

BACH: An Environment for Computer-Aided Composition in Max

by Andrea Agostini and Daniele Ghisi

Initially published in: Proceedings of the ICMC 2012, pp. 373–378; <http://hdl.handle.net/2027/spo.bbp2372.2012.068>

Original PDF with new page layout.

Comment by Kerry Hagan

In 2012, Agostini and Ghisi presented *bach*, which is a music-notation environment/object for Max. So many people present their software, but that doesn't usually mean it will be adopted. I keep coming across people who are using it. So, in that regard, it has some impact.

Post scriptum by Andrea Agostini and Daniele Ghisi

This was the first journal article we published together, and the very first to introduce *bach*. At the time, *bach* was a small independent project aiming to bring computer-aided composition into the real-time world. Thirteen years later, it has grown into a mature ecosystem with tens of thousands of users and a family of related packages extending its philosophy in new directions. Some things we thought would be easy turned out to be tricky or impossible, while others came together in surprising and unexpected ways. Still, rereading this article today suggests that the original intuition that symbols, sound and performance deserve to be accessible through a common toolset was and still is a valid one.

ABSTRACT

Environments for computer-aided composition (CAC for short), allowing generation and transformation of symbolic musical data, are usually counterposed to real-time environments or sequencers. The counterposition is deeply methodological: in traditional CAC environments interface changes have no effect until a certain 'refresh' operation is performed, whereas real-time environments immediately react to user input. We shall present in this article a library for Max, named *bach: automatic composer's helper*, which adds highly refined capabilities for musical notation and symbolic processing to a typically real-time environment, in order to recompose the fracture between computer-aided composition and the real-time world.

1. INTRODUCTION

Since the advent of computers there has been great interest on how to take advantage of their superior precision, speed and power in music-related activities. The probably best-known (and commercially successful) direction has proven being the generation and transformation of sound. In recent years, inexpensive personal computers (and lately even top-end mobile phones) have gained the ability to perform professional-quality audio transformation and generation in real-time. On the other hand, several systems have been developed to process symbolic data rather than acoustic ones - 'notes' rather than 'sounds'.

These systems can be roughly divided into tools for computer-assisted music engraving (such as Finale, Sibelius, Lilypond...) and tools for computer-aided composition (CAC for short, allowing generation and transformation of symbolic musical data, such as OpenMusic¹ [1], PWGL², Common Music³...). Moreover, at least two graphical programming environments, the closely related Max and PureData, have MIDI control and sound generation and transformation among their main focuses - but at the same time they are capable to deal with arbitrary set of data, input/output devices and video. Indeed, the boundaries between all these categories are fuzzy: music engraving systems often allow non-trivial data processing; some sequencers also provide high-quality graphical representation of musical scores and sound treatment; modern CAC environments include tools for sound synthesis and transformation. It should though be remarked that Max and PureData have very crude native support for sequencing, and essentially none for symbolic musical notation.

Another, orthogonal distinction should be made between real-time systems, which 'immediately' react to interface actions (such as Finale, MaxMSP, ProTools...) and non-real-time systems, where these actions have no effect until a certain 'refresh' operation is performed (such as Lilypond, OpenMusic, PWGL). The latter is the case of typical CAC environments; yet, in some cases this is unnatural, and it might be argued that there is no deep reason why symbolic processing should not be performed in real-

time. This does not mean that every compositional process should benefit from a real-time data flow, but some might, as we shall exemplify at the end of the paper. Real-time is a resource, rather than an obligation. Yet, the lack of this resource has pushed, up to now, the development of CAC techniques only in the off-line direction.

In our own experience, the real-time or non-real time nature of an environment for music composition deeply affects the very nature of the compositional process. Composers working with sequencers, plug-ins and electronic instruments need them to immediately react as they change their parameters; likewise, composers working with symbolic data might want the machine to quickly adapt to new parameter configurations. As composers ourselves, we believe that the creation and modification of a musical score is not an out-of-time activity, but it follows the composer's discovery process and develops accordingly.

This issue has been faced by Miller Puckette in [11]:

"While we have good paradigms for describing processes (such as in the Max or Pd programs as they stand today), and while much work has been done on representations of musical data (ranging from searchable databases of sound to Patchwork and OpenMusic, and including Pd's unfinished data editor), we lack a fluid mechanism for the two worlds to inter-operate."

Arshia Cont in [5] adds:

"The performers of computer music have been faster to grab ideas in real time manipulations and adopting them to their needs. Today, with many exceptions, a wide majority of composed mixed instrumental and electronic pieces are based on simplistic interactive setups that hinder the notion of interactivity. This fact does not degrade the artistic value of such works in any sense but underlies the lack of momentum therein for serious considerations of interactivity among the second group."

Of course, this dichotomy has already been addressed. Several interesting projects have been developed, linking real-time environments to graphical representations of both classical and non-classical (and potentially non-musical) scores, including OpenTimeLine⁴ [9] and INscore⁵ [7]. In at least one case, namely MaxScore⁶ [6], this is augmented by a very sophisticated editing interface. A more general approach is FTM's [12], which provides a powerful framework for data representation and processing with a focus on musical structures, including some facilities for graphical display of simple scores.

Resuming the ideas of [2, 3], with the library *bach: automatic composer's helper* we have tried to achieve a coherent system explicitly designed for computer-assisted composition. *bach* takes advantage of Max's facilities for sound processing, real-time interaction and graphical programming, combining interactive writing and algorithmic control of symbolic musical material.

2. PROGRAMMING PARADIGMS

bach complies with the graphical data-flow programming paradigm of Max, in which information is represented as a vertical, top-down flow. Data, typically coming from some user interaction, enter the program at its top, are acted upon by a chain of specialized operators connected by lines called 'patch cords' and exit the program at its bottom. A simplified model of this mechanism, as seen from a lower-level point of view, might appear as follows: each operator is a function, usually written in C or C++, and the data entering it are the arguments of the function call. After performing its work upon the data it has received, each operator calls the function corresponding to the next operator in the chain, passing it the acted-upon data. In this way a call stack is built, in which the operator at the top of the graphical patch corresponds to the function at the base of the stack, and the operator at the bottom of the graphical patch corresponds to the function at the top of the stack. It is crucial to note that all these functions have no return value: the last operator of the chain simply passes the data to an arbitrary output device. In this way, the perception on the user's side is that the program essentially behaves like a musical instrument, in which an action (e.g., pressing a piano key) triggers a series of reactions (levers moving, hammers striking) leading, in a measurable but usually negligible time, to the production of a sensible result (sound).

The major graphical computer-aided composition environments, that is the Patchwork family [10, 4] (Patchwork, OpenMusic, PWGL), are based upon the Lisp programming language. Although superficially similar to Max from the point of view of the user interface (data and functions are represented by graphical elements connected by lines representing the flow of elaboration), the underlying programming paradigm is radically different. Essentially, the graphical program is indeed a representation of a Lisp expression, with elements on the top of the patch corresponding to the deepest elements of the expression. The user requests the evaluation of an operator, which in turn will request evaluation of the operators above it, and so on. From a lower-level point of view, a call stack is built in this scenario as well; the difference is that all the functions in the stack have a return value, and the final return value is returned to the user through a console. Of course, in some cases the side effects of the evaluation (e.g., a change in an user interface widget, or the production of a MIDI stream) are more important than the result itself. This paradigm applies *a fortiori* to textual Lisp-based environments such as Common Music or Impromptu.

The difference between the two paradigms is crucial: if we assume that parameters are handled at the beginning of the process, a bottom-up process (like within the Patchwork paradigm) will ultimately be a non-real-time process, since parameter changes cannot immediately af-

¹<http://repmus.ircam.fr/openmusic/home>

²<http://www2.siba.fi/PWGL/>

³<http://commonmusic.sourceforge.net/>

⁴<http://dh7.free.fr>

⁵<http://inscore.sourceforge.net>

⁶<http://www.computermusicnotation.com>

fect anything below them, unless some bottom-up operation is requested on some lower elements. Moreover, the Max paradigm, not having to depend on return values, easily allow for much more complexly structured patches: a single action can trigger multiple reactions in different operators (a function can call several other functions, one after another has returned). The Patchwork paradigm, on the other hand, has the advantage of allowing seamless integration with textual coding, which can be an extremely useful resource whenever conceptually complex operations must be implemented. Moreover, representing musical notation (from single notes to an entire score) requires sufficiently powerful and flexible data structures, which the Lisp lists certainly are.

3. THE BACH ENVIRONMENT

As already stated, *bach* is a library of objects and patches for the software Max, the distinction between objects and patches concerning more the implementation than the actual usage of these modules. At the forefront of the system are the *bach.score* and *bach.roll* objects. They both provide graphical interfaces for the representation of musical notation: *bach.score* expresses time in terms of traditional musical units, and includes notions such as rests, measures, time signature and tempo; *bach.roll* expresses time in terms of absolute temporal units (namely milliseconds), and as a consequence has no notion of traditional temporal concepts: this is useful for representing non-measured music, and also provides a simple way to deal with pitch material whose temporal information is unknown or irrelevant. It should also be noted that the implementation of traditional temporality concepts in *bach.score* is in fact quite advanced, as it allows multiple simultaneous time signatures, tempi and agogics. Besides this fundamental difference, the two objects offer a large set of common features, among which:

- editing by both mouse and keyboard interface, and by Max messages (see Fig. 1);
- support for microtonal accidentals of arbitrary resolution (see Fig. 2);
- wide possibility of intervention over the graphical parameters of musical notation;
- ability to associate to each note various types of meta-data, including text, numbers, files and breakpoint functions (see Fig. 6);
- variable-speed playback capability: both *bach.score* and *bach.roll* can be seen as advanced sequencers, and the whole set of data (such as pitch, velocity and duration information) and meta-data associated to each note is output at the appropriate time during playback, thus making both objects extremely convenient for controlling synthesizers and other physical or virtual devices.

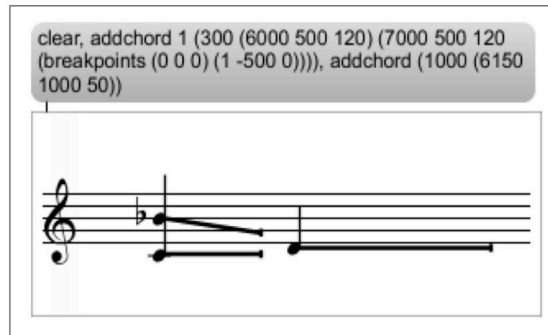


Figure 1. Any notation object can be edited by both GUI interaction and Max messages. In this case we're clearing the *bach.roll*, and then adding two chords.



Figure 2. Semitonal, quartertonal and eighthtonal divisions are supported via the standard accidental symbols (upper example). All other microtonal divisions are supported as well, but symbolic accidentals will be replaced by labels with the explicit fractions of tone (lower example), or with cents differences from the diatonic note.

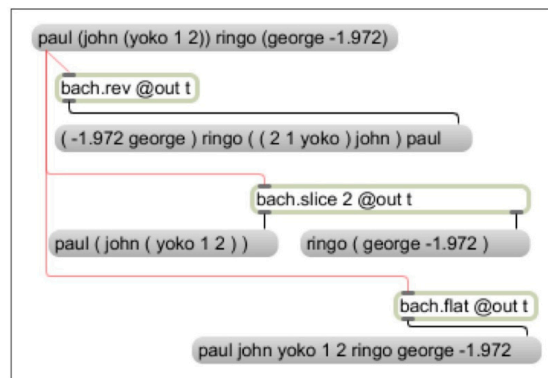


Figure 3. *bach* has a wide range of objects capable to perform standard structure operation on *llls*, such as reversing, slicing, flattening, rotating and so on. In the picture, we see the results of reversing, slicing and flattening a list. Moreover, most operations can be constrained only in some levels of depth.

3.1. Data types

bach also provides Max with two new data types: rational numbers and a nested list structure called *llll*, an acronym for *Lisp-like linked list*. Rational numbers are extremely important in music computation, as they express traditional temporal units such as 1/2, 3/8 or 1/12 (that is, a triplet eighth note) as well as harmonic ratios. The nested list has been chosen for both similarity with the Lisp language, in a way to ease communication with the major existing CAC environment, and the need to establish a data structure powerful enough to represent the complexity of a musical score, but flexible enough to be a generic data container lending itself to arbitrary manipulations through a relatively small set of primitives. In fact, the large majority of the modules of the *bach* library are tools for working upon *llls*, performing basic operations such as retrieval of individual elements, iteration, reversal, sorting, splicing, merging and so on (see Fig. 3). Some subsets of the library are applicable to *llls* satisfying certain given conditions: e.g., it is possible to perform mathematical operations over *llls* solely composed by numbers; a set of operators for matrix calculus only works with appropriately structured *llls*; and so on. It is important to stress that all these operators are indeed Max objects, and while the kind of operations performed may bear some resemblance with Lisp, the actual implementation and interface are radically different, and as integrated as possible with the Max system. On the other hand, at least one Common Lisp interpreter designed to run as Max objects has been developed, Brad Garton's *maxlispj* [8]: it is extremely easy to exchange data with this object, in order to take advantage of the expressive power of Lisp textual programming within a Max patch.

3.2. Music representation

At the intersection between the modules for musical notation and the list operators is a family of objects performing operations upon *llls* containing musical data.

It is worth noting that different *bach* objects exchange musical scores in the form of specifically-structured *llls*, whose contents is entirely readable and editable by the user; this is different from what happens e.g. in OpenMusic, where the exchange of musical data often involves opaque objects.

This allows much easier and more transparent manipulation of the musical data themselves. As a consequence, strictly musical operations such as rhythmic quantization are just extremely specialized operations upon *llls*, which of course can be performed only if the *llll* itself is structured properly, and if its content is consistent from the point of view of musical notation.

The structure of a *llll* representing a *bach.score* (Fig. 4) might appear quite complex at first sight, but the or-

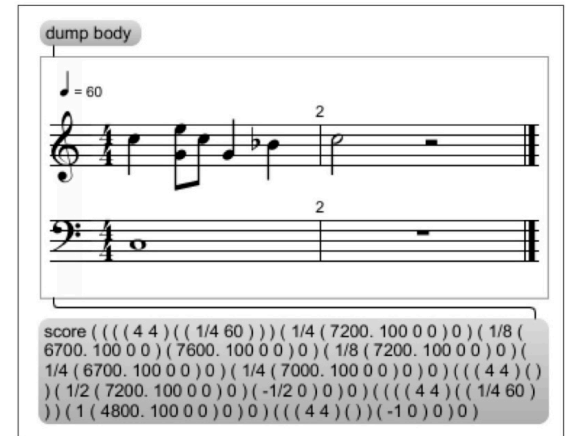


Figure 4. The structure of a simple score in *llll* form, with branches for voices, measures, chords and notes. (The header, containing additional information such as clefs, keys, types of meta-data, has not been dumped).

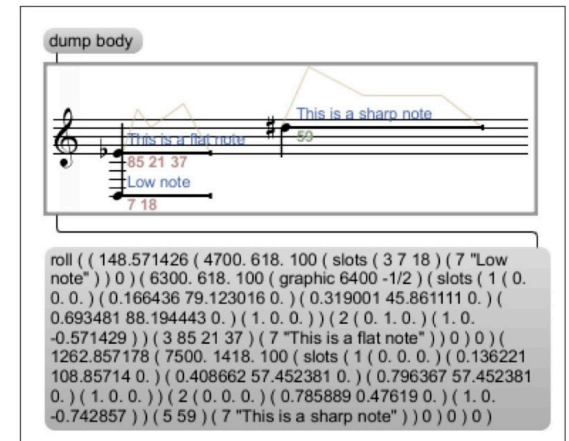


Figure 5. The structure of a non-measured score in *llll* form, with branches for voices, chords and notes. Notice the meta-content contained in each note, appearing in the *llls* starting with the *slots* symbol. The form (type, range, domain...) of each slot appears in the header, which has not been dumped.

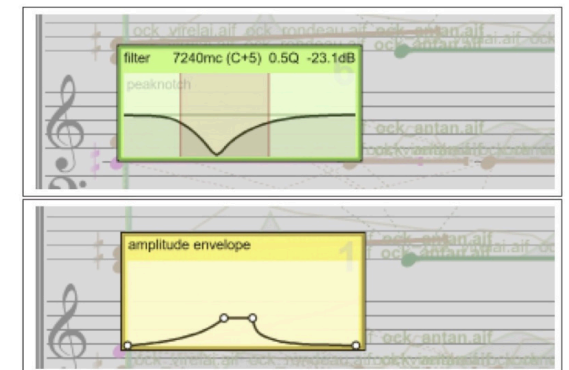


Figure 6. Two examples of slot windows.

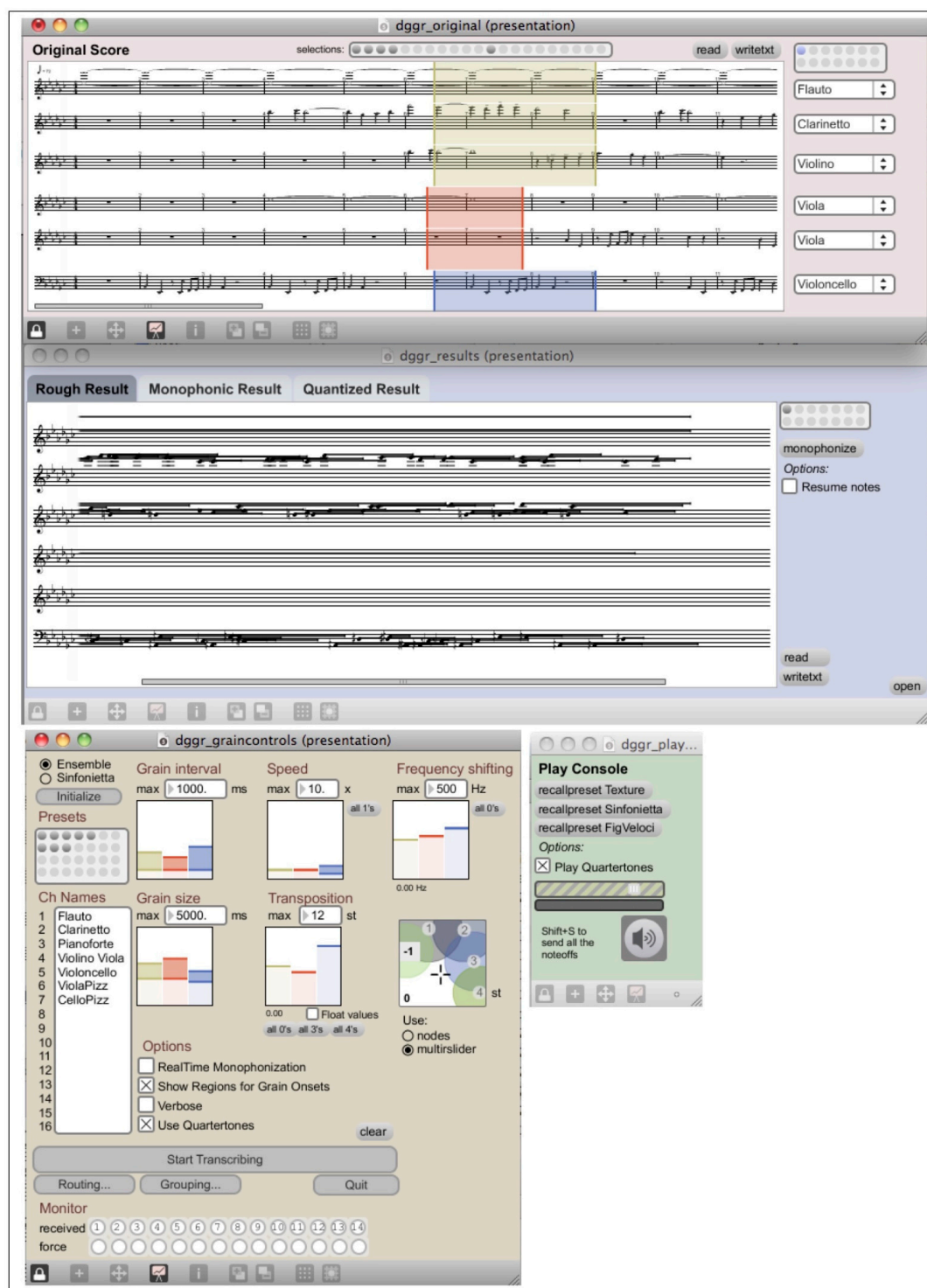


Figure 7. Screenshot of a patch achieving a real-time symbolic granulation. The original score (upper reddish window) has some markers to determine and modify the grain regions. Parameters are handled in the lower ochre window. When the user presses the ‘Start transcribing’ button, the result appears and accumulates in the middle blue window. If desired, one may make it monophonic, retouch it, and finally quantize it. Every parameter is user-modifiable and affects the result in real-time, as in any electroacoustic granulation machine.

ganization of its contents is meant to be extremely rational: after a header section containing global information such as the clefs or the types of meta-data appearing in the score, we find a sub-tree whose branches correspond to one voice each; each voice branch contains branches for each measure; each measure branch contains some measure-specific information (such as time signature) and branches for each chord; each chord branch contains some chord-specific information (such as its duration) and branches for each note; and each note branch contains pitch and velocity leaves, as well as possible further specifications, such as glissando lines, enharmonic information, articulations and meta-data. The *llll* representing a whole *bach.roll* has essentially the same structure (Fig. 5), except that the measure level is not present. With the provided set of list operators, specific pieces of information referring to single elements or sections of the score are not difficult to locate and manipulate. Moreover, both *bach.score* and *bach.roll* provide simplified ways to retrieve and enter only specific sets of values to operate upon (e.g. pitches or velocities only), which greatly eases the implementation of most algorithmic operations.

3.3. Data handling mechanism

As the goal of *bach* is allowing real-time interaction, a great amount of work has been spent to improve the stability and efficiency of the system. All the operations in *bach* are thread-safe in the context of the Max threading model, and the passing of *lllls* between objects happens by reference, rather than by value, unless the user explicitly requests otherwise, which is the case whenever the contents of a *llll* need to be passed to a non-*bach* Max object (that only accepts data passed by value). Thus, *lllls* are copied only when strictly necessary, and in all the other cases a reference counting mechanism is used to ensure that the lifetime of data structures and the usage of memory are correctly managed. On the other hand, all this is transparent to the user, who never needs to cope with the cloning of *lllls*, or the distinction between destructive and non-destructive operations - as, on the contrary, it is often the case with Lisp.

3.4. Practical applications

Taking all this into account, it should be clear that *bach* is somehow placed at the convergence of several categories of musical software. Its capabilities of graphical representation of musical scores typically belong to music engraving systems - although it should be noted that, in its current state, *bach* lacks some essential features of this kind of programs, first of all a page view. On the other hand, most of its features are conceived in order to make it a tool for Computer Aided Composition as powerful as the traditional Lisp-based environments, and able to communicate with them. It can be used as the core of an extremely advanced and flexible sequencer, with the ability to drive virtually any kind of process and playback system. Finally, it can of course lend itself to innovative ap-

plications exploiting the unique convergence of these different paradigms and its specific real-time behavior (such as the symbolic granulation example shown in Fig. 7).

4. FUTURE DEVELOPMENTS

At the time of writing, *bach* is in its alpha development phase: although the system is usable, not all the intended features have already been implemented. Some of the planned additions are:

- Support for rhythmic tree representation, which will allow, for example, nested tuplets to be represented, whereas now a triplet containing a quintuplet is represented as a ‘flat’ 15-uplet. This feature is currently under development, together with an intuitive measure linear editing system for the note insertion. The underlying challenge is to keep the tree and linear representations of durations always compatible, so that users should concretely deal with the tree representation only when they explicitly ask to (e.g. when they insert as rhythm a nested rhythmic structure), or when they perform hierarchical operations (e.g. when they split a chord). Users will also be able to rebuild a default rhythmic tree from the linear representation at any moment.
- Implementation of hierarchical structures within a score, allowing the user to group elements by name, where an element can be a chord, a note, a marker, or another group.
- Support for import and export of MIDI, MusicXML and SDIF files.
- A solver for constraint satisfaction problems.

Notice that the software development situation might have changed at the time of publication, and some or all of the hereby proposed features might already be partly or fully implemented.

5. REFERENCES

- [1] C. Agon, “OpenMusic : Un langage visuel pour la composition musicale assistée par ordinateur.” Ph.D. dissertation, University of Paris 6, 1998.
- [2] A. Agostini and D. Ghisi, “Gestures, events and symbols in the *bach* environment,” in *Proceedings of the Journées d’Informatique Musicale*, Mons, Belgium, 2012, pp. 247–255.
- [3] —, “Real-time computer-aided composition with *bach*,” *Contemporary Music Review*, 2012, to appear.
- [4] G. Assayag and al., “Computer assisted composition at Ircam: From patchwork to OpenMusic,” *Computer Music Journal*, no. 23 (3), pp. 59–72, 1999.

- [5] A. Cont, “Modeling Musical Anticipation,” Ph.D. dissertation, University of Paris 6 and University of California in San Diego, 2008.
- [6] N. Didkovsky and G. Hajdu, “Maxscore: Music Notation in Max/MSP,” in *Proceedings of the International Computer Music Conference*, 2008.
- [7] Y. Fober, Y. Orlarey, and S. Letz, “An Environment for the Design of Live Music Scores,” in *Proceedings of the Linux Audio Conference*, 2012.
- [8] B. Garton. (2011, Jan.) maxlispj. [Online]. Available: <http://music.columbia.edu/~brad/maxlispj/>
- [9] D. Henry, “PTL, a new sequencer dedicated to graphical scores,” in *Proceedings of the International Computer Music Conference*, Miami, USA, 2005, pp. 738–741.
- [10] M. Laurson and J. Duthen, “Patchwork, a graphical language in preform,” in *Proceedings of the International Computer Music Conference*, Miami, USA, 1989, pp. 172–175.
- [11] M. Puckette, “A divide between ‘compositional’ and ‘performative’ aspects of Pd,” in *Proceedings of the First International Pd Convention*, Graz, Austria, 2004.
- [12] N. Schnell, R. Borghesi, D. Schwarz, F. Bevilacqua, and R. Müller, “FTM - Complex Data Structures for Max,” in *Proceedings of the International Computer Music Conference*, 2005.