

A CONVERSATION-BASED FRAMEWORK FOR MUSICAL IMPROVISATION

BY

WILLIAM FRANKLIN WALKER

B. S., University of Illinois, 1987

M. S. University of Illinois, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

Abstract

Many recent computer applications seek to support or even participate in structured collaborations with humans. Examples range from computer-supported collaborative work to interactive music systems. This thesis presents a general model for structured collaboration based on conversational turn-taking. According to the model, collaboration participants share both a representation of the collaboration's structure and the algorithms governing navigation through that structure. ImprovisationBuilder, an object-oriented framework written in Smalltalk-80, supports this model for musical improvisation. The framework demonstrates how computers can participate in musical improvisation, and how the study of conversation can improve that participation. Two musical improvisational domains are described. The first is a free-form three part canon defined by Sal Martirano's Sound and Logic system. The second is the tradition of small jazz ensemble performance made popular in the 1940's and 50's. ImprovisationBuilder was successfully applied to these domains.

Acknowledgments

This work was supported and inspired by Sal Martirano and my colleagues at the CERL Sound Group. Sal's pioneering work in interactive music systems gave me fertile ground in which to grow my project. Kurt Hebel provided boundless Smalltalk-80 expertise and numerous design insights. Carla Scaletti provided thorough proofreading, constant encouragement, and many exciting brainstorming sessions. Lippold Haken's leadership and enthusiasm made CERL a great place to do sound and music research. I could not have survived without the camaraderie and interaction of Kelly Fitz, Bryan Holloway, Andrew Horner, Paul Christensen, Mark Smart, Rich Baraniuk, Miles Jackson, Bill Schaeffer, Sherwin Gooch, and Charlie Robinson.

The computer science members of the thesis committee have contributed much to my work. Ralph Johnson ensured that both my Smalltalk framework and my English were up to standards. Simon Kaplan advocated the relevance of this work to CSCW and introduced me to its literature. Michael Faiman gave sound advice about dissertating and kept me from making unsubstantiated claims about artificial intelligence and music.

My tenure with the Race Street Jazz Conspiracy provided real world experiences I needed, both to fuel my research in improvisation and to remain sane. Special thanks to Patric Cohen, Kelly Fitz, Frank Mauceri, Tucker Robison, and our faithful fans.

Rick Lundell has been my traveling companion through the dissertation journey. His accomplishments spurred me to action, and his analytical and rhetorical skills helped me to write and think more clearly.

My wife, Mary Wisnewski, has borne my triumphs and doubts with grace and has given me the security I needed to finish graduate school. Her proofreading and insightful questions have also been invaluable. My brother, John, was always there for me. My mother, Shirley Walker, has given me a perspective on life that every graduate student needs. My father, William Walker, of the Civil Engineering department, has never minded sharing our names, and has been my life-long role model for excellence in academe.

Table of Contents

1.	Introduction.....	1
1.1.	Problems for Human-Computer Musical Interaction.....	3
1.1.1.	Generation.....	3
1.1.2.	Interaction.....	6
1.1.3.	Re-use	8
1.2.	ImprovisationBuilder	9
2.	Problem Domain.....	10
2.1.	Sound and Logic Case Study.....	10
2.2.	Jazz Case Study	12
2.2.1.	The Structure of Jazz Performances.....	13
3.	Structured Collaboration.....	16
3.1.	Improvisation and Conversation.....	16
3.1.1.	Conversation Analysis.....	20
3.1.2.	Turn-taking for Jazz Solos.....	22
3.2.	Human-Computer Interaction.....	25
3.3.	A General Model of Collaboration	26
3.4.	Informing Framework Design.....	28
4.	An Object-Oriented Framework	30
4.1.	Music Representation.....	33
4.2.	Parsing Musical Input.....	36
4.3.	Representing Improvisational Structure	37
4.4.	Generating Musical Output.....	38
4.4.1.	Transformer.....	39
4.4.2.	Composer	40

4.5.	Coordinating Shared Information	42
4.6.	Realizing Musical Output.....	43
5.	Applying the Framework.....	46
5.1.	Sound and Logic Case Study.....	50
5.2.	Jazz Case Study	54
5.2.1.	Parsing Jazz.....	54
5.2.2.	Following Improvisational Structure.....	55
5.2.3.	Generating Jazz.....	57
6.	Implementing the Framework.....	59
6.1.	Preemptive versus Cooperative Multi-tasking	60
6.2.	Processing Keyboard Commands.....	62
6.3.	KymaTimbre.....	63
6.4.	Real-time Behavior.....	65
6.4.1.	Lead time for Scheduling.....	66
6.4.2.	Importance of Listener Readiness.....	67
7.	Summary and Future Work.....	68
7.2.	Future Work	70
	Appendix A. Sample Jazz Performance	71
	Bibliography.....	81
	Vita.....	89

1. Introduction

As computers grow more powerful, their interactions with humans become richer and more complex. Many different notions of human-computer interaction—graphical adventure games, computer-supported collaborative work, intelligent agents—point toward human-computer collaboration. Many collaborations are largely beyond the computer’s reach, such as conversing with a computer in a natural language. However, computers can perform better in collaborations that offer some structure—a dialogue about airline reservations rather than an unconstrained conversation, for example[28]. The structure allows the computer to better identify the roles of other participants and to better generate an appropriate contribution to the collaboration.

In this thesis, I offer a model of structured collaboration based on the idea of conversational turn-taking. A collaboration’s structure is modeled by a network of states, each of which defines roles for the participants and a set of criteria for determining transitions between states. While in each state, the participants construct turns out of discrete units. State transitions occur at transition relevance places, found at the end of each turn construction unit. The transition criteria are examined at each transition relevance place. If they are satisfied, the collaboration enters a new state, and the participants assume new roles.

I apply this model to musical improvisation. Mainstream jazz improvisation, for example, follows a conventional performance structure that includes statement of the melody, melodic solos based on that melody, and musical dialogue between the musicians. The transitions between jazz solos are equivalent to the transitions between speakers in a conversation. To demonstrate the effectiveness of this model for musical improvisation, I have written `ImprovisationBuilder`, an object-oriented framework for interactive music systems. Users can easily construct new systems by combining the framework’s built-in components. `ImprovisationBuilder`’s abstract classes provide a template for constructing interchangeable new components.

My motivation for this project is fourfold. As a musician, I rehearse and perform with a small jazz ensemble[43] and observe with great interest the musical interactions between my colleagues. Jazz musicians, especially musicians who have played together for a long time, anticipate when and what others will play, and coordinate their efforts with only the briefest discussions of what to do next. This research has helped me to better understand the collaborative processes of improvisation.

As a software designer, I want to design elegant, reusable software. The framework approach to object-oriented design helps to achieve this goal. Object-oriented frameworks are a relatively recent idea[15]. Thus, each successful framework further validates the framework approach. The creation of frameworks is especially significant in an infant field like computer improvisation, in which the lack of a broad understanding of the field means researchers duplicate each other's work using different terminology and communicate inadequately for lack of a common language. As frameworks become more prevalent, they can encourage a common language for computer improvisation.

As a software engineer, I am interested in building a robust working system. There is only one way to demonstrate competence in improvisation—to improvise. Whether the theoretical basis of this framework improves interactivity can only be determined by putting the system on-stage with human performers. This research therefore includes the development of a robust real-time platform for interactive music systems. As a computer scientist, I am trying to understand the general principles that underlie human-computer interaction and how diverse collaboration domains may be amenable to analysis by a single model. Such a model allows the transfer of insights from one domain to another.

ImprovisationBuilder (IB) addresses three central problems for human-computer interaction in music—generation, interaction, and re-use. IB has been applied to two musical domains—the free-form improvisation defined by Sal Martirano's Sound and Logic program, and the small jazz ensemble of the 1940's and 50's. IB proved quite appropriate to the two musical domains.

1.1. Problems for Human-Computer Musical Interaction

The history of computer music reflects the steady increase in the amount of computation available in real time. Until the mid-1980's, computer music researchers used mainframes to compute digital audio samples at a rate ten or a hundred times slower than real time. With a few exceptions, composers could only combine instrument and synthesized sounds by mixing live performers with prerecorded tape. Digital signal processors and fast personal computers have made many synthesis algorithms computable in real time. Hence, musicians now perform on synthesizer keyboards and interact on-stage with computers.

The nature of each composition determines the character of these human-computer interactions; in some compositions, human performers play from a score while the computer follows the human's progress through the score and performs its own, predetermined part[1, 12]. In other cases, the human performer improvises and the computer reacts in some algorithmic fashion[63]. Some composers combine several techniques in different sections of a piece[39, 58, 59].

Many researchers are currently investigating various aspects of computer participation in improvisation. This project differs from other efforts in that it does not seek to model a particular musical style or mode of interaction. Instead, it provides a model encompassing the interactions common to all structured collaborations. As an object-oriented framework, ImprovisationBuilder (IB) embodies this model, providing an environment in which existing musical algorithms for listening and composing cooperate to produce interaction. The next three sections illustrate three central problems in computer music software and how IB's design addresses them.

1.1.1. Generation

As with other computer forays into creative endeavors, many have speculated on the computer's ability to compose music. As in other domains, composers and researchers have

devised many different schemes for generating musical material. Some employ models of musical cognition, while others model musical structure. Some are intended to generate many kinds of musical output, while others are built to generate a single composition or family of compositions. Here are a few examples of generative algorithms.

- *Transformation*

Transformation is prominent in generative music systems. For example, Levitt has written a transformational system for generating jazz solos[31]. The system rates the “interesting” qualities of a phrase with simple musical criteria. The system transforms an interesting phrase from the melody to begin its solo. It applies further transformations to yield additional phrases. When the transformed version becomes uninteresting, the system chooses another phrase from the melody and the process begins again.

- *Formal Grammars*

Formal grammars can describe music at various levels, from notes to harmonic structures. While harmonic structures are usually devised by the composer, improvising musicians have evolved a set of chord substitutions for embellishing standard harmonies. Steedman formalizes blues chord substitutions as a context-sensitive grammar that can derive complex harmonic sequences from the traditional blues chord sequence[50].

Bel’s BP2 is a sophisticated system for using stochastically controlled context-free grammars to generate music in several improvised styles[4]. It was originally designed to validate a grammar for Indian tabla drumming. Giomi and Ligabue take a similar grammar-based approach to generating jazz solos[19].

- *Markov Chains*

Two commercial programs, M and Jam Factory, use Markov chains to generate output that is statistically similar to their input[63]. Markov chains were first used in 1959 as a probabilistic composition technique by Hiller[25]. Hiller used them to govern sequences of musical pitches, a technique M and Jam Factory also use. A low-order Markov analysis of an incoming pitch stream

captures only the set of pitches allowed in the current scale and the prevalent intervals between consecutive notes. A higher-order analysis begins to capture some of the short-term structure of the input, recording the shapes of small melodic phrases.

- *Rule-based Systems*

Several researchers have brought the standard AI techniques of searching with backtracking and rule-based systems to bear on music composition. Many of these have focused on four part chorales as a well-defined, highly constrained genre[16, 53]. These systems encode the rules taught to beginning music theory students (e. g., no parallel fifths) and use them to construct syntactically correct chorales.

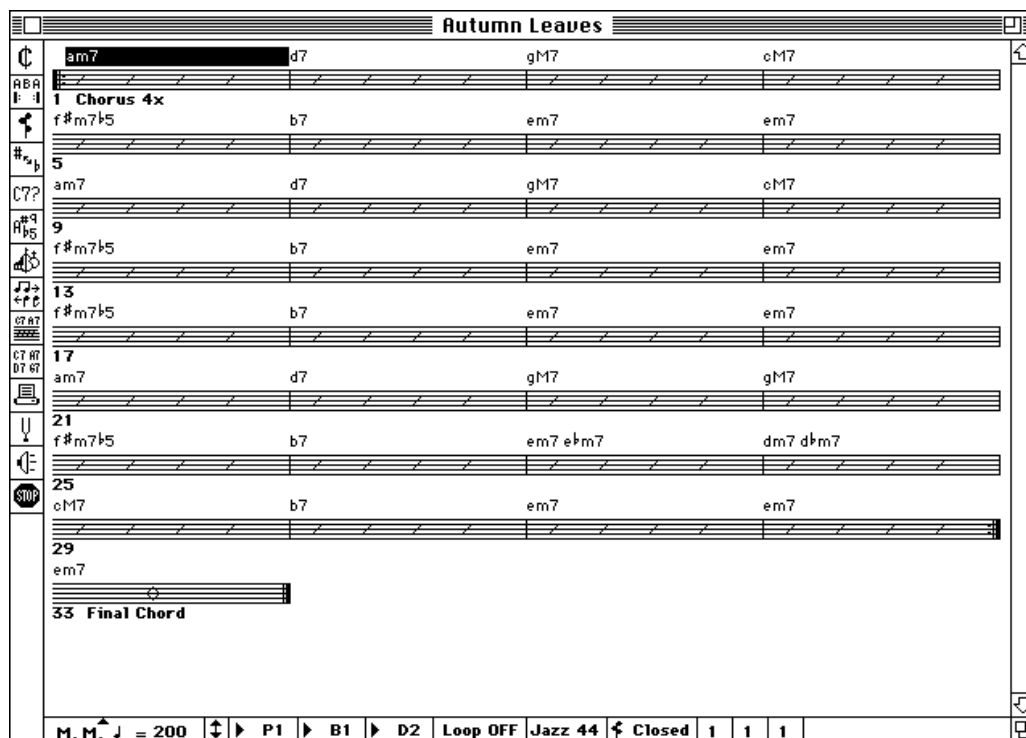


Figure 1.1. MiBaC Jazz screen showing harmonic representation

- *Pattern-based Systems*

Two commercial products, MiBaC Jazz and Band-In-A-Box bring pattern-based music generation to the marketplace[17]. These programs instantiate accompaniment patterns for piano, drum, and bass parts according the song's harmonic structure (see Figure 1.1). Such products are good jazz educational tools, providing the beginning improvisor with a tireless accompanist. In

contrast to the audio recordings of accompaniments sold to jazz students, computer accompaniments can be generated in any key or tempo for any amount of time.

IB can accommodate any of these generative algorithms. The IB design places generative algorithms in a hierarchy of *Composers* (see Section 4.4.2). *Composers* generate new *Phrases* on demand, and contain whatever state information their algorithm requires. Existing *Composers* are based many algorithms, including Markov Chains (*TransitionTable*, see Section 4.4.1) and pattern instantiation (*HarmonyGenerator*, see Section 5.2.3).

1.1.2. Interaction

Beyond the hard problem of computers generating music lies the harder problem of computers interacting with musicians. As summarized in Section 1.1, interaction became a practical topic for computer music as personal computers became powerful and affordable. Here again, some composers design custom software for each new interactive composition, while others work toward more general systems.

- *Controlling generation in real-time*

Many composers have created musical human-computer interactions by running generative algorithms in real time and controlling their parameters with the human performer's input. For example, the M program described above tabulates sequences of pitches performed by the human player. M uses this table probabilistically to create new sequences of pitches. The user controls M by feeding it musical input and by controlling analysis and generation parameters during performance.

- *Score Following*

Enabling a computer system to track a human performer's progress through a musical score is the subject of several investigations[1, 11, 56]. This task involves relating both the performed rhythm and pitches to those notated in the score. Matching the performance with the score can be

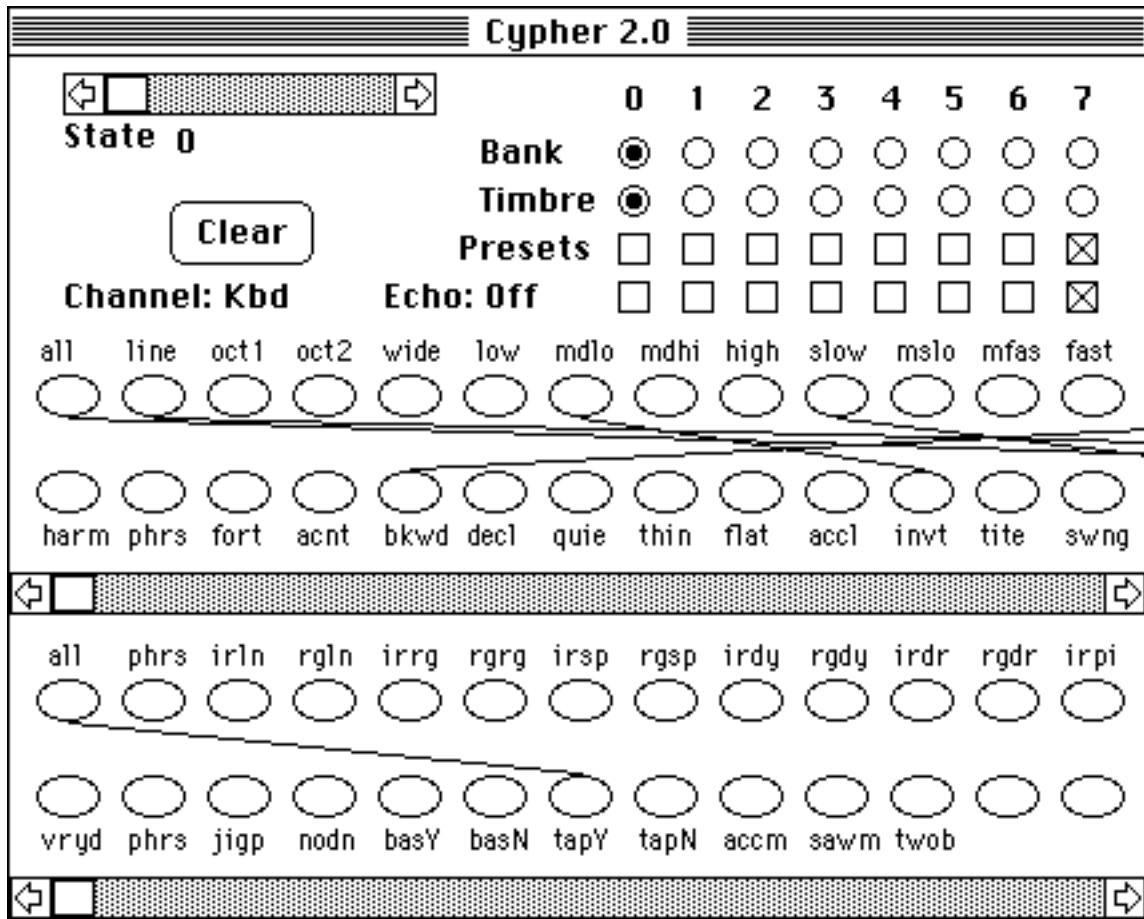


Figure 1.2. Cypher display showing agent interconnections

thwarted by absences (when the performer skips a note or a measure or a page in the score) or surpluses (when the performer plays extraneous notes).

- *Cooperating Agents*

Robert Rowe's Cypher is a system based on Minsky's notion of the Society of the Mind[46]. Minsky postulates that the mind operates by breaking complex tasks into simple ones, each of which is handled by a cognitive agent. Cypher decomposes the task of musical interaction into sub-tasks that are handled by musical agents. The user defines Cypher interactions by connecting agents that detect input qualities to agents that produce output qualities (see Figure 1.2). For example, the human performer playing loud notes can cause Cypher to play more trills. While such individual connections are straightforward, several simultaneous connections produce

complex and interesting behavior. Other agents can reconfigure the connections, causing the system to behave differently during different sections of a performance. Cypher combines the efforts of several agents to detect complex input phenomena such as meter and harmonic rhythm.

IB distributes responsibility for interaction over several parts of the IB design. IB places input parsing algorithms in *Listener* components (see Section 4.2). *Listeners* break the musical input into *Phrases* while storing in the *PolicyDictionary* other useful information about the input. The centralized *PolicyDictionary* allows communication between *Listeners* and *Composers*, facilitating the kind of connections exhibited by Cypher.

1.1.3. Re-use

As computer music software grows larger and more complex, software engineering issues become more important to the computer music community. The development of large computer music software has fostered interest in modularity and code re-use. Composers who formerly developed all their tools from scratch now want to re-use existing software. Various responses to this situation have surfaced.

- *Languages*

Languages for real-time musical event processing have become tremendously popular in the computer music community. This popularity parallels the development of the Musical Instrument Digital Interface (MIDI), a protocol for encoding and transmitting musical events over a serial communication line[32]. MIDI devices send and receive real-time messages specifying the beginning and ending of musical notes. These languages provide reuse of their facilities for real-time scheduling and MIDI interfaces, facilities each composer would otherwise have to construct for himself or herself.

Max, a popular interactive music language, provides a visual programming environment instead of a text editor and compiler[42]. The user uses a graphical editor to connect boxes

representing simple operators, yielding composite operators with more complex behavior. Interactor is another event processing language with a similar visual programming environment[9]. These languages form the basis of many interactive compositions.

- *Libraries*

Various libraries written in these languages have fostered greater levels of code reuse by making available frequently used subroutines. Simon Bolzinger developed DKompose, a set of Max routines useful for compositions in which the computer controls a MIDI-equipped Yamaha piano[6]. Todd Winkler developed FollowPlay, a collection of Max routines for interactive performances[59]. His library includes a score follower and other commonly used modules that composers would otherwise have to re-implement.

- *Frameworks*

As a framework, IB represents the next plateau of software reuse. Design is widely regarded as the most difficult aspect of software engineering; the framework approach focuses on reuse of design as well as code[61]. IB's abstract classes provide a template for easily creating new components, while its concrete classes provide composers with a library of ready-to-use components.

1.2. ImprovisationBuilder

This project differs from other efforts in that it does not seek to model a particular musical style or mode of interaction. Instead, it provides a model encompassing the interactions common to all structured collaborations. ImprovisationBuilder, an object-oriented framework written in Smalltalk-80, addresses the problems of generation, interaction, and reuse in a unified, extensible design. It uses generative techniques to create musical material. It incorporates interactive techniques to coordinate algorithms for listening and composing. It provides software reuse through the framework approach to design.

2. Problem Domain

In this chapter I will discuss ImprovisationBuilder’s problem domain, musical improvisation. The *Harvard Concise Dictionary of Music* gives the following definition[45]:

“Improvisation, extemporization. The art of creating music spontaneously in performance.” (p. 235)

Improvisation is part of most of the world’s musical traditions, and plays an important part in the work of many living composers and performers.

Within this broad space of possible collaborations known as musical improvisation, I will discuss two particular domains in detail. One is a free-form new music defined by Sal Martirano through his Sound and Logic software. The other is a restricted subset of “mainstream” small group jazz improvisation, as defined by its practitioners in the 1940’s and 50’s. In Chapter Five, I will show how I used the framework to construct participants for these domains.

2.1. Sound and Logic Case Study

ImprovisationBuilder began as a generalization of Martirano’s Sound and Logic (SAL) software. This program, as part of the YahaSALmaMAC orchestra, participated in numerous public performances and the CD recording of Martirano’s “SAMPLER: Everything Goes When The Whistle Blows” [35]. I will first summarize the work preceding SAL, and then describe the musical interaction for which it was designed.

Salvatore Martirano is a pioneer in interactive music systems. The first system was the Sal-Mar Construction, an interdisciplinary project involving Martirano, computer science graduate student Sergio Franco, and ILLIAC III designers Rich Borovec and James Divilbiss[18, 34]. It was based on the idea of “zoomable” control—being able to apply the same controls at any level,

from the micro-structure of individual timbres to the macro-structure of an entire musical composition. Weighing in at 1500 pounds, the Sal-Mar Construction provided digital control over analog synthesis modules through a unique touch panel consisting of banks of switches assignable to any level of control.

A later project was the YahaSALmaMAC orchestra, featuring MIDI synthesizers under control of the Sound and Logic (SAL) program. Martirano and David Tscheng implemented SAL in LeLisp on the Apple Macintosh. SAL participates in performances by transforming gestures played by the human performers into new gestures. The transformations used are designed to preserve the musical coherence of the input by preserving rhythmic and harmonic relationships while generating interesting variations.

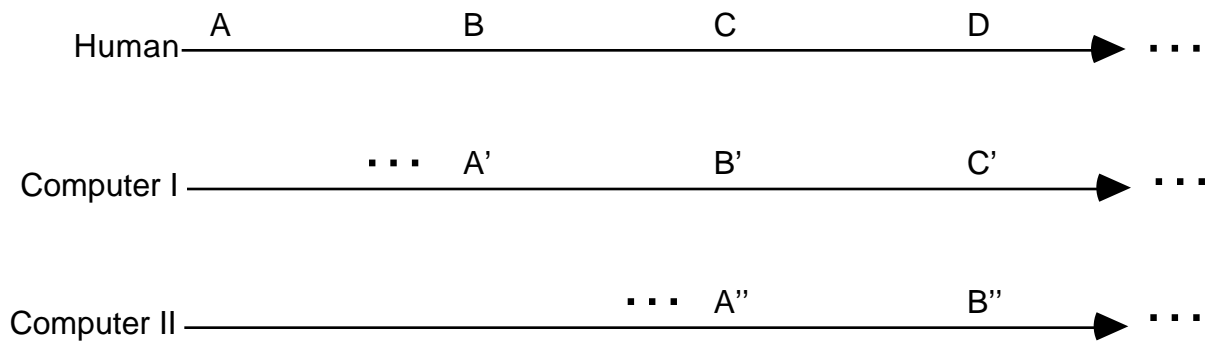


Figure 2.1. Sound And Logic three part canon

SAL is intended to participate in a loosely structured three part canon, with the human performers providing the first part and SAL providing the other two parts (see Figure 2.1). One computer part is a transformation of the human performance. The other computer part is a transformation of the first computer part. Each computer part provides a transformed musical stream delayed by approximately one minute. That is, the human player makes a musical gesture and hears the first computer answer with a transformed version one minute later. The second computer part will provide further transformed versions another minute later.

Over the course of the performance, the human musicians alternate between playing together with the computer and each playing alone with the computer. Accordingly, SAL shifts from transforming both performers' output to transforming just one performer's output. Its role

also changes as the human performer alters the parameters of SAL’s transformations. By using various keyboard commands, the human performer guides the computer through these role changes as the performance evolves (see Section 5.1).

2.2. Jazz Case Study

When considering target musical domains for ImprovisationBuilder, jazz stands out as music with a strong improvisational tradition. Throughout most of this century, jazz has evolved a variety of different styles—ragtime, swing, bebop, cool jazz, free jazz, fusion—all of which are still practiced today. Thus, the term ‘jazz’ has acquired many meanings. In this chapter, I will use the word ‘jazz’ to mean the kind of mainstream small group performances that came to prominence in the 1940’s and 50’s. This kind of jazz is well documented by an extensive body of instructional and analytical materials (for example, *Jazz Styles* by Mark Gridley[21]). Several of its features make it suitable for computer participation; a standard performance structure, a harmonic syntax for musical gestures, and clearly defined roles for the performers.

I Thought About You

The image displays a musical score for the song "I Thought About You" in 4/4 time. The melody is written on a single staff in treble clef. Above the staff, the harmonic structure is indicated by a series of chords: B^o, B^b7, A-7, D7, G7sus, A^b7sus4, G7sus, G7, G-7, F[#]-7, E-7, A7, D-7, D^b7, C-7, and F7. The melody consists of eighth and quarter notes, with a triplet of eighth notes in the first measure of the second line. The key signature has one flat (B-flat), and the time signature is 4/4.

Figure 2.2. Fake book excerpt showing melody and harmonic structure

Jazz improvisation usually takes place within the context of a particular song. The repertoire is drawn from popular music of the 1930’s and 40’s, Broadway musicals, and the original compositions of jazz composers. Jazz performers retain the melody of the song, as well as

its underlying harmonic structure. They discard all other information present in traditional sheet music, such as pre-composed accompaniment, leaving them free to create it extemporaneously. Jazz musicians work from “fake books” containing only the required information, as shown in Figure 2.2.

2.2.1. The Structure of Jazz Performances

The performance of a particular piece begins with the performers having determined only which song they will play, what key to play in, and at what tempo. These decisions determine some of their behavior, but shared knowledge of a large-scale structure is the main resource for coordinating their activities. In his textbook about jazz styles, Gridley outlines the mainstream jazz performance structure[21]:

- Play the song’s melody all the way through, so the audience will recognize which song is being performed.
- Have the performers take turns playing solos of unpredetermined length, repeating the harmonic structure of the song as a basis for improvisation.
- Have the performers trade phrases back and forth after all the solos (optional).
- End the performance with a restatement of the song’s melody.

This structure is represented by a time-line in Figure 2.3. Each repetition of the song’s harmonic structure is called a chorus. The performance begins with a subset of the musicians (usually brass instruments) playing the song’s melody (*i*). Another subset known as the rhythm section (piano, guitar, bass, drums) improvises an accompaniment based on the harmonic structure of the song. The first soloist then improvises a melodic line for one or more choruses (*ii*). During

the solos, the rhythm section continues to improvise an accompaniment, reacting to and supporting the soloists. The second soloist follows immediately for further choruses (iii). Each soloist takes several choruses, allowing him or her time to explore several musical ideas and elaborate on the melody.

At some point during the solos, one of the players may raise four fingers to propose “trading fours,” a standard procedure for exchanging short phrases among the musicians. The other musicians agree verbally or with an assenting gesture; if they do, trading begins immediately following the last solo (vi). Musicians trade fours in the same order they soloed in, each playing a four measure phrase. When the band includes a drummer, the drummer injects a four measure phrase between the soloists. The soloists are accompanied, but the drummer is not. Eventually, a musician will pat the top of his or her head to signal the end of trading fours. Trading fours concludes when the end of a four coincides with the end of a chorus¹. Trading fours (vi) is followed by a restatement of the melody (vii) and an ending (viii).

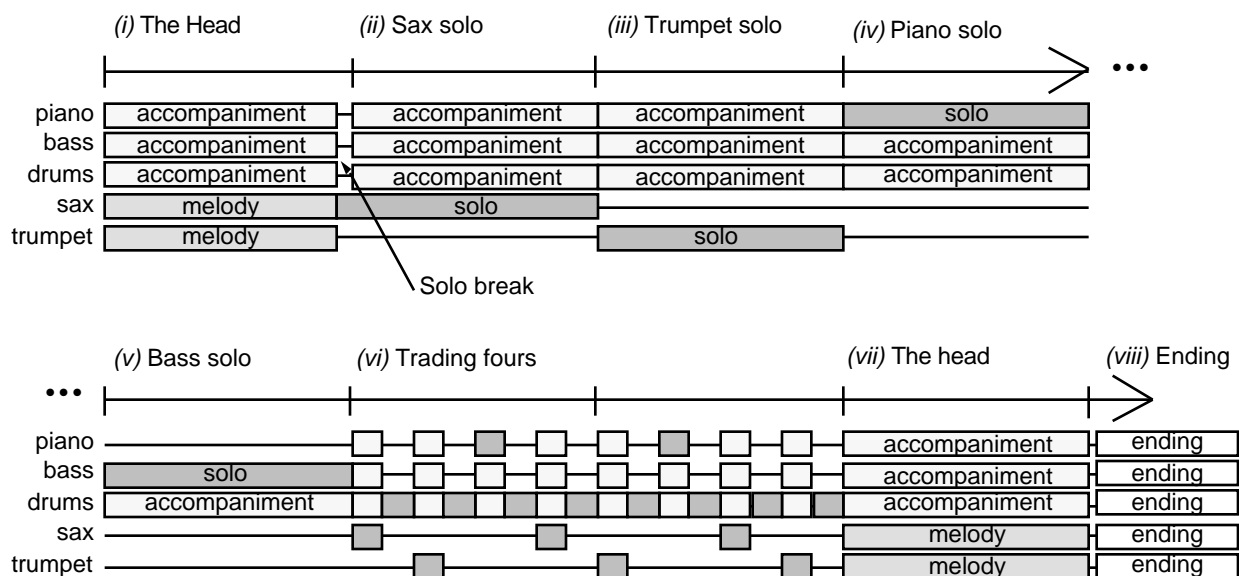


Figure 2.3. Time line of jazz improvisation structure

¹An optional constraint is to stop trading only after each soloist has played an equal number of fours.

Figure 3.2 shows an uneven allocation of fours.

Performers using the scheme shown in Figure 2.3 have many issues to negotiate during the performance, such as the order of soloists, whether the optional trading will take place, and when to end the trading. Throughout the solos (*ii*)—(*v*), musicians use musical, visual, and verbal techniques to negotiate the order of soloists, depending on the number of performers and their familiarity with each other. If only one musician plays the melody, he or she often solos first. Thereafter, either each soloist selects the next one, or a musician can self-select as a solo ends. When one of the musicians is the band leader, he or she may assume the responsibility of ordering the solos and even of determining their duration. Some musicians make these music structural negotiations part of the performance. In James Brown’s performances of “Sex Machine,” he engages in several shouting matches with the rest of the band, such as “Can I take it to the bridge?” “Yeah!” and “Can I hit it and quit?” “Yeah!” to mark transitions between various parts of the song[7].

3. Structured Collaboration

An object-oriented framework represents a specific outlook on the problem domain it addresses. Its authors use their background and theoretical premises to design the framework's components and collaborations. ImprovisationBuilder's design rests on two premises; that improvisation is conversation, and that designers should regard human-computer interaction as conversation. Taken together, they point toward building interactive music systems from ideas about conversation. Improvisation and conversation are two kinds of structured collaboration. Extending conversational turn-taking to musical collaboration forms the basis of a general model for structured collaboration. This chapter presents these premises, introduces the field of conversation analysis (CA), and shows its relevance to both improvisation and human-computer interaction (HCI), concluding with a summary of how the above ideas influenced the framework's design.

3.1. Improvisation and Conversation

Improvising music is making musical conversation². This analogy surfaces frequently in writings about jazz[23, 36, 40, 52]. As described in Section 2.2.1, jazz musicians exchange short phrases with each other in a practice called “trading fours.” Describing this in his book on the relationship between the notions of writer's and musician's style, Hartmann[23] wrote:

“A four measure phrase leaves the player enough room (say, three to six seconds) to develop one idea, to make one statement; yet there is no mistaking the dialogue within

²Reyes Ramos has even proposed the converse, that conversation is like improvisation[44]. His work suggests that a successful interactive music system may shed light on other collaborative domains.

which each statement takes its place, and often the musicians answer each other directly. The resemblance to conversation is uncanny.”

One can observe parallels between the two activities from many perspectives, ranging from the neurological to the sociological. Consider these correlations:

- *What participants do*

A group of participants collaborate to produce a sequence of sounds.

- *The structure of what they produce*

The individual sonic utterances conform to a syntactic structure. Several linguists and music theoreticians have analyzed music using the same generative grammar techniques applied to natural language[10, 27, 29, 30, 41, 54, 60].

- *How participants gained the ability to participate*

While explicit instruction can aid practitioners, competence is acquired implicitly, through prolonged exposure to and experimentation with exemplary collaborations. Sociologist Edward Hall, considering the distinction between learning and acquiring, wrote[22]:

“Every normal individual ‘acquires’ the basic foundations of language.... While all of us ‘learn’ things through the process of instruction, improvisation appears to be more closely allied to acquisition than to learning, which is one reason why it has such an ‘individual’ flavor.” (p. 227)

- *What participants predetermine*

Participants begin their collaboration having determined neither an overall duration nor a schedule of when they will act.

- *How participants synchronize their conduct*

The sounds they produce are both an end product and a shared basis for the timing of utterances. In his study of conversational organization, Charles Goodwin makes an analogy involving trapeze artists that stops just short of connecting improvisation and speech[20]:

“The stream of speech thus seems to provide a (perhaps innately recognized) reference signal capable of synchronizing the behavior of separate participants. (An analogy that comes readily to mind is the music that trapeze artists use to coordinate their separate actions. However, in conversation, the signal used to synchronize the action of the participants, the stream of speech, is itself a product of their coordinated action, much as if the music in the circus was not a preformulated melody but rather an emergent product of the coordinated actions of the performers and simultaneously a resource employed to achieve that very coordination.)” p. 28

Improvised music is exactly the “emergent [musical] product of the coordinated actions of the performers” that Goodwin’s analogy lacks.

- *How participants coordinate their behavior*

Social conventions provide normative structures for both improvisation and conversation. These structures form a body of shared knowledge such as the one discussed in Heath and Luff’s paper on collaboration in the London Underground control room[24]. Describing the engineers who work in the control room, they wrote:

“These practices appear to stand independently of particular personnel, and it is not unusual to witness individuals who have no previous experience working together, informally, yet systematically coordinating their conduct.” (p. 70)

Jazz musicians are often called on short notice to perform with musicians with whom they have never worked. Jazz musicians also participate in jam sessions, during which the same conditions apply. The presence of a shared structure for jazz performance allows these performances to succeed. Sociologist and pianist Harold Becker wrote[3]:

“Whoever called me would tell me where the job was, what time it began, and usually would tell me to wear a dark suit and a bow tie, thus ensuring that the collection of strangers he was hiring would at least look like a band because they would all be

dressed more or less alike.... The drummer would set up his drums, the others would put together their instruments and tune up, and when it was time to start the leader would announce the name of a song and a key— ‘Exactly Like You’ in B-flat, for instance—and we would begin to play. We not only began at the same time, but also played background figures that fit the melody someone else was playing and, perhaps most miraculously, ended together. No one in the audience ever guessed that we had never met until twenty minutes earlier.”

In the jazz context, knowledge of collaboration conventions so defines the task of listening that one must divide the audience into two groups, musicians and non-musicians. Approaching this from a semiotics perspective, Perlman and Greenblatt[40] wrote:

“When we say that the inside audience has structural competence, we mean that they can recognize basic elements in what is being played *as* it is being played.... The rest of the listeners—the outside audience—really do not hear or understand improvised solos... it may be as meaningful in some way for us to hear an impassioned—but untranslated—speech by the prime minister of Japan, especially if we see and hear it in its original context... we’ll never know exactly what the prime minister said.”

(p. 181)

- *Visual communication*

Visual cues provide an extra channel for conveying collaborative cues. In conversation, eye contact aids in coordinating conversation. Bastien and Hostager studied similar cues in jazz by videotaping a quartet consisting of experienced jazz musicians who had never before played together[2]. In order to study a restricted set of interactions, they constrained the order of solos, so that the saxophonist always soloed first. Concerning the group’s saxophonist, Bud Freeman, they wrote:

“Toward the end of Freeman’s solo, we observed two forms of communicative behavior by Freeman, behavior that signaled to the rest of the musicians that he was relinquishing the lead to Hodes. One such behavior was the music theoretic cue of ‘winding down’ the solo; Freeman signaled the end of his solo by directing his musical invention toward the full resolution of the current chord.... The other communicative behavior was a nonverbal visual cue that Freeman directed at Hodes; shortly (a beat or two) before the end of his solo, Freeman looked at Hodes in order to signal the end of his solo. Both behaviors accessed shared, cognitively held information on the social practice level by signaling that Freeman was indeed giving up the lead according to group expectations.” (p. 590)

3.1.1. Conversation Analysis

As enumerated above, conversation and improvisation have many parallels. Among these, the study of how participants coordinate their behavior has most directly influenced the framework. My goal is to translate ideas about coordinating collaborative behavior from the conversation domain to the musical improvisation domain. The approach to conversation most suited to this goal is conversation analysis (CA).

Conversation analysis regards everyday conversation as a highly ordered, collaborative social activity³. CA proceeds by empirical analysis of ordinary talk. Ordinary conversations are recorded and transcribed into a textual format that preserves information about pauses and overlaps (see Figure 3.1, taken from Sacks, Schegloff, and Jefferson[47]). Conversation analysis studies how people design and use language to structure conversational flow.

³See *Computers and Conversation* for an excellent introduction to conversation analysis in the context of human-computer interaction[33].

- A. Oh I have the– I have one class in the evening
 B. On Mondays?
 A. Y– uh::: Wednesdays.=
 B. =Uh– Wednesday, =
 A. =En it’s like a Mickey Mouse course.

Figure 3.1. Conversation transcript from Sacks, Schegloff, and Jefferson[47]

CA findings describe behaviors that occur frequently in ordinary talk. These findings, often formulated as rules, are seen as resources in maintaining the smooth flow of conversation. Skeptical readers can evaluate the findings for themselves by studying the conversations they were derived from.

One example of CA is Gail Jefferson’s study of list-making[26]. Jefferson notes how conversational lists normally contain three parts. This rule manifests itself in both speaker and listener behavior. Speakers produce lists of three items, and rely on listeners to continue listening until after the third list item. Conversely, listeners expect to hear lists of three items. When a speaker fails to fulfill that expectation, the listener can prompt for or cooperate in completion of the list.

- John: Who all is over there.
 Kitty: Oh, [Marcia and Judy] stopped by,
 (Pause)
 John: Who else.
 (Pause)
 Kitty: Oh, what’s his name,
 (Pause)
 Kitty: [Tom.]
 John: Oh.

Figure 3.2. Conversation transcript from Jefferson[26]

In Figure 3.2, Kitty’s initial answer to John’s question sets up the expectation that she is constructing a three part list. When her first turn yields only two elements, John prompts her for additional list items until his expectation is fulfilled. This illustrates the fundamental premise of conversation analysis that people use these rules as resources for speaking and listening and expect others to do likewise.

At the beginning of this chapter I argued that improvisation is like conversation. The application of CA to improvisation is my realization of this analogy. Norman and Thomas, in their summary of CA and HCI, point in this direction[37]:

“Conversation Analysis and its findings...reveal interaction to be an accomplishment more in the nature of ensemble improvisation than an enactment of predetermined dialogue rôles.” (p. 55)

Because improvisation and conversation exhibit so many structural and collaborational correlations, I believe that similar empirical rules coordinate musical improvisation and conversation. Musicians use these rules as resources for playing and listening. As a concrete demonstration, I will apply CA findings about turn-taking to the allocation of jazz solos.

3.1.2. Turn-taking for Jazz Solos

Taking turns at soloing is a central part of jazz improvisation. If a computer is to function in a jazz ensemble, it must detect and participate in the allocation of solos. The Sacks, Schegloff, and Jefferson paper “A Simplest Semantics for Turn-Taking” suggests a way to solve this collaborative problem[47].

Their paper presents a system for turn-taking in ordinary conversation in which conversational turns are constructed of discrete units—sentences, phrases, or even single words. A speaker is initially entitled to one such unit. The speaker constructs each unit so that listeners will correctly project its end. These projectable end points are called transition relevance places because they mark the location of a possible change in speakers. At the end of each unit, the following rules apply, in order:

- 1(a). If the content of the turn-so-far identifies a next speaker, then the party so selected has both the right and the obligation to begin a new turn.

1(b). If the content of the turn-so-far does not identify a next speaker, any listener may attempt to select themselves as the next speaker by beginning to speak. The first person to do so becomes the next speaker.

1(c). If the content of the turn-so-far does not identify a next speaker, the current speaker may continue, provided no other parties selected themselves as the next speaker (i. e, rule 1(b) does not apply).

2. If rule 1(c) applies, the current speaker is granted another turn construction unit, and the entire rule set applies again at the next transition relevance place.

Figure 3.3 shows two conversational fragments from the Sacks paper. These two examples contain overlapping speech, a failure of turn-taking. However, the system predicts these failures and accounts for their structure. In the first fragment, *B* correctly projects that *A*'s turn will end after the word "ask." The start of *B*'s turn overlaps the end of *A*'s, but the overlap is due to uncertainty about how quickly *A* will pronounce "ask" instead of disagreement about the turn's endpoint. In the second fragment, *A*'s turn consists of two constructional units, with a transition relevance place following the word "before." Having projected the end of *A*'s first constructional unit, *B* begins to speak, only to overlap with *A*, who adds an additional unit to his turn. These examples illustrate that both conversational turn-taking and overlap can be explained by the Sacks system.

A: Well, if you knew my argument why did you bother to
a: [sk
B: [Because I'd like to defend my argument [Crandall 2-15-68:93]

A: Uh you been down here before [havenche.
B: [Yeh. [NB:III:3:5]

Figure 3.3. Conversation transcript from Sacks[47]

I apply this system to jazz improvisation by regarding solos as turns and musical phrases as turn construction units. Jazz musicians build their solos from musical phrases. The end of each phrase is a transition relevance place. Unlike free-form conversations, the jazz described here imposes a constraint on when turns should end. In particular, solos are almost always an integral number of choruses. Thus, musicians evaluate the end of a phrase relative to their representation of the tune’s harmonic structure. The end of a phrase is much more likely to be the end of a solo if it occurs at the end of a chorus rather than the middle. In both improvisation and conversation, group agreement on transition relevance places allows for smooth transitions between speakers with minimal overlap or gap.

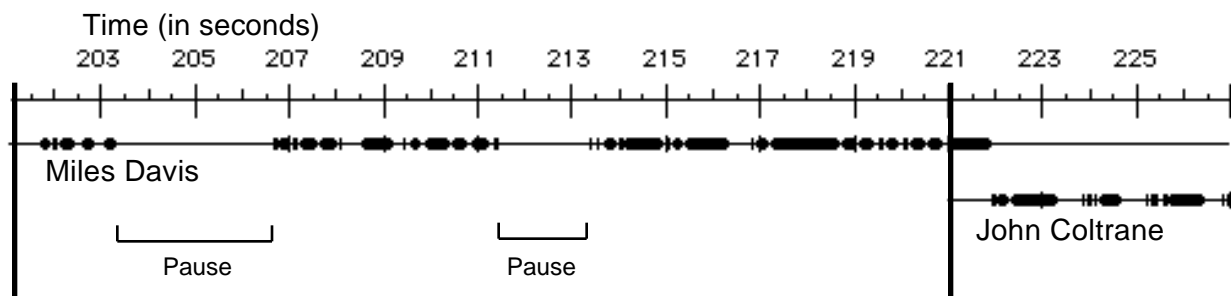


Figure 3.4. Time line showing solo phrases and pauses between them

Figure 3.4 is a time line⁴ representing an excerpt from Miles Davis’ “Freddie Freeloader”[13]. The last twenty seconds of Davis’ solo contains two gaps of several seconds that could imply the solo’s end. The actual end of the solo, however, comes shortly after the chorus boundary at 221 seconds. Davis’s final phrase spills over into the next chorus, and Coltrane begins immediately thereafter at time 222 seconds. The smoothness of this transition shows how Coltrane relied on the harmonic structure of the tune to project the end of the solo.

⁴Traditional music notation would have obscured the overall picture of musicians interacting, so I invented this notation. In the only paper I found showing a collaborational transcription of music, Peter Weeks explored an extension of traditional conversation analysis to document the interaction between musicians at an orchestra rehearsal[57]. However, he focused on collaborating about classical music during rehearsal, not on collaborating while improvising.

According to Sacks, speakers employ two strategies to allocate the next turn in a conversation. With the speaker-selects-next strategy, the speaker specifically indicates which listener should speak next. For example, the speaker may direct a question to a specific listener. With the self-selection strategy, a listener begins speaking at a transition relevance place. Jazz musicians use the same strategies to allocate solos. Sometimes soloists use a visual or verbal cue to indicate who should follow them; other times, they simply end their turn and leave allocation of the next solo to their colleagues.

3.2. Human-Computer Interaction

My second design premise is that HCI is best regarded as conversation. In particular, CA findings should be used to improve HCI, including interactive music systems. The general study of HCI includes a variety of modalities, including pictures, sounds, and text. Regardless of modality, researchers studying HCI have concluded that people regard interactive computer systems as social actors. In Lucy Suchman's study of human interactions with photocopiers, copier users construed their interactions as dialogues[51]. They expected the copier to take turns with them, to respond to their actions with appropriate reactions. When users failed to follow the copier's instructions for completing complex jobs, the copier failed to match its model of the user with input from its limited sensors. The copier ceased activity, and users became frustrated when the copier failed to "take its turn" as expected.

Perhaps the most famous example of computer as social actor is Weizenbaum's ELIZA program, which simulated a psychotherapist. Writing about HCI, Sherry Turkle notes[55]:

"Weizenbaum's students and colleagues who had access to ELIZA knew and understood the limitations on the program's abilities to know and understand. And yet, many of these very sophisticated users related to ELIZA as though it did understand, as though it were a person. With full knowledge that the program could not empathize with them, they confided in it, wanted to be alone with it." (p. 39)

If a photocopier or a simple natural language parser stimulates these kinds of expectations, it stands to reason that musicians will expect as much from a computer system that listens to what they play and reacts to it. The responsive, reactive nature of such a system encourages musicians to construe their interactions with it as dialogues. Furthermore, the system is attempting to engage in a musical collaboration with many conversational qualities. To improve the way these collaborations conform to human social expectations, interactive music systems should be informed by the sociology of conversation—conversation analysis.

3.3. A General Model of Collaboration

The many correlations between conversation and improvisation suggest they are two kinds of structured collaboration amenable to analysis by one general model. As discussed above, turn-taking can be used to explain both conversational and musical behavior. Hence, the general model uses the idea of conversational turn-taking to explain the more general issue of navigating through the structure of a collaboration. My general model for structured collaboration consists of the following elements.

- *Kinds of states*

A structured collaboration is characterized by passing through distinct states over its lifetime. Each state defines a role for every participant. For example, a “Trumpet Solo” state means the trumpet player solos while the other musicians accompany. In courtroom proceedings, states would include “Defense Attorney Asks Question” and “Witness Answers Question.” The particular states of a structured collaboration must be analyzed and made explicit when building a computer participant for that collaboration.

- *Connections between states*

A structured collaboration is further characterized by arcs connecting the states described above. The arcs add structure to the model by defining transitions between states. My model of jazz performance would not allow transitions from “Trumpet Solo” to “Ending” to “Restatement of

Melody,” nor would a courtroom proceedings model allow a transition from “Defense Attorney Asks Witness a Question” directly to “Prosecuting Attorney Asks Witness a Question” with no intervening states. Once the states of a structured collaboration have been identified, one can find the allowed state transitions by analyzing examples.

- *Transition relevance places*

Collaboration works best when all participants make a state transition simultaneously. To further this goal, participants agree to allow transitions only at certain points in the collaboration—transition relevance places. In speech, these occur at the boundaries of sentences; in music, at the boundaries of musical phrases. Since participants construct their utterances out of discrete units, the participants agree that state transitions occur only at the boundaries of these discrete utterance units.

- *Criteria for making transitions*

Upon reaching a transition relevance place, all participants apply the same criteria and all arrive at the same decision. These criteria typically relate to utterances of certain participants or to other relevant information. Ideally, all participants mark a transition within their own mental copies of the state machine, leading each of them to a new role in the new state. For example, when the prosecuting attorney finishes uttering a syntactically complete question, all the courtroom participants should agree on a transition from “Prosecuting Attorney Asks Question” to “Witness Answers Question,” a new state in which the witness becomes the new speaker. For a computer to participate successfully in a structured collaboration, it must be given an accurate algorithm to determine whether a transition is taking place.

3.4. Informing Framework Design

Conversation analysis reveals three keys to participating successfully in conversation. Each contributes to the design of ImprovisationBuilder, a framework for interactive music systems. In both improvisation and conversation, a participant must:

- Engage in the distinct tasks of listening to other participants, composing new utterances, and realizing⁵ the new utterances. These distinct aspects of conversation provide a logical decomposition of the task of improvisation.
- Follow both changing participant roles and the collaboration's evolving structure. Protocols that model the flow of conversations become resources for structured improvisations.
- Realize the utterances it composes in synchrony with the other participants. CA asserts the importance of timing to collaboration.

My goal was to translate these three principles into an interactive music system. The first principle led to the construction of an *ImprovisorComponent* class hierarchy, with abstract classes *Listener*, *Composer*, and *Realizer*. The second principle led to the development of *ImprovisorStates* that represent improvisational structure. The third principle prompted considerable work on a reliable real-time *MusicScheduler* and highly optimized Smalltalk primitives for music input and output.

⁵In this context, “realize” means “to bring into concrete existence” rather than “to be fully aware of”[62]

4. An Object-Oriented Framework

Given that good design is the hardest part of building software, a design is more valuable if it can be reused in multiple implementations. I have attempted to achieve this reusability by expressing my design as ImprovisationBuilder (IB), an object-oriented framework. Frameworks seek to capture designs in a manner that maximizes their potential for reuse. In their survey of object-oriented programming research, Rebecca Wirfs-Brock and Ralph Johnson define an object-oriented framework as having two essential parts[61]:

- *A reusable design expressed as a set of abstract classes.* IB defines abstract component classes corresponding to the fundamental tasks of improvising and an *Improvisor* class to act as a container and coordinator for the components. IB also provides *ImprovisorStates* to represent different states of an improvisation, *Chords* and *Phrases* to represent music, and numerous support classes such as a real-time *MusicScheduler*.
- *A description of how instances of those classes collaborate.* IB's *Improvisor-Components* are connected in linear chains and collaborate by passing musical representations (*MusicMessages*, *Chords*, *Phrases*) along the chain. They also collaborate by storing shared information in a *PolicyDictionary*.

By providing a library of interactive music system components, IB facilitates rapid prototyping and testing[21]. Users construct their own systems by configuring ensembles of existing components. Users can also create new components based on the protocols defined in IB's abstract classes (see Figure 4.2). New components inherit code provided by the abstract class and implement only the unique aspects of their behavior. By following a standard protocol, new

components substitute interchangeably for existing ones. IB also provides a musical representation used by all components.

The architecture of IB closely resembles that of a framework developed at Tektronix for oscilloscope software[14]. Both frameworks can be understood as three layers of abstraction. The bottom layer contains the representation of objects in the domain (musical phrases or electrical signals). The middle layer describes components for manipulating these representations (IB's *ImprovisorComponents*). The top layer describes mechanisms for dynamically reconfiguring the components in the middle layer (IB's *ImprovisorStates*). Reuse of such a framework comes from devising new configurations of or adding components to the upper two layers.

Figure 4.3 shows a typical IB configuration. *ImprovisorComponents* are composed sequentially, with a *Listener* at the start and a *Timbre* at the end. *Listeners* process the incoming music, parsing it into phrases and focusing the system's attention. *Transformers* and *Composers* create new phrases, either by transforming phrases supplied by the listener or through some compositional algorithm. *Realizers* express the phrases appropriately, both through timely presentation and by controlling *Timbres* that represent sound generating hardware. An *Improvisor* coordinates all the components and their parameters, as well as a *PolicyDictionary* of communal information. Figure 4.3 also shows the *MusicScheduler* and other support components that exist independently of the *Improvisor*. They connect any number of *Improvisors* to input and output devices and handle scheduling details.

The previous chapter enumerated correlations between conversation and improvisation. From that list, *ImprovisationBuilder* addresses the following aspects of structured collaboration:

- *The structure of what participants produce*

IB's hierarchical musical representation models the structure of music generated within systems of harmonic and rhythmic constraints.

- *What participants predetermine*

IB is designed to interact with other musicians, providing compositional algorithms with the information necessary to operate within an evolving musical structure.

- *How participants synchronize their conduct*

Realizers and *Listeners* use the musical output and input to ensure synchronous behavior.

- *How participants coordinate their behavior*

The *ImprovisorStates* and *Listeners* define a model of the structured collaboration and the algorithms governing navigation through the structure.

- *Visual communication*

Extra-musical communication between performers is modeled with a set of keyboard commands for informing the computer of changes in the improvisation.

Class	Input	Output
<i>ChordStream</i>	<i>MusicMessages</i>	<i>Chords</i>
<i>Listener</i>	<i>Chords</i>	<i>Phrases</i>
<i>Transformer</i>	<i>Phrases</i>	<i>Phrases</i>
<i>Composer</i>		<i>Phrases</i>
<i>Realizer</i>	<i>Phrases</i>	<i>Chords</i>
<i>Timbre</i>	<i>Chords</i>	<i>MusicMessages</i>

Figure 4.1. *ImprovisorComponents* by input and output type

The various *ImprovisorComponents* convert between and manipulate the musical representations (see Figure 4.1). The following sections describe interactions among these components. Section 4.1 describes how IB represents music. Section 4.2 shows how IB parses input from other performers. Section 4.3 describes IB's conversation-based representation for improvisational structure. Section 4.4 shows how musical phrases are transformed or created algorithmically. Section 4.5 shows how IB coordinates shared information. Section 4.6 describes the process of realizing musical phrases.

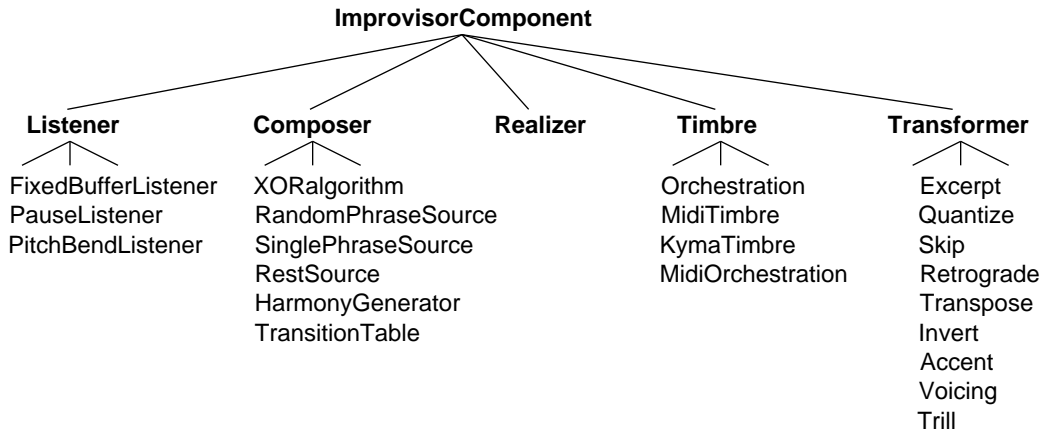


Figure 4.2. *ImprovisorComponent* class hierarchy

4.1. Music Representation

IB’s music representation serves the same purpose as the conversation transcripts present in CA papers. Those transcripts retain detailed information about the timing of events while preserving hierarchical information such as phrases and sentences. IB accomplishes the same goal with three levels of representation. At the lowest level, the *MIDIInputChannel* represents a performer’s musical output as a stream of time-tagged *MIDIMessages* denoting the beginnings and endings of notes.

MIDIMessages are the subset of the *MusicMessage* class hierarchy that enable communication with devices that conform to the Musical Instrument Digital Interface (MIDI) standard. MIDI devices communicate using messages that start notes, stop notes, and select timbres. The computer sends and receives these messages over an input and an output port, each of which contains sixteen channels. The sixteen output channels can each control a distinct synthesizer timbre, while the sixteen input channels can each supply data from a different performer. In Smalltalk, each subclass of *MIDIMessage* corresponds to a message defined in the MIDI protocol—*NoteOn*, *NoteOff*, and *PatchChange*, for example.

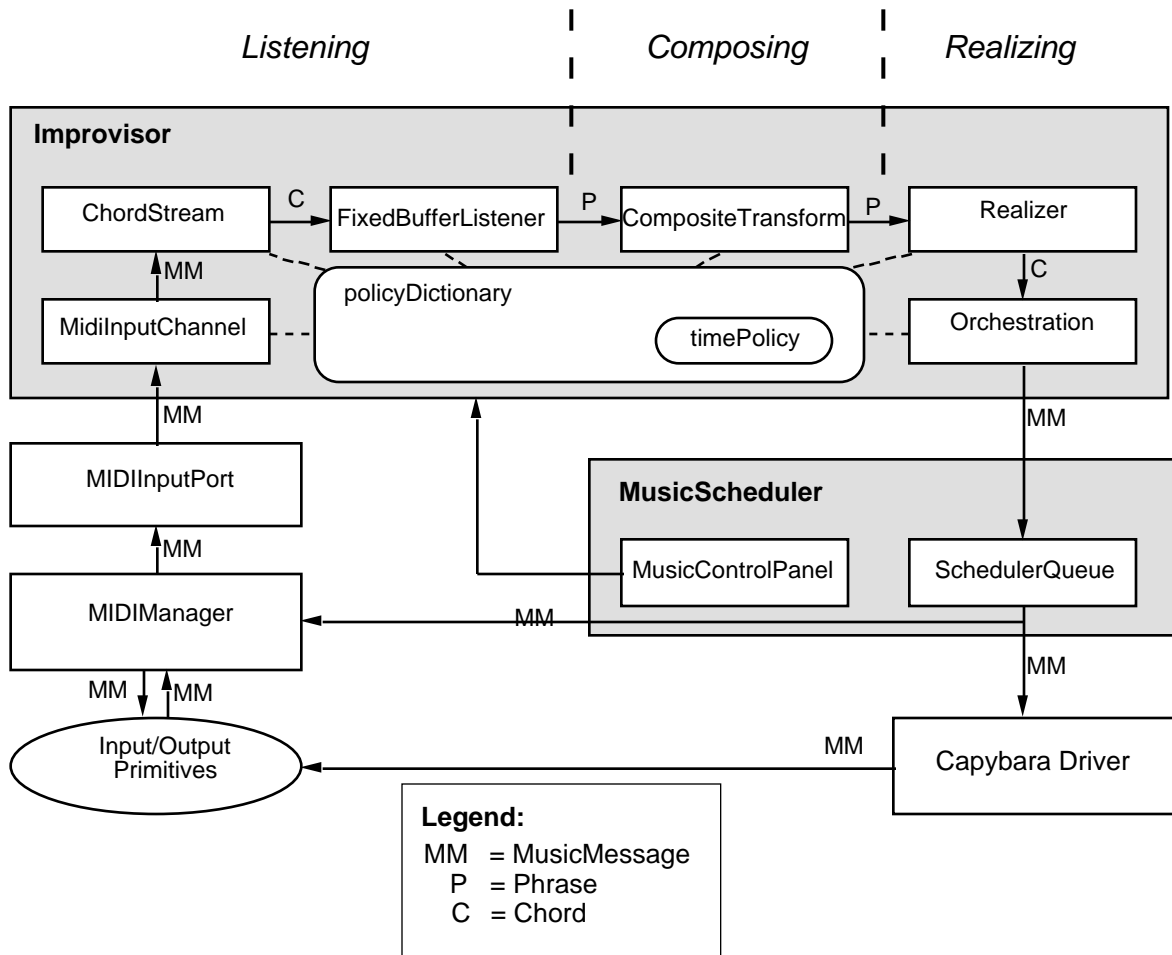


Figure 4.3. Typical ImprovisationBuilder configuration

The second level of music representation is the *Chord*, which specifies an array of pitches, a duration, and a volume. The *Chord* class differs from the standard musical notion of a chord in several ways. Since a *Chord* may have zero or more pitches, it encompasses the notion of rests and single notes in addition to the conventional notion of two or more simultaneous pitches. Since it represents all the notes sustained by a performer over a particular time interval, it may encompass pitches that would be notated as separate chords in standard music notation. Note that, unlike *NoteOns* and *NoteOffs*, *Chords* do not contain explicit start times. The start times for a sequence of *Chords*, however, can be computed from their durations.

Class	Role	Collaborations
<i>MusicMessage</i>	Represents lowest level musical events (i. e., note on, note off)	Maintains pointer to its mate (i. e., corresponding note on or note off)
<i>Chord</i>	Represents pitches sustained for a certain duration at a certain volume	Arranged in a linked list with other <i>Chords</i>
<i>Phrase</i>	Represents a musical gesture	Can be part of a linked list of <i>Phrases</i>

Figure 4.4. Classes for music representation

Figure 4.5 shows two overlapping notes, each represented by a *NoteOn* message and a *NoteOff* message. IB represents the temporal intervals between these messages as *Chords*. Thus, the two notes give rise to three *Chords*, representing the onset of the first note, the overlap of the two notes, and the remainder of the second note, respectively. When several *NoteOns* occur at the same time, their volumes are averaged to compute a volume for the *Chord*. This single *Chord* volume has proven sufficient for present purposes.

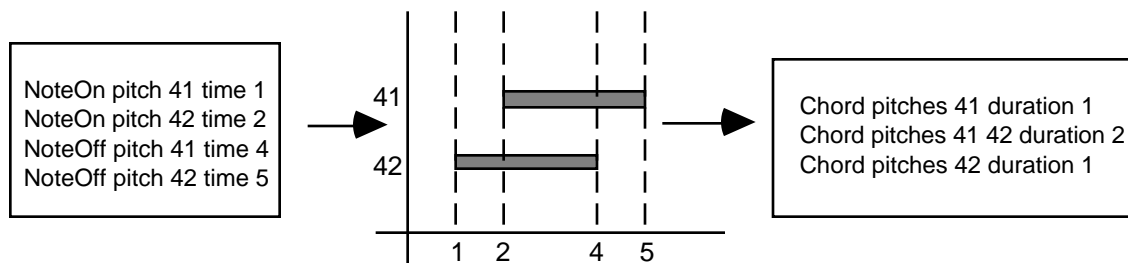


Figure 4.5. *MIDI*Message and *Chord* representations

The third level of representation is the *Phrase*, which contains a sequence of *Chords*. In addition to acting as an ordered collection of its *Chords*, *Phrase* maintains information about the *Chords* (such as the sum of their durations) and defines some collective operations (such as copying and iteration). By controlling access to the sequence of *Chords*, the *Phrase* ensures internal consistency. For example, a *Phrase* ensures that the last *Chord* in its collection is always a rest, to ensure that all notes have a well-defined duration.

Streams of musical events can be represented in many ways. However, the structure of a note list representation determines the complexity of manipulating it. Representing a *Phrase* as a stream of *Chords* simplifies common transformations such as *Retrograde*. This representation also

streamlines music playback, since a *Timbre* can easily convert a stream of *Chords* into a time-ordered stream of *MusicMessages*.

MusicMessages, *Chords*, and *Phrases* often represent music generated within certain rhythmic constraints. IB's *TimePolicy* classes explicitly represent these rhythmic constraints. When an improvisation has meter but no clear harmonic structure, each *Improvisor* contains a subclass of *TimePolicy* called *MetricalTime*. *MetricalTime* contains a beat size in milliseconds, the number of beats per measure, and an offset specifying the start of the first beat. When there is a fixed chord structure, a subclass of *MetricalTime* called *ChordChanges* provides additional information about what underlying harmony is in effect for each beat.

4.2. Parsing Musical Input

IB's input components parse a stream of low-level *MusicMessages* into a stream of musically useful *Phrases*. This process involves two steps. First, a *ChordStream* processes the *MusicMessages* from the *MIDIInputChannel* into *Chords*. The *ChordStream* creates a new *Chord* for each new *MusicMessage*. Since the *ChordStream* performs a straightforward conversion between representations, it is reused in all IB configurations. Second, a *Listener* groups the *Chords* from the *ChordStream* into *Phrases* based on the *Chords*' pitches and durations.

Class	Role	Collaborations
<i>MIDIInputPort</i>	Buffer <i>MusicMessages</i> from driver	Supply <i>ChordStream</i>
<i>ChordStream</i>	Convert <i>MusicMessages</i> to <i>Chords</i> Compute information about input	Supply <i>Listener</i> Report information to <i>Improvisor</i>
<i>Listener</i>	Convert <i>Chords</i> to <i>Phrases</i>	Supply <i>Phrases</i> to <i>Transformer</i>
<i>Improvisor</i>	Configure components Route parameter changes	Receive commands from <i>MusicControlPanel</i> Receive updates from <i>ImprovisorComponents</i>

Figure 4.6. Classes for parsing input

The *Listener* connects the input components to the rest of the system. Typically, a *Transformer* periodically polls the *Listener* for its next *Phrase*. In such a situation, the *Listener* determines which portions of the MIDI input stream the *Transformer* can process. A *PauseListener* detects silences longer than two seconds and begins a new *Phrase* after each pause, placing each incoming *Chord* into exactly one *Phrase*. In contrast, a *FixedBufferListener* stores the incoming *Chords* in a fixed size buffer and converts the buffer's contents into a *Phrase* when the *Transformer* needs a *Phrase* to transform. If a *FixedBufferListener* receives too many *Chords* before it produces a *Phrase*, it discards the oldest *Chords*. Thus, a *Transformer* connected to a *PauseListener* will have a chance to transform all the notes played by the human performer, while a *Transformer* connected to a *FixedBufferListener* may not. In other words, the *Listener* focuses the system's attention and defines its attention span (see Section 1.1.2 for a discussion of interaction techniques).

4.3. Representing Improvisational Structure

Conversation Analysis asserts that people assume different roles during conversation (such as speaker and listener), and that they use various collaborative cues to negotiate transitions between roles[20, 47]. Some specialized verbal interactions develop intricate collaborative structures, such as the ones studied by Heath and Luff in the London Underground control room[24]. This idea formed the basis of *ImprovisationBuilder*'s representation for collaborative structures in improvisation (see Figure 4.7).

Class	Role	Collaborations
<i>ImprovisorState</i>	Defines role for <i>Improvisor</i> Defines entry into and exit from state	Governs behavior of <i>Composers</i> , <i>Transformers</i> Monitors information from <i>Listener</i>
<i>Improvisor</i>	Contains state machine	Supervises transitions between <i>ImprovisorStates</i>
<i>Listener</i>	Scans input	Supplies information relevant to state transitions

Figure 4.7. Classes for representing improvisational structure

Each *ImprovisorState* performs two important tasks. The first is to control the *ImprovisorComponents* so that the *Improvisor* produces output appropriate to its role in the improvisation. When the *Improvisor* enters an *ImprovisorState*, the *ImprovisorState* changes the parameters of the *ImprovisorComponents*. For example, entering a *Solo* (see Section 5.2.2) causes the *Solo* to instruct the *HarmonyGenerator* to generate melody and chordal accompaniment. Conversely, entering an *Accompany* state causes the *HarmonyGenerator* to construct a walking bass line. *ImprovisorStates* can send parameter changes to any *ImprovisorComponent*, allowing them to represent many different kinds of interaction.

The second task of an *ImprovisorState* is to define its exit condition. When the *Listener* computes new information about the human musician's performance, the *ImprovisorState* examines them to determine whether the current role is ending. If so, the *Improvisor* makes a transition into the next *ImprovisorState* in its state machine. For example, *Accompany* interprets low notes played by the human musician to mean it should relinquish the role of accompanist (see Section 5.2.2). Since these exit conditions can draw on information about the input as well as information in the *PolicyDictionary*, the *ImprovisorState*'s duration may depend on many factors, thus allowing it to model many kinds of improvisational structure.

4.4. Generating Musical Output

Class	Role	Collaborations
<i>Transformer</i>	Transform <i>Phrases</i>	Supply <i>Realizer</i> or other <i>Transformer</i>
<i>Composer</i>	Make new <i>Phrases</i>	Supply <i>Realizer</i> or <i>Transformer</i>
<i>KeyboardMusicController</i>	Defines keyboard commands	Sends parameter changes to <i>Improvisor</i>
<i>Improvisor</i>	Maintain dictionary of components and parameters	Configure components Route parameter updates to components

Figure 4.7. Classes for designing output

Having parsed input from the other performers, the system's next task is to generate its own musical contribution. IB generates music with *Composers*, *Transformers*, or a combination of

the two. *Composers* contain a musical algorithm that will create new *Phrases*. *Transformers* transform the output of a *Listener*, *Composer*, or another *Transformer*. Varying the parameters of *Composers* and *Transformers* can tailor them to specific musical situations. The human performer sets these parameters during performance by means of keyboard commands sent to the *KeyboardMusicController*. *Composers* and *Transformers* can also derive their parameters from the *Improvisor's PolicyDictionary*, a repository for shared information about the improvisation.

4.4.1. *Transformer*

Martirano's Sound and Logic program has been the basis for many of the *Transformers* currently in the system (see Figure 4.9). Simple *Transformers* (such as *Voicing* or *Skip*) produce one *Chord* from each *Chord* in the source *Phrase*. Other transformers (such as *Trill* and *GraceNote*) turn a single *Chord* into several (see Figure 4.10). While most transformers produce one transformed *Phrase* from each input *Phrase*, *Excerpt* caches each input phrase and produces several excerpts from it.

Name	Description
<i>Inversion</i>	Inverts the <i>Phrase's</i> pitch contour
<i>Retrograde</i>	Reverses the entire <i>Phrase</i> in time
<i>Skip</i>	Turns some <i>Chords</i> with one or more pitches into rests
<i>Transposer</i>	Shifts all pitches in the <i>Phrase</i> by a constant offset
<i>Trill</i>	Turns long single notes into rapid alternation between adjacent pitches
<i>Voicing</i>	Rearranges chord pitches according to a cyclic permutation scheme

Figure 4.9. Standard *Transformers*

Each *Transformer* has a probability parameter governing the likelihood of transforming each source *Phrase*. Some *Transformers* use additional parameters to govern their behavior. For

example, *Transposer* contains a set of weighted transposition intervals. The *Transposer* chooses one interval for each *Phrase* it transforms. While the standard *Transformers* are individually simple, applying several *Transformers* in series to the output of a *Listener* can yield *Phrases* that are both musically interesting and quite different from the original.

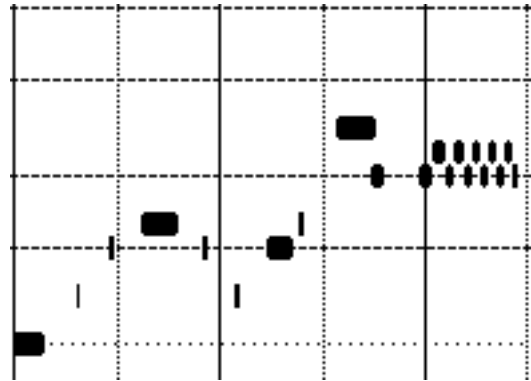
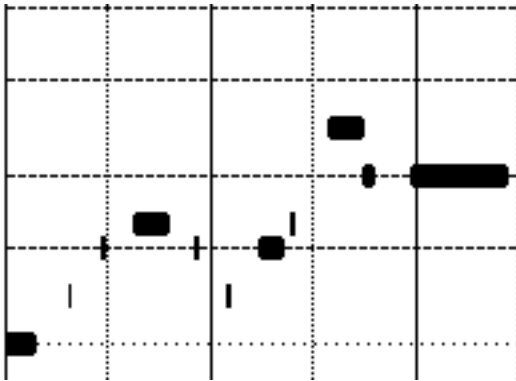
4.4.2. *Composer*

Composers create new *Phrases* by means of compositional algorithms. The existing *Composer* classes implement a variety of algorithms. *HarmonyGenerator* uses a library of chord voicings and rhythmic templates to create accompaniment or solo parts for a given set of chord changes. It serves as the basis for participating in jazz improvisation, and is discussed at greater length in Section 5.2. *RandomPhraseSource* generates completely random *Phrases*, which are used for testing purposes. *DrumPatternBuilder* uses the same rhythmic templates as *HarmonyGenerator* to produce drum patterns with random variations. *TransitionTable* uses Markov chains to create material (see Section 1.1.1 for descriptions of other generative techniques).

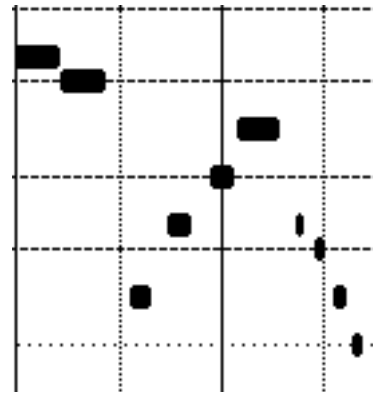
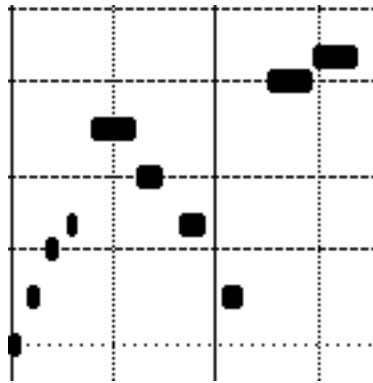
Before

After

Trill



Retrograde



Transposition

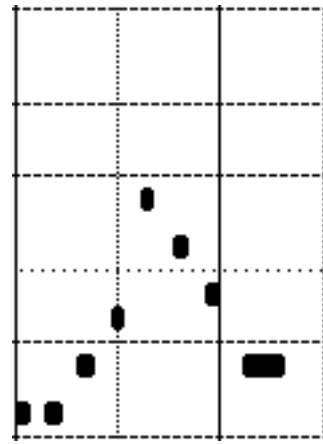
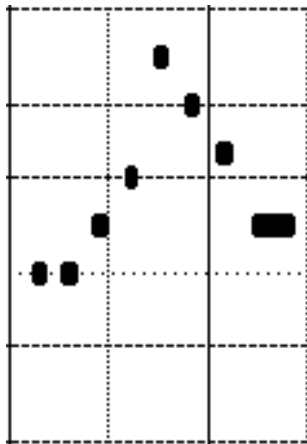


Figure 4.10. Sample transformations

Users can create *Composer* subclasses to implement their own compositional algorithms. IB simplifies the design and testing of such algorithms, providing both a device-independent music representation and a real-time scheduler. More significantly, the *Composer* abstraction provides an environment in which compositional algorithms react in real-time to input from human performers.

4.5. Coordinating Shared Information

In conversation, information gathered while listening governs how and when to respond to other participants. IB models this kind of information sharing with a centralized *PolicyDictionary*. The *PolicyDictionary* contains information independent of any single component. For example, *MetricalListener* stores information about its input in the *PolicyDictionary*. *ImprovisorState* and *HarmonyGenerator* use this information to determine what kind of output to generate (see Section 5.2).

Either the *ImprovisorComponents* or the *MusicControlPanel* can send information to the *PolicyDictionary*. Through the *MusicControlPanel*, the human performer can alter the parameters of any *ImprovisorComponent* or configure the connections between components. A *DispatchTable* maps computer keyboard commands to messages that are sent to *Improvisors* and their components. Rather than processing these messages immediately upon detecting the key press, the *MusicControlPanel* places them in a *MusicScheduler* queue. The *MusicScheduler* processes them during its idle time, so that keyboard commands don't interfere with the real-time musical output (see Section 6.2).

4.6. Realizing Musical Output

The final task of the *ImprovisorComponent* chain is to convert the *Phrases* generated by *Transformers* and *Composers* into actual sounds. This performance involves a series of components (see Figure 4.11) designed to control sound hardware efficiently.

Class	Role	Collaborations
<i>Realizer</i>	Convert <i>Phrases</i> into <i>Chords</i>	Provides source for <i>Timbre</i>
<i>Timbre</i>	Convert <i>Chords</i> into <i>MusicMessages</i>	Provides source for <i>MusicScheduler</i>
<i>MusicScheduler</i>	Sends <i>MusicMessages</i> to synthesizers Merges output from <i>Timbres</i>	Poll <i>Timbres</i> for next message, pick earliest

Figure 4.11. Classes for realizing output

Producing sounds from *Phrases* begins with a *Realizer*. The *Realizer* takes *Phrases* from a *Composer* or *Transformer* and supplies them one *Chord* at a time to the *Timbre* for playback. The *Realizer* can use information from the *PolicyDictionary* to withhold or alter these *Chords*. For example, the *Realizer* may scale *Chord* durations to make them conform to a changing *TimePolicy*. The *Realizer* can also flush its current *Phrase* at any time, allowing the system to react quickly when *ImprovisorComponents* are reconfigured.

The *Timbre* processes *Chords* from the *Realizer*, generates a corresponding stream of *MusicMessages*, and sends them to the *MusicScheduler* for playback. Subclasses of *Timbre*, such as *MIDITimbre* and *KymaTimbre*, isolate code that depends on particular sound hardware. *Orchestration* is a special *Timbre* that realizes *Chords* with combinations of other *Timbres* (see Figure 4.12).

Timbre is the only *ImprovisorComponent* that maintains a time clock. As a *Timbre* processes each *Chord*, it increments the clock by the *Chord*'s duration. *Timbres* perform two other kinds of clock adjustment. First, in a metrical improvisation, the *Timbre* will align its internal clock to the current metrical structure as defined by the *MetricalTime*. Second, a *Timbre* may advance its clock when it falls behind real time. This postpones the next output event, allowing the other components more time to catch up.

The *Realizer* determines whether to allow clock advances. This facility is disabled for performances with a fixed harmonic structure; were the *Realizer* to advance its clock one measure while improvising over changing harmonies, the computer improviser would thereafter be one measure behind the other performers. In any case, a *Timbre* sometimes sends *MusicMessages* to

the *MusicScheduler* after they should have been played. Each subclass of *MusicMessage* acts differently in this situation. Late *NoteOns* will disable themselves to prevent a flurry of late notes as the system catches up. Late *NoteOffs* are always sent to the synthesizer; otherwise, some notes might sustain indefinitely.

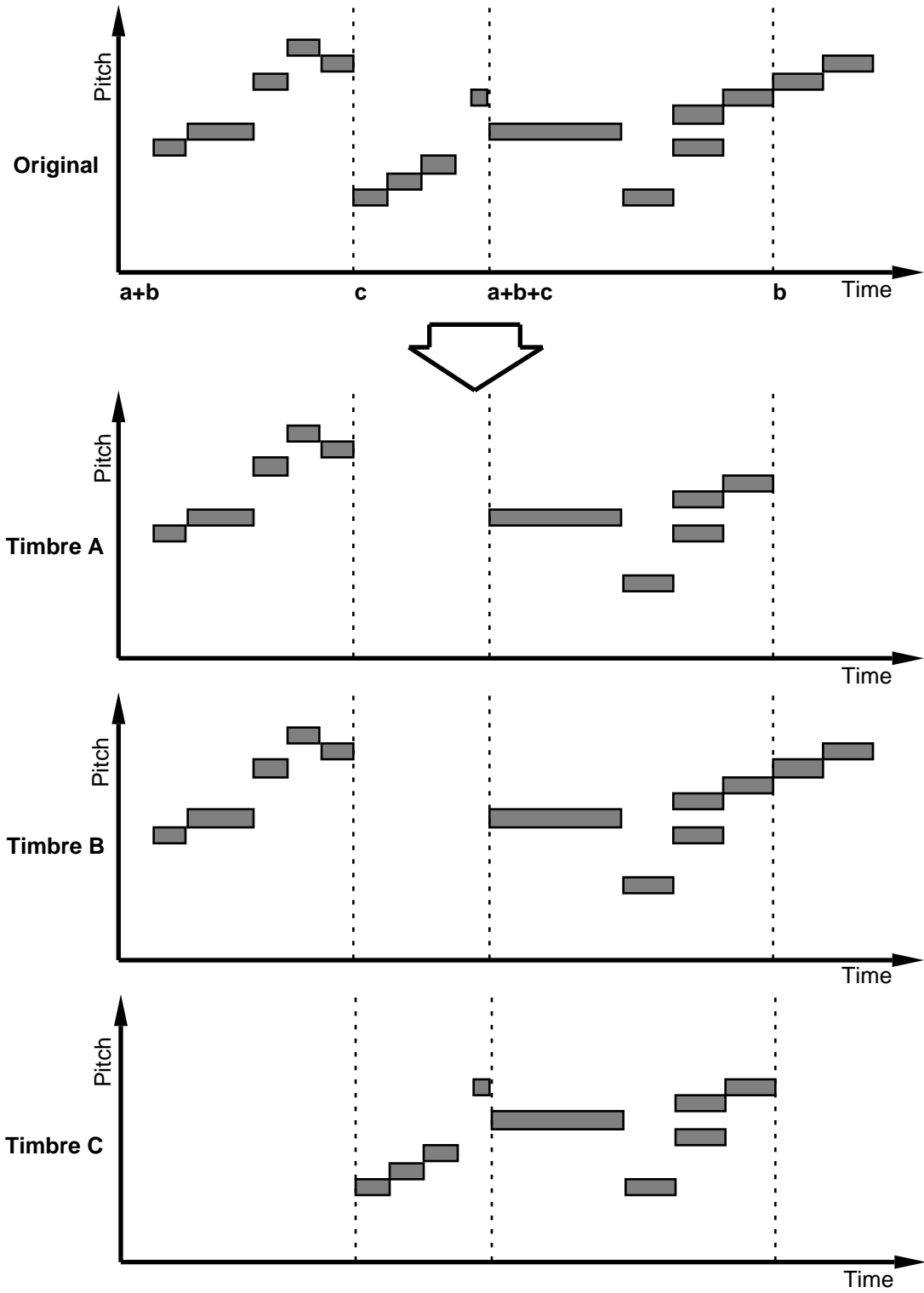


Figure 4.12. *Orchestration with three Timbres*

5. Applying the Framework

An IB user builds a configuration for his or her improvisation by instantiating *ImprovisorComponents* and installing them in an *Improvisor*. Figure 5.1 shows code to construct a *PauseListener*, a *ScalarTranspose*, and a *MIDITimbre*, install them in an *Improvisor*, and open the *MusicScheduler*. This code also defines an initial set of transposition intervals and specifies that the *Realizer* may postpone notes if it falls behind real time. The *ImprovisationBuilder* class acts as a repository for configuration code.

```
pauseListenerTest
| aScheduler anImprovisor |

anImprovisor := Improvisor
                with: (PauseListener on: 1)
                with: (ScalarTranspose new scale: #(0 2 4 5 7 9 11))
                with: (MIDITimbre on: 1).

self defaultParametersTo: anImprovisor.
anImprovisor
    setParameter: #intervals
    toValue: (StepFunction withValues: #(1 2) weights: #(0.5 0.5)).
anImprovisor setParameter: #permitSlipping toValue: true.

aScheduler := MusicScheduler with: anImprovisor.

MusicScheduler open: aScheduler withDispatchTable: self buildDemoDispatchTable
```

Figure 5.1. Sample IB configuration code

There are three ways to customize an IB configuration. First, the user can construct a new state machine, thus defining a new structure for improvisation. The state machine can contain instances of both predefined and new *ImprovisorStates*. When defining a new *ImprovisorState*, the crucial methods to write are **enterState**, which send parameter changes to the *ImprovisorComponents* upon entering the state, and **changeState:**, which periodically examines the *PolicyDictionary* and determines whether to advance to the next state. *ImprovisorStates* are connected by setting their **nextState** fields, and by informing the *Improvisor* of its **startState**.

Second, the user can write new components, thus modeling new musical styles or ways of interacting. Figure 5.2 shows which methods are commonly redefined when constructing various kinds of *ImprovisorComponents*. In many cases, a new component can inherit most of its behavior, leaving the user to write one or two Smalltalk methods. To construct a new kind of *Transformer*, for example, the user need only implement the **transform:** method.

Name	Method	Method Description
<i>Listener</i>	nextPhrase	Takes <i>Chords</i> from <i>ChordStream</i> and creates <i>Phrase</i>
	isReady	Answers whether the <i>Listener</i> is ready to supply a <i>Phrase</i>
<i>Transformer</i>	transform: aPhrase	Transforms argument <i>aPhrase</i> to produce a new <i>Phrase</i>
<i>Composer</i>	nextPhrase	Generates a new <i>Phrase</i> on demand

Figure 5.2. Methods to write when constructing new *ImprovisorComponents*

Third, the user can control the components' parameters, thus tailoring the components to particular musical situations. Each *ImprovisorComponent* responds to the **parameters** message with descriptions of its parameters. These descriptions are stored in the *Improvisor's* *PolicyDictionary*. To set the initial values of these parameters, the user sends the **setParameter:toValue:** message to the *Improvisor* with the parameter name and desired value. In Figure 5.1, the 'intervals' parameter is set to a probability distribution that equally weights the values one and two, and the 'permitSlipping' parameter is set to true.

The user can alter these parameters during performance by defining new keyboard commands. A *DispatchTable* defines IB's keyboard commands. A new user configuration starts with the minimal command set and adds appropriate commands. The code to add these commands is kept in class *ImprovisationBuilder*.

```

addOrchestrationMinMaxControlsTo: dispatchTable
  dispatchTable
    bindValue:
      (Message
        selector: #setParameter:toValue:
        arguments: (Array with: #numberOfChords with: (3 to: 5)))
      to: $[ followedBy: $1.

```

Figure 5.3. Defining a new keyboard command

Figure 5.3 shows code to bind the key sequence ‘[’ ‘1’ to a *Message* that sets the parameter ‘numberOfChords’ to the range 3 through 5. This allows the user to control how many chords the Orchestration will play before switching to a new set of Timbres (see Figure 4.11). This general-purpose mechanism allows the user to bind any one or two key sequence to any parameter change. Notice that the **setParameter:toValue:** used here is the same message used to set initial values.

Some hypothetical examples will further illustrate what kind of programming is involved and what kinds of interactions can be created:

- *Ostinato with adaptive volume*

During a performance, the computer’s role may be to repeat a musical phrase, matching its own volume to that of the human performer. This allows the human performer to control the loudness of the ensemble through his or her own loudness.

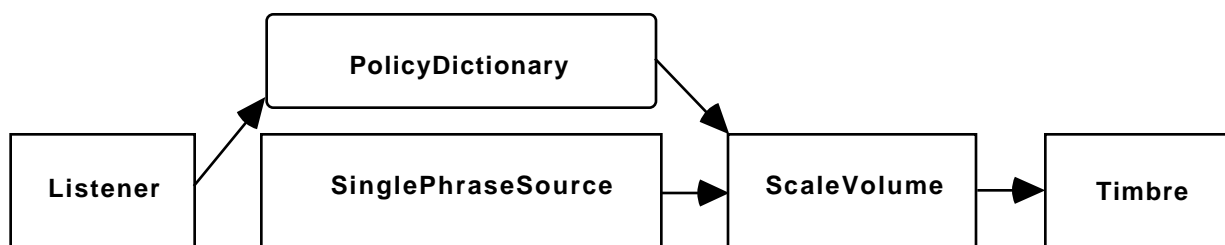


Figure 5.4. Ostinato with varying loudness

Figure 5.4 shows one implementation of this interaction, in which a *SinglePhraseSource* provides copies of the ostinato *Phrase* to a *ScaleVolume*, which scales the volume of its input. The scale factor is determined by a *Listener*, which would contain an algorithm for determining the average volume of the input. Since the *SinglePhraseSource* always responds positively to the **isReady**

message, the *Timbre* will continuously play the scaled copies of the ostinato *Phrase*. Of these components, the framework user would write only the *Listener*.

- *Pre-set sequence with musical trigger*

In a performance containing both improvised and pre-composed material, it may be desirable for the computer to playback some pre-composed material upon receiving a specific musical cue (such as a particular pitch).

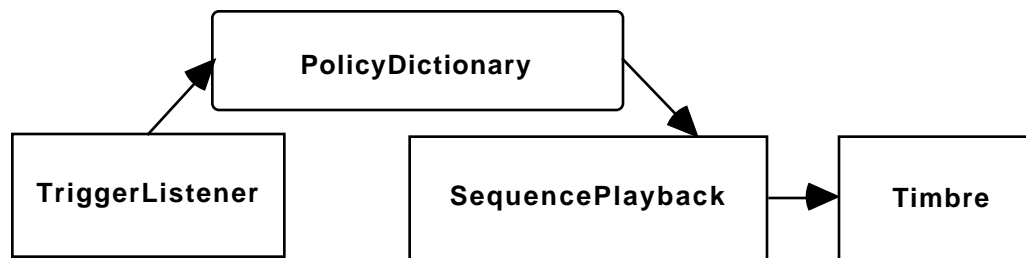


Figure 5.5. Pre-set sequence with musical trigger

Two crucial components in the Figure 5.5 solution to this problem are the *TriggerListener* and *SequencePlayback*. The *TriggerListener* would examine the incoming *Chords* and, upon detecting the musical trigger, send a message to the *PolicyDictionary*, which would forward that message to the *SequencePlayback*. The *SequencePlayback* would contain the pre-composed musical sequence in IB's music representation. Prior to receiving the trigger, it would respond negatively to the **isReady** message (see Section 6.4). As soon as the trigger was received, the *SequencePlayback* would assert its readiness and provide its *Phrases* to the *Timbre* for playback.

The framework user would write the *SequencePlayback* component (which would be very similar to the *SinglePhraseSource* in the previous example) and the *TriggerListener*.

- *Drum patterns derived from user input*

If the computer is attached to a synthesizer capable of generating drum sounds, it may be desirable to generate drum accompaniment patterns from the user input, rather than relying on pre-programmed material. Achieving this goal involves two components as shown in Figure 5.6.



Figure 5.6. Drum patterns derived from user input

The *MetricalListener* quantizes the rhythms of the user input, yielding more consistent rhythms. Drum synthesizers typically assign different drum sounds to different pitches. For a *Phrase* to function as a drum pattern, it should contain only those pitches to which the drum synthesizer has assigned drum sounds. Thus, the second step is to take these quantized *Phrases* from the human performer and map all the possible pitches onto those pitches that the drum machine has assigned to its drum sounds. The *PitchMap* provides a general, table-driven pitch mapping that can serve this purpose. Several pitch mappings could be selected from the computer keyboard during performance, allowing for a variety of drum accompaniments. The framework user's only task would be to construct the appropriate pitch mappings for the drum synthesizer being used.

The preceding chapter gave an overview of the framework components and how they interact. This chapter shows how I reused that framework design in constructing computer participants for the two musical domains discussed in Chapter Two. In each case, I designed some new components and combined them with existing ones to build working systems.

5.1. Sound and Logic Case Study

ImprovisationBuilder (IB) began as a Smalltalk-80 generalization of Martirano's Sound and Logic (SAL) and evolved into a framework. Martirano designed SAL for two human performers (a keyboard player and violinist) with two additional parts generated by the computer. In SAL80, the IB implementation of SAL, this structure becomes a pair of *Improvisors* (see Figure 5.4). Where the IB jazz configuration generates new musical material, SAL80 transforms material from the human performer. Within this transformational context, the computer's role is defined in three ways.

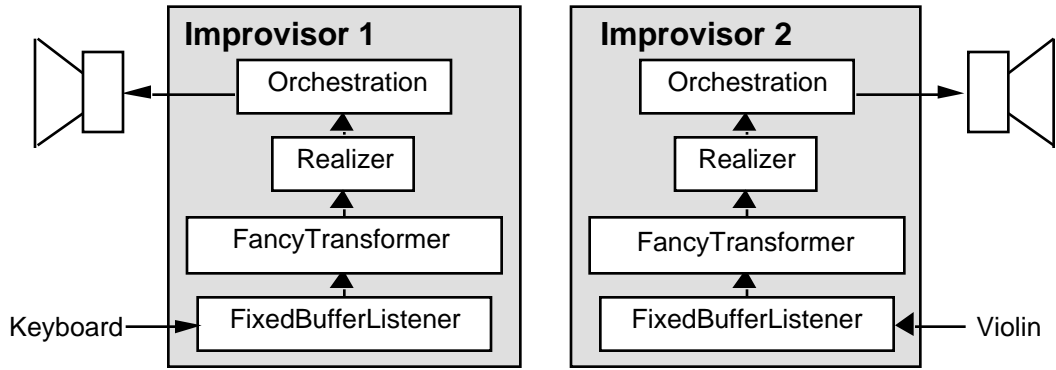


Figure 5.7(a). Sound and Logic 80 duet configuration

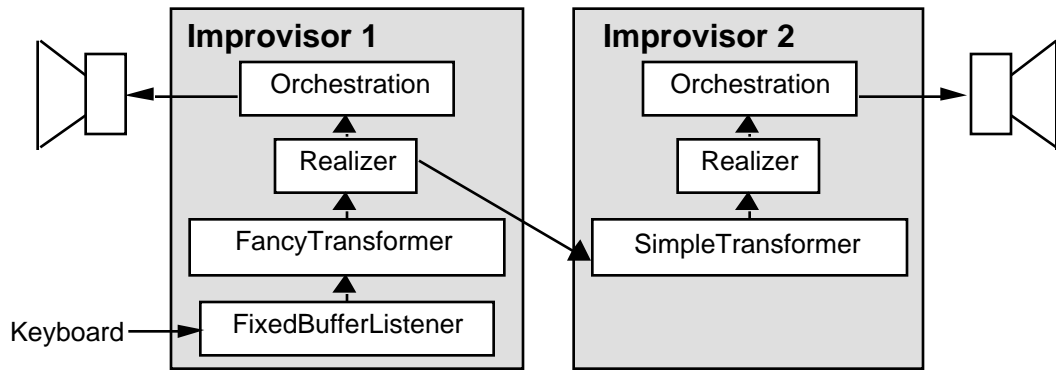


Figure 5.7(b). Sound and Logic 80 solo configuration

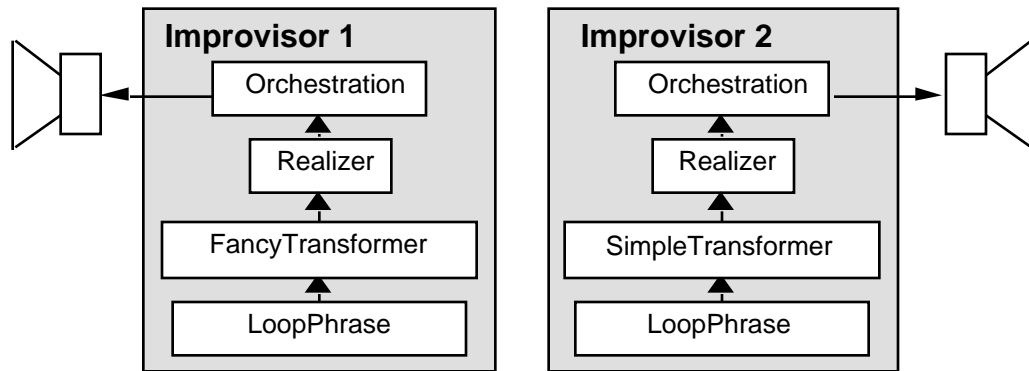


Figure 5.7(c). Sound and Logic 80 looping configuration

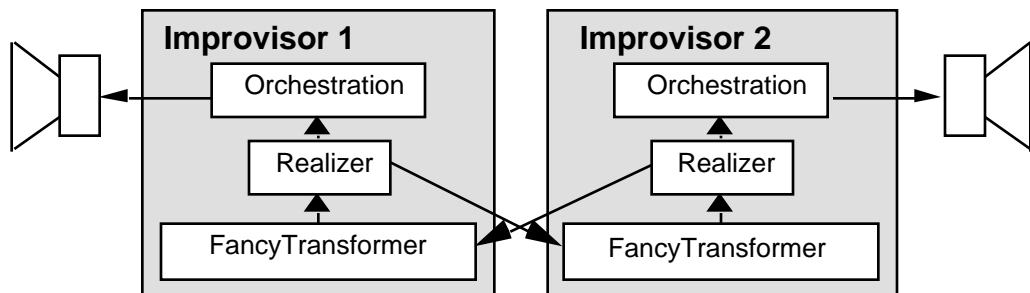


Figure 5.7(d). Sound and Logic 80 navel gazing configuration

- *Source for Transformers*

Over the course of the performance, the human musicians alternate between each playing alone with the computer and playing together with the computer. Accordingly, SAL80 can transform the output of either or both performers. In the latter case, the computer assigns one *Improvisor* to each human performer (Figure 5.7(a)). In the former case (Figure 5.7(b)), the two *Improvisors* are arranged sequentially, with one processing input from the human player and the second processing input from the first. The *MusicControlPanel* provides keyboard commands to switch between these configurations.

SAL80 collects the original SAL transformations into a *CompositeTransformer*. The ‘Fancy’ *CompositeTransformer* processes the output of a human player, while the ‘Simple’ *CompositeTransformer* processes the output of the ‘Fancy’ *CompositeTransformer*. Each *Improvisor* uses one of the *CompositeTransformers*, depending on how the *Improvisors* are configured.

In two other configurations, SAL80 ignores the human performers. In the looping configuration, the computer constructs a short loop *Phrase* from its own output. The *Transformers* use it as their source, yielding a stream of constantly changing variations on the loop *Phrase* (see Figure 5.7(c)). The human performer uses this configuration to emphasize something particularly interesting in the computer’s output. The navel gazing configuration involves each *Transformer* operating on the output of the other (see Figure 5.7(d)). This yields a stream of variations over whatever *Phrases* were queued in the *ImprovisorComponents* when the command was issued.

- *Partitioning human output*

SAL80’s role is defined by which portions of the human performer’s output it processes. The *FixedBufferListener* and *Excerpt* transformer define the attention span and focus of the system. The *FixedBufferListener* retains the 200 most recent chords from the input stream. Periodically, the *FancyTransformer* will request a *Phrase* from the *FixedBufferListener*, causing the *FixedBufferListener* to convert its buffer into a *Phrase*. The *FancyTransformer’s Excerpt* scans through the new *Phrase* for pauses. It creates between two and four excerpts from the *Phrase*, each

starting at one of the pauses and lasting between eight and fifteen seconds. Only after the *CompositeTransformer* consumes all the excerpts will the *FixedBufferListener* again be polled and a new set of excerpts generated. The system thus dwells on a particular portion of the input for between fifteen and sixty seconds, creating an on-going interplay between human and computer (see Section 2.1)

Key sequence	Target	Parameter	Effect
t 1	<i>Transposer</i>	#(0)	No transposition
t 2	<i>Transposer</i>	#(-6 -3 0 3 6)	Transpose by minor thirds
t 3	<i>Transposer</i>	#(-8 -4 0 4 8)	Transpose by major thirds
t 4	<i>Transposer</i>	#(-6 -5 0 1 6 7)	Irregular transpose
t 5	<i>Transposer</i>	#(-7 -5 0 5 7)	Transpose by fourths and fifths
shift z	<i>Inversion</i>	toggle %50	Toggle between 0% and 50% likelihood
shift x	<i>Retrograde</i>	toggle %50	Toggle between 0% and 50% likelihood
shift 5	<i>Skip</i>	48%	Skip roughly half the notes

Figure 5.8. Sample SAL keyboard commands

- *Parameter changes*

SAL80's role is defined by varying the parameters of its transformations. Each *Transformer* can use several sets of parameters, as selected by keyboard commands (see Figure 5.8). Depending on parameter variations, *Transformers* can leave the human performer's output essentially intact or alter it radically. The computer's output can be made very dense or quite sparse. Other keyboard commands select sets of timbres for the MIDI synthesizers connected to IB, allowing the human performer to adjust the timbres to the musical texture.

The human musician issues these parameter changes as appropriate to the changing musical texture. For example, if the human musicians begin to play long notes in quartal harmony (i. e., harmonies dominated by the musical interval of a fourth), they may instruct the *Transposer* to

transpose up or down by a fourth, to preserve that harmony. They might also instruct the MIDI synthesizers to adopt a set of slowly evolving timbres with long attack times to accentuate the long notes.

5.2. Jazz Case Study

The second case study focuses on the small group jazz style of the 1940's and 50's. Participating in this style of improvisation requires several specific skills. One is the ability to parse other musicians' playing into metrical units and distinguish between soloing and accompanying. A second is the ability to generate musical phrases that satisfy the harmonic and rhythmic constraints of the tune being performed. A third is the ability to follow the conventional performance structure of melody, solos, trading fours, and restatement of melody. Each ability required new IB components, which were combined with existing components to produce a fledgling jazz improviser.

5.2.1. Parsing Jazz

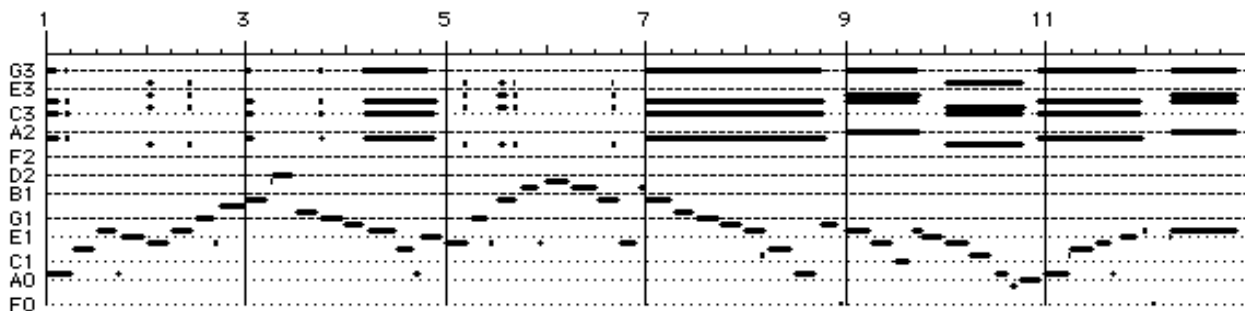


Figure 5.9. Typical walking bass and chord accompaniment

The *MetricalListener* determines whether human musicians are soloing, accompanying, or not playing. Its first job is to assess the pitch range and density of notes being played. Figure 5.9 shows a walking bass and chord accompaniment that a pianist might play, while Figure 5.10 shows one chorus of a typical piano solo. Both figures depict time in beats along the horizontal

axis and pitch along the vertical axis. The *MetricalListener* distinguishes between solo and accompaniment by testing whether the lowest pitch observed in the last few seconds falls in the pitch range typical for a bass line. If the human stops playing, the *MetricalListener* tracks the duration of the pause. As noted in Section 3.1.2, a pause in the musical input may denote the end of a solo, or merely a gap within the solo.

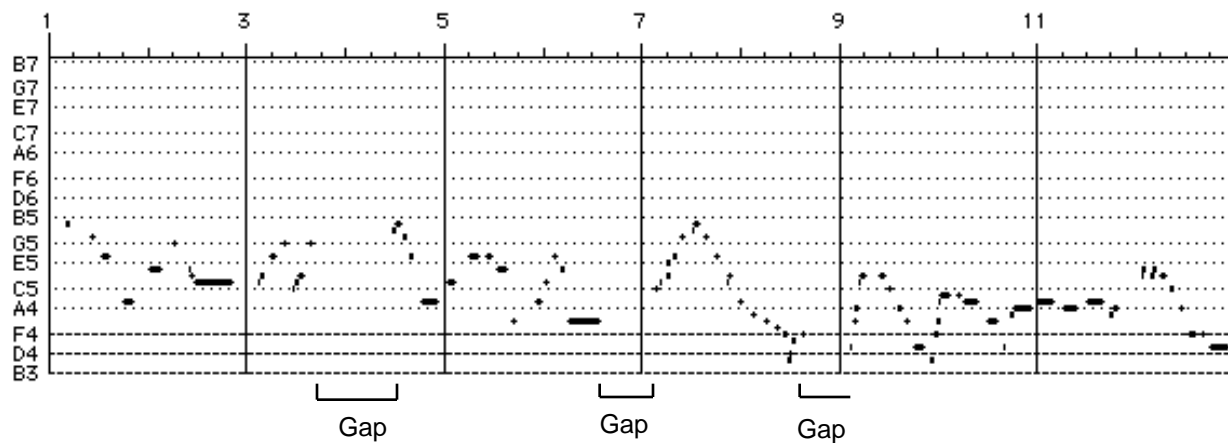


Figure 5.10. Typical piano solo

The *MetricalListener* also breaks the input into *Phrases* of one measure each, tags them with the chord changes for that measure, and stores them in a dictionary indexed by their tags. When generating a solo or accompaniment, the *HarmonyGenerator* can look in the dictionary under the appropriate chord symbol and find a collection of *Phrases* ready for use (see Section 5.2.3).

5.2.2. Following Improvisational Structure

IB represents improvisational structure by a finite state machine, with each state corresponding to playing a specific musical role during a specific segment of the performance—taking the first solo or accompanying the melody, for example. The state machine encodes normative paths through the set of states, in much the same manner as the conversation protocols of *ConversationBuilder*[28].

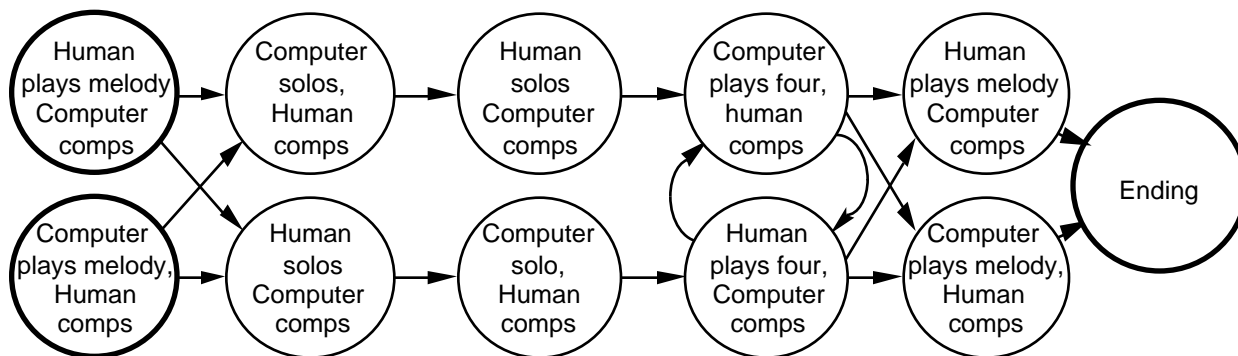


Figure 5.11. Finite State Machine for two piano jazz performance

During a performance, the *MetricalListener* feeds information about the behavior of the human musicians to the current *ImprovisorState*. Each *ImprovisorState* specifies an exit condition. Satisfying that exit condition means that improvisational roles are about to change. For an *Accompany* state, the presence of bass notes in the input indicates that another player is assuming the role of accompanist (see Figure 5.9) and the computer should stop accompanying. Conversely, for a *Solo* state the presence of higher notes indicates that another player is beginning a solo (see Figure 5.10) and the computer should end its solo. When the current state's exit condition is satisfied, it passes control over to the next state. *Solo* and *Accompany* only test their exit conditions near the end of each chorus, since roles change on chorus boundaries.

Figure 5.11 shows a finite state machine for a two piano performance based on a simplified version of the structure in Figure 2.3. Each state specifies a role for each performer; playing the melody, soloing, or accompanying. With only two musicians, the choices about who plays the melody and the order of soloing are greatly simplified. One of the musicians plays the melody, followed by one of the musicians taking the first solo. The other musician then takes the second solo. The two pianists can then trade fours. Trading fours is especially conversational in the absence of a drummer, since the two musicians trade four bar phrases directly.

An extended version of the two piano configuration collaborates with more than one other performer. The single *MetricalListener* is replaced by several, each following the output of a performer encoded on a separate MIDI channel. Each state's exit condition must consider information about the behavior of all other performers. These exit conditions are more complicated

than in the two performer case. In Figure 5.11, the computer need only determine that the human musician has finished his or her solo. With three performers, the computer must await the end of one human musician’s solo and then determine whether it or the other human musician solos next.

A small number of states easily represents the two piano performance in Figure 5.11. As the number of performers increases, however, a state machine that enumerates all possible orderings of solos becomes intractably large, suggesting that the improvisation’s structure is too complex to be well represented by a regular grammar. A higher-order grammar can more simply capture the standard constraint on solo ordering; namely, that each soloist appears only once.

Following the rules in the Sacks paper[47], the computer will defer if the current soloist explicitly selects the next soloist. Human musicians inform the computer of these decisions by simple keyboard commands, which are equivalent to the visual and verbal cues that humans use with each other. If no “speaker selects next” gesture is evident, the computer can attempt to self-select as the next soloist. Its likelihood of doing so can vary according to an “assertiveness” parameter. When self-selecting, the computer must monitor the other players to see if one of them is also attempting to begin a solo.

5.2.3. Generating Jazz

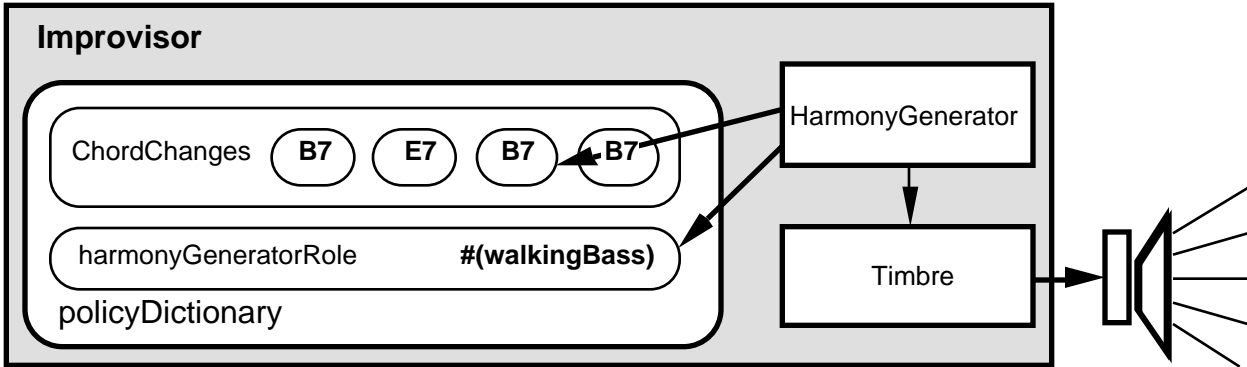


Figure 5.12. HarmonyGenerator and PolicyDictionary

HarmonyGenerator is a Composer that constructs musical phrases in the manner of a jazz pianist. It draws on two sources of information (see Figure 5.12). One is the ChordChanges,

which records the underlying harmony for each measure. The other is the current *ImprovisorState*, which describes the computer's current role. *HarmonyGenerator* uses this information to generate one measure of music and send it to the *Realizer* for playback. To provide variety, *HarmonyGenerator* combines several strategies for generating melodies:

- It combines short melodic fragments (“riffs”) from a library to produce phrases. Beginning human improvisors often employ this tactic, and even the most experienced players tend to rely more on their favorite riffs when playing at rapid tempos.
- It performs a random walk in the scale appropriate to the current chord structure, a tactic employed by only the most inexperienced players.
- It copies material extracted from the human musician's solo by the *MetricalListener*.

The *HarmonyGenerator* also generates accompaniment, consisting of a walking bass line played by the left hand and occasional chords played by the right hand. The walking bass lines are built by fitting together short patterns. The right hand chords are constructed using rules found in the piano jazz instruction materials of Tony Caramia[8]. As with its solos, the *HarmonyGenerator* sometimes extracts material from the human musician's accompaniment as catalogued by the *MetricalListener*.

6. Implementing the Framework

ImprovisationBuilder (IB) is written in ParcPlace ObjectWorks\Smalltalk for the Apple Macintosh[38]. IB handles musical input and output through several facilities. The Apple MIDI Manager provides access to a hardware MIDI interface through the Macintosh serial port. The Capybara driver uses a Macintosh NuBus interface card to provide access to the Symbolic Sound Capybara Signal Processor. IB uses a Smalltalk virtual machine with special C primitives that mediate between Smalltalk code and these drivers (see Figure 6.1).

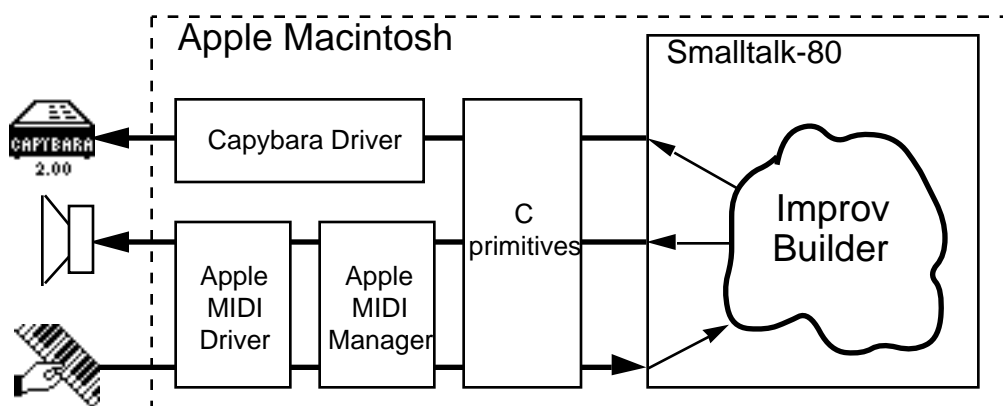


Figure 6.1. Input and output connections

Over the course of IB's development, the implementation evolved as framework design became more general. The initial implementation of some framework aspect was the most obvious one suggested by the design. As the design evolved, these initial implementations often proved inefficient or difficult to generalize. Here are four examples of how the implementation changed as it became more general, followed by a discussion of the design's novel consequences for IB's overall real-time behavior.

6.1. Preemptive versus Cooperative Multi-tasking

IB's design sees *ImprovisorComponents* as independent processes operating simultaneously. Accordingly, early versions of IB allocated a Smalltalk *Process* to each *ImprovisorComponent*. An augmented version of Smalltalk's *ProcessScheduler* enforced preemptive multi-tasking on the *Processes*. Each *Process* performed computations for a single *ImprovisorComponent*. The components transmitted *Phrases* to each other through *SharedQueues*. The *SharedQueues* used *Semaphores* both to enforce mutual exclusion and to suspend *ImprovisorComponents* while waiting on an empty queue. The *MusicScheduler* used *SharedQueues* to provide exclusive access to its *MusicMessageQueue*. Real-time performance during dense musical textures was poor, and code profiling revealed that as much as 20% of processor time was spent in *Semaphore* overhead.

Multi-tasking also made it difficult to reconfigure *ImprovisorComponents* during performance. Commands to reconfigure *ImprovisorComponents* should be processed as soon as they are issued. This meant interrupting or resuming the *Processes*, which led to process management overhead. For example, the *Realizer's Process* used a *Delay* to suspend itself while waiting to send a *MusicMessage*. To resume a *Realizer*, the *Delay's* internal *Semaphore* had to be signaled prematurely. To distinguish this from a normal resumption, the newly resumed *Realizer* consulted a special flag to detect whether it had been reconfigured. When such premature resumption was inappropriate, some processes were terminated and restarted, adding further process management overhead.

The current IB relies on non-preemptive sharing of one *Process*. The *Process* executes a loop in the *MusicScheduler*, polling the *Timbres* for their next output *MusicMessage* and then sending it. The *MusicScheduler's* **computeEarliestTimbre** method finds which *Timbre* has the earliest *MusicMessage* to send. The steps in **computeEarliestTimbre** are as follows:

1. Poll each *Timbre* with **isReady** and record the result. If the *Timbre* is ready, record when to send its next message. If the *Timbre* is not ready, record *nil*.
2. Consider the results recorded above and perform one of three actions:
 - 2(a). If none of the *Timbres* knows when to send its next message, then the entire system is waiting for human input. Display a “waiting for input” message and return to step 1.
 - 2(b). If all the *Timbres* know when to send their next message, return the *Timbre* with the earliest message.
 - 2(c). If some *Timbres* know when to send their next message and others do not, continue to poll with **isReady** as in step 1 until shortly before the earliest known message should be sent. Then, return the *Timbre* with the earliest known message.

The **isReady** message serves two purposes. First, **isReady** returns a *Boolean*, indicating whether the receiver is ready to supply a *Phrase* to the component connected to it. The readiness of one component often depends on the readiness of its source. For example, a *Transposer* is ready to supply a *Phrase* if and only if its source is ready. Second, each component within its implementation of **isReady** performs a small amount of the computation formerly handled by its own *Process*. The calls to **isReady** eventually propagate through the entire chain of *ImprovisorComponents*, offering each an opportunity to perform computation. Since only one process executes at a time, and since each component defines the boundaries of its own time slice, no *Semaphores* are required to ensure mutual exclusion, and reconfiguration of components takes place without suspending and resuming *Processes*.

6.2. Processing Keyboard Commands

IB provides a general mechanism for defining keyboard commands and issuing them during performance (see Section 4.5). The *KeyboardMusicController* first uses a *DispatchTable* to map key presses to action requests, then creates and queues a *MessageSend* encoding the requested action. The *MusicScheduler* dequeues and executes these commands during its idle time, avoiding any impact on real-time music processing.

These commands fall into several categories. The largest group of commands are those that alter the parameters of an *ImprovisorComponent*. For example, the human performer may choose between several sets of transposition intervals used in a *Transposer*. The earliest version of *Improvisor* held each component in a separate instance variable, and commands were forwarded directly to the affected component. However, one *Improvisor* might not have any *Transformers*, while another *Improvisor* might have several. Creating a new subclass of *Improvisor* for each new set of components would have been inelegant.

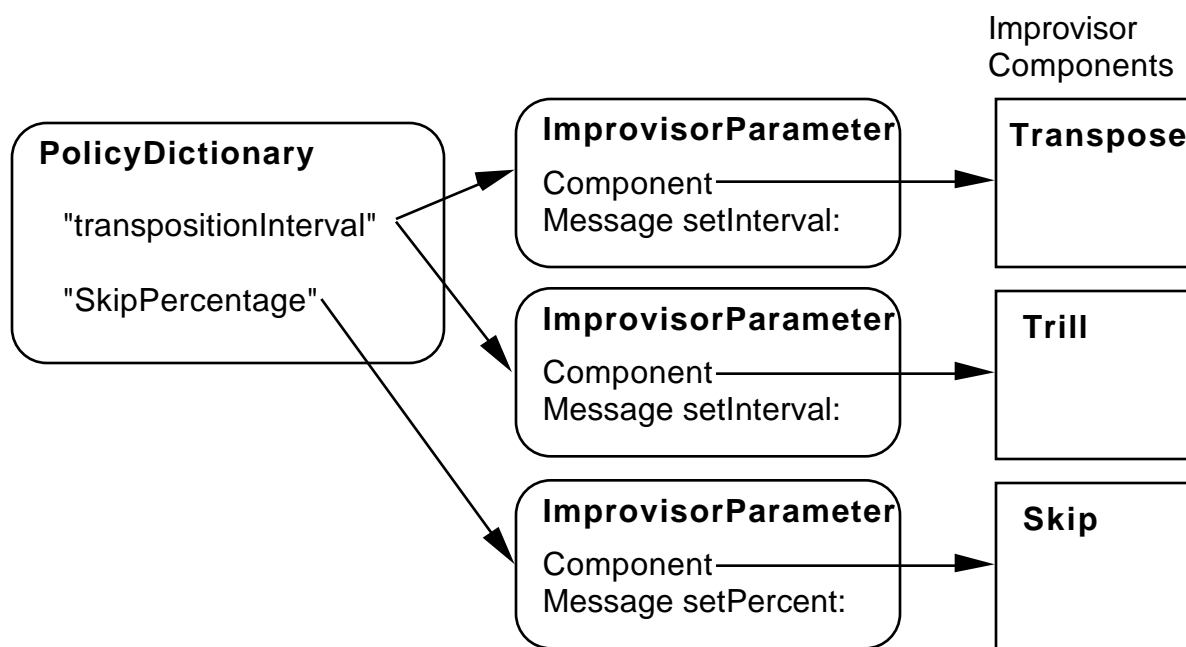


Figure 6.2. *PolicyDictionary* and *ImprovisorParameters*

To provide a consistent mechanism for accessing parameters, each *Improvisor* maintains a *PolicyDictionary* that maps each parameter name to an array of *ImprovisorParameters*. Each *ImprovisorParameter* represents one parameter of one *ImprovisorComponent* (see Figure 6.2). To change a parameter, the *MessageSend* representing the command forwards the parameter's new value to the *ImprovisorParameters* listed for that parameter in the *PolicyDictionary*. The *ImprovisorParameters* in turn forward the new value to the affected *ImprovisorComponents*. The *PolicyDictionary* of *ImprovisorParameters* allows a single keyboard command to be routed through several *ImprovisorParameters* and affect several *ImprovisorComponents*. When created, each *ImprovisorComponent* sends a list of its *ImprovisorParameters* to the *Improvisor*. Thus, the *Improvisor* implicitly obtains a complete list of its components' parameters.

6.3. *KymaTimbre*

IB's *Timbre* hierarchy is designed to place all sound hardware dependencies in a single class hierarchy. This hierarchy encompasses control over both MIDI synthesizers and Kyma sounds. These are two very different conceptions of sound. MIDI regards music as a sequence of musical events with start and stop times. Kyma is a sound specification language that does not distinguish between signal level and event level processing or between the concepts of orchestra and score; these models are supplanted by arbitrary hierarchical structures constructed by the composer out of uniform Sound objects[48, 49]. Kyma Sounds are synthesized in real-time by the Capybara, a digital signal multiprocessor. A Macintosh driver for controlling the Capybara enables any application to play a Kyma Sound and to control its parameters in real time. The goal of the *Timbre* hierarchy is to present a common protocol for *MIDITimbre* and *KymaTimbre*.

A *KymaTimbre* represents a Kyma sound designed for use with IB. Regardless of the Kyma sound's internal structure, IB represents it as a collection of *KymaVoices*. MIDI synthesizers are responsible for their own voice management. Thus, *MIDITimbre* sends *NoteOns*

and *NoteOffs* directly to the MIDI synthesizer, while *KymaTimbre* performs this voice management internally before generating *KymaNoteOns* and *KymaNoteOffs*.

In particular, *KymaTimbre* must handle situations in which the number of simultaneous notes exceeds the number of *KymaVoices*. The solution, called voice stealing, is built into most MIDI synthesizers. When instructed to start more new notes than the number of available voices, the *KymaTimbre* turns off and reallocates the voice playing the oldest note. Despite the differences between *KymaTimbre* and *MIDITimbre*, both inherit and reuse the **basicStartChord:** method defined in *Timbre*.

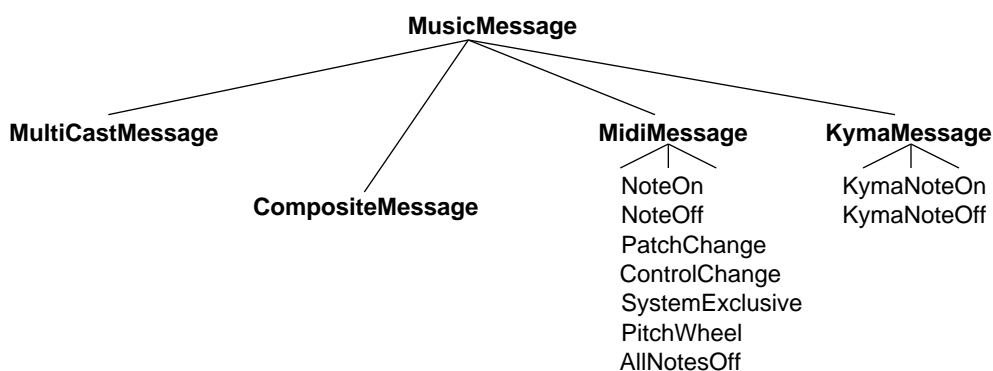


Figure 6.3. *MusicMessage* class hierarchy

Each *KymaVoice* contains a set of *ExternalInputs* that represent *ExternalInput* sound objects in Kyma. These sound objects allow IB to control the Kyma sound in real-time. Each Kyma sound may have a different set of *ExternalInputs*, and constructing a subclass of *KymaTimbre* for each of them would be inelegant. Instead, IB uses a *SoundNameParser* to dynamically parse the *ExternalInput* names and construct the appropriate collection of *KymaTimbres*, *KymaVoices*, and *ParsedExternalInputs*.

The *ExternalInput* names encode information about what kind of information should be sent to them. The first two characters of the name indicate whether the *ExternalInput* should receive triggering information, frequency information, or volume information. Additional characters specify arithmetic modifications of these parameters. For example, an *ExternalInput* named “frV1T1” receives the frequency of the note and another *ExternalInput* named “frM2V1T1”

receives twice the frequency. This avoids performing unnecessary calculations with the signal processor, freeing it for sample generation. The “V” and “T” characters precede voice number and timbre number, respectively.

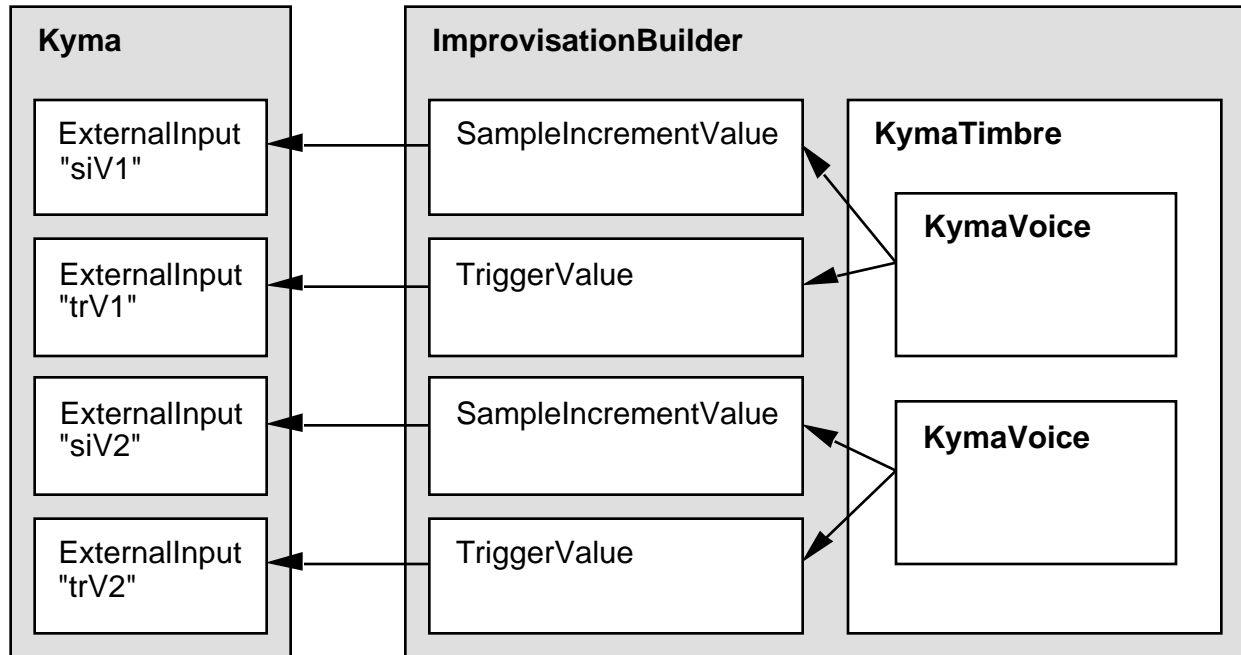


Figure 6.4. Kyma sound and *KymaTimbre*

For each Kyma ExternalInput named with this convention, IB builds a corresponding Smalltalk object—typically a *SampleIncrementValue*, a *VolumeValue*, or a *TriggerValue*. These classes encapsulate information about what to send each kind of value at the beginnings and ends of notes. A *TriggerValue* must be set at the beginning of the note and reset at its end, while a *SampleIncrementValue* need only be set at the beginning of the note. A *KymaVoice* collects all these changes from its *ParsedExternalInputs* into a *KymaMessage*. A *KymaMessage* is sent by invoking the Kyma driver and setting the appropriate parameter.

6.4. Real-time Behavior

IB’s design means that a configuration’s overall behavior results from the combined behaviors of many *ImprovvisorComponents*. Since the components are arranged in a chain, each

component can have a strong effect on the overall behavior. The next two sections represent novel issues raised by such an arrangement.

6.4.1. Lead time for Scheduling

The *MusicScheduler* sends *MusicMessages* generated by one or more *Timbres* at the appropriate time. Apple MIDI Manager, an operating system extension that processes time-tagged MIDI data, greatly simplifies the handling of *MusicMessages*. The *MusicScheduler* simply sends *MIDI Messages* to the MIDI Manager shortly before their intended play time. This lead time acts as a buffer against variations in both the density of the musical output and the available computation time. However, the lead time also defines the fastest possible reaction to an input message. A longer lead time provides more buffering but slower interactivity. Thus, the lead time between sending and playing is a compromise between computation time and interactivity.

Since the Capybara driver processes messages immediately, the *MusicScheduler* must assume the responsibility for sending *KymaMessages* exactly at playback time. No compromises between computation and interactivity are involved. IB uses a special Smalltalk timing mechanism based on the MIDI Manager's millisecond clock to make the Capybara playback timing as accurate as possible.

A short lead time can be better tolerated by reducing unnecessary overhead. Several low-level optimizations have improved the system's MIDI throughput. The conversion between arrays of data bytes and *MIDI Messages*, once handled in Smalltalk, is now performed in the C primitives themselves. The user primitives maintain a large circular buffer of incoming MIDI data to avoid unnecessary interruptions from the Apple MIDI Driver. The Apple MIDI Driver's internal buffer was enlarged, enabling it to process the amount of data associated with playing large chords on multiple MIDI channels.

6.4.2. Importance of Listener Readiness

Listeners construct *Phrases* in different ways, and the pattern of their readiness to supply a *Phrase* can largely determine the character of the system's interactivity. The *FixedBufferListener* maintains a buffer containing the most recent input chords. The *FixedBufferListener* is considered ready if the buffer is fuller than a given threshold. When asked to create a *Phrase*, the *FixedBufferListener* empties the buffer and begins anew. Thus, the *FixedBufferListener* is designed to process a constant stream of input. In cases when input is sparse or sporadic, the *FixedBufferListener* may have an inadequately full buffer when the rest of the *ImprovisorComponents* complete their handling of the previous *Phrase*. When this happens, the *FixedBufferListener* forces *ImprovisorComponents* further along in the chain to wait until it accumulates enough material to permit creation of a new *Phrase*. Thus, a sufficiently long pause in the input will yield a pause in the output.

The *PauseListener* defines a phrase as all the notes between silences of at least two seconds. It accumulates *Chords* until a sufficiently long silence is detected. The accumulated *Chords* are then made into a *Phrase* and placed in an output queue. Depending on the distribution of these silences in the input, the *PauseListener* may queue up several *Phrases* between requests for them. More typically, though, the *PauseListener* forces other components to wait until it detects the end of each phrase, causing a "call and response" in which the human plays a phrase and IB responds when the human is done.

7. Summary and Future Work

This thesis uses results from conversation analysis to inform the application of computers to musical improvisation. The original contributions of this research are threefold.

- *Recognizing that the strong relationship between conversation and improvisation should inform the design of interactive music systems*

These two activities have many parallels. Both are collaborations in which participants generate highly structured sounds that help them synchronize their conduct. Both employ the same kinds of collaborative cues for navigating through the same kinds of collaborative structures. Many writers have elaborated and considered their similarities [23, 40, 44, 52]. This relationship was made concrete by applying conversation analysis to musical improvisation.

- *Designing an object-oriented framework based on this relationship*

Conversation analysis informed the design of *ImprovisationBuilder*, an object-oriented framework written in Smalltalk-80. The roles of various components (*Listener*, *Composer*, *Realizer*), the hierarchical musical representation, the representation of collaborative structure and participant roles, and the attention to real-time scheduling all resulted by analogy with conversation. *ImprovisationBuilder* provides both ready-to-use components and templates for creating compatible new components.

- *Applying the framework to improvisational situations*

Two musical domains served to test *ImprovisationBuilder*'s reusability. The first is a free-form three part canon defined by Sal Martirano's Sound and Logic system. The second is the tradition of small jazz ensemble performance made popular in the 1940's and 50's. IB configurations were developed to participate in these domains. The Sound and Logic 80 configuration contains two *Improvisors*. In the first, a set of *Transformers* operates on the output of a *FixedBufferListener* to produce variations on the human musician's performance. A second

Improvisor further transforms the output of the first. The two *Improvisors* thus provide two of the three parts in a canon.

The jazz configuration uses a finite state machine composed of *ImprovisorStates* to model the standard structure of mainstream jazz performances. A *HarmonyGenerator* produces accompaniment and solo in the proper keys, while a *MetricalListener* stores measures from the human musician's performance for later re-use.

From an object-oriented designer's point of view, *ImprovisationBuilder* succeeds if it fulfills the standard expectations for a framework—that it can be used to construct a diversity of working systems, that its reusable design is appropriate for these diverse systems, and that the collaboration defined by its abstract components enables the construction of new systems with minimal effort[15, 61].

The configurations described above fit well with the framework's design. In each case a few new components were added to the existing ones to produce a new system. New components can be debugged while the system is running, and their modularity makes them easy to maintain. IB is being used by Professor Brian Belet at San Jose State University as part of a composition for computer and pianist[5]. During Belet's work at the CERL Sound Group's 1993 Intensive Workshop on Sound Computation, he designed several new *Transformers* for trills and grace notes. These new components were combined with the existing *FixedBufferListener* and *Excerpt* to produce some novel interactions. He also added several new keyboard commands to control these new *Transformers*.

IB has been proven robust. Salvatore and Dorothy Martirano performed with the Sound and Logic 80 configuration at the *1993 International Week of New Music* festival in Bucharest, Romania. During the twenty minute performance, the program processed sometimes dense input from two virtuoso performers and transformed it to provide relevant accompaniment in real-time.

7.2. Future Work

I intend to continue developing the framework, testing its generality by applying it to more kinds of musical improvisation. Some areas for development include:

- *Higher-order model of collaboration structure*—A state machine model of collaboration structure provides an easily understood, computationally efficient way to govern the computer's behavior. Some collaboration structures will be best represented by a higher-order construct, such as a context-free grammar.

- *Long-term Memory*—The *MetricalListener* stores *Phrases* from the human performer's solo and accompaniment for use by the *HarmonyGenerator*. At present, IB discards this material following each performance. A more sophisticated way to store and index these *Phrases* would allow IB to retain them across several performances and thereby adapt to the human performer.

- *Application to extra-musical domains*—The general model of structured collaboration embodied in *ImprovisationBuilder* should be applicable to other collaboration domains. One example is generating musical accompaniment for graphical adventure games. Current games splice together pre-composed musical segments as the player moves from room to room, yielding a fragmented musical experience. *ImprovisationBuilder* could be configured to monitor the player's moves in the game, track his or her progress through a representation of the game's structure, and improvise a continuous, unified musical accompaniment.

Appendix A. Sample Jazz Performance

Figure A.1 shows a graphical representation of a performance by IB and the author. The computer's part is indicated by solid bars, the human's part by hollow bars. This performance follows the structure of Figure 5.5, with the human playing the melody in chorus 1, the computer soloing in choruses 2 and 3, and the human soloing in choruses 4 and 5. After trading fours in choruses 6 and 7, the human issues a keyboard command to end trading fours and plays the melody one final time (chorus 8), after which the computer stops. Notice how pauses in the human's playing are not seen as transition places if they occur in the middle of a chorus.

In practice, IB's jazz performance is quite respectable, and shows great promise as a jazz teaching tool. While the bass lines and melodies generated by the system are rather formulaic, its ability to detect the beginnings and endings of solos and borrow from the human performer's solo allow much more interaction than playing along with a prefabricated accompaniment.

Figure A.2 shows the screen of an Apple Macintosh running *ImprovisationBuilder*. Along the bottom are boxes representing the states of the state machine, while immediately above them are boxes representing the *ImprovisorComponents*. The top part of the window is reserved for status messages and controls.

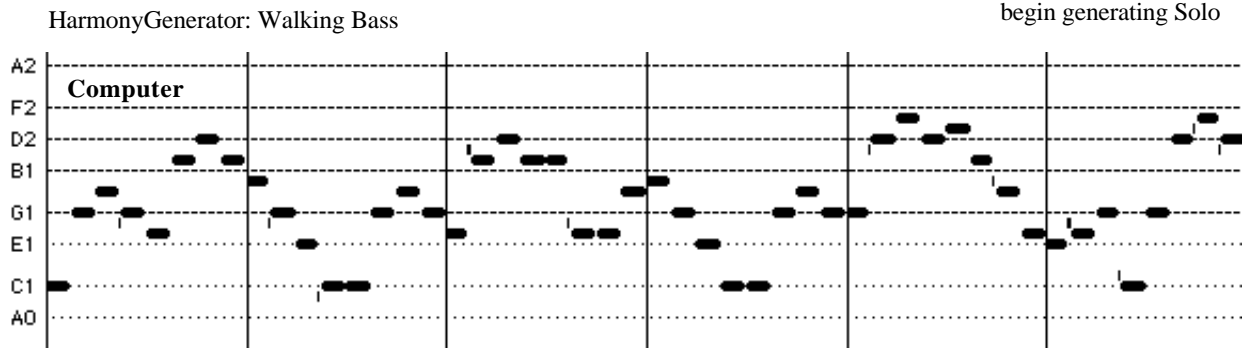
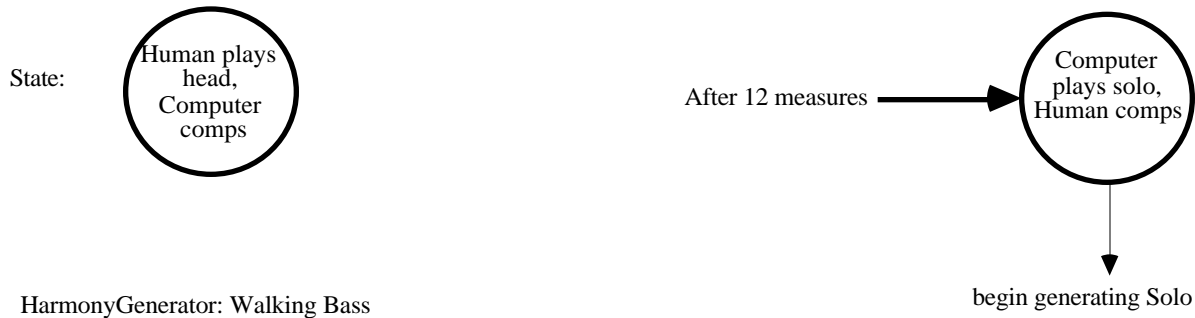
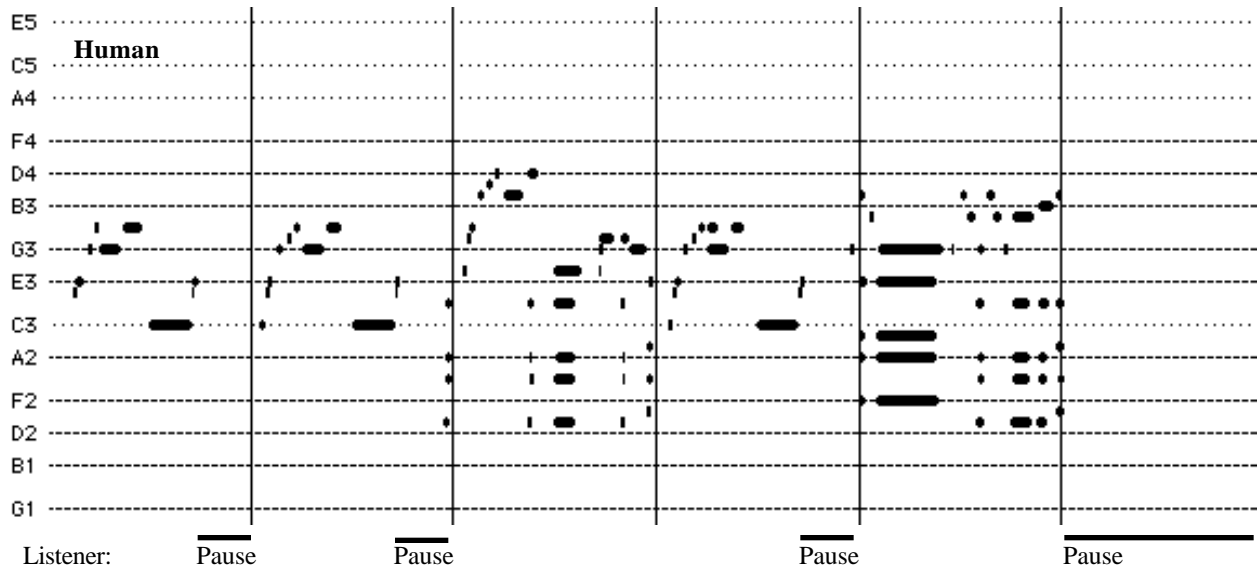
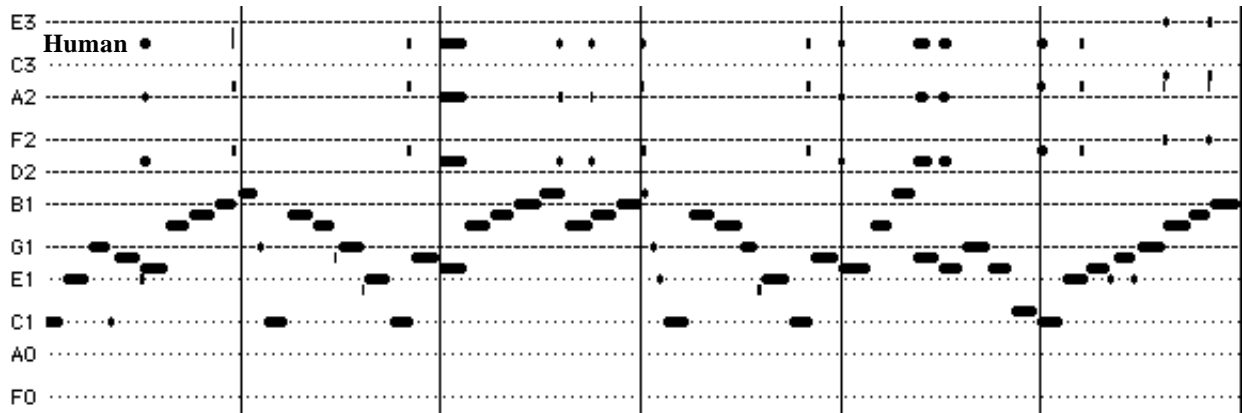


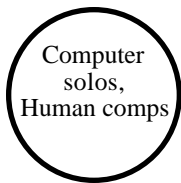
Figure A.1(a) Chorus 1—Human plays melody, Computer accompanies



MetricalListener:

(no pauses)

State:



HarmonyGenerator: generate Solo

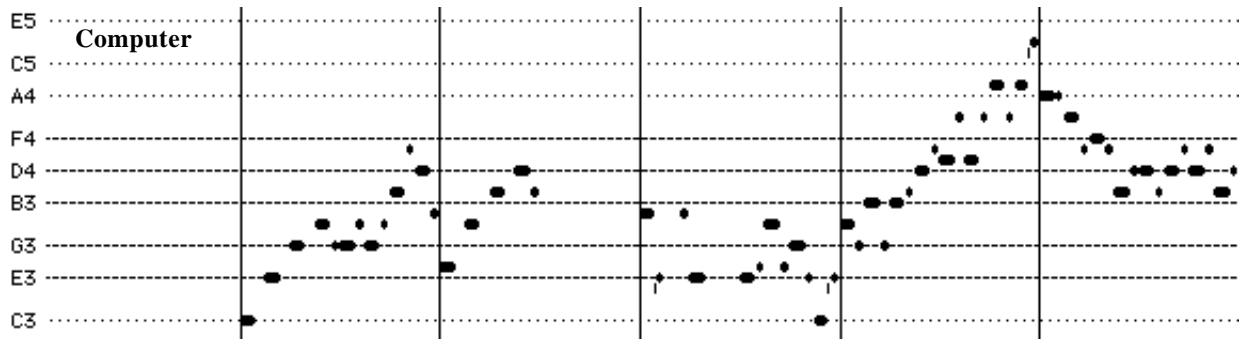


Figure A.1(b) Chorus 2—Computer solos, Human accompanies

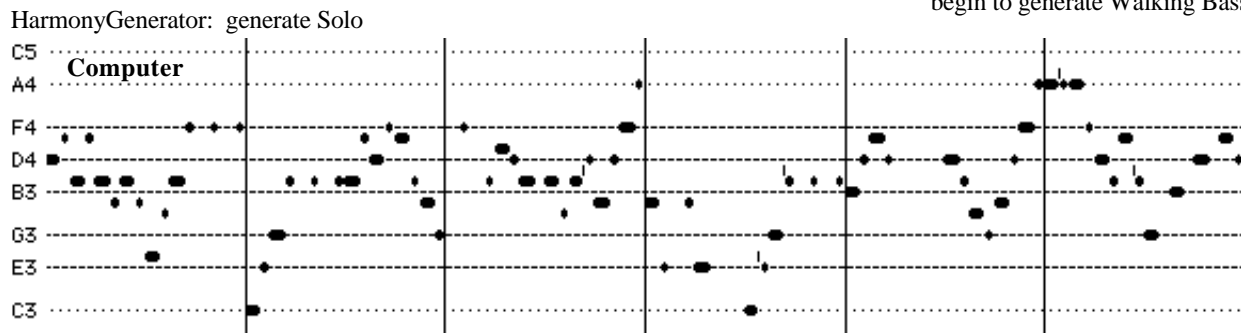
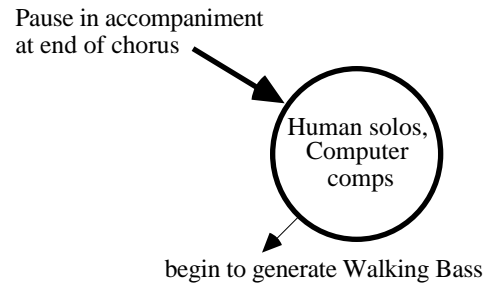
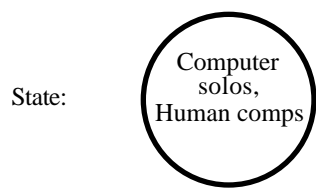
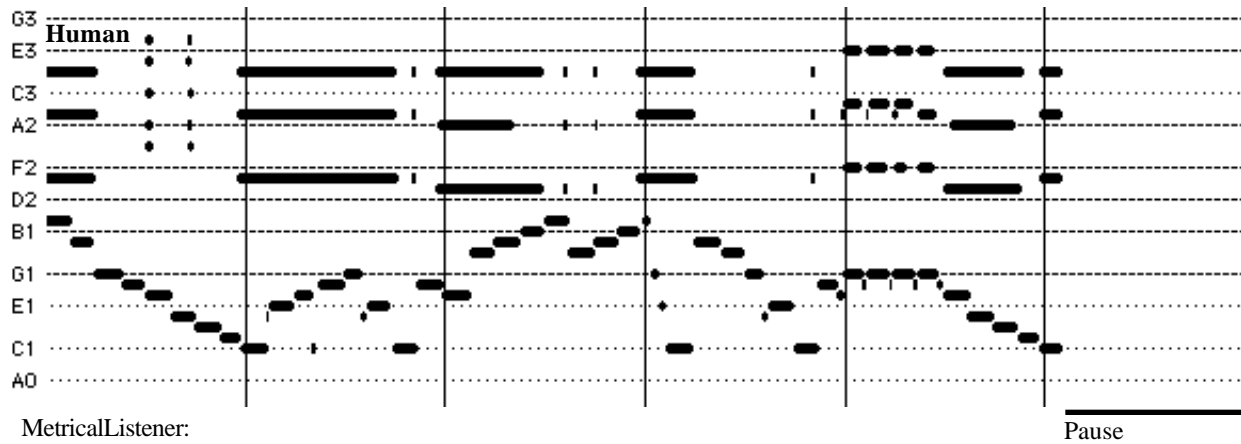


Figure A.1(c) Chorus 3—Computer solos, Human accompanies

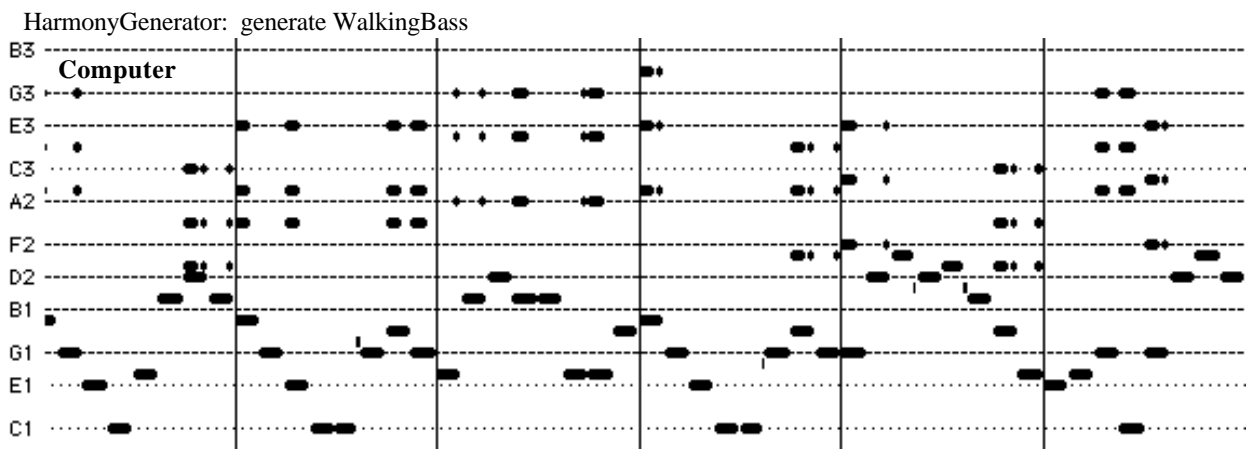
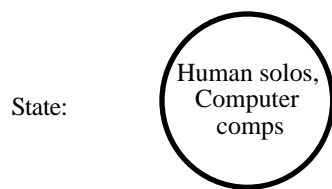
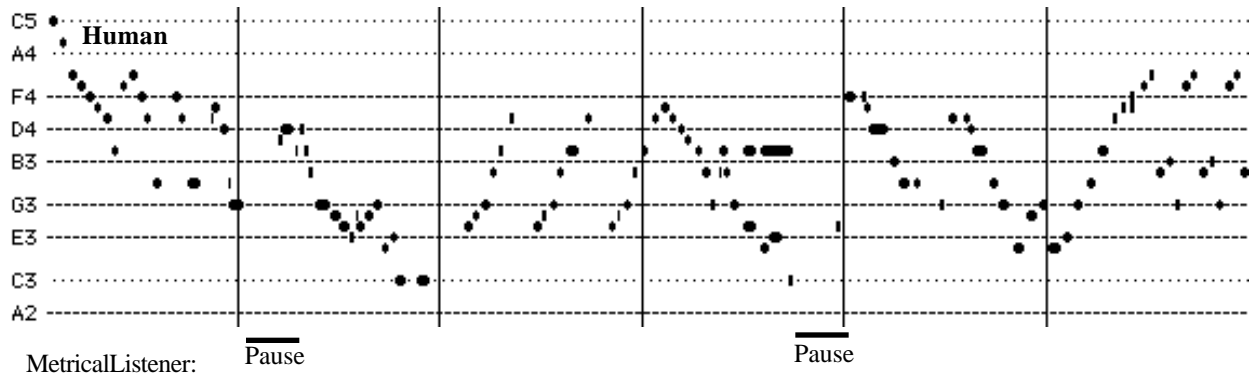
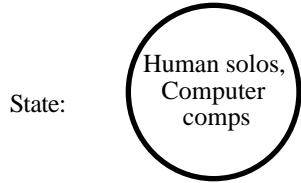
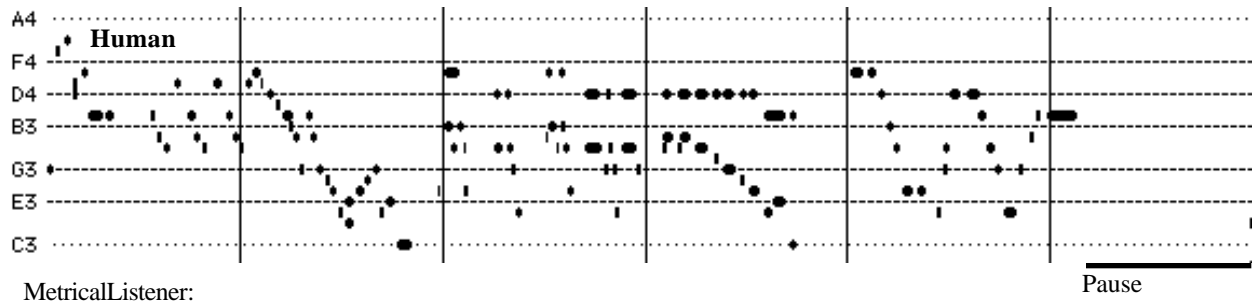


Figure A.1(d) Chorus 4—Human solos, Computer accompanies



pause in Human solo at
end of Chorus

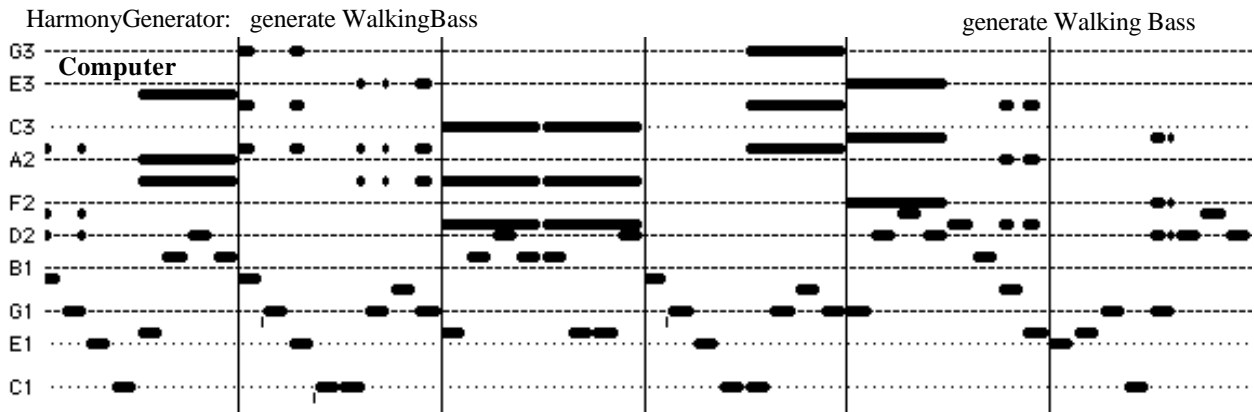
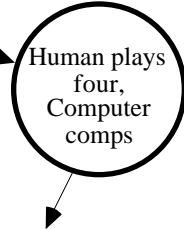
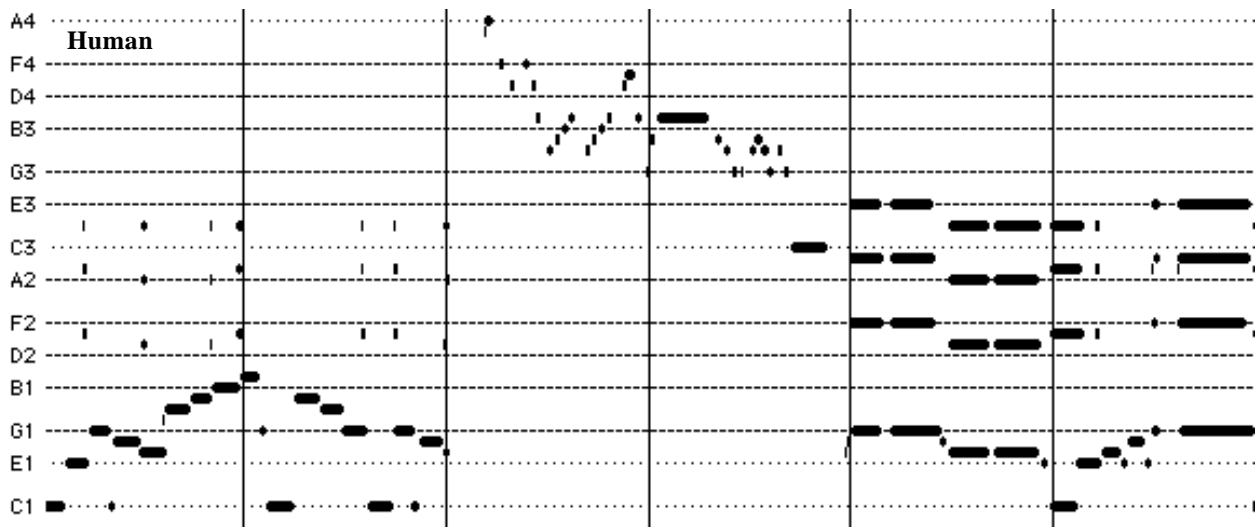
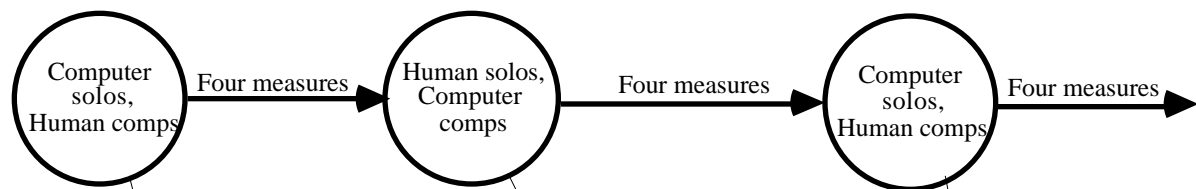


Figure A.1(e) Chorus 5—Human solos, Computer accompanies



State:



Harmony
Generator:

generate Solo

generate Walking Bass

generate Solo

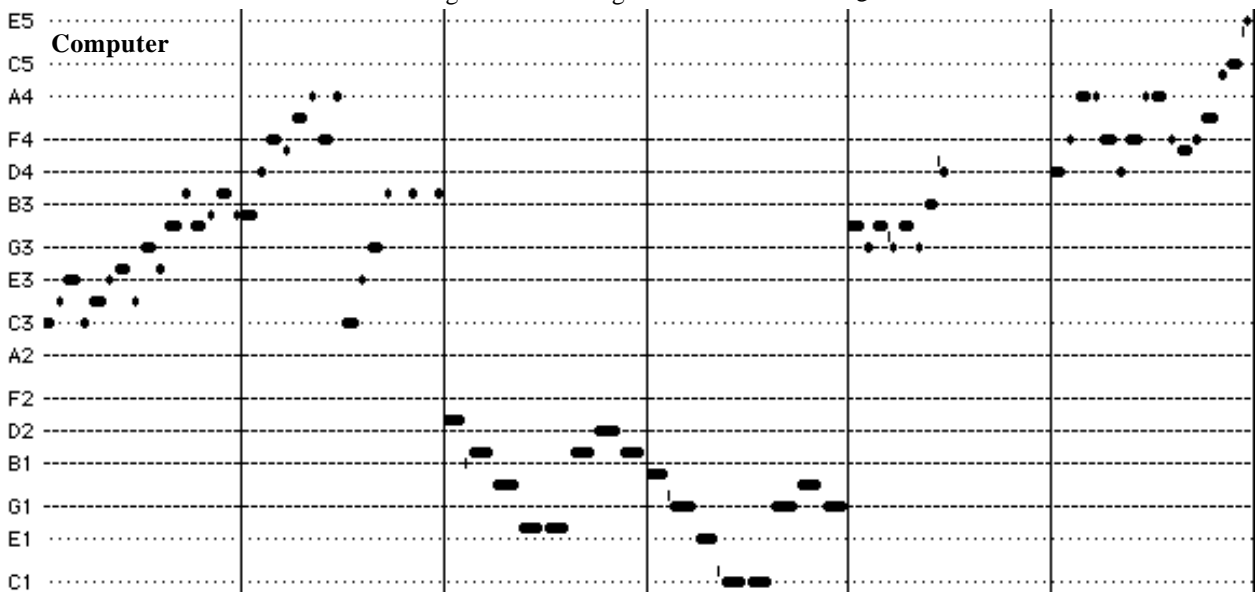
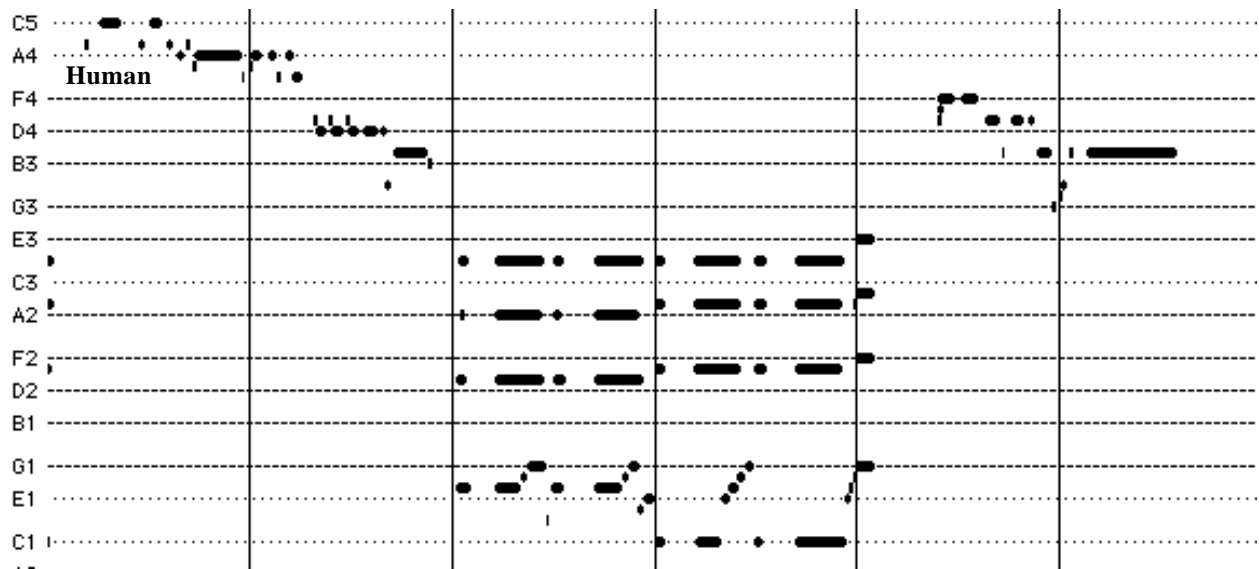


Figure A.1(f) Chorus 6—Trading Fours (computer, human, computer)



(Keyboard command breaks out of trading fours loop)

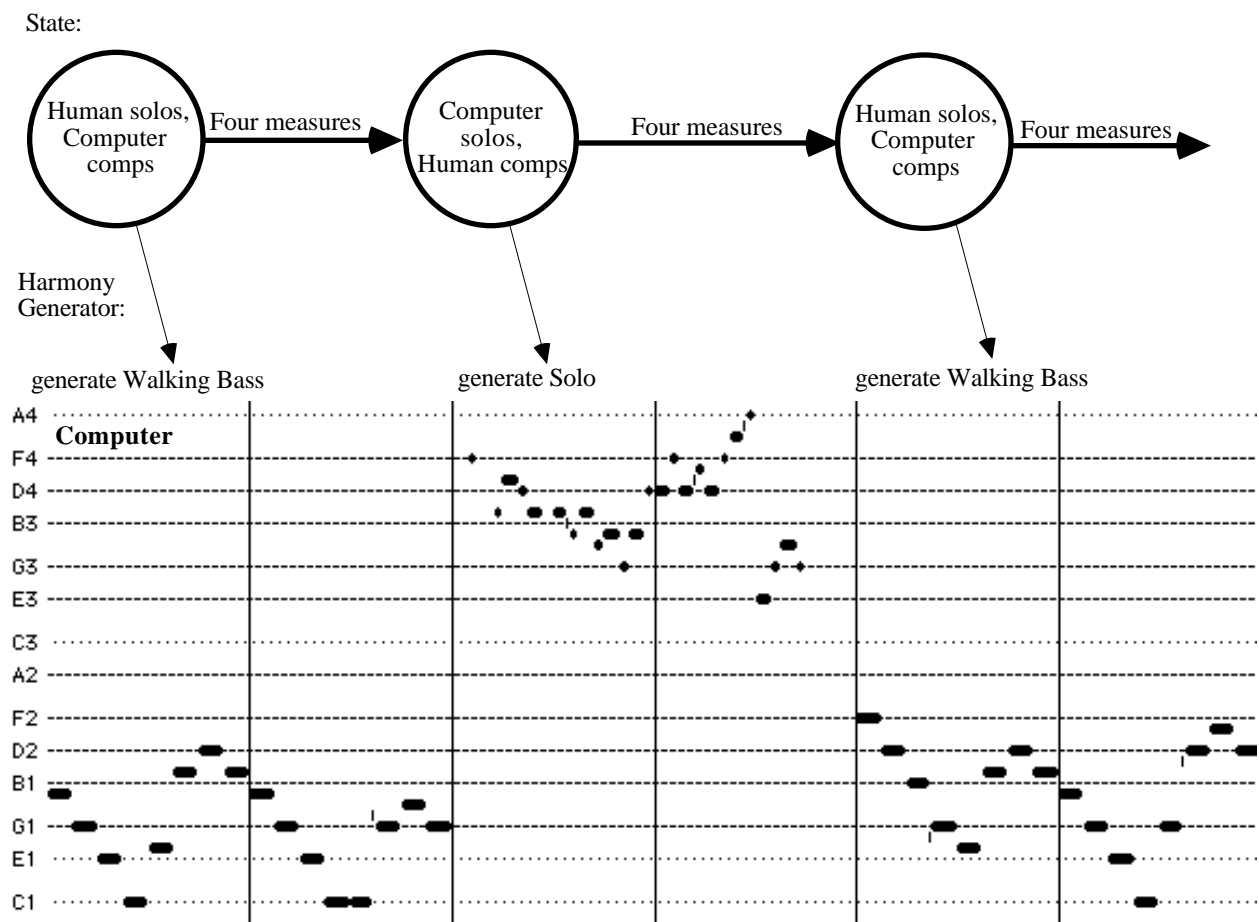


Figure A.1(g) Chorus 7—Trading Fours (human, computer, human)

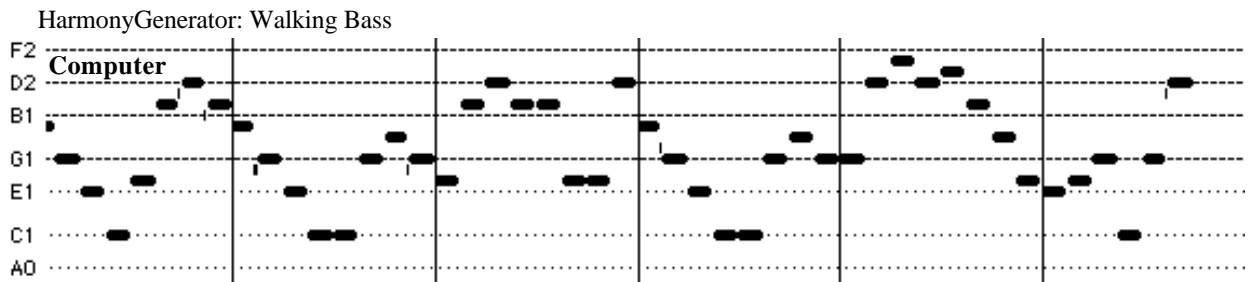
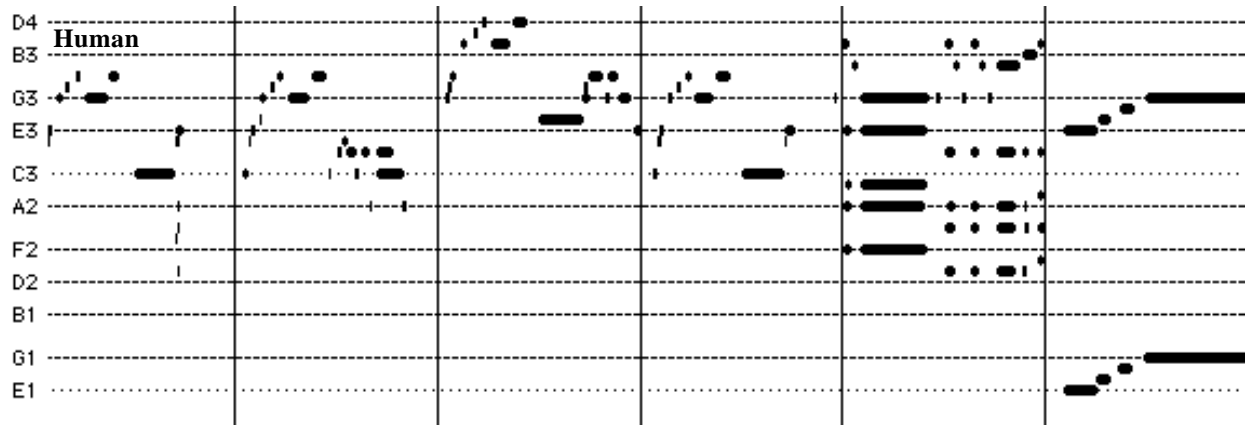


Figure A.1(h) Chorus 8—Human plays melody, Computer accompanies

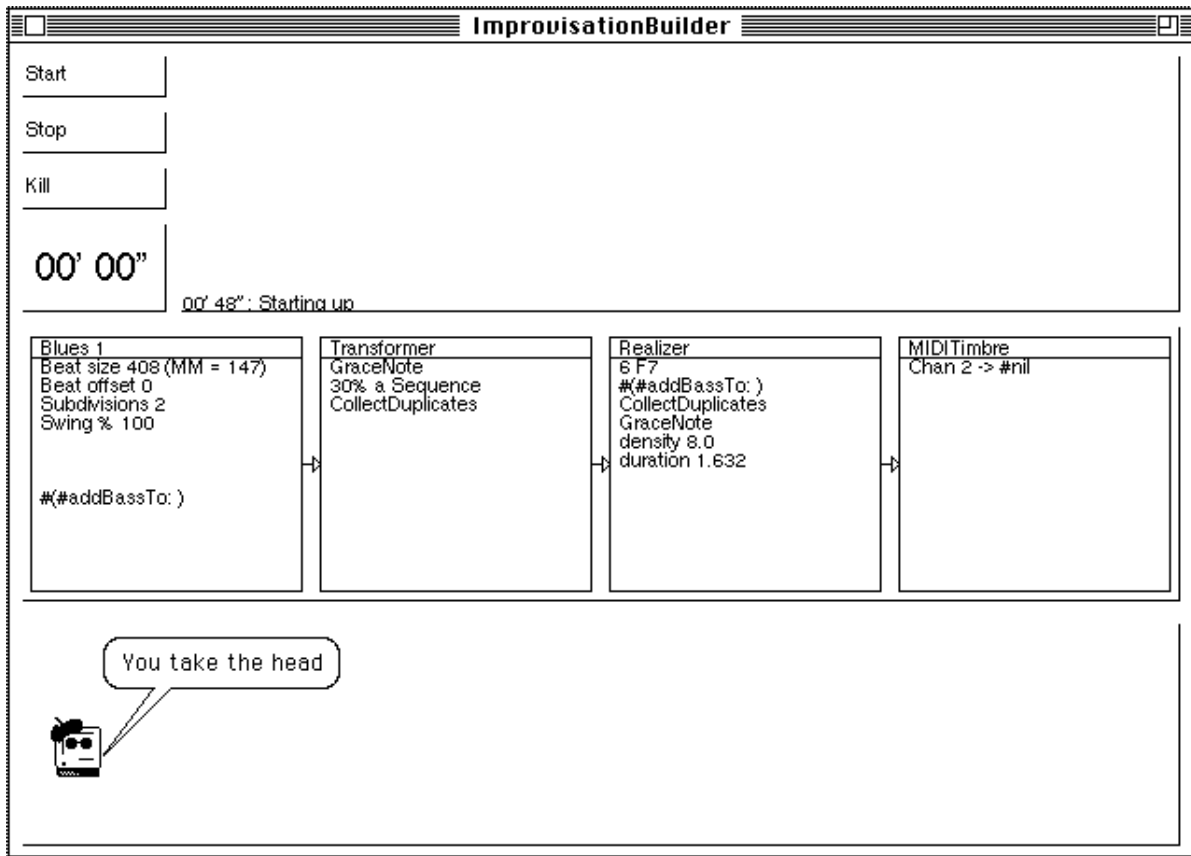


Figure A.2. ImprovisationBuilder screen

Bibliography

1. Baird, B. *The Artificially Intelligent Computer Performer on the Macintosh II and a Pattern Matching Algorithm for Real-Time Interactive Performance*. in *International Computer Music Conference*. 1989. Columbus, Ohio: International Computer Music Association.
2. Bastien, D. and T. Hostager, *Jazz as a Process of Organizational Innovation*. *Communication Research*, 1988. 15(5): p. 582—602.
3. Becker, H., *Doing Things Together*. 1986, Evanston: Northwestern University Press.
4. Bel, B., *Bol Processor BP2*. *Communications of the ICMA*, 1992. 11(4A).
5. Belet, B. *Integrating real-time interactive software synthesis, pre-processed resynthesis, MIDI data, and acoustic piano in composition and live performance*. in *Society for Electro-Acoustic Music in the United States*. 1994. Middlebury, Vermont: SEAMUS.
6. Bolzinger, S. *DKompose: A Package for Interactive Composition in the Max Environment, Adapted To The Acoustic MIDI Disklavier Piano*. in *International Computer Music Conference*. 1992. San Jose: International Computer Music Association.
7. Brown, J., *Sex Machine* 1970, Polygram Records (CD #314 517 984-2): New York.
8. Caramia, T., *A Guide for Jazz Piano Harmonization*. 1983, San Diego: Neil A. Kjos Music Company.

9. Coniglio, M. *Introduction to the Interactor® language*. in *International Computer Music Conference*. 1992. San Jose: International Computer Music Association.
10. Cope, D., *Computers and Musical Style*. 1991, Madison: A-R Editions.
11. Dannenberg, R., *Real-Time Scheduling and Computer Accompaniment*, in *Current Directions in Computer Music Research*, M. Mathews and J. Pierce, Editors. 1989, The MIT Press: Cambridge, Mass.
12. Dannenberg, R. and B. Mont-Reynaud. *Following an improvisation in real-time*. in *International Computer Music Conference*. 1987. Urbana, Illinois: International Computer Music Association.
13. Davis, M., *Kind of Blue* 1959, Columbia: New York.
14. Delisle, N. and D. Garlan, *A Formal Specification of an Oscilloscope*. *IEEE Software*, 1990. 7(5): p. 29—36.
15. Deutsch, P., *Design reuse and frameworks in the Smalltalk-80 system*, in *Software Reusability, Volume II*, T. Biggerstaff and A. Perlis, Editors. 1989, ACM Press: p. 57–71.
16. Ebcioğlu, K. *An Expert System for Harmonizing Four-Part Chorales*. in *International Computer Music Conference*. 1986. Computer Music Association.
17. Ellinger, J. and M. Baker, *MiBAC™ Jazz Improvisation Software* 1990, MiBAC Music Software: Northfield, Minnesota.

18. Franco, S., *Hardware Design of a Real-Time Musical System* 1974, University of Illinois at Urbana-Champaign:
19. Giomi, F. and M. Ligabue. *An Interactive System for Music Improvisation*. in *International Computer Music Conference*. 1988. Cologne, Germany: International Computer Music Association.
20. Goodwin, C., *Conversational Organization: Interaction between Speakers and Hearers*. *Language, Thought, and Culture: Advances in the Study of Cognition*, 1981, New York: Academic Press.
21. Gridley, M., *Jazz Styles*. 1985, Englewood Cliffs: Prentice-Hall.
22. Hall, E., *Improvisation as an Acquired, Multilevel Process*. *Journal of Ethnomusicology*, 1992. 36(2): p. 223—235.
23. Hartmann, C., *Jazz Text: Voice and Improvisation in Poetry, Jazz and Song*. 1991, Princeton: Princeton University Press.
24. Heath, C. and P. Luff. *Collaborative Activity and Technological Design*. in *ECSCW*. 1991.
25. Hiller, L. and L. Isaacson, *Experimental Music*. 1959, New York: McGraw-Hill.
26. Jefferson, G., *List construction as a Task and Resource*, in *Interaction Competence*, G. Psathas, Editor. 1990, University Press of America: Lanham.

27. Joy, J. *Automated Linguistic Analysis of Musical Compositions*. in *International Computer Music Conference*. 1991. Montréal: International Computer Music Association.
28. Kaplan, S., A. Carroll, and K. MacGregor, *Supporting Collaborative Processes with ConversationBuilder*. ACM Special Interest Group in Office Information Systems, 1991. 12(2/3).
29. Lerdahl, F. and X. Chabot. *A Theory of Poetry as Music and its Exploration through a Computer Aid to Composition*. in *International Computer Music Conference*. 1991. Montréal: International Computer Music Association.
30. Lerdahl, F. and R. Jackendoff, *A Generative Theory of Tonal Music*. 1985, Boston: MIT Press.
31. Levitt, D., *A Melody Description System for Jazz Improvisation* 1981, Massachusetts Institute of Technology:
32. Loy, G., *Musicians make a Standard: The MIDI Phenomenon*. *Computer Music Journal*, 1985. 9(4).
33. Luff, P., N. Gilbert, and D. Frolich, ed. *Computers and Conversation*. *Computers and People Series*, eds. B. Gaines and A. Monk. 1990, Academic Press: London. 284.
34. Martirano, S., *An Electronic Music Instrument which combines the Composing Process with Performance in Real-Time*. 1971, unpublished: University of Illinois at Urbana-Champaign.

35. Martirano, S., *Everything Goes When the Whistle Blows* 1988, Centaur: Urbana.
36. Monson, I., *Music Interaction in Modern Jazz: An Ethnomusicological Perspective* 1991, New York University:
37. Norman, M. and P. Thomas, *The Very Idea: Informing HCI Design from Conversation Analysis*, in *Computers and Conversation*, P. Luff, N. Gilbert, and D. Frolich, Editors. 1990, Harcourt Brace Jovanovich: London. p. 51–65.
38. ParcPlace, *Objectworks\Smalltalk* 1992, ParcPlace Systems: Sunnyvale, California.
39. Pennycook, B., *The PRAESCIO Series—Composition-Driven Interactive Software*. *Computer Music Journal*, 1991. 15(3).
40. Perlman, A. and D. Greenblatt, *Miles Davis Meets Noam Chomsky: Some Observations on Jazz Improvisation and Language Structure*, in *The Sign in Music and Language*, W. Steiner, Editor. 1981, University of Texas Press: Austin.
41. Pope, S. *A Tool for Manipulating Expressive and Structural Hierarchies in Music*. in *International Computer Music Conference*. 1991. Montréal: International Computer Music Association.
42. Puckette, M. *The Patcher*. in *International Computer Music Conference*. 1988. International Computer Music Association.
43. Race Street Jazz Conspiracy, *Jazz Live* 1994, WILL-FM: Urbana, Illinois.

44. Ramos, R., *The use of Improvisation and Modulation in Natural Talk: an alternative approach to Conversational Analysis*. *Studies in Symbolic Interaction*, 1978. 1(1): p. 319—337.
45. Randel, D., *Harvard Concise Dictionary of Music*. 1987, Cambridge, Massachusetts: Harvard University Press.
46. Rowe, R., *Interactive Music Systems*. 1993, Cambridge: M. I. T. Press.
47. Sacks, H., E. Schegloff, and G. Jefferson, *A Simplest Systematics for the Organization of Turn-taking for Conversation*. *Language: Journal of the Linguistic Society of America*, 1974. 50(4): p. 39.
48. Scaletti, C., *The Kyma/Platypus Computer Music Workstation*, in *The Well-Tempered Object: Musical Applications of Object-oriented programming*, S. Pope, Editor. 1991, M. I. T. Press: Boston.
49. Scaletti, C. *Kyma: An Object-oriented Language for Music Composition*. in *International Computer Music Conference*. 1987. International Computer Music Association.
50. Steedman, M.J., *A Generative Grammar for Jazz Chord Sequences*. *Music Perception*, 1984. 2(1): p. 52–77.
51. Suchmann, L., *Plans and Situated Actions*. 1987, Cambridge: Cambridge University Press.

52. Suhor, C., *Jazz Improvisation and Language Performance: Parallel Competencies*. Et cetera, 1986. 1(1): p. 133—140.
53. Thomas, M.T.e.a. *Cantabile: a rule-based system for composing melody*. in *International Computer Music Conference*. 1989. Columbus: International Computer Music Association.
54. Titon, J., *Early Downhome Blues: A Musical and Cultural Analysis*. 1977, Urbana: University of Illinois Press.
55. Turkle, S., *The Second Self: Computer and the Human Spirit*. 1984, New York: Simon and Schuster.
56. Vercoe, B. *The Synthetic Performer in the Context of Live Performance*. in *International Computer Music Conference*. 1984. Computer Music Association.
57. Weeks, P., *Musical Time as a Practical Accomplishment: A Change in Tempo*. *Human Studies*, 1990. 13(4): p. 323—359.
58. Wessel, D. *Improvisation with Highly Interactive Real-Time Performance Systems*. in *International Computer Music Conference*. 1991. Montréal: International Computer Music Association.
59. Winkler, T. *FollowPlay: A MAX Program for Interactive Composition*. in *International Computer Music Conference*. 1992. San Jose: International Computer Music Association.
60. Winograd, T., *Linguistics and the Computer Analysis of Tonal Harmony*. *Journal of Music Theory*, 1968. 12((Spring)): p. 2—49.

61. Wirfs-Brock, R. and R. Johnson, *Surveying Current Research in Object-Oriented Programming*. Communications of the ACM, 1990. 33(9): p. 104—124.
62. Wolff, H., ed. *Webster's New Collegiate Dictionary*. 1977, Merriam Webster: Springfield, Massachusetts.
63. Zicarelli, D., *M and Jam Factory*. Computer Music Journal, 1987. 11(2).

Vita

William F. Walker was born on April 28th, 1966 in Urbana, Illinois. He received his Bachelor's and Master's degrees in Computer Science from the University of Illinois at Urbana-Champaign. At the University of Illinois he was a graduate research assistant for the CERL Sound Group and Sal Martirano of the School of Music. He was also a teaching assistant for the Department of Computer Science. A list of his publications follows:

- “ImprovisationBuilder: Improvisation as Conversation” in *Proceedings of the 1992 International Computer Music Conference*. Co-authored with Kurt Hebel, Salvatore Martirano, Carla Scaletti.
- “Extending the McAulay-Quatieri Analysis for Synthesis with a Limited Number of Oscillators” in *Proceedings of the 1992 International Computer Music Conference*. Co-authored with Kelly Fitz, Lippold Haken.
- “Programming Models for Digital Samples Generation” in *Array*, Vol. 11, No. 2, 1992.
- “Computer Science Graduate Student's Guide To Life,” a comprehensive guide for new graduate students. Distributed by the University of Illinois Department of Computer Science since 1991.
- “Lime™ Music Notation Software for the Macintosh®” in *Proceedings of the 1991 International Computer Music Conference*. Co-authored with Lippold Haken, Dorothea Blostein.
- “Kiwi: A Parallel System for Software Sound Synthesis” in *Proceedings of the 1989 International Computer Music Conference*.