Programming in style with *bach*

Andrea Agostini[1], Daniele Ghisi[2], and Jean-Louis Giavitto[3]

[1] Conservatory of Turin
[2] Conservatory of Genoa
[3] CNRS, STMS – IRCAM, Sorbonne University

Abstract. Different programming systems for computer music are based upon seemingly similar, but profoundly different, programming paradigms. In this paper, we shall discuss some of them, with particular reference to computer-aided composition systems and Max. We shall subsequently show how the *bach* library can support different programming styles within Max, improving the expression, the readability and the maintainance of complex algorithms. In particular, the forthcoming version of *bach* introduces *bell*, a small textual programming language embedded in Max and specifically designed to facilite programming tasks related to manipulation of symbolic musical material.

Keywords: Programming paradigms, computer-aided composition, Max, *bach*, *bell*

1 Introduction

In spite of the way it is advertised, its own Turing-completeness and the sheer amount and complexity of things that have been done with it, programming in Max is difficult. Whereas setting up simple interactive processes with rich graphical interfaces may be immediate, it has been long observed that implementing nontrivial algorithms is far from straightforward, and the resulting programs are often very difficult to analyse, maintain and debug.

Several other popular programming languages and environments for computer music, such as OpenMusic [1], PWGL [10] and Faust [11], share with Max a superficially similar, but profoundly different, dataflow programming paradigm, which makes them better suited for 'real' programming and less for setting up highly interactive and responsive systems. This is reflected in the types of artistic practices these systems are typically used for, and mirrors the oft-discussed rift between composition- and performance-oriented tools in computer music.

We are convinced that this rift is by no means necessary or natural, and, on the contrary, has proven problematic with respect to a wide array of practices lying somehow between the two categories, such as extemporaneous, 'intuitionistic' approaches to composition (including, but not limited to, improvisation), sound-based and multimedia installations, live coding and more.

In this paper, we shall investigate this divide and its reasons from the point of view of computational models, and consider how it can be bridged, or at least narrowed, through the use of the *bach* package for Max [3].

2 Dataflow computational models

The concept of data flow is an old one, dating back at least to [5], which first introduced the idea of independent computational modules communicating by sending data (discrete items) among directed links. Over the years, many kinds of dataflow computation models have been developed. In this section, we shall review some of them and how they apply to different languages and software systems for computer music.

2.1 The pipelined and functional dataflow and the Kahn principle.

Several computer music languages and systems (such as Reaktor, OpenMusic and Faust, but also Reaper, Live, ProTools and various software synthesisers) are based upon the pipelined dataflow model. This means that programs written in those systems have the following features, strictly linked to one another:

- Programs are represented as directed graphs; each node of the graph implements an abstract process consuming data on its input links and producing data on its output links; the links are the only interactions between the processes; the sequence of data traversing a link is called a *stream*.
- Processes have *referential transparency*: we can always replace every variable and function by its definition, and a function called twice on the same data will always return the same result.
- The resulting programming style is *declarative*: programmers only specify the properties of the objects they want to build, rather than the way to build them.
- Programs are mathematical objects, and can be treated as such. It has been proven by Gilles Kahn [9] that a pipelined dataflow program is equivalent to a set of equations, taking the form of a fixed point equation. What is known now as the Kahn Principle states that the stream associated to each edge of the dataflow graph is the solution of the previous set of equations. As a consequence, algebraic reasoning on the operational properties of programs is possible and useful.
- Just like in the process of solving an equation, there is no notion of temporal sequencing of actions, but rather of algebraic relations between parts of the equation. Therefore, the program graph is acyclic (as feedback loops are only meaningful if they establish a temporal delay), and an input link can only accept one single incoming graph edge.

All this being considered, whether to write a functional dataflow program as a graphical patch or as a set of equations specified textually is a matter of taste. Faust is an example of a textual, functional dataflow programming language in which a program is a set of equations.

Although programs are not explicitly expressed as equations, OpenMusic and PWGL are essentially based upon the same assumptions as Faust.[4] How-

[4] It should be remarked that the computational model of OpenMusic and PWGL is not as pure as described here, since it includes imperative traits like storing and retrieving mutable states through variables.

ever, evaluation in Faust is driven by the availability of the data, that is, it happens whenever data enter the nodes. On the contrary, the evaluation process of OpenMusic and PWGL is demand-driven, that is, the user requests a result to the bottom node of the graph, which in turns requests values to the nodes connected to its input links, and so on.

2.2 Asynchronous, non-functional pipelined dataflow: Max

Max implements two different dataflow systems, respectively devoted to audio signals and control messages.[5] The former is a relatively simple case of synchronous pipelined dataflow, whose functional nature is somewhat less explicit than that of Faust but not too different from it: in fact, the functional dataflow view fits very well with the audio graph representation of signal processing. Our discussion will only focus on the latter and its significantly different paradigm. In what follows, we shall assume in the reader a basic, practical knowledge of Max, and only review some fundamental concepts when needed.

A Max patch can be seen as a set of nodes working asynchronously with respect to each other: if, when and how each module 'fires' depends on the data processed, and, generally speaking, only one message can traverse the patch at any given time. This means that nodes with more than one input link must have mechanisms for storing data for later use. This is accomplished through the so-called 'hot' and 'cold' inlets (that is, input links in the Max jargon): when a hot inlet receives a message, it performs its computation and delivers the result; but when a message is received in a cold one, it gets stored for later use and nothing else happens. Most Max objects have at least one hot inlet, and many have one or more cold inlets.

This structure, which actually involves many other details and is not without exceptions, has some profound consequences:

- Objects have mutable states, which may change over time in response to a single evaluation request (see, e.g., the 'cold' inlet of any arithmetical operator).
- Objects have no referential transparency. The order of messages on a link is not enough to determine the global behavior of a patch: the precise timestamp of these messages is semantically meaningful. Altering the order in which data are sent from a node to others in response to a single piece of incoming data (that is, to a single computation request) may change the performed computation.
- Multiple links ('cords' in the Max jargon) can be connected to a single inlet: as data are always transmitted sequentially, this means that the inlet will receive data from its incoming cords one after another, and act consequently.

[5] Most of the general principles described here also apply to Max's sibling system, Pd, which we shall not discuss as the *bach* library is currently not available for it.

2.3 Pros and cons of different computational models

Max's computational model is motivated by the fact that, unlike the other systems described above, it was not conceived as a programming language but, in its own creator's words [12], as a *musical instrument*. With respect to this end, Max has the merit of being extremely economical in terms of its basic principles and quite adaptable to very different use cases.

On the other hand, as hinted at above, representing nontrivial algorithms in Max is often more complicated than with other systems. Two of the authors became painfully aware of this complicatedness while working at the *cage* package [2], which implements a comprehensive set of typical computer-aided composition operations. *cage* is entirely composed of abstractions, and during its development the shortcomings of Max programming became so evident that the seeds for the work presented in this article were planted.

The reasons for this difficulty are multiple, and include the following:

– The greater freedom Max grants in building the program graph easily leads to far more intricate patches than functional dataflow models, with spaghetti connections that can grow very hard to analyse.
– Typical Max patches often have their state distributed through many objects whose main, individual purpose is not data storage.
– Max lacks, or implements in quite idiosyncratic ways, some concepts that are ubiquitous in modern programming languages, such as complex, hierarchical data structures, data encapsulation, functions and parametrization of a process through other processes.

On the other hand, Max allows to incorporate, on top of its basic paradigm, traits reminiscent of various programming styles, such as imperative, object-oriented and functional. Moreover, it includes various objects enclosing entire language interpreters, thus allowing textual code in various languages to be embedded in a patch.

These features may prove useful when nontrivial processes have to be implemented, as is the case when working in contexts like algorithmic and computer-aided composition. Whereas Max was not conceived with these specific applications in mind, it quickly became clear that it could be a valuable environment for them, and several projects have been developed in this sense [14,13,7]. We shall focus on one of them, the *bach* package, which has been conceived and maintained by two of the authors.

2.4 The *bach* package

The *bach* package[6] for Max is a freely available library of more than 200 modules aimed at augmenting Max with advanced capabilities of symbolic musical representation. At its forefront are two objects called `bach.roll` and `bach.score`, capable of displaying, editing and playing back musical scores composed of

[6] `www.bachproject.net`

both traditional notation and arbitrary time-based data, such as parameters for sound synthesis and processing, textual or graphical performance instructions, file paths and more.[7]

One of the main focuses of *bach* is algorithmic generation and manipulation of such scores. To this end, *bach* implements in Max a tree data structure called *llll* (an acronym for Lisp-like linked list), meant to represent arbitrary data including whole augmented scores. *bach* objects and abstractions exchange *lllls* with each other, rather than regular Max messages, and their majority is devoted to performing typical list operations such as reversal, rotation, search, transposition, sorting and so on.

Generally speaking, *bach* objects abide by the overall design principles and conventions of Max, but it should be remarked that, whereas standard Max objects can control the flow of *lllls* in a patcher just like they do with regular Max messages, they cannot access their contents unless *lllls* are explicitly converted into a Max-readable format, which on the other hand has other limitations (for a detailed explanation, see [3]). Thus, *bach* contains a large number of objects that somehow extend to *lllls* the functionalities of standard Max objects. For example, whereas the `zl.rev` object reverses a plain Max list, the `bach.rev` object reverses an *llll* by taking into account all the branches of the tree, each of which can be reversed as well or not according to specific settings. Whereas it is possible to convert an *llll* into Max format and reverse it with `zl.rev`, in general the result will not be semantically and syntactically correct.

Since its beginnings, *bach* has been strongly influenced by and related to a number of other existing projects: for an overview of at least some of them, see [3]. The synthesis of different approaches that lies at the very basis of the conception itself of *bach* has been validated by a large community of users, who have developed many artistic and research projects in several domains[8], as well as the fact that it provides the foundation for the *cage* and *dada*[9] libraries [8].

In the following sections, we shall review a few programming styles and approaches and see how *bach* can be helpful with adopting them in Max: namely, we shall show how some fundamentally imperative, functional and objected-oriented traits of Max can be leveraged through the use of specific *bach* objects and design patterns; moreover, we shall discuss a recent addition to *bach*, that is, a multi-paradigm programming language called *bell* and meant to facilitate the expression of complex algorithms for manipulating *lllls*.

[7] `bach.roll` and `bach.score` differ in that the former represents time proportionally, whereas the latter implements a traditional representation of time, with tempi, metri, measures and relative temporal units such as quarter notes, tuplets and so on.

[8] The website of *bach* showcases some interesting works that have been developed with the library, mostly by people independent of its developers.

[9] The *dada* library contains interactive two-dimensional interfaces for real-time symbolic generation and dataset exploration, embracing a graphic, ludic, explorative approach to music composition.

3 Different programming styles and approaches in Max

3.1 Imperative approach

It has been observed that Max is essentially an imperative system in disguise [6]: as stated before, any nontrivial program in Max requires to take care of states and the order of operations, and analysing even a moderately complex patch can only be done by following the flow of data and the evolution of states over time.

It is possible to make this imperative style more explicit by adopting some good practices, such as widely using specific objects, such as `trigger` and `bangbang`, that can help with keeping the evaluation order under control. Moreover, Max contains two objects whose only purpose is holding data associated with a name: `value` and `pv` (for 'private value'), whose role can be seen as corresponding to that of variables in traditional imperative programming languages. Each instance of those objects has a name, and every time it receives a piece of information it retains and shares it with all the other objects with the same name. It is subsequently possible retrieve the stored data from any of them. The `value` and `pv` modules differ in their scope: the former's is global, that is, data are shared through all the open patches in the Max session, whereas the latter's is local, in that data are only shared within the same patcher or its subpatchers. By combining `value` and `pv` with the aforementioned sequencing objects, it is possible to use Max in a much more readable, essentially imperative programming style.

bach implements its own variants of these objects, respectively named `bach.value` and `bach.pv`. Besides dealing correctly with *lllls*, they can open a text editing window if double-clicked, allowing to view and modify the data they hold. Moreover, *bach* contains an object called `bach.shelf`, which acts as a container of an arbitrarily large set of *lllls*, each associated to a unique name. `bach.shelf` objects can be themselves named, thus defining namespaces: this means that *lllls* associated to a name within one named `bach.shelf` object will be shared only with other `bach.shelf` objects with the same name. Although still somewhat crude (it might be interesting, for example, allowing non-global namespaces), this is a way to improve data localization and data encapsulation, and reduce the proliferation of storage objects in complex scenarios.

3.2 Object-oriented approach

The fact that a Max program is built of independent blocks responding to messages they send to each other in consequence of callbacks triggered by events gives it a strong object-oriented flavour, and the Smalltalk influence is both apparent and declared. At a lower level, in fact, each Max object in a patch is an instance of a specific class, with member variables containing the object's state and methods roughly corresponding to the messages it accepts for modifying and/or querying the state.

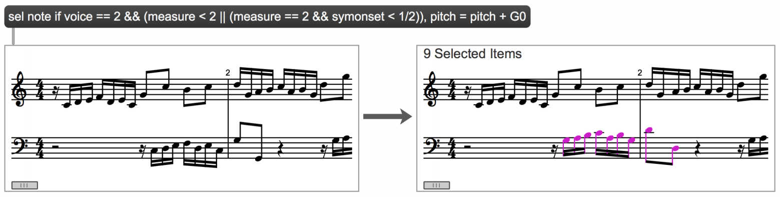
The two main *bach* editors, `bach.roll` and `bach.score`, comply with this object-oriented approach. However, a distinction can be made about the kinds of messages they accepts: some control and query the object's appearance (background color, zoom level, etc.), whereas others are dedicated to the direct management of the editor's content. This distinction between the two kinds of messages is explicit in the syntax of the messages they receive.

Messages dealing with the editor's contents enable the creation, the edition and the deletion of individual notation items, such as a single measure or a single note. These messages can actually be seen as methods of the items themselves, which are arranged according to a precise hierarchy and share a certain number of common properties (such as having a symbolic name, being selectable, etc.).

In fact, there are several ways to modify a score. One of the simplest involves dumping its parameters from some outlets, modifying them via appropriate Max and *bach* modules, and feeding the result into a different editor object.

In contrast, one can send direct messages to the editor, asking for specific elements of the score to be created or modified through the so-called *bach in-place syntax*, with no output from the object outlets (unless explicitly requested). Modifications are immediately performed and the score is updated (see Fig. 1). This mechanism is strongly inspired by an object-oriented approach: first, references to the notation items to be modified are acquired via a selection mechanism, and then messages are sent to them. For example, a set of notes can be selected graphically, or through a query in the form of a message such as `sel note if voice == 2 and pitch % C1 == F#0`. After this, those notes can be modified by means of messages such as `duration = velocity * 10`.

In fact, this kind of approach allows much more complex operations than the ones described here, as there are many classes of notation items, each having a large number of properties and related messages. In spite of the richness of the data it can manipulate, though, the in-place syntax is not very flexible, but there are plans to extend it through the *bell* language (see below).



```
sel note if voice == 2 && (measure < 2 || (measure == 2 && symonset < 1/2)), pitch = pitch + G0
```

Fig. 1. A very simple example of in-place modification: notes belonging to the second voice and whose onset lies before the middle of the second measure are selected and transposed up a perfect fifth (the image shows both the state of the score before and after the click on the message).

3.3 Functional approach

Max shares some similarities with functional languages, mostly by handling values through a variety of nodes implementing functions on these values. It is then possible to build patches that somehow behave functionally, and whose appearance is extremely similar to that of equivalent ones in a functional graphical system such as PWGL. *bach* extends the functional traits of Max in a few areas.

As hinted at before, it implements the *llll*, a tree data type quite similar to a Lisp list, and provides a large number of modules for dealing with *lllls*. Although, of course, list operators are not inherently functional, they are quite customary in functional languages, and the corresponding *bach* objects can be connected in a way corresponding to the composition of list functions in functional languages such as Lisp or Haskell.

Secondly, generalized versions of functions such as *sort* and *find* require some way to specify, respectively, a custom ordering or an arbitrary search criterion. In several languages, these generalized functions are conveniently implemented as higher-order functions, i.e., functions taking other functions as arguments. This requires to handle functions like ordinary data. A Max patcher lacks the concept of function, but several *bach* objects implement a design pattern called the *lambda loop* (see Fig. 2), whose role is somehow akin to that of higher-order functions.

A lambda loop is a patching configuration in which one or more dedicated outlets of a module output data iteratively to a patch section, which must calculate a result (either a modification of the original data, or some sort of return value) and return it to a dedicated inlet of the starting object [3].

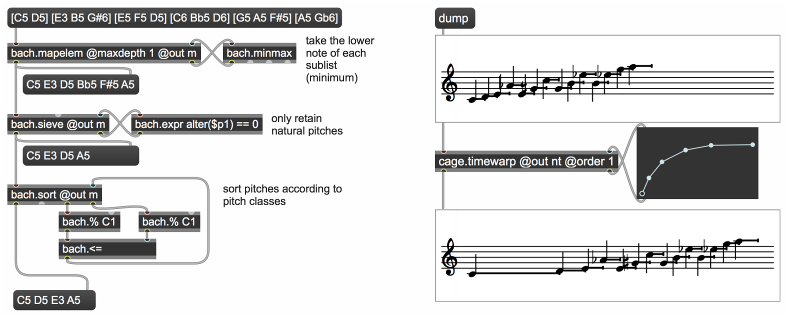

[C5 D5] [E3 B5 G#6] [E5 F5 D5] [C6 Bb5 D6] [G5 A5 F#5] [A5 Gb6]
bach.mapelem @maxdepth 1 @out m
bach.minmax
take the lower note of each sublist (minimum)
C5 E3 D5 Bb5 F#5 A5
bach.sieve @out m
bach.expr alter($p1) == 0
only retain natural pitches
C5 E3 D5 A5
bach.sort @out m
sort pitches according to pitch classes
bach.% C1
bach.% C1
bach.<=
C5 D5 E3 A5
dump
cage.timewarp @out nt @order 1

Fig. 2. The cross-connected and loop-connected patch cords attached to *bach.mapelem*, *bach.sieve*, *bach.sort* and *cage.timewarp* modules form several instances of the so-called lambda loop. The left-side example should be straightforward. In the right-side example, the temporal distribution of events in a musical score is altered through the provided transfer function, with time on the X axis and speed on the Y axis. At a superficial level, patches like these appear to be quite similar to how the same processes might be implemented in a functional dataflow system.

Lambda loops are used by some *bach* modules directly inspired by functional programming practices, such as `bach.mapelem` (performing a map operation) and `bach.reduce` (recursively applying a binary function on elements); all these modules can be helpful to translate programs conceived functionally into Max patches. The number of modules taking advantage of this design pattern is, however, much larger, and include basic operators such as `bach.sieve` (only letting some elements through) and `bach.sort` (performing sort operations), but also advanced tools such as `bach.constraints` (solving constraint satisfaction problems) as well as some of the modules in the *cage* package.

4 Textual coding

The approaches described so far are based on the idea that individual objects carry out elementary operations, and they are connected graphically so as to build complex behaviors.

A different, but not incompatible, point of view is embedding an algorithm, even a potentially complex one, into a single object by means of textual coding, and subsequently insert it into a patch. In graphical, Lisp-based systems such as OpenMusic and PWGL, this is easily accomplished by inserting graph boxes containing Lisp code in the patcher. Whereas it is possible to embed in Max textual code written into various programming languages including C, Lua, Java and JavaScript, we feel that none of those language bindings provides the ease and directness of embedding of a Lisp code box in OM or PWGL.

On the other hand, Max contains a family of objects, namely `expr`, `vexpr` and `if`, that allow defining textually mathematical expressions and simple conditionals which might otherwise require fairly complicated constellations of objects in a patch. *bach* adds another member to the family, called `bach.expr`, allowing to define mathematical expressions to be performed point-wise on *lllls*.

Whereas the `expr` family syntax is not a full-fledged programming language, it can be seen as the basis for one. We therefore decided to include in the latest release of *bach* a new object to the family, called `bach.eval`, implementing a new, simple programming language conceived with a few, conceptually simple points in mind:

– Turing-complete, functional syntax, in which all the language constructs return values, but also including imperative traits such as sequences, variables and loops.
– Full downward compatibility with the `expr` family.
– Inclusion of list operators on *lllls* respecting, as far as possible, the conventions and naming of the corresponding *bach* objects.
– Implicit concatenation of elements into *lllls*, meaning that by simply juxtaposing values (be they literals, or the result of calculations) they are packed together into an *llll*. In this way, a program can be seen as an *llll* intermingled with calculations, not unlike what happens by combining the `quote` operator and `unquote` macro in Lisp.

– Maximum ease of embedding of the object into a Max patcher, with, among the other things, no need for explicit management of inlets and outlets.

The resulting language is called *bell* (standing for *bach evaluation language for lllls*, but also paying homage to the historic Bell Labs). A detailed description of its syntax can be found in [4], whereas, for the scope of this article, a few examples should suffice (see Fig. 3, 4 and 5).

bell code can be typed in the `bach.eval` object box or into a dedicated text editor window, loaded from a text file and even passed dynamically to the host object via Max messages.

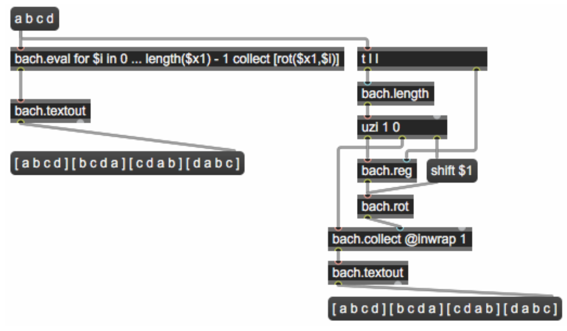

a b c d
bach.eval for $i in 0 ... length($x1) - 1 collect [rot($x1,$i)]
t l l
bach.length
bach.textout
uzi 1 0
[a b c d] [b c d a] [c d a b] [d a b c]
bach.reg
shift $1
bach.rot
bach.collect @inwrap 1
bach.textout
[a b c d] [b c d a] [c d a b] [d a b c]

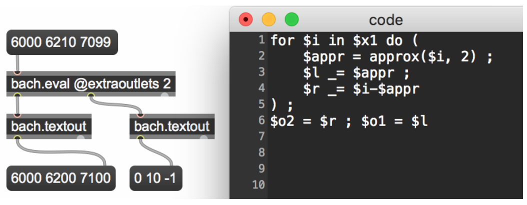
Fig. 3. A comparison between an *llll* manipulation process described through a snippet of *bell* code (in the `bach.eval` object box) and the corresponding implementation within the standard graphical dataflow paradigm of Max. The code should be mostly straightforward for readers familiar with the *bach* library and a textual programming language such as Python, considering that the `[ ... ]` paired operator encloses one or more elements into a sublist, according to the general syntax of *lllls*.

The intended usage paradigm of `bach.eval` is similar to that of the expr family: `bach.eval` objects are meant to carry out relatively simple computational tasks, and to be sprinkled around the patcher among regular *bach* and Max objects taking care of the UI, MIDI, DSP, event scheduling and so on.

Snippets of *bell* language can also be passed to other objects as well for fine-tuning their behavior, as a replacement for lambda loops. Moreover, an intended (albeit not straightforward) development is to allow `bach.score` and `bach.roll` to be scripted in *bell*, thus allowing far more complex interactions than what is already possible through the syntax described above.

5 Conclusions and future work

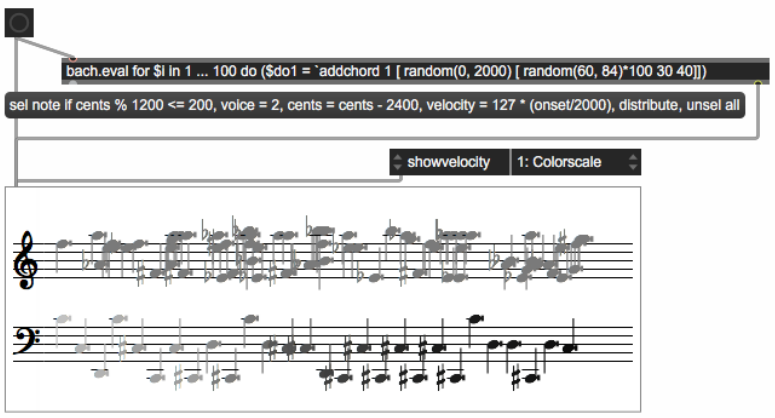
We have presented some historical and theoretical background about the computational models of Max and other related programming languages and envi-



```
6000 6210 7099
bach.eval @extraoutlets 2
bach.textout
bach.textout
6000 6200 7100
0 10 -1
```

```
for $i in $x1 do (
    $appr = approx($i, 2) ;
    $l _= $appr ;
    $r _= $i-$appr
) ;
$o2 = $r ; $o1 = $l
```

Fig. 4. A snippet of *bell* code approximating a list of midicents to the nearest semitone, and returning the distances from the semitone grid from a different outlet. Here, the code has been typed in a separate text editing window (shown on the right). The `$o1` and `$o2` pseudovariables assign results to the extra outlets declared in the `bach.eval` object box. The main, rightmost outlet returning the actual result of the computation (which, in this example, is the last term of the sequence defined by the `;` operators, that is, the value of the `$l` variable as passed to the first extra outlet) is left unused here. The language has several other features not shown here, including named and anonymous user-defined functions with a rich calling mechanism.



```
bach.eval for $i in 1 ... 100 do ($do1 = `addchord 1 [ random(0, 2000) [ random(60, 84)*100 30 40]])
sel note if cents % 1200 <= 200, voice = 2, cents = cents - 2400, velocity = 127 * (onset/2000), distribute, unsel all
```

Fig. 5. An example of usage of *bell* in combination with `bach.roll`'s in-place syntax: 100 notes are generated in the first voice with random onsets (between 0 and 2 seconds) and random pitches (between middle C and the C two octaves above, on a tempered semitonal grid); then all C's, C♯'s and D's are selected (i.e. notes whose remainder modulo 1200 is less than or equal to 200), assigned to the second voice, transposed two octaves below, remodulated with a velocity crescendo and distributed equally in time.

ronments, and subsequently described how the *bach* library can be helpful with writing clear and maintainable programs, through some specific features aimed at implementing different programming approaches and styles on top of it.

Overall, we think that time is ripe for advocating the adoption of more structured and theoretically grounded approaches to working with this successful and widely used tool. We hope that this article may be a step in that direction: further steps should involve, on the one hand, an actual survey of real-life use cases, possibly with the involvement of the community of *bach* users; and, on the other hand, a more precise and organic formalisation of good and scalable programming practices in Max, which might prove quite different from the ones typical of more traditional programming languages.

References

1. C. Agon. *OpenMusic : Un langage visuel pour la composition musicale assistée par ordinateur.* PhD thesis, University of Paris 6, 1998.
2. 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.
3. A. Agostini and D. Ghisi. A Max Library for Musical Notation and Computer-Aided Composition. *Computer Music Journal*, 39(2):11–27, 2015/10/03 2015.
4. A. Agostini and J. Giavitto. *bell*, a textual language for the bach library. In *Proceedings of the International Computer Music Conference (to appear)*, New York, USA, 2019.
5. M. E. Conway. Design of a separable transition-diagram compiler. *Communication of the ACM*, 6(7):396–408, 1963.
6. P. Desain et al. Putting Max in Perspective. *Computer Music Journal*, 17(2):3–11, 1992.
7. N. Didkovsky and G. Hajdu. Maxscore: Music Notation in Max/MSP. In *Proceedings of the International Computer Music Conference*, 2008.
8. D. Ghisi and A. Agostini. Extending bach: A family of libraries for real-time computer-assisted composition in max. *Journal of New Music Research*, 46(1):34–53, 2017.
9. G. Kahn. The semantics of a simple language for parallel programming. In *proceedings of IFIP Congress'74*, pages 471–475, 1974.
10. M. Laurson and M. Kuuskankare. PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL. In *Proceedings of International Computer Music Conference*, pages 142–145, Gothenburg, Sweden, 2002.
11. Y. Orlarey, D. Fober, and S. Letz. Faust: an efficient functional approach to dsp programming. *New Computational Paradigms for Computer Music*, 290:14, 2009.
12. M. Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.
13. S. Scholl. *Musik — Raum — Technik. Zur Entwicklung und Anwendung der graphischen Programmierumgebung "Max"*, chapter Karlheinz Essls RTC-lib, pages 102–107. Transcript Verlag, 2014.
14. T. Winkler. *Composing Interactive Music: Techniques and Ideas Using Max.* The MIT Press, 1998.