Contemporary Music Review,
1996, Vol. 13, Part 2, pp. 63–76
Reprints available directly from the publisher
Photocopying permitted by license only

Extending Music Notation Through Programming

Roger B. Dannenberg

Traditional notation has led to the continuation of a traditional music approach in which scores are static descriptions to be realized by performers. Borrowing programming concepts from computer science leads to score-like descriptions with the capability of expressing dynamic processes and interaction with performers. The implications for composition, performance, and future research are discussed.

KEYWORDS music notation, programming, improvisation, computer composition

1. Introduction

The composition of notated music is a central concept in Western Art Music. Traditional compositions are static in that they have a fixed description, the score. Although no two performances are alike, performances are often considered to be mere renderings of information already present in the score.

Advances in computer science and the ubiquitous computer are changing our perception and mental models of the world. More and more, we tend to think of things and systems as dynamic decision-making processes rather than static forms. Non-deterministic scores and aleatoric music (probably unrelated to computing) marked the beginnings of this transition in Western Art Music, but one could argue that jazz embraced these ideas earlier and took them further.

I conjecture that music notation and the composer/performer tradition is largely responsible for continuing the static nature of musical compositions. I hope to show how computers and digital signal processing offer new possibilities for composers and performers by introducing decision-making, interaction and cognitive processing into what might be called a score. Notation plays a critical role in this transition.

2. Traditional Notation

Traditional music notation has had a deep-rooted effect upon contemporary music. Concepts as basic as beats and scales are firmly rooted in traditional notation, and it is only natural that these concepts form the basis of formal musical training. With such pervasive influence, it is no wonder that music notation exerts a Whorfian restriction and regulation of fundamental concepts in contemporary

music. In other words, the language and notation we use exerts a large influence on what we think and create.

Acceptance of the limitations of music notation only leads to a deeper entrenchment of old concepts and a difficulty imagining what lies beyond. The following sections will show how music notation is being extended in an interesting direction. The point is not so much that there is anything wrong with music notation as it stands, rather that there is rich new territory to be explored by considering notations with fundamentally new properties and semantics.

Traditional musical notation gives rise to models of how music is made and what music is; for example, composition, orchestration, and performance are all part of the traditional model of music-making, and these terms are closely tied to notation. We traditionally consider a "work" to be somehow embodied in the notation. No two performances are alike, and yet each performance is (for good reasons) not considered to create a new work. Only the static notated aspect of the creation is honored with the title "composition". We should expect that fundamentally new notations can give rise to new models or paradigms for making music. In addition to describing new notation, I will show how new notation points toward changes in fundamental concepts of music structure, including the relation and even the meanings of composition and performance.

3. Computer Languages

One of the great triumphs of computer science has been the formal understanding of computation. We not only know how to express computational processes, but we can prove that the notations we use, called programming languages, are "universal" in the sense that they can express *any* computable function. That is, we can show that some languages are as powerful as any language could be. We have even learned that there are some functions that cannot be computed. In contrast, music notation is incapable of expressing more than a very limited form of computation, so there is much to be gained from the languages of computer science.

We should take care to note that terms like "programming language", "express", and "powerful" must be defined carefully for any of this to make sense. Rather than digressing into a long essay on theoretical computer science, let us not worry too much about the details. The important thing is that we are talking about computation and programming languages that express computations.

What does it mean to express computation in a musical work? In general, this means that the music includes a specific procedure to be followed. For practical reasons, the procedure, called an *algorithm* (Knuth 1975) must (1) be finite, that is, an algorithm must terminate in a finite number of steps, (2) be unambiguous, that is, each step must be well defined and have an exact meaning, (3) have zero or more inputs, that is, information may be provided for the algorithm to process, (4) have one or more outputs, that is, information is computed by the algorithm, and (5) be effective, that is, each step of the algorithm must be doable in a finite amount of time using a finite amount of resources.

The concept of computational music or computer programs as scores has, until recently, been an interesting way to augment the capabilities of a composer in

producing a more traditional form of score. In this scenario, the output of the program is a traditional score. Modern computers and computer music systems have introduced a new possibility: that a score might be made to interact with decisions made *during* a performance. In this case, performances may be quite different from one another. For algorithms used in musical performances, the input is usually considered to be a stream of performance information obtained from live players and the output is control information sent to synthesizers to produce sound.

The possibility of interactive or reactive scores is not entirely new, but the precision, speed, and consistency with which procedures can be followed by computers during a performance is unprecedented. A consequence is that a composer can – by careful specification of procedures – maintain control over performances while at the same time giving human performers, conductors, and even listeners an active decision-making role.

Following the premise that notation has a large effect on the music it expresses, it is only logical to investigate notation as a step toward a new kind of computationally oriented music. In the following sections, I will present a notation and a model of computation that seems particularly adept at expressing musical procedures. The notation is called TPL, for Temporal Programming Language, and is intended to promote an interesting set of working concepts. Since the emphasis is on concepts, many practical considerations are ignored here. The author's current research is on practical realizations of TPL concepts.

4. Related Work

The use of formal techniques in composition is hardly a new idea, but the transition from informal theories and methods to completely specified formal procedures is fairly recent. John Cage, in his efforts to leave musical decisions to chance, developed highly formal procedures which were carried out by hand (Cage 1969) and by computer (Austin 1968). Hiller and Isaacson (1958, 1959), Xenakis (1971), and Koenig (1970) were among the first to use computers to perform compositional procedures.

The use of compositional procedures in live performance has been a more recent occurrence. Among the first composers of this genre are Salvatore Martirano, Joel Chadabe (1984), and George Lewis (Roads 1985). With the advent of personal computers and VLSI–based digital synthesizers, there are now hundreds of performers and composers making interactive computer-based compositions. Robert Rowe's book (1993) is an excellent text on the subject.

The Temporal Programming Language (TPL) is derived from the language Arctic (Dannenberg 1984a, Dannenberg & McAvinney 1984, Dannenberg, McAvinney & Rubine 1986, Rubine & Dannenberg 1987) which has roots in early sound synthesis languages such as Music V (Mathews 1969) and also the 4CED system (Abbot 1981). TPL is a declarative language in contrast to the imperative approaches of Formes (Rodet & Cointe 1984) and Formula (Anderson & Kuivila 1986) or the visual programming language approaches of Kyma (Scaletti 1989), Peter Desain (1986), or The Patcher (Puckett 1988). Music Structures (Balaban 1989) offers a notation related to TPL, intended for music analysis rather than real-time performance.

5. Notation and Improvisation

As a composer and improvising performer, I am very much aware of certain strengths and weaknesses of conventional notation. Notation is idea for planning, refining concepts, and working out multiple levels of interrelationships in music. The consistent use of motives, modes, and rhythms requires some sort of planning and therefore a notation to preserve the plan.

On the other hand, conventional notation is very constraining to performers. Concentrating on a score inevitably distracts the performer from other tasks and restricts the set of musical options available to the performer. Of course, great music is made this way, but great music of a different nature is made by improvisers. As a performer, I want music to capitalize on the potential of unconstrained improvisation.

Can we have the advantages of both notation and improvisation? Jazz musicians usually rely on notated (and memorized) chord progressions and melodies to bring a compositional framework to a performance. Composers often insert improvisational or aleatoric passages to relax the usual constraints of composed music. With enough rehearsal, great performers can internalize a composer's instructions to the point that they do not seem constraining or distracting. Computer systems provide a new and interesting way to deal with this question. The fixed framework of most compositions (including traditional jazz performances) is a direct result of the static nature of traditional notation. In order to work out structures in time, time must be represented. Traditional notation leads us to timeline representations with a beginning, an end, and a total ordering of events in between. It is hard for a composer to give meaningful choices to a performer because of what might be called "musical predestiny": the ending is fixed in notation before any improvisation even begins.

Imagine, instead, that a composer could stop time in the middle of a performance. For simplicity, assume that one player improvises and others perform a notated score. As the improviser makes choices, the composer stops time, adapts the composition now in progress to accommodate the choice, and lets time continue. The composer and improviser now operate on a more equal footing, and musical predestiny is no longer a problem. Notice that the composer and therefore the composition, can respond to the performer in any fashion. The improviser can set the musical direction and travel into new territory, or the composer can cleverly deflect improvisation initiatives to support a predetermined idea or goal. The range of possibilities for musical structures and new genres is hard to imagine.

Although it is unlikely that human composers will ever be able to stop time, computer composers can make decisions so quickly that for all practical purposes, we can regard time as having stopped. It might be impossible for humans to play scores that change rapidly, but music synthesizers have no difficulty responding to changes in a few milliseconds. Thus, if we accept some limitations, it is possible for an improviser and composer to interact in real time in a way that releases the improviser from the bonds of a static score and yet retains a great amount of structural planning and control for the composer.

The practical realization of this concept is the interactive computer music composition. Typically, one human performer is monitored by a computer system which in real time responds to the performer by directly controlling music synthesizers. The computer runs a program written by a composer. In essence, the program is an intelligent surrogate for the composer. To the extent that this surrogate can make musical decisions that satisfy the composer, this approach can be effective.

This section has outlined in some detail one of the motivations for interactive scores. It should be clear that notation for expressing interactive decision making and music making is critical to this endeavor. The following section begins a description of one approach to a high-level expressive notation for interactive music compositions.

6. A Simple Music Language

Any music notation must have two aspects: the specification of sound material and the specification of the control structure that organizes those materials. For example, in common practice notation, most symbols denote sound material. Notes, the staff, key signatures, articulation markings, and ties all specify aspects or details of the sound to be produced. Common practice notation also includes a control structure. Some of the control structure is graphical; that is, notes are placed in order from left to right and simultaneous events are aligned vertically. Other control information is explicit: rests, repeat signs, first and second endings, the sign, and codas.

Computation enlarges the possibilities for both sound material and for control structure. Here, we will focus only on the control structure possibilities of a new notation, ignoring almost completely the sound material aspect. In that sense, the notation to be presented is incomplete – what computer scientists would call a "toy language" – in that it is intended to explore language design principles only. By taking this new notation as a kernel, various practical languages can be developed that share a common semantic foundation. (Rubine & Dannenberg 1987, Dannenberg 1989, Dannenberg & Fraley 1989).

In TPL, we will assume that there is already a means for specifying individual sounds or notes. The exact interpretation of this specification is independent of other aspects of the notation, so we are justified in not elaborating on sound specifications. The notation we use will be functional notation; for example, we might specify a note with a pitch of A4 and a duration of one quarter by writing *note*(A4,Q). In many of our examples, we will use letters such as p, q, r, and s to designate sounds. We assume that all sounds have an intrinsic duration; the duration of s is denoted by s_{dur}.

When we combine sounds, the result will always be a new, usually more complex sound. TPL can be used to arrange notes in time as does traditional notation, or TPL can combine simple sounds (not necessarily notes) to form complex sounds. An example is mixing and shaping sinusoids to form a complex set of harmonics in a single tone. In this way, TPL can extend the concept of score into the "interior" of notes without additional linguistic mechanisms.

7. Sequence, Simultaneity, and Timing

All music organizes sound in time, so any notation must address this issue. To specify simultaneous sounds, the following notation is used:

$$[q; r; s; \ldots],$$

where q, r, and s are all sounds, and the semicolons ';' are notational elements that indicate that the sounds are to be simultaneous. The ellipsis indicates that there may be arbitrarily many sounds, and the set of simultaneous sounds is called a *collection*. A collection has an associated duration which is the duration of the longest component:

$$[q; r; s; \ldots]_{dur} = max(q_{dur}, r_{dur}, s_{dur}, \ldots)$$

To indicate a sequence of sounds, the "|" separator is used:

$$[q \mid r \mid s \mid \ldots].$$

In this case, q starts at relative time zero, r starts at q_{dur} (recall that this denotes the duration of q), s starts at $q_{dur}+r_{dur}$, and so on. We call this a *sequence*.

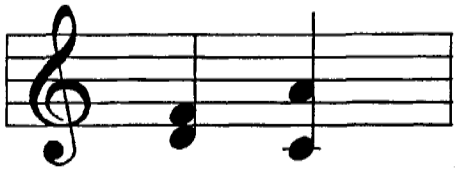

Figure 1.

Consider the example of common practice notation in Figure 1. We might notate this in TPL as follows:

[[*note*(D4, Q) | *note*(C4, Q)] ;
[*note*(F4, Q) | *note*(A4, Q)]]

Here, we have expressed the pitch sequence D4-C4 starting simultaneously with the sequence F4-A4. An alternative expression would be:

[[*note*(D4, Q); *note*(F4, Q)] |
[*note*(C4, Q); *note*(A4, Q)]]

In this case, we have described a sequence of two two-note chords, D4-F4 followed by C4-A4. According to the definition of sequences given above, the second chord starts after a delay equal to the duration of the first chord, which is a collection.

To specify timing in the general sense, the "@" or *shift* operator is introduced. The operator can be applied to any expression in order to shift the starting time of the designated sound by a given amount. For example,

$$[q \mathbin{@} t_q; r \mathbin{@} t_r; \ldots]$$

means to start q at (relative) time t_q and to start r at (relative) time t_r. Another way to describe the *shift* operator is in terms of musical rests. Suppose there is a special

sound function called *rest(d)* that represents a silence or rest of length *d*. Then a shift of a sound by *d* is equivalent to a rest of length *d* followed by the sound:

$$q \mathbin{@} d = [\ rest(d)\ |\ q\ ]$$

8. Abstraction and Parameterization

We now have a simple language for organizing sounds in time. The language is already almost as expressive (if not so convenient) as conventional music notation. An interesting extension is to allow for hierarchical decomposition of sounds. For example, suppose we would like to construct a melodic sequence G-A-E-D and use it many times. We can define a new sound as follows:

define *melody* **is**
[*note*(G, Q) | *note*(A, Q) | *note*(E, Q) | *note*(D, Q)];

Defined terms such as *melody* are called *abstractions*. Wherever an abstraction is used, the term's definition can be substituted. For example:

[*melody* | *melody*] is equivalent to
[[*note*(G, Q) | *note*(A, Q) | *note*(E, Q) | *note*(D, Q)] |
[*note*(G, Q) | *note*(A, Q) | *note*(E, Q) | *note*(D, Q)]];

This example is not too compelling; after all, how many times does one want an exact replication of the same sound? It is far more useful to be able to express an entire class of similar sounds within a single definition. We use *parameters* to control variable quantities within the class of similar sounds. For example, this program shows a melody transposed according to a parameter;

define *mel(root)* **is**
[*note*(*root* + 5, Q) | *note*(*root* + 4, Q) |
note(*root* + 3, Q) | *note*(*root* + 11, Q)];

Here, *root* stands for an arbitrary pitch to be provided when *mel* is used. For example, *mel*(D4) is equivalent to:

[*note*(*D4* + 5, Q) | *note*(*D4* + 4, Q) |
note(*D4* + 3, Q) | *note*(*D4* + 11, Q)];

Notice that D4, called the *actual* parameter is substituted for *root*, called the *formal* parameter, throughout the definition of *mel* to obtain the resulting expression. Since all pitches were written as offsets from the parameter *root*, the resulting phrase is transposed accordingly. Consequently, *mel*(E4) would sound a whole step higher than *mel*(D4). The following example shows a more elaborate use of *mel*:

[*mel*(D4) |
[*mel*(G4) ; [*note*(E4, H) | *mel*(A3)]]]

The resulting sound will contain three instances of *mel* at different transpositions. For example, the first occurrence, *mel*(D4), will produce the sequence G4-F#4-F4-B4. This is because *root* will first have the value D4, and the pitch expression *root* + 5 is G4, *root* + 4 is F#4, and so on.

The behavior *mel* is an *abstraction* because it describes the behavior of a large class of simple melodies that differ only in transposition. More complex abstractions that differ in rhythms, loudness, and even structure can be constructed and designed to generate sounds according to a few parameters.

9. Control Constructs

Increased expressiveness is offered by additional language elements called control constructs. The sequential and simultaneous are examples of control constructs. Another one is the conditional. For example,

if *c* **then** *q* **else** *r*

means to evaluate *c* at the beginning of the sound. If *c* is true then sound *q* is produced; otherwise, sound *r* is produced. The Boolean (truth) value *c* and the sounds *q* and *r* can be arbitrarily complex expressions.

Another control construct is the *iteration* construct, which specifies repetition of a given sound. The expression

repeat *i* **from** 1 **to** 10 **in** *q*

means to repeat sound *q* ten (10) times. The variable *i*, essentially a form of parameter, takes on a new value from the sequence {1, 2, ... 10} on each iteration of *q*. As with the conditional, *q* can be any sound expression and the starting and ending values for *i* (1 and 10 in this case) can be any numerical expression.

A good example to illustrate the use of conditional and iteration constructs is a model of the traditional first and second endings;

```
repeat i from 1 to 2 in
  [ MainPart |
if i = 1 then FirstEnding
         else SecondEnding]
```

where *MainPart*, *FirstEnding*, and *SecondEnding* are all sounds. Each of these can be defined in terms of other sounds using abstraction.

Recursive definitions are also possible in TPL. The following defines a sound of infinite duration:

```
define infinite is
[ note(C4, Q) | infinite ];
```

The sound *infinite* is defined to be the note C4 followed by another instance of the sound *infinite*. This recursion generates an infinite repetition on C4.

10. Termination

Sounds can be stopped before they reach their normal point of completion using the do-until construct:

do *q* **until** *c* **then** *r*

means play sound *q* until *c* becomes true, then play sound *r*. If *q* finishes before *c* becomes true, then the **do-until** construct is complete and *r* is not played. The condition *c* may be an expression to be continuously monitored until it becomes true, or it may be an **event** expression:

event *key*(*i*)

which is true whenever *key* is invoked anywhere in the program. The expression may be used in conjunction with a test of the parameter(s):

event *key*(*i*) **and** $i = 50$

is true whenever an instance of *key* is invoked with the parameter 50.

11. Response to External Input

TPL programs can respond and interact with (musical) external events. The mechanism is to invoke an abstraction (see Section 8) at the time a corresponding event occurs in the external world. For example, assume that a music keyboard is connected to a TPL program during performance. Key #20 is pressed at time 10, key #23 is pressed at time 15, and pedal #1 is depressed at time 16. We define this to be equivalent to evaluating the following expression concurrently with the program:

[*key*(20) @ 10; *key*(23) @ 15; *pedal*(1) @ 16]

In other words, events in the real world correspond to the evaluation of abstractions in TPL.

12. An Example

To put everything together, we will examine a short program in TPL that interacts with a keyboard performance. First, we will define a sequence to be played (with transposition) when certain keys are pressed:

```
define sequence1(p) is [
  note(p, Q) | note(p -1, Q) |
  note(p + 3, E) | note(p + 5, E) |
  [ note(p - 2, Q); note(p - 3, Q) ] |
  note(p - 5, E) | note(p - 6, E) |
  note(p + 5, E) | note(p + 8, E) |
  [ note(p + 4, E); note(p + 2, E) ] ];
```

Another sequence will play a trill:

```
define sequence2(p) is [
  note(p, S) | note(p + 2, S) | sequence2(p) ];
```

Assume that ten (10) additional sequences are defined, but not shown for the sake of brevity.

Now, we will define a connection between these sequences and a piano-like

keyboard that is responsible for generating the events *keydown*(*k*) and *keyup*(*k*) when key *k* is pressed and released, respectively. The function *keyisdown*(*k*) is true if and only if key *k* is pressed.

```
define keydown(k) is [
if k > 12
  then [
if keyisdown(1)
then do sequence1(k) until (keyup(n) and k = n);
if keyisdown(2)
then do sequence2(k) until keyup(n) and k = n);
. . .
if keyisdown(12)
then do sequence12(k) until keyup(n) and k = n);
]
];
```

In this program, keys 1 through 12 act like switches (or perhaps organ stops) that enable the other keys to perform the sequences. If key 1 is down and key 20 is pressed, then *keydown*(20) is evaluated. This expression *keyisdown*(1) will be true, so the expression

do *sequence1*(*k*) **until** *keyup*(*n*) **and** *k* = *n*

will be evaluated. This will in turn start *sequence1*(*20*), 20 being the value of *k*. This sequence will continue until it either finishes on its own or until key 20 is released. In this case the **until** part is evaluated, with *n* = 20.

keyup(20) **and** *k* = 20

Since *k* does equal 20 in this expression, *sequence1* is terminated.

If key #2 is also down when key #20 is pressed, the second **if** expression will also be evaluated and *sequence2* will be started at the same time as *sequence1*. The same applies to sequences 3 through 12. Thus, the keyboard has a fairly elaborate response, and the system as a whole has aspects of a conducting system in addition to an instrument. All of this can be expressed compactly using TPL.

13. Programs and Compositions

With short examples like these, it is impossible to demonstrate the full potential of computational music notations. Just as more conventional scores occupy many pages and take months or years to compose, computer programs are often thousands of lines in length and also take months or years to write[1]. Now that we have illustrated some principles with examples, let us consider some of the ways that programs might be used for composition.

One possibility is to extend the notion of musical instrument by using programs to determine the relationship between a performer's gestures and the resulting

[1] Large software systems require over a million lines of program text and hundreds of man-years to develop.

sound. The first example shows how piano keys might be used to trigger melodic sequences. Like the prepared piano, programmed instruments allow essentially new instruments to be developed by composers to achieve specific effects that would be impossible otherwise. Hyperinstruments (Machover & Chung 1989) are a good example of this approach.

A second possibility is to incorporate memory. Memory allows the output of programs to depend upon earlier input, and various transformations of recorded performance information are possible. Recorded information can also be combined with new input, for example repeating a previous pitch sequence according to a new rhythm.

Another use of memory is to store conventional scores that are synchronized with live performers. Max Mathews (Mathews & Abbot 1980) and Morton Subotnick have explored approaches to "conducting" the playback of scores stored in computer memory. Dannenberg (Dannenberg 1984b, Bloch & Dannenberg 1985, Dannenberg & Mukaino 1988) and Vercoe (Vercoe & Puckett 1985) have built systems that follow a soloist in a score and synchronize the automated performance of a stored score.

These examples are in one way or another based on existing models: the instrument, the tape recorder, serialism, the conductor, the performer. As the sophistication of these approaches increases, the programmed behavior becomes more sophisticated and more complex. At some point, programs reach a degree of complexity described in Section 5, where we can say that the programs are "composing" or "improvising".

There are key differences between human composition/improvisation and the sort that is programmed for computers. First, any moderately sized program will have a fairly restricted musical vocabulary. This will tend to make its output more stylistically consistent from one performance to the next. Secondly, the computer-generated music can respond to other performers almost instantly. This is also true of good jazz performers, but only to a limited extent: an intricate musical response or musical interplay takes time to develop; hence, the more elaborate the response, the less immediate it will be.

The result of this computer-based approach is a hybrid of composition and improvisation, where composers and performers share in the creative process and where the "composition" is the live interaction between the performer and machine. Experience has shown that there is a wide range of choices available to the composer in terms of how much of the structure is predetermined. The style of dynamic interaction among computers, performers, conductors, and audience is open to exploration.

14. Future Research

An interesting question is "at what level do musicians and machines communicate?" The example TPL programs show how complex interactions can take place without any deep understanding or intelligence on the part of the machine. This might be called the "performance gesture" level of interaction. Input gestures and sequences are processed, and output gestures are obtained. One of the goals of recent research has been to raise the level of musical interaction between musicians and machines above the level of gestures.

Music Understanding refers to the recognition of structure and pattern in music by computer, and at least fairly low-level music understanding capabilities have been achieved already. For example, computer systems that can follow a performer in a score (Bloch & Dannenberg 1985, Vercoe & Puckett 1985) or follow a conductor's baton (Mathews & Abbot 1980) have been developed. An experimental system that listens to a blues improvisation to determine the tempo and identify chord changes has been described (Dannenberg & Mont-Reynaud 1987). Music understanding will allow computer systems to perceive performers' tempo changes, phrasing, themes, and harmonic directions (Rowe 1993). This in turn will give composer/performers a new vocabulary for creating interactive compositions.

Another direction for research is toward integration and control of sound parameters at many levels. It is now possible to vary dynamics, articulation, spectral content, spatial distribution, reverberation and other effects with both synthesized and live audio. Computer systems can control these parameters in real-time and therefore composers can use all of these parameters to their musical advantage.

15. Conclusion

We began with a look at notation as a force that ties contemporary music to tradition. Accepting the hypothesis that a more expressive notation might lead to new musical territory, we examined a notation (TPL) based on computer languages. TPL can express traditional static musical structures as well as dynamic structures.

Equipped with new notation and the concept of interactive programs as musical scores, we can begin to think about the implications for contemporary music. A hybrid between composition and improvisation emerges as one possibility, offering an intriguing blend of tradition and technology.

Technology has many shortcomings that require research, including problems of music understanding and the rapid obsolescence of computers and sound processors with each new electronic generation. Purely musical research is also needed. The roles of performers, composers, conductors, and listeners must be reexamined. Even the role of live music changes to become a creative rather than a recreative process. There is much to explore.

16. Acknowledgments

I wish to thank Georges Bloch, Xavier Chabot, and George Lewis for sharing their musical insights and experience regarding interactive computer music systems. The School of Computer Science, the Music Department, and the Studio for Creative Inquiry (formerly the Center for Art and Technology) at Carnegie Mellon and Yamaha have generously supported much of the development of these ideas.

References

Abbot, C. (1981) The 4CED Program. *Computer Music Journal* 5(1):13–33 (spring).

Anderson, D.P. and Kuivila, R. (1986) Accurately Timed Generation of Discrete Musical Events. *Computer Music Journal* 10(3):48–56 (fall).

Austin, L. (1968) An Interview with John Cage and Lejaren Hiller. *Source: Music of the Avant-Garde*, Issue 4, 2(2):11–19. Reprinted in *Computer Music Journal*, 16(4):15–29 (Winter), 1992.

Balaban, M. (1989) *Music Structures: A Temporal-Hierarchical Representation For Music.* Ben Gurion University Department of Mathematics and Computer Science Technical Report FC-TR-021 MCS-313.

Bloch, J.J. and Dannenberg, R.B. (1985) Real-Time Computer Accompaniment of Keyboard Performances. In *Proceedings of the 1985 International Computer Music Conference*, pp. 279–290. Computer Music Association.

Cage, John. (1969) *Silence.* Cambridge, MA: MIT Press.

Chadabe, J. (1984) Interactive Composing: An Overview. *Computer Music Journal* 8(1):22–27.

Dannenberg, R.B. (1984a) Arctic: A Functional Language for Real-Time Control. In *1984 ACM Symposium on LISP and Functional Programming* pp. 96–103. Association for Computing Machinery.

Dannenberg, R.B. (1984b) An On-Line Algorithm for Real-Time Accompaniment. In *Proceedings of the 1984 International Computer Music Conference* pp. 193–198. Computer Music Association.

Dannenberg, R.B., McAvinney, P. (1984) A Functional Approach to Real-Time Control. In *Proceedings of the 1984 International Computer Music Conference* pp. 5–15. Computer Music Association.

Dannenberg, R.B., McAvinney, P., Rubine, D. (1986) Arctic: A Functional Language for Real-Time Systems. *Computer Music Journal* 10(4):67–78 (Winter).

Dannenberg, R.B. and Mont-Reynaud, B. (1987) Following an Improvisation in Real Time. In *Proceedings of the 1987 International Computer Music Conference* pp. 241–248. Computer Music Association.

Dannenberg, R.B. and Mukaino, H. (1988) New Techniques for Enhanced Quality of Computer Accompaniment. In *Proceedings of the 1988 International Computer Music Conference*, pp. 243–249. Computer Music Association.

Dannenberg, R.B. (1989) The Canon Score Language. *Computer Music Journal* 13(1):47–56 (spring).

Dannenberg, R.B., and Fraley, C.L. (1989) Fugue: Composition and Sound Synthesis With Lazy Evaluation and Behavioral Abstraction. In *Proceedings of the 1989 International Computer Music Conference*, pp. 76–79. Computer Music Association.

Desain, P. (1986) Graphical Programming in Computer Music, a Proposal. In *Proceedings of the 1986 International Computer Music Conference*, pp. 161–166. Computer Music Association.

Hiller, L. and Isaacson, L. (1958) Musical composition with a high-speed digital computer. *Journal of the Audio Engineering Society* 6(3):154–160 (July).

Hiller, L. and Isaacson, L. (1959) *Experimental Music: Composition with an Electronic Computer.* New York: McGraw Hill.

Knuth, D. (1975) *The Art of Computer Programming.* Addison Wesley.

Koenig, G.M. (1970) Project 1. In *Electronic Music Reports*, pp. 32–44. Institute of Sonology, Utrecht University.

Machover, T. and Chung, J. (1989) Hyperinstruments: Musically Intelligent and Interactive Performance and Creativity Systems. In *Proceedings of the 1989 International Computer Music Conference*, pp. 186–190. Computer Music Association.

Mathews, M.V. (1969) *The Technology of Computer Music.* Boston: MIT Press.

Mathews, M.V. and Abbot, C. (1980) The Sequential Drum. *Computer Music Journal* 4(4):45–59 (winter).

Puckette, M. (1988) The Patcher. In *Proceedings of the 1988 International Computer Music Conference*, pp. 420–429. Computer Music Association.

Roads, Curtis. (1985) Improvisation With George Lewis. *The Computer Music and Digital Audio Series. Composers and the Computer.* Los Altos: William Kaufmann, pp. 74–87 (Chapter 5).

Rodet, X. and Cointe, P. (1984) FORMES: Composition and Scheduling of Processes. *Computer Music Journal* 8(3):32–50(fall).

Rowe, R. (1993) *Interactive Music Systems.* Cambridge: MIT Press.

Rubine, D. and Dannenberg, R.B. (1987) Arctic Programmer's Manual and Tutorial. *Carnegie Mellon University Technical Report* CMU–CS–87–110.

Scaletti, Carla. (1989) The Kyma/Platypus Computer Music Workstation. *Computer Music Journal* 13(2):23–38 (summer).

Vercoe, B. and Puckette, M. (1985) Synthetic Rehearsal: Training the Synthetic Performer. In *Proceedings of the 1985 International Computer Music Conference*, pp. 275–278. Computer Music Association.

Xenakis, I. (1971) *Formalized Music*. Bloomington: Indiana University Press.