jMax recent developments

François Déchelle Maurizio De Cecco Enzo Maggi Norbert Schnell
dechelle@ircam.fr dececco@ircam.fr maggi@ircam.fr schnell@ircam.fr

IRCAM - 1, place Stravinsky - 75004 Paris

Abstract In this paper, we present recent developments of jMax. Currently supported platforms are listed. The evolution of the jMax computation engine toward a multi-threaded architecture and the new scripting tools of jMax’s graphical user interface are described. Musical applications are presented, together with considerations on Max’s programming paradigm.

1 Introduction

jMax is a new implementation of the Max language. It is a programming environment for real-time performance interaction and audio processing. It features a high level of portability, and is based on a Java graphical user interface.

2 Platforms

jMax is currently supported on the following platforms:

- SGI stations, both mono and multi-processors, using the SGI standard audio interfaces
- Linux on Intel compatible processors, using the Open Sound System audio and MIDI drivers

The Linux version features a very good performance and has shown that a machine comparable, in term of computing power, to the ISPW [LIN 91] can now be found in entry-level PCs.

For detailed benchmarks of supported platforms, the reader is invited to visit jMax home page at www.ircam.fr/jmax for the latest informations.

3 The FTS kernel

The framework of the jMax environment, based on a client/server architecture, has been described in [DÉC 95, DÉC 98a, MAG 96]. The two components of the system are the C written real-time computation engine, already known as FTS, and the JAVA graphical user interface.

The architecture of the FTS server is described in [DÉC 95, DÉC 98a, DÉC 99]. It consists mainly of a kernel containing the message system interpreter, the DSP computation engine [DÉC 95, DÉC 96], the scheduler and the device based input/output system [DÉC 95].

The FTS kernel has been rewritten to become completely multi-threaded in order to make full use of modern symmetric multi-processors machines, which are now most of the time based on cache-coherent memory architecture and support thread level programming interfaces. The currently used thread API is the Posix thread API, but the multithreading capabilities are not restricted to Posix and will use the native threading API on different platforms.

The first use of the multi-threading capabilities has been the re-implementation of the direct-to-disk services. Now based on a device, the sound files accesses are executed in separate threads, thus being able on multi-processors machines to run on a different processor from the processor running the DSP code interpretation. This feature allows the user to build applications using many channels of direct-to-disk real-time access, and still have one processor completely dedicated to signal processing and control execution.

Another use of the multi-threaded architecture, still in development, is the capability to separate control computation into “synchronous control”, running in the same thread as the DSP computation, and “asynchronous control”, running in a low priority thread. The advantage given by this possibility is that the “asynchronous control” computation load peaks will not interrupt the execution of signal processing and synchronous control, eliminating the need to increase the audio buffer size and therefore the latency, and keeping tight synchronization between signal processing and synchronous control computation.

4 Scripting

The jMax user interface integrates a scripting language [DÉC 98a, DÉC 99], used at several levels: configuration (choice and parameterization of devices or choice of packages), definition of packages
(i.e. dynamically loaded libraries of objects and Java editors), extension language (for instance definition of new edition commands).

The first implementation of scriptability in jMax used a JAVA implementation of Tcl, which has shown poor JAVA integration and lack of support. It has therefore been decided to switch to another scripting language, and the choice has been to move to Java implementations of the Scheme Lisp dialect. Scheme features numerous Java implementations that all exhibit good runtime performance and high Java integration.

5 Applications

Recent application development include real-time physical modeling based on the Modalys package [ECK 95, IOV 99], additive synthesis using inverse FFT [ROD 92] and score recognition for real-time soloist synchronization. The implementation of these applications was the occasion to exercise the variables mechanism introduced in jMax [DÉC 98a] and to experiment different computation paradigms.

5.1 Experimenting with variables and reference types

Traditionally, the Max programming paradigm allows two ways of defining an interaction between objects:

1. graphical connections from an object’s outlet to another object’s inlet represent either the flow of messages or audio signals

2. an object references another object by name in order to share a part of its state

The data transported by messages was so far restricted to array of atoms (i.e. type tagged values), each atom representing an integer, a floating-point number or a symbol. Recent developments implement atoms of reference types containing complex data like vectors of integer or floating-point numbers, floating-point matrices or spectral representations.

References by global names (like between the table objects) where replaced by the concept of named variables [DÉC 98a]. This concept allows an object to define a variable by exporting an atom of an arbitrary type which will be bound to a local name when the object is instatiated (e.g. mytab : table 100). Another object in the same scope can use a reference to this variable as an instantiation argument, using a $-operator with the variable name like in table $mytab.

The new implementation of modal synthesis [ECK 95, IOV 99] allows to construct modal synthesis algorithms using named variables to define bidirectional interactions between two objects and the association of an object to a computation context.

Since the jMax message system does not provide an adequate mechanism (including garbage collection) to handle copying reference types at runtime, the use of references were first restricted to objects instantiation. Recent developments explore as well the use of reference types in messages.

For messages containing arguments of simple types, Max gives a notion of a data flow language (it must be noted that sending a message always invoke a function call with arguments passing by value). Without the introduction of a complex handling of reference types including garbage collection, this notion can not be kept for reference types. Even if the introduction of computation by side effect finally breaks the data flow illusion of the Max language, it represents an efficient and powerful way of computation which is worth exploring.

The example of figure 1 shows a simple jMax patch of two graphs doing the same operations in two different ways. A table of amplitudes is filled with data from a file before it is multiplied, element by element, by a vector of weights and finally multiplied by 2. By convention the operator (like load and * in the example) output the received reference after applying their operation to it. Since the t object in the second graph outputs the incoming variable in right to left order from each output, both graphs produce identical outputs on the display.

Figure 2 shows an example of additive synthesis using computation by side effect in a different way. The object FMO outputs a reference to an optimized representation of an empty FFT spectrum for each synthesis frame (every 128 samples) in a message. Each of the FMOharmonics objects accept this message, add the harmonics received by their parameter inlets to the spectrum and forward it to the outlet. These calculations are done at the same logical time during jMax's message cycle driven by a periodic alarm ran by the FMO object. When the FMO object is scheduled by the DSP subsystem it computes an inverse FFT (256 samples) on the spectrum created from the three sets of harmonics and overlaps it with the previous synthesis frame before the sound output. This way efficient additive synthesis algorithms can be created in a modular way, while all modules share the same computation model and the same inverse FFT. Other classes are implemented, which add arbitrary partials and noise bands to an FMO spectrum using the same paradigm as FMOharmonics.
5.2 The score recognition project

A set of experimentation have been carried out with JMax in relation to the score recognition problem, in order to offer a wider set of tools and strategies for this somehow classical area of research. Given a computer-readable “solo” partition and a flow of data coming from a external performer, the task is to be able to recognize from the input stream specific events coded in the solo partition, and react accordingly following a second, “accompanyist” partition. The researches started from the analysis of the available tools, almost all based on variations of the Dannenberg strategy as presented in [DAN 84]. Some limitations of this model have arised, regarding the input domain, the matching criteria, and the error handling.

A new model to isolate the matcher from the control (what to do if the input does not match) have been implemented, together with a language to express the matcher partition as a set of logical rules of arbitrary complexity. Furthermore, different strategies have been experimented, that use classical approximate pattern matching techniques already used in character pattern matching. The work in this direction has shown promising results.

Last, a research has been carried out on a specific subproblem: the recognition of the pitch of a singer’s voice. The available tools in this field are somehow inaccurate and unstable, and this degrades all the score recognition process when it is used in live situations.

6 Conclusion

As performance is the main issue, JMax evolutions naturally targets improvements in this area, featuring extensions of the multi-threading kernel toward multi-processors execution of DSP as well as control. Other extensions focus on language extensions, in particular more explorations of the different computing paradigms already introduced.

It must be noted that at the time of the writing of this article, JMax is distributed under GNU’s General Public License, thus making its development opened to the computer music community.

References


ICMC Proceedings 1999 — 447 —
computation by side effect

Figure 1: A jMax patch demonstrating computation by side effect

additive synthesis by inverse FFT

"FFT minus one"

Figure 2: A jMax patch implementing additive synthesis