New Techniques for Enhanced Quality of Computer Accompaniment

Roger B. Dannenberg

Computer Science Department and
Center for Art and Technology
Carnegie Mellon University
Pittsburgh, PA 15213 USA

Hirofumi Mukaino

Yamaha Music Technologies, USA Inc.
Wood Island Building, Suite 2B
80 East Sir Francis Drake Blvd.
Larkspur Landing, CA 94939 USA

Abstract

A computer accompaniment system has been extended with several techniques to solve problems associated with reliably recognizing and following a real-time performance. One technique is the use of an additional matcher to consider alternatives when the performance could be interpreted in two ways. Another is to delay corrective action when data from the matcher is suspect. Methods for handling grace notes, trills, and glissandi are also presented. The implementation of these and other techniques to produce an enhanced computer accompaniment system is discussed.

1. Introduction

The basic technology for real-time computer accompaniment[1] of live musicians was first presented in 1984 [1,2], however the first systems left a lot of room for improvement. We have been developing new computer accompaniment systems for two years in an effort to obtain higher reliability in the accompanist and also to deal with a more complete musical vocabulary. Like the first systems, the task is to accept a score describing music to be performed by a human and machine. The machine listens to the human performance and synchronizes to it in real time.

[1]Computer accompaniment is the subject of a U.S. patent.

A variety of techniques have been developed to increase accompaniment reliability. An important improvement has been the use of a limited amount of nondeterminism or parallelism to allow the system to follow two competing hypotheses until one is seen to be superior. A second technique is to delay output from the matcher in order to create a short time window during which decisions can be reversed. Even traditional scores are not completely specified and we have incorporated several extensions to our matcher to deal with grace notes, trills, and glissandi. We indicate the ornamentation in the score so that the system can have an expectation before the ornament is performed. The input is processed according to expectations.

In this paper, we will describe the structure and implementation details of an advanced polyphonic computer accompaniment system. To begin with, in Section 2., we will show the reader the basic structure of this accompaniment system and introduce some terminology. In Section 3., we describe how we implement non-deterministic matching. Then in Section 4., we show how ornamentations such as trills or glissandi are handled. In Section 5., we explain the idea of delayed decision making. Finally, in Section 6., we will describe miscellaneous improvements we have done.

2. Basic Structure

In this section, we present the basic structure of our accompaniment system. We will describe only the funda-

mental ideas of accompaniment systems here. For more details, see the previous papers [1,3].

For convenience, we will first define some terms. Before starting a performance, a score is read into the system. The score has two parts, the *solo score* which is to be played by the human and the *accompaniment score* which is to be played by the system. To distinguish what the composer wrote (the solo score) from what is actually played, we will call the latter the *solo performance* or simply *performance*. Time in the score is called *virtual time* to acknowledge the difference between the notated time and actual or real time. The score consists of a set of *events* in time. In our work, the solo score consists of note-on events corresponding to pressing a key on a keyboard or detecting a pitch from an acoustic instrument. A *compound event* is a group of events, which are played at the nearly same time. Since our events are note beginnings, a chord is the most common compound event.

Our accompaniment system has three important parts.

Preprocessor This module processes input from MIDI-IN and makes compound events by using timing information. If this preprocessor gets two or more events within a short time period, it will put them together and make one compound event. Also, it detects and processes trills and glissandi.

Matcher The matcher compares the performance to a stored solo score. The Matcher reports correspondences between the performance and the solo score to the Accompanist part. To finish the matching computation within reasonable amount of time, it uses only a small portion of score at any given time. We call this section of the score a *window*.

Accompanist The accompanist module plays the accompaniment part given in the score. It changes its position and tempo in real time based on information from the matcher.

Because we use MIDI for input and output, commercial keyboard controllers and pitch detectors can be used for input, and synthesizers can be used for output. Although necessary for performance, we do not consider these to be part of the system.

3. Multiple Matchers

Motivation. The matcher considers only a subset of the score, called the window, at any given moment. If the solo player is not playing inside of the window, the matcher will not be able to track the solo player. This problem can be reduced by making the window larger, but since computation is proportional to the window size, the problem cannot be eliminated. One particular problem arises when the soloist temporarily stops playing. The accompaniment continues independently until the soloist reenters. Where should we expect the soloist to start playing? Reasonable guesses are that the soloist will either reenter where he stopped or he will reenter in synchrony with the accompaniment. Unfortunately, it may not be possible for the window to span both of these locations. Our solution is to allow multiple matchers strategically placed at places likely to match.

Matchers as Objects. We first conceived of the matcher as an algorithm which finds a match between the performance and the stored score. We have changed our point of view and implementation from procedure to object so that we can create as many matcher instances as we want. By that change, we can create a new matcher whenever we need it, and we can dispose of it whenever we do not need it.

Creating Matchers. The matcher is a time consuming module, so we do not want to have many matchers running all the time. Only when we are particularly uncertain about the soloist's position will we invoke another matcher and center its window on an alternate location. We will create a new matcher when we have more than one reasonable guess about the position of the soloist. Typically, this happens when the solo player stops his performance or when he plays many extra notes. In our system, we check the "virtual time" of the accompaniment and the "virtual time" range of the matcher's window. If current "virtual time" is not in that range, we will create a new matcher at the position whose "virtual time" is equal to the current accompaniment virtual time (see Figure 1).

Terminating a matcher. When one of the matchers finds a match, we terminate the other one. We keep the object in an inactive state until it is needed again.

4. Trills and Glissandi

In our previous systems, all notes were defined in advance. If we think of a compound event as one event, the matcher's job is fundamentally to find a one-to-one mapping between "performance" and "score" (see Fig-

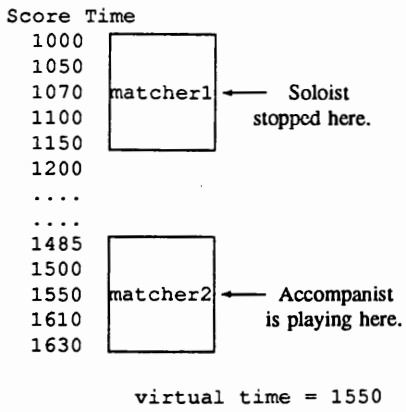


Figure 1: The two matchers' different behavior.

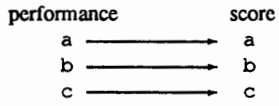


Figure 2: One-to-One Mapping.

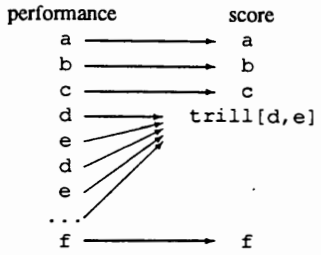


Figure 3: Many-to-One Mapping.

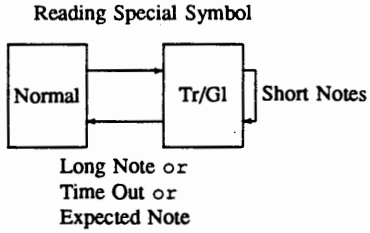


Figure 4: State Transition Diagram.

ure 2).

But this is not true in case of trills or glissandi, because these ornamentations depend on the player and are not precisely specified in the score. For instance, a trill does not specify how many notes to play, nor does a glissando always specify what pitch to begin from or exactly what notes to play. This is determined by the player and may vary from performance to performance. For example, we cannot rewrite the score, from trill[a,b] into [a,b,a,b,a,b,a,b,a,b] because we do not know how many iterations there will be. In order to cope with this kind of unpredictability, we need to prepare some mechanism to find a "many-to-one mapping" (see Figure 3).

Although we could change the matcher, we are hesitant to increase its complexity and decrease its performance. Instead, we use the preprocessor to make one special compound event from the trill using some help from the score. Because the preprocessor converts trills into single events, the matcher does not need to consider whether the data is special or not.

The Preprocessor. The preprocessor consists of a finite state machine. When it detects special symbols like "trill" or "glissando" in the score, it changes its internal state from "normal" to "trill/glissando". It also gets some other information such as expected termination time of the trill/glissando from the score. The character of these ornamentations is that notes are far shorter than a quarter note. Thus, the preprocessor will not exit the trill/glissando mode until one of the following conditions: the preprocessor gets a long note, the termination time of that ornamentation arrives, or the next note after the trill/glissando is performed near its expected time (see Figure 4).

When the preprocessor enters the trill/glissando state, it sends a special symbol to the matcher. While the preprocessor is in that state, it does not send information to the matcher (see Figure 5).

Informing the Preprocessor. Every time the matcher gets a match, it can predict the next note. If this is a special symbol indicating trill or glissando, the matcher tells the preprocessor about it. That causes a state transition in the preprocessor.

performance	preprocessor ->matcher	score
a	a	a
b	b	b
c	c	c
d	Tr/Gl	Tr/Gl
e		
d		
e		
d		
f	f	f

Figure 5: How input is processed by the preprocessor.

5. Delayed Decisions

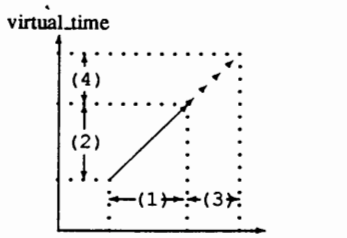
The matcher reports its location to the accompanist whenever a newly performed note leads to a better overall match than any match obtained earlier. Sometimes, this rule does not work. For instance, the soloist may play some grace notes or a trill or a glissando. Sometimes, the preprocessor cannot handle this input correctly, and it sends some extra notes to the matcher. In these cases, there is a small possibility of finding a wrong match. To prevent that kind of accidental matching, we should avoid trusting all reports from the matcher. The Accompanist can reliably trust the match at the n^{th} note if a match of the $n-1^{th}$ note was reported the last time. This consecutive match increases the accompanist's confidence about where it should be playing. But, if a new match is not consecutive to the last match, the system should be suspicious about that match because it indicates that the player made some mistakes recently. In such a situation, the matcher delays its report to the accompanist. If the soloist is playing grace notes, a trill or a glissando, the next note will come soon. Then the preprocessor will get a note within the delay time, realize something is wrong (because the matcher cannot match the note) and the delayed report is canceled. If nothing happens to cancel the report, it is sent to the accompanist.

We delay reporting a match only in case of a dubious match, and only for a short period of about 100 ms. Nevertheless we have to compensate for that delay time. If we suspend a match N ms, and speed is defined as

$$speed = \Delta virtual\_time / \Delta real\_time$$

delay in virtual time will be

$$virtual\_time\_delay = N * speed$$



(1) $\Delta real\_time$
(2) $\Delta virtual\_time$
(3) N
(4) $virtual\_time\_delay$

Figure 6: Compensation for the delay.

We pretend as if the match, whose virtual time equals

$$virtual\_time\_of\_that\_match + virtual\_time\_delay,$$

occurred at

$$real\_time\_of\_the\_event + N$$

in realtime (see Figure 6).

This computation removes any first order effects of delay from the calculation of current virtual time and speed, which is normally recomputed by the accompanist on every reported match.

Implementation Details. We keep a buffer to save one delayed report from the matcher. When we want to delay a report, we save the place of the match, when it occurred, and so on, and a flag is set to indicate the buffer is full. We also save the time limit of the delay, which is the sum of real time and delay time. So, if the match occurred at 1200 ms, and suspension time is 100 ms, then the time limit will be 1300 ms. To cancel the report, we merely reset the flag to indicate the buffer is now empty. We check the buffer frequently by polling, and if the time limit of a full buffer passes without getting a new note, we send a report the accompanist, but if we get a new note within the delay time, we will cancel the report.

6. Other enhancements

Delayed decisions, dealing with ornaments and multiple matchers make up most of our work, but there are some other changes worth mentioning. These last enhancements are described below.

```
(performance)  0010 1010 0001
(score)        0010 1001 0001
                                EOR
(result1)      0000 0011 0000
(score)        0010 1001 0001
                                AND
(result2)      0000 0001 0000
```

Figure 7: Bitwise operations.

```
X    Y
1    0 => must be played
2    0
3    1 => allow one unplayed note.
4    1
5    2 => allow two unplayed note.

X --- Number of note
        in the score event
Y --- Allowed unplayed notes
```

Figure 8: Table of allowed unplayed notes.

6.1. Octave Equivalence

We decided to neglect octave differences in pitch. By doing this, the soloist can play in any octave. Also, he can play an opened chord even if it is specified as a closed chord. However, there are some drawbacks to dropping the octave information. Since we treat C3 the same as C4, we have a small risk of getting a wrong match. Although the best approach depends upon the music and the performance, we are happy with this choice, especially in light of the optimization described next.

6.2. Using Bit Vectors

To deal with polyphonic notes, we use bit vectors to represent sets. The vector is a 2 byte integer, whose LSB corresponds to 'C', and whose next bit corresponds 'C♯/D♭', and so on. In this data structure, we can express any combination of notes in 2 bytes. This is possible because we limit the note value from 0 to 11. If we used pitch value in the full MIDI range(0 - 127), we would need 16 bytes. Because matching calculations are time critical, it is very important to make note data and operations as compact and fast as possible. By taking a bitwise "exclusive or" between the score and performance events, we get "result1", the bitmap of the difference between the two. Then we "and" result1 with the score. We then have "result2", which shows which notes in the score are not played. See figure 7. By the first operation (exclusive or), we know the difference, which includes unplayed notes and extra notes. Extra notes may be a grace note, so we choose to ignore extra notes. This is accomplished by the "and" operation. After that, we have only unplayed notes. (This can be simplified slightly: $result2 = score \wedge \overline{perf}$.) By counting the bits, we can get the number of unplayed notes. If this number is smaller than some constant which is associated with the number of the notes in the score event, we will treat it as a match. Thus the accompaniment system can allow the player to play a difficult chord imperfectly, and can report a match earlier than it would if it waited until a chord was completed.

Let's take figure 7 as an example. The number of unplayed notes is 1, and the number of notes in the score event is 4, so 1 unplayed note is allowed. Thus, we say that these two match.

The merits of these operations are that we can avoid time consuming set operations in the matcher which would be required if we were to use other data structures. These operations also effectively eliminate the grace note, and allow the performer to add some extra notes without penalty. An interesting consequence is that notes can be purposefully omitted from score events. For example, instead of [C E G], one can specify [C] in the score. Our matching algorithm will eliminate [E G] automatically from the performance, and report a match. In this case, any chord with a C, or just the single note C will match.

7. Evaluation

We succeeded in getting a very robust system using the ideas described above. The techniques for handling grace notes work especially well in comparison to our earlier systems. A problem arises when the soloist does not play correctly just before the beginning of a trill or glissando. In this case, the matcher fails to inform the preprocessor about the next note; the preprocessor then fails to change its internal state, and it sends the all of the ornamentation notes to the matcher. The matcher will be confused

by the unexpected notes and will not be able to find a match. Even if that happens, the second matcher, which is located near the current virtual time, will find match after soloist finishes trill or glissando.

8. Future Work

Viewed as an expert system, this computer accompaniment system is still in a knowledge acquisition phase. We have improved the system performance by creating richer processing models based on our intuitive understanding of how we, as musicians, process and understand performance information. In the future, we are planning to improve the system further as outlined below.

At present, we have to specify trills or glissandi by hand. If the player wants to make the score by giving a real-time performance, we need to provide an automatic detector of trills and glissandi. Also, it would be very nice if we could provide a very useful general and integrated environment so that we could play, record and edit music easily. For example, the CMU Musician's Workbench Project[4] will consist of graphic score editors that exchange data through a common music representation used for performance capture and synthesis. The *eled* facility is another example of a score manipulation system [5].

Performers sometimes ignore repeat signs, miss the coda, or skip a page of music. In our implementation, the score is a one-dimensional array. Thus, we cannot express some musical branch structure specified in the paper score such as "repeat","D.S",etc. By adding structures to the score to encode repeats or branches, we can use an extra matcher to detect when the player jumps to the wrong spot.

Once our system gets completely lost, the only chance for recovery is if the performer starts following the accompaniment. (Recall that we keep the window of one matcher around the current virtual time of the accompaniment.) To prevent catastrophes, we want to have a last resort, that is, the capability of searching the whole score when all else fails.

Improvisation is called for in contemporary art music as well as popular music and jazz. The current accompaniment system cannot deal with much variation in the performance; however, small and predictable variations like grace notes are handled very well, and we have used the preprocessor to filter more complex sequences (trills and glissandi). These techniques might be extended to handle improvisatory sections. Ultimately, an accompaniment system should "listen" to an improvisation rather than filter it out. We are investigating entirely new techniques toward this goal [6].

9. Summary

We have extended our earlier polyphonic accompaniment systems in order to handle trills, glissandi, and grace notes. We have also made the system more robust in the face of performance errors. The two primary techniques are the use of multiple matchers and the use of delays in reporting matches. Multiple matchers allow the accompaniment system to look for matches within disjoint intervals of the score. With just two matchers we can make the system much better at recovering from performance errors. By delaying reports of matchers to the accompanist, we can filter out grace notes, performance "glitches", and other short-lived mistakes. Delays allow us to avoid making a hasty conclusion when the data is suspect.

10. Conclusions

Computer accompaniment offers the composer and performer much of the flexibility that was lost in the transition from live performance to mixed live and taped music. Before computer accompanists become commonly accepted, they must be reliable, responsible, and capable of following the full range of standard performance practice. We believe our contributions have brought us close to this goal.

11. Acknowledgments

The authors wish to thank the Yamaha Corporation, the Center for Art and Technology and the Computer Science Department at Carnegie Mellon University for their support.

References

[1] Dannenberg, R. B. 1984. "An On-Line Algorithm for Real-Time Accompaniment." *Proceedings of the 1984 International Computer Music Conference,*

1984. San Francisco: Computer Music Association, pp. 193-8.

[2] Vercoe, B. 1984. "The Synthetic performer in the context of live performance." *Proceedings of the 1984 International Computer Music Conference, 1984*. San Francisco: Computer Music Association, pp. 199-200.

[3] Bloch, J. J. and Dannenberg, R. B. 1985. "Real-Time Computer Accompaniment of Keyboard Performances." *Proceedings of the 1985 International Computer Music Conference, 1985*. San Francisco: Computer Music Association, pp. 279-89.

[4] Dannenberg, R. B. 1986. "A Structure for Representing, Displaying, and Editing Music." *Proceedings of the 1986 International Computer Music Conference, 1986*. San Francisco: Computer Music Association, pp. 153-60.

[5] Decker, S. L. and Kendall, G. S. 1985. "A Unified Approach to the Editing of Time-Oriented Events." *Proceedings of the 1985 International Computer Music Conference, 1985*. San Francisco: Computer Music Association, pp. 69-77.

[6] Dannenberg, R. B. and Mont-Reynaud, B. 1987. "Following an Improvisation in Real-Time." *Proceedings of the 1987 International Computer Music Conference, 1987*. San Francisco: Computer Music Association, pp. 241-8.