FTM — COMPLEX DATA STRUCTURES FOR MAX

Norbert Schnell *Riccardo Borghesi* *Diemo Schwarz* *Frederic Bevilacqua* *Remy Müller*
IRCAM - Centre Pompidou
Paris – France
Real-Time Applications Team & Performing Arts Technology Research Team

1. INTRODUCTION

This article presents FTM, a shared library and a set of modules extending the Max/MSP environment. It also gives a brief description of additional sets of modules based on FTM. The article particularly addresses the community of researchers and musicians familiar with *Max* [14] or Max-like programming environments such as *Pure Data* [16].

FTM extends the signal and message data flow paradigm of Max permitting the representation and processing of complex data structures such as matrices, sequences or dictionaries as well as tuples, MIDI events or score elements (notes, silences, trills etc.).

The consequent integration of references to complex data structures in the Max/MSP data flow opens new possibilities to the user in terms of powerful and efficient data representations and modularization of applications. FTM is the basis of several sets of modules for Max/MSP specialized on score following, sound analysis/re-synthesis, statistical modeling and data base access. Designed for particular applications in automatic accompaniment, advanced sound processing and gestural analysis, the libraries use a common set of basic FTM data structures. They are perfectly interoperable while smoothly integrating into the modular programming paradigm of the host environment Max/MSP.

Inheriting most of its functionalities and implementation from the former jMax project [4], FTM concentrates on providing a set of optimized services for the handling and processing of data structures related to sound, gesture and music representations in real-time. FTM includes a small and simple C-written object system and graphical Java editors embedded into Max/MSP. FTM is distributed in form of a shared library and a set of external modules under the LGPL open source license.

The acronym FTM is a reference to FTS *Faster Than Sound* [15], the real-time monitor underlying the Max software on the ISPW platform, which became later the sound server of other real-time platform projects at IRCAM. One can imagine FTM standing for *Faster Than Music* or *Funner Than Messages*.

The original motivation for the development of FTM were the need for a flexible score representation related to score following and an efficient representation of matrices and vectors to allow the modular implementation of various analysis/re-synthesis algorithms in a unified framework. Today, modules for both domains have been implemented in form of two sets of external modules based on FTM, *Suivi* and *Gabor*. Further packages are following addressing gesture analysis and data base access (see 3.1).

In order to avoid confusion this article consistently refers to instances of FTM classes as *objects* while using the term *modules* for Max/MSP externals.

2. FTM FEATURES AND SERVICES

The features of FTM can be summarized as follows:

- static and dynamic creation of data structures (*FTM objects*) of predefined classes
- editors and visualization tools
- expression evaluation including functions, method calls, and arithmetic operators
- graphical editors (written in Java) and visualization tools
- import/export of text, standard MIDI, SDIF [20] and the usual sound file formats
- object serialization and persistence

2.1. Data Structures and Operators

FTM allows for static and dynamic instantiation of FTM classes creating FTM objects. Static FTM objects are created in a patcher using a dedicated Max/MSP external module. Dynamic object creation is provided by a new-function within the FTM message box and by other external modules (see 2.1.2). The objects are represented by references, which can be sent within the data-flow between the Max modules as arguments of lists and messages.

FTM strictly separates data objects and operators. Only basic operations on FTM objects are implemented as methods of the FTM classes which can be invoked within the FTM message box or by sending a message to a statically created object. More complex calculations and interactions with objects are implemented as Max/MSP external modules receiving references to FTM objects into their inlets or referencing objects by name as their arguments.

FTM objects can contain (references to) other FTM objects. A simple garbage collector handles transparently the destruction of dynamically created FTM objects referenced by multiple elements of an application.

Static FTM objects in a Max/MSP patcher can be named within a global or local scope and marked as persistent in order to be saved within the patcher file.

2.1.1. Classes and Objects

The following FTM classes are currently provided with documentation:

mat ... matrix of arbitrary values or objects
dict ... dictionary of arbitrary key/value pairs
track ... sequence of time-tagged items
fmat ... two-dimensional matrix of floats
fvec ... reference to a col, row or diag of an *fmat*
expr ... expression
bpf ... break point function
tuple ... immutable array of arbitrary items
scoob ... score object (note, trill, rest, etc.)
midi ... midi event

FTM classes are predefined. They are implemented in C and optimized for real-time performance. The classes themselves are kept as generic as possible providing a maximum of interoperability.

In the current packages based on FTM the matrix and dictionary class are mainly used to organize data. Since they can contain references to other object they easily allow for building up recursive structures such as matrices of matrices or dictionaries of sequences. The *tuple* class gives the possibility of creating lists of lists (i.e. tuples of tuples).

The classes *fmat* and *bpf* are optimized to perform real-time processing of sound and movement capture data (see 3.1).

The generic two-dimensional float matrix *fmat* represents various data such as vectors of sounds samples, spectral data, coefficients and movement capture data. Complex calculations are implemented within specific methods, functions or processing modules requiring two-column matrices as input.

The *track* class implements a generic chronological sequence of arbitrary time-tagged values. Each track is typed and contains a single type of values, which can be primitive (int, float or symbol) or FTM objects. A *track* of *scoob* or *midi* objects is used to represent a musical score. A *track* of *fmat* objects is used to represent an SDIF file. Methods for import and export are provided for import and export of MIDI and SDIF files (see 2.2.2). The support of a complex score format such as MusicXML [2] is planned. A more detailed description of the *track* class and the score editor is given in section 2.2.1.

2.1.2. Modules, Messages, Names and Expressions

FTM is released with a set of external Max/MSP modules providing basic functionalities for the creation and handling of objects and operations related to the provided classes. All operations on FTM objects requiring or providing additional memory, timing or visualization are implemented in form of external Max/MSP modules rather than methods of the FTM classes.

The modules `ftm.object` and `ftm.mess` are two graphical Max external modules, which integrate FTM objects and expressions into the Max/MSP graphical programming interface (i.e. the patcher). Both modules use a parser for expressions built into FTM allowing for the evaluation of arithmetic expressions, functions (mathematical or others), method calls, '$'-prefixed name references and access to elements of data structures using brackets (e.g. `$mymat[2 7]`). The syntax is kept simple and integrates infix expressions using binary operators with prefix notation for functions and method calls.

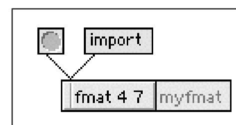

Figure 1. Example of a static FTM object named *myfmat* in a Max/MSP patcher

Figure 1 shows an FTM object statically created with the `ftm.object` module in a Max/MSP patcher. The module defines a *fmat* matrix of floats with 4 rows and 7 columns. The module redirects all incoming messages to the defined FTM object. A bang message causes the module to output a reference to the object from the left outlet. The object can be given a name within global or local scope. Local scope is limited to a patcher file such as a loaded top level patcher or an instance of an abstraction. This way local names can be defined and used within a patcher file and all its sub-patches, while different patcher files and abstractions can have each their private name definitions. FTM names are used with a leading '$' character in all FTM modules including the FTM message box. Dynamic names can be easily created and handled using a *dict* object.

The persistence of the content of a static FTM object as well as its name and scope can be set by graphical interactions with the module or using an associated Max/MSP inspector. A persistent object saves its content within the Max/MSP patcher file and restores its content when it is copied and pasted to a patcher using the serialization mechanism described at the end of 2.1.3. More over, the content of any statically defined object can be saved to and restored from a text file.

The description of an on object consists of a class name followed by instantiation arguments and optional initialization messages separated by commas. The description of persistent objects saving their content within the patcher file is reduced to a minimum description (i.e. in most cases the class name) in order to avoid conflicts between the initial state of the object specified by its description and the content restored from the file when loading a patcher. The same module, `ftm.object`, can be used to give

a name to a number. The description is in this case an expression resulting in a single value.

The arguments for instantiation and initialization can be represented by an expression and thus reference other objects by name. The redefinition of a named object or value will cause a redefinition at all objects using its name in their description. Static object definitions are invalid (i.e. `ftm.object` appear opaque) when they include a reference to an undefined name and turn automatically into valid objects as soon as the name is defined by another `ftm.object` module.

Editing the description of an `ftm.object` can cause the recursive re-instantiation of multiple other modules.

The module `ftm.mess` implements an extended Max message box. The object provides the possibility to compose and output messages in a way which is similar to the usual message box built into Max/MSP. As an extension the FTM message box allows the dynamic evaluation of expressions. Figure 2 shows several examples of expressions in the `ftm.mess` module. Function calls require parenthesis around the function name followed by the arguments. Method calls are similar within parenthesis starting with an object followed by the method name and arguments. Methods can be invoked on objects referenced by name or by references received into the inputs using numbered references (i.e. $1, $2, etc).

In general numbered references are evaluated as values received by the inlets of the message box referring either to the elements of an incoming list (like the Max/MSP message box) or single values received by the inlets. FTM message boxes can have an arbitrary number of inlets. The syntax '$*'allows to use an incoming list or a list of all input values in an expression. An initialization values can be specified for each numbered reference in the inspector of the `ftm.mess` module.

It is planned to allow the definition of functions by expressions and Max patches or abstractions in the future.

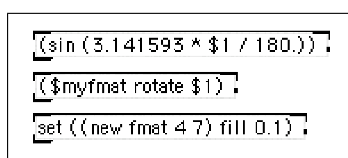

Figure 2. Three examples of messages using expressions in the FTM message box

Since methods of FTM classes have return values the result of a method call can be used in an arithmetic expression or as argument of another method or function call.

FTM expressions of the same syntax as used in the `ftm.object` and `ftm.mess` modules can be represented by objects of the class *expr*. The *expr* class is instantiated with a symbol containing an unparsed expression and has a single method *eval*. The *eval* method accepts a dictionary and a list of values as arguments. The dictionary is used to resolve '$'-prefixed name references in the evaluation of the expression while the list of values is referred to by the numbered references. Names which can not be resolved in the given dictionary are resolved in the global scope (local scope is not defined).

Since expression object generally don't have a scope, names referenced inside the expression are resolved in the global scope and using the dictionary given as argument of the *eval* method. The additional arguments of the method are replacing the numbered '$'-references of the expression.

FTM expressions can not be used within ordinary Max modules such as those provided with FTM apart from `ftm.object` and `ftm.mess`. Nevertheless, FTM modules can use the '$'-syntax to reference objects or numbers by name in their instantiation arguments. Similar to static object definitions FTM modules are automatically re-instantiated when the definition of a name reference changes. Many FTM modules follow the convention for attributes using symbols with a leading '@'-character in order to specify initialization arguments by name.

The following external modules are currently provided with FTM:

- `ftm.object` ... static object definition
- `ftm.mess` ... extended message box
- `ftm.copy` ... copy objects to an internal reference
- `ftm.clone` ... copy objects to new references
- `ftm.value` ... store and output any value
- `ftm.list` ... convert an object to a lists
- `ftm.iter` ... iterate on an object
- `ftm.schedule` ... delay incoming value
- `ftm.print` ... FTM version of `print`
- `ftm.play` ... play a *track*
- `ftm.record` ... record to a *track*
- `ftm.midiparse` ... raw bytes to *midi* objects
- `ftm.midiunparse` ... *midi* objects to raw bytes
- `ftm.buffer` ... buffer~ interface for *fmat* class
- `ftm.vecdisplay` ... *fmat* graphical display

2.1.3. *References, Data-flow and Persistence*

The introduction of references to complex data structures into the Max data-flow creates new possibilities as well as unusual programming paradigms. While Max messages are immutable and copied in order to perform successive calculations module by module following the patches connections, the FTM objects floating in a Max patch are often modified by the modules they traverse.

Figure 3 shows a simplified example of a patch calculating the logarithmic magnitude of an FFT spectrum and a smoothed spectral envelope of a frame of 512 samples of a speech sound. Each of the messages boxes invoke one or two methods performing an in-place calculation which destructively transforms the content of the matrix. Some methods even change its dimensions.

Max and FTM provide four different ways of programming successive method calls: (1) message boxes connected in parallel or (2) in series, (3) recursive expressions

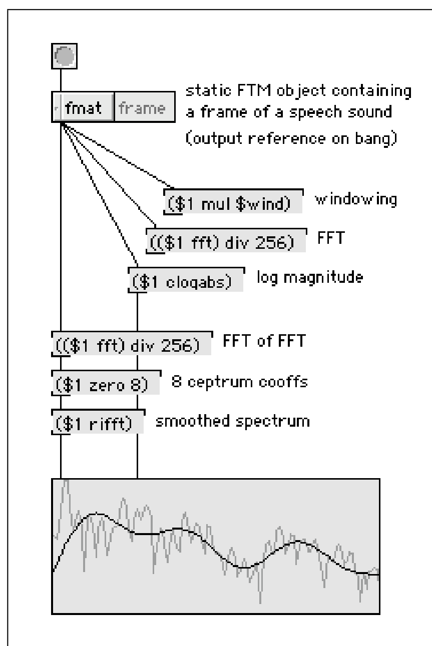

Figure 3. Max data flow with in-place calculations

and (4) comma or semicolon separated expressions in a single message box. The example shows three of them.

The Max control flow will execute the message boxes in right-to-left and top-to-bottom order. Since all of the *fmat* methods used in the example (`mul`, `div`, `clogabs`, `zero` and `rifft`) return a reference to the object itself, method calls can be chained such as shown in the expression '`(($1 fft) div 256)`'. Here an FFT is computed on the incoming *fmat* object before dividing each element of the resulting two-column matrix — representing the complex spectrum — by the scalar 256. Each of the expressions in the message boxes of the example result in a reference to the *fmat* named *frame* defined at the top of the patcher. The references are output by the message boxes and since all calculations in the example are destructive it doesn't make any difference whether one message box is directly connected to the next or two message boxes are connected in parallel. The right-to-left order has to be respected. Since the execution order of two Max modules connected to the same outlet depends on the graphical position of the modules, moving around FTM modules can change drastically the result of the calculation.

In addition `ftm.mess` allows (like the Max message box) to separate expressions by comma causing the successive evaluation and output of the resulting values or lists in left-to-right order. Using a semicolon instead of a comma after an expression suppresses the output while the expression is still evaluated.

Max/MSP patches based on FTM rely more systematically on side-effect and destructive in-place operations than other applications based on the usual set of Max modules. It turns out that the extension to the Max programming paradigm introduced by FTM integrate consistently into the environment. In general the user has to be more conscious of the execution order when working with FTM than when he is using other Max modules. Various techniques are available to avoid side-effect and destructive operations on FTM objects where it is not desired by copying and thus generating a new reference such as a dedicated FTM module called `ftm.copy`.

FTM objects such as a *mat*, *dict* or *track* can contain references to other objects. More over many FTM modules, such as the message box or the module `ftm.play` interpreting a *track* sequence, store references to objects. The destruction of statically or dynamically created objects is handled by a simple reference count garbage collector. Objects which have been referenced by other objects or FTM modules are immediately destroyed when the last reference to the object has been released. When deleting a `ftm.object` statically defining an FTM object which is still referenced by other components of a patch, only the static definition is erased. The object remains until the last reference is released.

For persistency, FTM provides a serialization mechanism recursively saving the content of objects and the objects contained as references. Using this mechanism, FTM objects containing FTM objects can be saved and restored within a Max/MSP patcher file and copied and pasted between patchers. When copying, objects containing references to an object not included in the set of copied objects, the references will be identically restored when pasting. When saving a patcher containing a reference to a statically defined object which is not itself included in the saved patcher (i.e. scope), the reference will be automatically replaced by a reference to a copy of the object. Reloading the patcher the data will be correctly restored but without any relationship to the originally referenced object.

2.2. FTM Interfaces, Interchange and Integration

2.2.1. Graphical Editors

FTM provides editors for most of the complex classes. Similar to other Max/MSP modules an editor is opened when double-clicking on the an `ftm.object` module if an editor is available for the class of the defined object. While some editors such as those for the *track* and *bpf* class allow for a graphical representation of the objects content others consist of a simple textual table view (e.g. *mat* or *dict*).

All editors are using Java [8] and integrate into Max/MSP using the *mxj* Java interface. The `mxj` module has been originally provided to allow for prototyping and developing Max/MSP objects in Java. The FTM library uses a hidden `mxj`-module to embed the FTM graphical user interfaces consistently into the Max/MSP environment. The choice of Java as the programming environment for the graphical interfaces of FTM is inspired by the portability and simplicity of development and the availability of a rich set of graphical components.

The communication between the FTM runtime environment and the Java editors is handled by a centralized service. It is easily imaginable to distribute both components on different machines and redirect their communication over a simple message protocol such as OSC [25].

The currently most developed FTM editor is available for the *track* class when representing a sequence of *scoob* objects. The editor provides a chronometric representation for musical scores as shown in figure 4. The representation integrates score events such as notes, rests and trills, with a temporal structure of bars and additional markers. The same content can be edited in parallel in a table view describing the displayed score objects and markers as a textual list.

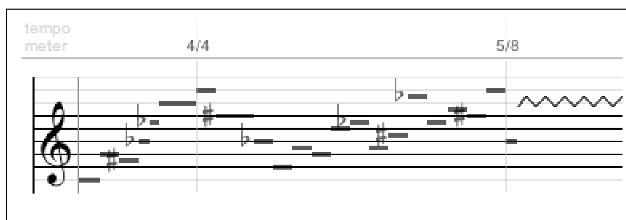

Figure 4. Small detail of a screenshot of the score editor

The *scoob* class was mainly developed for the graphical representation of scores driven by the needs of score following applications. The *scoob* objects have a type (note, interval, rest, trill, etc.) which is associated to a specific graphical representation. The chosen chronometrical score representation is designed as a compromise between the traditional symbolic music representation and usual representations of audio and MIDI data in popular sequencer applications.

2.2.2. SDIF

The Sound Description Interchange Format (SDIF) [20] is a file format of increasing popularity for the storage and exchange of sound data in various representations including frequency domain descriptions such as partials, spectral envelopes, STFT frames, FOF parameters or LPC coefficients but also PCM samples or PSOLA markers. SDIF is also used for motion capture data and other data sets with a temporal development.

SDIF is integrated into FTM in form of import and export methods of the *track* class. A *track* object can represent an SDIF file as a sequence of *fmat* objects, each *fmat* object representing a matrix of the SDIF file. Additional arguments of the import method can specify the selection of single streams, frame types and matrix types of a given SDIF file. Using the SDIF selection syntax [20], also time range and specific matrix columns or rows can be chosen.

FTM uses the IRCAM SDIF library [23].

2.2.3. Max/MSP Integration

FTM can be seen as partly independent from Max/MSP and can easily be integrated into other environments. However special care is taken to assure the seamless integration of FTM into Max/MSP. Especially the graphical modules embedded to the Max/MSP patcher window, `ftm.object` and `ftm.mess`, are to be mentioned in this context, but also other modules such as `ftm.list`, which transforms any FTM object into a Max lists respecting the conventions for lists and messages of Max/MSP.

Most FTM objects treat and store only single values or references to objects. Max messages can be easily transformed to FTM tuples and back to messages in order to store them in FTM objects such as *track* sequences or *dict* tables.

It has been chosen to represent references to FTM objects in the Max data-flow on the level of elementary types such as int, float and symbols. Single FTM objects are sent as a single argument of a special message “ftm.obj“, which is only understood by the FTM external modules. As a consequence some Max/MSP modules such as `pack` don’t apply to FTM objects. In this case an FTM specific replacement is provided.

The module `ftm.buffer` has been added to the FTM object set as an interface between the FTM *fmat* class and the Max/MSP module `buffer~`. The `buffer~` module is mainly used as container of sound samples and Max/MSP provides several other modules to record, play, display and edit the content of the module. In order to create a maximum of interoperability between these modules and the *fmat* class as well as the FTM modules operating on float matrices, `ftm.buffer` gives the possibility to interface an *fmat* object with the set of objects compatible with `buffer~` without creating any overhead of memory use nor computation.

3. FTM APPLICATIONS AND PLATFORMS

3.1. Packages

Several packages dedicated to different domains of application are available for FTM. The very first operational package relying on FTM has been *Suivi* which implements recent score following algorithms formerly available within the jMax environment for Max/MSP. The package *Gabor* followed and lately two other packages, *MnM* and *FDM* joined the family of modules exploring the new possibilities introduced by FTM.

3.1.1. Suivi: Score following

The package *Suivi* contains modules performing score following based on *Hidden Markov Models* (HMM) [12] [13].

The package consists mainly of two objects performing score following on MIDI and audio input. They reference *track* objects containing the score information.

3.1.2. Gabor: "Microsound" and analysis/re-synthesis

The *Gabor* package is a toolbox for analysis/re-synthesis applications [24]. The name of the package is an homage

to Dennis Gabor [1] the inventor of the concept of atomic sound particles [7].

The modules of the *Gabor* package are built around the notion of generalized granular synthesis. They treat atomic units of short sound (samples) [17]. *Gabor* provides a unified framework for granular synthesis [22], PSOLA [11], phase vocoder [5] [10] and other overlap-add techniques for the time-domain as well as the spectral domain [19] operations. Based on the FFT-1 method [3] [18] also additive synthesis is well integrated in this framework.

Gabor provides several modules to transform (i.e. cut) a Max/MSP signal processing sound stream to a stream of overlapping FTM *fmat* vectors within the Max/MSP message scheduler. Each of the modules is oriented towards a different timing-paradigm requiring parameters for the period and size of the generated vectors precisely in milliseconds or samples or automatically synchronizing to the pitch of an incoming monophonic signal. Another basic module of *Gabor* reconstitutes a Max/MSP signal processing sound from FTM *fmat* vectors precisely respecting the timing of the incoming control stream. The module uses the 64-bit floating point time-tags provided as the logical time with each Max/MSP message and optionally performs interpolation in order to provide a timing-precision beyond the sampling period.

The other *Gabor* modules implement specific analysis/synthesis algorithms such as optimized FFT, convolution or FFT-1 additive synthesis as well as auxiliary functions for windowing and waveform generators. Many *Gabor* applications extensively use SDIF formats.

3.1.3. MnM: Mapping and statistical modeling

The package *MnM* [1] is a set modules providing basic linear algebra, mapping and statistical modeling algorithms such as *Principal Component Analysis* (PCA), *Gaussian Mixture Models* (GMM) and *Hidden Markov Models* (HMM).

MnM stands for “Music is not Mapping“. The close integration of motion capture with complex statistical models and sound analysis/re-synthesis within in an environment such as Max/MSP is a promising platform encouraging the development and composition of new artistic applications going far beyond simple mappings.

3.1.4. FDM: Data base access

The most recent package based on FTM introduces data bases and the access to indexed data using FTM classes. *FDM* includes the *SQlite* [9] library to implement generic data base storage and retrieval of FTM *fmat* float matrix objects.

The package anticipates basic modules for envisaged future implementations of concatenative data driven synthesis [21].

[1] Nobel prize in Physics 1971 for his invention and development of the holographic method

3.2. Platforms and Environments

FTM is available for Max/MSP on Mac OS X and Windows. The porting of FTM to *Pure Data* [16] on Linux is in an advanced state.

3.3. Platform Independent API for External Modules

FTM provides an API for the development of FTM modules such as Max/MSP externals independently from a specific real-time environment. The API is currently implemented for Max/MSP and *Pure Data*. It assures the possibility of easily porting the available FTM modules to any Max-like environment. The API supports the declaration of Max/MSPstyle attributes and transparently includes a redefinition mechanism for named references to FTM objects in instantiation arguments as explained in section 2.1.2

3.4. License and releases

FTM is released under the *Lesser GNU Public License* (LGPL)[6]. Recent releases are available from the web page of the IRCAM Real-Time Applications Team [2]. The sources of FTM are available via CVS from the FTM SourceForge project [3].

The packages *Gabor*, *MnM* and *FDM* are released with the FTM distribution for Max/MSP. *Suivi* as well as advanced examples and additional phase vocoder components are available within the IRCAM Forum [4].

4. CONCLUSIONS

FTM provides a consistent set of features integrated to Max/MSP opening new possibilities for the development of interactive music and multi-media applications. FTM successfully absolved a phase of proof-of-concept and is today freely distributed with a set of packages oriented towards different domains forming a coherent ensemble around a kernel of basic FTM modules.

FTM and its libraries have been successfully employed in various concert, dance and theatre performances for score following, voice and sound processing, mapping and gesture recognition.

5. ACKNOWLEDGMENTS

The development of FTM wouldn’t have been possible without the contribution of musical assistants and composers at IRCAM. All in all FTM integrates the work of about ten years of continuous development including different predecessor projects such as Max/FTS and jMax. Especially Francois Dechelle has to be acknowledged for his commitment to these projects.

Special thanks goes to Roland Cahen, Jean-Philippe Lambert, Olivier Pasquet, Romain Kronenberg, Benoit

[2] http://www.ircam.fr/ftm
[3] http://sourceforge.net/projects/ftm
[4] http://forumnet.ircam.fr/

Meudic, Alexis Baskind and Gilbert Nouno who patiently tested early versions of FTM. Additional acknowledgments are owed to innumerable partners of continuous exchanges inside and outside of IRCAM who have contributed to the understanding of sound and music processing woven into FTM.

6. REFERENCES

[1] F. Bevilacqua, R. Muller, and N. Schnell. MnM: a Max/MSP Mapping Toolbox. In *Proceedings of the International Conference on New Interfaces for Musical Expression, NIME*, Vancouver, Canada, 2005.

[2] G. Castan, M. Good, and P. Roland. Extensible Markup Language (XML) for Music Applications: An Introduction. In *The Virtual Score: Representation, Retrieval, Restoration*, pages 95–102, MIT Press, Cambridge, MA, 2001.

[3] P. Depalle and X. Rodet. Synthese Additive par FTT Inverse. Technical report, IRCAM, Paris, 1990.

[4] F. Dechelle et al. jMax: a new JAVA-based editing and control system for real-time musical applications. In *Proceedings of the International Computer Music Conference, ICMC*, Ann Arbor, Michigan, 1998.

[5] J. L. Flanagan and R. M. Golden. Phase Vocoder. *Bell System Technical Journal*, November:1493–1509, 1966.

[6] Inc. Free Software Foundation. GNU Licenses. Web page, 1996. http://www.gnu.org/licenses/.

[7] Dennis Gabor. Acoustical Quanta and the Theory of Hearing. *Nature*, 159(4044):591–594, 1947.

[8] J. Gosling and H. McGilton. The Java Language Environment, A White Paper. *Sun Microsystems Computer Company*, 1995.

[9] D. R. Hipp. SQLite Home Page. Web page, 2000. http://www.sqlite.org/.

[10] J. Laroche and M. Dolson. New Phase Vocoder Technique for Pitch-Shifting, Harmonizing and Other Exotic Effects. In *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, Mohonk, New Paltz, New York, 1999.

[11] N. Schnell and G. Peeters et al. Synthesizing a Choir in Real-time Using Pitch-Synchronous Overlap Add (PSOLA). In *Proceedings of the International Computer Music Conference, ICMC*, Berlin, Germany, 2000.

[12] N. Orio and F. Dechelle. Score Following Using Spectral Analysis and Hidden Markov Models. In *Proceedings of the International Computer Music Conference, ICMC*, Havana, Cuba, 2001.

[13] N. Orio, S. Lemouton, D. Schwarz, and N. Schnell. Score Following: State of the Art and New Developments. In *Proceedings of the International Conference on New Interfaces for Musical Expression, NIME*, Montreal, Canada, 2003.

[14] M. Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, 1991.

[15] M. Puckette. FTS: A Real-time Monitor for Multiprocessor Music Synthesis. *Computer Music Journal*, 15(3):58–67, 1991.

[16] M. Puckette. Pure Data. In *Proceedings of the International Computer Music Conference, ICMC*, pages 269–272, San Francisco, California, 1996.

[17] Curtis Roads. *Microsound*. The MIT Press, Cambridge, Massachusetts, 2002.

[18] X. Rodet and P. Depalle. Spectral Envelopes and Inverse FFT Synthesis. In *Proceedings of the 93rd Convention of the Audio Engineering Society, AES*, New, York, 1992.

[19] D. Schwarz and X. Rodet. Spectral Envelope Estimation and Representation for Sound Analysis-Synthesis. In *Proceedings of the International Computer Music Conference, ICMC*, Beijing, China, 1999.

[20] D. Schwarz and M. Wright. Extensions and Applications of the SDIF Sound Description Interchange Format. In *Proceedings of the International Computer Music Conference, ICMC*, Berlin, Germany, 2000.

[21] Diemo Schwarz. New Developments in Data-Driven Concatenative Sound Synthesis. In *Proceedings of the International Computer Music Conference, ICMC*, Singapore, 2003.

[22] B. Truax. Real-time Granular Synthesis with a Digital Signal Processor. *Computer Music Journal*, 12(2):14–26, 1988.

[23] D. Virolle, D. Schwarz, and X. Rodet. SDIF – Sound Description Interchange Format. Web page, 2002. http://www.ircam.fr/sdif.

[24] M. Wanderley, N. Schnell, and J. B. Rovan. Escher – Modeling and Performing Composed Instruments in Real-time. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, San Diego, California, 1998.

[25] M. Wright and A. Freed. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proceedings of the International Computer Music Conference, ICMC*, Thessaloniki, Greece,, 1997.