MUSCIAN - A MUSIC PROCESSING AND SYNTHESIS SYSTEM

A.Ban1 and J.A.Makowsky
Laboratory for Computer Music Engineering2
Technion - Israel Institute of Technology
Haifa, Israel

Abstract: MUSCIAN is a system for music processing and synthesis intended to work with a digital or analog synthesizer. It is best used for experimentation with sounds and sound effects. The MUSCIAN system is a tool for building interactively high-level effects from low-level music and signal processing operations. In this paper it is a first step toward an object-oriented programming environment for integrated synthesizer/computer systems.

The user of MUSCIAN can define for himself musical data types, transformations and interface architectures which fit his aesthetic predilection in his musical work. It accepts the composition of new sounds from previously built sound samples rather than synthesizing each sound primitive from scratch. It supports modularity and good feedback in alphanumeric, graphic, or acoustic form. MUSCIAN, once minimalized and professionalized, can serve as a waveform device for sound recording and return.

The present prototype of MUSCIAN is known in FORTRAN77 and runs on a PDP11/23/113M or VAX11/750 environment with an attached Minilab analog processor. The array processor is included to give the system almost realistic capabilities.

1. Introduction

Suppose you have built a synthesizer. It is an advanced piece of work, built with state-of-the-art technology. It has 256 channels and can synthesize 1 million samples per second. It has a complex and sophisticated communication protocol, involving matrices as special as envelope generation and time scheduling. And it works. However, if you have not thought about it earlier, it will soon become apparent to you that unless you have a software system at least as beautiful as your synthesizer to control it, you will not find much intrinsic advantage of your machine's design. Or assume you have a ready-made synthesis system complete with its musical keyboard interface and editing functions, and you are experimenting with sounds. You may often find that you are trying to do something, which, although it seems to be within the power of the system, is nevertheless impossible to implement because of the rigidity of your user interface. Thus again, you may be given the tools of designing a musical system, of selecting elements and functions, and combining them in a powerful and useful way. Most often you will rely heavily on experience and published work, depict a prototype, and then make output or minor modifications according to feedback. But if your system is not well-tailored for doing experimentation, such a course may be difficult and onerous.

The above problems may seem confined, but are typical of problems facing computer-music systems users and designers. At bottom the questions involve very basic issues of flexibility, programmability, user interface, the need for I/O tools and the high effort in software development and maintenance.

These questions are not new. Neither are they especially unique to computer music. They prevail in every aspect of software design. Usually the solutions to these questions depends very much on the particular application. The software system was written for und general majesty as Object Oriented Programming [LAD, RCH] or LOGIC Programming [BMER] do not solve these problems yet. The developers of computer music systems have been preoccupied in the past two decades by technical problems, but this activity is now reaching a stage where the more abstract and conceptual problems of software development become crucial and are allowing progress [GABLE, Lehm, LAI].

1 Computer Science, Technion, Haifa
2 Laboratory for Computer Music Engineering, Technical University of Denmark, Lyngby
It is with the intention of finding a way out of such problems, or rather, of acquiring having to run into them, that we have developed a new music processing and synthesis system at the Laboratory of Computer Music Engineering at the Technikon in Haifa. We call our system MUSICIAN [Hall85].

2. Introducing MUSICIAN

As a system MUSICIAN is somewhat difficult to characterize, since it plays a double role as an independent system, and of a music-system generator. As a generator of music systems MUSICIAN offers an interactive programming language for defining functions and procedures, as well as communication protocols which enable MUSICIAN to be integrated in other systems. It is open-ended in both ends of system hierarchy, i.e. it may be used as the computational tool doing the actual synthesis, with the user-interface format prescribed from above, or alternatively, as a high-level driver for external synthesis hardware.

On the other hand, MUSICIAN serves also as an independent musical system, with its own defined user-interface and connection to an audio-DAC. In this role MUSICIAN serves as a flexible tool for development and experimentation. MUSICIAN's alter-ego of a system generator then makes the transition from a developer's system to a production system easy to smooth. The extent of MUSICIAN users may include programmers, music theorists and musicians (the people, not the system...), which all use basically the same vocabulary in dialog with the system.

MUSICIAN's command language is highly mathematical in its base. This basic language is quite general and contains only few musical elements. This generality makes MUSICIAN useful for many non-musical applications, especially to signal processing and mathematical-physical problems. In spite of its generality MUSICIAN deserves the title of a musical system through the ease and efficiency by which musical concepts such as FM synthesis, non-linear distortion and other widely used music synthesis procedures can be implemented. The modularity of MUSICIAN is such that its procedures, once implemented, can be viewed as integral parts of it.

MUSICIAN is based on a number of concepts and features, some of which will be familiar from other systems, such as non-musical. In particular we use

AFL's concepts of an interactive language and workspaces; non-procedural and functional programming style; aspects of concurrent programming (semaphores and data streams);

finally, as array processor is used to speed up computations.

MUSICIAN is intended to run on small to medium computer systems. We have written its first version on a DEC PDP-11/24, with a CPM Mini-1024 array-processing system attached. Even with this rather modest architecture we are able to handle many music processing tasks in real-time. We plan to implement MUSICIAN on a micro-computer system soon.

3. The MUSICIAN System

3.1 Interactive command processing

MUSICIAN is an interactive system which holds a command and response dialogue with its user. The dialogue is held in a simple language, which, in common with many high-level languages, is capable of expressing variables, constants, names, expressions etc.

MUSICIAN operates either in conversational mode or in background mode. In conversational mode the system talks dialogically with a human operator. The operator may define or modify functions and procedures, or execute them on-line. If media equipment is available, sounds may be processed on-line. In background mode, MUSICIAN's still converses with a user, but this "user" is now not human but a (user-written) process which supervises MUSICIAN's activities. In the Background MUSICIAN we see an array of communication devices which enable MUSICIAN to synchronize and share data with its parent process.

3.2 Array Expressions

In MUSICIAN all data is numerical, and all numbers are elements in vectors. MUSICIAN uses most common operators and functions used in ordinary programming languages, redone in some natural way as vector operations.

For example:

A := [1,2,3];
B := [4,5,6];
C := A*B;  //14,30,18;

A := [1,2,3];
B := [1,2,3];
C := A*B;  //1,4,9;
Less trivial is the implementation of the indexing operator (assumed by (j))

\[ A(i,j) \]

which gives the 5th, fourth and first elements of \( A \), in that order, or:

\[ a[0,0,1] \]

which takes from the vector \( a \) the elements whose indices are given to \( \theta \).

3.3. Definition of Functions.

MUSCIAN comes with an extensive list of operators and expressive functions which are "built-in", but these are only given in a starting for the user's definition of his functions.

Defining a function in MUSCIAN is as commonplace as adding 1 and 2. One simply takes an expression and gives it a name:

\[ \operatorname{EXPR:STARTC}(C1,C2) \]

which makes \( \operatorname{STARTC} \) the first ten elements of \( C \). Not necessarily of \( C \)'s current value, but from whatever value may have what \( \operatorname{STARTC} \) is invoked. The function expression is just as ordinary MUSCIAN expression (an array-expression, of course), with the added capacity of including formal parameters, denoted by \( \text{SUBJ} \text{SUBJ} \text{etc.} \). For example is:

\[ \operatorname{DEF AVERAGE}(X) \]

(\[ \text{if } X \text{ defaults to } 2, \text{ the first two arguments of } \text{the } \text{function} \text{ have zero-argument, and its } \text{value is the sum of all arguments divided by the number of arguments. } \text{SUM} \text{ is a built-in function, but it is just as well may have been another user-defined function, such as in} \]

\[ \operatorname{DEF VARIANCE}(\text{AVERAGE}(X),X) \]

which computes the squared variance of its vector arguments. This "stacking" of functions may continue without limit, in principle.

3.4. Non-procedural programming in MUSCIAN.

The combination of \( \text{array-expression} \) with the ability to define new functions make MUSCIAN expressions very concise, elegant and powerful. It should be evident that the power of such expressions may lead to a large degree do away with the need for conventional, procedural programming. Some MUSCIAN operators have been included specifically with this aim in view, and can be viewed as "programming" operators, or rather as programming-replacing operators. These include the various forms of the sequence generator, the indexing operator (\( \text{COUNT} \)), the co-extension operator (\( \text{COUNT} \)) and the count sequence (\( \text{COUNT} \)). The following example shows all of them in action in a function definition:

\[ \operatorname{DEF ROTATE}((A),\text{SUBJ},\text{SUBJ}) \]

which rotates its vector argument circularly one element forward. In particular, loops, which are so common in procedural programming, tend to disappear when array expressions are used, especially when the sequence generation operator is used:

\[ \operatorname{DEF SINEWAVE}(\text{SUBJ}) \]

which generates a specified number of periods of a sine wave, sampled evenly by a specified number of points.

In spite of this, we have not been tempted to depend completely on the use of non-procedural programming (as in pure Lisp), but have included such ordinary constructs as control structures and conventional loops.

3.5 Built-In Functions

We have included in MUSCIAN about two dozens of pre-defined functions, covering such diverse subjects as elementary arithmetic, trigonometry, complex numbers, random number generation as well as more specifically musical or signal-processing functions. These functions are all array functions, i.e., their arguments or result or both are vectors. Most of them are implemented in our prototype using an array processor.

3.6 Workspaces

Like in APL, we call the aggregate of all variables, functions and system settings the "workspace" and allow it to be written or read to or from a storage device. We define the workspace concept very conveniently. Indeed, a MUSCIAN user will not normally write a program in the ordinary sense, but will create a workspace containing functions, variables and procedures dealing with a specific subject.

4. The Background MUSCIAN.

4.1. Working in the background.

The Background MUSCIAN is a key feature of the MUSCIAN system, which enables MUSCIAN to be integrated in a "production" system. This feature enables a complete musical system to take advantage of all of MUSCIAN's capabilities, without putting any serious limitation on the design of synthesizer architectures or user interface.
In the background MUSICAN mode, MUSICIAN works under the supervision of another program running concurrently. In this mode MUSICIAN will usually disappear into the background and let the concurrent process do the talking with the operator in whatever means is judged suitable. At the same time MUSICIAN continues to function in the context of its current workspace, but this activity is transparent to the operator. The two processes are linked together in several ways:

- MUSICIAN continues to process commands in its own usual syntax, but these now originate from its parent process on an internal software queue.
- MUSICIAN shares a common block of memory with its parent, thus allowing the two processes to share data. Specific MUSICIAN commands enable to map MUSICIAN variables on this common block, thus allowing the co-processes to define a common data space.
- A collection of semaphores (or 'sync' flags) are used to synchronize the processes and to select one of the other process' current states.
- MUSICIAN's output stream is made available as its parent (MUSICIAN's output stream is a specific data area which is filled by some MUSICIAN commands, and is usually used to control the system to an audio-DAC).
- The parent task may switch temporarily to conversational mode, thus giving the operator direct access to MUSICIAN. MUSICIAN responds to Background mode when the operator is done.

The Background MUSICIAN is essentially a programmer's option, since a front-program has to be coded. We cite a few possible applications to show its power:

- The front process may act as a controller of a musical keyboard and an array of synthesis function keys, such as in many commercial synthesizers. The front process' task is to translate the effect of these function keys into MUSICIAN commands and functions. These commands will be executed in the context of a work space similar to implement the synthesizer's functions. Notice that the actual effect of the user interface may be modified at any time with no programming involved, by means of conversational mode and making alterations in the work space.
- The Background MUSICIAN may be used to connect MUSICIAN to a synthesizer which implements synthesis functions in hardware. This synthesizer may use a private communication protocol requiring data structures such as data en/queue or waveform tables, time scheduled parameters and so on.

The front task is intended here to translate MUSICIAN's output stream into a format which will conform with the communication protocol. The front task's activities in this example consist of activating its synthesizer preprocessor MUSICIAN, then switching to MUSICIAN-conversational mode. Throughout the rest of the session the synthesizer will be manipulated by the operator using MUSICIAN commands, while the front-task, now itself in the background, is continuously converting MUSICIAN's output stream to synthesizer control commands.

- The above two examples may of course be combined to enable MUSICIAN to work both with a preprocessor, user interface and prescribed audio equipment.

- MUSICIAN's output stream is usually expected to an audio-DAC or some other audio equipment. For cases where real-time synthesis is not possible or if the audio equipment cannot be controlled, a simple foreground task can be written to continuously record the output stream on a storage medium for offline playing.

4.2 Supporting features

MUSICIAN contains several supporting features which make it a practical independent system:

- A graphic display of vectors, operating on a normal alphanumeric terminal. This representation is indispensable when working with large vectors.
- A facility is included for defining and executing command procedures in the MUSICIAN language.
- Special commands and functions enable the system to read data from any data set file, with only elementary restrictions on format.
- A sophisticated keyboard routine enable to edit commands as well as recall and modify previously entered commands.
- Control commands will list information about the contents of the workspace and the status of the system.
MUSICIAN is intended to run on small to medium computer systems. We have written its first version on a DEC PDP-11/44, with a CP/M and BASIC operating system, equipped with this rather modest architecture we are able to handle many music processing tasks in real-time. We plan to implement MUSICIAN on a micro-computer system. Currently, MUSICIAN-II is being written in the language C. It is being developed on a VAX 780, it is intended to run on a INTEL 310 with an X-P 4 user processor. In this configuration we plan to use it as a host for the AMOS synthesizer, designed by Y. Saito and implemented in our laboratory. AMOS was presented at the International Computer Music Conference, Paris 1984 (Saito). MUSICIAN-II was designed in summer 1984 (Nabli) and will be equipped with a composer-friendly interface which allows for modular definitions of visual characteristics in the style of programming in the large and which has an innovative graphic presentation of music.

MUSICIAN-II is also intended to serve as a basis for an Environment Specification System (ESS) which will allow to specify, define and modify musical instruments in a modular way and to use them in real time after compilation. The latter is being developed in our laboratory as a Ph.D. project by S. Makleb (Makleb), and several student projects under the guidance of Dr. U. Gomory and the second author.

To conclude the paper we quote from [Larsen, p. 260]: 'It seems that there are at least two streams of music language development. Those that primarily address the complexity of music representation and analysis are more concerned with efficiency, such as FOF and FORML, and those that stress emphasis efficiency in order to address real-time signal processing issues, and more concerned with high-level representation, such as MuCC and ACME. Ultimately this gap will have to be closed. We think that MUSICIAN and its subsequent development will do exactly this.'