TITLE GOES HERE

By

LUIS F. VIEIRA DAMIANI

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2018

Dedication goes here.

ACKNOWLEDGMENTS

Acknowledgments go here.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS 4

LIST OF TABLES 6

LIST OF FIGURES 7

CHAPTER

ABSTRACT 8

1 LITERATURE REVIEW 9

1.1 Software Synthesis Languages 9
1.2 Real-Time Synthesis Control Languages 13

APPENDIX

BIOGRAPHICAL SKETCH 19

LIST OF TABLES

Table page

1-1 Unit-generator based software synthesis languages [2, 789-790]. 11

1-2 Unit-generator based languages for control of real-time DSP [2, 807-808]. 16

LIST OF FIGURES

Figure page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

TITLE GOES HERE

By

Luis F. Vieira Damiani

August 2018

Chair: Dr. Beverly Sanders
Major: Computer Science

Abstract goes here.

CHAPTER 1
LITERATURE REVIEW

Arguably the first notable attempt to design a programming language with an explicit intent of processing sounds and making music was that of *Music I*, created in 1957 by Max Mathews. The language was indented to run on an IBM 704 computer, located at the IBM headquarters in New York City. The programs created there were recorded on digital magnetic tape, then converted to analog at Bell Labs, where Mathews spent most of his career as an electrical engineer. *Music I* was capable of generating a single waveform, namely a triangle, as well as assigning duration, pitch, amplitude, and the same value for decay and release time. *Music II* followed a year later, taking advantage of the much more efficient IBM 7094 to produce up to four independent voices chosen from 16 waveforms. With *Music III*, Mathews introduced in 1960 the concept of a *unit generator*, which consisted of small building blocks of software that allowed composers to make use of the language with a lot less effort and required background. In 1963, *Music IV* introduced the use of macros, which had just been invented, although the programming was still done in assembly language, hence all implementations of the program remained machine-dependant. With the increasing popularity of Fortran, Mathews designed *Music V* with the intent of making it machine-independent, at least in part, since the unit generators' inner loops were still programmed in machine language. The reason for that is the burden these loops imposed on the computer [1, 15-17].

1.1 Software Synthesis Languages

Since Mathews' early work, much progress has been made, and a myriad of new programming languages that support sound processing, as well as domain-specific languages whose sole purpose is to process sounds or musical events, have surfaced. In [2], we see an attempt to classify these languages according to the specific aspect of sound processing they perform best. The first broad category described is that of *software synthesis languages*, which compute samples in non-real-time, and are implemented by use of a text editor with a general purpose computer. The *Music N* familiy of languages consist of software synthesis languages, and

Tab. **??** presents a list of software synthesis languages developed until 1991. A characteristic common to all software synthesis languages is that if a toolkit approach to sound synthesis, whereby using the toolkit is straightforward, however customizing it to fulfill particular needs often require knowledge of the programming language in which the toolkit was implemented. This approach provides great flexibility, but at the expense of a much steeper learning curve. Another aspect of software synthesis languages is that they can support an arbitrary number of voices, and the time complexity of the algorithms used only influences the processing time, not the ability to process sound at all, as we see with real-time implementations. As a result of being non-real-time, software synthesis languages usually lack controls that are gestural in nature. Yet, software synthesis languages are capable of processing sounds with a very fine numerical detail, although this usually translates to more detailed, hence verbose code. Software synthesis languages, or non-real-time features of a more general-purpose language, are sometimes required to realize specific musical ideas and sound-processing applications that are impossible to realize in real time [2, 783-787].

Within the category of software synthesis languages, we can further classify those that are *unit generator languages*. This is exactly the paradigm originally introduced by *Music III*. In them, we usually have a separation between an orchestra section, and a score section, often given by different files and sub-languages. A unit generator is more often than not a built-in feature of the language. Unit generators can generate or transform buffers of audio data, as well as deal with how the language interacts with the hardware, that is, provide sound input, output, or print statements to the console. Even though one can usually define unit generators in terms of the language itself, the common practice is to define them as part of the language implementation itself. Another characteristic of unit generators is that they are designed to take as input arguments the outputs of other unit generators, thus creating a signal flow. This is implemented by keeping data arrays in memory which are shared by more than one UG procedure by reference. The score sub-language usually consists of a series os statements that call the routines defined by the orchestra sub-language in sequential order, often without

Table 1-1. Unit-generator based software synthesis languages [2, 789-790].

Application	Year	Authors	Platform	Language
Music III	1960	M. Mathews	IBM 7090	Assembler
Music IV	1963	M. Mathews J. Miller	IBM 7094	Macro assembler
Music IVB	1965	G. Winham H. Howe	IBM 7094	Macro assembler
Music V	1966	M. Mathews J. Miller	GE 645	Fortran IV
MUS10	1966	J. Chowning D. Poole L. Smith	DEC PDP-10	PDP-10 assembler
MUSIGOL	1966	D. MacInnes W. Wulf P. Davis	Burroughs 5500	Burroughs Algol
Music 4BF	1967	H. Howe G. Winham	IBM 360	Fortran II and BAL assembler
Music 360	1969	B. Vercoe	IBM 360	BAL assembler
Music 7	1969	H. Howe	Xerox XDS Sigma 7	Assembler
TEMPO	1970	J. Clough	IBM 360	BAL assembler
B6700 Music V	1973	B. Leibig	Burroughs 6700	Fortran and Algol
Music 11	1973	B. Vercoe S. Haflich R. Hale H. Howe	DEC PDP-11	Macro-11 assembler
MUSCMP	1978	Tovar	Foonly 2 DEC PDP-10	FAIL assembler
Cmusic	1980	F. R. Moore D. G. Loy	DEC VAX-11	C
Cmix	1984	P. Lansky	DEC PDP-11	C
Music 4C	1985	S. Aurenz J. Beauchamp R. Maher C. Goudeseune	DEC VAX-11	C
Csound	1986	B. Vercoe R. Karstens	DEC VAX-11	C
Music 4C	1988	G. Gerrard	Macintosh	C
Common Lisp Music	1991	W. Schottstaedt	NeXT	Common Lisp

making use of control statements. Another important aspect of the score sub-language is that it defines function lookup tables, which are mainly used to generate waveforms and envelopes. When *Music N* languages became machine-independent, function generating routines remained machine-specific for a period of time, due to performance concerns. On the other hand, the orchestra sub-language is where the signal processing routines are defined. These routines are usually called instruments, and basically consist of new scopes of code where built-in functions are dove-tailed, ultimately to a unit generator that outputs sound or a sound file [2, 787-794].

The compilation process in *Music N* languages consists usually of three passes. The first pass is a pre-processor, which optimizes the score that will be fed into the subsequent passes. The second pass simply sorts all function and instrument statements into chronological order. The third pass then executes each statement in order, either by filling up tables, or by calling the instrument routines defined in the orchestra. The third pass used to be the performance bottleneck in these language implementations, and during the transition between assembly and Fortran implementations, these were the parts that remainded machine-specific. Initially, the output of the third pass consisted of a sound file, but eventually this part of the compilation process was adapted to generate real-time output. At that point, defining specific times for computing function tables became somewhat irrelevant.

In some software synthesis languages, the compiler offers hooks in the first two passes so that users can define their own sound-prcessing subroutines. In amny cases, these extensions to the language were given in an altogether different language. With *Common Lisp Music*, for example, one could define the data structures and control flow in terms of Lisp itself, whereas *MUS10* supported the same features by accepting Algol code. In *Csound*, one can still define control statements in the score using Python. Until *Music IV* and its derivatives, compilation was sample-oriented. As an optimization, *Music V* intoduced the idea of computing samples in blocks, where audio samples maintained their time resolution, but control statements could be computed only once per block. Of course, if the block size is one, than we compute control values for each sample, as in the sample-oriented paradigm. Instead of defining a block size,

however, one defines a control rate, which is simply the sampling rate times the reciprocal of the block size. Hence a control rate that equals the sampling rate would indicate a block size of one. With *Cmusic*, for instance, we specify the block size directly, a notion that is consistent with the current practice of specifying a vector size in real-time implementations. The idea of determining events in the language that could be computed at different rates required some sort of type declaration. In *Csound*, these are given by naming conventions: variables whose names start with the character 'a' are audio-rate variables, 'k' means control rate, and 'i'-variables values are computed only once per statement. *Csound* also utilizes naming conventions to determine scopes, with the character 'g' indicating whether a variable is global [2, 799-802].

1.2 Real-Time Synthesis Control Languages

Some of the very first notable attempts to control the real-time synthesis hardware were made at the *Institut de Recherche et Coordination Acoustique/Musique* in the late seventies. Many of these early attempts made use of programming languages to drive the sound synthesis being carried out by a dedicated DSP. Tab. **??** presents a list of real-time synthesis control languages developed until 1991. At first, most implementations relied on the concept of a *fixed-function* hardware, which required significantly simpler software implementations, as the latter served mostly to control a circuit that had an immutable design and function. An example of such fixed-function implementations would be an early frequency-modulation synthesized, which contained a dedicated DSP for FM-synthesis, and whose software implementation would only go as far as controlling the parameters thereof. Often, the software would control a chain of interconnected dedicated DSP's, which would in turn produce envelopes, filters, and oscillators. The idea of controlling parameters through software, while delegating all signal processing to hardware, soon expanded beyond the control of synthesis parameters, and into the sequencing of musical events, like in the New England Digital Synclavier. Gradually, these commercial products began to offer the possibility of changing how exactly this components were interconnected, what is called a *variable-function*

DSP hardware. Interconnecting these components through software became commonly called *patching*, as an analogy to analog synthesizers. The idea of patching brought more flexibility, but imposed a steeper learning curve to musicians. Eventually, these dedicated DSP's were substituted by general-purpose computers, wherein the entire chain of signal processing would be accomplished via software [2, 802-804].

Commonly in a fixed-function implementation there is some sort of front panel with a small LCD, along with buttons and knobs to manage user input. In the case of a keyboard instrument, there is naturally a keybord to manage this interaction, as well. The purpose of the embedded software is then to communicate user input to an embedded system which contains a microprocessor and does the actual audio signal processing, memory management, and audio input/output. All software is installed in some read-only memory, including the operating system. With the creation of the *Musical Instrument Digital Interface* standard in 1983, which was promptly absorbed my most commercial brands, the issue of controlling sound synthesis hardware transcended the interaction with keys, buttons, and sliders, and became a matter of software programming, as one could easily communicate with dedicated hardware, by means of a serial interface, MIDI messages containing discrete note data, continuous controller messages, discrete program change messages, as well as system-exclusive messages. As a trend, many MIDI libraries were written at the time for general-purpose programming languages such as APL, Basic, C, Pascal, Hypertalk, Forth, and Lisp. In addition, most descendants of the *Music N* family of languages began to also support MIDI messages as a way to control dedicated hardware [2, 804-805].

The implementation of a software application to control variable-function DSP hardware is no mundane task, as it requires knowledge of digital signal processing, in addition to programming in a relatively low level language. Dealing with issues of performance, memory management, let alone the mathematics required to process buffers of audio samples, often imposes an unsrumountable burden to musicians. Many solutions were invented in order to work around this difficulties, including the use of graphic elements and controllers, but

ultimately it was the concept of a unit generator, borrowed from software synthesis languages, that most influenced the creation of higher-level abstractions that were more suitble for musicians. This is notably the case of the *4CED* language, which was developed at IRCAM in 1980, and owed greatly to *Music IV* and *Music V*. The resemblance extended as far as to comprise a separate orchestra sub-language for patching unit generators, a score sub-language, and a third command sub-language for controlling effects in real-time, as well as to link both orchestra and score to external input devices such as buttons and potentiometers. The hardware these languages drove was IRCAM's 4C synthesizer. The result of nearly a decade of research at IRCAM culminated in *Max*, a visual programming language that remains to this day one of the most important real-time tools for musicians. *Max*, which will later be discussed in more detail, eventually transcended its hardware DSP and implemented itself in C the sound-generating routines. But that was not until the 2000's, ten years after it became a commercial software application, independent of IRCAM [2, 805-806].

Example 1.2.1. *[2, 809]* Music 1000 *is a descendant of the* Music N *family of languages that was designed to drive the Digital Music Systems DMX1000 signal processing computer, in which we can clearly observe the unit-generator concept in action:*

```
fnctn func1, 512, fourier, normal, 1, 1000
instr 1
        kscale amp, knob1, 0, 10000
        kscale freq, knob2, 20, 2000
        oscil x8, #func1, amp, freq
        out x8
endin
```

In the code above, a fnctn *statement assigns to variable* func1 *an array of 512 samples using a* fourier *series of exactly one harmonically-related sine, whose (trivial) sum is* normal-*ized. The amplitude of 1000 is then meaningless, but a required argument. In fact,* func1 *takes a variable number of arguments, where for each harmonic partial, the user specifies a relative amplitude.*

Table 1-2. Unit-generator based languages for control of real-time DSP [2, 807-808].

Application	Year	Authors	Platform	Language	DSP
4B	1978	D. Bayer	DEC LSI-11	Assembler	IRCAM 4B
SYN4B	1978	P . Prevot N. Rolnick	DEC LSI-11	Assembler	IRCAM 4B
4PLAY	1978	C. Abbott	DEC PDP-11	Pascal	IRCAM 4C
Musbox	1979	G. Loy W. Schottstaedt	DEC PDP-10	Sail	Samson Box
4CED	1980	C. Abbott	DEC PDP-11	C	IRCAM 4C
Music 1000	1980	D. Wallraff	DEC LSI-11	Assembler	DMX-1000
4X	1981	J. Kott	DEC PDP-11	Assembler	IRCAM 4X
FMX	1982	C. Abbott	DEC VAX-11	C	Lucasfilm ASP
Music 400	1982	M. Puckette	DEC PDP-11	C	Analogic AP-400
Cleo	1983	C. Abbott	Sun	C	Lucasfilm ASP
Music 320	1983	T. Hegg	MC68000	Assembler	TI TMS 32010
Music 500	1984	M. Puckette	DEC VAX-11	C	Analogic AP-500
4XY	1986	R. Rowe O. Koechlin	DEC VAX-11	C	IRCAM 4X
Csound	1989	N. Bailey A. Purvis I. Bowler P. Manning	Inmos Transputer	Occam	Inmos Transputer
Music 56000	1989	K. Lent R. Pinkston P. Silsbee	IBM PS/2	Assembler	Motorola DSP56001
NeXT Music and Sound Kits	1989	D. Jaffe L. Boynton J. Smith	NeXT	Objective-C	Motorola DSP56001
Digital Signal Patcher	1990	A. Pellecchia	IBM PC	C	TI TMS320C30
MUSIC30	1991	J. Dashow	IBM PC	Prolog	TI TMS320C30
IRCAM Max	1991	M. Puckette	NeXT	C	Intel i860
IRIS Edit20	1992	P . Andenacci E. Favreau N. Larosa A. Prestigiacomo C. Rosati S. Sapir	Atari SM1000	C	MARS
Unison	1992	J. Bate	Apple Macintosh	C C	Motorola DSP56001

The block that follows defines an instrument, in which the unit generator oscil *takes as aruments the output of three other unit generators, which are respectively the wavetable previously computed, as well as amplitude and frequency parameters, whose values are in turn captured by two knobs attached to the machine. The knobs produce values between 0 and 1, and the subsequent arguments to* kscale *are scaling parameters. Finally,* out *is a unit generator that connects the output of* oscil *to the digital-to-analog converter.*

REFERENCES

[1] C. Roads, M. Mathews, *Computer Music Journal* **4**, 15 (1980). Available from: http://www.jstor.org/stable/3679463.

[2] C. Roads, *The Computer Music Tutorial* (The MIT Press, 1995).

BIOGRAPHICAL SKETCH

Bio goes here.