LISP in Max:
Exploratory Computer-Aided Composition in Real-Time

Julien Vincenot

Composer, PhD candidate
Harvard University
julien.vincenot@gmail.com

ABSTRACT

The author describes a strategy to implement Common Lisp applications for computer-aided composition within Max, to enrich the possibilities offered by the bach library. In parallel, a broader discussion is opened on the current state of the discipline, and some applications are presented.

1. INTRODUCTION

Computer-aided composition (CAC), as we know it, has been around for decades. Following the pioneers of algorithmic music, the emergence of the first relevant graphical interfaces allowed the discipline to expand in several directions. CAC exists today under a variety of forms, languages and tools available to composers.

This tendency has crystallized around visual programming environments such as *OpenMusic* (*OM*) and *PWGL*, both derived from the program *PatchWork* (or *PW*) developed by Mikael Laurson since the late 1980s [5]. A specificity of this family of programs, but also many others since then[1], is that they were all built on top of the same language: *Common Lisp*[2].

Since the early days of artificial intelligence, Lisp has always been considered a special language, and its popularity is quite uneven among professional programmers. However, the interest among musicians never completely declined. This probably has to do with the inherent facility of Lisp to represent, through nested lists (see Fig. 1), the hierarchical structures of music notation, but also any kind of conceptual encoding. Lisp also brings several assets: its clear and elegant syntax, its efficiency working with recursion (a powerful concept which has many concrete applications in the musical field), and its ability to generate code dynamically through macros.

One could argue that the most recent innovations in interactive systems and real-time sound processing, as well as the development of various ways of representation and control, have pushed such practice to the background of computer music. In a certain way this is understandable, since purely symbolic approaches, mostly dedicated to traditional notation, concern a small population of musicians.

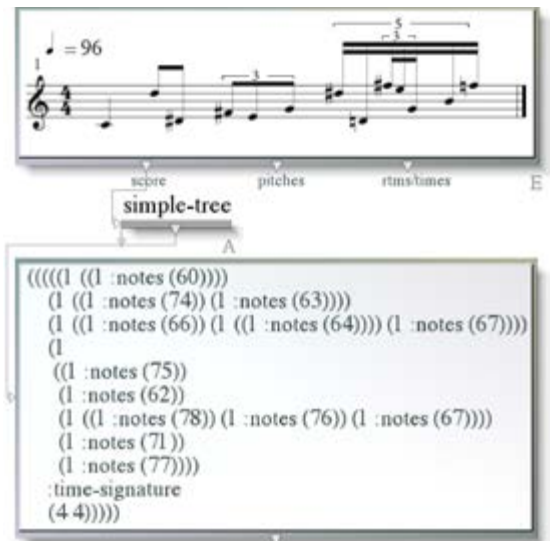

Figure 1. A Score object in PWGL and its inner representation as a Lisp linked-list or tree.

Nevertheless, we can observe today a real interest in renewing those paradigms. The discipline of CAC is going through a phase of transition, with the emergence of new tools oriented toward real-time, interactivity, and performance such as the "reactive mode" in OpenMusic [4], and of course the *bach* library for Max, developed by Andrea Agostini and Daniele Ghisi [1].

All these recent developments are stimulating and we cannot yet imagine the applications that will result, especially for the conception of open/generative scores and forms in composition, and for interactive installations in particular. But most of all, they redefine deeply the access to these tools for the "outside-time" work of the composer, as well as the distance between concepts and realization.

[1] We can mention for instance *Common Music* or more recently *OpusModus*, which both rely on a textual programming interface.
cf. commonmusic.sourceforge.net | opusmodus.com

[2] One of the oldest programming languages, *LISP* (standing for « list processing ») was originally designed by John McCarthy in 1958. *Common Lisp* is one of the several Lisp dialects derived from the original language, standardized in the early 1980s, and one of the most popular with *Scheme*, *Clojure* and *Emacs LISP*.

2. LISP CAC IN REAL-TIME

2.1. Lisp vs. the *bach* library

My whole practice as a composer, during the last decade, has revolved around PWGL. Clearly, I did not use it only as a way to generate and transform pre-compositional elements, but as a working space, favoring a certain slowness and distance from the material, and helping me to think about music in general [7]. Like OM, PWGL can be seen as an environment to program in Lisp *visually*, so I ended up learning to code in Common Lisp as well, and it became to me a very natural medium to manipulate musical ideas.

My first approach of the *bach* library was problematic, considering my practice of CAC so far. While it certainly borrowed some of the flavor of Lisp-based environments, it was clearly designed by its creators with other paradigms in mind, pretty far from Lisp. The proximity with the PW family was only a façade, and I was reluctant to dive deeper into it. However, as I was conceiving the prototype for a set of pedagogical tools that would become *MOZ'Lib* (see below), I got more and more seduced by the huge possibilities of the library, in terms of interactivity and exploratory composition.

It was only later that I realized its hidden power as a language. *bach* relies heavily, as its main data structure, on the well-name "lisp-like linked lists" (or *llls*), which borrows Lisp's parenthesized structure. This is, of course, fundamental, as I already explained, in order to represent musical data in a hierarchical manner. But *llls* also hold an unexpected feature: they can be "hijacked" to generate fully fledged Lisp code in real-time.

2.2. Just another language for Max?

At this point, one might wonder about the interest of bringing an old-fashioned language such as Lisp inside Max. Beyond the ability to write new objects in C with the dedicated SDK, Max users already have access to several embedded languages: Gen~, Lua, Javascript and Java. This has already proven very useful when hitting the limits of visual programming, especially in the case of control structures such as loops and recursion.

As we will see below, offering Max a stable bridge with Common Lisp not only enriches the possibilities of the *bach* library — by 40 years of formalization and compositional techniques, from several esthetics and computational paradigms — but also allows to renew a myriad of historical practices that were restricted to "deferred-time" (non real-time) working environments.

As a matter of fact, Lisp was already accessible in Max through a previous library, *MaxLISPj*[3], developed by Brad Garton. In this approach the Lisp interpreter, *ABCL* (*Armed Bear Common Lisp*, an implementation based on a Java virtual machine), was encapsulated within the *mxj* object and ran directly in Max. There was therefore a risk of slowing down or even crashing Max in case of coding mistakes. Besides, the author pointed out that the stability of his implementation was not guaranteed in case of heavy calculations. Finally, the output of MaxLISPj could not exceed 2000 characters — possibly due to the inner limitations of Max messages — therefore it was impracticable to evaluate complex musical structures. All these reasons justify looking for an alternate approach.[4]

2.3. How to format Lisp expressions with *bach*

In order to generate Lisp code easily with the *bach* library, this article proposes a pretty straightforward method. The starting point is the object *bach.join* (formerly *bach.append*), which is an equivalent to the venerable *list* function in Lisp.

As seen in the figure below, the object needs several arguments and attributes in order to format proper Lisp notation, or *s-expressions*. First, the number of inputs must take into account the number of arguments for the s-expression, including the name of the function itself. So `(+ 1 2)` will require a *bach.join* with 3 inputs.

The attribute @set allows to initialize arguments for the list: here we define the name of the function, of course, but also constants if necessary. The two remaining attributes, @triggers 0 and @outwrap 1, make sure that every input of the box is "hot" (i.e. can trigger an evaluation), and that the resulting list is output in-between parentheses, respectively.

One must understand at this point that *bach* objects, like *bach.join*, do not output a standard Max message, but a sort of pointer called "native" format. This allows *bach* objects to exchange *llls* with virtually no limitation of size or depth.

Figure 2. Formatting Lisp code with *bach.join*

[3] cf. http://sites.music.columbia.edu/brad/maxlispj/

[4] *Lisper*, a library developed by Alex Graham between 2011 and 2013, pursued a similar goal. This approach, unlike *MaxLISPj*, relied on the OSC protocol to establish a communication between Max 4 or 5 and a Lisp implementation such as Clozure CL. Since this work was brought to the author's knowledge only recently, no clear comparison could be established between this approach and the one described in this article. However the use of such network protocol will be carefully investigated for future development, as well as its possible impact, positive or negative, on the ease of use for average users of the system. cf. https://github.com/thealexgraham/lisper

In this example, the list is converted to "text" format, with the *bach.portal* object, only for illustration purposes. Two *bach.join* objects are combined to produce a valid s-expression, ready to be sent to a Lisp interpreter. Changing the values in the number boxes will immediately update the s-expression, turning constants into variables. In the same manner, one could input a symbol or a pre-formatted list with the help of the quote function, preventing the interpreter from confusing them with a variable name or an s-expression with an incorrect function name.[5]

2.4. Evaluations on-the-fly

Obviously, generating Lisp code dynamically has little interest without producing any result. After several attempts[6], a strategy requiring the use of *SBCL* (*Steel Bank Common Lisp)*[7] in parallel with Max was designed. The Lisp interpreter is called in Max through a command-line interface, with the help of the *shell* external[8] on macOS (a similar approach is considered for Windows 10, using *mxj DOShack*).

At this point, the *bach.write* object is used to turn the generated code into a temporary text file, with the *.lisp* extension. This operation is pretty immediate and there is no limit to the size of the script, thanks to the *llll*s format. The script is now ready to be evaluated with the following shell command:

sbcl --script path-of-script.lisp

In order to receive the result in Max, things get more complicated. Indeed, just like MaxLISPj, the shell object returns a string limited in size, which we cannot use for results longer than a few characters. The solution to this problem was to write eventually the result of our code into another temp file, from SBCL itself.

For this purpose a simple Max abstraction, *pw.eval-box*, was created. It is now available as part of MOZ'Lib (see below). Its purpose is to complete our Lisp code just before it is sent to the *bach.write*, by adding a few s-expressions. This supplement of code makes sure that:

- the Lisp package for evaluation is defined by the *in-package* function, allowing to work with a given user library (see below);
- the *random-state* global variable is initialized at each evaluation, preventing random processes to return always the same value;
- eventually, the result is written to another temp file at a defined location.

Whenever the evaluation is finished, the second output of the shell object returns a bang. This is used to trigger a *bach.read* to import the content of the resulting temp file. It is now up to the user to decide what to do with this new data: control real time processes in Max itself, or display the result as standard notation with *bach.roll* or *bach.score* for instance. Of course, when needed, one might interrupt any endless evaluation by sending the *pkill* message to the shell object.

Obviously, this approach is not real-time, *per se*, since it relies on temporary files and needs to wait for SBCL to return a result, then for Max to read it. The latency between both steps is minimal in our experience, even with heavy computations, and remains fairly close to running SBCL by itself, thus allowing to implement the system inside reactive applications. However, a slightly longer delay can become perfectly acceptable — depending on the goal of the evaluation, the complexity of the script and the length of the result — as long as such high-level algorithms become relevant to a given artistic context.

2.5. Extended vocabulary

As previously mentioned, it is possible with the *pw.eval-box* to specify a start-up package for our evaluations. For Common Lisp, packages are a way to define separate namespaces while coding. For instance, this feature is frequently used in OM and PWGL to avoid conflicts between several user libraries, in case functions or variables would share the same name.

The richness of environments like PWGL and OM lies precisely in the multitude of libraries they brought to the public through the last decades, ranging from very personal techniques by composers such as Tristan Murail (Esquisse) or Brian Ferneyhough (Combine), to more general systems of music generation or analysis. These are only a few examples of a huge variety of artistic research whose scope could be greatly extended through real-time interfaces.

SBCL allows to use a pre-defined environment instead of the one provided in the official distribution. The function *save-lisp-and-die* (from the SBCL package *sb-ext*) can store any set of packages, functions and variables as a binary image, or .core file. Therefore a newly generated image will include all standard functionality of Common Lisp and SBCL, extended by those user definitions. Consequently, evaluating a script will require to install only two files on the user's machine: the SBCL executable itself and a proper core file.

---

[5] A rule of Lisp is that any s-expression must start with a function call, otherwise it will return an error. `(+ 1 2)` is correct because the symbol + is understood as a function. The flat list `(1 2 3)`, on the other hand, will return an error since its first element 1 is not a function. The proper syntax would use the list function `(list 1 2 3)` or the quote notation : `'(1 2 3)` or `(quote (1 2 3))`. Similarly, a single symbol like foo must be quoted `'foo` or `(quote foo)`, otherwise it will be interpreted as an unbound variable. In other words, `quote` allows to turn a piece of code into a piece of data.

[6] This approach was initially discussed at IRCAM at the occasion of the International PRISMA Meeting 2015. Several members had noticed that, under certain conditions, heavy computations were really difficult to handle by PWGL or OM, but would be evaluated without hassle in pure Lisp. A first strategy was using *sprintf* to format Lisp code, but became quickly problematic for multiple reasons (confusing user interface, stability compromised by the memory management of symbols in Max, etc.). This method was explored for several months in collaboration with the composers Örjan Sandred, Hans Tutschku, Johannes Kretz and Jacopo Baboni Schilingi, then optimized and standardized by the author.

[7] Obviously, any implementation of Common Lisp, that can be accessed through a command-line interface, could be used instead. SBCL is one of the most popular implementations today: it is open-source, cross-platform, and offers great performance. Clozure CL is a good alternative, and offers similar advantages. cf. www.sbcl.org | ccl.clozure.com

[8] This external was initially developed by Bill Orcutt, then updated by Jeremy Bernstein in 2013. cf. cycling74.com/toolbox/bernstein-shell/

In order to load SBCL with a given environment, simply run the following command:

sbcl --core path-of-image.core --script path-of-script.lisp

Since this system is operational, a non-comprehensive work of adaptation of existing libraries from PWGL has been made:

- Mikael Laurson's *PMC*[9], the original constraint solver from PatchWork (also known as PWConstraints);
- Jacopo Baboni Schilingi's *CMI*, *Profile* (with Mikhail Malt) and *Constraints*;
- Frédéric Voisin's *Morphologie*, a set of tools for the analysis of symbolic sequences, exploring various paradigms.

Most of these libraries relied on several iconic functions common to both OM and PWGL (and inherited from PatchWork), such as *flat-once*, *x-append* or *posn-match*, whose definitions were retrieved and adapted from OpenMusic's sources[10]. Today an important part of this shared lexicon, as well as the aforementioned libraries, are available to use as an example binary core file provided with MOZ'Lib. Sources are accessible on simple request for now, and will be included to an autonomous Max package in a near future.

Another long-time development is in progress, in collaboration with the composers Örjan Sandred and Torsten Anders, in order to make available the *Cluster-Engine* library to Max with the same method. This new constraint solver[11], initially developed by Sandred for PWGL, allows to control several musical voices in parallel by using a network of semi-independent constraint engines – one for each parameter of each voice, including metrical structure, pitches, and rhythms. The results of this research, in terms of computational time and workflow, are already very promising, and will be eventually the object of a full publication.

3. APPLICATIONS

3.1. Accessible CAC to teach composition

As already mentioned, the need for a suitable Lisp interface in Max arose during the development of *MOZ'Lib*[12], an experimental set of pedagogical tools, designed to explore, at the same time, musical writing, creation and computer programming.

The library includes several modules under the form of *bpatchers* – inspired by BEAP and Vizzie modules in Max 7 – and mainly based on the *bach* library for its interface. Each of these modules represents a composition idea or technique, allowing the user to interact through various intuitive interfaces. Naturally, every modules can be combined together, often in unexpected ways, to imagine and realize new musical ideas.

Among those modules, several could not have been achieved without the involvement of Lisp. For instance, several techniques were directly borrowed from Jacopo Baboni Schilingi's libraries (cf. above). Other parts used Lisp as a shortcut to write functions that seemed too complex or too specific to be handled only with *bach* objects. This was the case for the *rotations* module (see Fig. 3), which uses Mikael Laurson's PMC solver to produce easily a circular permutation of a melody that maintains heuristically the overall *shape* of the original input.

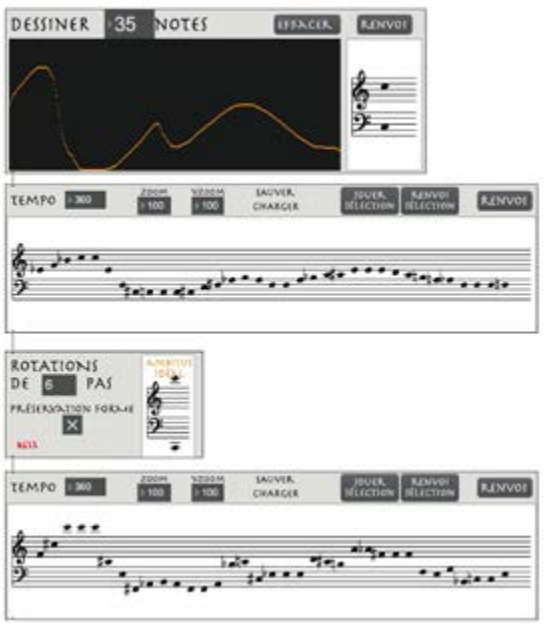

Figure 3. A simple patch using three modules from MOZ'Lib: draw_notes, see_notes and rotations

3.2. Renewing a practice with pre-existing code

The main interest of this approach, in my opinion, is to extend the very usage of compositional techniques that were initially designed for a work in "deferred-time". This is the case, of course, for most processes created with PWGL and OM, but even more before PatchWork, when the main interface was Lisp code itself.

A perfect example is a project realized in collaboration with the composer Jean-Baptiste Barrière, around his personal library *Chréode* [3]. The first version of the code was written in the early 1980s at IRCAM, and ran on *Le Lisp* (French implementation developed by INRIA) inside the CHANT and FORMES environments. *Chréode* was conceived with the ambition to realize « a grammar of formal processes, a morphogenesis » [2], that could

[9] Here it must be noted that, since PWGL is free but not open-source, the code for the PMC is accessible only as a partial port realized by Örjan Sandred for OpenMusic, under the name of *OMCS*.

[10] I take the occasion to thank the Music Representations team from IRCAM, for letting these sources accessible to the public.

[11] The Cluster-Engine is the successor of Sandred's libraries *OMRC* and *PWMC* [6], for OM and PWGL respectively.

[12] MOZ'Lib is currently maintained by the author in collaboration with the composer Dionysios Papanicolaou. It was initially supported by Ariane# (funded by Franche-Comté region in East of France), an initiative focusing on extending pedagogy with the help of digital tools. For a general introduction to MOZ'Lib, cf. bachproject.net/2016/10/15/mozlib/

determine the whole behavior of complex sound synthesis as well as instrumental scores.

This work started from an adaptation made by Kilian Sprotte as a library for OM and PWGL. As a matter of fact, the original code on *Le Lisp* was almost undecipherable because of its very specific syntax: *Chréode* is mainly based on object-oriented programming, and its interface was nothing comparable with the standardized *CLOS* (*Common Lisp Object System*) that we know today.

In a way very similar to MOZ'Lib, *Max-Chréode* consists of a palette of *bpatchers*. Each of them, just like our example in Fig. 2, aims to generate a piece of Lisp code dynamically. The user simply needs to connect boxes together, according to the rules of the system. The whole code is then sent to a *pw.eval-box* variant (*eval-chreodes*), and the result can be observed as graphs (plot~) or scores in real-time.

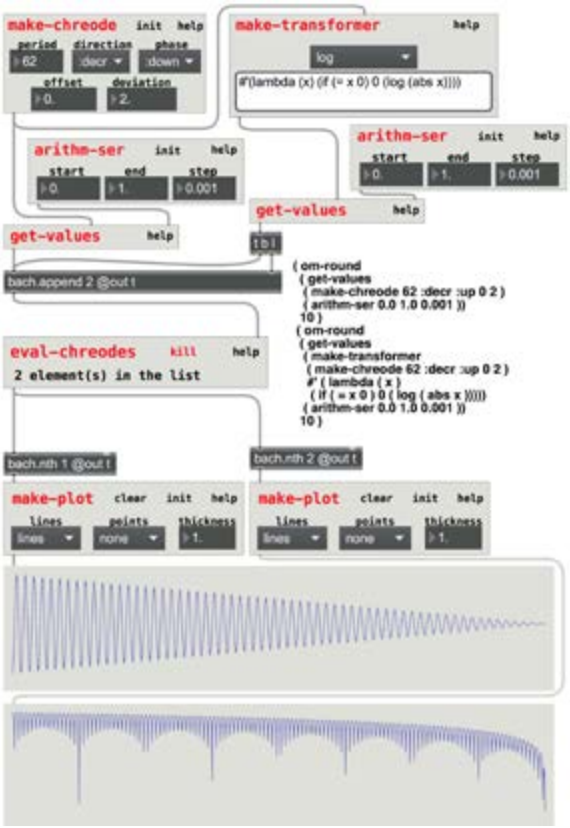

Figure 4. A simple patch with *Max-Chréode objects, on two parameters, showing code generators (with their output in a comment) and visual representations of the result by plot~*

3.3. CAC for interactive installations and performances

A last example of application is related to the project *Pre-Tensio,* by the composers Colin Roche and Jacopo Baboni Schilingi. This project, situated at the thick border between installation and performance, aims to represent the creative *tension* felt by composers during their working process.

Developed in collaboration with Colin Roche, *Le Livre des Nombres* (*The Book of Numbers)* heavily relies on Max and Lisp for its interactive apparatus. The performance, lasting 24 hours, was already presented at several occasions in art galeries in Paris. The composer, equipped with heart-rate sensors, writes music on his table, also rigged with contact microphones. The audience is able to listen, through headphones, to an amplification of the composer's heartbeat, as well as the various sounds produced by his pen on paper.

In parallel, a Max patch records the evolution of the composer's heart rate over a long time span, and eventually triggers an analysis on SBCL, to reveal its general tendencies. Afterwards, the script translates this morphology into a series of tempi modulating through time, and the long resulting list is printed automatically on receipt paper, as a metaphor of the cost in time spent by the composer during his work. Fragments of these large *scores of silence* are eventually transcribed by hand, in standard notation, and offered to the audience.

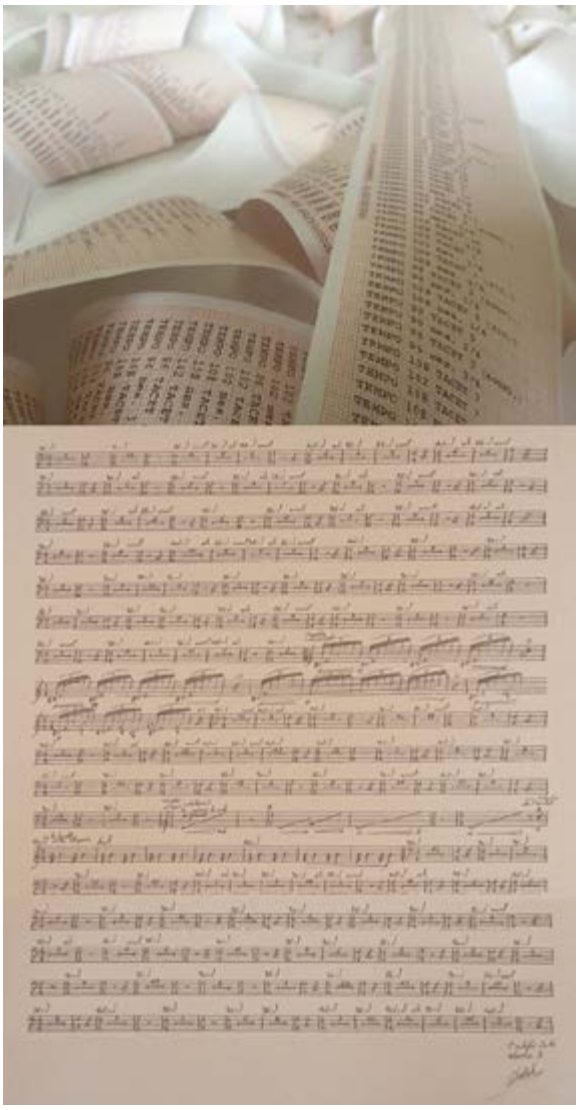

Figure 5. Example of generated receipt and manuscript transcription for *Le Livre des Nombres (photo by kind permission of the composer, © Colin Roche 2016)*

4. EVALUATION

Several improvements to this approach will have to be made. The lack of transparency for the ongoing process, exacerbated by the use of a binary .core file as knowledge repository, requires to dive into the sources to get a better understanding. For the user, it implies, of course, a familiarity with Common Lisp itself, but also with the more specific functions that might be invoked.

An important effort of documentation and dissemination would help a broader public to acquire these tools. As already mentioned, a distribution as a Max package would be a clear improvement to facilitate installation. A standard way to access, directly within Max, the documentation string of any Lisp function involved, but also a list of all the libraries loaded into the environment, would also be very helpful, in order to minimize the "black box" phenomenon.

Moreover, it is still relatively difficult to debug Lisp code from Max. For smaller mistakes, it is possible to retrieve the result of any *print* function from the first output of the shell external, which is not used otherwise. However, the SBCL debugger cannot be accessed to understand deeply nested errors. It is therefore necessary to work, on the side, on the terminal or on a dedicated IDE such as Emacs or SublimeText, to identify where a problem happens in a given temp file.

In any case, my own practice so far has been to make prototypes directly in PWGL, before starting to translate patches into Max with *bach* objects. Here lies an inherent issue to this approach, which was pointed out by Jean-Baptiste Barrière after his experience with *Max-Chréode*: patching with code generators can bring a confusion between the *semantic* and *pragmatic* aspects of a given abstraction. While patching in PWGL seems very similar to Lisp (since the lexicon and rules are broadly the same), each box actually returns its own result after evaluation. On the other hand, code generators return nothing but nested Lisp code, which makes direct patching more cryptic.

Also, the dependency to the shell external is still problematic. Since there is no official support from Cycling'74 for command-line interface, the future of this approach is not guaranteed and relies, at least for now, on the good will of the user community.

5. CONCLUSION

I presented a method to evaluate Lisp in a real-time application such as Max, and shown the various benefits of this additional language, not only for the existing users of CAC but also to develop interactive applications requiring the execution of high-level algorithms in the background. Many other possibilities than the ones covered in this article could be imagined, for instance the control of real-time audio processes, transformation or synthesis, with the help of sophisticated Lisp code.

I belong to a small community of composers who is really attached to CAC in general, and to PWGL in particular, as a privileged working environment. This development is clearly not aiming to replace OM or PWGL, but on the contrary to make their inherent possibilities accessible to different practices and contexts.

However, we can't deny that Lisp environments dedicated to CAC are today in a tight spot, compared to the rest of computer music, and that they hardly survive in socio-economic contexts that have little understanding of why they matter. In the event they would come to disappear, such an initiative could be seen as a measure of preservation for the legacy of generations of composers and researchers.

Just as sketches, scores and recordings, computer applications, whether they take the form of Max patches or Lisp code, are a support for craft knowledge and artistic expression, which must not be allowed to vanish.

6. REFERENCES

[1] Andrea Agostini & Daniele Ghisi, « Real-Time Computer-Aided Composition with *bach* », in *Contemporary Music Review*, 1(32), 2013.

[2] Jean-Baptiste Barrière, « *L'informatique musicale comme approche cognitive: simulation, timbre et processus formels* », in *La musique et les sciences cognitives,* Ed. S. McAdams & I. Deliège, Pierre Mardaga, Brussels, 1988.

[3] Jean-Baptiste Barrière, « "Chréode": the pathway to new music with the computer », in *Contemporary Music Review*, 1(1), 1984

[4] Jean Bresson & Jean-Louis Giavitto, « A Reactive Extension of the OpenMusic Visual Programming Language », in *Journal of Visual Languages and Computing*, 4(25), 2014.

[5] Mikael Laurson, *PatchWork: A Visual Programming Language and some Musical Applications*, PhD thesis, Sibelius Academy, Helsinki, 1996.

[6] Örjan Sandred, « PWMC, a Constraint-Solving System for Generating Music Scores », in *Computer Music Journal*, 34(2), 8-24, 2010.

[7] Julien Vincenot, « On "slow" computer-aided composition », in *The OM Composer's Book Vol.3*, Ed. J. Bresson, C. Agon & G. Assayag, Éditions Delatour / IRCAM-Centre Pompidou, Paris, 2016.