USING COMPUTER PROGRAMS AS GENERATORS OF COMPOSITIONS

Aditya P. Mathur*

Department of Computer Science

Purdue University

W. Lafayette, IN 47907, USA

apm@cs.purdue.edu

June 25, 1996

Abstract

An approach to music composition that utilizes a computer program as a generator of music is reported. This approach allows a composer to select a computer program P and develop a mapping of static and dynamic events within the program to musical elements of the desired composition; the musical elements being, for example, theme, pitch, and instrumentation. This mapping is then specified formally using a language called the Listen Specification Language (LSL). A compiler for LSL then compiles program P and the specification into object code. When interpreted the object code generates MIDI data sent to a synthesizer which in turn generates audio. The music so generated depends on the mapping and the input to the program. The run-time system associated with LSL allows the composer to alter various characteristics of the music interactively, i.e. during program interpretation. This approach has so far been used to generate short works based on simple sort programs and compilers. The effect is often pleasing and depends to a great degree on the mastery of the composer in determining a suitable mapping.

1 Introduction

Music composition is a complex task making use of knowledge gathered by the composer from a variety of experiences. Traditionally, it is the human composer who serves as the generator of the composition. The composition itself has a variety of elements. At an abstract level there may be an overall theme. At less abstract levels there are subthemes, instrumentation, melodies, rhythm, and several others. Through a complex derivation process a composer creates and combines these

*This work was supported in part by NSF award CCR-9102331. All reports and programs referenced in this paper may be obtained by writing to the author at the address given or sending a request via email.

elements into a single composition. The basis of the derivation process and the "object" that drives the creation and combining of these elements is essentially a complex mental process.

Mostly out of curiosity, it was decided to investigate the use of computer programs as "objects" that could be used to derive compositions. In this sense of the word "object", a computer program is treated as a generator of composition. A computer program, hereafter referred to as simply "program", is a statement of logic for solving a well specified problem. This statement itself is always made in a language known as a programming language. The program is a static object when written; it becomes a dynamic object when the logic it uses is interpreted. The interpretor's behavior is controlled by the logic of the program. The interpretor is known as a "computer". The behavior of the computer, while interpreting a program P, is commonly referred to as the behavior of the program P itself. This research is about experimenting with ways to map static elements of P and its behavioral patterns to the elements of music. Observations from such experiments might lead to (a) improved understanding of the composition process, (b) new forms of music, (c) new ways of generating music, and (d) other forms of human knowledge.

The remainder of this paper reports the status of this research. Specifically, Section 2 develops an analogy between behavioral elements of programs and those of music. Section 3 provides an overview of a language, named `LSL`, developed to specify a mapping between program behavior and music. How one uses `LSL` and a system that incorporates `LSL` is described in Section 4. A summary of this work and our conclusions so far appear in Section 5.

2 Programs and music

A program is considered to be a collection of functions that operate on data. The functions "call" each other according to a pattern governed by the logic that interconnects them and the data input to the program during its interpretation. A function is considered "active" when it is called by another function and remains active until it terminates. A function that calls another function gets "suspended". Programs may be sequential or parallel. In sequential programs only one function may be active at any instant in time; multiple functions may however be suspended. Parallel programs allow multiple active functions at any time instant.

There are several, possibly infinite, ways of mapping static and dynamic elements of a program to elements of music. Table 1 shows one mapping for a subset of all possible program elements; an explanation of some entries of this table follows. One way to imagine the mapping between programs and music is to consider each function in one to one correspondence with a symphonic "track" or a distinct "instrument". The execution of a function may correspond to the elaboration of a theme. Thus, when a function is called, a theme is initiated and elaborated while the function is active. The theme gets suspended when the corresponding function does. Access to data items is mapped to accented tones or arpeggiations. Evaluations of expressions and conditions is mapped to melodic motives. Assignment of values to variable data items is mapped to start or end of a phrase.

Table 1: A sample mapping of program elements to the elements of music.

Element type	Program element	May be mapped to
Dynamic		
	Function call	Theme initiation or resumption
	Function execution	Theme elaboration
	Suspended function	Theme in background or suspended
	Data access	Accented note(s)
	Array scan	Arpeggiation
	Assignment	Start or end of a phrase
	Expression evaluation	Melodic motive
	Condition evaluation	Melodic motive
	Function call pattern	Rhythm
Static		
	Function name	Pitch or instrument
	Data name	Pitch or instrument

Parts of the mapping in Table 1 are derived from what one might understand to be a "meaning" of a function; the other parts are mostly arbitrary. Though a formal basis for reasoning and specifying such a mapping is presented in the next section, it is this mapping which brings in the element of creativity in using programs as generators of music.

3 Specification of program-music mapping

A formal basis for the program-music mapping, borrowed from previous work [2], is illustrated in Figure 1. To design a language for specifying a variety of mappings, a quantification of two domains is established. Let E be the domain of occurrences during the execution of any program. Function call, function return, expression evaluation, are examples of such occurrences. Let S be the domain of all possible sound patterns that may be associated with each element of E. Note sequences "C4E4G4", D3F$\sharp$4A5" are sample sound patterns that may be articulated in a variety of ways. A mapping from E to S is an association of sound patterns in S to occurrences in E. Such a mapping is specified as a set of pairs (e, s) where $e \in E$ and $s \in S$. The program-music mapping for any program P is a set $\{(e_1, s_1), (e_2, s_2), \ldots, (e_n, s_n)\}$, where each $(e_i, s_i), 1 \leq i \leq n$, is an association of an occurrence to a sound pattern.

Note that the above specification is static. It merely states "what" sounds are to be emitted when P behaves in a certain way; the sounds emitted when P is interpreted depends on the sequence

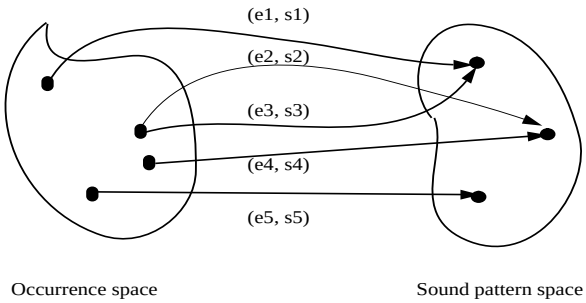

Figure 1: A domain based view of program-music mapping.

```
begin auralspec
specmodule paper-example
begin paper-example

        syncto mmabs q = 120;

        notify rule = prog_begin using Begin_snd;
        notify rule = prog_end using End_snd;

        notify rule = for_body_begin
                using Sticks_snd
                in func = ‘‘bubble_sort’’ and func = ‘‘selection_sort’’;

        dtrack temp
when rule = function_entry:’’exchange’’
                until rule = function_return:’’exchange’’
                using Flute2\_snd;

end paper-example;
end auralspec.
```

Figure 2: Sample program-music mapping for a “selection sort” program.

of occurrences during one interpretation of P. This sequence may differ from one interpretation to another due to variations in input data.

An illustrative program-music mapping is shown in Figure 2. This mapping, reported earlier in [1], specifies how the behavior of a program to sort numbers is to be mapped to sound. The program itself is found in [1]. The music related commands begin from the fourth command in the figure. The `syncto` command specifies that the music is to be played with 120 quarter notes per minute. Each `notify` command specifies the sound to be generated when an event occurs. For example, the first `notify` command specifies that the start of program execution is to be mapped to the `Begin_snd` which is a predefined sound for the Roland SC-55 synthesizer. The `dtrack` command specifies that changes to the value of the data item named `temp` are to be mapped to the `Flute2_snd` which is another predefined sound for the same synthesizer.

Arbitrary patterns of notes are used to specify a sound such as `Flute2_snd` in the above example. A pattern contains a note sequence, instrument on which it is to be played, and how the notes within the pattern are to be played. One may specify arbitrary chordal patterns to be played once when an event occurs or continuously until a `pause` command is issued during interpretation. Multiple continuously played patterns form a theme and may be interrupted or resumed during program interpretation.

4 Using `Listen`

To generate music from a program, a system named `Listen` has been developed. Using `Listen` one may follow the steps given below to map a program P to music.[3]

1. Develop a mapping from P to the elements of music. Create a formal specification S of this mapping in `LSL`.
2. Use `Listen` to compile[1] P and S into an interpretable program P'.
3. Interpret P' on a computer.[2] During program interpretation the events and activities in P are mapped to sound data played through a MIDI device such as the Roland SC-55 synthesizer.

During the interpretation of P' the listener is able to control the mapping specified as S. Thus, for example, a theme may be turned "off" or "on" at will. This allows a listener flexibility to experiment with the composition generated by the program. `Listen` allows the tempo to be set statically through an `LSL` command. The tempo can be altered during program interpretation. Examples of music generated by using the `Listen` system are found in [1, 3].

5 Summary and conclusions

An approach to the generation of music from computer programs is described. A language named `LSL` has been developed to specify the mapping between the elements of a program and music. A system, named `Listen`, that compiles arbitrary C programs and an `LSL` specification, has been developed. Experiments carried out so far have been mostly trivial in nature. Relatively small computer programs have been mapped to music. Music so generated is sometimes exciting enough for the listener to begin dancing; on other occassions it is meaningless and appears to be random noise. A key to the generation of "pleasing" music when using `Listen` is the `LSL` specification; of course what is pleasing to one may not be so for others. By varying the specifications and the data input to the program a wealth of compositions may be obtained.

[1] Compilation of a program is its transformation from a high level language in which it is written, such as C, to a low level language which a computer understands.

[2] The current `Listen` system operates on any Sun Sparc computer under the Solaris operating system. It is available upon request from the author.1

`Listen` 3.0 system was originally developed for application in another area of Computer Science, namely "program debugging." The system has several limitations when considered for program-music mapping. An enhanced version, `Listen` 4.0, is currently under development. Once available the system will allow experimentation with large programs that are perhaps capable of generating larger musical works such as symphonies.

References

[1] D. Boardman, "LISTEN: An Environment for Program Auralization," Master's Thesis, Department of Computer Science, Purdue University, W. Lafayette, IN 47907, August 1994.

[2] D. Boardman and A. P. Mathur, "Preliminary Report on Design Rationale, Syntax, and Semantics of LSL: A Specification Language for Program Auralization," Technical Report, SERC-TR-143-P, August 1993, available from Software Engineering Research Center, Department of Computer Science, Purdue University, W' Lafayette, IN 47907.

[3] D. Boardman, G. Greene, V. Khandelwal, and A. P. Mathur, " LISTEN: A Tool to Investigate the Use of Sound for the Analysis of Program Behavior," *Proceedings of the Nineteenth Annual International Computer Software & Applications Conference (COMPSAC'95)*, IEEE Computer Society Press, August 9-11, 1995, Dallas, Texas, pp 184-193.

Acknowledgements

Thanks to my teachers Verna Abe and Helen Brown who introduced me to the formal elements of music and encouraged my enthusiasm for pursuing it in new directions. Thanks also to Dave Boardman, Vivek Khandelwal, Geoff Greene, Nate Nystrom, and Howard Chen for the endless hours they put into the development of `Listen` 3.0; they have created a system that allows unlimited and exciting experimentation in music composition using computer programs as generators.