A compendium of bach's shortcomings as it turns ten — along with a few ideas on how to overcome them

Daniele Ghisi
University of California, Berkeley
CNMAT
danieleghisi@berkeley.edu

Andrea Agostini
Conservatory of Turin
andrea.agostini@conservatoriotorino.eu

ABSTRACT

The *bach* ecosystem is a collection of libraries for real-time computer-aided composition in Max, which we started developing around 2010. Over the following years, it has gained a certain momentum within the computer music community as a reactive alternative to the traditional Lisp-based environments. As *bach* turns ten and becomes open source, it is time to describe not only the features it contains, but also the issues that affect it: after presenting the motivations behind the project, this article details a numbers of choices we regret, along with the solutions we have implemented (or we have planned to implement), in the hope that it may be of interest for *bach* users puzzled by some apparent and substantial weirdnesses of *bach*, and of use to people willing to undertake a similar enterprise in the future.

1. INTRODUCTION

Our first ideas about the need of a real-time computer-aided composition tool were discussed around 2008, when we were both students at the Composition course at IRCAM. Although we were quite naive in terms of knowledge of music technology and musical informatics, we both had a certain background in computer programming and musical formalisation, which sparked an immediate interest in the capabilities of OpenMusic in terms of symbolic manipulation of music. At the same time, we were fascinated by the potential of real-time interaction through Max, and we started to feel the need for a stronger link, or some sort of common ground, between these two worlds, which showed astonishing superficial similarities (both required connecting boxes representing data and operations over a graphical canvas) and, at the same time, profound differences at a deeper level (it can be said that their evaluation models are opposite, data-driven vs. demand-driven). We were by no means the only people gravitating around IRCAM who where experiencing such feelings: among the others, a renowned composer such as Philippe Manoury had published both an internal report and a post on his personal blog [1] denouncing what he perceived as the lack of a fundamental reflection upon the actual needs of composers interested in both real-time sound processing and score-based instrumental and electronic composition.

When we actually started undertaking the development of what would later become *bach* [2, 3] — which, in the beginning, was only planned to be a graphical notation editor and a general tool for manipulating Lisp-inspired tree structures meant to represent scores — we did it in a really amateurish way, without considering lots of implications. Still, we somehow had the beginner's luck of taking a couple decisions which, in hindsight, have proven being the right ones: working inside Max, rather than building a new system from scratch; and trying to abide as much as possible by Max's own conventions, rather than trying to force into Max the underlying computational principles of OpenMusic. We did consider the latter option at some point: we studied the feasability of a demand-driven ecosystem in Max, in which evaluation would be triggered 'from the bottom' rather than 'from the top'; and we made some attempts at basing our own work upon Brad Garton's Lisp binding for Max, the *maxlispj* object [1] . Fortunately for the project, neither route was successful, and we found ourselves forced to take a much more 'Maxian' than 'Lispian' way in the end.

The Lisp influence has remained nonetheless in the data type we chose for representing our scores, the so-called *Lisp-like linked list* (or *llll* for short) — a list with parentheses representing levels of hierarchy across elements. More recently, the parentheses have by default become square brackets (although the old syntax is still supported), to facilitate the coding mechanisms of `bach.eval`, a new module implementing a small scripting language designed to operate on *llll*s. The majority of *bach* modules are designed to operate on this tree structure, in a similar manner to what objects of the `zl` family do on regular Max lists.

The relation with Max is now one of the foundational tenets of *bach*: porting it to another system would essentially mean reimagining it from scratch, and making a separate, standalone application out of it would amount to create an obscure cousin of OpenMusic. It is true that Max is a commercial, closed-source application, and this has a number of actual and potential drawbacks: for all the friendly relations we maintain with several persons in the Max development team, who have been vocally and actively supportive of our work on various occasions, we do not have access to Max's source code, and we are susceptible to any minor or major commercial and technical choice of Cycling '74 and its parent company, Ableton. On the other hand, Max is a mature software tool, very actively main-

[1] http://music.columbia.edu/ brad/maxlispj/

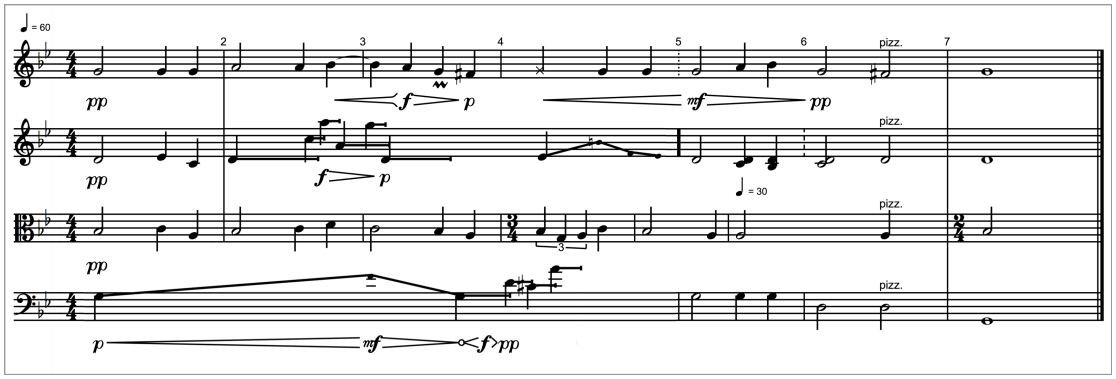

Figure 1. Imaginary *bach* editor that could have combined `bach.score`'s standard notation with `bach.roll`'s ability to represent proportional notation.

tained and developed; it has an excellent user interface, which is crucial as a great part of our work relies upon it; its public API is flexible enough to accommodate for *bach*'s complex and somewhat unorthodox features; and, last but not least, it has a vast and thriving community of users, which has been a fertile ground for *bach*'s own user community to grow.

There could be one obvious alternative host environment for *bach*: Pure Data. Pd shares with Max not only the basic computational paradigm, but also a large number of modules; moreover, its API is very similar to that of Max. Its graphical user interface is somewhat cruder, but the fact that it is an open source project could open interesting possibilities: one could even imagine a fork of Pd natively integrating *bach*, thus removing the need for some workarounds that make the Max version less seamless than it could ideally be. In principle, the C/C++ codebase of *bach* could be ported to the Pd API with a manageable amount of effort, provided that a few basic but not unsurmountable incompatibilities are overcome. But *bach* also contains a vast collection of Max patches — abstractions, help files, examples, tutorials and more — which should be rewritten in Pd from scratch, sometimes only taking inspiration from the originals. Since the *bach* code is now open source, anyone may decide to fork it into a child project and finally start a Pd version, to be maintained and upgraded separately. The fact that *bach* is the forefather of what is now a family of libraries [4] (*cage* and *dada* currently being the other members) is an additional complication in the porting, but does not make it infeasible.

All things considered, we defend the idea of having built *bach* within Max: it is not a prison, rather an opportunity to use notation in interactive scenarios we did not anticipate.

But there have been other decisions we regret or that, although still appearing to us as the least contemptible solution, we know don't represent an ideal scenario and might be overturned in some future, when some Columbus' egg is found or, more likely, when someone more skilled than ourselves suggests us the right idea.

In this article, inspired by the seminal article [5], we shall run through a few of these critical choices. We hope that this will cast some light upon some of the weirdnesses of *bach*, for the amusement and perhaps the convenience of *bach* users; that this examination of conscience may constitute an interesting case study; and, finally, that someone trying to undertake a similar enterprise may find our considerations useful — we certainly would have, had we read them ten years ago!

2. NOTATION OBJECTS

2.1 How many editors?

At the forefront of the *bach* library are two notation objects: `bach.roll` and `bach.score`, respectively dedicated to representing scores in proportional and classical notation. A certain number of initial choices were a consequence of a sort of adaptation to Max of policies inspired by OpenMusic and PWGL; therefore, back then, the idea of having two different editors for proportional and traditional notation made sense, inasmuch as it allowed, among other things, easy bridging and communication across these systems. In retrospect, one may wonder whether having a single object merging the two behaviors may have led to more interesting usage cases. This would have been at the expense of efficiency: although the two objects share most interface functions, `bach.score` is largely more complex and CPU-consuming. We have tried to narrow the difference in the paradigms by allowing `bach.score` to be displayed in proportional notation (with the same spacing as `bach.roll`), and vice versa by allowing non-uniform temporal grids for `bach.roll` (among other things, one can warp its temporal grid to match the spacing of a specific `bach.score`). The two object can be kept graphically aligned, so that one may use them almost as 'voices' with different notation styles; this is however not a replacement for a truly 'mixed' object, featuring proportional notation in some segments and traditional notation in others. We have often mused about having proportionally defined measures in a future version of `bach.score`, which would partially bridge the gap (see Fig. 1), but for the time being this is not the case.

2.2 Which inlets and outlets?

Inspired by the Patchwork family of computer-aided composition tools, we decided to equip *bach*'s notation objects

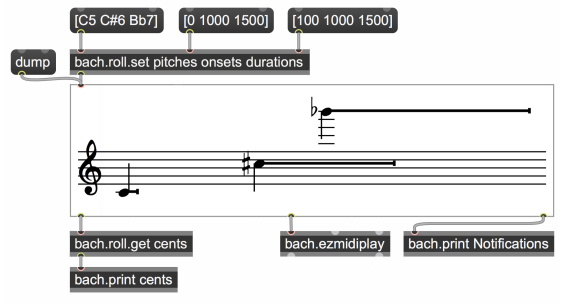

Figure 2. Imaginary behavior of `bach.roll` with no separate inlets/outlets; separate parameters may have been set and get through helper tools.

with a set of so called 'separate' inlets and outlets, where one would input or retrieve the information about single parameters of the score (onsets, pitches, durations, and so on). This seemed a straightforward decision at the time, but it became less and less obvious in the following years, as we realized that working with messages modifying the state of the object in-place is often more convenient than inputting or retrieving separate parameters. Moreover, musical representation in OpenMusic is based upon the concept of classes; musical notation editors such as *chordseq* are a graphical representation of the classes themselves; and the main inlet/outlet pair actually sets and returns the specific instance for the class, whereas the others are setters and getters of individual properties. The paradigm of *bach* is profoundly different: an editor is just an editor, and the data it represents is just a specifically formatted *llll*, so the actual motivation for the existence of whole-score and parametric inputs and outputs vanishes.

In retrospect, we could have made the cleaner choice to drop the separate inlets and outlets altogether, hence approaching the notation objects to a scripting paradigm—and, more profoundly, moving away from the idea that a score is simply a direct product of orthogonal parameters. Users would still have had the chance to introduce or retrieve separate parameters via helper modules similar to `bach.beatbox` and `bach.beatunbox` (see Fig. 2).

2.3 How to input single notes and chords?

As stated above, we decided not to copy OpenMusic's hierarchy of classes, featuring individual editors for notes, chords, sequences of chords, and polyphonic sequences of chords. In *bach*, all these four editors are subsumed into the single `bach.roll`. The idea is that if one needs a single note, one still has to produce a proportionally notated score with a single voice, having a single chord, bearing that note. This is convenient because there is a single type of editor; on the other hand, it requires additional infrastructure in order to display the note properly. We have imagined that in the future one may have a set of abstractions easing the creation, display and manipulation of simple musical items.

2.4 What syntax?

Designing an appropriate syntax for a score is certainly not a simple task, as musical scores are far from being unambiguous, rationally structured collections of information. For one thing, one often ends up having not just one, but a few variations of the same syntax to be used in slightly different scenarios (and *bach* was not exception to this). All in all, we are not unhappy with the syntax choices we made, with one important exception: most notation items feature in their data a numerical flag (namely, a bitmask) representing their state and detailing, among other things, whether the item is in a solo, mute or lock state. This flag being almost always zero, the syntax is very often unnecessarily long. Whenever we teach it in classes or seminars, it is a bit cumbersome to explain that an additional integer number always shows up, representing some secondary properties in a way that is hard to grasp for users unfamiliar with bit masks. We think we should have decided to encode those properties in a different way, such as sublists of the kind `[flags ...]`, using symbols or letters instead of integers, and only when the notation item had some non-trivial flag to detail. As of now, it is not easy to change the syntax without breaking backward compatibility.

3. DATA TYPES

3.1 *llll*s or dictionaries?

When we started developing *bach*, in the Max 5 era, the only data structures Max objects could directly exchange with each other were plain lists and Jitter matrices, and it was immediately clear that neither was apt to the musical score representation we were working upon. The Max API also provided a few more advanced data structures: a doubly-linked list, a hash table and the so-called 'dictionary', which is actually a combination of the two. Since our primary inspiration was Lisp, and our proficiency as programmers was very limited at the time, we first implemented our score using Max's doubly-linked list, which very soon proved being extremely inefficient and cumbersome to use in the context of *bach*. On the other hand, the die was cast and, when we decided to replace the original data structure with a more streamlined one, we implemented our own version of a doubly-linked list, and devised a mechanism for passing such lists, which we called *llll*s, from object to object, through reference to an internally maintained table.

When Max 6 was released, with a new set of objects exposing dictionaries to the end user, we questioned our choice of using a custom data type, with all the intricacies that came with it, rather than the one provided by Max, providing a better integration with the host environment and a richer data representation at the cost of being less efficient and apt to the specific task. It must be stressed out that the basic representation of a musical score we chose (which we still think is a quite natural, if somewhat simplistic, one) is in terms of a hierarchically structured sequence of items, rather than an ordered collection of key-based values. Moreover, Max's dictionary only accepts symbols as keys, making it impossible, for example, to treat measure or voice numbers as such. Eventually, we still think that

the *llll* is overall a better choice than the dictionary, but we pay the price for this choice everytime we need to devote a lesson in our courses about the attributes and objects necessary to pass back to Max data produced or processed by *bach*.

3.2 *llll*s or databases?

*llll*s are designed to contain arbitrarily large collections of hierarchically organized data, but they offer relatively slow access to them. Both positional (i.e., looking for a piece of data according to its position) and semantic (i.e., looking for a piece of data according to its qualities) access happen in linear time, which means that all the elements preceding the one we are looking for must be traversed. Max lists allow constant-time positional access (i.e., accessing the 10.000th element requires the same, short amount of time needed for accessing the first), but, in turns, they have a relatively low limit in size. Semantic access with specialized data structures, such as hash tables, can be performed in linear time, but provides no way for structuring information hierarchically. In order to access large amounts of data in sublinear time, some more advanced data structure was needed.

We tackled this issue in the *dada* library, most notably with the `dada.base` object, implementing a SQLite database, organized in tables, with a certain number of fields (columns) and a certain number of entries (rows). Each column can store numbers, symbols, or even *llll*s (although with the important limitation that queries cannot be performed on them). Being based on a true database structure, queries on numbers and symbols are performed much quicker than the corresponding searches inside *llll*s. On the other hand, an SQL database is a way more complex structure, requiring more infrastructure and more computational time for simple tasks and small data collections.

We are currently considering the possibility of equipping other portions of *bach* with database-like capabilities.

3.3 How many numbers are there?

We painfully miss Lisp's arbitrary precision integer arithmetic. We did not rely on C libraries for arbitrary precision arithmetic when we started, and we built an interface for rational numbers using standard 32- and 64-bit arithmetics. This means, however, that even the simple product of the first 16 prime numbers has overflow issues. It is fairly easy in *bach* to sum random rational numbers and quickly hit the limits of the representation. Since all the duration management system of `bach.score` is based on rational numbers (including the rhythm parsing and the computation of beaming), we had to prevent the worst case scenarios by introducing a maximum denominator, albeit a very large one. This was a difficult decision, as there were symptoms of a vicious circle: we based our arithmetics of durations in `bach.score` on rational numbers in order to be as precise as possible, while on the other hand we could not afford it and we had to put a hard-coded approximation limit.

Composers working with Lisp-based environments are used to inputting sequences of the most outlandish kinds of fractions and get exotic but yet consistent results. The same is not true for our library, due to the limits of the native C integer arithmetics. In the future, we may consider to equip *bach* with arbitrary precision arithmetics, at the expense of computational speed and memory usage (Fig. 3). Another, less radical option could be implementing internally rational numbers as high-precision floats (such as 80- or 128-bit), and have some conversion routines to be called whenever the actual rational representation is needed. This would have the major advantage that numbers that cannot be represented exactly would undergo an actual approximation, rather than being completely misrepresented as it happens with the current implementation; moreover, all the mathematical operations would be much faster and their implementations would be trivial. Such a representation, however, would not guarantee that sets of rational numbers having equal sum would continue having equal sum — which is a crucial detail when dealing with rhythmic synchronicities in a score. Moreover, such a solution would also have a much greater computational cost whenever a rational number must be retrieved as such, because of the expensive conversion needed, and the potential loss of precision in cases where the current representation is inherently exact.

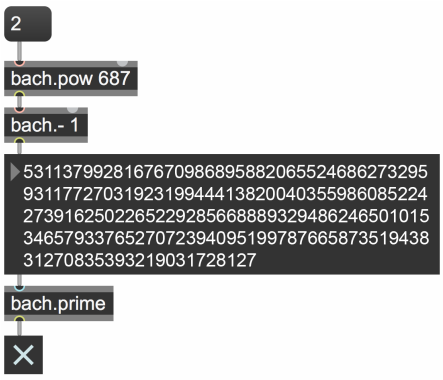

Figure 3. Imaginary behavior of arbitrary precision arithmetics to check a Mersenne number for primality.

3.4 What is a pitch?

At the beginning of *bach*, we borrowed from OpenMusic the idea that musical pitch is measured in MIDIcents (or cents, for short). This was a bit of a peculiar choice in Max, whose pitch-based tools all worked with MIDI numbers [2]. But, all in all, the cents were a rather widespread unit in the field, and they guaranteed an integer representation for quartertonal and eighth-tonal systems. However, using cents as a unit resulted in flattening all the enharmonic information of a pitch onto a real-valued parameter, which was very limiting in any context where diatonicism, to some degree, mattered. Prior to *bach* 0.8, we dealt with enharmonicities by assigning a 'graphics' property for each note, defining the cents of the displayed note (without accidentals) as a float and the displayed accidental as a rational number, representing a fraction of a whole step. This was, at best, cumbersome. In *bach* 0.8, we switched

[2] This was the case ten years ago; now some objects, such as the pitch shifting tools, also accept MIDIcents as input.

to a new system, by introducing a novel data type: the pitch [6]. Crucially, pitches are intervals and intervals are pitches. Pitches, just like numbers, can be added or subtracted (in fact, that is how transposition is achieved), or can be multiplied by a scalar value (that is how intervals are replicated). Pitches can be parsed from standard readable representations (such as `Db4` or `C#6`) or from more precise specifications (such as `C#3+2/27t`). The middle C in *bach* is `C5` — a decision we do not regret, since it is by far the most convenient choice, as explained more in detail in [6].

3.5 Is this a symbol?

As we have seen, the *bach* library introduces some new data types, including rational numbers, pitches and *llll*s. This also means, in turns, that symbols such as `A1`, `0/1`, `Help(1965).mp3` are understood by *bach* objects as being, respectively: a very low A note; the number 0; and a list containing the symbol *Help*, the number *1965* (inside a level of parentheses) and the symbol *.mp3*. Of course, we needed an escaping mechanism to force the parser to interpret incoming symbols in a literal way. Using the double quotes was not an option because of Max's own parsing mechanism [3] .

To force the parsing of a symbol in a literal way, we used the backtick sign (`) as escaping symbol. This was a convenient choice at the beginning: the backtick is a relatively rarely used symbol of the ASCII chart, so one could write `` `A1 ``, `` `0/1 `` or `` `Help(1965).mp3 ``. Eventually, we also developed two utility objects named `bach.textin` and `bach.textout`: among other things, the former takes care of interpreting some types of input literally without any need of backticking, while the latter takes care of managing the output of backticks in Max format.

However, an important issue with backticking remains, due to the fact that the escaping has no ending marker, and the symbol to be interpreted literally ends as soon as a whitespace is reached. The idea was that if one needed to escape a symbol with spaces, the ordinary double quotes would have worked perfectly in Max. Although this is true, it became a problem when `bach.eval` was introduced, and with it, the ability to retrieve sublists of a given list by key (in a sort of dictionary-like fashion): if the variable `MyList` contained `[ 'pan [ "Bass Flute" -0.5 ] [ 'Clarinet 0.3 ] ]`, in order to retrieve the values one should have had to add spaces: `MyList. 'pan . 'Clarinet`, `MyList. 'pan . "Bass Flute"`. This quickly became too inelegant to be acceptable especially when nested sublists had to be accessed, as spaces had to be added almost everywhere, often in non-intuitive places. In retrospect, we would not have had any problem had we used both a starting and an ending backtick for the task: `` `A1` ``, `` `0/1` ``, `` `Help(1965).mp3` ``. As of now, we do not want to break backward compatibility, so we decided to live with this quirk; nonetheless, we equipped `bach.eval` with another escaping mechanism, the pair of single quotes: `bach.eval $l1.'pan'.'Bass Flute'`. Once again, in order to avoid breaking existing patches, we do not plan to extend this behavior to the whole *bach* system.

[3] Max uses double quotes only to produce symbols containing spaces and automatically deletes them in any other case: typing *"A1"* in a message box is automatically rewritten simply as *A1*

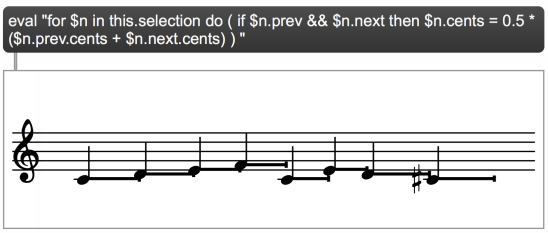

Figure 4. Imaginary behavior of a *bell* language scripting support within `bach.roll`.

4. DATA FLOW

4.1 Patching or scripting?

There are several ways to modify a score: one of the simplest involves dumping its parameters from some outlets, modifying them via appropriate modules, and feeding the result into a different editor. This is the traditional OpenMusic pipeline, which we borrowed while designing *bach*, and even more so while creating the *cage* library. Alternatively, one can send direct messages to an editor, asking for specific elements of the score to be created or modified, with no output from the object outlets (unless explicitly requested). For example, doubling the duration of one or more notes can be done by entering the message `duration = duration * 2` after selecting the desired notes — which can be achieved through specific messages as well. All in all, these 'in-place' modifications have much more to do with scripting than with visual programming.

The *bach* editors are flexible enough to allow both kinds of interaction, to a large extent. However, for working with *llll*s, a single type of interaction was possible until recently: visual data-flow programming, i.e., combining atomic modules to produce a complex algorithm. As a result of Max's very peculiar take at data-flow, this typically resulted in intricate patches with spaghetti-like connections, very hard to analyze and maintain, whereas a few lines of textual code may have achieved the same result with greater readability.

This consideration was at the root of the development of the *bell* language, a scripting language explicitly designed to handle these situations, fully implemented in the `bach.eval` object but also exposed in a number of other modules as a way to define the details of their behavior. [7].

In the future, it would be important to expand the *bell* language to also include an object system, so as to make the *bach* editors also directly scriptable in *bell*, hence allowing more complex interactions than the ones possible at the current stage (Fig. 4).

5. MODULE DESIGN

5.1 How is it called?

One of the most annoying kinds of mistake we have made has been choosing inconvenient names for some of our modules. The worst example is probably `bach.append`, a module designed to join two *llll*s, which is essentially what *zl.join* in Max does for standard Max lists, and which is sig-

nificantly different from how the *append* object behaves. This became especially irksome when we introduced the long needed `bach.prepend` module, adding an element at the front of the incoming existing list. One would have wished for a `bach.append` module mirroring such behavior on the other end of the list, but that could not be the case.

Eventually we deprecated the `bach.append` module and created the more appropriate `bach.join` instead; nevertheless, for backward compatibility, we could not change the behavior of `bach.append` to the desired one. We decided to create a `bach.postpend` module instead: its naming is admittedly ugly, but at least its behavior is proper. A number of other modules have undergone similar fate, with lesser consequences.

5.2 How does it work?

Most of the modules in the *bach* library are designed to deal with *llll*s in a wide variety of ways; they differ, however, on how they treat lists with more than one level of depth. For instance, by default `bach.rev` performs the reversal of the list on *all* levels of depth, whereas `bach.rot` only rotates the elements at root level. One can modify this behavior by using the *maxdepth* and *mindepth* attributes; however, the default behavior is uneven across all the family of tools, which makes them rather unpredictable at a first glance.

At times, it also happened that we chose as the default configuration one that was relatively rare to come across; possibly one of the most annoying examples is the fact that `bach.iter` and `bach.mapelem` both work, by default, across *all* levels of depths, while in most cases one only needs them to operate at root level (hence they are often employed with *maxdepth* attribute set to 1).

In the *bell* programming language we tried to make amends for this, and all the list-processing functions and operators only work on the root level by default. This makes *bell* more consistent and predictable, but of course adds another contradictory layer of information in that, for example, the `rev` function behaves differently from the `bach.rev` object.

Anyway, in a future version of *bach* we may add two objects performing the same operations by default only at root level. By borrowing the naming of the functions of `bach.eval`, we may call them `bach.for` and `bach.map`.

Incidentally, in `bach.rot` positive numbers rotate left, which mimics what happens in OpenMusic's *rotate* function — opposite to what *zl.rot* does. This is yet another addendum to the list of the things that should *not* have been borrowed from OpenMusic.

Additionally, some of the modules accept their parameter in their right inlet (e.g. `bach.rot`), whereas for others it is an attribute (e.g. `bach.slice`). This is however a lesser issue, since a future harmonization is still possible without breaking any backward compatibility.

5.3 Where does it belong?

The development of the *bach* library has been far from linear; most importantly, at the beginning we had no idea that we would eventually author *cage* and *dada* as well, where some of the modules would have been a better fit. In the most evident cases of misplacement, we deprecated the old modules and moved them to the more appropriate library (as is the case for `cage.combinevoices`, formerly `bach.combinevoices`). In other cases, the boundary is rather blurred.

Furthermore, in *cage* we decided to organize a certain number of modules into sublibraries (e.g. `cage.profile.*`, handling melodic profiles, or `cage.sdif.*`, handling SDIF import/export). The result is relatively clean, although verbose: if one needs to display a SDIF file containing partial tracking information inside a `bach.roll`, it is quite reasonable to invoke the `cage.sdif.ptrack.toroll` module. We did not make the same choice in *bach*, although perhaps we should have: for instance, all the objects dealing with matrices could have been named `bach.mat.*`.

5.4 Objects or abstractions?

The *bach* library is composed by both objects and abstractions: objects are compiled objects written in C or C++, whereas abstractions are Max subpatchers, containing other objects and/or abstractions. Much of our effort has been targeted at reducing the gap between the two kinds of modules: abstractions come with their own set of help files and documentation, just like objects do; their nesting is made consistent at initialization thanks to the `bach.args` utility object; their outlet policy works exactly as for any other object in *bach* thanks to the `bach.portal` object [4, section 3.1]. However, there are still differences between the two kinds of modules, in both performance (abstractions are slower) and documentation (Max seems to handle them more difficultly with respect to autocompletion, documentation cross-referencing, etc.). Another issue we have been consistently facing with abstractions is the impossibility (unless cumbersome scripting mechanisms are put in place, with their own set of drawbacks) of adapting the number of inlets and outlets to the arguments of the object box. This is why, for instance, `cage.join` works differently from `bach.join`.

When we started the *bach* library, we decided to build some modules as abstractions because we thought that they could have been easier to maintain and debug; that was true only to a certain extent: some abstractions became so complex that it was easier to rewrite them from scratch as objects (`bach.transcribe` being a prominent example). This could almost always be done without breaking any compatibility, so whenever we felt that some abstraction was better suited to be an object, we simply coded it (there are complex abstractions still waiting to undergo this process, such as `bach.counter`).

An exception to this rule is the *cage* library, where converting abstractions to objects was not an option: one of the main ideas behind *cage* was to develop all modules as patchers, in order to allow users to analyse and adapt them [8]. In practice, however, this pedagogical idea resulted in many modules being rather complex to debug and reverse-engineer (such is the case, for example, for `cage.fm`), quite the contrary of what we would have liked. As explained in section 4.1, `bach.eval` is meant to help

mitigating this issue.

6. CODING POLICIES

6.1 C or C++?

Writing the *bach* code is how we learned to code better — which means that, unfortunately for whoever might decide to read it, the older a portion of code is, the more confused it is likely to be, with comments and naming conventions being rather volatile and actual bad coding practices all around. Given the scope of the work (the line count on the *bach* sources is currently well above 300k), this makes the code sometimes difficult to navigate, especially for the most complex portions of the library, such as the modules dealing with the treatment of musical notation.

In addition, we started to code following and expanding the Max C API, with as little C++ code as possible. Simply put, that was the way we were used to work, and it seemed as clean and as elegant as it could get. In retrospect, having relied on C++ more heavily, and especially on syntactical features such as operator overloading and template metaprogramming, would have produced much simpler and more mantainable code. Some portions of the library, such as the rational number interface, have already been ported to C++ for convenience; others have not, although they would definitely benefit from it, like the *llll* interface. On the other hand, making a C++ class at least for ubiquitous structures such as the *llll*, with all the nice cleanup that could come with it, would require a major rewrite of a large part of the codebase, so this is very unlikely to happen in the foreseeable future.

6.2 Dedicated structures or *llll*s?

Although in the *bach* ecosystem modules communicate data in the form of *llll*s, complex objects, such as `bach.roll` or `bach.score` have their own internal, dedicated representation of the data they contain in the form of C structures tailored for the task. This produces a small overhead (*llll*s must be parsed and deparsed), but it simplifies greatly any subsequent operation on data, in terms of both clarity and of CPU time.

There are however some other complex modules in which we chose to rely solely on *llll*s to carry the internal state, and we reckon that this was a poor decision. Possibly the worst module, in this respect, is `bach.quantize`, which is currently quite hard to maintain and debug due to the lack of dedicated structures for the task.

7. CONCLUSIONS

Since we usually learn more from failure than from success, we believe it was important to detail in this paper a list of obstacles we have been facing in *bach* due to some poor decision taken in the past. We have presented in this paper our choices from a developer's point of view. This is a personal feedback on the project: performing a general survey of what the *bach* user base considers the major problems of the system would definitely be an interesting subject for further research, and would lead to a completely different paper, much more concerned with the outcomes of design choices, rather than their motivations.

Every complex project is bound to have some sort of fundamental issues; this article is about our mistakes in creating *bach* as much as our effort to overcome them. Sometimes, we were able to solve the issues in what we still think are elegant and novel ways (such has been the case with pitches); sometimes, the need to preserve backward compatibility directed us towards more inelegant, although still effective, solutions (such was the case with escaping); other times we have not solved the issue altogether (such was the case with infinite precision arithmetics).

All this being said, the fact that, with all its flaws, *bach* has been adopted by thousands of persons makes us think that a limited tool is better than no tool anyway, and encourages us to improve what can be improved without pursuing the impossible feat of perfection. If everything goes well, we shall spend the next ten years trying again, failing again and, hopefully, failing better, and our *cahier de doléances* will be much longer in 2030.

8. REFERENCES

[1] P. Manoury, "Sur quelques thèmes de recherches musicales actuelles," http://www.philippemanoury.com/?p=4786, 2011.

[2] A. Agostini and D. Ghisi, "Real-time computer-aided composition with *bach*," *Contemporary Music Review*, no. 32 (1), pp. 41–48, 2013.

[3] ——, "A Max library for musical notation and computer-aided composition," *Computer Music Journal*, vol. 39, no. 2, pp. 11–27, 2015.

[4] D. Ghisi and A. Agostini, "Extending bach: A Family of Libraries for Real-time Computer-assisted Composition in Max," *Journal of New Music Research*, vol. 46, no. 1, pp. 34–53, 2017. [Online]. Available: http://dx.doi.org/10.1080/09298215.2016.1236823

[5] M. Puckette, "Max at seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[6] A. Agostini and D. Ghisi, "Pitches in bach," in *Proceedings of the International Conference on Technologies for Music Notation and Representation–TENOR*, Montréal, Canada, 2018, pp. 128–137.

[7] A. Agostini, D. Ghisi, and J.-L. Giavitto, "Programming in style with bach," in *14th International Symposium on Computer Music Multidisciplinary Research*, Marseille, France, 2019, p. 91.

[8] A. Agostini, E. Daubresse, and D. Ghisi, "*cage*: a High-Level Library for Real-Time Computer-Aided Composition," in *Proceedings of the International Computer Music Conference*, Athens, Greece, 2014.