach proved unviable due to a number of issues : Max's hard limit on message length, difficulties handling nested parentheses, and the accumulation of new symbols that inevitably led to bloated RAM usage after a few hours. The solution came from a critical insight by Jean-Baptiste Barrière : the *bach* ecosystem relies on a custom data structure, the *llll* (or Lisp-like linked list), which is virtually indistinguishable from s-expressions, and has no inherent limitations on length or depth. Using *bach* objects to generate Lisp code on the fly made PWforMax fully stable and operational[13].

Originally, the feature allowing to use Lisp in Max was hidden in some of the original MOZ Transformers that were marked with a lambda symbol, such as Deviateλ or Hybridizeλ, based on the JBS-CMI library. The most advanced modules in this category, namely Rotateλ and PitchRulesλ, both made use of Laurson's PMC engine (a constraint solver which will be further discussed below), allowing to apply a combination of true/false and heuristic rules to pitch materials.

It was only after a few years testing these features in a variety of scenarios, that PWforMax was officially announced and fully documented as the advanced core behind MOZ. It is in fact this addition that justified distributing MOZ'Lib as a public package. For the overwhelming majority of the software industry, including most computer music specialists, Lisp is usually dismissed as a “dead tongue”, a relic from the past. Yet, over the years it became obvious that a dedicated, albeit narrow, community of users remained strongly attracted to its possibilities[14], as a way to modernize older (or even lost) CAC techniques, but also to apply them in real-time performances or installations.

Unlike the few other past and present Lisp implementations in Max[15], the initial design choice behind PWforMax — Max / *bach* handling code and SBCL evaluating it separately— has led to develop two complementary approaches for music Lisping, familiar to all users of PW(OM)GL : coding *and* patching.



[10] This particular PRISMA workshop, hosted at Ircam, was triggered by a hypothesis from Vincenot, after comparing performances in PWGL and OpenMusic (both developed with LispWorks) against SBCL. It turned out the two CAC environments came with strict limitations on heap size in recursive functions. This prevented users from performing heavier calculations, such as analyzing a large corpus of data, or generating longer sequences with a constraint engine. A (rather absurd) idea was proposed : to offload these more demanding calculations outside of PWGL or OM, letting SBCL run in the background when necessary. This was prototyped directly in Max by PRISMA members, and later adopted for MOZ’Lib.

[11] SBCL (Steel Bank Common Lisp) is one of the most popular Lisp implementations today. It is open-source, cross-platform, and offers great performances. cf. sbcl.org

[12] Terminal commands are supported non-officially in Max thanks to Jeremy Bernstein's *shell* object.
cf. github.com/jeremybernstein/shell

[13] For more technical details about the implementation of the system, cf. Vincenot, 2017

[14] As of today, the core audience for PWforMax comes primarily from the nebula of Lisp-based composition tools. This includes the PW family of course — PatchWork, OpenMusic and PWGL, which all share a similar design philosophy and culture. It also extends to independent projects like the proprietary environment Opusmodus. Every now and then, a newcomer who discovered modern CAC with *bach* gets also seduced by the “old ways” and starts to explore the unique possibilities of Lisp for composition, in particular constraints.

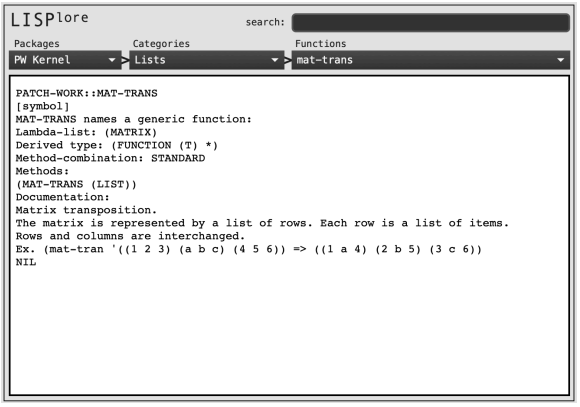
[15] Notable implementations are Brad Garton's *MaxLispj* based on ABCL (2011), Alex Graham's *Lisper* based on Clozure CL (2011-2013) and Iain Ducan's *SchemeForMax* based on s7 Scheme Lisp (2022). They all run directly in Max (unlike MOZ), and essentially rely on Max messages to interact.

The main entry point to evaluate Lisp code is the *lisp.quickeval* object. It receives an s-expression — either as a Max message or a *bach llll* — and automatically handles communication with SBCL. After evaluation, the result is read back into Max as a *bach llll* pointer by default. Users may also choose to have the output translated into text for immediate readability.

Beyond simple Max messages, "pure Lisping" in text form was facilitated by introducing *LISPedit*. The module can be linked with any favorite Lisp editor or IDE, enabling features like syntax highlighting, parenthesis matching, auto-indentation, and more. LISPedit can send a piece of code directly to evaluation, but it is mostly useful to prepare "code snippets" — a feature of lisp.quickeval which allows to include new function definitions in the environment. This is the easiest way to enrich the "kernel" of Common Lisp and PW functions that is provided with MOZ'Lib (more on this in the next section).

The other approach to generate Lisp code in MOZ relies on the visual programming paradigm itself, which reshaped the landscape of computer music in the mid-1980s, first with Miller Puckette's *Patcher*, and shortly after with Mikael Laurson's *PatchWork*. Despite the fact that, under the hood, these are fundamentally different programs (as will be further discussed below), PWforMax gladly takes advantage of the visual analogy between them. In practice, attempts to reproduce the behavior of a PWGL patch would often result in a Max equivalent with a very similar shape and vocabulary. This proved to be a great time-saver when porting older patches to the new environment. In order to streamline the whole process, a small suite of abstractions (called "Lisp-scripting boxes") was created to compose s-expressions (*lisp.expr* and *lisp.lambda*), to perform loops (*lisp.iterate*), help debugging with the Max window (*lisp.print*) or quote an ordinary Max list before it enters the Lisp domain (*lisp.quote* or *lisp.backquote*).

To facilitate as much as possible the documentation about Lisp directly within Max, a special helper tool was designed, LISPlore, which gives access to docstrings of all the PW "kernel" functions provided out-of-the-box with MOZ'Lib. This was also meant as a partial solution to the "black box" effect experienced by several users of the early prototypes. The knowledge base of PWforMax — a pre-compiled SBCL .core, cf. next section — being particularly opaque, it was indispensable to provide a simple way to browse among its most common and useful functions. In addition, the PWforMax Overview also provides a Lisp introduction pdf, as well as a few suggestions for accessible textbooks and online references, for all those interested in learning the language.



4. PATCHWORK FAMILY (AND FRIENDS…)

In order to program some of MOZ'Lib's pedagogical modules, it was necessary to access a range of functions from PWGL libraries[16]. These higher level CAC tools, however, did not rely only on the ANSI Common Lisp standard lexicon. They were also built with numerous "helper functions" specifically written for CAC purposes, many of which had been passed down from PatchWork into both OpenMusic and PWGL. Among them are some iconic examples such as *flat-once*, *x-append*, *posn-match*, *mat-trans*, etc., which remained virtually identical across all three environments. Most of these essential functions were eventually borrowed from OpenMusic's sources[17], simplified into vanilla Common Lisp, and included in the PWforMax kernel[18].

Hundreds of functions are readily available under the :PW package name, although most of them remain poorly documented in Max for lack of time or necessity. They are also rarely mentioned explicitly in PWforMax tutorials. The aforementioned LISPlore module addresses this gap : it allows to navigate some of the known packages included in the .core file via dropdown menus, organized into explicit subcategories. It also features a search function, allowing users to retrieve the

[16] When discovering a new CAC environment, it usually does not come as a "naked" tool. A few years after its release, PWGL already offered a couple dozen libraries, often passed hand to hand between composers, researchers, etc. Many users learned to use the environment — and in some cases, learned to compose — by experimenting with these higher-level tools, discussing with their authors, studying their code, etc. Most importantly, users built upon each others' work : porting the most recent tool in a chain often implies to carry its whole lineage. For reference, below is a public repository gathering over 50 libraries which were circulating in CAC circles until PWGL's development was discontinued.
cf. github.com/JulienVincenot/PWGL-community-library/
A similar page gathers libraries for OpenMusic.
cf. openmusic-project.github.io/libraries.html

[17] We would like to thank again the Music Representations team at IRCAM, in particular Jean Bresson, Gérard Assayag and Karim Haddad, for keeping these accessible and well documented over the years. This entire project would not have been possible otherwise.

[18] The "save-lisp-and-die" function in SBCL allows to save an entire Lisp environment, including declared packages, functions and variables, as a single binary file called a core. This .core file can be easily reloaded using the --core flag when calling SBCL from the terminal. The main advantages of this method is the portability of generated .core files (within a same architecture), and instant startup even with dozens of libraries preloaded.

docstring of any function declared in the environment—even those not listed in the menus.

Over the years, a number of libraries from OM and PWGL have been ported to MOZ'Lib using the same method, often in collaboration with their original creators. The procedure requires little effort, even for casual Lispers. It allows not only to preserve some previous work, programs and sketches in a more enduring environment like Max, but also to transform radically how users interact with them. The new real-time context calls for much more reactive types of interfaces (again, with minimal latency) and naturally broadens the range of artistic contexts in which to use these advanced techniques. This includes the whole world of audio processing, synthesis and analysis in Max (similarly to OpenMusic) but also video, lighting, sensors and networks, etc. It also brings functionalities that may seem trivial at first, yet prove transformative, such as sophisticated systems to handle presets as meta-parameters.

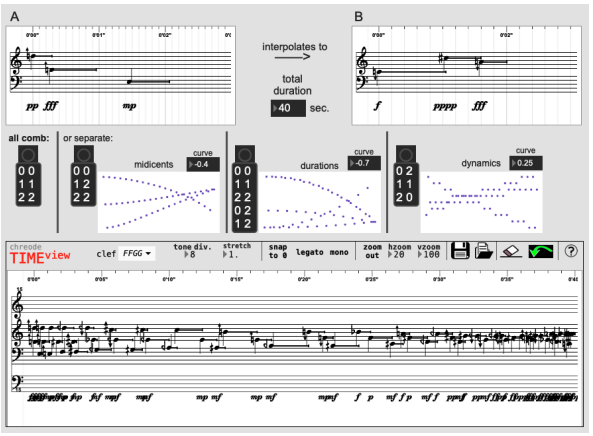
Until very recently, the only durable way to add a new library to PWforMax was for its creator to hardcode it directly into the MOZ'Lib package. As a result, Vincenot adopted a form of "curated" approach to libraries, highlighting different approaches to music thinking through CAC[19]. Some of them, labelled "guest libraries", are showcased as part of the PWforMax project, and receive special attention. Not only is their source code adapted to run in the new environment, but custom interfaces are also designed on top of key functions to better fit the Max context. They are also fully documented, with plenty of example patches available in the package overview. A more sophisticated and appealing UI aims to attract a broader audience, but also to make the underlying concepts of these tools more accessible to newcomers.

Guest libraries in PWforMax include so far :

• the reconstitution of two historical libraries, Jean-Baptiste Barrière's *Chréode* and Kaija Saariaho's *Transkaija*[20], both developed in the 1980s with CHANT/FORMES — a text-based CAC environment created at IRCAM ;

• Frédéric Voisin and Jacopo Baboni Schilingi's *Morphologie*[21] ; Baboni Schilingi's *CMI*[22] and *Profile*[23], all three originally created for PatchWork in the late-1990s, and later ported to OM and PWGL ;

• Baboni Schilingi's *JBS-Constraints*, created for PWGL in the mid-2000 (see more in the next section).



As a tool clearly oriented towards pedagogy, MOZ'Lib needed to find a good balance between reliable transmission of ideas — i.e. not constantly reinventing the wheel — and yet not trying to aim for exhaustivity, but rather focus on quality. The project is not only about the perpetuation of some abstract culture, but bringing forward again the embodied practices of musicians who developed those systems, grew with them, augmented them, and often tried to propose alternatives to canonical recipes of algorithmic composition[24].

PWforMax cannot pretend to contain all the "flavors" of Lisp-based CAC or algorithmic composition. In comparison, OpenMusic remains the largest repository of historical knowledge available. It has been actively maintained for decades at IRCAM and has the most vibrant user community still attached to the PW family. But after years as committed PWGL users, we now clearly favor the combination of Max and *bach* as the most modern CAC environment available today, in particular in terms of accessibility and pedagogy. Yet, no one can tell how this minuscule corner of computer music will evolve in the next decade. Losing PWGL has been a rather traumatic experience for a part of the CAC community, and it did

[19] Most of the libraries included so far were created by former teachers and friends from the PRISMA group, which already formed a non-negligible part of the PWGL community.

[20] This project was realized in collaboration with Barrière with early versions of MOZ. Both libraries were originally written in *Le Lisp*, a dialect now considered outdated. Reconstitutions required plenty of hypotheses from sources — hardly readable and unexecutable — compared with intermediary versions in PW(OM)GL, archived documents, and of course Barrière's own memories.

Chréode uses a palette of bpatchers to generate Lisp code dynamically, which the user must assemble following a given syntax. The final code is evaluated on the fly, and results can be observed as graphs or bach scores in real-time. cf. Barrière, 1984

Transkaija is a great example of the "debate" around music metamorphosis in early days of CAC. Saariaho takes a more subtle approach than the typical interpolation — i.e. regular sampling of a line between two values. It allows to transition across multiple parameters in parallel : chords, melodies, but also durations and dynamics. One can set explicitly which elements in one entity should point to in the other, thereby generating original scenarios. cf. Saariaho, 1987

[21] *Morphologie* is dedicated to analysis, classification, and reconstruction of numerical / symbolic musical sequences. It supports applications ranging from isolated transcriptions to large-scale corpus analysis, depending on how data is encoded. Designed to bridge physical and cognitive representations, it aims to serve both musicological and compositional purposes.
cf. Baboni Schilingi & Voisin, 1999.

[22] *JBS-CMI* is a collection of techniques for generating and manipulating musical structures, in particular matrices, complex permutations, groupings and segmentations.

[23] *JBS-Profile*, initially written in collaboration with Mikhail Malt, focuses on controlling the contour of any kind of parameter, including melodic profiles, with a variety of geometric transformations : perturbations, reflections, derivations / integrations and interpolations.

[24] In that vein, see a discussion around the idea of interpolation vs. musical metamorphosis. cf. Vincenot, 2016

not improve confidence in institutions to support such fragile projects in the long term. Two options are without a doubt better than one.

This reality reinforces the necessity to cultivate a form of *technodiversity*[25] within CAC approaches. Environments like OM or Max/*bach* (with the help of MOZ), cannot make the economy of a critical perspective on their own apparatus of preservation and passing of knowledge. For the same reasons, computer music in general, and CAC in particular, must also acknowledge what alternatives exist and have existed within the historic label of "artificial intelligence" and what they can offer to music creators. More than anything, we must avoid promoting a single, dominant paradigm of AI as much as possible.

5. NO-BRAINER : AI FLAVORS YOU CAN'T RESIST

In this bigger section we would like to give a panorama of generative constraint systems showcased in MOZ'Lib. We hope to communicate why we believe they remain a powerful tool for music creation and CAC today, in contrast with more recent trends of ML-driven generative paradigms. We will not go into too much technical depth for this comparison, which would go beyond the scope of this paper. Instead, we will focus on what we can actually speak about, which is interacting closely with algorithms to compose music.

For readers less familiar with the subject, it is worth recalling that rule-based approaches are historically rooted in (symbolic) AI research[26]. Unlike the most visible trends of AI today, they do not rely on large-scale data, but instead on logical properties and relations, combined with the knowledge (and craft) of humans working with them. This is particularly valuable in music, where massive, annotated datasets are rare or non-existent[27], but also for those interested in creative tools that go beyond the statistical reproduction of pre-existing materials. It can also become a vital alternative for creators concerned about the ethical, environmental, and cognitive footprints associated with current generative AI approaches.

In his 1996 thesis, Mikael Laurson presents his approach of music constraints in those terms : « When using a procedural programming language, such as C or Pascal, or a functional language, like Lisp or PW, the user has to solve a problem in a stepwise manner. […] Descriptive (declarative) languages, like Prolog, offer an alternative way to look at this problem: instead of trying to solve a problem step-by-step, the user describes a possible result with the help of a set of rules. It is then up to the language to find solutions that are coherent with the descriptions. This approach is probably more natural for individuals with a musical background.[28] » Constraints solvers like Laurson's use a backtracking search engine to explore possible solutions sequentially. At each step, the engine picks a value from a given domain, and retains the ones that satisfy the given rules. If a logical conflict arises, the system backtracks to earlier choices and keeps exploring alternative paths—much like a composer testing and refining ideas by hand. This recursive mechanism allows to efficiently eliminate invalid options, and gradually converge towards a satisfying outcome.

This is quite different from more “decisive”, traditional approaches to algorithmic composition. There, we usually focus on well-defined recipes designed —or repurposed— to generate musical results with certain properties and internal relations. A composer may choose, for instance, an implementation of *l-systems*, in order to generate certain typologies of rhythmic texture. On the opposite end of the spectrum, ML-driven (data-based) and constraint-driven (rule-based) offer a very similar mode of interaction to artists, who can focus on describing desired properties or relations they wish to find in a resulting entity, whether it is through code or natural language prompts.

Algorithms enabling such transfer from description to result often fade into the background, so to speak. They can almost be perceived as neutral or agnostic — a necessary illusion for the user, but an illusion nonetheless. Different constraint algorithms may have different purposes and limitations, due to architectures of varying degrees of complexity. And therefore they may have, potentially, very different “flavors” in usage. In the context of PWforMax, no matter what constraint engine is chosen, users need to express what they want to obtain either as truth values (true/false rules) or as numerical values (heuristic rules). In PWforMax this is done by way of Lisp code or graphically, with Lisp scripting boxes[29].

We believe these techniques constitute a great value not only as a specific tool to use, but as a process to live, and learn with. By that we mean, learning to understand what it is that we are looking for, learning to describe it, to create a conscious “transduction” between some abstract wish and the very concrete result of a music score — and keep interacting with that score, through multiple evaluations and refinements of the rules. When composing music with computers, the speed of interaction does matter, so that the flow of

[25] The concept was proposed by philosopher Yuk Hui : « Instead of a universal history describing one technology with various stages of development, we can step back for a moment and instead describe technological development as involving different *cosmotechnics*. I call this *technodiversity*. Here, we must revisit the question [...] in terms of systems of knowledge. Michel Foucault [...] understood them as ways of life — ways of sensing and ordering experience, producing in turn certain forms of knowledge. » cf. Dunker, 2020

[26] cf. Norvig, 1992

[27] By this we do not mean only datasets such as MIDI-based transcriptions (mostly relying on pitch and rhythm), but the whole spectrum of notated music to be performed by human instrumentalists. In recent years, we have often heard lamentations, among colleagues in computer music, about the lack of annotated data in particular — i.e. music paired with descriptive, musically meaningful labels — compared with the visual domain. Yet we see this not as a shortcoming, but as an opportunity.

[28] cf. Laurson, 1996, p.147

[29] For future updates, we hope to develop a system to help users generate rule code in Lisp based on some natural language descriptions, potentially with open-source, specialized SLMs ran locally. This could potentially improve the learning curve for Lisp learners, but also facilitate access to more casual users who may be reluctant to learn the language otherwise.

imagination is not interrupted. Yet, a user may need a few minutes, or even a few hours to write a complicated rule. On the other hand, an engine with well-written rules may be re-evaluated over and over, and produce results extremely fast, and reliably. Rules may also be re-used in very different contexts and projects, with a reasonable time of adaptation. This is clearly a strength when compared with the rather "slow" interaction loop one might encounter with well-known, general-purpose LLMs or prompt-based systems — typically "single use" results, often unusable as is, and with no guarantee of continuity beyond the context window. We would add that, in the context of music composition, mistakes allowed by purely heuristic systems can easily become unforgivable, especially in instrumental scores intended for musicians. Unlike ML-based approaches we have experimented with, constraints solvers remain a strong and expressive compromise between this ideal of music precision and extravagant exploration of the possible.

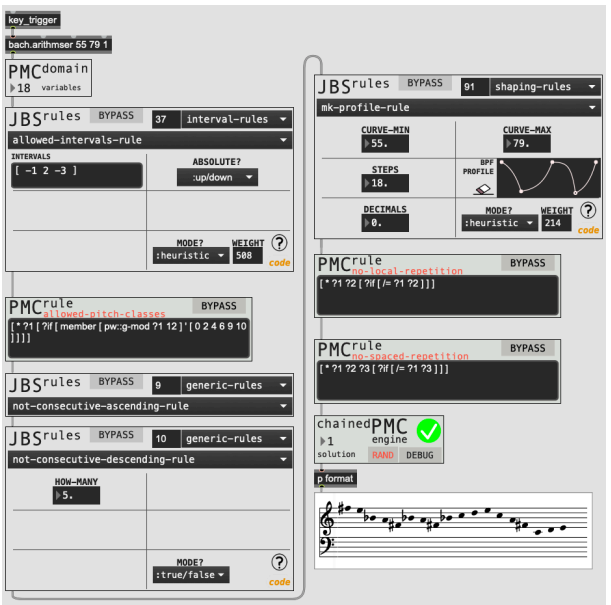
5.1. PMC engine — the polyvalent solver

Not a PWforMax "guest library" per se, *PWConstraints* was included in the MOZ'Lib kernel since the very beginning. It was originally created in 1995 by Mikael Laurson for PatchWork, then ported to PWGL[30]. The library revolves around the PMC engine, a Lisp function searching for potential solutions to a musical problem described by the user with a set of rules. Solutions are generally a sequence of predetermined size for a single parameter at once, i.e. either pitch, rhythm, or any other symbolic representation that the user can imagine. This makes it a robust and highly versatile implementation, which can also be applied to problems that are not explicitly musical.

In MOZ'Lib, the way of patching and interacting with the library has been extensively redesigned to fit the usual practices of Max users. Early in the development, it became clear that the typical "star-shaped", demand-driven layout of constraints patches in PW(OM)GL did not suit the top-down logic of Max — and went against the intuitive approach that was hoped for MOZ. For this reason, Sandred proposed the idea of the "chained" engine, first prototyped for the PMC then generalized with the Cluster-Engine. This implementation of PWConstraints revolves around a small set of bpatcher modules. The candidates (defined in *PMCdomain*) traverse a string of rule modules (*PMCrule*, *PMCheuristic-rule*, *JBSrules*) that act metaphorically like "filters", then reach the *PMCengine*[31] where the computation really takes place.

PMC rules, whether in true/false or heuristic form, are written using a dedicated syntax[32] based on Lisp, While this may seem demanding at first, it remains relatively accessible for beginners and constitutes a great introduction to text-based programming. In practice, writing simple rules only requires learning a few core elements of Lisp syntax, and a small set of basic arithmetic and logical operators.

In order to make PWConstraints more accessible to musicians without a programming background, Baboni Schilingi developed JBS-Constraints for PWGL in 2007. The library contains over 200 rules compatible with the PMC engine, grouped by musical topics such as pitches, intervals, shaping, etc. Each rule is presented as a function that generates code dynamically according to user-defined parameters, making them far more intuitive to manipulate than static code. In Max, the entire library was re-implemented as a single bpatcher named JBSrules, whose interface and parameters are updated dynamically when another rule is selected. With a click, the module returns the code for the active rule with its current parameters. This can help users learn the PMC syntax, and inspire the formalization of their own rules, like a "rule dictionary" of sorts. But it is not just a reference tool — it is also a functional rule module in its own right, that is applied as soon as it is inserted in the chain. A built-in "mode-switch" allows each rule to toggle between its true/false and heuristic variant, making it easier to solve logical contradictions. Even with minimal knowledge of Lisp and PMC syntax,

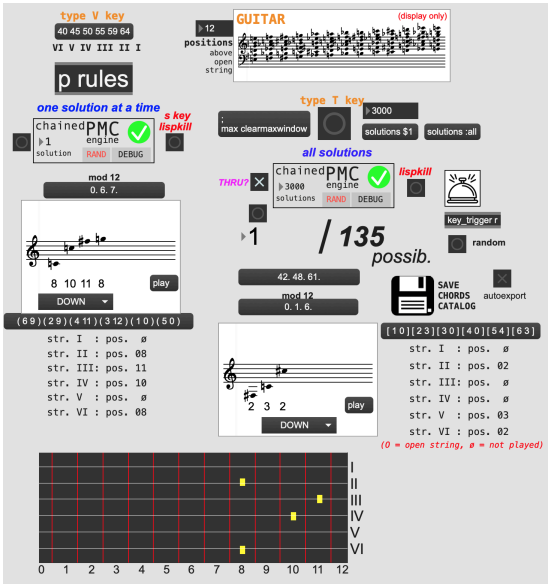


[30] PWGL's sources always remained closed, even though the software itself was distributed freely. This is why the version of PWConstraints that is now included with MOZ'Lib was based on a port realized by Sandred from PatchWork to OpenMusic in 1999, called OMCS. This version may not be the most up-to-date iteration of Laurson's system, but it is fully functional.

[31] Recently added to PWConstraints modules is the *SMC engine* based on Screamer, originally a PWGL library by Kilian Sprotte. Screamer is an extension of CommonLisp that adds support for nondeterministic programming. This alternate engine is compatible with the other PMC modules (and rule syntax), but it operates on different premises. Notably, it allows to generate results with no fixed length, which can be controlled with a dedicated "stop-rule" module.
cf. nikodemus.github.io/screamer/

[32] For a thorough description of the PMC syntax, one can refer to the documentation of MOZ'Lib, as well as the fifth chapter of Laurson's doctoral dissertation, which is entirely dedicated to PWConstraints. cf. Laurson, 1996, chapter 5 (pp.145-186)

users can achieve sophisticated scenarios of music generation by chaining a few instances of this module. For this reason, it constitutes a great entry point to experiment with constraints before studying more complex approaches.



Bringing such system to a polyvalent environment like Max allows to apply constraints not only in the context of notated music — that is the typical domain of CAC — but also in a great variety of scenarios — sound synthesis and processing, live music, interactive installations, multimedia projects, etc.[33] In Sandred's multimedia installation *Sonic Trails*[34], the PMC engine continuously guides the progression of audio/video processing of environmental recordings. Another example is Vincenot's piece 藏镜人—*puppetmaster*[35], where parts of the digital score are calculated during the live performance, some of which by using the PMC engine to re-arrange pre-existing materials or generate new sequences. We can also mention some ongoing research project by Vincenot, aiming to calculate optimal playing positions for string instruments, based on topological and musical rules. These patches have proven quite effective both as a writing aid during the composition process, and an exploratory tool to generate catalogs of materials to be further developed "in time" with the *Cluster-Engine*[36].

5.2. Cluster-Engine — the polyphonic solver

Created by composer Örjan Sandred in 2010 for PWGL[37], *Cluster-Engine* is a system for solving specifically musical constraint satisfaction problems. As its name suggests, it uses a cluster of smaller search engines (similar to Laurson's PMC) that collaboratively search for a solution. Results take the form of a score which satisfies a set of rules defined by the composer.

In music, a melodic line aligns well with the sequential logic[38] of a search engine like the PMC. However, in polyphonic music, multiple voices exist in parallel, making the sequential order of individual notes less obvious. In this case, a search engine must adhere to a specific search order[39]. In Cluster-Engine, the rhythmic content of voices is generated dynamically during the search, and one cannot predict in advance which notes will align across voices to form chords. Cluster-Engine handles search order by assigning a search engine to each parameter of each voice. Each individual engine follows its own sequential order, but it has access to data from the others and can trigger a neighboring engine to backtrack if necessary. Since there is no central or global search engine, the relative search order between voices or parameters is not as critical. Cluster-Engine was specifically designed to

[33] This was already imaginable with *bach*'s own implementation, namely *bach.constraints*. The latter in particular offers some accessible interface where rules are written with *bach* objects within a "lambda loop" structure, typical in that ecosystem. Different engines are available — strong (comprehensive, with back-tracking), heuristic (optimizing a solution based on a global score) and hungry (hybrid approach) — and some musical examples are showcased. In spite of interesting design decisions, and a much better adaptation of the interface to the context of Max, here are a few reasons why we tend to prefer the Lisp-based alternatives : 1) while *bach.constraints* may seem more versatile, our few tests revealed than the PMC engine's performances were superior for identical problems ; 2) we believe the original PMC syntax is actually more accessible to beginners than the *bach* lambda loop ; 3) finally, Cluster-Engine offers far more possibilities for musical exploration than any other engine available, and therefore is a favorite solution for advanced musical problem. Of course, comfort and personal preferences matter most, so we encourage musicians interested in constraints to try different implementations and see which is best for their own use.

[34] *Sonic Trails* (2021) was created in close collaboration with Max Sandred. The installation explores the deconstruction and reconfiguration of recorded audio and video environments using semi-transparent screens and immersive surround sound. The PMC engine controls various parameters of those processes.
cf. sandred.com/sonictrailsinfo

[35] 藏镜人—*puppetmaster* for flute, clarinet, piano and live computer was premiered by Trio de Nice in Cadillac·Shanghai Concert Hall, April 2023.

[36] These instrumental patches were initially commissioned by Jean-Baptiste Barrière in 2022, first focusing on the violin and cello. The same principles have been later applied to the guitar, for Vincenot's collaboration with guitarist Rafaël Carosi.

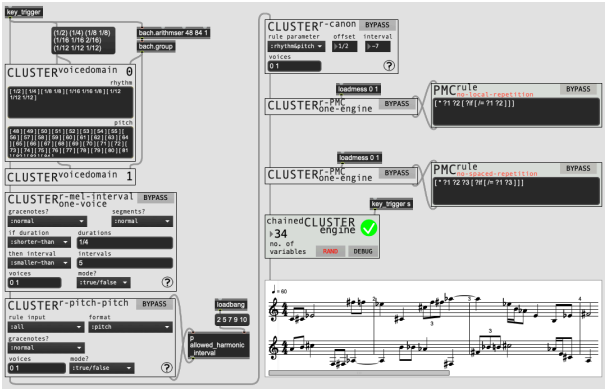
[37] Cluster-Engine was tested with early prototypes of PWforMax, but only completed in 2020. The project is maintained by Örjan Sandred and Torsten Anders as a pure Lisp library, independently from any CAC environment. As of 2025, it has been also ported to Opusmodus (by Anders) as well as OpenMusic and OM# (by Paulo Raposo), and is used by a growing community of composers.
cf. github.com/tanders/cluster-engine

[38] A fundamental requirement for a search engine is that it must explore possible solutions sequentially. To ensure that a valid solution is eventually found, "backtracking" (i.e. solving logical conflicts identified in a sequence) must occur in the reverse order of the search.

[39] In Laurson's *ScorePMC* — a variant of the PMC available in PW(GL) — the engine searches for pitch values within a predefined polyphonic rhythmic structure. The search proceeds in chronometric order, traversing all notes from lowest to highest voice — following the assumption that bass notes influence the pitch choices of upper voices — and giving priority to longer note values. This approach introduces a trade-off : adjacent pitches within a melodic line may be separated in the search sequence, which can reduce the system's efficiency for melodic rules but enhances its effectiveness for harmonic ones. cf. Laurson, 1996, pp.213-215.

improve search speed — in some cases, it delivers results up to 600 times faster than earlier versions[40].

A single voice is represented by two sequences : one for pitches and one for durations (including rests). An additional sequence represents the metric structure of the score, shared by all the voices. Cluster-Engine brings a lot of flexibility when defining the initial domain or search space : candidates can be either single values or motifs, i.e. predefined sequences of durations, pitches or chords, intervals, etc. A carefully designed domain can have a strong impact on the result, even before adding a single rule. In MOZ'Lib, the *CLUSTERengine* interface, reimagined for Max, is made of a large collection of bpatchers. They further develop the idea of the "chained engine" introduced with the PMC. One can build a functional generator of polyphonic scores by chaining any number of "voice-domain" modules (each dealing with rhythm and pitch candidates for a given voice), an optional "metric-domain" module, and the engine module itself.



Rules are easily inserted between the domains and the engine, and can be written in a great variety of ways. They can affect properties or relations not only in a single parameter of the score, or in a single voice (between pitch and rhythm), but also between a voice and the metric structure, between different voices, etc. Rules usually require two things to work properly : an accessor and a code part. The accessor, or rule applicator, determines which music parameter will be visible to the rule. About twenty accessors or rule-related modules[41] are available to combine. The code part is "the rule itself" so to speak — it checks properties of musical parameters made visible to it by a given accessor. It is written as a Lisp *lambda* — either in text form, or visually by using scripting boxes — and behaves similarly to PMC true/false and heuristic rules. Depending on the accessor's scope — adjustable with various parameters in the modules — incoming data from the lambda will be represented differently, thus changing how the rule can influence the result.

Cluster-Engine has been Vincenot's main composition tool since *silent_data_corrupt* for saxophone and live computer (2014). The Max version was used in a similar fashion, to generate materials for 藏镜人—*puppetmaster* (mentioned above) and *Hyperlinks—asset(s) failed to load properly* for flute and electric guitar (2023-25). The new environment opened up new contexts of application that were previously out of reach. For instance, the engine can now constrain a wider range of musical parameters — not only pitch, rhythm, and meter. This is made possible by the "Multidomains" modules proposed by Vincenot, which take advantage of the multiple "slots" assigned to each note in *bach* notation objects. Vassallo's piece *Elevator Pitch* for cello and electronics[42] made extensive use of this feature. Finally, Cluster-Engine has been integrated into patches for live performance — handling the creation of variations in live electronic responses in Sandred's *Sketches of Shifting Landscapes* for string quartet and live electronics (2024).

6. WEIRD PATCHES & (BILINGUAL) DESIGN ISSUES

Now that we have a broader view of MOZ'Lib and the techniques it offers, we need to address some of its shortcomings in terms of readability, accessibility, and the design of its programming interfaces.

MOZ has remained committed to pedagogy since its inception. In this regard, it shares many similarities with *bach* itself, which is arguably the most heavily documented Max package available. MOZ'Lib's overview gathers to this day a bit more than 300 patches, including "guest libraries"[43]. These aim to clarify technical aspects, to provide some (historical or musical) context, or even to explain some abstract theory attached to certain techniques. This likely reflects a propensity towards complexity inherent to CAC environments : higher level compositional topics

[40] Cluster-Engine is the culmination of Sandred's research on constraints since 1997. It builds on previous music solvers : *OpenMusic Rhythmical Constraints* (1998), then *PatchWork Musical Constraints* (2005). Both OMRC and PWMC relied on a single engine — Laurson's PMC itself — iterating over complex, composite objects including melodic motifs, rhythmic values and meter. The speed increase in Cluster-Engine stems from combining multiple search engines and the use of "back-jumping". This technique allows an engine to detect when a logical conflict arises in some earlier variable, and jump directly back to that point to try another value. By skipping intermediate steps, and erasing only the portion of the solution within the local sequence, the system reduces the need to reconstruct already valid parts of the solution. cf. Sandred, 2021

[41] A few examples of accessor modules are *r-PMC-one-engine* (rhythm or pitch rule on a single voice, backward compatible with PMC rules syntax), *r-rhythm-pitch-one-voice* (relation between pitch and rhythm for a given voice), *r-canon* (pitch and/or rhythmic canon between two voices), *r-pitch-pitch* (harmonic relations between n voices), etc.

[42] Vassallo experimented with "Multidomains" mainly to constrain the temporal behavior of the cello's bowing position and pressure, dynamics, articulation, tremolos and vibratos. cf. Vassallo, 2024

[43] Just as reading all *bach* tutorials and help files would be counterproductive, we do not encourage anyone to read the MOZ Overview from start to finish. Instead, users should find precise information on any topic exactly when they need it. Here, MOZ takes direct inspiration from *bach*'s "Help center", meant to be kept open on the side, like an encyclopedia. While *bach* offers a search function based on a limited set of keywords, MOZ draws on libraries with potentially uncommon vocabulary. Therefore a grep-based search was implemented, letting users find tutorials containing any term.

often require more verbose tutorials than the average Max audio-processing library.

Early design choices continue to shape the patching style in MOZ, which may seem unconventional compared to typical Max/*bach* patches. The modules' interfaces were designed to be as “eye-friendly” as possible, but they occupy more screen estate than most UI objects, which often requires adjusting patching strategies. To improve navigation and readability in complex patches, most modules include a “collapse” feature that shrinks them into a more compact form — either by clicking their title or sending a message.

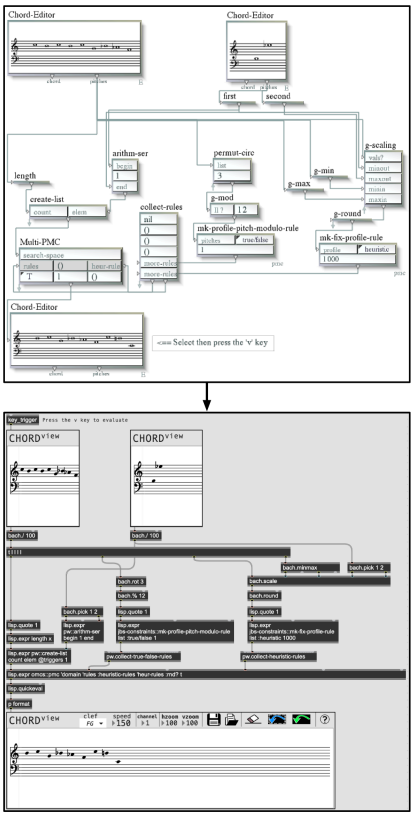
In terms of their inner workings, MOZ modules are as “vanilla *bach*” as can be[44]. Just like *cage*'s abstractions, users are encouraged to use the tools also as patching references for building their own systems. This leads to a known issue : early in the development, it was decided to let users choose their favorite mode of initialization (preset, *pattrstorage*, dictionaries, etc.) if they need one. As a result, adding several *loadbang* or *loadmess* to modules may be required, a reason why MOZ patches often seem a bit more cluttered visually.

Another issue is the apparent contradiction in Max between objects as *blueprint* and objects carrying some sort of memory. In the PW family, the full state of a Lisp function or UI object (like a score) is always saved with the patch, and not re-initialized when re-opening the patch. Things are different in Max, where every object comes with dozens of attributes — hundreds for certain *bach* objects — which complicates the visibility of what is memorized and what has to be handled manually by the user. This may seem trivial, but can have a very direct impact on compositional practice. In PW(OM)GL, it is common to duplicate an object, say a score, and move it aside in the patch before evaluating further, just like one might naturally move paper sketches around a desktop. But Max tends to resist this mode of interaction, and only certain objects like *bach.roll*, *bach.score* or *bach.reg* — with an “embed” attribute — can do this. Max abstractions and especially MOZ bpatchers cannot carry an internal memory elegantly when duplicated — without some heavy preset mechanism — and instead a brand new object is instantiated. It is to remedy this issue that certain “tricks” like the *MOZsafe*[45] module were imagined.

Attempting to create an intuitive, consistent bridge between the PW family model (bottom-up, demand-driven) and Max’s (top-down, event-driven), had a strong influence on what a typical PWforMax patch might look like. Once again, a key reference was *bach* itself, whose creators deliberately steered closer to Max than, say, OpenMusic[46]. While early MOZ modules leaned heavily into PW(GL) heritage, PWforMax orbits around an even stronger model : the Lisp language itself. This has led to some idiosyncrasies in the context of Max, like the need to add a "lisp.quote" before a Max list or *bach llll* enters the Lisp domain, or making chains of objects connected only via their second or third inlets, yet remaining functional.

An issue of readability we already mentioned is the “black box” effect. Patches made of objects generating bits of Lisp code can obscure the role played by a given abstraction, making patches more difficult to write and debug. While patching in PW(OM)GL already looked very similar to Lisp represented as a patch tree, each box could return its intermediate result with a single keystroke. Code generators in MOZ’Lib return nothing but nested code — evaluated later down the line — which makes patching more cryptic. This motivated the creation of *lisp.quickeval*, to be able to evaluate Lisp code anywhere in a patch more easily[47].

Likewise, when copying a patch from PW(OM)GL, or imagining a new one from scratch, a common issue is deciding when to switch from Max-*bach* to Lisp. This typically requires juggling between identical or near-



[44] With the exception of Bernstein's *shell* object — the only external included — MOZ uses exclusively pure Max or *bach* family objects.

[45] *MOZsafe* is a snippet that contains an embedded subpatcher, not an abstraction, therefore it carries its own memory after duplication. Its interface allows to “lock” or “unlock” parts of a patch on demand, an idea inspired by a similar feature in the PW family. This can be used, for instance, to prevent upper sections of a patch to erase some lower result, and to provide some intermediate initialization value.

[46] This is very visible in their choices for objects naming, consistency for instance with *zl* (a lower-level list processing family of objects in Max), attribute logic, or the way they conceived *bach*-style “lambda loops”. However, Ghisi and Agostini have often admitted this wish was not always entirely fulfilled, in particular concerning notation objects. cf. D. Ghisi & A. Agostini, « A compendium of *bach*'s shortcomings as it turns 10 », SMC 2020 Virtual Conference. Video available at : youtube.com/watch?v=_W8ghhJcTz8

[47] As this paper was being finalized, a new object was added to PWforMax : *lisp.antenna*. It allows to display the result of any scripting box used in a patch (*lisp.expr*, *lisp.iterate*, *lisp.quote*, etc.), as long as the user hovers the mouse over that object. This feature brings usage even closer to the PW experience, although it still requires a trigger to happen higher in the patch for it to work — typically via a simple key stroke.

equivalent functions in both worlds — *bach.trans* vs. *pw::mat-trans*, *bach.join* vs. *pw::x-append*, etc. A good practice to follow, in order to keep patches readable, is trying to rely on *bach* alone for as long as possible, until "something" simply could not be done without Lisp, or when switching would lead to a cleaner solution[48]. From that point on, the patch should remain in the Lisp domain until a full expression is evaluated. The result would appear as a *bach llll* or a simple Max list, then the patch would resume to pure Max/bach — possibly returning to Lisp again later if needed.

This leads us to what is arguably the biggest issue when programming with PWforMax : switching back and forth between the Max scheduler and *asynchronous* processes in SBCL. One cannot always guarantee when a Lisp result will be returned — especially with more complex, constraint-based processes. Consequently, even the most basic rule of Max dataflow, i.e. a trigger's outlets fire from right to left, can become unreliable[49]. Another very concrete problem arises when a user wants to take a portion of a patch into a loop. Iteration is notoriously tricky in Max, but *bach* can handle it gracefully with a *bach.mapelem* or combining a *bach.iter* and *bach.collect*. Yet in PWforMax, boxes only generate code, and therefore *bach* iterators are unusable. Instead, loops need to happen on the Lisp side. To address this, objects based on the "Iterate" library have been introduced, mimicking the appearance of *bach* iterators, including the use of "lambda loops". This solution has lead to very promising developments for composition in the last year, introducing two new constraint modules, *PMClooper* and *CLUSTERlooper*[50].

7. FUTURE OF MOZ'LIB

As a form of conclusion we will discuss future plans for MOZ'Lib, some of which are already under development. We can briefly mention the collection of pedagogical modules, which has reached a rather stable form in 2019 and is no longer a development priority. Improvements will keep being offered, and new modules are still introduced on occasion, essentially with practical or "quality of life" purposes in mind. The original modules essentially revolved around the two usual suspects of CAC : pitch and rhythm. This limitation may be addressed in the future, at least partially[51]. Yet again, it is not the intention for MOZ'Lib to be comprehensive and relevant on every musical front. Limiting its scope can also serve as encouragement for users to study *bach* more in depth, and start developing their own tools with it.

Since PWforMax has also stabilized as its own ecosystem — after years of testing, we believe it is reliable enough even for live situations — the next chapters will most likely be focused on "content", i.e. extending its knowledge base. New "guest libraries" are already being considered and discussed with their respective authors. The priority is given to high profile libraries that would be the most relevant in the current ecosystem, but also that could benefit from some extra time designing custom interfaces in Max and substantial documentation.

Here are a few libraries which may join the existing collection in the near future :

• Frédéric Voisin's *FV-Morphologie*, a 2010 "reboot" of the 1990s *Morphologie* library, dedicated to analysis, classification, recognition and reconstruction of numeric or symbolic sequences ;

• Torsten Anders' *Cluster-Rules*, compatible with Sandred's Cluster-Engine, concerning various musical scopes (rhythm, melody, harmony, counterpoint, etc.) ;

• Vincenot's library *JV-Components* (2008-2015), various techniques around analysis, supervised interpolation, instrumental constraints ;

• Sandred's ongoing development — codenamed "New Engine" — is a new approach to music constraints. It is similar in scope to the PMC, but allows both absolute and relative values to coexist in the domain definition : pitch and intervals, time positions and intervals of time. Over the course of its development, the engine has become increasingly focused on real-time applications. In particular, it supports the use of live input during an ongoing search, allowing the system to dynamically adapt while retaining the ability to backtrack—up to the point at which it has already produced output for live playback[52].

[48] A third scenario is often encountered by PW veterans, who may simply not know that an equivalent function exists in *bach* or how to call it. Some naming choices can increase confusion for Lispers, for instance the Lisp *member* and its Max counterpart *bach.belong*.

[49] This can lead to rather convoluted patches, for instance : a value that would logically be received last, in a pure Max context (leftmost outlet of a trigger), might arrive before some intermediate Lisp result. To fix this, the Lisp evaluation should always come last, or at least it would need to re-trigger some stored value received Max-side, so that it hits the "hot" inlet at the right time.

[50] The new modules allow to generate series of distinct PMC sequences or Cluster-Engine scores with varying search spaces, sharing a set of identical rule modules — taken in a *bach* "lambda loop". Dedicated UI elements help to "capture" some of these rules' parameters to vary on each iteration step, or to enable some rules only on given steps. Finally, the possibility to store intermediate results between iteration steps (Lisp-side) allows for even more complex scenarios. For instance, while generating sequences A, B, C,… certain rules could access the cumulated memory and make sequences D and F to imitate B and A respectively, or simply allow for smoother transitions from one segment to another. The two modules, still in development, can be seen as "meta-"constraint engines, allowing to generate complex scores with rich formal contrast.

[51] A collection of modules have been developed outside of MOZ in the last year, around a composition project by J.-B. Barrière. These modules are similar to the original MOZ modules, but allow to process entire scores in full, similarly to *cage*.

[52] The engine allows to extend a search indefinitely, while outputting results continuously. This property was used in Sandred's Shifting Landscapes for string quartet and live electronics (2024), where the generated electronic part continuously adapts to the performance.

A significant addition to MOZ'Lib over the last year has been a fully-fledged “user-library” feature[53]. Originally, adding new Lisp libraries to PWforMax could be only done in two ways. By default, those needed to be "hardcoded" by Vincenot in the official MOZ distribution, which is rarely desirable. Advanced users were still able to recompile a new .core file after including their own code to MOZ'Lib's own “sources” folder, but this would prove rather inconvenient with every subsequent update of the package. The newest solution relies on the Max package architecture (containing abstractions, help files, snippets, etc.), by adding a dedicated "sources" folder containing a Lisp library loaded via a dedicated .asd file. A Max patch found in MOZ's extras menu allows to select one or more “MOZ-compatible” packages, then can compile a fresh SBCL .core with a click, including not only the newly added libraries but also the default environment of PWforMax.

A couple of upcoming libraries are already being developed around this model, and will be distributed in the coming months :

• *NeuralConstraints*, developed by Juan Vassallo. This extension for PMC and Cluster-Engine introduces a new type of heuristic rules, influenced by a simple neural network trained on a given music corpus[54].

• *MOZ-Screamer*, a library on non-deterministic programming and music generation, originally developed by Paulo Raposo as OM-Screamer[55].

Since its first prototype in 2015, the development of MOZ'Lib has been slow but steady. It owes everything to the collective efforts from which it stemmed from, in first line the PRISMA research group, but also the community of Lisp-based CAC at large. Beyond its most seasoned audience, MOZ'Lib aims to give better access to composition to anyone, no matter their music background or technical training. More than ever, we believe these tools can help musicians develop a deeper relation with written music and music creation in general. Ten years after its inception, and perhaps for another decade, we hope its various flavors of “good old fashioned AI” for music generation can remain attractive and relevant for the toolbox of the 21st century composer.

[53] Implementing such a feature in a package that is itself a satellite of a multi-package ecosystem — all hosted within a larger programming environment — may all seem quite absurd. Another symptom of the profusion of packages, devices, tutorials and other propositions available in Max since its inception. The sheer density of this offer can easily become intimidating or even paralyzing at times — for both newcomers and veterans. But we hope this is a positive sign of the impact even the smallest (and hyper-specialized) endeavors allowed by such an environment. Most importantly, it may reintroduce in MOZ one of the defining qualities of its predecessors : the sharing of CAC libraries and techniques among composers.

[54] The main principle behind *NeuralConstraints* is to use the “mean absolute error”, returned by the trained network, as a heuristic value for a given rule. As a result, it allows to combine the properties inferred from the neural network with the more conventional, logical rules. cf. Vassallo, Sandred & Vincenot, 2025

[55] cf. github.com/PHRaposo/OM-Screamer

8. REFERENCES

1. Agostini, A. & Ghisi, D., “A Max Library for Musical Notation and Computer-Aided Composition,” *Computer Music Journal*, Volume 39, No. 2, p. 11–27, 2015.
2. Barrière, J.-B., “‘Chréode’: the pathway to new music with the computer,” in *Contemporary Music Review*, 1(1), M.I.T. Press, 1984.
3. Saariaho, K. “Timbre and harmony,” in *Contemporary Music Review*, 2(1), M.I.T. Press, 1987.
4. Baboni Schilingi, J. & Voisin, F., *Morphologie: Fonctions d'analyse, de reconnaissance, de classification et de reconstitution de séquences symboliques et numériques*, IRCAM, Paris, 1999.
5. Dunker, A., “On Technodiversity: A Conversation with Yuk Hui,” in *Los Angeles Review of Books*, June 2020.
6. Laurson, M., *PatchWork: A Visual Programming Language and some Musical Applications*, Studia musica no.6, doctoral dissertation, Sibelius Academy, Helsinki, 1996.
7. Norvig, P., *Paradigms of artificial intelligence programming: Case studies in Common Lisp*, Morgan Kaufmann, 1992. Also freely available at norvig.github.io/paip-lisp
8. Sandred, Ö., “Constraint-Solving Systems in Music Creation,” in *Handbook of Artificial Intelligence for Music*, Springer, 2021.
9. Vassallo, J. S., Sandred, Ö. & Vincenot, J., “‘NeuralConstraints’: Integrating a neural generative model with constraint-based composition,” in *Frontiers in Computer Science*, Sec. Human-Media Interaction, Volume 7, April 2025.
10. Vassallo, J. S., “Exploring Musical Procedural Rhetoric: Computational Influence on Compositional Frameworks and Methods in the piece ‘Elevator Pitch’,” *IJMSTA*, July 01; 6 (2): 1-16, 2024.
11. Vincenot, J., *Placeholders for [failure?]*, doctoral dissertation under the direction of Hans Tutschku and Chaya Czernowin, Harvard University, 2024.
12. Vincenot, J., “LISP in Max: Exploratory Computer-Aided Composition in Real-Time,” in *ICMC proceedings*, Shanghai Conservatory of Music, 2017.
13. Vincenot, J., “On ‘slow’ computer-aided composition,” in *OpenMusic Composer’s Book Vol.3*, Éditions Delatour – IRCAM, Paris, 2016.