AMIFAST: An Architecture for MIDI Flows as Sonification Tools

Delfina Malandrino, Pasquale Meo, Giuseppina Palmieri, Vittorio Scarano
Dipartimento di Informatica ed Applicazioni
Università di Salerno, 84081, Baronissi (Salerno) Italy
{delmal,palmieri,vitsca}@dia.unisa.it

Abstract

We describe a framework in Java to create sonification applications with minimum effort from the programmer and musician. Our tool, AMIFAST, offers a set of modules that can be easily assembled to produce sonification of off-line as well as on-line (i.e. real-time) applications. Moreover, the programmer can easily add new functionalities

In AMIFaST, we included a sonification technique that we introduce here, Markov Chain Perturbation.

1. Introduction

Sonification is an important technique to convey information via sounds, in such a way that phenomena are easily recognized. Sonification is usually performed on data that are available before the sonification process starts. In this context, it has proven itself a useful technique in several areas (meteorological data, medical information, output for visually impaired users, etc.).

Only recently, research on sonification focussed on the proposal of generic frameworks to design data analysis tools that were not oriented to specific applications. In this context, we can name MUSART [10] SonART [3] and the Sonification Sandbox [21] .

Our goal is to provide a framework to realize an application that sonifies off-line data (i.e. available before the application is started) as well as on-line processes; in this case the application sonifies a process (of any kind) that produces data when the sonification application is running.

Several are the applications of real-time sonifications. For example, sonification can be useful to monitor on-going processes (such as web servers, printer queues, availability of distributed services, etc.) that makes up a large part of a computer system administrator's workday. In fact, it is well known that sonic information is processed by a different part of the brain than the one used to process visual information. A monitoring system may leverage on multimodal representation to appeal to different "modes" of attention by the managers: when sound varies enough to trigger conscious awareness, attention may be directed to more informative (though completely dedicated) visual representations.

As a first successful usage of sonification for monitoring complex environments, we can refer the reader to ShareMon system [6] that conveys status information of a server in the background using subtle sounds. Another example where real-time sonification was used is the real-time monitoring of Apache web servers [1] , [2] where the sonification was not only an alternative visualization technique but was able to supplement additional information compared to the usual log-analyzers.

Our framework, called "Architecture for MIDI Flows as Sonification Tools" (AMiFaST), is a set of Java modules that can be easily assembled and managed to create sonification applications of off-line data as well as of on-line real-time processes.

In this paper we describe the architecture of AMiFaST and substantiate our claims of generality by providing several examples of applications of our framework. We also present a new sonification technique, Markov Chain Perturbation, that uses algorithmic composition techniques for the sonification. Finally, we describe an experiment on recognition of dual musical characteristics of the music generated by Markov Chain Perturbation in order to provide guidelines on sound mapping.

Previous works

MUSART (MUSical Audio transfer function Realtime Toolkit) [10] is a general purpose sonification toolkit that uses particular functions, called ATF (Audio Transfer Function), for the data to MIDI sound mapping. This tool also provides a graphical interface that allows the end user to specify a file for the input information and, in addition, for specifying information about the ATF configuration and the parameters mapping relatively to the characteristics of the produced sound.

MUSART realizes the off-line sonification by allowing the end user to configure the mapping and, eventually, change it during the execution. For these reasons MUSART is an optimal tool for off-line experiments of sonification, that however does not take into account the on-line monitoring techniques.

SonART (The Sonification Application and Research Toolbox)[3] is an open source framework that allows several sonification techniques, by exhibiting large flexibility in the process of the mapping of sound parameters. The SonART framework consists of three components: the Synthesis Tool Kit (STK), the parameter engine (or synthesizer) and the scheduler. The STK is a collection of C++ classes for creating different synthesis algorithms and audio processing systems. The parameter engine modifies the parameters of such algorithms through a graphical user interface. The scheduler provides a clock for programming such changes in the time. Just like MUSART, SonART was designed for off-line sonification of applications.

The Sonification Sandbox [21] is another example of sonification tool. It is developed in Java and has different characteristics than MUSART. In fact, while MUSART realizes the sonification by adding individual sound elements, Sandbox focuses its attention on the context concept. Sandbox maps data to multiple parameters and adds context to the sonification using a graphical interface in the form of click track, constant or repeating tone. Finally, it is also focussed with off-line data sonification.

Several programming frameworks that offer a certain programmability are present in literature. Nevertheless, their programmability is mainly used to visual combination of modules for the specification of sounds (CSound [20] Pd [15]), or Lisp-like syntax for composition and sound synthesis (Nyquist [14]). Our framework is strongly enhanced by the Java language and the consequent easiness of interfacing with external processes/devices.

Organization of the paper. In the next section we describe our technique of Markov Chain Perturbation, then we briefly illustrate (in Sec. 3) the architecture of the framework and provide some examples (in Sec. 4). In Sec. 5 we report on an experiment on how successful was the sonification technique introduced in Sec. 2. Then we conclude with some final remarks in Sec. 6.

2. Sonification by algorithmic composition

The first result of our paper is a technique for using algorithmic composition as a way to sonify a (real-time) process.

Algorithmic composition represents an automatic process to produce a musical piece. It is an interesting (and sometimes controversial) mature field of research that has leveraged on several algorithmic techniques and models such as cellular automata [11] , [12] , [16] Petri Nets [9] , language grammars [5] , fractals [8] ,[13] , genetic algorithms [4] . While the mixture of sonification with algorithmic composition is certainly not new (see, e.g., [1] , [7] , [18]) our approach is to use algorithmic composition to express information rather than using sonification to produce musical pieces.

The model we choose is, maybe, one of the simplest: Markov chains, popular tool for generating musical structures in the 70s. The goal was to devise a mechanism so that aesthetically pleasant music can be generated algorithmically, in real-time, in such a way that its variations can be useful to represent information. The real-time requirement was crucial in our choice: in fact, Markov chains are both light-weight processes (and can be therefore placed on ordinary desktops) and can produce output in real-time as the information flow enters the chain. The pleasantness and personalization of the generated music in ensured by choosing Markov chains that were obtained by any piece available in MIDI file.

A Markov chain is a stochastic process with the *Markov property*. The process is composed by a sequence $X_1, X_2, X_3, \ldots$ of random variables taking values in a "state space", where the value of X_n is "the state of the system at time *n*". The discrete-time Markov property says that the conditional distribution of the "future": $X_{n+1}, X_{n+2}, X_{n+3}, \ldots$ given the "past", $X_1, \ldots, X_n$, depends on the past only through X_n. Each particular Markov chain may be identified with its matrix of "transition probabilities", often called simply its transition matrix. The entry *i, j* in the transition matrix are given by:

$$p_{ij} = P(X_{n+1} = j \mid X_n = i)$$

that is the probability of moving to state *j* from the state *i*. This kind of Markov chains are said to have *memory 1.* Of course, it is possible to generalize the definition of the transition probability in such a way that memory *m* is achieved, i.e., the state at time *n+1* is influenced by *m* states of the chain.

Our usage of Markov chains as a composer is quite simple and intuitive: each state represents a note and moving (probabilistically) from a state to another is done according to the transition probability. Each time a state is reached the note is played. In order to get a transition matrix that is able to play melodies that are pleasant we choose to analyze the frequencies of notes in well-known pieces. In this way, the Markov chain is going to play something that resembles the piece and, still, looks new. In AMiFaST, of course, the designer can choose any transition matrix to set the behaviour of the Markov chain.

2.1 Sonification by Markov Chain Perturbation

Our usage of Markov chains is based on a technique of perturbations. In fact, we want to identify significant and recognizable characteristics that can be perturbed during the generation. The (classical) dimensions used by our technique are: pitch, loudness, timbre, and rhythm.

All the perturbations occurs on the output of the Markov chain. Therefore, they must be carefully chosen in order to be recognizable and aesthetically pleasant.

About pitch perturbation, several considerations made us choose the augmented 4th interval shift since they are easily recognizable, even to an untrained ear. Moreover, after two augmented 4th interval shifts in the same direction, the melody is played one octave higher/lower.

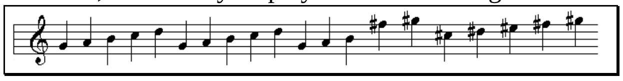


Figure 1: An example of pitch perturbation on G-scale. At the 14th note there is a perturbation.

By using intensity perturbation we can obtain typical figures such as the *crescendo* and the *diminuendo*.

Timbre perturbation is obtained by changing the MIDI instrument that plays the melody. In this case, a well-thought choice of instruments allow an easily recognizable perturbation.

Rhythmic perturbations were the most difficult to choose, since they refer to sequences of several notes. Moreover they technically present some problems in the speed generation of the Markov chain and some buffering problems. We designed a hierarchy of rhythmic patterns: each event can move one step higher/lower in this hierarchy. In AMiFaST the hierarchy can be configured but our choice is presented here. At each step of the hierarchy we use the “doubling” and the syncopation: the first technique replaces each note with two notes such that their duration adds up the original note, while the second moves the rhythmic accents. Experimentally, each of these steps are easily recognized by the users.

In Figure 2 we show the hierarchy that we used in AMiFaST and that was also tested in the experiments.

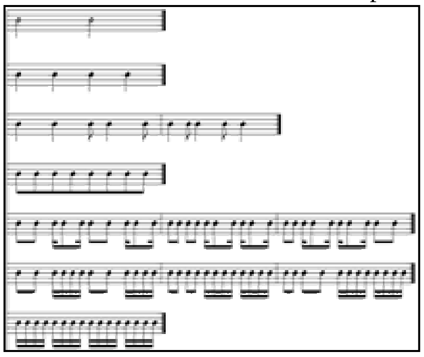


Figure 2: The hierarchy of rhythmic patterns. The first and the last two levels use the doubling while the other levels go from a regular rhythm to a syncopation and vice versa.

3. AMiFaST Architecture

AMiFaST is a programming framework that lies between the MIDI API and the sonification application. The overall objective is to provide the programmer a modular architecture that is configurable and can provide high-level functionalities for sonification of on-line and off-line data.

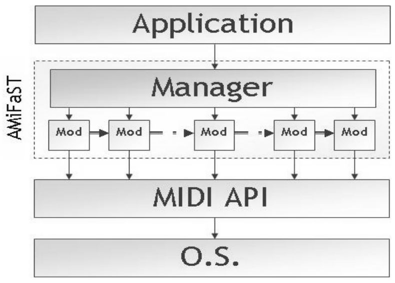


Figure 3: AMiFaST architecture and its layers

AMiFaST is divided into two layers: the Manager and the modules (see Figure 3). The Manager is in charge of instantiating the modules and to invoke services (operations) on the modules. The MIDI flow passes through the modules and each module can access MIDI APIs to perform transformations on such flow. Three categories of modules are available: the Generators, the Transformers and the Players. We describe, now, the two layers and their interaction.

3.1. The Manager

The Manager can be seen as AMiFaST operating system and is completely portable (more comments on AMiFaST portability follows in this section). Any application that is based on AMiFaST begins by instantiating the Manager and passing (as parameters to the constructor) the modules that must be loaded.

The Manager, first, performs some syntactical checks such as, for example, that only the first module has to be a Generator, and that the last module is a Player. Then, the Manager instantiates the modules and creates the communication pipeline between the modules.

The Manager offers the programmer a generic method callMethod() that calls methods of the modules. The first usage of callMethod() is to set the initial values of the modules. In fact, each module has to implement a method initialize().

At the end of the initialization phase, the Manager starts the Generator and waits for invocation of its callMethod() by the application.

3.2. The Modules

AMiFaST modules represent the core of the sonification process. They are managed as Java threads by the Manager. As we said before, there are three categories of modules: Generators, Transformers and Players. They all share some general characteristics and services but their difference is in the role they play in the pipeline.

3.2.1. Generators. A Generator has no input channel, means that it can only be at the beginning of the pipe, since it has an output channel. The generation of the MIDI flow happens in three ways. First, the generation can occur by invoking any scheme of algorithmic composition. Second, it can read a MIDI file and, third, it can receive MIDI data directly from the application via a method feed(). In this last case, we describe the Generator as an EaterGenerator. A Generator can offer methods to change the speed of the production of the flow, which is useful when rhythmic variations occur, later in the flow.

3.2.2. Transformers. These modules can change the MIDI flow that they receive in input. The modification can alter the characteristics of single notes or (short) sequences of notes. The result is then sent on the output channel.

3.2.3. Players. These modules are the counterpart of the Generators at the end of the pipeline and their goal is (in general) to produce the musical representation of the MIDI flow. The main difference is that they also send out a MIDI flow on the output channel. In this way, several players can be present in a pipeline, allowing, for example, to play the MIDI flow (in a player) and save the output in a file (by another player). The Manager transparently instantiates a fake module (called DevNull) and connect it at the end of the pipe so that it can simply consume the flow.

3.3. A brief tour of the available modules

AMiFaST comes with a certain amount of modules ready to be used (as matter of fact, these modules were used to realize the examples described later).

We provide here a very brief tour of the characteristics of the modules, divided in the three categories we have seen. AMiFaST can be working in *scheduled* mode (each note is produced on a timeline and then appropriately played, by using the library NoSuchMIDI [19]) or in *real-time* (notes are played as soon as they are produced). The names of the modules that work in real-time begin with RT*. Moreover, for several modules there is the “multitrack” counterpart, since all of them (with some annotated exception) work with a single track. We describe only the modules whose goal is not obviously given by its name.

3.3.1. Generators. Here is a list of the available Generators: MarkovChainGenerator, EaterGenerator, RTEaterGenerator, MIDIFileReader.

The MarkovChainGenerator produces notes according to a Markov chain as defined in Section 0. The Markov chain is stored in file with filetype .cat (as adjacency lists) and can be created with a command-line utility we provide, that analyzes a MIDI file (given the track that contains the melody and few other parameters). It is relevant to notice that this Generator also includes methods to gauge the speed of the flow of notes that are produced, according to the perturbation in rhythm.

The EaterGenerator is used when the generation of the notes occur directly in the application and, therefore, it only receives a stream of notes from the outside application and forwards it through the pipeline. It can be initialized so that the application is in charge of providing also the rhythm or in such a way that the rhythmic information are generated by the module.

The MIDIFileReader simply reads and forwards in the pipeline a MIDI file whose name is given as a parameter. The modifications on this MIDI file will occur directly by specific players (TrackManagerPlayer, TrackManagerFileWriter) that will turn on/off specific tracks, as instructed by the Manager. This is a technique that has proven itself useful in processes monitoring [1] .

3.3.2. Transformers. Here is a list of the available Transformers: PitchChanger, InstrumentChanger, VelocityChanger, DynamicChanger, PanChanger, RTPitchChanger, RTVelocityChanger, RTPanChanger.

DynamicChanger is able to smoothly move the velocity of the note to a destination, within a given time interval.

3.3.3. Players. Here is a list of the available Players: Scheduler, MonoTrackMidiFileWriter, RTPlayer, TrackManagerPlayer, TrackManagerFileWriter.

Scheduler prepares the notes to be played on-line, by positioning the notes on the timeline.

MonoTrackMidiFileWriter writes in a temporary file the sequence of notes and, at the end, writes everything on a MIDI file.

RTPlayer also takes care of changing the instrument since it acts on an object of the class Synthesizer in JavaSound.

A considerable strength of our framework is that each programmer can easily write his/her own module to support any requirement. To program a module, the programmer must derive one of the Java abstract classes Generator (or SuspendableGenerator), Transformer or Player. Each of these classes implements an interface Module that extends Runnable, since each module is a thread.

3.4. Some comments

3.5.1 Portability. The portability of AMiFaST is only partial. Of course, being written in Java most of its parts are completely portable everywhere a Java Virtual Machine 1.4.2 is available. Nevertheless, by using the library NoSuchMidi [19] we inherently limit the portability to the machines with MS Windows but it poses a limitation only on the portability of real-time applications i.e. that need this library for producing scheduled flows of notes.

3.5.2 Off-line and real-time sonification. It is important to notice that our framework is tailored to provide a tool for off-line and on-line generation of music, thereby allowing uses both on (classical) sonification of a large amount of data and on applications in charge of monitoring a process (that need to produce music while still acquiring data).

4. Applications of AMiFaST

As we said before, AMiFaST goal is to make easy to realize sonification applications of diverse nature. To validate AMiFaST versatility we describe several applications of AMiFaST in different settings (from classical off-line sonification to on-line processes). The examples complexity range from the trivial to the more complicated. In any case, the complexity in assembling the application was mainly concentrated in the musical representation of data/process and not on the technical details of the implementation. For some of the examples, once clarified the technicalities to be connected to the data source, the realization of the application took only a couple of hours. In this way, the designer can focus on parameter mapping and tuning of the application.

4.1. Monitoring the load of a system

This is the first, and the simplest, application that we present: the goal is to visualize by sound the load of a Linux machine. It has a client-server architecture: a server module, named UptimeServer, sends data to the client, UptimeClient. The server runs on the Linux machine and simply runs periodically the uptime command that contains (among other information) the load (number of jobs) in the last minute. While the server collects the data, the client periodically asks the load in the previous minute to the server and, then, produces the sonification by using the Markov chain perturbation technique and rhythm pattern changes.

The effect that is obtained is very intuitive: higher loads are represented by a larger number of notes in the same time interval. In particular, some informal testing reported that the rhythm was the most easily distinguishable pattern to represent the load (compared to pitch and loudness) even by person with no musical knowledge/attitude.

4.2. Mouse movement

This application was designed to obtain an auditory display of the mouse position. It can be useful for helping short-sighted users or users with movement coordination handicap. Because of Java limitations in acquiring information above the JVM, we had to write it as a very simple client-server application.
The client component is a simple C program that monitors the coordinates of the mouse and sends it to the server. The sonification is without scheduling and with a single track. The height of the cursor is mapped to the pitch (from 32 to 122). The right-left position is represented intuitively by a combination of pan and velocity (volume) to (roughly) simulate a 3D space. In fact, the perception is greatly enhanced if the user is given the impression of having the source moving on a semi-circle in front of him, with different heights: the more the source (i.e. the mouse) is on the border, the louder is the sound.

4.3. Sonification of Meteorological data

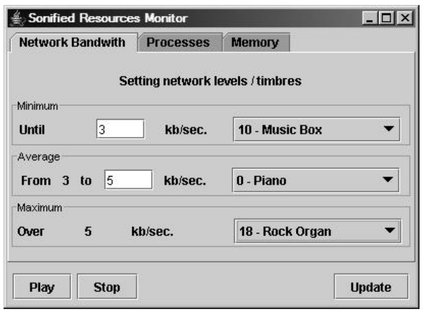
It is a classical example of off-line sonification of a large amount of data. As an example, we used data about monthly temperature and rain that are available in *http://www.ismaa.it/bosco.htm*. We used two tracks: drums and flute. Every month corresponds to a crotchet. The pitch represents temperature: low pitch = low temperature. We used a mapping to notes that are at a distance of a 4th augmented. In this way we made easier the recognition of different notes. Drums represent rain with its rhythm.

4.4. Monitoring the resources of a system

Resources monitoring is an important tool for users of PCs. Usually, graphical representations are provided with operating systems. Unfortunately, they use a very limited resource on current PCs, i.e. screen real-estate and therefore, sonification is useful in this context.

Our application supports monitoring of 3 important parameters: network bandwidth, number of active processes and the total memory that is used.

We had, again, to use a client-server architecture because of Java limitations on acquiring information on the machine. The client (on MS Windows) periodically calls "netstat –e" to get information about the bandwidth and "pslist" (from the pstools package, available at *http://www.sysinternals.com*) to get info about processes and memory.



Setting network levels / timbres

Minimum
Until 3 kb/sec. 10 - Music Box

Average
From 3 to 5 kb/sec. 0 - Piano

Maximum
Over 5 kb/sec. 18 - Rock Organ

Figure 4: The Resource monitoring applications

The sonification technique is the Markov chain perturbation. For network bandwidth we defined three levels, each with an associated instrument. Number of processes is mapped to the rhythmic perturbations while memory occupation is mapped to pitch. All the values and the mapping are completely configurable (Figure 4).

4.5. Monitoring the load of a cluster of PCs

This is the most complex application that is described here. The goal was to monitor a cluster of 5 nodes (connected via a 1.2Gb Myrinet network switch) that was the testbed architecture for a distributed proxy server for Scalable Edge computing Services (SEcS) that was developed and tested in our Lab.

Particularly crucial was to determine how good was the algorithm that balanced the requests load among the nodes. Therefore, we developed and deployed a module (collector) on each node, in charge of measuring the load of a single node of the cluster. Each collector provides the value on request by a client (the sonification application) that is in charge of parameter mapping.

We use again the Markov Chain Perturbation technique. The user is given the possibility to choose the Markov chain that is going to be used and also to save the output on a MIDI file for further playback and checks. Each of the five nodes is associated an instrument, chosen in such a way to be easily distinguishable. In our example, we chose: Piano, Church organ, Brass ensemble, Syn drum, Music box.

When the cluster is well-balanced, the application plays with Syn Strings while in case of underload or overload, it plays the corresponding timbre. In case of several machines out of balance, the timbre associated with the most unbalanced will be used (also more loudly). The difference between under and overload is given by pitch (1 and 1/2 octave lower or higher). The rhythm is used to monitor the overall load of the cluster.

The application was developed during an intensive testbed of a distributed application and the operators (after a brief training session) were able to immediately recognize the behaviour of the cluster under test, by listening to temporary overload of some nodes that were immediately balanced among the other nodes by load balancing strategies

5. Preliminary experiments

The experiments was aimed at proving the efficacy of the Markov Chain Perturbation technique i.e., identifying which characteristics were more effective and recognizable for users. In particular, we were interested in which combination would be more effective in conveying simultaneous information on 2 events. We report here on some preliminary results of the analysis of answers.

The experiment was conducted on 20 volunteers, undergraduate CS students, graduate CS students, and CS researchers. Only 3 participants had formal music training and 5 of them played an instrument. Each participant was given a brief training session to recognize rhythm, pitch, instrument and volume perturbation. Then, they were asked to listen to 6 pre-built MIDI files that mapped two events (called, abstractly, A and B) on each pair of the 4 different characteristics (rhythm, pitch, instrument and volume). To avoid any fatigue, memory and overloading effect, the 6 MIDI files were randomly presented to each participant.

At the end, we asked few questions (3 answers plus "Don't know / don't remember") to extrapolate the following information about events A and B: (I) which was the most frequent event, (II) how many were the occurrences of each event (choice among 3 ranges), (III) the comparison of absolute values of any event at the beginning and at the end of the piece (i.e. "*Was A higher at the beginning than at the end?*").

The results are that: (I) all the characteristics, but the rhythm, had comparable good performances (around 47% successful recognition). (II) the instrument change had very good performances (around 73.3% success) while the others were good (slightly above 50%). (III) rhythm and pitch were around 55% of success, instrument was around 50% while volume was around 46% of success.

6. Conclusions

We propose here a framework to build sonification applications that work also for on-line monitoring. We described several applications that were built with minimum effort and that proved effective. We also introduced a sonification technique based on algorithmic composition and reported on a preliminary analysis of experimental data that showed that different musical characteristics have to be used when different information must be interpreted.

Acknowledgements

The authors gratefully acknowledge stimulating discussions with Alberto Negro and the friendly support by ISIS Lab and its members for the applications of AMiFaST and the experiments (especially grateful to Ugo Erra and Rosario De Chiara).

References

[1] M. Barra, T. Cillo, A. De Santis, T. Matlock, U. Ferraro Petrillo, A. Negro, V. Scarano, P. P. Maglio. "Personal Webmelody: customized sonification of web servers". Proceedings of International Conference on Auditory Display, Helsinki, Finland, 2001

[2] M. Barra, T. Cillo, A. De Santis, U. Ferraro Petrillo, A. Negro, V. Scarano "Multimodal Monitoring of Web Servers". IEEE Multimedia July-September 2002 (Vol. 9, No. 3)

[3] O. Ben-Tal, J. Berger, B. Cook, M. Daniels, G. Scavone. "SONART: The sonification application research toolbox". Proceedings of International Conference on Auditory Display, 2002 http://www.icad.org/websiteV2.0/Conferences/ICAD2002/proceedings/32_bental.pdf

[4] A. R. Burton and T. Vladimirova. "Generation of musical sequencies with genetic techniques". Computer Music Journal, 23(4):59–73, 1999.

[5] W. Buxton, W. Reeves , R. Baeker, and L. Mezei. "The Use of Hierarchy and Instance in a Data Structure for Computer Music". Computer Music Journal 2, 2 (1978), 10 – 20.

[6] Cohen, J. "Monitoring background activities." In G. Kramer (Ed.), Auditory display: Sonification, audification, and auditory interfaces. Reading, MA: Addison-Wesley, 1994.

[7] A. Dorin, "Liquiprisms: generating polyrhythms with cellular automata", Proceedings of International Conference on Auditory Display, 2002 http://www.icad.org/websiteV2.0/Conferences/ICAD2002/proceedings/36_AlanDorin.pdf

[8] R. Greenhouse. URL: http://www-ks.rus.uni-stuttgart.de/people/schulz/ fmusic/wtf/ , 1995.

[9] G. Haus and A. Sametti. "ScoreSynth: A System for the Synthesis of Music Scores Based on Petri Nets and a Music Algebra". In D. Baggi, editor, Computer-Generated Music, pages 53--77. IEEE Computer Society Press, 1992.

[10] A. Joseph, S. K. Lodha. "MUSART: Musical audio transfer real-time toolkit". Proceedings of International Conference on Auditory Display, 2002 http://www.icad.org/websiteV2.0/Conferences/ICAD2002/proceedings/01_AbigailJoseph.pdf

[11] E.R. Miranda (2001). "Composing Music with Computers". Oxford, UK: Focal Press.

[12] E.R. Miranda, (1993). "Cellular Automata Music: An Interdisciplinary Music Project". Interface (now Journal of New Music Research) 22:1, 3-21.

[13] G.L. Nelson. "Real time transformation of musical material with fractal algorithms". http://www.timara.oberlin.edu/people/%7Egnelson/gnelson.htm , 1993.

[14] "Nyquist", http://www-2.cs.cmu.edu/~music/nyquist/

[15] M. Puckette "PD: Pure Data", http://www.crca.ucsd.edu/~msp/Pd_documentation/index.htm

[16] C. Roads. "The Computer Music Tutorial". Chapter 18-19. MIT Press, 1996.

[17] C. Roads. "Grammars as Representations for Music". Computer Music Journal 3, 1 (1979), 45 – 55.

[18] B. L. Sturm, "Sonification of particle systems via De Broglie's hypothesis", Proceedings of International Conference on Auditory Display, 2000. http://www.icad.org/websiteV2.0/Conferences/ICAD2000/PDFs/BobSturmICAD.pdf

[19] T. Thompson. "NoSuch MIDI". Java Library at http://nosuch.com/nosuchmidi/

[20] B. Vercoe. "CSound manual". MIT 1986. http://www.leeds.ac.uk/music/Man/Csound/contents.html

[21] B. N. Walker, J. T: Cothran. "Sonification sandbox: a graphical toolkit for auditory graphs". Proceedings of International Conference on Auditory Display, 2003 http://www.icad.org/websiteV2.0/Conferences/ICAD2003/paper/39%20Walker1-%20sandbox.pdf