LATEST EVOLUTIONS OF THE JMAX REAL-TIME ENGINE: TYPING, SCOPING, THREADING, COMPILING.

Déchelle, Francois, dechelle@ircam.fr
Borghesi, Riccardo, borghesi@ircam.fr
De Cecco, Maurizio, dececco@ircam.fr
Maggi, Enzo, maggi@ircam.fr
Rovan, Butch, rovan@ircam.fr
Schnell, Norbert, schnell@ircam.fr

IRCAM - 1 place Stravinsky - 75004 PARIS - France

Abstract

In this paper we present the latest evolution of the execution engine of jMax, the Ircam environment for real-time musical applications. Already known as FTS, this DSP engine has now evolved into the virtual machine of a language which is a superset of the well-known Max "message system." Details of the new architecture will be presented, as well as issues regarding portability to other platforms.

Introduction

jMax is the latest generation of the programming and runtime environment developed and used at Ircam for real-time audio and signal processing applications. FTS, the DSP server for the system, provides a real-time multi-threaded executive and a message-based object system; details of the architecture have been previously presented at ICMC'95 (Déchelle -1995a). With its modular and configurable structure, FTS is suitable for a wide range of applications, from rapid prototyping and algorithmic composition to embedded sound processing applications.

1 - Architecture

The framework of the jMax environment remains a client/server architecture, wherein the JAVA user interface is a client program connecting to an FTS server via a portable protocol. This flexible architecture, described in (Déchelle et al., 1994; Déchelle & De Cecco, 1995a), makes the development of the two components more independent, and can be used to run the graphical user interface and real-time server on different machines. The real-time engine can also be run without a graphical interface, for example within a plug-in environment.

In the previous version of FTS, using either the NeXTSTEP or the X-Windows version of Max (Déchelle et al. 1994, Déchelle and De Cecco 1995a), a C library--the "client library"--provided a low level access to the client/server protocol and handled communication with the FTS server. As a result of the decision to rewrite the client environment in JAVA, the client library has been rewritten and extended in order to offer higher level services to the application programmers, taking full advantage of the object-oriented features of the implementation language.

The "application layer" is this new version of the client library. This set of JAVA classes provides JAVA applications with services such as server handling, protocol implementation, object instantiation, deletion, connection, message sending and receiving. The application layer also introduces the concept of "properties" (a named data attached to an FTS object) and provides services to set a property of an FTS object, or to watch the changes of a property from the FTS side.

2 - Variables

The integrated handling of named variables constitutes one of the most desired extensions to the language of the Max programming environment. Variables provide a flexible patch customization mechanism, and a naming mechanism for objects with simple but flexible scoping rules.

A variable in jMax is a name bound to a value; variables are object instantiation time entities, i.e., are a mechanism to give patch programmers more flexibility in building up complex patches and patch libraries. Any object can refer to a variable value in its instantiation arguments; for example, int $samplingRate will be an integer whose value is the value of the variable named "samplingRate".
Variables’ values can be all the standard jMax data types. Variables can be created inside a patcher, by specifying them in the patcher box: foo 44100 10 unit = msec. This box defines the patcher foo, with two variables defined positionally ($1, of value 44100, and $2, value 10) and one variable defined by name, unit, of value "mSec".

A variable can also refer to an object; an object bound to a variable is implicitly named by means of that variable, and can be referred by name in the patch. For example, in coefs : table the created table object is set to the value of the variable named "coefs"; other objects can access the table by accessing the value of the variable "coefs".

A variable is visible within the patcher where it has been created, and in all its subpatchers; variables are not visible outside a patcher, but the "." operator in expressions (see below) provides a way to access variable values using a complete name path.

The system avoids any loops in variable references by forbidding the instantiation of an object with a reference to an unbound variable; also, editing operations that cause a variable redefinition will automatically reinstantiate or reinitialize all the objects referring to that variable, so that the patch status is always consistent with the state of the editing session.

3 - Expressions

Expressions are the logical extension of the variable concept in object definition; any object argument can be defined not only by a constant or a variable, but also by any expression made up of constants and variables and a rich number of operators. For example, float (1/$samplingRate * 1000) is a float object whose value is the number of milliseconds between two samples at the current sampling rate.

All the standard C operators are implemented (arithmetic, bitwise, logical); also, the "." operator allows access either to a patcher internal to a variable, if applied to a patcher, or to a constant, instantiation time property of an object. For example, int $coefs.size is an integer whose value is the size of the coefs table (if defined as above), while int $Spat.inputSection.nchans is an int object whose value is the value of the variable nchans, within the subpatcher "inputSection" of the patcher bound to the variable "Spat" in the context where the int is created.

Expressions also include access to indexed variables and the possibility to call either user-defined functions or library functions.

4 - Templates

Expressions and variables provide a flexible way to parameterize a patch: by declaring any saved patcher as a template, the parameterization mechanism becomes a way to add new object primitives to the language. In this case, the template instantiation arguments are simply used as the patcher instantiation arguments; in this way, template arguments are essentially the same thing as patcher variables.

Thanks to variables and expressions, templates substitute for, and largely extend, the original "abstraction" mechanism in Max, giving a more flexible language to build parameterized libraries.

5 - Data types

The most important new extension to the jMax engine is the introduction into the message system of an algebra of data types, a la C"types". Type instances can be used to represent the state of an object, and can be sent as messages.

Data types are built from basic types, using classic type constructors like "structof" and "listof". When a type is defined, a complete family of objects for dealing with the type is generated; the family includes constructor, selector, and variable objects to store the values.

Having flexible composed data types implies using standard techniques like substructure sharing, which implies the need to provide a garbage collector to conveniently handle memory management.

6 - Canonical Objects

Max objects (i.e. language constructs) are usually built around a number of canonical styles, based on the respect of a number of conventions.
These styles correspond to specific semantic patterns in the object definition; by formally defining some of these styles, and by providing a system support for them, we can largely simplify the implementation of new objects for the newly defined data types. Also, as a consequence, the language moves toward a more formally specified semantic, allowing future evolutions such as compilation of control.

A canonical object is thus an object implementing a canonical style, and implemented using system support for it. The simplest canonical object can be defined by generalizing the behavior of a classical object (the "+" operator, for example); each object has a state (in the case of "+" the state is made of the two arguments), a number of inlets corresponding to the components of this state (in the case of "+", two inlets), one outlet, and a transformation function that computes the output from the state (addition, for the "+" object). This transformation function is typically generic (int and float addition for "+") and is triggered by an input value in the left-most inlet, or an explicit "bang" message.

More complex canonical object models can be defined by generalization, adding multiple outlets with multiple transformation functions, and so on. In any case, most complex models can be interpreted as patches built upon the simplest models.

Ultimately, a canonical object is simply a graphic wrapper for a function, as the "+" object is a graphic wrapper to represent and use addition in a graphic Max program. By allowing the definition of generic functions for data types, the system allows the use of existing and new canonical objects with user-defined types.

7 - DSP execution engine

The DSP execution engine, named FTL (Déchelle & De Cecco, 1995a; Déchelle et al., 1996), has moved toward an engine designed for multi-threaded execution of DSP computation. Desktop multi-processor machines are now universally based on cache-coherent memory architecture, and multi-processing programming interfaces are always at the thread level. The evolution of the execution engine from a message-based paradigm, based on the ISPW architecture (Lindemann et al., 1991; Puckette, 1991), to a shared memory architecture was thus a natural progression.

The current--yet still experimental--implementation of multi-threading is based on the Posix threads (Butenhof, 1997). This threading standard is widely accepted and supported by most Unix platforms, and its rich programming interface offers a flexible platform for multi-threaded applications development.

One major drawback of the ISPW multi-processor architecture was the absence of orthogonality between the computational structure of a patch and its partitioning information. As a result of this, ISPW users must rewrite their applications in order to make them run on a different multi-processor architecture.

A possible answer to this problem is the availability of an automatic partitioning algorithm, which can be applied to a non-partitioned patch in order to produce an efficient partitioning without modifying the functional structure, thus allowing users to easily scale up applications for multi-processor machines. Such algorithms are under study for the jMax real-time engine; preliminary experiments made using the Metis graph algorithm library (Karypis & Kumar; 1997), have shown the feasibility of such a partitioning tool.

The FTL vector computation engine provides a symbolic representation of the DSP computation. Before execution, this representation is compiled to a portable bytecode which is a linear list of vector function calls. A significant speed gain can be obtained by generating C code instead of portable bytecode, thus eliminating the load/stores of the intermediate results and taking full advantage of modern optimizing C compilers. An experimental version of C code generation has been implemented and the results of applying this technique to a simple arithmetic patch have shown that in favorable cases a speed gain of 2 can be achieved.

8 - Portability issues and supported platforms

Portability has always been an essential issue of the jMax project. Hardware platforms currently running jMax include Silicon Graphics (both mono and multi-processors), Macintosh running Rhapsody, and PC running Linux. As benchmarks for some of the platforms are constantly changing, the reader is directed to the jMax webpage at: www.ircam.fr/jmax/ for the latest information.

Throughout the project, several guidelines have been key in obtaining a good level of portability of both jMax clients and server. The FTS portability is due to both software quality and software engineering issues (Horton, 1990; Stallman, 1998), as well as to the FTS internal architecture. From the beginning of the project the jMax
development team has used coding standards, and has published a C programming style guide (Déchelle & De Cecco, 1995b).

In order to handle non-portable code, the FTS internal architecture introduces generic programming interfaces that encapsulate hardware and operating system dependencies. For instance, the device abstraction (Déchelle & De Cecco, 1995a) supports a small number of I/O and control operations with an API that is totally hardware independent. Both audio and MIDI I/O are handled by devices, and porting FTS to a new platform is a matter of merely writing devices for these platforms, often using the audio library provided by the underlying operating system. Experience has proven the validity of this approach, with a porting cost to a Unix-like operating system that can be measured in weeks, in most cases less than two.

9 - Evolutions

Future developments in FTS will focus on performance optimization. Since performance is the main issue, the system will naturally move to a fully compiled system. The key to such compilation will be strict typing, which will thus be the other main enhancement of the language. Finally, a more complete JAVA integration will be considered in the near future.

Summary

In this paper we presented the latest generation of FTS, the execution engine of jMax. The new "application layer" set of JAVA classes, the integration of named variables and expressions into the Max language, as well as the addition of templates, data types, and canonical objects, were detailed. Finally, some preliminary findings regarding multi-processor support, automatic partitioning, as well as portability issues were presented, in light of a future focus on performance optimization.

References


