Andrea Agostini[*] and Daniele Ghisi[†]
[*]Conservatory of Cuneo
Via Roma 19
12100 Cuneo, Italy
[†]Institut de Recherche et Coordination
Acoustique/Musique
1, Place Igor-Stravinsky
75004 Paris, France
{andreaagostini, danieleghisi}@bachproject.net

A Max Library for Musical Notation and Computer-Aided Composition

Abstract: This article introduces a library of external objects for real-time computer-aided composition and musical notation in Max. The library provides Max with a set of tools for the graphical representation of musical notation, manipulation of musical scores through a variety of approaches ranging from GUI interaction to constraint programming, and sequencing. The library is oriented to real-time interaction, and is meant to interoperate easily with other processes or devices controlled by Max, such as DSP tools, MIDI instruments, or generic hardware systems. These features and design choices place our software at the intersection of various categories of musical software environments and approaches, allowing it to help reduce the gap found between tools for sound-based, electroacoustic musical practices, and for symbol-based, traditional composition. The library is called "bach: automated composer's helper."

A long-standing tradition has differentiated two main fields of interest in computer music: digital signal processing (DSP) on the one hand, and treatment of symbolic musical data on the other. The former has reached a high level of refinement and public awareness in the last decades, with professional-grade DSP algorithms nowadays widely available and used in an increasingly broad set of devices and applications, including mobile phones and embedded systems. The latter, on the other hand, seems confined to a small number of specialist software systems such as OpenMusic (Agon 1998), PWGL (Laurson and Kuuskankare 2002), Strasheela (Anders, Anagnostopoulou, and Alcorn 2005), or Common Music (Taube 1991). Moreover, DSP is used today to various extents in virtually any type of musical production, as well as many nonmusical contexts. On the contrary, the use of systems for symbolic processing is generally a prerogative of certain specific styles of avant-garde experimental music. The only exception to this relative isolation is represented by musical engraving systems such as Finale or Sibelius. At their basis, they represent possibly the simplest approach to the processing of musical notation. Still, it might be simplistic to consider them just as a strict translation of the word-processing principle to music, because they all implement some basic, yet significant, operations on musical scores, such as transposition, augmentation, and playback. To some extent, all these software systems can be effectively considered as having computer-aided composition capabilities, which in some cases can be taken to a nontrivial degree of complexity, for example, through the use of certain advanced plug-ins in Finale, or through ManuScript, the scripting language of Sibelius.

Still, in our own compositional activity, we feel there is a lack of systems and environments that would allow one to easily cross the boundaries between these categories. *Musique mixte* (music using traditional instruments along with electroacoustic sounds) demands tools capable of representing and processing complex instrumental scores associated with rich acousmatic contents. Composition of electroacoustic music is often performed within sequencers such as ProTools or Logic, but this approach works best for music that is essentially based upon a compositional paradigm of montage. Music for synthesizers most often needs a more flexible kind of score, as exemplified by Csound, which—in spite of its age and the crudeness of its graphical interface and programming language—probably remains the most widely used system of this kind. Improvisational and performative musical practices benefit from the ability to represent musical scores in real time, and having them interact with audio analyses and processes, MIDI streams, sensors, and actuators. Even a more traditional approach to instrumental music composition may be greatly enhanced by the

Computer Music Journal, 39:2, pp. 11–27, Summer 2015
doi:10.1162/COMJ_a_00296

immediacy of the real-time response of modern DSP systems and audio sequencers. Puckette (2004) and Cont (2008b) have, on several occasions, tackled the need to bridge the gap between software tools for audio-based, electroacoustic musical practices and symbol-based, traditional composition. The computer-assisted orchestration project carried out at the Institut de Recherche et Coordination Acoustique/Musique (IRCAM) is another approach to a similar concern (Carpentier and Bresson 2010; Esling and Agon 2010).

Our attempt to give a partial response to these various challenges is a library for the software Max named BACH: Automated Composer's Helper (hereafter, simply "bach"), which we started developing in 2010 and which has since undergone continuous development, also giving birth to a derived system named "cage."

The library provides Max with graphical interfaces for the display, editing, and playback of augmented musical scores, as well as more than 200 modules for processing these scores through various low- and mid-level operations and paradigms, ranging from basic list processing to constraint programming. Its tight integration within a popular programming environment for audio and mixed media, as well as its focus on real-time responsiveness, are meant to bridge the gap between sound-based and symbol-based systems. The bach library achieves this by taking advantage of existing technologies, helping artists and media producers to avoid facing the possibly steep learning curve of a totally new software system.

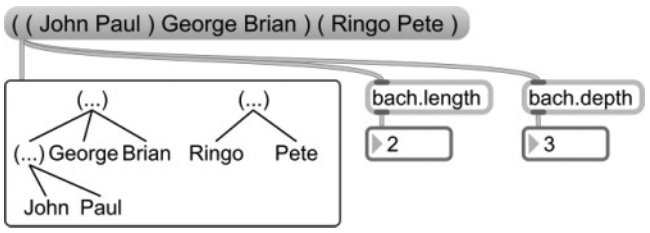
The Choice of Max

The choice of Max as the host environment for bach was prompted by several considerations, the most important being the already cited ease of integration with a multitude of processes and devices, including DSP, MIDI, visuals, and virtually any hardware system. Another important consideration was the maturity and stability of the Max graphical user interface, and the power of its graphical API. The fact that Max is scriptable through several programming languages (C/C++, Java, JavaScript, Lua, Common Lisp, Clojure, Python, and more) was considered another interesting feature. In fact, all the most critical sections of bach are written in C or C++, which allow precise control of the graphical display, flexible management of custom data structures, and great computational speed. On the other hand, interaction with languages other than C/C++ has proven more problematic than we had anticipated, because of the peculiar data structures introduced by bach in the Max software ecosystem. This actually leaves an open problem: Max, as a programming language, has a strong focus on building rich interactions and user interfaces, but implementation of nontrivial algorithms for computation and data processing is often less than straightforward because of the graphical programming paradigm itself. Nonetheless, such nontrivial algorithms are often necessary in the fields of computer-aided and algorithmic composition, in order for composers to set up their own mechanisms for the manipulation of symbolic musical data.

Overall Structure of the Bach Library

The bach library is currently composed of roughly 110 Max externals (precompiled modules written in C and C++) and 120 abstractions (editable Max patches that can be reused within other patches and that are constituted of externals and other abstractions). The choice of implementing a given operator as an external or as an abstraction is the result of several factors. First, there are cases in which there is no actual choice: It is generally impossible, or extremely cumbersome and inefficient, to implement as abstractions complex user interfaces, operations requiring maximum computational speed, primitive operators upon custom data structures, or modules with certain problematic features such as the ability to change the number of their own inlets. Besides these specific cases, in the early versions of the library the approach was to make abstractions whenever possible, because this was a way to validate and test the externals upon which the abstractions themselves were based, to figure out whether new externals were needed, and to build a set

Figure 1. An example of an llll (a "Lisp-like linked list"), displayed in tree form via the bach.tree object. The length of such an llll is the number of elements at the root level (the topmost level in the tree representation). The depth is the maximum number of nested levels, including the root level.


((John Paul) George Brian) (Ringo Pete)
(...)
(...)
(...) George Brian Ringo Pete
John Paul
bach.length
2
bach.depth
3

of pedagogical patches showing how other bach modules could be used within variously complex contexts. This approach changed over time, and more recently we have tended to build abstractions only for very simple operations that can be carried out with small patches. We even re-implemented several modules that had previously been released as abstractions, rewriting them as externals to make them more efficient or to enhance them with capabilities that were impossible, or very complicated, to implement otherwise.

Data Structures

The first and main goal of the bach library is the manipulation of musical scores, which are highly complex data structures. Max offers a limited choice of data structures, all of which are rather simple. The only exception is the dictionary, which is a sort of ordered hash table that we did not find well-suited to the kind of data representation we needed to implement. Indeed, we think that musical scores can more easily be seen as collections of arbitrarily long sequential collections of data on various hierarchical levels (e.g., voices containing measures containing chords), rather than as associative maps based upon symbolic keys.

OpenMusic and PWGL typically represent a score as an opaque object, accessible only through a set of predefined methods, and consisting of simpler opaque objects, such as voices, notes, or rests—thus reflecting the hierarchical paradigm described above. Although elegant from a purely theoretical point of view, we find this approach to be somewhat limiting with respect to the user's freedom of performing arbitrary operations that we might not have foreseen. Our choice was therefore to provide Max with a new data structure that is powerful enough to represent a score, but simple enough to be manipulated through a generic and, as far as possible, a "language-agnostic" set of tools. At the same time, we wanted this data structure to allow easy exchange of data with the major Lisp-based systems, such as the aforementioned OpenMusic and PWGL. Hence, we chose to implement a tree structure inspired by the Lisp list, called llll (pronounced *el-el-el-el*, standing for "Lisp-like linked list"). An llll can contain all the standard Max data types, i.e., integers, floats or symbols, as well as rational numbers and other lllls, to an arbitrary level of depth (see Figure 1). Most bach objects are designed to work upon lllls.

An example of an llll is `1 2 (3 4) 5 ((6/7 8/9) 1.0) 1.1 one point two`, where pairs of parentheses enclose sublists, i.e., lllls within lllls. On a side note, an llll can be used as an associative array by simply treating its first element as a key, although this is less efficient than a hash-based data structure. This approach, similar to the association list paradigm in Lisp, is indeed used quite often within bach. In most practical cases the cost of traversing the list to find the desired key can be considered negligible, because this strategy is most frequently used with relatively small sets of key-value pairs. Further, the object devoted to this task, `bach.keys`, can be instructed to search the llll only at the specific depth levels at which the keys are expected.

Readers acquainted with the Lisp programming language will notice that an llll, in contrast to a Lisp list, is not itself enclosed in a pair of parentheses. This difference is meant to ease the relation with the Max environment, as in this way any standard Max message, such as `foo bar 1 2 3`, can be considered as an llll containing no sublists. On the other hand, this prevents objects from exchanging Max- or Lisp-style atoms, since, for example, the message `foo` will, in any case, be treated by bach objects as an llll containing only a single element. This brings a substantial simplification with respect to the data-type system of both Max and Lisp. Moreover, lllls are always passed by value between objects, through a combination of deep copying and reference counting techniques that is completely transparent to the user and mimics the behavior of standard Max messages. This paradigm has the advantage of being more easily predictable by the user, who never has to worry about cloning or the difference between destructive and nondestructive operations. Extensive testing showed that the performance penalty caused by deep copying is generally negligible in common, real-life usage scenarios.

Data Representations

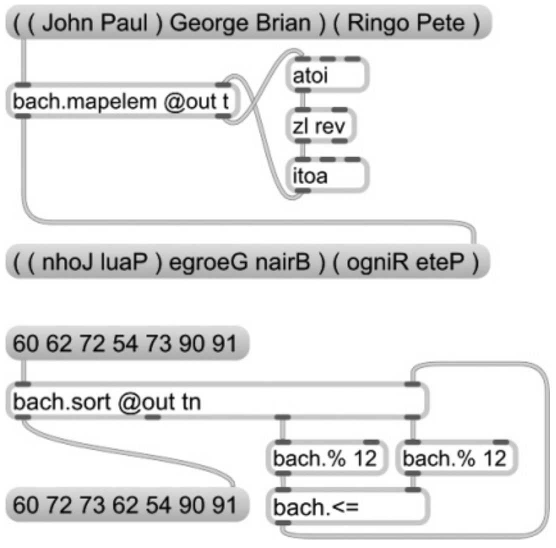
Max messages have an intrinsic length limitation which, as of Max 7, is set to 32,767 elements. Although comfortably sufficient for most uses, this limitation can prove to be extremely restrictive when dealing with musical scores, as they can be composed of a very large number of complex items, each represented by several constituting elements. Moreover, passing lllls as regular Max messages, with sub-lists being expressed through pairs of parentheses in a sequential array of elements, requires parsing these arrays and converting them into an actual tree structure in order for them to be operated upon. This mechanism is indeed implemented in all the bach objects, and is essential because it allows standard Max objects to access the contents of lllls, provided that they are not too long. This is called the llll *text format*. But the preferred passing mechanism for lllls, whenever they are exchanged between bach modules, is the so-called *native format*. In native format, when viewed through a regular Max object such as `print`, lllls appear as opaque messages consisting of the `bach.llll` message followed by an integer, which is the key for an internally maintained table that bach objects can access, and that associates these numeric keys with the actual llll data structure. In this way, bach objects can exchange lllls of arbitrary length, and the llll exchange itself can happen at a much greater speed than if it required the parsing described earlier. Although standard Max objects cannot access the contents of an llll in native format, they can control its flow. Thus, objects such as `gate`, `trigger`, and `send` can correctly deal with native-format lllls. All the bach objects that can produce lllls accept the attribute "out," which specifies whether the lllls must be sent out their outlets in native or text format. On the other hand, all the bach objects can accept lllls in both formats, and automatically interpret them correctly.

Lambda Loops

Some bach objects and abstractions consistently implement a specific mechanism called a *lambda loop*, whose goal is providing a sort of replacement for lambda functions within the Max environment, because concepts such as anonymous functions or lambda calculus would be completely extraneous to the overall programming philosophy and paradigm of Max. It should therefore be made clear that the "lambda" name is not used literally in this context, but rather as an allusion and a hint for the user.

One typical practice in computing is to combine two operations in order to specify a given subclass of a more generic problem. For example, one might want to combine a modulo 12 operation with a generic sorting algorithm in order to sort MIDI pitches according to their pitch classes. Different programming languages have various constructs for this, and Lisp's lambda functions are probably among the most elegant and powerful. Lambda loops are not an actual language feature, but rather a design pattern taking advantage of Max's general callback-based operation model. In short, a lambda loop is a patching configuration in which one or more dedicated outlets of an object or abstraction output data iteratively, in the usual form of native or text lllls, to a patch section. This patch section calculates a result to be returned to a dedicated inlet of the starting object. The result for each iteration is typically a modification of the original data or a number, which can represent a qualitative value (such as in `bach.constraints`, described in the section Constraint Programming) or a Boolean. The modification must be immediately fed back into the object originating the lambda loop, because the subsequent course of the iteration may depend on it. In bach terminology, the outlets and inlets dedicated to these transactions are called *lambda outlets* and *lambda inlets*, respectively, and are always the rightmost outlets and inlets of the

Figure 2. Two examples of lambda loops. In the top patch, we modify each element of the incoming llll by reversing the symbol. Symbols are sent, one by one, through the right, "lambda" outlet of `bach.mapelem`, reversed via the `itoa`, `zl rev`, and `atoi` objects, and fed back into the lambda inlet of `bach.mapelem`. (Because of the way the `atoi` object handles characters beyond the traditional, seven-bit ASCII character set, the patch would need additional logic to handle those characters.) In the bottom patch, we sort an incoming list of MIDI pitches (corresponding to the notes C4, D4, C5, F♯3, C♯5, F♯6, and G6) according to their pitch classes. Pairs of elements of the incoming llll are sent through the two rightmost (lambda) outlets of `bach.sort`, and before using the standard ordering given by `bach.<=`, we apply a modulo operation on both of them. This tells `bach.sort` that the two elements will be in the correct order if the remainder modulo 12 of the left element is less than or equal to the remainder modulo 12 of the right element. Subsequently, `bach.sort` sends out its outlet a list of MIDI pitches corresponding to the notes C4, C5, C♯5, D4, F♯3, F♯6, G6.



object or abstraction. It should be remarked that on no occasion can a lambda loop cause a stack overflow, because lambda inlets are always "cold" (i.e., following Max terminology, they do not trigger an immediate result).

The example of sorting by pitch class would thus be implemented by using the `bach.%` and `bach.<=` modules, in combination with the `bach.sort` object, as shown in Figure 2.

Bach Modules by Family

The modules composing the bach libraries can be subdivided into a number of families, according to their purpose and behavior.

Score Editors and Sequencers

At the forefront of bach are the `bach.roll` and `bach.score` objects. They are essentially two score editors, whose main difference is the representation of time. The `bach.score` object implements a traditional, measured representation including notions of tempo, time signature, measures, and rests, whereas `bach.roll` always represents time in milliseconds, and does not support the aforementioned notions. This distinction is motivated by several considerations. These include the fact that the criteria for horizontal spacing with respect to time are radically different (strictly proportional for `bach.roll`, based upon more complex relationships and spacing tables for `bach.score`). Furthermore, as explained in the following, the data structures underlying the graphical representations are different. The `bach.roll` object can generally be considered as the simpler one and—besides actually displaying proportionally notated music—is the usual choice for representing pitch structures whose temporal parameters are irrelevant or nonexistent. The `bach.roll` and `bach.score` objects share the vast majority of their features, however—the most important of these being the fact that every aspect of their contents can be edited both by messages and with mouse-and-keyboard interaction, thus allowing them to be used in a variety of scenarios, from completely interactive to completely algorithmic. In every case, the score display is always immediately updated.

Generally speaking, the most basic element in both objects is the note. Notes belong to chords (although in `bach.score` even noteless chords can exist, and are treated as rests). In `bach.roll` chords belong directly to voices, whereas in `bach.score` chords belong to measures, which in turn belong to voices. In `bach.roll` each chord has an onset (a temporal starting point in the score), whereas in `bach.score` each chord has a duration (chord onsets can be inferred from the sequence of chord durations and the tempo). In both `bach.roll` and `bach.score` each note has a pitch (expressed in MIDI cents, that is, hundredths of a semitone above MIDI pitch 0) and a MIDI velocity. In `bach.roll` each note in a chord also has an independent duration, and in bach.score each note also carries an additional flag representing tie information.

Besides these usual properties, notes can contain a large amount of metadata of various types: numbers, text, lists, breakpoint functions, file names, and DSP parameters such as filter definitions or spatialization trajectories. These are organized as structures called *slots*. Slot data can be input via the graphical interface as well as via message interaction. The latter approach makes both objects capable of recording virtually any stream of data in real time, as long as it is associated with note events.

Both bach.score and bach.roll can play back the scores they contain, and each time a note is encountered all its slot data is sent, along with the standard set of note properties, through the object outlets. This feature makes bach.roll and bach.score two extremely flexible sequencers, capable of driving any possible process or device that can communicate with Max in real time. Real-time playback, by which events are automatically scheduled in time in Max's scheduler thread, is activated by the play message and ends either when a stop message is received or when there is no more score content to be played. Several options are available, including the ability to define a looping region, to play a rectangular score portion, or to play the currently selected items (which may be noncontiguous). There is also the so-called offline playback, a kind of time-wise, virtually instantaneous dump (as opposed to the standard voice-wise dump, or the real-time, time-wise playback) of the musical score. This can be useful for processes depending on the temporal relations among the various voices. Typical usage examples include synthesizers or DSP processes requiring parameters that are too numerous or too complex to be easily controlled through a standard MIDI sequencer.

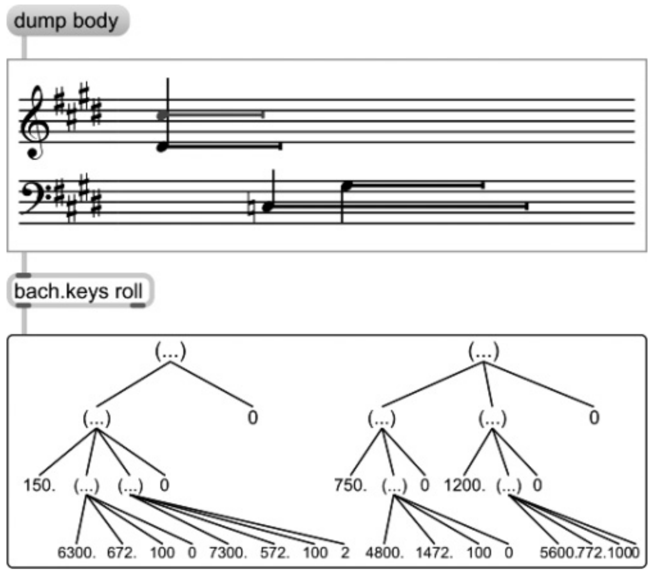
The graphical appearance of scores is highly customizable through the association of note data and metadata with graphical parameters, or—in extreme cases—through the layering of Max's standard drawing tools, such as the lcd object, on top of bach.roll or bach.score. This is made possible by the fact that the pixel position of each score item can be queried at any time. By taking advantage of these features, virtually any kind of graphical music representation can be displayed through bach.roll or bach.score.

In addition, both bach.roll and bach.score support the placement of markers and the assignment of symbolic tags to each score item, allowing tag-based selection, grouping, and editing of musical elements.

The bach.roll and bach.score objects can exchange whole scores with other objects in the form of lllls, whose structures essentially reflect the structure of the internal representation described above. After a header section containing a set of global information, such as clefs, keys, or the data types contained in the slots, the actual body of the score follows (see Figure 3). The body of the llll representing a score contained in a bach.roll object is structured in sublists, each representing a voice; each voice is structured in sublists, each representing a chord. Each chord sublist contains a global onset and a sublist of notes; each note contains information about its pitch, velocity, duration, plus some optional specifications, structured in sublists themselves, determining the enharmonic properties, the glissando breakpoint function, and all the metadata associated with the note. The body of a bach.score llll is potentially much more complex than a bach.roll llll because of the intricacies of the traditional rhythmic representation, such as grace notes, nested tuplets, or different beaming options. Still, for simple cases it closely resembles that of bach.roll, except for the fact that chords are contained in measures contained in voices, notes may have starting or ending ties, duration is a property of chords rather than notes, and some other minor differences. Users can algorithmically edit a score either by working upon its complete llll representation, or by separately addressing its parameters such as pitches, duration, or slot content.

Both bach.roll and bach.score support microtones of arbitrary precision: semitones, quarter tones, and eighth tones are represented by their standard symbols; for all the other subdivisions, the deviation from the reference "white key" pitch can be represented as a fraction or in MIDI cents (see Figure 4). Moreover, the scores contained in

Figure 3. The score of a simple bach.roll shown in tree form. The four levels in the tree representation correspond to voices, chords, notes, and parameters of notes. The last element of each level (except the top level) is a flag determining, among the other things, whether the corresponding voice, chord, or note is either locked, muted, or soloed. The tree shows that the score has two voices, corresponding to the two sublists at the top level. The first voice contains one single chord and a 0 flag, meaning that the voice itself has no special properties; the chord has an onset of 150 msec, two notes, and the flag set to zero. The first note (that is, the lower one) has a pitch of 6,300 MIDI cents, a duration of 672 msec, a MIDI velocity of 100, and its flag set to zero. The second note (the higher one) has a pitch of 7,300 MIDI cents, a duration of 572 msec, a MIDI velocity of 88, and its flag is set to two, meaning that the note is muted (on a color display the note will be shown in blue to indicate the mute). Correspondingly, it is easy to recognize the second voice, containing two single-note chords, in the tree representation. This figure does not show the header section, containing data such as clefs, keys, or markers. The full score data, including header, can be obtained by substituting the `dump body` message with a simple `dump` message.



bach objects can be imported and exported in several file formats, such as standard MIDI files, MusicXML, LilyPond files, and PDF (actually rendered by LilyPond, provided that it is installed). Scores also can be directly imported and exported to and from OpenMusic and PWGL. The bach library is also able to read and write Sound Description Interchange Format (SDIF) files, but because of the complexity and flexibility of the SDIF specification their contents are not directly imported by the score editors. This task can, however, be accomplished through a family of abstractions belonging to the cage library.

Other User Interface Objects for Musical Representation

Two other user interface objects allow alternative musical representations. The `bach.circle` object is a classical clock diagram, useful to represent pitch classes or rhythmic positions on a looped grid. The object `bach.tonnetz` implements a *Tonnetz*, that is, a two-dimensional lattice diagram representing the space of pitches generated by two arbitrary diatonic intervals.

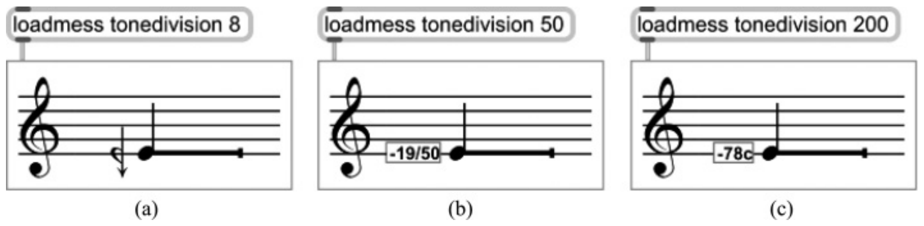
Generic Data-Processing Objects

A large subset of the bach library is aimed at processing lllls. Several of these modules are inspired either by their Lisp counterparts or by standard Max list processing objects, such as `zl`. In contrast to standard Max objects, though, not only is the llll structure correctly recognized, but it is possible, wherever meaningful, to perform each operation at all the depth levels of the incoming llll, or at specific depths. This is generally accomplished through the attributes "mindepth" and "maxdepth," which indicate the depth of the outermost and innermost elements, respectively, that should be affected by the operation. These depths can be calculated starting from either the root of the llll, or the deepest level of each of its branches. In the latter case, the depth is entered as a negative number. Among the other things, this allows the user to specify that a given operation must be performed at all levels, regardless of the overall depth of the llll, by setting the attributes respectively to 1 (that is, the root level) and −1 (that is, the innermost level of each branch of the llll). See Figure 5 for an example.

This subset of bach can be further divided into several categories:

1. Basic operations (rotation, reversal, flattening, splitting, chaining, etc.)
2. Retrieval of general information such as the length or the depth of an llll, or the data types it contains
3. Insertion, substitution, and retrieval of elements based upon positional or qualitative criteria
4. Set operations (union, intersection, difference, symmetric difference)

Figure 4. Three different `bach.roll` objects displaying the same note with pitch 6322.2 MIDI cents but with different graphical approximations set by the attribute "tonedivision." The deviation from E4, approximated to the closest eighth tone, using a conventional accidental notation (a); as a fraction, approximated to 1/50 of a whole tone (i.e, four cents) (b); as a difference in cents, or 1/200 of a whole tone (c). The internal representation of the pitch is never approximated unless the user explicitly asks to snap it to the microtonal grid.



5. Iteration over the elements constituting an llll, with several options allowing control of the behavior with respect to the structure and the contents of a single llll or several lllls to be iterated against each other
6. The collection of elements received sequentially, with the possibility of defining and navigating the structure of the llll that is being built

These two last categories, when combined, allow the implementation of virtually every process involving element-by-element data manipulation.

Specialized Data-Processing Objects

Other modules can operate only on appropriately structured lllls. For example, the `bach.expr` object can evaluate mathematical expressions on lllls composed solely of numbers, including rationals. There is also a family of modules that can perform matrix operations on lllls that have a matrix structure. Other objects will only be able to work, or at least to return meaningful results, if they are provided with lllls representing musical content of various kinds. For example, `bach.n2mc` will convert note names such as C4 or Re#2 into the corresponding values in MIDI cents. Of course, the GUI objects themselves belong to this broad family, as they can only operate upon lllls structured according to the syntax specific to each of them, such as the `bach.roll` and `bach.score` syntaxes described earlier. Some automatic conversions are made: For example, between numeric types (integers, floating point, and rational fractions) where necessary and meaningful, and between note names and the corresponding MIDI cent values (but only for the limited number of modules dedicated to notation).

Rhythmic Quantization

`bach.quantize` is a highly customizable tool designed to solve rhythmic quantization problems. In the simplest case, `bach.quantize` receives `bach.roll`'s proportional scores and converts them into a `bach.score`, taking as an additional input either some direct information about tempo and meter, or a set of markers from which to infer the metric information. Such markers are typically contained in the original `bach.roll`.

Once the meter and tempo are known, `bach.quantize` cuts the proportional score into chunks, essentially corresponding to the beat units, and tries to achieve a rhythmic quantization inside each chunk. This "box-oriented" approach implements the idea that the quantization algorithm must assure synchronicities at least on an important set of time instants (the beats by default, or some other custom set). Inside each box, `bach.quantize` runs a greedy backtracking algorithm that tries to optimize the onset error at each step. Each output duration is chosen from linear combinations of a basic set of durations, called *minimal units* and expressed as an attribute of the object. For instance, with minimal units of 1/16 and 1/12, any duration in the resulting score will be either a multiple of a sixteenth note, a multiple of a triplet eighth note, or, optionally, the sum of the two. The default behavior, however, is to allow only one of the minimal units inside each box. This implies that, for each box, all the resulting durations will be multiples of just one quantization unit (see Figure 6). This keeps the tuplet writing in the output simple and also allows the algorithm to run faster. The user can, however, change this behavior at any time, allowing mixed combinations of minimal units.

The attribute "Minimal Units" is probably the most important parameter that the user might

Figure 5. The reversal operation performed by `bach.rev` is limited to the second and third levels of depth. The root level and the fourth level are not affected by the reversal (lowercase letters still precede uppercase letters, and letter–number pairs are left unaltered).

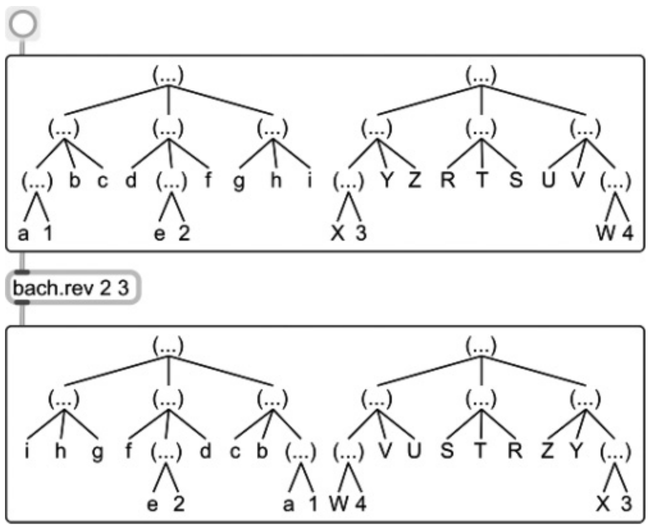
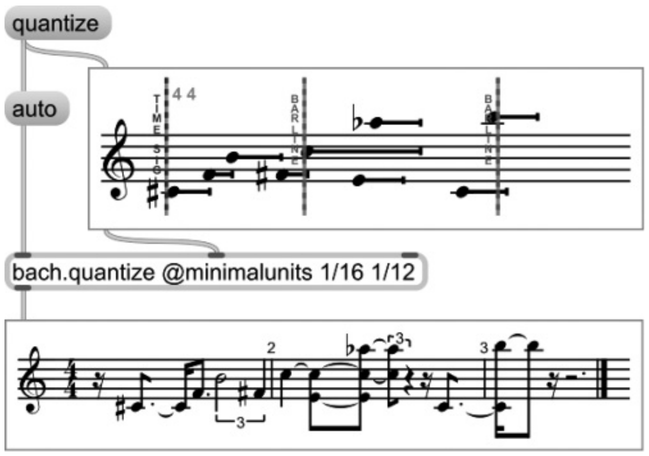

(...)
(...)
(...) b c d (...) f g h i (...) Y Z R T S U V (...)
a 1
e 2
X 3
W 4
bach.rev 2 3
i h g f (...) d c b (...) (...) V U S T R Z Y (...)
e 2
a 1 W 4
X 3

Figure 6. A simple quantization of a `bach.roll` into a `bach.score`, using the metric information inferred from the time signature and barline markers. The result will use only multiples of an eighth note, or multiples of a triplet eighth (i.e., one twelfth), as set by the attribute "Minimal Units."



want to modify. A configuration with many small minimal units would probably lead to an accurate, but slow, quantization process. With fewer, larger minimal units, the result would be more approximative, but the computation would be faster. The minimal units can also be assigned dynamically, depending on the density of events of each chunk. For instance, one might want to quantize fast, dense musical events using more-detailed rhythmic subdivisions; slower, sparser gestures could use a coarser quantization to yield simpler rhythmic figures.

As previously indicated, `bach.quantize` is a highly customizable tool, featuring a large number of attributes. For example, the quantization is, by default, calculated voice by voice. This means that synchronous events across different voices might not be preserved during the process. This behavior can be changed by activating the attribute "Preserve Synchronicities."

To give another example, `bach.quantize` by default renders as grace notes those rhythmic events that are too small to be represented normally. The attribute "Small Notes Policy" lets the user override this behavior.

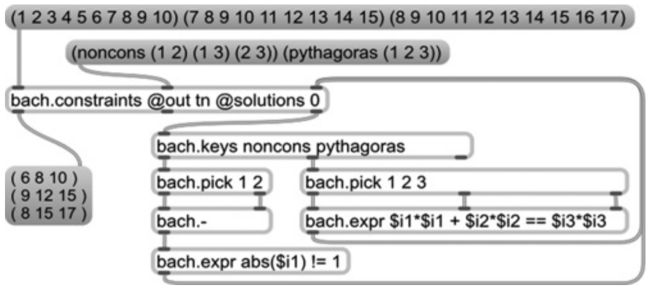
Constraint Programming

It is often convenient to state musical formalization problems in terms of the qualities of the desired result, rather than in terms of the steps necessary to generate it. Constraints programming is a programming paradigm widely used for this kind of problem, and most systems for computer-aided composition provide tools implementing it in some form, e.g., PWConstraints (Laurson 1993) in PatchWork, PMC (Anders and Miranda 2011) in PWGL, OMClouds (Truchet and Codognet 2004) or OMCS and Situation (Bonnet and Rueda 1998) in OpenMusic. At the extreme of this range, some software systems are entirely devoted to musical constraint programming, such as Strasheela (Anders, Anagnostopoulou, and Alcorn 2005).

The bach library contains a simple constraint-solving object, called `bach.constraints` (see Figure 7). The main design feature of `bach.constraints` is that it does not require any textual coding proper. The object receives the domains of the variables and a list of the relations between each variable (represented by a numeric index) on the one hand, and the rules that directly affect its labeling (rules are represented by arbitrary symbolic names), on the other. During the computation, the object iteratively sends proposals of partial assignments of values taken from the domains to variables out of its lambda outlet. Each proposal is evaluated in a lambda loop according to one named rule. The result of the evaluation is subsequently returned to the lambda inlet of `bach.constraints`.

Two constraint-solving engines are currently available in the `bach.constraints` object. One

Figure 7. A simple example of constraint solving via `bach.constraints`*. The leftmost inlet receives the domain for each of the three variables of the problem (specified by sublists of the main llll); the middle inlet defines the names and the scopes of the constraints; finally, the rightmost inlet and outlet implement the constraint rules via a lambda loop. We define a "noncons" constraint acting on pairs of variables, and corresponding to the request that no variables be consecutive numbers; and a "pythagoras" constraint acting on the three variables at the same time. Combinations of values, associated with the name of the rule against which they have to be tested, are sent out of the lambda outlet, routed according to the rule name, and evaluated; the result of each evaluation is re-injected in the lambda inlet. In this example, the search is performed by the backtracking engine, but the hill-climbing engine might be invoked by setting the attribute "weak" to one.*



```
(1 2 3 4 5 6 7 8 9 10) (7 8 9 10 11 12 13 14 15) (8 9 10 11 12 13 14 15 16 17)
(noncons (1 2) (1 3) (2 3)) (pythagoras (1 2 3))
bach.constraints @out tn @solutions 0
bach.keys noncons pythagoras
bach.pick 1 2
bach.pick 1 2 3
bach.-
bach.expr $i1*$i1 + $i2*$i2 == $i3*$i3
bach.expr abs($i1) != 1
( 6 8 10 )
( 9 12 15 )
( 8 15 17 )
```

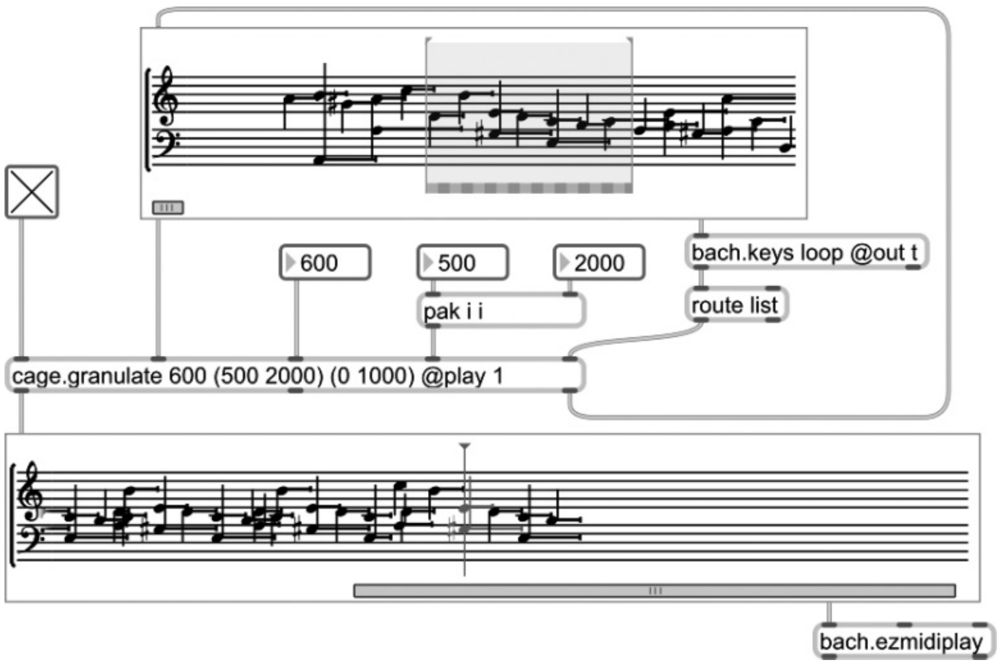
is based on an optimized backtracking algorithm, the other is based on a hill-climbing algorithm with taboos.

The first of these is useful for solving classic constraint-satisfaction problems and only accepts Boolean values (represented as zero or one) in the lambda inlet. Therefore, it can only return solutions solving the problem completely, and it has the ability to find all the problem solutions if required. In principle, the backtracking algorithm is guaranteed to find a solution if it exists, although in practice the time necessary for the computation can be extremely long for large problems. A feature that is currently under development, and that is only applicable to the backtracking engine, is the ability to perform searches on relation domains, i.e., a sort of multilevel search in which the actual variable domains at one level depend on the solution of a constraint-satisfaction problem at the level below. This approach can be useful for representing, for example, harmonic problems in which one set of rules governs a sequence of harmonic classes, and another set of rules governs how pitches of different harmonies interact with each other. Tonal harmony is one classic case of this kind of problem.

The second constraint-solving engine, hill climbing, has two main fields of application. First, it can solve weight-based optimization problems, in which, instead of searching for exact solutions, the goal is to find "reasonably good" solutions (i.e., solutions for which it is reasonable to assume that they are among the best possible within the problem space). In this case, the result of the evaluation that is fed back into the lambda loop is not a Boolean, but rather a numeric value indicating the quality of the partial assignment being considered. Second, in most cases it can solve classic constraint-satisfaction problems much more efficiently than the backtracking solver, as long as a single solution is required. On the other hand, because the hill-climbing algorithm has only a very local notion of the topology of the search space, it is in principle impossible for it to find all the solutions of a problem, and it is possible that no exact solution is ever found if the problem is a particularly hard one.

As stated earlier, all the rules for `bach.constraints` are expressed outside the object itself, in the lambda loop: This means that no optimization based upon the specifics of the particular problem is possible. Moreover, the communication between objects in a Max patch is relatively slow, and even slower if the data to be passed are lllls, because of the complexity of the data structure itself. Although a performance penalty is not noticeable in most practical cases, it often is with `bach.constraints` because of the great number of lllls that potentially traverse the lambda loop. This makes `bach.constraints` impractical if large problems need to be solved, especially when compared with engines based on textual coding, such as Gecode (www.gecode.org), which has been interfaced to OpenMusic and Max by Lemouton (2011). As an initial strategy to improve usability, if not efficiency, the lambda loop of `bach.constraints` can run in a custom thread if the "parallel" attribute is set to 1. Indeed, this is explicitly forbidden by the Max documentation for developers, as it can cause serious stability problems to the whole Max environment. On the other hand, if some simple precautions are taken (such as not deleting or modifying objects and connections in the lambda loop, nor saving or closing the patch while the process is running), such a parallelized behavior appears to be perfectly reliable, and it can be very useful, allowing Max to perform other operations while long searches are performed.

Figure 8. An example of real-time score granulation generated by the `cage.granulate` module. The grains are extracted from the loop region of the top `bach.roll` and added in real time to the `bach.roll` at the bottom of the patch. The resulting `bach.roll` is also being played. The grains have a distance of 500 msec and a duration between 500 and 2000 msec.



We are currently considering other, more specific options to improve efficiency, at least for some categories of problems. At the same time, we think that the nontextual approach, consistent with the general Max syntax and the lambda-loop design pattern, is well suited for quick prototyping, in particular for users without a strong traditional programming background.

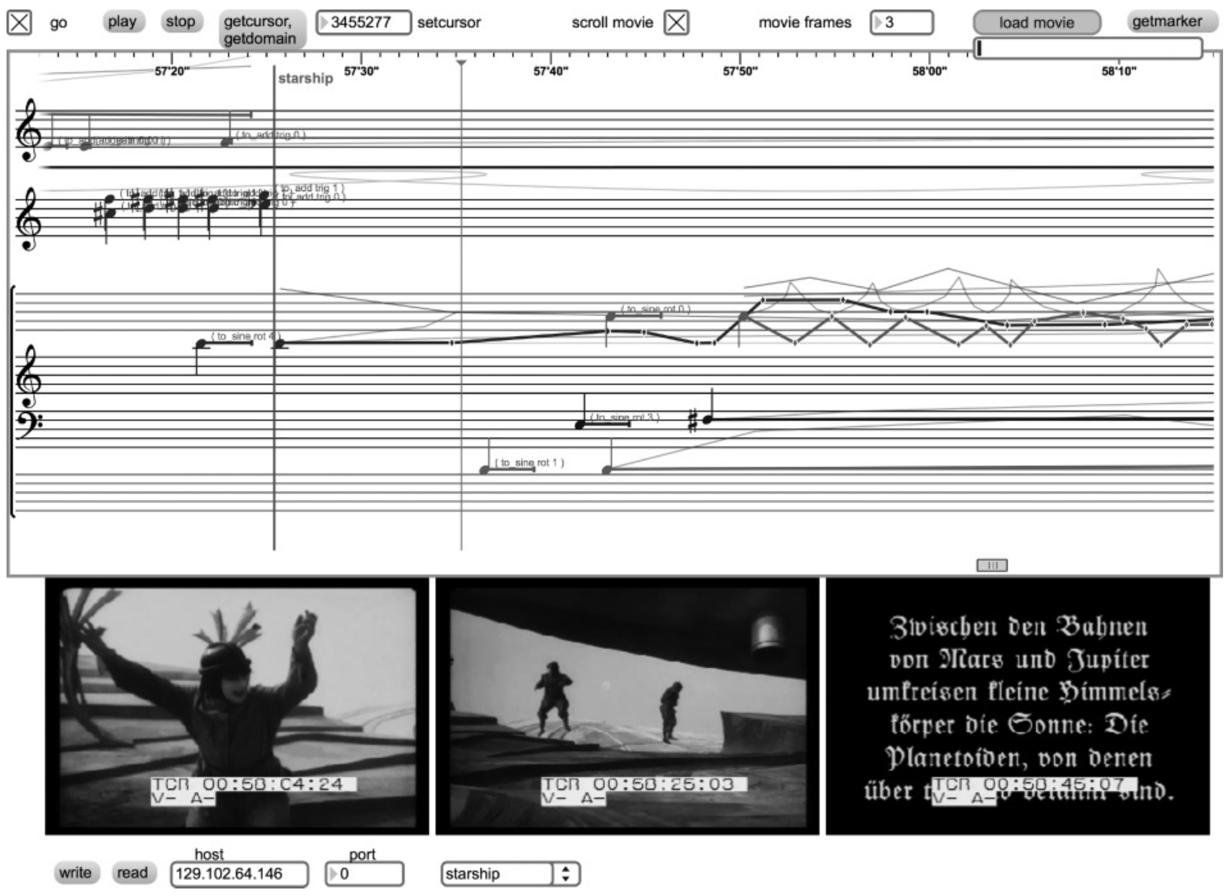
The Bach Family

Possibly the most ambitious project linked to bach, for the short and medium term, involves situating the bach library inside a wider family of tools for real-time computer-aided composition in Max. The idea is that bach might constitute the core of the family, and each additional library would expand bach to deepen some aspect of interactive computer-aided composition. All the bach family will inherit from bach the basic principle of allowing computer-aided composition processes in real time, as well as some standard mechanisms and paradigms, such as the lambda-loop design pattern, or the fact that communication between the different modules happens mostly by means of lllls.

The Cage Library

Currently, the only project built starting from bach is the cage library (Agostini, Daubresse, and Ghisi 2014). Whereas most bach modules are tools for low-level or conceptually basic operations, cage is aimed at easing higher-level manipulation of symbolic musical data (see, for instance, Figure 8) and solving typical problems in computer aided-composition. These include generation of pitches, generation and processing of melodic profiles, symbolic processes inspired by digital signal processing, harmonic and rhythmic interpolations, cellular automata, L-systems, tools for musical set theory, and tools for score generation and handling. The cage library, whose development is supported by the Haute École de Musique in Geneva, has a chiefly pedagogical function, because all the modules in the library are abstractions, lending themselves to easy analysis and modification.

Figure 9. In Andrea Agostini's music for the film Wunder der Schöpfung, *the timeline for a complex, custom sound synthesis system is entirely managed by one* `bach.roll` *object. In this example, note pitches are ignored by the synthesis process. Notes are, thus, only containers for numeric data, breakpoint functions, and textual instructions to be routed to the synthesizers. The synthesis timeline is frame-synchronized with the video file of the film during both editing and playback, by querying the* `bach.roll` *object for the boundaries of the displayed section of the score.*



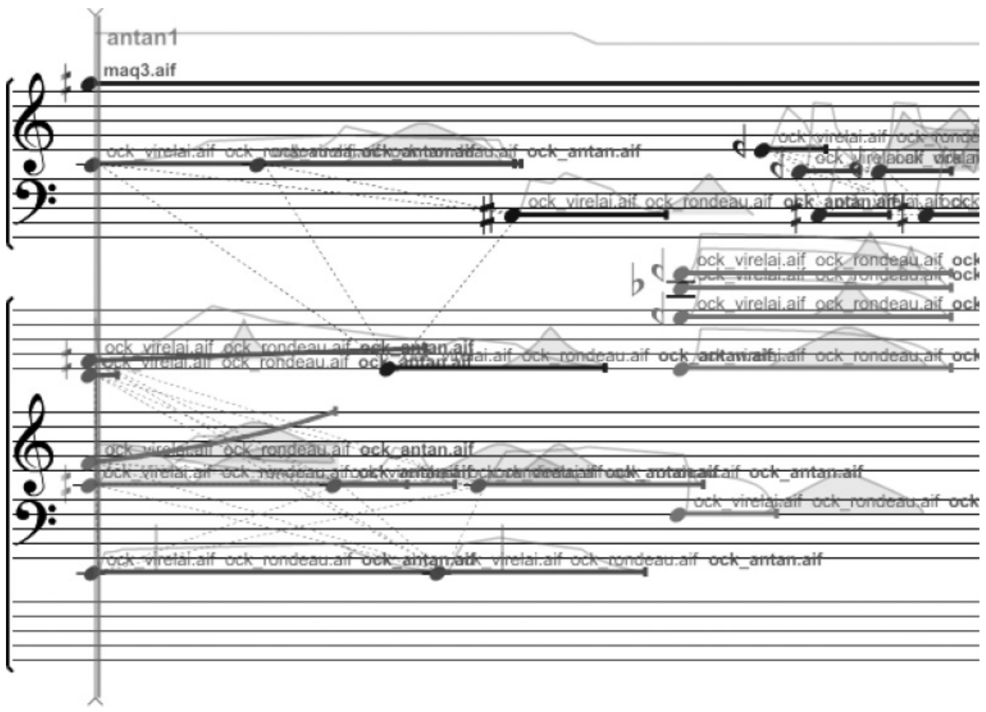
Perspectives

Imagining and sketching the future members of the bach family is a thrilling and creative task, although such members still belong to the realm of planned possibilities.

For one thing, by design bach only implements essentially traditional representations of music. This is both its strength and its limitation. It is a strength, inasmuch as it allows bach to be a general-purpose, highly adaptable tool. It is a limitation since we think that, as a research project into performative paradigms in computer-aided composition, it should also cover experiments with new interfaces alongside more traditional ones. To fill this gap, a library named "dada," completely focused on reactive, nonstandard GUI interfaces for real-time computer-aided composition, is currently under study. Among other things, this library might also feature some high-level, graphical tools to facilitate a semi-automatic, "intelligent," user-friendly quantization, also by taking inspiration from the OM-Kant library for OpenMusic (Agon et al. 1994), which is now no longer supported.

Another member of the bach family, which we are considering as a potential future project, might contain DSP objects linking sound generation and processing with symbolic representations. It might feature, for instance, real-time audio analysis tools designed and implemented to communicate quickly and easily with `bach.roll` and `bach.score`.

Figure 10. A portion of the electronic score for Daniele Ghisi's Chansons, *consisting of a* `bach.roll` *object, whose notes are grouped into larger structures to better represent electronic gestures in the score. (On-screen, the elements are differently colored.) Some metadata, such as amplitude envelopes and filters, are also displayed.*

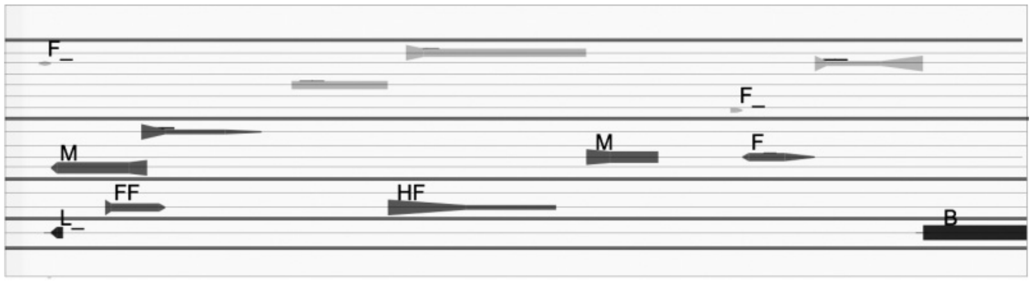


Relation to Other Software

The bach library has a strong relation to different families of software. For one thing, its design is explicitly inspired by the family of Lisp-based computer-aided composition software originating with PatchWork (Laurson and Duthen 1989), and in particular OpenMusic (Agon 1998; Assayag et al. 1999) and PWGL (Laurson and Kuuskankare 2002). The "reductionistic" operational and representational paradigm of bach, based upon the hierarchical arrangement of the different parameters of a score and their individual manipulation, as well as some aspects of the system ergonomics, is directly borrowed from such environments. For example, the arrangement of the inlets and outlets of `bach.roll` and `bach.score` mostly matches that of the OpenMusic objects `chord-seq` and `voice`, with the left-most inlet/outlet pair handling global information, and a set of inlet/outlet pairs handling information about separate musical parameters (pitch, duration, velocity, etc.). This legacy makes learning bach easier for users accustomed to OpenMusic or PWGL. Furthermore, to facilitate the exchange of musical scores, a direct import and export mechanism has been developed from and to OpenMusic and PWGL. (In the case of the latter, the conversion also preserves most of the metadata contained in the note slots.) Apart from these similarities, however, all the modules of bach are designed to operate in the dataflow, real-time paradigm of Max, which is substantially different that of OpenMusic and PWGL, and many of the possibilities that the bach paradigm offers are simply not achievable in the aforementioned environments (see, for instance, Figure 8). On the other hand, there are still domains in which those environments largely outscore bach: In particular, processing-intensive computations, such as combinatorial processes, and operations that are inherently non-real-time, such as batch processing or generation of sound files.

The bach library shares with projects such as `psw.uscore` (www.peterswinnen.be), MaxScore (Didkovsky and Hajdu 2008), and InScore (Fober, Orlarey, and Letz 2012) the ability to operate with symbolic musical representation in real time; it

Figure 11. A `bach.roll` displayed in a patch used in Carlo Barbagallo's Tatàmme, *written for the* Table de Babel, *an instrument set conceived, built, and played by Jean-François Laporte. The graphical parameters of `bach.roll` are adjusted to display a nonstandard score that is generated in real time during the performance. Besides the `live.line` objects used to display the custom-positioned horizontal lines, no layering of other Max graphical widgets has been used.*



shares with note~ (Resch 2013) and, to some extent, Antescofo (Cont 2008a) the goal of providing Max with advanced sequencing capabilities; it shares with FTM (Schnell et al. 2005) and MuBu (Schnell et al. 2009) the characteristic of implementing new data structures that are more powerful and complex than those natively provided by Max. It is a fact that several aspects of all these systems, including bach, overlap. At the same time, each has its own peculiar features that strongly distinguish it from the others. For example, the quality and flexibility of musical typesetting of MaxScore and InScore are clearly higher than those of bach; the event-sequencing capabilities are much more advanced in note~ and Antescofo; the FTM object system is richer and more complex; and the MuBu data containers are exactly tailored for the data types they are meant to hold. The idea at the basis of bach is that all its modules form a coherent computer-assisted composition environment, and its very goal is to make live computer-assisted composition, in a responsive, interactive world.

At least two other software projects share this same perspective: Peter Elsea's LObjects (peterelsea.com/lobjects.html) and Karlheinz Essl's RTC-lib (www.essl.at/works/rtc.html), both predating bach by several years. The former is mostly devoted to enhancing the list-processing capabilities of Max, with only a small number of objects dealing with problems more specifically related to MIDI. The latter implements both generic data processing and strictly compositional operations, with a leaning towards post-serial and aleatoric techniques. It is a less powerful system than bach, as it does not provide graphical user interfaces, sequencing tools (except for a simple step sequencer), or advanced operations such as constraint solving or rhythmic quantization. Its data representation capabilities are confined to the standard Max list—which, in turn, means that RTC-lib does not have a notion of score. On the other hand, all this makes RTC-lib extremely lightweight, and its composing modules are easy to integrate individually into generic patches. In contrast, bach modules are chiefly designed to interact with each other.

Current Status and Future Developments

The development of bach has been uninterrupted since 2010, and during the past few years the library has earned a user base whose size we estimate to be around 1,000 people, according to the download statistics and the interactions on the dedicated forum. The bach library can be downloaded free of charge from the official project Web site (www.bachproject.net) and runs in conjunction with Max on both Windows and Mac OS X.

A large part of our development efforts go in the direction of reliability and usability. Because of the massive number of features of some modules, particularly `bach.score` and `bach.roll`, a custom, tag-based documentation system has been implemented, and it is being maintained to include detailed descriptions and examples for every new feature of every module of the system. Furthermore, each object and abstraction is documented by a detailed help patcher and a complete reference sheet. Finally, a broad collection of tutorials covers most key concepts of the system and exemplifies some typical usage scenarios.

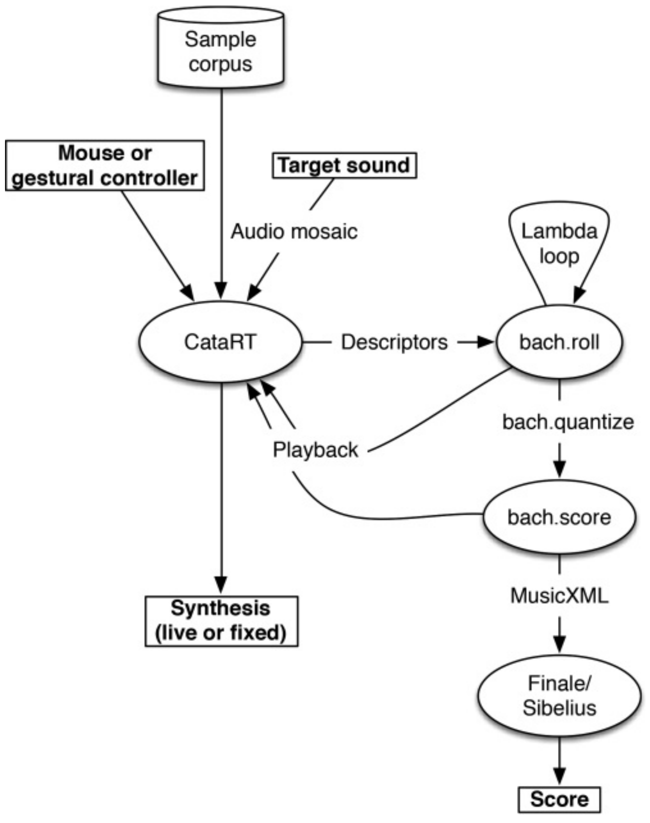


Figure 12. Aaron Einbond and Christopher Trapani are currently developing a tool for controlling corpus-based concatenative synthesis through the interaction of CataRT *and bach. Grains played by* CataRT *are recorded in a* `bach.roll` *object as notes whose properties and metadata represent the various parameters and descriptors associated with the grains themselves. The content of the* `bach.roll` *object can subsequently be played through* CataRT*, edited in place with a lambda loop, or used for other processes such as control of synthesis or score generation. (Image by Aaron Einbond; see Einbond et al. 2014)*

Medium- to long-term development plans include new major features to be implemented. One of the most important issues currently under study is the absence of a page layout for score display: both `bach.roll` and `bach.score` represent musical scores as a single, scrollable staff system, and the only way to achieve a sort of "page view" is to use several `bach.score` objects, each showing one portion of the same score. Although technically feasible, this approach is cumbersome and inefficient. A direct consequence of the absence of a page representation is the fact that it is currently difficult to print scores or generate PDFs directly from bach. One possible workaround for this is to export a bach score to a music engraving system, and to have that system manage page layout. As alluded to earlier, this process can be done automatically through LilyPond. Thus, we are currently researching the possibility of implementing a notion of page into `bach.score` and `bach.roll`, although we do not consider bach as a tool for music engraving, but rather for interactive manipulation of musical materials.

Another major issue, already mentioned, is the fact that the main strength of Max lies in its interactivity and the quality of its graphical interface, rather than ease of representing complex algorithms or computational power. In fact, during the implementation of the cage library, we ourselves realized that processes that could have been easily represented in a text-based programming language required very complex patches to be implemented through Max and bach. At the same time, as discussed earlier, some specific operations may be slower than optimal because of the mechanisms of message passing in Max and llll passing between bach objects, and because of the impossibility of performing direct, low-level operations on list nodes. The only currently viable solution to these problems is implementing such operations as external objects written in C. For this reason, we provide an SDK allowing other people to code their own externals while making use of the bach data structures. Still, C coding for Max is intimidating for most Max and bach users, and problematic because subtle mistakes can cause serious stability problems for the whole system. We are therefore studying the possibility of implementing a high-level, Lisp-inspired, textual language allowing users, especially those familiar with Lisp, to write routines for llll manipulation. This language will be JIT-compiled on the fly and will therefore run at native speed.

Examples

Figures 9 through 12 present concrete examples of how bach can be used: a flexible sequencer for controlling a video-synchronized synthesis system (see Figure 9), a hierarchically organized electronic score (see Figure 10), a graphical score obtained by refining the display parameters of a `bach.roll` object (see Figure 11), and a tool for controlling corpus-based concatenative synthesis (see Figure 12).

Conclusions

After more than five years of development, the basic structure and working principles of bach are rather stable. At the same time, it is hard to foresee whether and when a version of the library will be considered definitive. Many are the future challenges, including our wish to protect the total independence of the work, which has never received any formal institutional support, and at the same time to reach the largest possible user base and collect their suggestions and requests.

Acknowledgments

The conception and the development of the bach library have been partially undertaken while the authors were students at the Course in Composition and Musical Informatics at IRCAM, and then as composers-in-research at that institution.

We wish to thank Carlos Agon, Arshia Cont, Éric Daubresse, Emmanuel Jourdan, Serge Lemouton, Jean Lochard, Mikhail Malt, and all the other people at IRCAM who have supported and encouraged the work, although it was not an internal IRCAM project.

We also wish to thank Sara Adhitya, Jeremy Bernstein, Davide Bordogna, Joshua Kit Clayton, DaFact, Phil Gaskill, Eric Grunin, Mika Kuuskankare, Andrew Pask, Andrea Sarto, and David Zicarelli.

References

Agon, C. 1998. "OpenMusic: Un langage visuel pour la composition musicale assistée par ordinateur." PhD dissertation, Université Paris VI.

Agon, C., et al. 1994. "Kant: A Critique of Pure Quantification." In *Proceedings of the International Computer Music Conference*, pp. 52–59.

Agostini, A., E. Daubresse, and D. Ghisi. 2014. "cage: A High-Level Library for Real-Time Computer-Aided Composition." In *Proceedings of the International Computer Music Conference*, pp. 308–313.

Anders, T., C. Anagnostopoulou, and M. Alcorn. 2005. "Strasheela: Design and Usage of a Music Composition Environment Based on the Oz Programming Model." In P. Van Roy, ed. *Multiparadigm Programming in Mozart/Oz*. Berlin: Springer, pp. 277–291.

Anders, T., and E. R. Miranda. 2011. "Constraint Programming Systems for Modeling Music Theories and Composition." *ACM Computing Surveys* 43(4). Available online at dx.doi.org/10.1145/1978802.1978809 (subscription required).

Assayag, G., et al. 1999. "Computer Assisted Composition at IRCAM: From Patchwork to OpenMusic." *Computer Music Journal* 23(3):59–72.

Bonnet, A., and C. Rueda. 1998. "Situation: Un Langage visuel basé sur les contraintes pour la composition musicale." In M. Chemillier and F. Pachet, eds. *Recherches et applications en informatique musicale*. Paris: Hermes Science, pp. 65–74.

Carpentier, G., and J. Bresson. 2010. "Interacting with Symbol, Sound, and Feature Spaces in Orchidée, a Computer-Aided Orchestration Environment." *Computer Music Journal* 34(1):10–27.

Cont, A. 2008a. "ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music." In *Proceedings of the International Computer Music Conference*, pp. 33–40.

Cont, A. 2008b. "Modeling Musical Anticipation: From the Time of Music to the Music of Time." PhD dissertation, University of California, San Diego.

Didkovsky, N., and G. Hajdu. 2008. "MaxScore: Music Notation in Max/MSP." In *Proceedings of the International Computer Music Conference*, pp. 483–486.

Einbond, A., et al. 2014. "Fine-Tuned Control of Concatenative Synthesis with CataRT Using the bach Library for Max." In *Proceedings of the International Computer Music Conference*, pp. 1037–1042.

Esling, P., and C. Agon. 2010. "Composition of Sound Mixtures with Spectral Maquettes." In *Proceedings of the International Computer Music Conference*, pp. 550–553.

Fober, Y., Y. Orlarey, and S. Letz. 2012. "INScore: An Environment for the Design of Live Music Scores." In *Proceedings of the Linux Audio Conference*, pp. 47–54.

Laurson, M. 1993. "PWConstraints." In *Atti del X Colloquio d'Informatica Musicale*, pp. 332–335.

Laurson, M., and J. Duthen. 1989. "Patchwork, a Graphical Language in Preform." In *Proceedings of the International Computer Music Conference*, pp. 172–175.

Laurson, M., and M. Kuuskankare. 2002. "PWGL: A Novel Visual Language Based on Common Lisp, CLOS and OpenGL." In *Proceedings of the International Computer Music Conference*, pp. 142–145.

Lemouton, S. 2011. "Using Gecode to Solve Musical Constraint Problems." In G. Assayag and C. Truchet, eds. *Constraint Programming in Music*. Hoboken, New Jersey: Wiley, pp. 103–132.

Puckette, M. 2004. "A Divide Between 'Compositional' and 'Performative' Aspects of Pd." In *Proceedings of the First International Pd Convention*. Available online at puredata.info/community/conventions/convention04/lectures/tk-puckette/puckette-pd04.pdf/at_download/file. Accessed January 2015.

Resch, T. 2013. "note˜ for Max: An Extension for Max/MSP for Media Arts and Music." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 210–212.

Schnell, N., et al. 2005. "FTM: Complex Data Structures for Max." In *Proceedings of the International Computer Music Conference*, pp. 423–426.

Schnell, N., et al. 2009. "MuBu and Friends: Assembling Tools for Content Based Real-Time Interactive Audio Processing in Max/MSP." In *Proceedings of the International Computer Music Conference*, pp. 423–426.

Taube, H. 1991. "Common Music: A Music Composition Language in Common Lisp and CLOS." *Computer Music Journal* 15(2):21–32.

Truchet, C., and P. Codognet. 2004. "Solving Musical Constraints with Adaptive Search." *Soft Computing* 8(9):633–640.