# Programming in style with bach (extended version)

Andrea Agostini, Daniele Ghisi, Jean-Louis Giavitto

# Programming in style with *bach*

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

[1] Conservatory of Turin, andreaagostini@conservatoriotorino.eu
[2] University of California, Berkeley – CNMAT, danieleghisi@berkeley.edu
[3] Sciences et Technologies de la Musique et du Son (STMS) – CNRS, IRCAM, Sorbonne Université, jean-louis.giavitto@ircam.fr

**Abstract.** Several programming systems for computer music are based upon the data-flow family of programming paradigms. In the first part of this article, we shall introduce the general features and lexicon of data-flow programming, and review some specific instances of it with specific reference to computer music applications. We shall then move the discussion to Max's very peculiar take on data-flow, and evaluate its motivation and shortcomings. Subsequently, we shall show how the *bach* library can support different programming styles within Max, improving the expression, the readability and the maintenance of complex algorithms. In particular, the latest version of *bach* has introduced *bell*, a small textual programming language embedded in Max and specifically designed to facilitate 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 [15] and Faust [19], 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 [22].

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 [4].

## 2 Dataflow computational models

The concept of *dataflow* is an old one, dating back at least to [8], 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 dataflow model

The term *dataflow* refers to a whole family of execution models that can be described by autonomous entities, called *actors* or *processes* interacting only through links connecting their *input* and *output ports*[4]. An actors triggers a computation according to the availability of data on its input ports and delivers its results on its output ports. The sequence of data passing through a link make up a *stream* and each value in the sequence is called a *token*. A token is sent through an output port and received by one or more destination actors, each of which retrieves the token through an input port. Often a dataflow graph (DFG) defines a whole computation that must be iterated periodically on a sequence of data (periodic dataflow). This is simply modeled by considering the stream of inputs.

The different dataflow execution models differ in the static or dynamic nature of the connection graph, the activation strategy of the actors, the capacity of the links to temporarily store the data produced (FIFO buffer), and the type of computation performed by each actor. By varying these parameters, the dataflow model is able to describe, in a unified manner, execution models ranging from electronic circuits to functional programming through audio graphs.

### 2.2 Homogeneous synchronous dataflow

One specific dataflow model is particularly useful, since it corresponds to the kind of computations done in real-time signal processing: the *synchronous dataflow* (SDF), also called the *static dataflow*. In this specific model, the graph is fixed and the order in which actors trigger their computations does not depend on the data that is processed (*i.e.*, the value of the token).

In *homogeneous SDF*, an actor fires its computation when there is a token on each of its input port and produces a token on each output port. In other word,

[4] In the usual Max terminology, actors correspond to boxes (object or message) and input and output ports corresponds respectively to inlets and outlets.

the connections are FIFO of size one and the activation of actors never leads to a buffer overflow. Homogeneous SDF corresponds well to audio graphs where tokens correspond to audio buffers and actors to unit generators, delays, filters and other effects. It also matches well the behavior of synchronous (clocked) electronic digital circuit: an actor is an electronic gate and a token is the value of the voltage on a pin. A clock is used to acquire and deliver the tokens simultaneously for all the gates.

### 2.3 Dynamic dataflow

The SDF model is not particularly expressive. A number of dataflow variants have been developed that relax the constraints of SDF. We loosely refer to them as *dynamic* dataflow (DDF). In DDF, the activation of an actor can be more complicated and the firing strategy may change in time, for instance with some internal state of the actor. The number of tokens produced can also vary on each output ports. This makes possible to express conditional firing, *i.e.*, an actor triggers its computation only if the input token has a particular value.

For instance, the audio graph in Max[5] is a SDF, while the control graph is a DDF in which the firing of an actor is driven by the so-called 'hot' inlets, and not by the availability of the incoming messages on all inlets. Which inlet is 'hot' may also depend on the specific data arriving in non-'hot' (or 'cold') inlets (as for example in the *switch* box). Moreover, the activation of a box does not lead necessarily to each of their outlets firing (as for example in the *gate* box). As for patchcords, the ones connecting an outlet to a cold inlet do not store the whole sequence of messages in transit before being consumed by the target, as only the last one (*i.e.*, the most recent) is kept until its consumption.

DDF is also needed to handle several rates in an audio graph, for instance for the handling multimedia streams (audio and video).

### 2.4 Data-driven and demand-driven implementation strategies

Historically, there are two main approaches for the software implementation of dataflow: the *data-driven* approach and the *demand-driven* one.

*Data-driven.* In the data-driven implementation, tokens travel in the interconnection graph and fire according to the actors. The activation of the actors propagates in the graph like a wave starting at the inputs and ending at the outputs, *i.e.* actors that have no output ports and that are used to 'deliver' the result of the calculation (be it in the form of text, audio, MIDI data, image display, etc.).

In SDF, tokens can be presented asynchronously to the input ports but the actors fire only when all their inputs bear a token. If the graph is homogeneous, then a token is produced on each of the output ports and, subsequently, actors that lie at a distance 1 from the inputs are activated, and so on. Therefore, the

[5] We shall talk diffusely about Max in the later chapters, and only introduce it as an example here for readers already familiar with it.

functioning of the dataflow graph in time can be seen as a sequence of 'rounds', that is, transactions in which the first token of all input streams is consumed and a token is produced for all output streams (hence the term "synchronous"). In this context, the order of activation of the actors can be fixed *a priori*, through a static analysis of the graph itself, and can be computed by a simple topological sort of the graph, as long as it is loop-free (also see below).

Both Max's and Pure Data's audio and control graphs are implemented using a data-driven approach. Once the user has laid out graphically the audio graph, the program analyses it statically and, if there are no errors (typically resulting from a feedback connection) an internal representation of it is generated, also establishing a fixed sequence of activations. Things are different for the control graphs: since their behavior depend on the data they receive, their scheduling cannot be fixed statically.

*Demand-driven.* The demand-driven approach is more convenient to handle DDF. In this approach, the outputs of the graph require a token from their connected actors, which in turn transmit the request to their inputs, so that a wave of requests propagates through the graph, from its outputs to its inputs. Each input of the graph answers the request by sending the requested token, which is then processed, creating a second wave of computations from the inputs to the outputs.

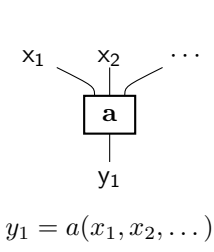
The demand-driven approach is interesting because it may handle easily non-homogeneous dataflow graphs. It is not by chance that it was chosen a model for the Patchwork family of tools, OpenMusic and PWGL. For instance, in OpenMusic's dataflow graphs, each node implements a function. Functions can be non-strict, i.e., they may use only part of the inputs to produce the output. An example is the conditional operator *if* that returns the value of one of its two inlets following the boolean value present on a third inlet. Handling correctly non-strict functions is necessary for defining recursive functions: if the conditional operator were strict, the retrieval of both inputs would be required and the computation of recursive functions would loop forever, since even in the terminal case all branches would have been computed.

The demand-driven approach makes the implementation of non-homogeneous dataflow easier. In these models, the firing of an actor may consume an arbitrary number of token on each input, and may produce an arbitrary number of tokens on each output. In this case, an actor would request to its inputs the number of tokens it needs. This can be used to simplify the handling of multirate audio graphs.

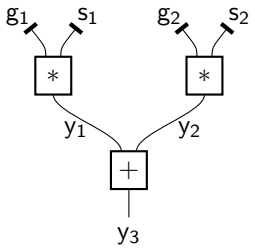
### 2.5 The Kahn principle: reconciling the operational and the algebraic view

The previous considerations sketch an *operational* view of the functioning of a dataflow graph: the graph is a machine similar to an electrical circuit and the functioning of the graph amounts to moving tokens along the links and to transform the values of the tokens when passing through an actor.

Another, more abstract, view of the graph has an algebraic flavour. The idea is to assign a variable to each connection from an output port to an input port and to describe the interaction graph by a set of equations linking such variables — one equation for each output port. We can assume that there is only one output port labeled with variable $y$ for an actor **a**.[6] Then, the equation $y = a(x_1, x_2, \dots)$ represents the relationships between the inputs $x_1, x_2, \dots$ and the output $y$ (cf. the diagram below, at left). The symbol $a$ refers to the behavior of actor **a** and there are several mathematical ways to model this behavior. For instance, $a$ can be a function of the tokens. In this case, the variable denotes the "current token value" instead of the entire stream.




$y_1 = a(x_1, x_2, \dots)$




$$\begin{cases} y_1 = g_1 * s_1 \\ y_2 = g_2 * s_2 \\ y_3 = y_1 + y_2 \end{cases}$$

With this interpretation, the set of equations associated to a DFG can be interpreted as an ordinary set of equations between real values. The right side of the diagram above models a simple two-input mixer, with gain $g_1$ and $g_2$. Notice that the set of equations is a set of *fixed point equations*, that is, each equation takes the form

$$\textit{variable} = \text{expression-involving-variables}\,.$$

The interesting point is that, in this example, solving the algebraic equations associated with the DFG gives the same result as the operation of the DFG in terms of tokens movement and tokens transformation (the operational view).

However, restricting the behavior of a node to a function of the current input tokens is too restrictive: for instance, this mathematical model cannot represent a node whose behavior depends of an internal memory, like a delay (see next section). But the actor operation in the example above can be interpreted as function acting on entire streams, the operators $*$ and $+$ being the point-wise multiplication and the point-wise addition of the stream's elements.

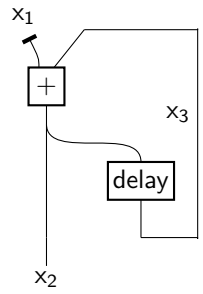
A DFG where the behavior of each actor can be modeled as a continuous function of the entire input streams to the output streams is said to be *pure*. It has been postulated by Kahn [14], and proven only ten years later by Faustini in [11], that the operational behavior of a pure DFG computes *the least fixed point solution of its associated set of equations*. The understanding of this result requires to define precisely what is meant by *continuous* and *least*. Intuitively,

[6] If an actor has multiple output ports, it can be divided into as many actors as outputs, each taking the same inputs and having only one output.

we can say that a set of equations may have zero or more solutions. This result states that there is at least one solution, and that the graph computes the stream solution that is minimal according to a carefully chosen stream comparison order [16].

### 2.6 Feedback loops

The previous example of fixed point equations is not very interesting: these equations remain simple because there is no circular dependencies between variables. Fixed point equations of this kind are easily solved by substituting known variables (here $g_1, s_1, g_2$ and $s_2$) in the expressions at the right hand side of the equations and repeating this process until there is no further unknown variable. In the example, the first iteration defines variables $y_1$ and $y_2$ and the second iteration defines $y_3$.




$$\begin{cases} x_2 = x_1 + x_3 \\ x_3 = \mathit{delay}(x_2) \end{cases}$$

The Kahn principle is valid for any pure DFG, including DFG having feedback loops. In the example at left, $x_1$ is an input of the DFG and the operator *delay* is on the feedback loop going from the output of + to its input. Here *delay* is a function shifting the entire stream $s = v_0 \cdot v_1 \cdot v_2 \cdot \ldots$ in time: $\mathit{delay}(s) = 0 \cdot v_0 \cdot v_1 \cdot v_2 \cdot \ldots$. Operationally, the delay is a memory initially filled with 0.

When a token is present at the input, the *delay* outputs the value in the memory and then update the memory with the input value. One can convince oneself that this DFG computes the sum of its successive inputs.

The corresponding equations exhibit a self-dependency: if we substitute the definition of $x_2$ into the right hand side of $x_3$, we get $x_3 = \mathit{delay}(x_1 + x_3)$ which cannot be solved by substituting the variables but we can look instead at what happens at the level of the stream's elements: let the stream $x_i$ be $x_i^0 \cdot \ldots \cdot x_i^j \cdot \ldots$

$$x_3^0 \cdot \ldots \cdot x_3^j \cdot \cdots = 0 \cdot (x_1^0 + x_3^0) \cdot \ldots \cdot (x_1^j + x_3^j) \cdot \ldots$$

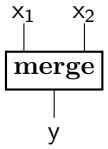
that is,

$$x_3^0 = 0, \qquad x_3^{j+1} = x_1^j + x_3^j$$

or, in other word, $x_3$ is the running sum of the successive values of $x_1$.

### 2.7 Time in dataflow: synchronous languages

In the previous example, one may notice that *delay* is a function acting on stream, not a function acting on tokens. This allows the specification of behaviors implying a transfer of information from one token to the next one. However, this is not enough to model the behavior of a node acting to merge to streams.




The merge actor simply transmits the tokens that present themselves at its inputs towards its outputs, in their arrival order. The first problem is to decide what happens when tokens are simultaneously available on the two inputs. One may chose to transmit the token on the first input and to drop the second token. This way, the merge operator has a deterministic behavior. But even so, it cannot be modeled as a stream function.

The reason is that SDF does not provide enough information on the passing of time. For instance, suppose that stream $s$ represents the value 0 produced every two seconds starting from date 0, while $s'$ represents the value 1 produced every two seconds, starting from date 1. Then $\mathsf{merge}(s, s')$ has for result the stream $0 \cdot 1 \cdot 0 \cdot 1 \cdot \ldots$ Let $s''$ be the stream representing the value 1 produce every four seconds, starting from date 1, then we have $\mathsf{merge}(s, s'') = 0{\cdot}1{\cdot}0{\cdot}0{\cdot}0{\cdot}1{\cdot}0{\cdot}0{\cdot}0{\cdot}1{\cdot}\ldots$ even if $s' = s''$. Obviously, the lack of temporal information in the stream makes the stream $s'$ and $s''$ indistinguishable, even if they model two different processes and so there is no function *merge* on streams.

The problem was recognized in the real-time programming community and lead to the development of synchronous languages like LUSTRE [13]. In this dataflow programming language, the idea is to "align" each stream so the production of the $i^{\text{th}}$ value of any stream takes place on the same date. This is achieved by introducing a special token value $\perp$ meaning "an actual value is not available". With this token, the *synchronized streams* $s'$ and $s''$ are presented by:

$$s' = \perp \cdot 1 \cdot \perp \cdot 1 \cdot \perp \cdot 1 \cdot \perp \cdot \ldots$$
$$s'' = \perp \cdot 1 \cdot \perp \cdot \perp \cdot \perp \cdot 1 \cdot \perp \cdot \ldots$$

The Kahn principle holds for fixed point equations on these synchronized streams and more actors can be specified as stream functions.

### 2.8 Declarative programming

If we insist on Kahn's principle, this is because it leads to a *declarative programming language*. It is commonly said that that declarative programming focuses on what the program should accomplish (whereas imperative or procedural programming focuses on how the program should achieve the result). This description is rather vague, and a more effective way to look at it is to characterize declarative programming languages as programming languages making possible equational reasoning, where an entity may always be substituted by its definition. In other word, the statement of the language can be seen as a set of mathematical definitions, that is, a set of fixed point equations, and program execution amounts to finding a solution to these equations.

In this context, a variable in a program is handled exactly as a mathematical variable: a reference to a well defined although possibly unknown value. This departs considerably from the notion of variable in an imperative program where

a variable refers to a memory location which can holds various values during the program execution.

By definition, declarative programming enjoys *transparential referency*: any expression can be replaced with its corresponding value without changing the program's behavior [24].

A program becomes a mathematical object that can be manipulated using classical mathematical methods, for instance to prove program properties for efficient compilation, to replace an expression by an equivalent expression less costly to compute, to rearrange program computations much more freely, possibly to execute some tasks in parallel.

The Kahn principle give us a tool to solve some equations on streams. It also allows us to reason algebraically on the operational properties of a dataflow program.

## 3 Dataflow and computer music programming languages

The previous section may seem very technical but they have far reaching consequences in the realm of computer music programming languages. Several languages and systems, such as OpenMusic, PWGL and Faust, are based upon more or less pure SDF:

- The core of OpenMusic and PWGL is a demand-driven implementation of a pure dataflow. However, the language embeds some imperative features that are useful for interacting with users (*e.g.*, fixing some computed value to be reused in later evaluation). Interestingly, a specific version of OpenMusic, OM#, extends the demand-driven execution model towards an hybrid data- and demand-driven execution model which simplify considerably the programming of reactive system [5].
- Faust [18] is a perfect example of data-driven, pure synchronous dataflow. Equational reasoning is heavily used in the Faust compiler, for code generation (by program transformation) but also for the automatic parallelisation [20].
- Synchronous programming—which involves synchronous streams, either in the context of dataflow or in the context of imperative languages—is the model embraced by signal processing languages like CHUCK [25] or KRONOS [17].

### 3.1 Dataflow programming in Max and Pd

As previously mentioned, Max and PureData implement two different dataflow systems, respectively devoted to audio signals and control messages. The earliest versions of Max only implemented the former system, as Max itself had been originally conceived as a tool for building interfaces controlling external audio hardware. The audio system was only added to Max in the late 1990s, and was so distinctly separated from the 'core' of Max itself that it used to be sold separately under a different name, MSP.

The Max audio graph is a relatively simple case of pure SDF, whose functional nature is somewhat less explicit than that of Faust, but not too different from it. Our discussion will only focus on the control graph, its significantly different paradigm and the consequences this bears with respect to the different applications Max lends itself to, with a specific focus on the implementation of compositional processes.

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 a profound consequences: there is no transparential referency in the control DFG.

For example, multiple links ('cords' in the Max jargon) can be connected to a single inlet corresponding to an implicit merge. But there is no notion of synchronous stream in Max: there is no notion of timestamp that can be used to "align" the streams values and to recover transparential referency. Another example of non-transparential referency: sending the same message, that is, outputting a token on some link, does not achieve the same effect if done on the timer thread or on the main thread (because of different message priorities, its subsequent handling may differ).

### 3.2 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 [21], 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, iteration, 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 [26,23,10]. We shall focus on one of them, the *bach* package, which has been conceived and is maintained by two of the authors.

### 3.3 The *bach* package

The *bach* package[7] for Max is an open source 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 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.[8]

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 *llll*s 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 *llll*s in a patcher just like they do with regular Max

[7] `www.bachproject.net`

[8] `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.

messages, they cannot access their contents unless *llll*s are explicitly converted into a Max-readable format, which on the other hand has other limitations (for a detailed explanation, see [4]). Thus, *bach* contains a large number of objects that somehow extend to *llll*s 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 the 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 [4]. 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[9], and by the fact that it provides the foundation for the *cage* and *dada*[10] libraries [12].

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 *llll*s.

## 4 Different programming styles and approaches in Max

### 4.1 Imperative approach

It has been observed that Max is essentially an imperative system in disguise [9]: 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. This is complicated by the fact that many objects whose purpose is carrying out specific operations also maintain a state that can be used to store values for later use. For example, most arithmetic operators have two inlets: the left one, called *hot*, sets the first term of the operation and triggers the calculation; the right one, called *cold*, only sets the second term of the operation. So, the typical way to perform, say, a sum of two numbers is first setting the right term, and then the left term, thus calculating the result. This somewhat idiosyncratic design can be leveraged to perform more complex tasks in quite a synthetic way. For example (see Fig. 1), a typical way to build a running accumulator is to feed

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

[10] 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.

each new number to be accumulated in the hot inlet of a `+` operator, and feeding back the result in the right inlet, so as to have it ready to be summed to the next incoming value. Likewise, a differentiator can be built by sending each new value to the hot inlet of a `-` object first and the cold inlet immediately after, so that it is ready to be subtracted from the next incoming value. Whereas this may be convenient for such simple cases, it can become extremely complicated as the complexity of the problem grows. It is easy to build patches in which state is distributed into many seemingly random objects, with spaghetti connections that can only be made sense of by following attentively the flow of every bit of information through the patch graph.

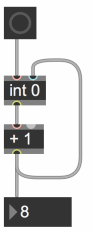



**Fig. 1.** The typical design of a simple accumulator in Max: the *int* module stores the current number in a hidden state; when the top button is clicked, the state is output, incremented, and stored in the *int* module anew. Notice that, as a general rule in Max, the internal state is hidden from the user: at the moment in which the screenshot was taken the internal state had been updated to 8 (as the bottom number displays) although the argument of the *int* module still shows '0', which represents the *initial* state. Note also that the functionning here is asynchronous and controled through hot and cold inlets which differs from the synchronous functioning of the DF accumulator in sect. 2.6

On the other hand, it is possible to make this imperative style more explicit by adopting some good practices, like 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. Considering the examples in some widespread textbooks [6][7][26] and the Max documentation, as well as some informal reckoning of the patches that users share on the official Max forum, it seems to us that these objects are

seldom used. One likely reason is that they are virtually never necessary, and tend to make patches larger and slightly less efficient. On the other hand, 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 *llll*s, 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 *llll*s, each associated to a unique name. `bach.shelf` objects can be themselves named, thus defining namespaces: this means that *llll*s 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.

### 4.2 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.

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). The operations are immediately performed and the score is updated (see Fig. 4). These messages enable the creation, the edition and the deletion of individual notation items, such as a single measure or a single note, and can actually be seen as methods of the items themselves, arranged according to a precise hierarchy and sharing a certain number of common properties (such as having a symbolic name, being selectable, etc.).

This mechanism is strongly inspired by an object-oriented approach: references to the notation items to be modified are acquired via a selection mechanism, and then messages are sent to them. A set of items can be selected

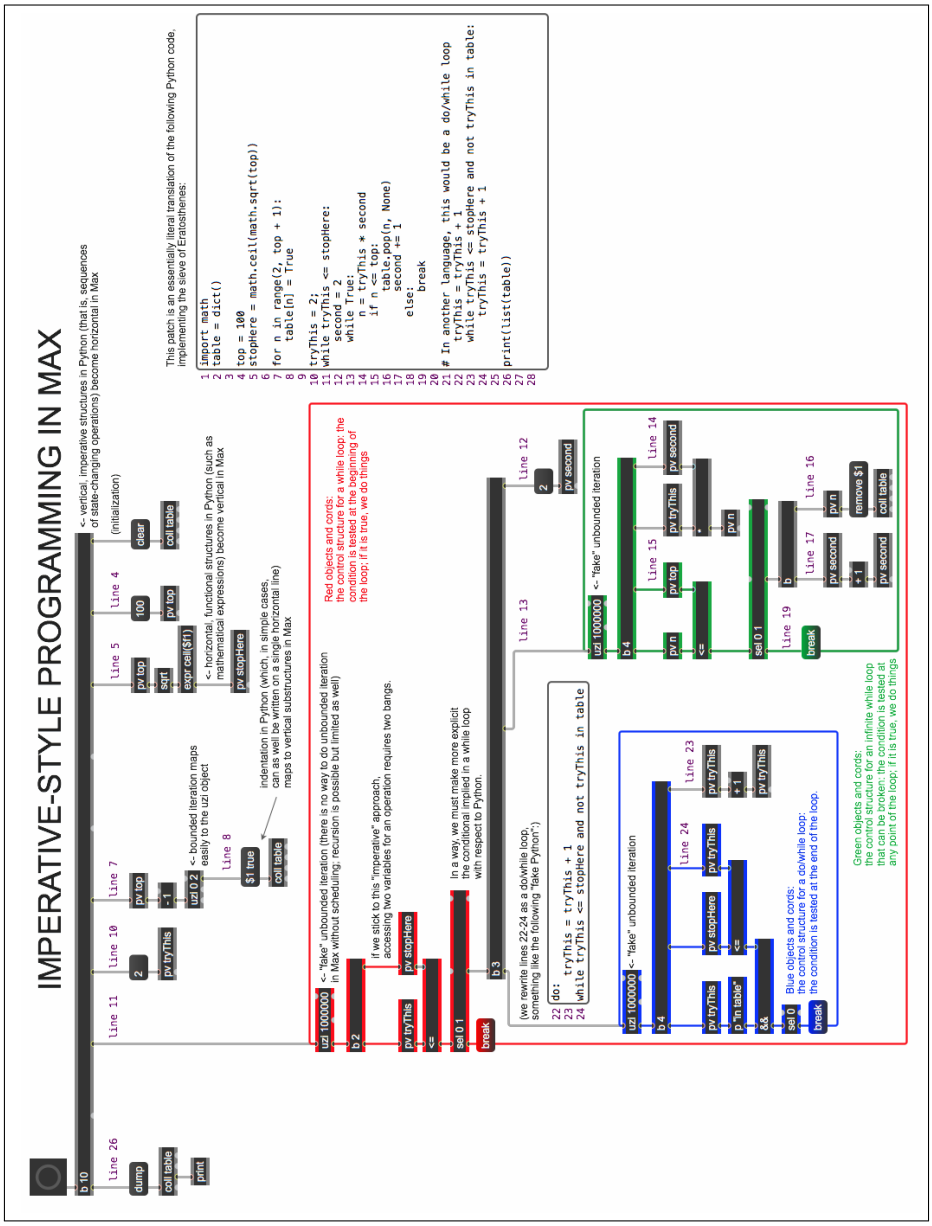



**Fig. 2.** An implementation in strict imperative style of the sieve of Eratosthenes, an algorithm for finding prime numbers. The patch is a direct translation of the Python code shown in the box. We have chosen not to write very idiomatic Python code for making it closer to pseudocode. In the patch, we have avoided — among the other things — storing data in the cold inlets of objects: every piece of data meant to be reused is stored in `pv` objects, from which they are retrieved at need. The result is quite redundant, but the structure of calculation is clear.

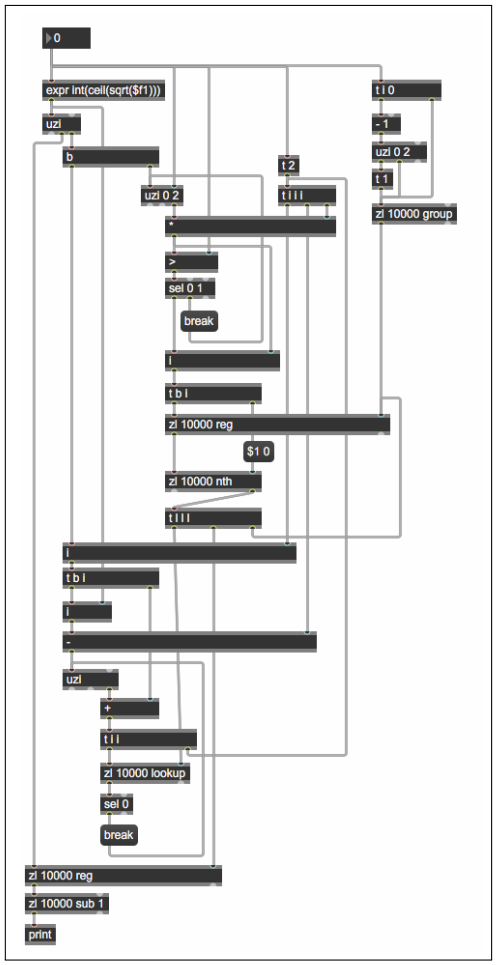



**Fig. 3.** A more synthetic implementation of the sieve of Eratosthenes. Compared to the example of fig. 2, the patch is more compact but also much less readable.

graphically, or through a query in the form of a message such as `sel note if voice == 2 and pitch % C1 == F#0`, and then modified by means of messages such as `duration = velocity * 10`. Most musical properties can be modified in this way, and the expressions determining the assignation support a standard set of predefined variables, capturing the current state of the object (`onset`, `cent`, `duration`, `velocity`, `index`, `part`, `grace`, and so on).

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).

Moreover, there are available methods to perform routine tasks such as copying/pasting score content or slot[11] information, inserting or deleting pitch breakpoints, modifying portions of score, snapping items to a temporal grid, making selection monophonic, adding or modifying slot content, renaming, distributing elements evenly in time, and so on. Notation objects send notifications whenever their state is changed, so that any of the aforementioned methods can be also triggered by user operations on the score, in a reactive way. Some of these messages work in conjunction with the playback system, so that users can, among other things, retrieve properties of the currently played notes or move the playback cursor.

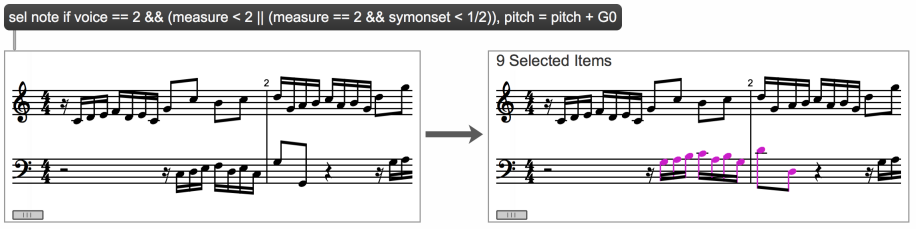



**Fig. 4.** A very simple example of in-place modification: notes belonging to the second voice and whose onset lies before the middle of the second measure are selected and transposed up a perfect fifth (the image shows both the state of the score before and after the click on the message).

### 4.3 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 *llll*s. 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.

[11] Slots are containers of arbitrary data attached to notes and chords.

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. 5), 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 [4].

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.

## 5 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.

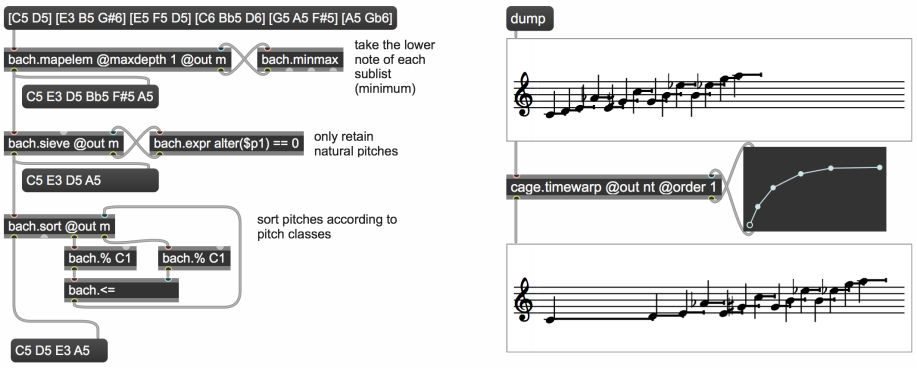



**Fig. 5.** The cross-connected and loop-connected patch cords attached to *bach.mapelem*, *bach.sieve*, *bach.sort* and *cage.timewarp* modules form several instances of the so-called lambda loop. The left-side example should be straightforward. In the right-side example, the temporal distribution of 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.

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.

The corresponding, native way to do the same in Max is writing an external object in C. Whereas this was originally meant to be a part of the regular Max workflow, it is undoubtedly a quite complicated task for today's average Max user, requiring to master the C programming language and the compilation chain. Moreover, the write-test-debug cycle requires to restart the whole Max environment at every modification made to the object, and errors in the code are not unlikely to crash Max. Finally, unlike what happens in OM and PWGL, the code for a Max object is required to include a quite heavyweight infrastructure taking care of the communication with the Max environment and only remotely related to the actual problem meant to be tackled.

As expressing algorithms through textual coding can be quite convenient but the C API has the aforementioned drawbacks, over the years various other programming languages have been embedded into Max through higher-level APIs, including Java, JavaScript, Lua, all included in the Max distribution. Although very effective for various kinds of operations, these bindings are not optimal for interacting with *bach*, for a number of reasons that are detailed in [3] and mostly amounting to two areas:

- As mentioned before, the all-encompassing data structure of *bach* is the *llll*, which is not easily expressed in any of the above languages.
- These bindings require some pieces of quasi-boilerplate infrastructure, such as the explicit management of inlets, outlets and messages sent to the enclosing object, that make the writing of code significantly more complex, compared to the ease and directness of embedding Lisp code in Open Music and 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 *llls*.

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 *llll*s respecting, as far as possible, the conventions and naming of the corresponding *bach* objects.
– Implicit concatenation of elements into *llll*s, 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 llll*s, but also paying homage to the historic Bell Labs). A detailed description of its syntax can be found in [3], whereas, for the scope of this article, a few examples should suffice (see Fig. 6, 7 and 8).

*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.

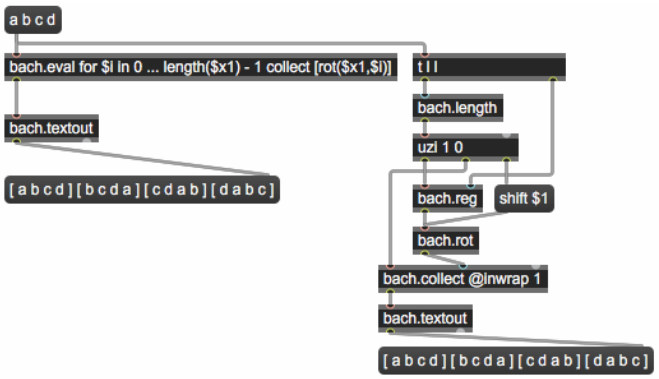



**Fig. 6.** A comparison between an *llll* manipulation process described through a snippet of *bell* code (in the `bach.eval` object box) and the corresponding implementation within the standard graphical dataflow paradigm of Max. The code should be mostly straightforward for readers familiar with the *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 *llll*s.

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 disseminated 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 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

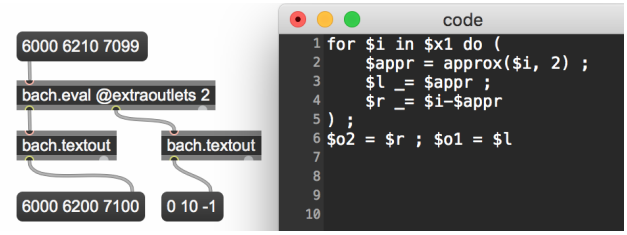



**Fig. 7.** A snippet of *bell* code approximating a list of midicents to the nearest semitone, and returning the distances from the semitone grid from a different outlet. Here, the code has been typed in a separate text editing window (shown on the right). The `$o1` and `$o2` pseudovariables assign 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.

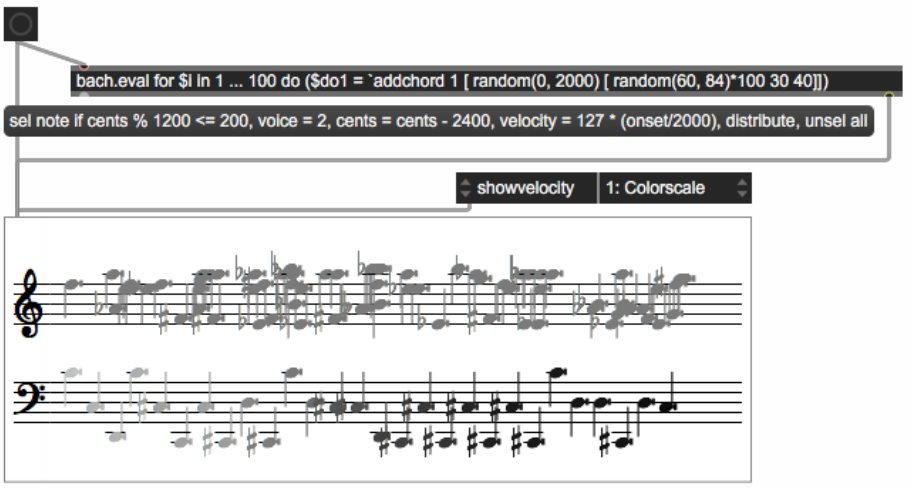



**Fig. 8.** An example of usage of *bell* in combination with `bach.roll`'s in-place syntax: 100 notes are generated in the first voice with random onsets (between 0 and 2 seconds) and random pitches (between middle C and the C two octaves above, on a tempered semitonal grid); then all C's, C♯'s and 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.

scripted in *bell*, thus allowing far more complex interactions than what is already possible through the syntax described above.

## 6 Conclusions and future work

We have presented some historical and theoretical background about the computational models of Max and other related programming languages and environments, 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. These features are rooted in practical considerations and experience, and allow one to escape the limitations of pure formal models.

More generally, 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; 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; and, most likely, the conception and development of new tools to encourage them and facilitate their adoption.

## References

1. Agon, C.: OpenMusic : Un langage visuel pour la composition musicale assistée par ordinateur. Ph.D. thesis, University of Paris 6 (1998)
2. Agostini, A., Daubresse, E., Ghisi, D.: *cage*: a High-Level Library for Real-Time Computer-Aided Composition. In: Proceedings of the International Computer Music Conference. Athens, Greece (2014)
3. Agostini, A., Giavitto, J.: *bell*, a textual language for the bach library. In: Proceedings of the International Computer Music Conference (to appear). New York, USA (2019)
4. Agostini, A., Ghisi, D.: A Max Library for Musical Notation and Computer-Aided Composition. Computer Music Journal **39**(2), 11–27 (2015/10/03 2015). https://doi.org/10.1162/COMJ_a_00296, `http://dx.doi.org/10.1162/COMJ_a_00296`
5. Bresson, J., Giavitto, J.L.: A reactive extension of the openmusic visual programming language. J. of Visual Languages & Computing **25**(4), 363–375 (2014)
6. Cipriani, A., Giri, M.: Musica Elettronica e Sound Design. ConTempoNet (2013)
7. Colasanto, F.: Max/MSP: Guía de Programación para Artistas. CMMAS (2010)
8. Conway, M.E.: Design of a separable transition-diagram compiler. Communication of the ACM **6**(7), 396–408 (1963)
9. Desain, P., et al.: Putting Max in Perspective. Computer Music Journal **17**(2), 3–11 (1992)
10. Didkovsky, N., Hajdu, G.: Maxscore: Music Notation in Max/MSP. In: Proceedings of the International Computer Music Conference (2008)

11. Faustini, A.A.: An operational semantics of pure dataflow. In: Nielsen, M., Schmidt, E.M. (eds.) 9th International Colloquium on Automata, Languages, and Programming. LNCS, vol. 120, pp. 212–224. Springer Verlag (1982), equivalence sem. op et denotationelle
12. Ghisi, D., Agostini, A.: Extending bach: A family of libraries for real-time computer-assisted composition in max. Journal of New Music Research **46**(1), 34–53 (2017)
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE **79**(9), 1305–1320 (1991)
14. Kahn, G.: The semantics of a simple language for parallel programming. In: proceedings of IFIP Congress'74. pp. 471–475. North Holland (1974)
15. Laurson, M., Kuuskankare, M.: PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL. In: Proceedings of International Computer Music Conference. pp. 142–145. Gothenburg, Sweden (2002)
16. Mosses, P.D.: Handbook of Theoretical Computer Science, vol. 2, chap. Denotational Semantics, pp. 575–631. Elsevier Science (1990)
17. Norilo, V., Rautatiekatu, P.: Introducing kronos-a novel approach to signal processing languages. In: Proceedings of the Linux Audio Conference. pp. 9–16. Maynooth: NUIM (2011)
18. Orlarey, Y., Fober, D., Letz, S.: Syntactical and semantical aspects of faust. Soft Computing **8**(9), 623–632 (2004)
19. Orlarey, Y., Fober, D., Letz, S.: Faust: an efficient functional approach to dsp programming. New Computational Paradigms for Computer Music **290**, 14 (2009)
20. Orlarey, Y., Fober, D., Letz, S.: Parallelization of audio applications with faust. In: Proc. of the 6th Sound and Music Computing Conference, Porto, PT. pp. 99–112 (2009)
21. Puckette, M.: Max at seventeen. Computer Music Journal **26**(4), 31–43 (2002)
22. Puckette, M.: A divide between 'compositional' and 'performative' aspects of Pd. In: Proceedings of the First Internation Pd Convention. Graz, Austria (2004)
23. Scholl, S.: Musik — Raum — Technik. Zur Entwicklung und Anwendung der graphischen Programmierumgebung "Max", chap. Karlheinz Essls RTC-lib, pp. 102–107. Transcript Verlag (2014)
24. Søndergaard, H., Sestoft, P.: Referential transparency, definiteness and unfoldability. Acta Informatica **27**(6), 505–517 (1990)
25. Wang, G., Cook, P.R., Salazar, S.: Chuck: A strongly timed computer music language. Computer Music Journal **39**(4), 10–29 (2015)
26. Winkler, T.: Composing Interactive Music. The MIT Press (1998)