An Object-Oriented Real-Time Simulation of Music Performance Using Interactive Control

Thesis by

Lounette M. Dyer

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1991

(Defended 28 May 1991)

to Zona Delphine

Acknowledgments

I would like to express my deepest gratitude to Carver Mead for his many contributions to both my professional and personal development. I, like so many others, have been greatly inspired by his vision, wisdom, and humanity.

I owe a special thanks to Max Mathews and Al Barr. Max gave me day-to-day encouragement, guidance, and inspiration while writing this thesis and shared many wonderful hours of stimulating conversation. Al supported me and was there for me when I needed it most. Al also helped me make a thesis out of a hundred pages of words and made the defense of this thesis an enjoyable experience. I would also like to thank Chuck Seitz for being on my thesis defense committee and for his continued support over the years.

I am deeply indebted to Richard Steiger for his constant support and encouragement. He served as a sounding board while I was developing the ideas in this thesis and was the source of many valuable insights.

I would like to also thank Adele Goldberg for reading an early draft of this thesis. Her experience as a writer and editor were invaluable in helping me organize the document. I also want to thank Adele for gently prodding me until the thesis was finally finished.

I feel very fortunate to have had the opportunity to work on this research in two very special environments: at Caltech in Carver Mead's research group and also at CCRMA at Stanford University. I would like to thank the members of "Carver's group" for creating an exciting environment to work in. I would also like to thank John Chowning and the staff of CCRMA for providing me with a stimulating environment and a home while I completed my research.

I also owe a special thanks to many friends for their patience and understanding while I was finishing this thesis. Without their support I might have moved to Brazil to study hand drumming instead.

This research was supported by the System Development Foundation and Hewlett-Packard.

Abstract

This thesis presents a software architecture for interactive control of real-time music performance by sound synthesizers. The architecture is based on a model of a real world orchestra performance. An object-oriented paradigm is used to define objects that are one-to-one with the real world entities: a conductor, performers, instruments, a score, and parts. Methods are defined for the objects to simulate some of the dynamic behavior of the conductor and performers during the performance. A detailed design of each of the objects is presented, and the objects and their real world counterparts are compared. An abstract digital music representation is defined to represent the musical composition that is to be performed by the system. The device independence of the representation is highlighted. A real-time control mechanism is described that allows a human user to control various aspects of the performance in musically expressive ways. The model is implemented in a system called ZED, which has been shown to simulate some of the dynamic behavior of a live orchestra. Issues concerning the trade-off between runtime efficiency and runtime flexibility are addressed in detail, as well as how these issues affect real-time scheduling. Optimization techniques are presented that help insure timeliness. ZED provides two levels of programmability: the orchestration of a score and the interpretation of real-time inputs can be defined in a configuration file; and new methods and subclasses can be added to the system to provide new functionality. The architecture, coupled with the object-oriented features of inheritance and encapsulation, are shown to provide the system with flexibility and extensibility, making ZED an ideal platform for developing and evolving real-time interactive control applications.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Twenty-five centuries ago Pythagoras and his disciples developed the first musical scales through the mathematical measurement of vibrating strings. These experiments became the basis of Western music and tonality [Grout, 1973]. For hundreds of years, musical instruments were invented and their tunings and timbres were refined so that the instruments could be played together in ensembles. At the same time people developed skills to play these instruments. The result of this evolution is the modern symphony orchestra, which has remained virtually unchanged for the last one hundred or so years.

Musical instrument invention saw very little activity until three decades ago when a new generation of musical instrument inventors developed the first electronic musical instruments. Two separate schools of inventors evolved: those who developed analog sound synthesizers that operated in real-time; and those who developed digital sound synthesizers that did not operate in real-time. The digital computer revolution was just beginning and the analog synthesizers were to go the way of analog computers. Digital computer technology would later make it possible for digital sound synthesizers to operate in real-time, thus offering the interactive capabilities of the analog synthesizers.

The first digital sound synthesizers were computer programs that computed sound samples that could be played back through digital-to-analog converters (DACs). These programs were written for the state-of-the-art computers: large, expensive, batch mainframe computers that took tens of minutes to compute a single second of sound. Because of the limited availability of computers and the *non-real-time* nature of the synthesis, evolution in the field was slow. Computer music compositions were generated in segments by a computer and recorded onto audio tape. Audio tapes lack the real-time spontaneity and expressiveness that audiences are accustomed to experiencing in live music performance. Therefore, early computer music compositions were not really *performed* but were *rendered* and played back in the same way that static frames of visual images can be created with computer graphics and stored on film or video tape for future animation playback. The primary difference between playback of a recording and a performance is that recordings are identical every time they are played.

As computer technology evolved, in particular, VLSI technology, music synthesis algorithms evolved to take advantage of the additional computational capacity. *But faster sound synthesis hardware did not change the way computer music was thought about and produced.* Faster sound synthesis hardware only changed the quantity of computer music and the quality of the sounds that could be produced. It was not until sound samples could be computed in *real-time* that the computer technology *revolution* reached computer music.

Max Mathews described the changing emphasis in the field of computer music that resulted from real-time sound synthesis in the following way:

> **The 'problems' of computer music are no longer that of technology but rather of our ability to control it [Mathews, 1989a].**

Thus, now that real-time sound synthesis hardware was available, how could we control the technology to make *music*, rather than simply sound?

The first computer programs that were written to control real-time synthesis hardware were similar to non-real-time synthesis software. Lists of parameters were created that defined the sound. These parameters were sent to the sound synthesizer in real-time to create the performance. Such programs still lacked the expressive and spontaneous components of live performance. The performance was still the same every time just as with the audio tape performances of non-real-time sound synthesis. In order to take full advantage of the real-time nature of the sound synthesis hardware, user interaction was required to provide the creative and spontaneous elements found in live orchestra performances. Therefore, in addition to real-time sound synthesis hardware, real-time input controllers were required to map a human user's physical gestures into real-time synthesis parameters for the sound synthesis hardware.

A standard interface protocol called the Musical Instrument Device Interface (MIDI) [MMA, 1987] was invented to allow a variety of controllers to be used to control parameters of many different synthesizers. Initially there were two basic types of controllers: musical instrument controllers such as piano keyboards, wind instruments, and string instruments; and special effects controllers such as sliders, wheels, and foot pedals. The musical instrument controllers allowed trained musicians to have direct control over a sound synthesis voice in much the same way that an acoustical instrument is played. The special effects controllers allowed other parameters of the sound to be controlled that could not be directly controlled with the instrument controllers.

More recently, a number of research projects have focused on unique and novel controllers. Examples include: the Daton [Mathews, 1989b], a mechanical device that detects three dimensions (x, y, and force) when it is struck; the Stanford Radio Baton (previously known as the "Stanford Radio Drum") [Boie and Mathews, 1989], a device that senses three-dimensional continuous motion of two sticks; the Polhemus sensor [Logemann, 1989], a device that senses a three-dimensional location and a three-dimensional orientation of a sensor; the Video Harp [Rubine and McAvinney, 1990], a device that responds to movements similar to those of harpists; and BioMuse [Knapp and Lusted, 1990], a system that senses biological signals, including eye movement, muscle flexing, and alpha brain waves. These controllers, when used in conjunction with computer software, allows an untrained musician to achieve subtle interactive control over a variety of independent sound synthesis voices.

Thus, with the advent of sophisticated real-time controllers, coupled with real-time sound synthesis hardware, the problem now is that of building software systems that will allow a user to expressively control many aspects of a real-time music performance by sound synthesis devices. Clearly the problem of real-time control of synthesis hardware has little in common with the early non-real-time sound synthesis programs. Despite this fact, much of the research to date in real-time music performance systems has focused on applying the concepts used in non-real-time software sound synthesis systems to the problem of real-time interactive control. But the concepts developed in non-real-time music synthesis do not extend to encompass the dynamic environment of interactive control.

This thesis presents a new software architecture that was inspired by Carver Mead's remarks concerning the VLSI revolution:

> **After an evolution of six or more orders of magnitude in the most important metrics of the underlying technology, we are still using the same conceptualization of computing that was common in the era of vacuum tubes and core memories. A quantitative improvement of many orders of magnitude makes a qualitative difference in the way one must conceptualize a field [Mead, 1983].**

The work described in this thesis introduces a new conceptual framework to the area of interactive computer music performance. A real world system that controls musical instruments in real-time—that of the live orchestra—provides a basis for thinking about the problem of controlling real-time sound synthesis hardware. The primary contribution of this thesis is the decomposition of the live orchestra into a model consisting of objects and methods that simulate some simple behaviors of a live orchestra. These objects and behaviors are implemented by a software simulation system called *ZED* that is used to validate the model. The resulting software architecture provides a general, extensible framework that is device independent and could be used as the basis for a variety of other real-time interactive control applications such as real-time animation and robotics.

ZED was designed using an object-oriented design methodology. We will show that the object-oriented concepts provided a natural paradigm for representing the components of a live orchestra. The resulting system has been shown to demonstrate responsiveness to a user's inputs. Furthermore, the system demonstrates some of the types of expressive control that a conductor has over a live orchestra. ZED can be used to create live, interactive concert performances of synthesized music. In addition, the system can be used by researchers to experiment with expressive musical control of computer generated sound in an effort to better understand sound synthesis models, and musical interpretation and expression.

A key component of the software architecture is the invention of a semantic digital music representation. This music representation provides many of the same features as the conventional music notation in use by composers of Western music for hundreds of years. These properties include instrument independence, extensibility, instrument specific extensions, and a clear separation between a note and its interpretation. These properties play an integral role in the success of ZED's design. The representation separates the representation of notes and their interpretation as a basis for affecting the interpretation with interactive control inputs.

Real-time music performance is an interesting computer science problem because it encompasses several research areas, including discrete event simulation, real-time systems, and object-oriented software design. Experimental implementations of the design were done in a variety of programming languages. Initially, portions of the system were implemented in Pascal [Jensen and Wirth, 1978] and C [Kernighan and Ritchie, 1978]. These languages did not provide sufficient data abstraction, extensibility, and polymorphy to make them practical for a system of this type. Then, the system was prototyped in Smalltalk [Goldberg and Robson, 1983]. Smalltalk provided an ideal environment for experimenting with various designs. The timeliness, however, was affected by the Smalltalk memory manager and Smalltalk was found to be unsuitable for actual real-time music performances. Objective-C [Cox, 1987; NeXT, 1989] was found to be an ideal compromise: it has basically the same semantics as Smalltalk so the prototype could be trivially ported from Smalltalk to Objective-C; Objective-C provides the ability to statically bind methods, thus increasing runtime efficiency; and Objective-C does not have a memory manager. The developer therefore has control over how the CPU cycles are spent. The Objective-C implementation demonstrates that object-oriented languages on modern workstations are suitable for implementing real-time systems with the timing requirements of music performance.

Previous Approaches

A number of real-time performance systems have been developed to date. One type of system supports low level MIDI *patching*—the routing of input events from MIDI controllers to parameters of a synthesis output device. A notable example is MAX [Puckette, 1986; Puckette, 1988], a MIDI patching application that allows a user to configure patches via a graphical user interface (GUI). Patches can be defined to use any input to control any synthesis parameter. MAX is based on an object model whereby objects, represented by boxes, receive inputs and generate outputs. Interobject communication is defined by messages, represented with lines between boxes. The user can program computations on the data, and can also write code for new objects using a conventional programming language such as C. MAX could be thought of as a visual programming language that allows a user to develop applications that manage the data flow of low-level MIDI data events.

Another type of real-time performance system is an *accompaniment system*. These systems extract timing information from inputs generated by a human user with a controller, and synchronize a synthesizer accompaniment with the user. These systems focus on the tight integration between a live performer and a sound synthesis accompaniment. Accompaniment systems are generally closed applications whose behavior cannot be changed by the user. The functionality of accompaniment systems could be programmed in MAX, but MAX is not in and of itself an accompaniment system.

One example of an accompaniment system is the Conductor Program [Mathews, 1989b]. The Conductor Program was written specifically for the Stanford Radio Baton. The Conductor Program allows a user to control the tempo by striking a surface with the baton. The program knows on which beats to expect the inputs and synchronizes the accompaniment with the input when it arrives. Other inputs from the baton can be used to control other aspects of the performance such as phrasing, balance, and dynamics. This system has been used in numerous concert performances, primarily with a live vocalist and a live performer of the baton. The system has demonstrated that it is possible to obtain expressive musical interpretation with synthesized sounds.

Two other accompaniment systems are those of Bloch and Dannenberg [Bloch and Dannenberg, 1985] and Vercoe and Puckette [Vercoe and Puckette, 1985]. Both of these systems use a musical instrument for input. The timing of the inputs from the performer are used to control the tempo of the accompaniment. But unlike the Conductor Program, these systems use the pitches as well as the time of the inputs. The system can adapt to errors in the live performer's performance, such as pitch mistakes and skipped notes.

Another type of software system that has been used for real-time music performance is the object-oriented tool kit. Tool kits are not complete applications, but rather they provide a basis for a composer/ programmer to develop real-time music performance applications. They are similar to MAX in this regard, but differ from MAX in that they require that a textual object-oriented programming language be used to program them. An example of an object-oriented tool kit that can be used with interactive control is the NeXT Music Kit [Jaffe, 1989]. The Music Kit can be used to build applications in Objective-C on the NeXT Computer. Music Kit applications can play score files on MIDI synthesizers and instruments defined in the NeXT DSP synthesis instrument library. MIDI input can be used to control synthesis but the tool kit does not have explicit abstractions for configuring real-time performances. Like MAX the Music Kit can be programmed to track a score, but no explicit notion of an accompaniment is provided.

These real-time performance systems have different goals. The accompaniment systems were designed to enable the concert performance of computer music under the real-time control of a live user. These systems use a specific controller and specific synthesis devices. MAX was designed as a tool for non-programmers to specify and execute real-time music performances with a variety of different controllers and synthesis devices. The Music Kit was designed as a framework for programmers to develop real-time performance applications for the NeXT Computer using MIDI and DSP synthesis.

ZED's design is based on the decomposition of the live orchestra performance and the implementation of objects that correspond to the live orchestra components. ZED's primary purpose is to validate the proposed model of a live orchestra performance. In addition to being a simulation system that can be used for concert performances, ZED also provides a framework for experimenting with expressive musical control of sound synthesis devices.

As a music performance system, ZED provides a number of features of the other systems. ZED provides the programmability and extensibility of MAX and the Music Kit. ZED also provides tight integration of a live performer and a synthesis accompaniment like that provided by the accompaniment systems. ZED defines an abstract digital music representation for scores that none of the other systems have. ZED also provides specific abstractions for the performance that are well beyond those of any of the other systems. For example, a conductor object provides a mechanism for routing real-time inputs to other objects in the performance. Performer objects are used that interpret abstract musical symbols and map them into synthesis parameters. Instrument objects are defined that hide the specifics of the synthesis device. This approach allows the scores and the real-time patches to be device independent. The specialization of performer objects through subclassing and message overriding provides ZED with easy extensibility. In addition, the use of performer objects creates an evolutionary path for the system so that new technologies and new knowledge of how to control the technology can be easily incorporated.

A more detailed description of these other systems and a more complete comparison with ZED can be found in *Appendix C* on page 102.

Thesis Overview

Chapter 2, *Background and Terminology*, provides a brief introduction to object-oriented programming concepts and terminology for readers who are not familiar the field. Models and simulation are briefly discussed and an overview of the issues involved in real-time control are presented. Chapter 3, *Music Performance Model*, describes the live orchestra. A model of the live orchestra is proposed by identifying, analyzing, and abstracting the components of the live orchestra into a set of object definitions and behaviors. Chapter 4, *Architectural Design*, describes the architecture of the software simulation system called *ZED* that implements the performance model. Chapter 5, *MUSE: A Digital Music Representation*, defines a generic score representation as a set of object definitions for the symbols that are used to represent printed music. Chapter 6, *Real-time Performance*, describes how interactive control inputs are used to control the performance. And finally, *Conclusions* summarizes the thesis and discusses ideas for future work.

A number of appendices are also provided. *Appendix A*, *Real-Time Scheduling*, provides a detailed design of ZED's scheduler, as well as an overview of scheduling algorithms used in other real-time music performance systems. *Appendix B*, *Score Files*, defines the MUSE score file format and also discusses the mapping of MIDI score files to the generic MUSE digital music representation. *Appendix C, Other Approaches*, gives an overview of some of the other real-time music performance systems developed to date. These systems are compared and contrasted with ZED. *Appendix D, MIDI Specification*, provides tables that show the details of the MIDI specification.

References

Bloch, J. J. and Dannenberg, R. B. Real-Time Computer Accompaniment of Keyboard Performances. In Truax, B. (ed), *International Computer Music Conference at Simon Fraser University*. San Francisco, CA: Computer Music Association, 1985, p. 279.

Boie, B. and Mathews, M. V. The Radio Drum as a Synthesizer Controller. In Wells, T. and Butler, D. (eds), *International Computer Music Conference at Ohio State University*. San Francisco, CA: Computer Music Association, 1989, p. 42.

Cox, B. J. *Object-oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley, 1987.

Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.

Grout, D. J. *A History of Western Music*. New York, NY: W. W. Norton, 1973, pp. 27-34.

Jaffe, D. A. Overview of the NeXT Music Kit. In Wells, T. and Butler, D. (eds), *International Computer Music Conference at Ohio State University*. San Francisco, CA: Computer Music Association, 1989b, p. 135.

Jensen, K. and Wirth, N. *Pascal User Manual and Report*. New York, NY: Springer-Verlag, 1978.

Kernighan, B. W. and Ritchie, D. M. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

Knapp, R. B. and Lusted, H. S. *A Bioelectric Controller for Computer Music Applications*. Computer Music Journal, 14(1):42-47, 1990.

Logemann, G. W. Experiments with a Gestural Controller. In Wells, T. and Butler, D. (eds), *International Computer Music Conference at Ohio State University*. San Francisco, CA: Computer Music Association, 1989, p. 184.

Mathews, M. V. *Personal communication*, 1989a.

Mathews, M. V. The Conductor Program and Mechanical Baton. In Mathews, M. V. and Pierce, J. R. (eds), *Current Directions in Computer Music Research.* Cambridge, MA: MIT Press, 1989b, p. 263.

Mead, C. A. VLSI and the Foundations of Computation. In Mason, R. E. A. (ed), *Information Processing 83.* Elsevier Science Publishers B. V. (North-Holland), 1983, p. 271.

MIDI Manufacturers Association, *MIDI Musical Instrument Digital Interface Specification 1.0.* North Hollywood, CA: International MIDI Association, 1987.

NeXT, *System Reference Manual.* Menlo Park, CA: NeXT Inc., 1989.

Puckette, M. Interprocess Communication and Timing in Real-time Computer Music Performance. In Berg, P. (ed), *International Computer Music Conference at Royal Conservatory, The Hague, Netherlands.* San Francisco, CA: Computer Music Association, 1986, pp. 43-46.

Puckette, M. The Patcher. In Barlow, C., Lischka, C. and Pannes, M. (eds), *International Computer Music Conference at GIMIK, Cologne, West Germany.* San Francisco, CA: Computer Music Association, 1988.

Rubine, D. and McAvinney, P. *Programmable Finger-tracking Instrument Controllers.* Computer Music Journal, 14(1):26-41, 1990.

Vercoe, B. and Puckette, M. Synthetic Rehearsal: Training the Synthetic Performer. In Truax, B. (ed), *International Computer Music Conference at Simon Fraser University.* San Francisco, CA: Computer Music Association, 1985, p. 275.

Chapter 2

Background and Terminology

This chapter presents a brief introduction to the concepts and terminology of object-oriented programming. Then, the concepts of modeling and simulation are defined as a basis for thinking about real-time music performance. In the final section, an overview of the issues involved in real-time systems is presented.

In this thesis, Smalltalk classes and methods are used to define objects and algorithms [Goldberg and Robson, 1983]. Class names appear in the **Helvetica bold font**. Instance variable names, method names, and Smalltalk code appear in the Helvetica font. Parameters are shown in the Helvetica underline font. The syntax for defining class hierarchies is a list of class names each followed by a pair of parenthesis containing instance variable names. A class name that occurs indented below another indicates a subclass.

Object-Oriented Programming

Object-oriented programming evolved from the programming language Simula which used objects and messages for defining simulations [Nygaard and Dahl, 1966]. The object and message paradigm provides the primary features of data abstraction, encapsulation and inheritance. The data abstraction provided by the object model and the message passing paradigm is important because it allows software to closely reflect real world situations. In addition, abstraction helps manage complexity because details can be hidden below high level semantic interfaces. Application *openness* and *extensibility* are fundamentally supported by object-oriented languages through inheritance and encapsulation. These features facilitate rapid prototyping and provide the ability to adapt systems to incorporate new technology and new requirements.

The following section presents the terms and concepts of object-oriented programming in general, and Smalltalk in particular. The material below was largely taken from the book *Smalltalk-80: The Language and its Implementation* [Goldberg and Robson, 1983].

Terminology

An *object* represents something that exists in the real world and consists of private data and a set of operations that can access that data. The data is private to the object and can only be manipulated by the object's own operations. A *message* is a request for an object to execute one of its operations. This operation is called a *method* and the command carried by the message is called a *selector*. The *receiver*, the object that a message is sent to, determines how to carry out the requested operation. The set of messages that an object responds to is called its *behavior*. The behavior defines the object's interface to the rest of the objects in the system. The only way to interact with an object is through this interface. Because the implementation of one object cannot depend on the internal details of other objects, only on the messages that they respond to, the objects and messages can be used to facilitate modular design of software.

A *class* describes the implementation of a set of objects that all represent the same kind of component. The individual objects described by a class are called its *instances*. A class describes the form of its instances' private data and how they carry out their operations. An object's private properties are a set of *instance variables* that make up its private data and a set of *methods* that describe how to carry out its operations. *Subclasses* of existing classes can be defined that *inherit* the instance variables and methods of the *superclass*. A subclass can define a new method with the same selector as a method in a superclass. This is called method *overriding*. In addition, a subclass may define new messages that its instances will respond to that will not be understood by instances of the superclasses. An *abstract* class is a class that has no instances but is the root of a hierarchy of classes that share basic semantics. A *concrete* class is a class that has actual instances.

Models and Simulation

A *model* can be defined as "a small representation of a planned or existing object" [Webster, 1979]. In the computer science field, a computational model can be defined as a computer representation of a planned or existing object. Models of physical objects like buildings or automobiles are generally smaller and abstracted from the real objects. Similarly, computer models are also abstracted and omit some of the detail while still maintaining the basic properties of the modeled object.

The word *simulate* means "to act or look like" [Webster, 1979]. Computer simulations generally model situations that change over time and often have actions or *events* that must be synchronized with some notion of time. A model can be implemented on a computer to *simulate* the behavior of the modeled system over time. Computer simulations provide a framework in which to understand the simulated situation.

Sometimes the notion of time is itself simulated. There are a number of ways to represent the actions of simulated objects with respect to real or simulated time. In one approach, a clock runs and at each tick of the clock, all objects are given the opportunity to take any desired action. Alternatively, the clock can be moved forward according to the time that the next event will take place. In this case, the system is driven by the next discrete action or event scheduled to occur. The implementation of a simulation using this approach depends on maintaining a queue of events (managed by a *scheduler*) that are ordered in time. When an event is completed, the next event is taken from the queue and the clock is moved to the event's time. This type of simulation is called *event driven*. In event driven simulations, a collection of independent objects exist, each with a set of tasks to do, and each needing to coordinate its activity's times with other objects in the simulated situation.

Real-Time Systems

A great deal of engineering research has focussed on understanding the nature of real-time systems. Real-time systems consist of a system that is being controlled, and a system that controls it. The controlled system has an *environment* in which the computer software of the controlling system interacts with the controlled system. Real-time systems are different from other computer software systems in that the correctness of the system depends not only on the logical result of the computation, but also on the system's *timeliness*—the time that the results are produced [Stankovic and Ramamritham, 1988]. Real-time systems are used extensively in the world for such things as airplane flight control, manufacturing process control, and robotics. Thus, real-time systems clearly must be fast and predictable, reliable and adaptive to their environment.

Real-time systems such as those listed above are often referred to as *hard* real-time systems. They are characterized as having catastrophic consequences if the logical or timing constraints of the system are not absolutely met. Applications in music performance are not *hard* as there are not catastrophic implications of errors such as playing a note slightly early or late. Musicians are rarely fired for such small imperfections. (There are, however, mistakes that may be catastrophic to a musician's career, such as a misplaced cymbal crash in the middle of a pianissimo aria!) In addition, real-time music performance systems may be thought of as *firm* in that it is important that events happen on time but the penalty for not being precisely on time is not enormous. Real-time music performance systems have some flexibility in timing because, according to psychoacousticians, onsets of musical notes that are separated by as much as 30 ms. are perceived by the audience as simultaneous [Rasch, 1978]. Many other aspects of the performance other than note onsets have even more relaxed timing constraints. These are considered *soft* constraints. *Criticalness* is the measure of how critical an operation is and how urgent is it that it happen at a precise time. Many aspects of hard real-time systems have a very high degree of criticalness. Because of this, hard real-time systems are much more difficult to design, simulate, and implement than real-time music performance systems. The work presented in this thesis does not address the timing demands placed on hard real-time systems. Only the timing constraints required to perform compositions of the complexity of a symphony in real-time are addresses in this thesis.

Real-time systems have explicit timing constraints attached to tasks that the system must accomplish. Some form of priority scheduling is used for the task, where the time constraint and criticalness are mapped into a single factor, namely the *priority*. Highly critical tasks typically occur at a lower frequency, thus reducing the contention for computing resources and insuring timeliness. Some real-time systems are *periodic* in that they perform tasks at regular intervals. Music performance systems are instead *aperiodic* because notes and interactive inputs do not necessarily happen at regular intervals.

Real-time systems can be *static* or *dynamic*. In static systems all events are known before runtime and *early* or *static* binding is used to precompute all values. Such systems are inflexible at runtime and do not respond to feedback or interactive input but have very low runtime overhead, making it easier to insure timing correctness. A static real-time music performance system would be one in which a score is compiled into precise synthesis parameters and the precise time that they are to be sent before the performance begins. Such a performance would be virtually the same as playing an the audio tape performances of non-real-time music because it would be identical every time. Dynamic systems have greater runtime overhead, but are more flexible at runtime because they use *late* or *dynamic* binding. A dynamic real-time music performance system is responsive to interactive input from a user and may also create new events during the performance, but such an approach may make it difficult to achieve the timing constraints.

An interesting property of real-time systems is that you can trade timeliness for quality. That is, if the system's response to an input can be delayed, the time can be used to compute a more accurate value for the input or to process the input more completely. In implementation terms this trade-off is shown in *responsiveness*. Therefore, in order to maintain both responsiveness and timeliness, the system must be optimized to begin the computation of the synthesis parameters at a time that precedes the time that the update is to be made by the amount of time the computation takes.

State-of-the-art software engineering methodologies, and object-oriented software engineering in particular, have introduced features such as modularity, abstract data types, and message passing. These features are being widely used for building complex, non-real-time applications that are maintainable and extensible over projected long lifetimes. These features are often perceived by researchers in real-time systems as being in conflict with real-time requirements. This thesis presents a real-time system that uses an object-oriented paradigm providing modularity and abstraction, while still providing the level of timeliness required by real-time music performance.

References

Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Reading, MA: Addison-Wesley, 1983.

Dahl, O. J. and Nygaard, K. *Simula—an Algol-Based Simulation Language.* Communications of the ACM, 9(9):671-678, 1966.

Rasch, R. A. *The perception of simultaneous notes such as in polyphonic music.* Acustica, 40(1):21-33, 1978.

Stankovic, J. A., and Ramamritham, K. *Tutorial: Hard Real-Time Systems.* Washington, D.C.: Computer Society Press of the IEEE, 1988, pp. 1-11.

Webster, *Webster's New World Dictionary of the American Language.* New York, NY: Warner Books, 1979.

Chapter 3

Music Performance Model

The first section in this chapter presents an overview of the real world orchestra in a concert performance. The overall dynamics of the orchestra performance are described. The orchestra is then decomposed into components, each of which is analyzed. The final section presents a model of the live orchestra that defines objects that are one-to-one with each of the real world entities. This model is the basis of a simulation system called ZED that implements the model.

Orchestra Performance Analysis

A live orchestra consists of a group of musicians, called *performers*, that are coordinated by a *conductor*. Each performer has one or more pages of printed music called a *part* and an *instrument* used to generate sound. The conductor has a *score* consisting of pages of printed music and contains all of the parts.

Conductor and Performers

The *conductor* is a human that oversees the performance and has some means of communicating to the performers. The conductor can affect the performers' interpretation of their respective parts, control the balance of the ensemble, and coordinate the group dynamics and tempo. A *performer* is a human who reads a part, interprets the symbols in the part, and generates appropriate inputs for a musical instrument based on those symbols. In addition, the performer accepts and interprets input from a conductor and adjusts the performance accordingly. The performer's interpretation of the symbols in the part is a result of training and taste, the style of the composition, the conductor's input in rehearsal, the conductor's gestures during the performance, and the balance of the ensemble as heard by the conductor (which is affected by such things as the acoustics of the concert hall.) The performer's interpretation may vary slightly in different performances.

Instruments

A musical *instrument* is an acoustical device that responds to gestural inputs from a performer by generating an acoustic audio signal that reflects those inputs—the type of gestural input that an instrument responds to varies dramatically across instrument families. For example, string instruments are bowed or plucked, wind instruments are blown, and percussion instruments are struck. Performers develop skills that are specific to their particular instrument, as one would not generate musical sound by blowing on a triangle or plucking a flute.

Score and Parts

A *score* is the printed representation of a composer's musical ideas. A score consists of a collection of parts, one for each performer. Each part consists of a collection time ordered symbols, called *notes*, that describe to performers in a high-level, abstract representation, how the music should sound. The symbols in the parts are basically instrument independent—they are the same for all instruments. A part may also contain symbols that are specific to an instrument such as pedal markings for piano and bowings for string instruments.

In addition to notes, the parts contain *interpretation symbols* defining how the notes are to be interpreted. Examples of interpretation symbols are tempo, dynamics, meter, and key. The score may also contain additional annotations and cues for the conductor specifying information that is to be communicated to the performers during the performance.

Performance Dynamics

An orchestra performance can be viewed as a real-time control system where the devices being controlled are acoustical musical instruments and the system controlling the instruments is a collection of human beings. The score and parts provide the definition of the composition that is to be played. There is a high degree of concurrency in a live orchestra performance. The performers play their instruments simultaneously while a conductor conducts them. Each performer provides direct control over their instrument with physical gestures. The conductor provides indirect control over the entire ensemble by communicating information to the performers with physical gestures that cause them to change the input to their instrument.

A diagram showing the components involved in a live orchestra performance are shown in *FIGURE 3.1*. The orchestra has two types of feedback loops. The performers listen to the sound they are generating and that of the other performers. They continuously adjust their sound by changing the volume, intonation, and other properties. The conductor listens to the sound as well and, through physical gestures, indicates adjustments to the performers for such ensemble properties as balance, dynamics, and tempo. The performer/ instrument feedback loop has a very short time constant because the performer can immediately respond to the sound. The conductor/performer/instrument feedback loop has a longer time constant because the conductor does not have direct control over the instruments. The conductor reacts to the sound with a

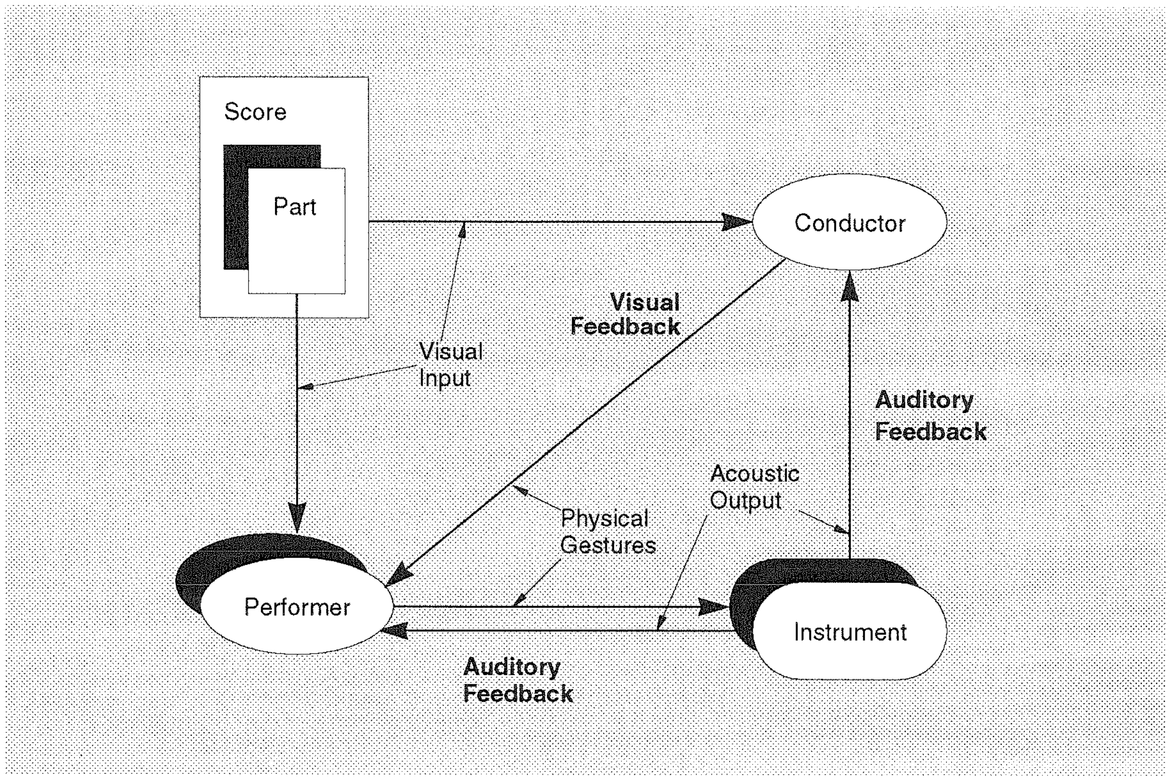


FIGURE 3.1 Orchestra performance dynamics.
Each performer receives visual input from a printed part and the conductor receives visulal input from the printed score. Each performer provides outputs to their instrument based on the part and visual input from the conductor. The acoustic output of the instruments provides auditory feedback for the performers and the conductor. The black drop shadows indicate that the performance has multiple performers, each with one part and one instrument.

physical gesture that is communicated to the performers. The performers see the gestures, interpret them, and adjust their sound accordingly.

Orchestra Performance Model

A model of orchestra performance can be defined that has objects representing the conductor, performers, instruments, score, and parts. There are, of course, many aspects of the live performance that cannot currently be modeled with a computer such as human hearing and music understanding. Although it is possible to extract some types of information from acoustical signals in real-time, this is a very hard problem, particularly for polyphonic music, and is currently unsolved. In addition, it is not currently possible to build computer models of human vision that can read printed music and understand physical gestures of a human conductor. Therefore, we will define a simplified model of the orchestra as shown in *FIGURE 3.2*.

Because a human auditory model cannot be directly implemented, the models for the conductor and performers are deaf. This eliminates the two feedback loops of the live orchestra. The performance model

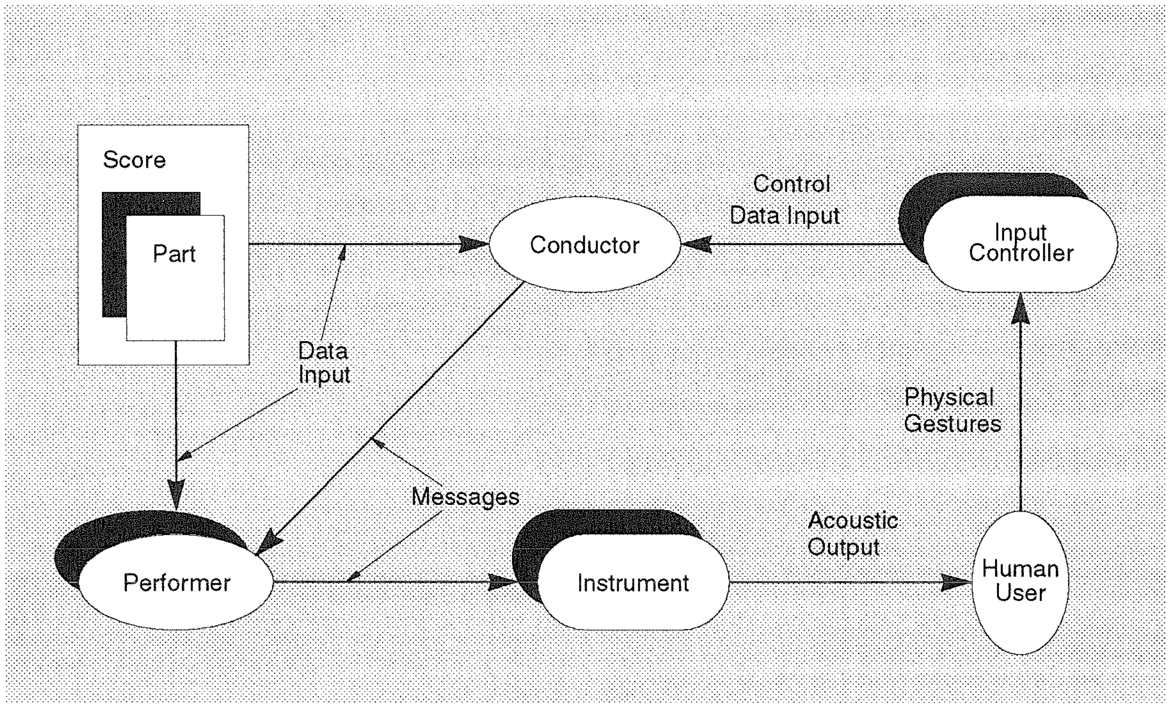


FIGURE 3.2 Orchestra performance model dynamics.
The abstracted components are shown. Communication is done among the objects in the model via messages. Feedback is provided indirectly via a human user with an input controller device. The user listens to the performance and can affect the performance. The conductor object receives interactive control inputs from the user and carries out the user's wishes by sending messages to other objects.

instead relies on a human user to provide an auditory feedback loop to the system. The user listens to the performance and interactively affects the performance via one or more input controller devices. The input controllers translate the user's physical gestures into control information that is transmitted to the conductor. The conductor then passes the information on to one or more performers. The basic dynamics of the performance are not substantially changed because the one feedback loop can simulate the two of the live orchestra. This is because the user is given two types of control: direct low level control of the synthesis parameters provided by the performer; and high-level indirect control similar to that of the live conductor.

The human user plays another important role in the simulation by providing the musical spontaneity and expressiveness to the performance. It is not currently possible to completely model human musical creativity and expressiveness. Artificial intelligence techniques such as rule based systems could be used to model human performers more closely [Frydén and Sundberg, 1984]. The ZED system defined in this thesis is designed to allow such techniques to be incorporated into the system.

Conductor and Performer Objects

The conductor and performers are represented with objects that have methods that mimic some simple behaviors of their human counterparts. The conductor object has a score object and each performer object has a part object and an instrument object. Human performers read a part of music by transforming the visual image of symbols on the page into a mental representation. Conductor and performer objects emulate this behavior with methods that read a *score file* from a computer disk and build an internal *digital music representation* of the composition. The score file is read before the performance, thus removing the need for disk I/O during the performance. This is somewhat analogous to a human conductor and performers memorizing their respective score and parts so that they do not require the printed music during the performance.

There is typically only one conductor object in a performance. The conductor object is responsible for coordinating the performance. The conductor communicates with performer objects through *message passing*. At the time that a note is to be played or some other event is to happen, the conductor object *cues* the performer object by sending a message to the appropriate performer object. The conductor object also acts on behalf of the human user during the performance. When control inputs are received from the user, the conductor object interprets them and may send a message to one or more performer objects, or may act on the input itself.

An alternative to having the conductor object send a message to the performer objects for each note would be to use separate concurrent intercommunicating processes for the conductor and performers. The performer processes would keep their own time instead of waiting passively for the conductor's next cue. The event driven model is used in ZED instead of the process model because of its conceptual simplicity and because its implementation is substantially more efficient. The only artifact of the event driven model is that the conductor cues every note, whereas in the live orchestra, the conductor only cues each note during brief ***rubato*** sections. Otherwise, the live conductor typically only cues the performers on the beats because it would not be physically possible to cue every note.

The primary task of performers is to map the abstract score data (in their part) into inputs for their instrument. When the performer object receives a message from the conductor to play a note, the performer object *interprets* the note in the context defined by the interpretation symbols in the part, score, and other information from the conductor. This interpretation is similar to the interpretation done by human performers. Performer objects, like human performers in the live orchestra, must have specific knowledge of their particular instrument so that appropriate inputs can be computed. ZED uses a number of different types of performers, each of which is specialized for a particular synthesis instrument. The use of specialized performer classes for different instruments is analogous to the live orchestra where, for example, a musician trained only on trumpet is generally not proficient on timpani.

Instrument Objects

An instrument object is used to represent the acoustic instrument played by a human performer. The performer object computes inputs for the instrument object, and the instrument object causes sound to be produced. The instrument object provides the performer object with an abstraction of the physical interface that connects a workstation to a synthesis device. After all, human performers needn't understand the physics of an acoustical instrument in order to be able to play it. Just as there are different types of acoustical musical instruments in an orchestra, ZED has different types of synthesis devices and interfaces, and different instrument classes for each of them. Therefore, specific instrument classes are provided for each type of synthesis device.

Score and Part Objects

The printed score of the real world orchestra is represented in the model with a *digital music representation.* A digital music representation called *MUSE* was designed specifically for ZED to address the issues involved in real-time music performance. MUSE defines objects for the score and parts. In addition, MUSE objects are defined that are one-to-one with the symbols in a printed score. MUSE, like the common music notation used in printed scores and parts, is device independent and separates the notes from the interpretation context.

MUSE notes have a pitch, time, duration, and an optional articulation symbol such as ***accent***, ***tenuto***, or ***staccato***. The properties of a note are represented relative to an *interpretation context* that is the same as that used in a printed score and parts. This interpretation context is the basis of the real-time interactive control. The interpretation context consists of objects for tempo, dynamics, key, meter, and style. The note's pitch is relative to the key; the time is relative to the tempo; the duration is relative to the tempo and the style; and the note's articulation is relative to the dynamics, style, tempo, and meter. The performer objects compute synthesis parameters for their particular instrument by applying the interpretation objects to each note. The interpretation objects defined in a performer's part are *local* to that performer. The interpretation objects defined in the conductor's score are *global* and therefore affect all performer objects that do not have a local interpretation defined.

References

Frydén, A. and Sundberg, J. Performance Rules for Melodies. Origin, Functions, Purposes. In Buxton, W. (ed), *International Computer Music Conference at IRCAM, Paris, France.* San Francisco, CA: Computer Music Association, 1984, pp. 221-224.

Chapter 4

ZED Architectural Design

This chapter describes the architectural design of an object-oriented simulation system called *ZED*. ZED is an implementation of the orchestra model described in the previous chapter. In the first section, an overview of the system is presented. Then, class designs for each of the objects in the model are presented. The adaptability and extensibility of the design is demonstrated with examples of how to incorporate new sound synthesis hardware into the system.

System Overview

ZED is a real-time music performance system that simulates live music performance. ZED was designed using an object-oriented design methodology. A diagram of the computer workstation environment that ZED is implemented on is shown in *FIGURE 4.1*. A computer workstation has connected to it, one or more input controllers and one or more sound synthesis devices. The audio signals from the sound synthesis devices are mixed, amplified, and are heard through a loud speaker. A human user interacts with the system via the input controllers. The control inputs can be used to control aspects of the performance including the tempo, dynamics, balance, note articulation, transposition, and the starting and stopping of sequences.

Performance Definition

ZED performances are defined by three files: a *score* file, a *configuration* file, and a *patch* file. The score file contains score data for the composition that is to be performed. The configuration file defines the orchestration of the composition, that is, what instrument is to play each part and what performer is to play each instrument. The conductor and the types of input controller objects are also defined in the configuration

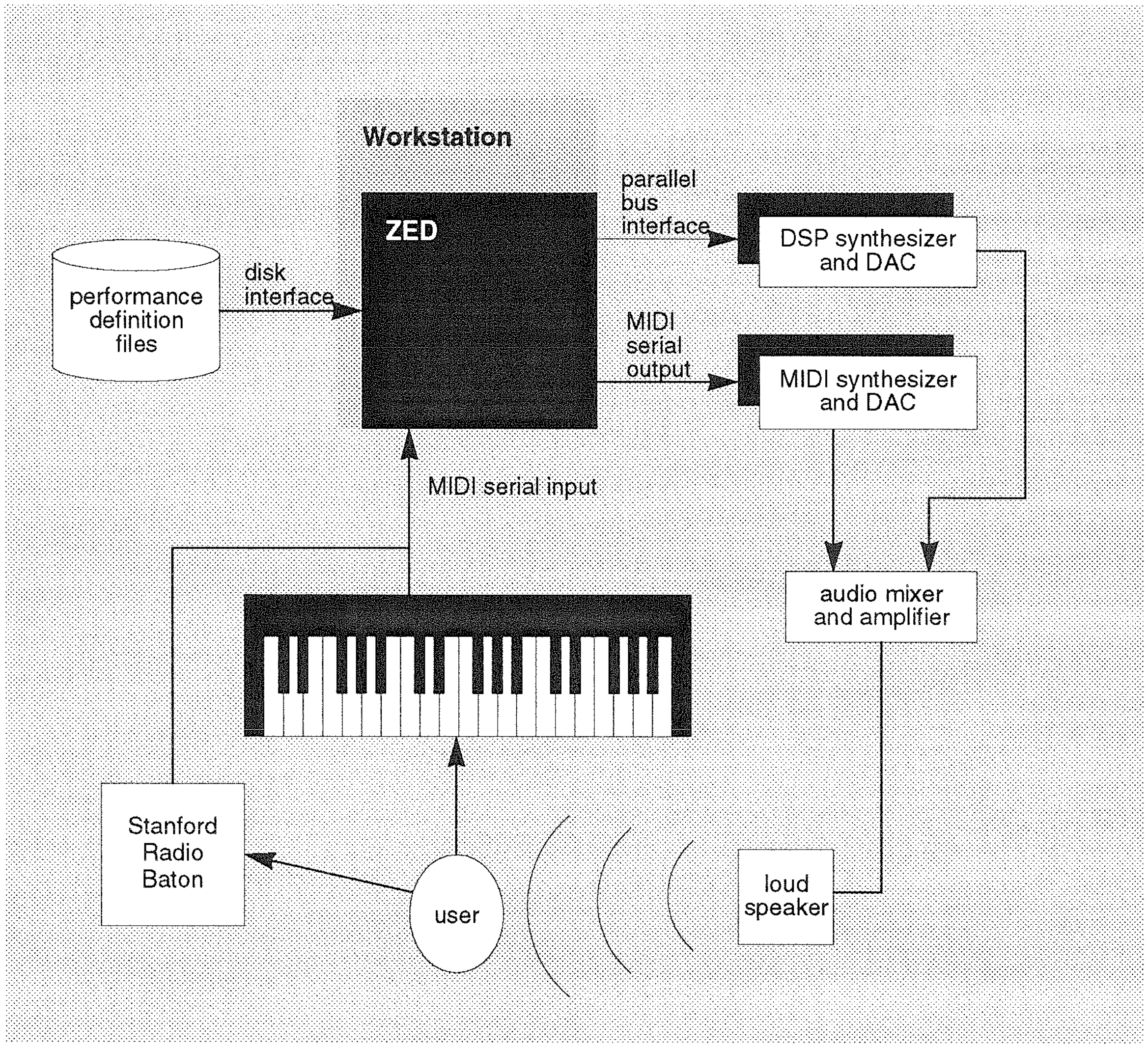


FIGURE 4.1 ZED real-time music performance workstation.
ZED uses the real-time inputs from the user to affect the performance by controlling the synthesizers in different ways. The performance files define the score and how the performance is to be controlled. Two MIDI input controllers are shown: the Stanford Radio Baton, and a MIDI keyboard. The two types of synthesizers shown are MIDI and DSP. The user listens to the performance transmitted through the loudspeaker and provides feedback to the system. The black drop shadows indicate multiple synthesis devices connected to the same interface: more than one MIDI synthesizer connected to the same serial port or more than one DSP on a single bus interface board.

file. The patch file contains a set of *patches* that define how the real-time inputs are to be interpreted and what action the conductor object is to take when specific inputs are received. Patches may be defined that affect the conductor or cause the conductor to send messages to one or more performer objects.

A given score can be performed in different ways: a different orchestration can be defined for a score by changing the configuration file to reassign the parts to different performers and instruments. In addition, a score with a particular orchestration can be performed in different ways by using different patch files that define different real-time control actions.

Performance Object Design

FIGURE 4.2 shows an overview of the basic performance objects defined in the model. The classes for the basic performance objects are subclasses of the abstract superclass **ZEDPerformanceObject**. All performance objects have a name instance variable that is used to uniquely identify the object. The name is used in the configuration files for specifying which instrument and part is associated with each performer object. The name is also used in the patch file to specify real-time control messages to be sent to the object. The class hierarchy is shown below. The instance variables in the *Helvetica italic font* indicate internal instance variables that are not shown in the diagram.

```
Object ( )
    ZEDPerformanceObject (name)
        Conductor (score performers inputControllers scheduler patches
                    interpretationContext currentEvents)
        Performer (conductor part instrument
                    interpretationContext scheduleMessage)
        Instrument ( )
        InputController (conductor)
```

Music Representation

An abstract digital music representation called *MUSE* is defined to represent the score and parts. The nature of this representation is such that it provides a common abstraction for musical ideas represented in a variety of different score file formats. Furthermore, the representation provides an abstraction from the specifics of the particular synthesis hardware that is to perform the composition. Thus, it is similar to the *common music notation* used by Western composers to notate music. It provides a single abstract representation that is, for the most part, independent of the particular instrument. *FIGURE 4.3* shows how score files are converted to the MUSE representation, that is then converted to device specific parameters by the performer objects. The generic representation simplifies the overall system architecture because the performance objects all operate on the same score objects regardless of the score file that the data came from.

Real-time Performance Overview

FIGURE 4.4 shows the ZED objects and the information flow through the system. The performer objects map the abstract score data to device specific synthesis parameters called *packets*. When real-time input is received during the performance, the conductor object acts on the input by sending messages to performer objects. These messages affect the mapping of the score data to the synthesis parameters, thus changing the sound of the performance.

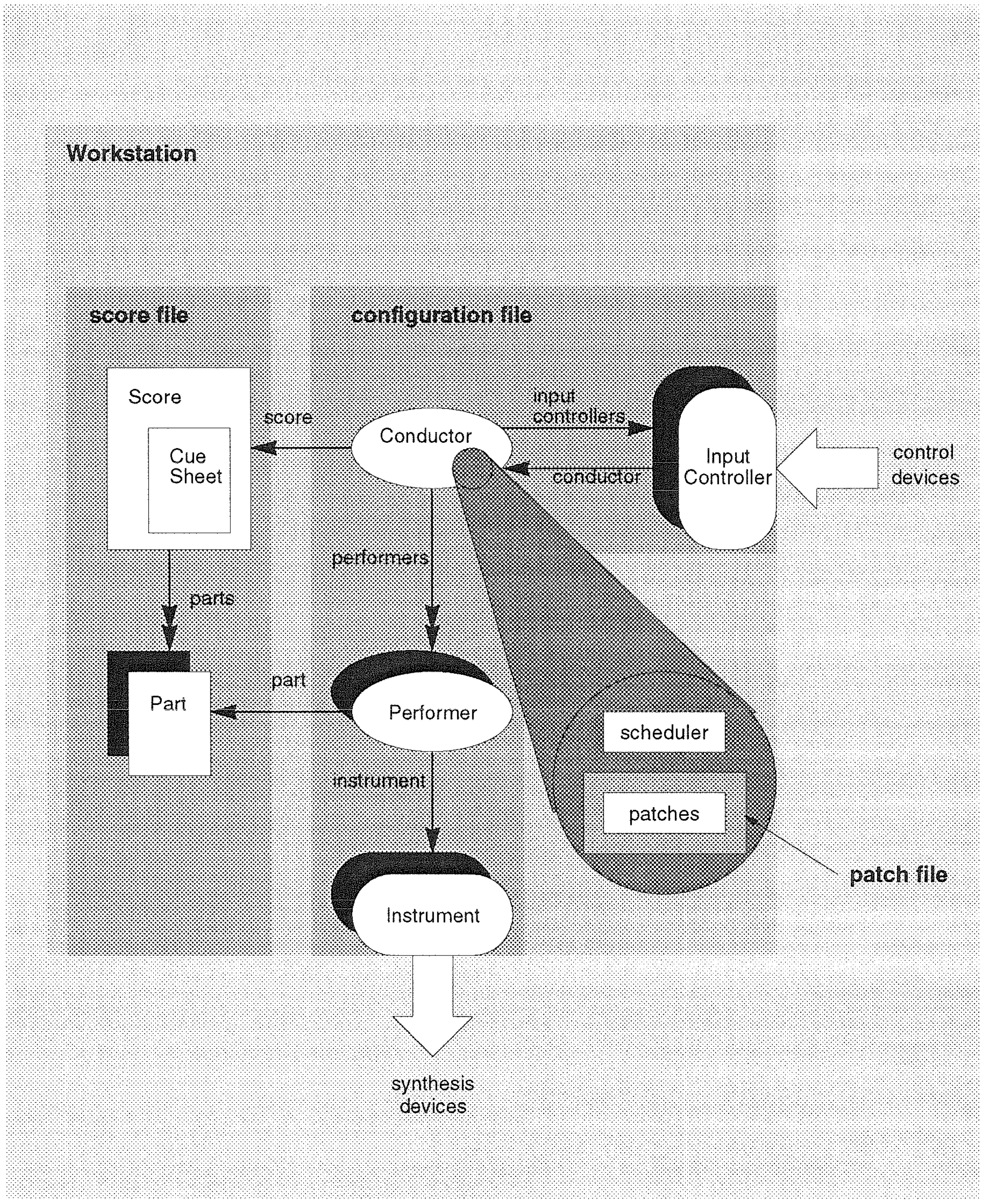


FIGURE 4.2 ZED performance objects.
The arrows are labeled with the corresponding instance variable name. The conductor has a scheduler that enables the coordination of events. The conductor also has a set of patches that define what actions are to be taken when real-time control inputs are received. Double arrows indicate a collection of objects of the specified type. Drop shadows indicate multiple instances of the class. An arrow between two objects with drop shadows indicates that each source object has one destination object. The gray arrows indicate how data enters and leaves the workstation.

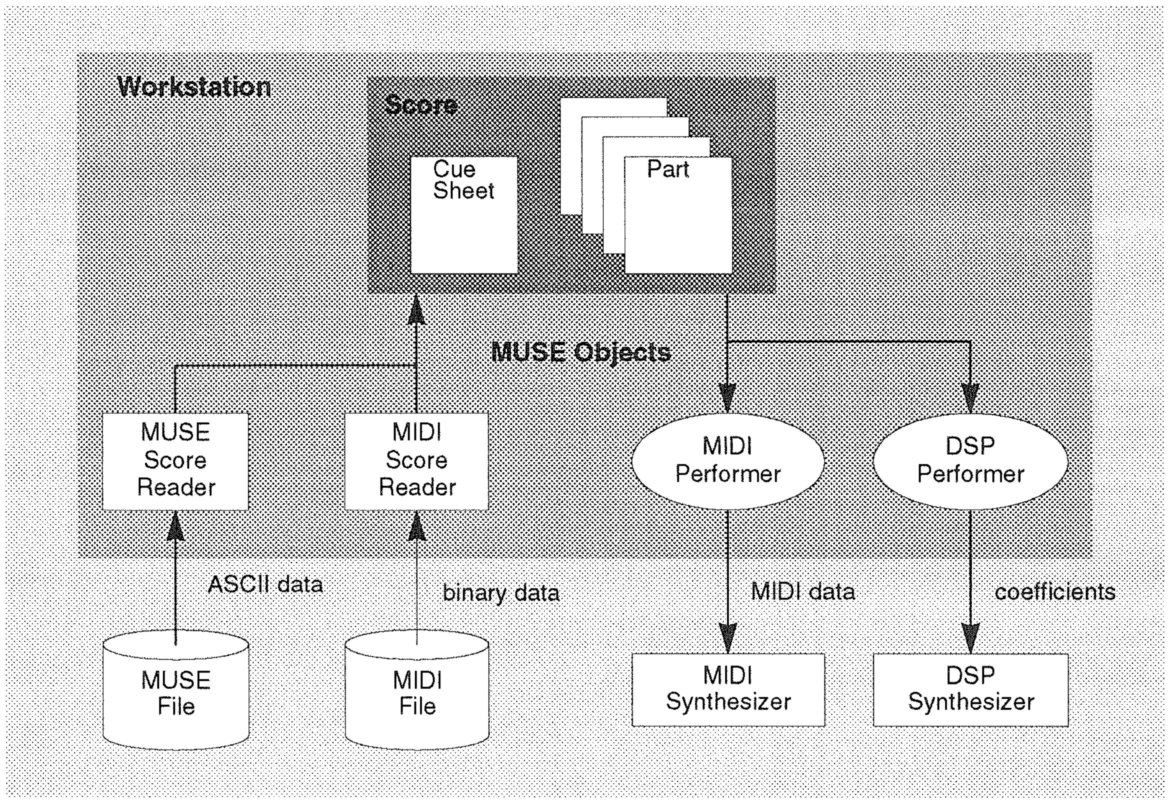


FIGURE 4.3 ZED score file and device independence.
The diagram shows score file independence and device independence. Score files are converted to MUSE objects by score readers and performer objects then convert the muse objects to packets for their particular instrument.

The Conductor

The definition of the conductor class is shown below. The instance variable score for the conductor is the entire score and the instance variable performers is a collection of all performers, one for each part in the score. The instance variable inputControllers contains input controller objects for real-time control, and is *nil* if the performance is not under real-time control. The instance variable scheduler contains an instance of the class **Scheduler**. The scheduler is used to manage events and coordinate the performance. The conductor also has an instance variable patches that contains a collection of patch objects that define how the real-time inputs affect the performance. The instance variables interpretationContext and currentEvents are used during the performance to cache the global interpretation context, and the events that are currently being played, respectively.

```
Object ( )
    ZEDPerformanceObject (name)
        Conductor (score performers inputControllers scheduler patches
                  interpretationContext currentEvents)
```

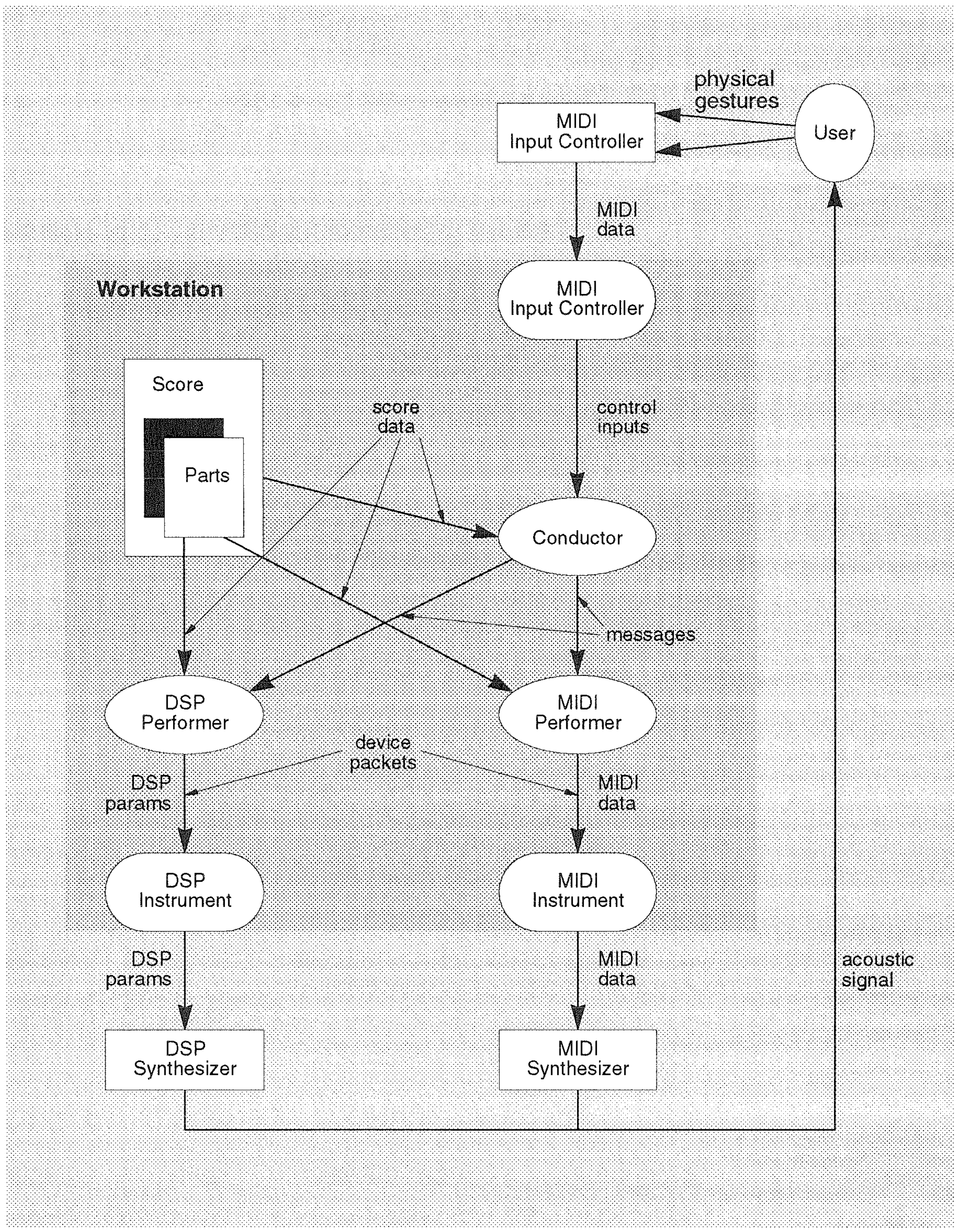


FIGURE 4.4 ZED objects and data flow.
The performer objects take the high-level score data from their parts and messages communicated from the conductor, and compute low level synthesis parameters for their particular sound synthesis device. Each instrument object encapsulates an output device driver for the particular physical interface with the workstation. Similarly, the input controller objects encapsulate an input device driver for the physical interface for the input controller device. The MIDI controller in the diagram represents any number of MIDI controllers attached to the same MIDI serial port.

Input Controllers

Input controller objects provide an interface between the input controller device and the conductor object. The class **InputController** is an abstract class defining the basic semantics for all controllers. One concrete subclass is defined for each *type* of controller interface. Each concrete input controller class encapsulates a software input device driver that receives data from the controller device via the controller's interface. For each subclass of **InputController** a corresponding *packet* class is defined as a subclass of the abstract class **ZEDEvent**. The packet class defines the object that is created by the input controller object from the input data and is passed to the conductor. The class hierarchy for a system that has only MIDI input controllers is shown below.

```
Object ( )
    ZEDPerformanceObject (name)
        Conductor (score performers inputControllers scheduler patches
                    interpretationContext currentEvents)
        InputController (conductor)
            MIDIController (port)
    ZEDEvent (time)
        MIDIPacket (statusByte data1 data2)
```

The class **MIDIPacket** defines objects that hold the data received from a MIDI interface. These MIDI packet objects are sent to the conductor from MIDI controller objects. (The same packet class is used by MIDI performers to send MIDI data to a MIDI instrument.) There is one instance of a concrete input controller class for each physical interface. For example, if MIDI input can be received through two separate serial ports on the workstation, there is one instance for each port. Additional classes can be defined for other types of serial input controllers, or for controllers that have bus interfaces. The data abstraction capabilities of object-oriented languages allow the details of the particular devices to be hidden from the conductor object, allowing the conductor object to operate on all control data in the same way.

Input controllers are referenced in patches by their name. Input controllers also have an instance variable containing the conductor object. When a control input is received, a message is sent to the conductor with the data. In the case of MIDI controllers, the controller has a port that identifies which physical serial port on the computer workstation will receive the data.

A variety of different control devices can be used to control the performance. Control devices are divided into two basic categories: triggers and continuous controllers. Triggers are devices that send "down" and "up" events and include devices such as pedal and button switches, or MIDI drums. Continuous control devices send continuous data values and include volume pedals, modulation wheels, and sliders. An interesting property of the Stanford Radio Baton is that it can provide both continuous control and trigger control. Trigger devices are often used for initiating or terminating a note, or a repeated sequence of notes. Continuous devices are most often used to dynamically update state variables in the performance. For simplicity it is assumed that all controllers that generate voltages, such as the Stanford Radio Baton, have their voltages converted to MIDI so that the data enters the system through a standard MIDI interface.

Scheduling

The scheduler helps the conductor coordinate the performance by maintaining a list of events ordered by the time that they are to be executed. ZED's scheduler is a hybrid scheduler that separately manages events whose time is statically bound, and those whose time is bound during the performance. ZED's scheduler also supports multiple time references, allowing different parts of the score to be under independent real-time tempo control. The complete hierarchy of classes that define ZED's scheduler are shown below.

```
Object ( )
    QueueEvent (receiver selector parameter next)
    QueueNode (time eventList next eventListTail)
    Queue (nodeList currentNode tempo nextQueue)
        QueueWithOffset (offset)
            RepeatedQueue (numberOfRepeats counter queueLength)
    ZEDScheduler (queueList currentTime)
```

The class **ZEDScheduler** has an instance variable queueList which is a linked list of queues. Each queue has a list of time ordered nodes that each have a list of all events that are to be played at the node's time. Each queue may have its own time reference. The scheduler merges the queues at runtime, selecting the node with the earliest time by comparing the next node of each of the queues. Each queue event in the node's eventList has a receiver object, a selector for the message, a parameter to the message, and a pointer to the next event that occurs at the same time. The instance variable tempo on the class **Queue** allows each queue to have a different time reference. The class **QueueWithOffset** is used to instantiate sequences at various points in the performance. The class **RepeatedQueue** is used to optimize repeated sequences.

A detailed discussion of real-time scheduling and ZED's scheduler design can be found in *Appendix A, Real-Time Scheduling*, on page 71.

Patching

The conductor's instance variable patches contains a collection of patch objects. Patch objects specify which control inputs are to be recognized by the conductor and what action is to take place when a control input is received. The class definition for patches is shown below.

```
Object ( )
    ZEDPerformanceObject (name)
        Patch (inputController filter receiver selector valueSelector)
```

Patch objects inherit the instance variable name from the superclass **ZEDPerformanceObject**. Each patch consists of: an inputController, containing the object that receives the input from the controller device; a filter that selects inputs based on some criterion; a receiver, defining the performance object that is to act on the input; a selector, specifying the message that is sent to the object when an input is selected by the filter; and a valueSelector, specifying a message that is sent to the input packet object that extracts a value that is the parameter to the message specified by the selector. Each time a real-time input is received, it is passed to all patches, effectively applying an "or" function across all the patches. *FIGURE 4.5* shows an overview of the patching mechanism.

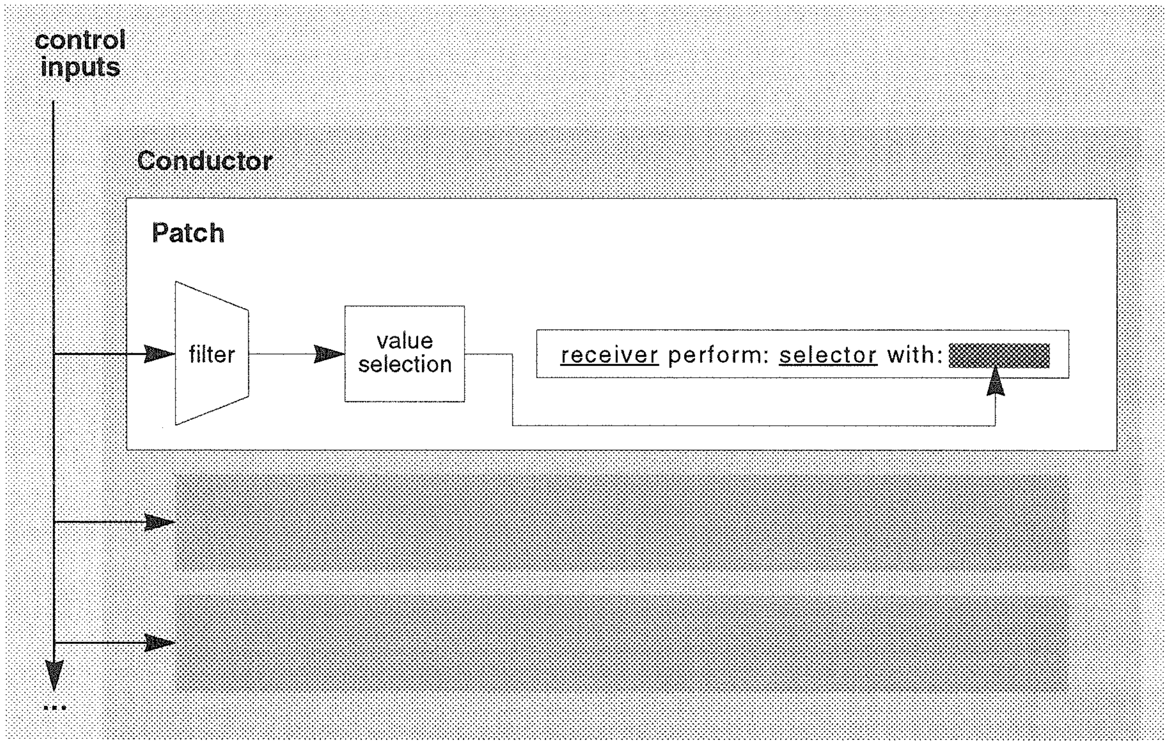


FIGURE 4.5 ZED patching overview.
Each input is sent to all patches. Each patch applies the associated filter and if the filter selects the input, a value is extracted from the input. The message is then sent to the receiver with the extracted value as the parameter.

Methods for selecting data are implemented on concrete subclasses of the abstract class **Filter**, and are specified by the data filter selector. Methods for data extraction are implemented by the input packet classes (i.e., **MIDIPacket**). These methods constitute a library of reusable methods that can be extended with new methods to process the control inputs in more sophisticated ways.

Filter Objects

In addition to an input packet class, a concrete filter class is defined for each type of input controller. The complete class hierarchy for handling MIDI input controllers is shown below.

```
Object ( )
    ZEDPerformanceObject (name)
        InputController (conductor)
            MIDIController (port)
        Patch (inputController filter receiver selector valueSelector)
        Filter (dataFilterSelector parameters)
            MIDIFilter (statusCode channels)
    ZEDEvent (time)
        MIDIPacket (statusByte data1 data2)
```

Filters inherit the instance variable name from the class **ZEDPerformanceObject**. Patches reference filters by their name. The filter's dataFilterSelector and parameters instance variables specify a message (and its parameters) that is sent to the input packet object to filter events based on the packet's data.

All input packets are instances of a subclass of the class **ZEDEvent**. **ZEDEvent** provides an instance variable for the time that the event is received. Control inputs received from a MIDI controller are instances of the class **MIDIPacket**. All MIDI inputs have a status byte defining the type of event that the filter selects. There are two basic types of MIDI inputs: channel events and system events (shown in *TABLE D.1*, and *TABLE D.2* in *Appendix D*). The status byte of MIDI channel events has two parts, a *code* and a MIDI *channel* number. MIDI channel events also have one or two data bytes (depending on the status code). MIDI system events have a status byte that is a code and has no channel. MIDI system events may have zero, one, or two data bytes. The same MIDI packet class is sufficient for representing either type of MIDI input.

The statusCode instance variable for MIDI filters selects all input packets with the specified status code. The instance variable channels specifies which MIDI channels events are selected. The instance variable dataFilterSelector specifies a message that is sent to the input packet object that will select or reject the packet (by returning true or false) based on the data values and using the filter's parameters. These methods may be arbitrarily complex, but they are generally quite simple and select packets with specific status codes, channels, and explicit values or ranges of values for data1 and data2.

Performers

The class **Performer** is an abstract class used to define performer objects. For each type of synthesis device, a subclass of the class **Performer** is required to translate the abstract score representation into specific synthesis parameters for the synthesis device being played. The device specific synthesis parameters computed by the performer object are held by a packet object. There is one packet class for each type of synthesis device interface and this packet class can be shared with input controllers that use the same interface. For example, MIDI synthesizers and MIDI controllers use the same packet class **MIDIPacket**. The packet class defines the interface between the performer and the instrument and is somewhat analogous to the instrument specific physical gestures that a live musician applies to an acoustic instrument.

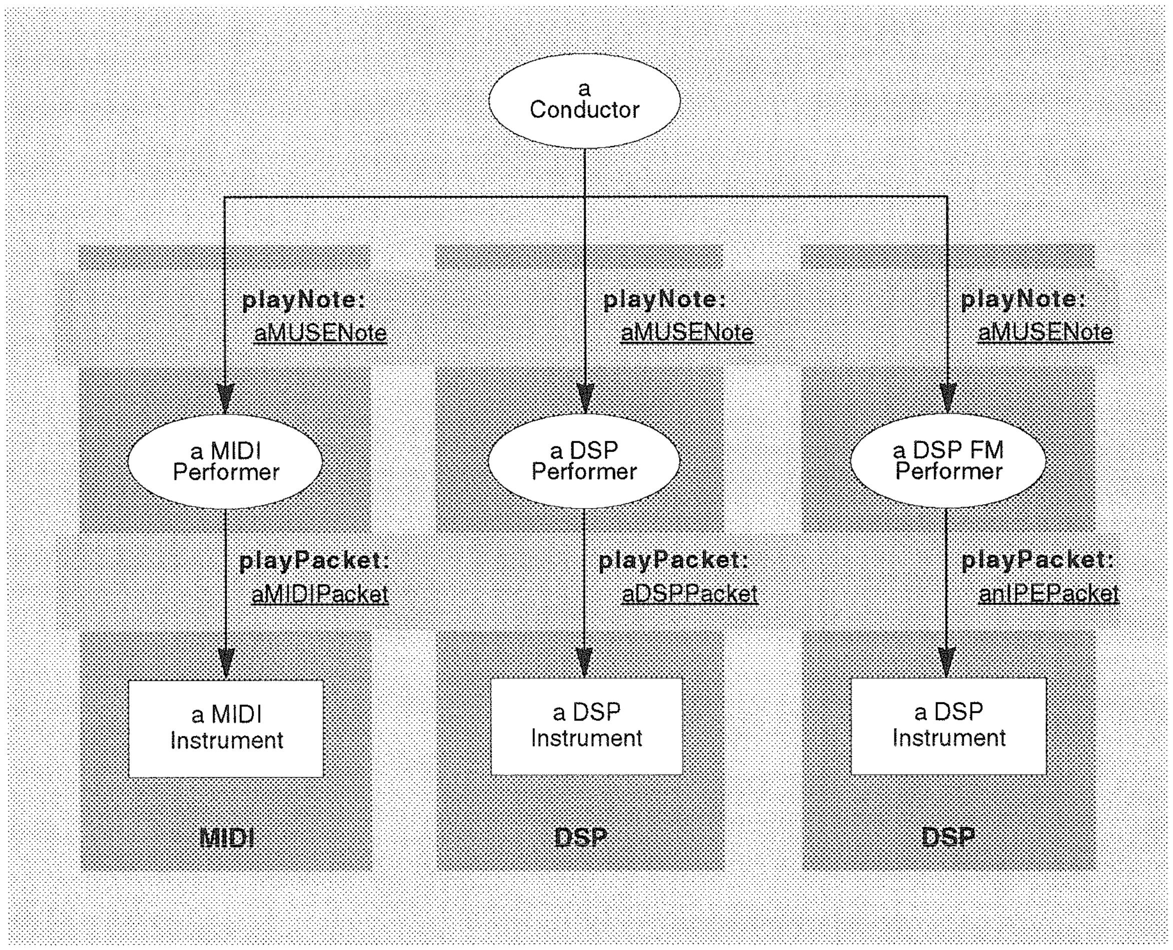


FIGURE 4.6 Performer and instrument interfaces.
Abstract MUSE notes are interpreted by each performer in their playNote: method. This method computes instrument packets for the particular instrument. These packets are then sent to the device via the playPacket: method.

The class hierarchy shown below defines the basic classes for MIDI and DSP performers and the corresponding packet classes.

```
Object ( )
    ZEDPerformanceObject (name)
        Performer (conductor part instrument
                    interpretationContext scheduleMessage)
            MIDIPerformer ( )
            DSPPerformer ( )
    ZEDEvent (time)
        MIDIPacket (statusCode data1 data2)
        DSPPacket (parameterValues)
```

There is one performer instance for each part in the score, and one instrument instance for each performer. The instance variable part for each performer is the performer's part from the score. The instance

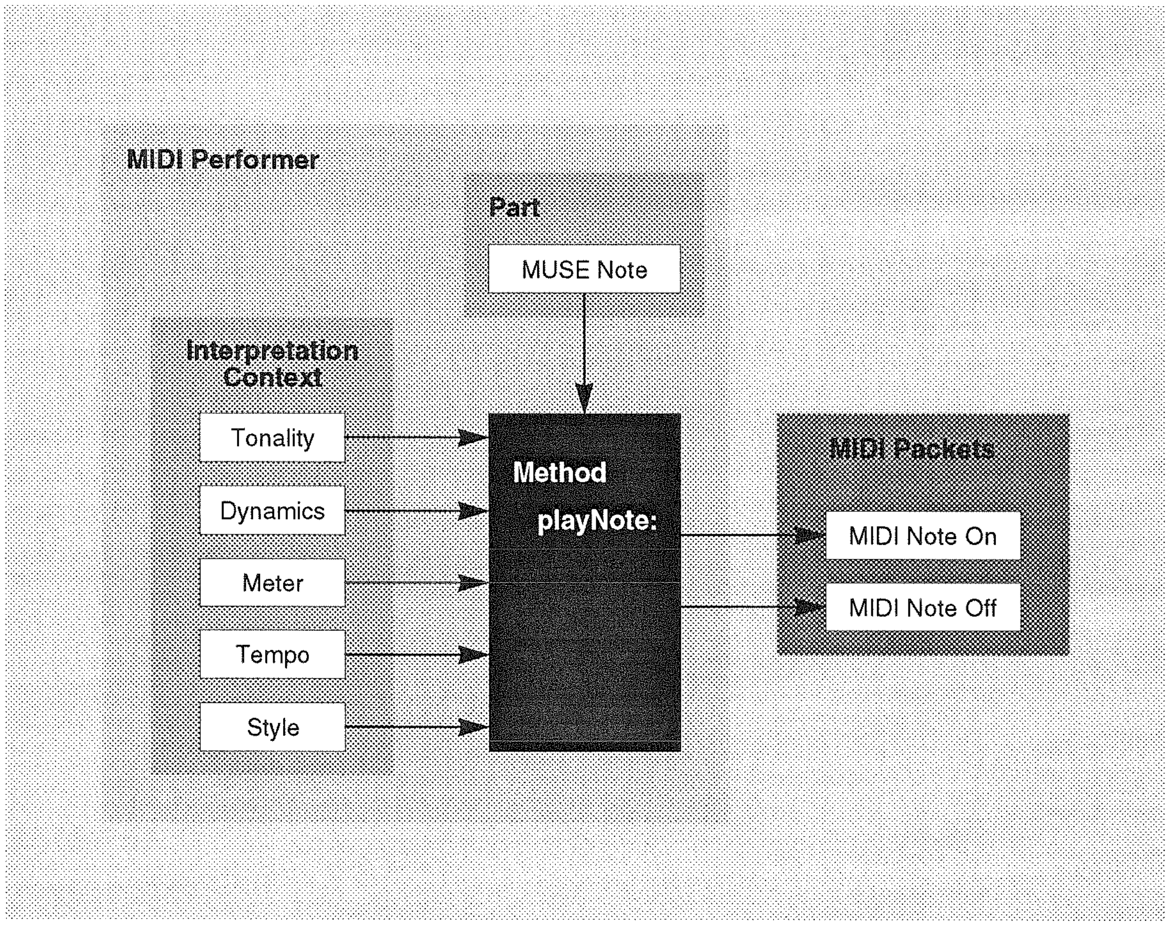


FIGURE 4.7 MIDIPerformer playNote: method.
The MIDI performer's method for playing a note is shown. The note is the parameter to the method and the method uses the performer's interpretation context. The interpretation context may be the global interpretation context defined by the conductor, or may be defined locally by the performer object.

variable instrument is an object that encapsulates the device that the performer object is controlling. (The instance variable *interpretationContext* is used as a cache during the performance and the instance variable *scheduleMessage* is used for scheduling events.) The instance variable conductor is the conductor object that has the performer object in its performers collection.

The primary function of the performer object is to compute device specific packets from the score data for each note in the score in the context of the interpretation symbols. This computation is done in a method called playNote: that is implemented by each performer class. The playNote: method computes one or more packets and sends them to the instrument object. Thus, the interface between the conductor and the performers is homogeneous regardless of the type of synthesis instrument being played as shown in *FIGURE 4.6*. *FIGURE 4.7* shows the inputs and outputs of this method for MIDI performers.

Instruments

An instrument object represents a particular *voice* or timbre that is implemented on the sound synthesis device. The class **Instrument** provides an abstraction of the physical synthesis device and, like the class **InputController**, requires a subclass for each type of synthesis device interface. The class hierarchy shown below defines the basic classes for MIDI and DSP synthesis devices.

```
Object ( )
    ZEDPerformanceObject (name)
        Instrument ( )
            MIDIInstrument (port channel)
            DSPInstrument (parameterAddresses)
    ZEDEvent (time)
        MIDIPacket (statusCode data1 data2)
        DSPPacket (parameterValues)
```

The concrete instrument classes **MIDIInstrument** and **DSPInstrument** encapsulate a software output driver. The packet classes **MIDIPacket** and **DSPPacket** define the type of object that is passed from the performer to the instrument. MIDI instruments have an instance variable port that refers to which serial port on the workstation is to be used. The instance variable channel refers to the channel of the instrument's timbre on the MIDI synthesizer. Each DSP instrument object has a set of parameterAddresses that are one-to-one with the DSP packet object's parameterValues. Each instrument class implements the method playPacket:. This method is sent by the performer object to generate sound. The parameter to the method is a packet object for the particular device. The playPacket: method moves the data to the synthesis device's hardware interface (a serial port or a DSP card on the workstation's bus) causing sound to be generated.

Performer Specialization

Each synthesis device implements a number of different timbres (or voices) that may have different semantics as well as different control parameters. Some instrument voices are percussive (not sustained). Other voices like those for wind instruments are sustained and may have a variety of capabilities that percussive timbres do not have. Most notably, sustained instruments need to be explicitly turned off. Some voices may control vibrato and may change timbre parameters to create different types of note attacks. DSPs are the most general synthesizers and can be used to implement a wide range of synthesis algorithms. The synthesis parameters and continuous control capabilities may vary greatly across these algorithms.

Specialized performer classes may be defined as subclasses of the basic synthesis device performer class to take advantage of timbre parameters of the instrument voice. Each performer subclass implements a playNote: method by computing data packets for the particular instrument voice. One performer class can be implemented for families of instrument voices that have similar control capabilities as shown below.

```
Object ( )
    ZEDPerformanceObject (name)
        Performer (conductor part instrument
                    interpretationContext scheduleMessage)
            MIDIPerformer ( )
                MIDIPercussionPerformer ( )
                MIDIWindPerformer ( )
                MIDIDX7Performer ( )
                    MIDIDX7MyPatchPerformer ( )
            DSPPerformer ( )
                DSPWaveGuidePerformer ( )
                DSPAdditivePerformer ( )
                DSPFMPerformer ( )
```

In the above class hierarchy, there are three subclasses of the generic class **MIDIPerformer**. The class **MIDIPercussionPerformer** in the example is optimized to only send "note on" packets whereas a **MIDIWindPerformer** sends "note off" packets and modulation (vibrato) packets. The class **MIDIDX7Performer** is used for a specific MIDI synthesizer, namely the Yamaha DX-7. Instances of this performer class access the specific timbre parameters provided by the DX-7. The performer class **MIDIDX7MyPatchPerformer** controls the timbre parameters for a specific DX-7 patch.

The methods for the specific DSP performer classes are related to the particular DSP instrument library that defines the synthesis algorithms being used. In general, a performer class is defined for each of the different synthesis techniques provided by the DSP instrument library. It is likely that these classes may be refined and enhanced over time as new synthesis techniques are developed. In the example, the playNote: method for the class **DSPFMPerformer** implements plays accented notes by increasing the *brightness*.

Encapsulating DSP Instrument Libraries

This thesis does not address the problem of real-time music synthesis. Therefore, ZED relies on existing technologies such as MIDI and DSP synthesis to create the performance. The classes for DSP synthesis presented in the previous sections provide a framework for developing DSP synthesis algorithms and encapsulating them with instrument objects. An attractive alternative to developing synthesis algorithms is to encapsulate existing DSP libraries such as those provided on the NeXT Computer [Smith et al., 1989]. To accomplish this, a class called **NeXTPerformer** is defined as shown below.

```
Object ( )
    ZEDPerformanceObject (name)
        Performer (conductor part instrument
                    interpretationContext scheduleMessage)
            NeXTPerformer ( )
```

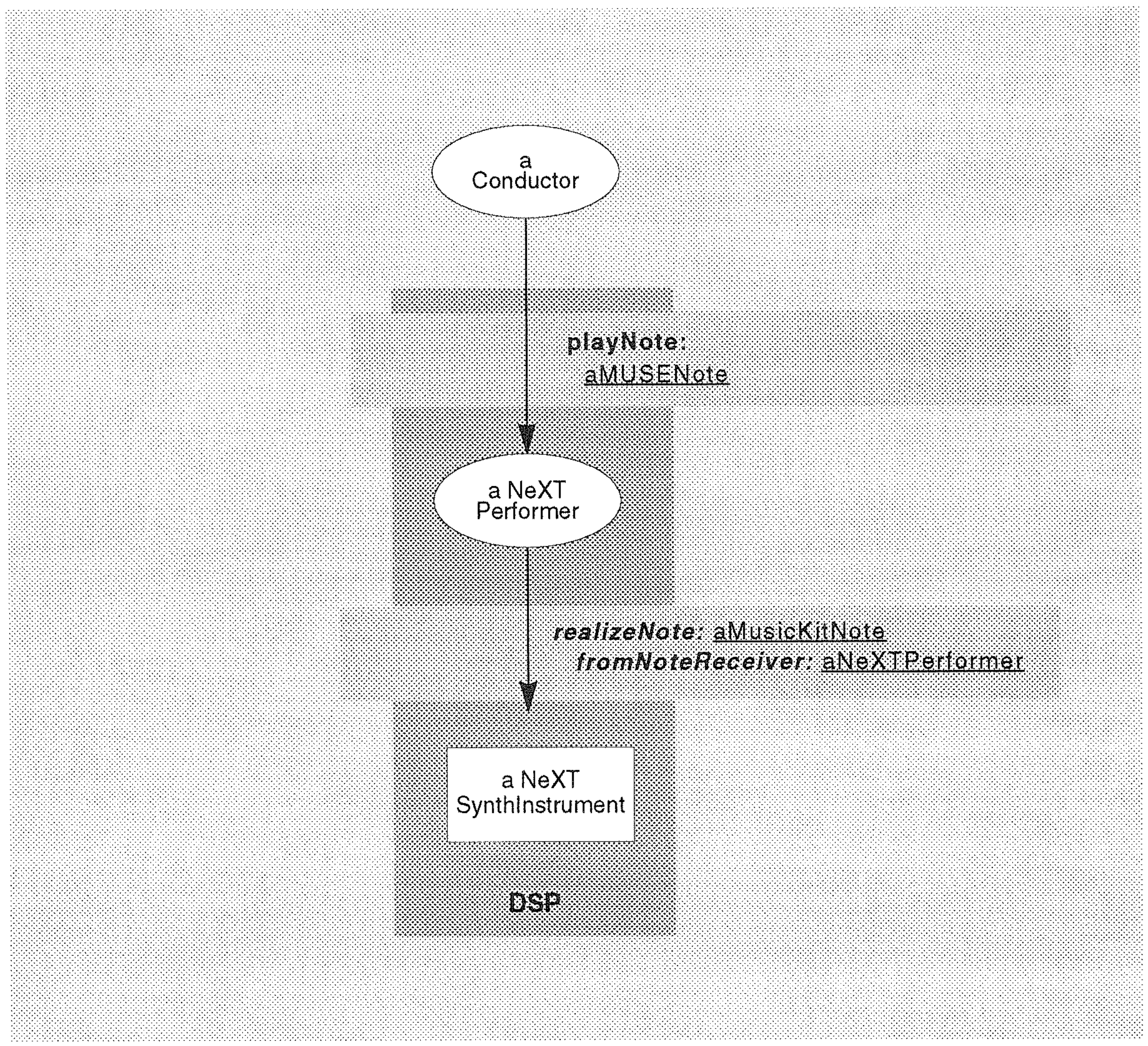


FIGURE 4.8 Encapsulating the NeXT Music Kit instrument library.
The messages for playing a note on a NeXT "SynthInstrument" are shown. The NeXTPerformer object converts the MUSE note to a NeXT Music Kit "Note" and sends the NeXT Music Kit message "realizeNote:from:" to a NeXT "SynthInstrument."

Each NeXT performer object has an instrument that is an instance of the Music Kit class "SynthInstrument." The Music Kit class "Note" defines the input to the "SynthInstrument" object. In the Music Kit, a note is played by the method "realizeNote:fromNoteReceiver:" implemented on the "SynthInstrument" class. Therefore, the NeXT performer implements the method playNote: to convert a MUSE note object to an instance of the Music Kit "Note" class, which is passed as the parameter to the "realizeNote:fromNoteReceiver:" method. *FIGURE 4.8* shows how a note is played by an instances of the class NeXTPerformer. The NeXT Music Kit takes care of the rest! The class NeXTPerformer could be further subclassed to provide additional specialization for particular synthesis algorithms implemented by patches in the Music Kit.

Incorporating Other Synthesis Technologies

Other synthesis technologies can be incorporated into ZED as well, such as the IPE synthesis hardware [Wawrzynek et al., 1984; Wawrzynek, 1987]. The IPE synthesis hardware has been used to implement physical models of musical instruments. A desirable property of physical models is that the parameterization of the instrument maps closely to intuitive parameters that correspond to physical gestures of live musicians. For example, an IPE struck instrument (a percussion instrument) has parameters that describe the "type of mallet," "how hard to strike," and "where to strike." An IPE wind instrument has parameters such as "how hard to blow" and "how breathy is the sound." High level, intuitive parameters make the synthesis models easier to use and understand by composers, and also reduce the control bandwidth required to control the synthesis. The class hierarchy below is an example of how the IPE hardware might be incorporated into ZED. The hardware could be connected to the bus of the workstation and the **IPEInstrument** would have a software driver that would memory map the coefficientValues into the corresponding coefficientAddresses. The IPE performer classes would have methods that compute the model coefficients from the abstract score data.

```
Object ( )
    ZEDPerformanceObject (name)
        Performer (conductor part instrument
                    interpretationContext scheduleMessage)
            IPEPerformer ( )
                IPEPercussionPerformer ( )
                IPEWindPerformer ( )
        Instrument ( )
            IPEInstrument (coefficientAddresses)
    ZEDEvent (time)
        IPEPacket (coefficientValues)
```

Summary

The ZED design defines objects that define methods for some of the simple behaviors of each of the components in the live orchestra. A digital music representation is defined for the score and parts that has the properties of the common music notation used by composers for printed music. This music representation provides a basis for real-time control because the note symbols and their interpretation are separated. Performer classes are defined that map the abstract MUSE notes to packets that are specific to the corresponding instrument class. ZED's design relies on the object-oriented features of data abstraction, inheritance, and encapsulation. The design's extensibility facilitates the definition of specialized performer classes that take advantage of specific sound synthesis algorithms. The definition of performer and instrument classes to incorporate new sound synthesis technologies is also facilitated. More sophisticated performer interpretations of MUSE notes can be implemented by subclassing a performer class and overriding the playNote: method.

Instrument classes hide the low level details of the synthesis hardware from the performers (just as the details of the physics of an acoustic instrument are hidden from the live performer). New synthesis devices can be incorporated into ZED by defining a new instrument class with a playPacket: method for transmitting the data, and a corresponding performer class with the method playNote:. Existing instrument libraries can also be used by ZED by defining a performer object that provides an interface to the library.

References

Smith, J. O., Jaffe, D. A., and Boynton, L. *Sound and Music on the NeXT Computer.* Preliminary Draft, Menlo Park, CA: NeXT Inc., 1989.

Wawrzynek, J. C. VLSI Concurrent Computation for Music Synthesis. Ph.D. Thesis, *Caltech Computer Science Technical Report,* Caltech-CS-5247:TR:87, California Institute of Technology, Pasadena, CA, 1987.

Wawrzynek, J. C., Lin, T. M., Mead, C. A., Liu, H., & Dyer, L. M. A VLSI Approach to Sound Synthesis. In Buxton, W. (ed), *International Computer Music Conference at IRCAM, Paris, France.* San Francisco, CA: Computer Music Association, 1984, pp. 53-64.

Chapter 5

MUSE: A Digital Music Representation

The orchestra performance model requires a digital music representation for representing score data. Because there was no existing digital music representation that provided the semantic power of the common music notation used by the live orchestra, a new representation was invented to be used with ZED. We call our digital music representation *MUSE*. The score file format for representing MUSE objects in ASCII files can be found in *§MUSE Score Files* in *Appendix B* on page 83.

The first section in this chapter discusses the common music notation used by composers for notating scores for live musicians. The key features are identified. Then, the MUSE score representation is defined as a set of classes defining objects that represent the symbols in common music notation. The extensibility of MUSE is highlighted to show how it can be extended beyond common music notation symbols, providing the ability to define new symbols that can be used to gain precise control over the expressiveness of the performance.

Common Music Notation

A music *notation* is a system of written symbols, a language if you will, by which musical ideas are represented and preserved for study and performance [Read, 1979; Rastall, 1982]. *Common music notation* is a notation that has evolved over the last few centuries for notating Western music [Byrd, 1984]. Thus the notation acts as a set of instructions to performers who create the sound of the music. A *digital music representation* could be thought of as a digital encoding of the symbols of music notation by which musical ideas are represented and preserved to be read and performed by *computers*.

Lukas Foss [Foss, 1976] commented on the balance between music notation and performance expressiveness in the following way: “. . . Performance also requires the ability to ‘interpret’ while at the same time allowing the music to 'speak for itself.'" This statement applies to music performance by computers as well as by humans. That is, the dynamic interpretation of a composition is an important component of performance. The underlying representations of music must allow the performer flexibility during the live performance while still conveying the composer's intent. To this end, common music notation (CMN) is a *symbolic* representation in which a graphical symbol represents a musical *concept* rather than instructions on how to play the instrument (as in tablature notations of the sixteenth century [Grout, 1973]). CMN support a separation between the *representation* of notes and the *interpretation* of them. Composers communicate the abstract ideas of the properties of the sound, and leave it up to the live performers and conductor to carry out their ideas. CMN is basically instrument independent. A composer can, however, also notate instrument specific information, such as bow markings for string players or mallet choices for percussion players. If such a part were to be played on another instrument these symbols would be ignored.

CMN has proven to be a powerful notation, allowing a variety of interpretations to be applied to the same composition, thus making each performance dynamic and unique. CMN is also flexible enough to allow composers to extend the vocabulary of symbols to express twentieth century musical scores such as Boulez's use of time-varying functions for tempo [Stone, 1975] and others [Read, 1978; Smith, 1975].

MUSE Overview

There are four basic properties of CMN that are the basis of the MUSE representation: high-level semantics; separation between the representation and the interpretation; instrument independence; and extensibility allowing the symbol vocabulary to be expanded to include instrument specific symbols, as well as symbols for non-Western and twentieth century musical concepts. MUSE is based on our earlier work in music representations [Dyer, 1986; Dyer, 1987]. MUSE's semantics are designed to be sufficiently rich so as to support the mapping of a variety of types of score files with different semantics to a single generic MUSE score that can then be played on any ZED instrument.

A number of digital music representations have been defined for use in particular computer music applications. Many of the representations for music synthesis applications are based on *note lists* after those of Music V [Mathews, 1969a]. This representation has a list of "notes," each with the set of synthesis parameters required to realize the note. Notes contain explicit frequencies, start time and end time, and timbre and envelope parameters. This type of note list representation has the same flavor as the tablature representations of the Renaissance in that they describe *how to play the instrument* rather than abstract musical ideas.

Music V's note representation is a practical and efficient way to represent non-real-time sound synthesis, but is inadequate for real-time performance. This is because the synthesis parameters are statically bound before the performance, thus preventing dynamics interpretation of the notes. Such note lists are therefore not well suited for interactive control which requires that the sound parameters be computed during the performance right before they are played.

MUSE Components

The primary symbols in a score are *notes* and *rests*, defining the initiation of sound and silence, respectively. Each note may have more detailed information for the attack and articulation, defining the complex envelope of the note. Articulation symbols may also include instrument specific symbols such as bowing marks for string instruments and pedal indications for piano.

Notes are interpreted by the performer in terms of an interpretation context that consists of five interpretation symbols: tonality, tempo, meter, dynamics, and musical style. The note's frequency, time, complex envelope, and loudness are not known without this interpretation information. Some interpretation symbols in a score are global in the sense that they apply to all performers, and others are local in the sense that they apply to only one performer or a small number of performers.

Numbers

All numeric values in a MUSE score, such as the time and duration of each note, are represented as instances of a subclass of the abstract class **Number**. The class hierarchy for numbers is shown below.

```
Object ( )
    Number ( )
        SmallInteger ( )
        Float ( )
        Fraction (numerator denominator)
```

A variety of different representations for numbers are supported because different score file formats use different types of numbers. **SmallInteger** is typically used for representing time in milliseconds or some other fractional part of a second. Some computer music systems represent numbers with **Float**, but this is often problematic due to round off error. Score file formats that are used in printing applications generally use some form of **Fraction** to allow the precise representation of rhythmic values such as triplets and more complex rhythms like 11:13, 15:17.

MUSE Symbols

Each MUSE symbol is defined as a concrete subclass of the abstract class **MUSESymbol**, shown below.

```
Object ( )
    MUSESymbol (time)
```

The class **MUSESymbol** defines the instance variable time that specifies when the symbol is to take place measured from the beginning of the composition. The time is in units of *beats* rather than physical time, and can be any number as described by the number class hierarchy. The distinction between abstract time and physical time is important because it allows the tempo to be under interactive control, thus changing the mapping of beats to seconds.

Interpretation Symbols

The interpretation context holds a set of the interpretation symbols that define how notes are interpreted: the tonality consisting of a key scale, a key note, and a tonal system; the meter defines the metrical pulse; the tempo defines how beats are mapped to physical time; the dynamics define how loud the notes are played; and the style defines the "feel" of the composition.

The conductor object has an interpretation context that is global to all performer objects, and changes to any of the interpretation symbols are communicated simultaneously to all performers. In addition, each performer object may have their own interpretation context that may be independent of the global context or may share some state with the global context. The primary way of controlling a ZED performance is by updating the state of the interpretation symbols based on real-time input, thus causing the performers to interpret their notes differently.

The class **InterpretationContext** is defined below.

```
Object ( )
    MUSEObject ( )
        InterpretationContext (tempo dynamics meter tonality style)
```

Interpretation symbols are *sticky*—when an interpretation symbol occurs in a score, its state variables stay in effect until the next interpretation symbol of the same type occurs. Exactly one interpretation symbol for each of the five types is in effect for each note in a performer's part.

The class hierarchy for interpretation symbols is shown below. Each interpretation symbol object may have a name. Interpretation symbol objects can be referenced in configuration files and patch files by their name. The subclasses of **InterpretationSymbol** are discussed in the sections that follow.

```
Object ( )
    MUSESymbol (time)
        InterpretationSymbol (name)
            Tempo (metronomeMarking)
            Dynamics (level)
            Tonality (keyNote tonalSystem)
            Meter (beatsPerMeasure referenceBeat stressSelector)
            Style (articulation)
```

Tempo

The tempo object controls the overall pace of the performance. The instance variable metronomeMarking stores the instantaneous or current metronome marking as the number of beats per minute. The tempo class implements the method secondsFor:. This method maps a number of beats to physical time, as shown below.

```
secondsFor: beats
    "Map the beats to seconds."
    ^beats * (60.0 / metronomeMarking)
```

Dynamics

The class **Dynamics** controls the overall volume of the performance. The instance variable level stores the current dynamics level. The dynamics level is expressed as the percentage of the maximum. This unitless value enables device independence, allowing each performer object to compute their own dynamics relative to the maximum level for their instrument.

Meter

The *meter* is the grouping of pulses or units within a single measure, or a frame of two or more measures [Creston, 1961]. The GRIN computer music system [Mathews, 1976] used a periodic amplitude function to represent primary and secondary accents for a particular meter. MUSE defines a class and a set of methods that represents the simple and compound meters of Western music (such as 4/4 and 6/8 respectively), and also arbitrary periodic functions such as those used by GRIN.

The instance variables beatsPerMeasure and referenceBeat are used to denote the time signature of the composition. The instance variable stressSelector is a selector specifying a method on the class **Meter**. The method has one parameter, a beat number, and computes the instantaneous stress for the parameter. The stress is also unitless and is expressed as a percentage of the value of no stress. Thus, if there is no metric pulse, the method that implements the stress selector returns the constant 1.0. These methods are used in the same way that GRIN uses periodic functions for the amplitude. An example method for a periodic meter for standard 4/4 time is shown below.

```
fourFourStressForBeat: beat
    "Return
        a primary accent on the first beat of every measure (beats 0,4,8 . . .);
        a secondary accent on the third beat of every measure (beats 2,6,10 . . .);
        no accent otherwise."
    (beat \\ 4) == 0 ifTrue: [^1.5]
    (beat \\ 4) == 2 ifTrue: [^1.2]
    ^1.0
```

Style

The *style* is generally notated in CMN with text, often Italian, such as ***allegro con moto***, ***marcato***, ***minuet***, ***swing***, ***waltz***, and ***adagio***. In the absence of any specific articulation symbol on a note, the style object provides the articulation and attack. The style may also set the meter and tempo. For example, if the composition is in a ***marcato*** style, the default articulation may be defined to reflect a slight accent on the beat and a slight separation between notes, and a tempo of 120 beats per minute.

The way that live performers affect their performance to reflect these symbols is largely a result of musical training and practices that have been handed down from teacher to student over hundreds of years. A complete exploration and formalization of musical style and a thorough investigation of possible computational models is note addressed this thesis. Some systems have developed sophisticated simulations of musical style [Frydén and Sundberg, 1984] and such algorithms could be incorporated into ZED through performer subclassing. The basic MUSE style class could also be subclassed to hold more precise information. For the purposes of this thesis, the style is defined simply as the default articulation for each note and further expression is provided by the user who is controlling the performance. The instance variable articulation contains an instance of the class **Articulation**, as described in *§Articulation* on page 48.

Tonality

The *tonality* defines how the notes' pitches are interpreted. The MUSE representation of the tonality is based on a group theoretic representation of Western tonality [Balzano, 1982]. The approach is generalized to represent a wide variety of pitch representations including non-twelve-tone scales, microtonal scales, and MIDI key numbers. The basis of the representation is a *group* that defines the set of all possible pitches in a composition. For the Western twelve-tone music, the group $C_{12} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ corresponds to the chromatic scale beginning on the note C. The note middle C is called the *origin* of the tonal system. Each of the half steps are numbered as shown in *FIGURE 5.1*. Pitch sets are also defined to

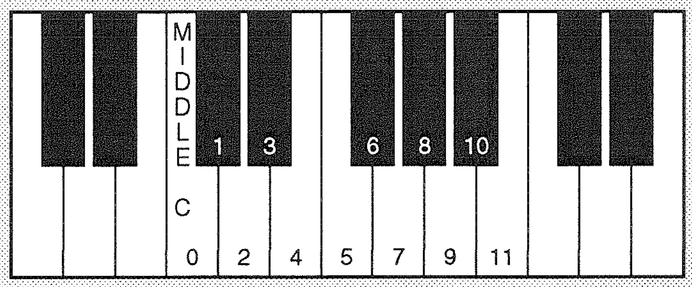


FIGURE 5.1 The group C_{12} and natural scale C_7.

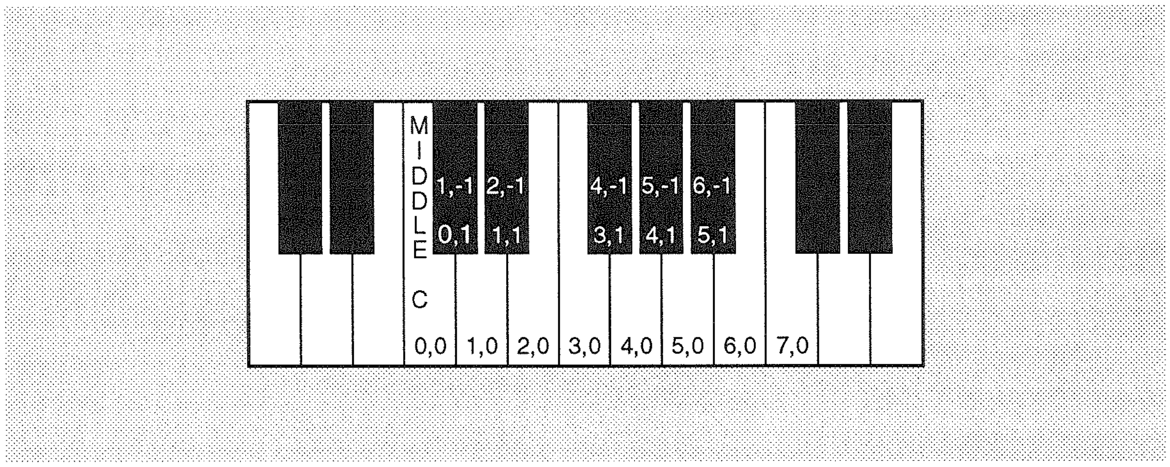


FIGURE 5.2 Pitches in a C Major scale.
The pitches in a C Major scale are shown. Common enharmonics are shown as two representations for the same piano key.

represent scales. For example, the pitch set {3, 5, 7, 8, 10, 0, 2} represents an E♭ major scale because the half step 3 corresponds to E♭, 5 to F, 7 to G, 8 to A♭, etc. A special pitch set, called the *natural scale*, is the set C_7 = {0, 2, 4, 5, 7, 9, 11}. This natural scale describes the pitches in the C_{12} group that are printed in common music notation without *accidentals* and represent the white keys on the piano.

The first note in a pitch set is called the *key note*. The pitch set can be normalized by subtracting the first note, the key note, from each of the pitches using modulo arithmetic. Therefore, the E♭ major scale can be represented as the key note 3 and the pitch set {0, 2, 4, 5, 7, 9, 11}, and all major scales can be represented with this pitch set and different key notes. The MUSE pitch representation expresses the key note as a pitch relative to the natural scale and the pitches in the score relative to a key scale. It also allows enharmonic pitches—two different spellings of the same pitch such as E♭ and D♯—to be distinguished. This is done by specifying an index into the natural scale and an offset in semitones. *FIGURE 5.2* shows how each note in a chromatic scale is represented when used as a key note.

The MUSE classes representing the tonality are shown below.

```
Object ( )
    MUSESymbol (time)
        InterpretationSymbol (name)
            Key (keyNote tonalSystem)
    MUSEObject ( )
        Pitch (step offset)
        Scale (pitchSet)
        TonalSystem (chromaticSize naturalScale keyScale tuning)
```

The class **Pitch** consists of a step within a scale and an offset representing a distance from the scale tone. *FIGURE 5.3* shows two octaves of pitches represented in the key of E♭ major.

TABLE 5.1 Examples of Common Tonal Systems

Name	chromaticSize	naturalScale	keyScale
Bohlen-Pierce	13	0, 1, 3, 4, 6, 7, 9, 10, 12	0, 1, 3, 4, 6, 7, 9, 10, 12
Major	12	0, 2, 4, 5, 7, 9, 11	0, 2, 4, 5, 7, 9, 11
minor	12	0, 2, 4, 5, 7, 9, 11	0, 2, 3, 5, 7, 8, 10
whole tone	12	0, 2, 4, 5, 7, 9, 11	0, 2, 4, 6, 8, 10
cents system	1200	0,100,200 . . . 1100	0,100,200 . . . 1100
Pentatonic*	12	0, 2, 4, 5, 7, 9, 11	0, 2, 5, 7, 9
MIDI	128	0 . . . 127	0 . . . 127
MIDI† with pitch bend	2,097,024	0 . . . 127	0 . . . 127
Frequency‡	2,000,000	0 . . . 20,000	0 . . . 20,000

* The pentatonic scale shown is based on the basic Chinese scale system. The pentatonic scale can be transposed to each of the twelve *lü* pitches and a five-tone scale can be constructed in the proper interval sequence. A discussion of representing pentatonic scales based on the Western twelve-tone scale can be found in [Malm, 1977].

† MIDI pitch bend values are represented in 14 bits (0 . . . 16,383). The MIDI pitch bend for no change is 8,192,. Half of the values cause the pitch to be raised and half cause it to be lowered. ZED normalizes the values to the range $-8{,}192 \le \text{value} \le 8{,}191$.

‡ The typical frequency range is 0-20kHz. The representation shown is a fixed point representation with a resolution of 0.01 Hz.

TABLE 5.2 Pitch Units for Common Tonal Systems

Name	pitch step units	pitch offset units	example pitch
Major	diatonic steps	semitones	C Major: $B^\flat = (6, -1)$
minor	diatonic steps	semitones	c minor: $B^\flat = (6, 0)$
whole tone	whole tones	semitones	c whole tone: $B^\flat = (5, 0)$
cents	semitones	cents	c based: $B^\flat = (11, 0)$
Pentatonic	pentatonic steps	<undefined>	$D^\flat$ pentatonic: $B^\flat = (4, 0)$
MIDI†	MIDI key number	<undefined>	(*key number*, 0)
MIDI with pitch bend	MIDI key number	pitch bend	(*key number*, *pitch bend*)
Frequency‡	hertz	1/100th hertz	440.15 Hz = (440, 15)

† MIDI tonal systems have a key note that is a MIDI key number in the range 0 to 127. This key number is added to the pitch to compute the absolute pitch. For example, transposition up a fifth is a key note of 7.

‡ The key note for the frequency tonal system is a floating point number that is multiplied by the pitch step frequency to compute the absolute frequency. For example, transposition up a fifth is 1.5.

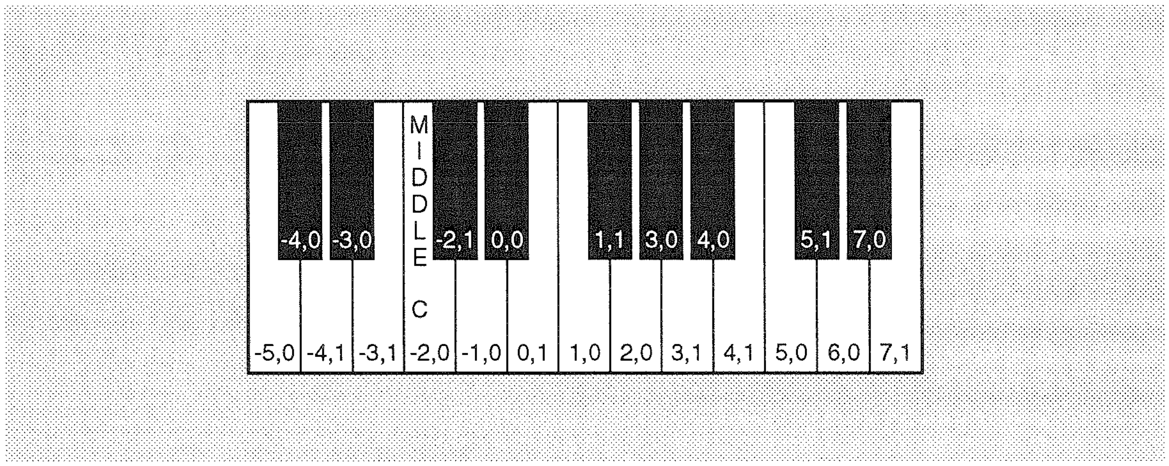


FIGURE 5.3 Pitches expressed in E♭ major.

The class **TonalSystem** has an instance variable chromaticSize that is the total number of pitches in the tonal system (12 for Western twelve-tone music). The naturalScale defines the pitches within the chromatic scale that have no accidentals—the C major scale for Western music. The keyScale is a selection of pitches from the chromatic scale that defines the pitches that are "in the key," that is, the set of pitches that have an offset of zero. The key scale is used to define tonality distinctions such as *major* and *minor* in Western music. The tonality's instance variable keyNote defines transposition. The key note is represented as an instance of the class Pitch, defined in the key defined by the naturalScale. Notes in a composition are represented as pitches with a step and offset relative to the key scale. The tonal system's tuning is used to map a pitch in the composition to a specific tuning for a synthesis instrument.

The generality of this pitch representation is demonstrated by the number of common tonal systems that can be represented. For example, the Bohlen-Pierce scale [Pierce et al., 1988], based on a thirteen pitch chromatic scale and nine pitch key scales, can be represented. The pitches are tuned with an even tempering as described with the following equation:

$$pitch_i = pitch_{i-1} \times \sqrt[13]{3}$$

TABLE 5.1 shows several examples of common tonal systems and how they could be represented in the generic tonal system. *TABLE 5.2* shows the pitch step and offset units for each of the tonal systems. As an optimization, MUSE allows MIDI key numbers to be represented with a single integer rather than an instance of the class **Pitch** with an offset of zero. In addition, frequencies can be represented with a floating point number, or as a fixed point number using an instance of the class **Pitch**.

During the performance, a tuning object is used to compute the pitch parameter for a particular synthesizer such as a frequency for a DSP instrument or a key number for a MIDI instrument. The instance variable tuning for the tonal system has a tuning object that is an instance of one of the classes defined below.

```
Object ( )
    MUSEObject ( )
        Tuning (frequencies)
            MIDITuning ( )
            FrequencyTuning ( )
            MUSEChromaticTuning ( )
                MUSEPythagoreanTuning (flatFreqs doubleFlatFreqs
                                       sharpFreqs doubleSharpFreqs)
```

The instance variable frequencies holds an array of 128 frequencies that are cached to increase runtime efficiency as frequency calculations may involve trigonometric functions, *n*th roots, or other costly computations. The performer object's playNote: method sends a message to convert the MUSE pitch to a device specific pitch. Each tuning object implements one method for each type of instrument pitch parameter. The methods for MIDI and DSP synthesis are midiForPitch:inKey: and frequencyForPitch:inKey: respectively. The frequency table may be initialized to values for any tuning system, including tempered tuning and just tuning.

The class **MIDITuning** is used when the pitches in a score are MIDI key numbers. The class **FrequencyTuning** is used when the pitches in the score are actual frequencies. The class **MUSEChromaticTuning** is used when the pitches are MUSE pitches. None of these tunings distinguish enharmonic pitches. The class **MUSEPythagoreanTuning** is used to demonstrate the use of different tunings for enharmonic pitches. The frequencies instance variable for **MUSEPythagoreanTuning** has only seven elements and holds the frequencies for the natural scale from middle C. The instance variables flatFreqs, doubleFlatFreqs, sharpFreqs, and doubleSharpFreqs hold the frequencies for the corresponding accidentals. In Pythagorean intonation, a base frequency is assigned to a pitch. Then the circle of fifths is traversed and for each fifth, the frequency of the previous pitch is multiplied by 1.5 and then normalized back into one octave. MUSE pitches with an offset of 0 use the instance variable frequencies; an offset of 1 use sharpFrequencies; an offset of -1 use flatFrequencies; an offset of 2 use doubleSharpFrequencies; and an offset of -2 use doubleFlatFrequencies. A complete description of Pythagorean tuning can be found in [Helmholtz, 1885].

The implementation of the MUSE tonal system and pitch representations includes an algebra that provides operations such as addition (transposition) and subtraction (inversion). Because pitches are expressed relative to a key note and are mapped to absolute pitches at runtime, an entire composition can be transposed changing only the key note rather than all the pitches.

Discrete Symbols

Notes, rests, and cues are referred to as discrete symbols. Cues provide a means of synchronizing a place in a score with a real-time input. The class definitions for discrete symbols are shown in the class hierarchy below.[1] The classes **Note**, **Rest**, and **Cue** inherit the instance variable time from the class **MUSESymbol**. Notes and rests have a duration that, like the time, is in units of beats. In addition to a time and duration, notes have a pitch that is an instance of the class **Pitch** defined in *§Tonality* on page 42.

```
Object ( )
    MUSESymbol (time)
        Cue ( )
        Note (duration pitch articulation)
        Rest (duration)
```

If a score file (such as MIDI) represents notes with explicit "note on" and "note off" events, MUSE's score reader pairs the events and represents them with a single note object with a time and duration. The "note on" occurs at the time of the note object and the "note off" is dynamically created during the performance and is executed at the note's time plus the duration. The duration is an abstract duration, rather than the actual length of the note. The actual length is described by the *duty cycle* [Mathews, 1969b] in the articulation of the note (described in *§Articulation* on page 48). For example, two quarter notes in 4/4 time both have a duration of one beat, but one may have an articulation of ***staccato*** and the other ***tenuto***, resulting in the actual lengths of the quarter notes being different. For monophonic synthesis instruments, the actual length does not exceed the duration. For polyphonic instruments like the piano, the actual length may exceed the duration by using the sustain pedal.

In common music notation, notes and rests have a duration and the time of a note or rest is implicit: each note or rest symbol begins when the previous one ends. Some score file formats such as MIDI use a *delta time* representation where each event has a time that is the number of time units after the previous event. The absolute time for an event is the sum of the delta times of all preceding events. The time and duration representation is isomorphic to the delta time representation. The equation below shows how the time and duration are computed from the delta time representation.

$$time_n = \sum_{i=1}^{n} deltaTime_i$$

$$duration_{note} = time_{noteOff} - time_{noteOn}$$

The time and duration representation can be mapped to delta time as well by sorting all symbols based on their time, including the implicit "note off" events that occur at the time of the "note on" event plus the duration. Then each symbol is given the delta time described by the equation below.

$$deltaTime_n = time_{n+1} - time_n$$

1. In many score files, such as MIDI score files, rests are not represented explicitly. In MIDI files rests are represented implicitly when a "note off" event is not immediately followed by a "note on" event, causing silence.

Score files that use delta time representations are converted to time and duration by a MUSE score reader when the score file is read. There are several reasons that time and duration are used by MUSE instead of delta time. First of all, it is more efficient to precompute the time of the symbol when the score file is read than it is at runtime. Secondly, merging individual "note on" and "note off" events into a single note with a duration makes it possible to interactively control the articulation and duty cycle of the notes during the performance. (If the "note on" and "note off" events were not paired into MUSE notes when the score is read, this would need to be done at runtime when the time for the "note off" event is bound.)

Articulation

Symbols for articulation and attack include ***staccato*** (short), ***tenuto*** (full length and perhaps slight emphasis), ***legato*** (smooth and connected), and ***accent*** (heavy accent or little decrescendo). These symbols often affect the duration and intensity, and effect the shape of the onset and release of the note.

The complex amplitude of a sound is often referred to as the *envelope* and can be defined by three segments: an attack, a sustain, and a decay [Mathews, 1969c]. In the simplest case, the attack, sustain, and decay (ASD) segments are simple linear functions. In general, however, each segment can be any function such that, when applied in sequence, they form a continuous function.

ZED does not attempt to provide sample level control of the envelope and instead relies on the real-time sound synthesis hardware for fine grain envelopes. (After all, humans cannot control acoustic instruments with the precision of 1/44056th of a second!) Real-time synthesizers implement the detailed ASD envelopes to reduce the control bandwidth required from the workstation. MIDI interfaces provide a maximum update rate of approximately 1,300 to 2,000 updates per second, divided across all instruments being controlled through the same MIDI interface.[2]

The maximum number of updates to DSP synthesis models is limited by the processor and bus bandwidths of the workstation. This number can be affected by the throughput of the operating system and can vary quite dramatically depending on such things as what other processes are running and their priorities. It is not practical to attempt to achieve maximum control bandwidth to the DSP because the CPU on the workstation is better utilized for interactive control. ZED therefore assumes that the necessary control bandwidth for DSP instruments is approximately that of MIDI, namely it does not exceed a few thousand updates per second.

MUSE provides two representations for articulation with subclasses of **Articulation** as shown below.

```
Object ( )
    MUSEObject ( )
        Articulation ( )
            ASDArticulation (attack sustain decay dutyCycle)
            SymbolicArticulation (selector)
```

2. The MIDI interface runs at 31,250 bits per second and MIDI updates are either two or three 8-bit bytes in length.

The class **ASDArticulation** is used for score files that explicitly represent the envelope, such as MIDI files.[3] The envelope is represented with instance variables for attack (key down velocity for MIDI), sustain (continuous pressure values for MIDI), and decay (key up velocity for MIDI). The attack and decay typically are scalar numbers. All values are expressed as a percentage of the maximum value, thus maintaining device independence.

The sustain is an array of pairs, each with the time that a sustain update occurs and the sustain value (normalized to a percentage of the maximum). The time is expressed as a percentage of the note's duration. Thus, the sustain times are scaled so that they fit in the time specified by the dutyCycle of the note. The decay, also normalized, is sent at the time of the note plus the duration scaled by the duty cycle.

A symbolic representation for articulation is defined by the class SymbolicArticulation. The instance variable selector contains the selector of a method that dynamically computes the envelope. Examples include ***accent, staccato***, and ***tenuto***. The message specified by the selector is sent to the performer at runtime. Methods can be implemented to do most anything and, because the envelope is dynamically computed, can take advantage of real-time inputs.

Representing Symbols with Messages

When symbols are encountered in a score file that do not correspond to any of the MUSE symbols described thus far, MUSE represents the symbol with an instance of the class **Message**. Messages are also used for device specific information in score files. Message objects represent messages that are sent during the performance to a performer, an instrument, the conductor, or any other named object. The class hierarchy is defined below.

```
Object ( )
    MUSESymbol (time)
        Message (receiver selector parameters)
```

The instance variable receiver contains the object to which the message is sent. The instance variable selector specifies what message is sent. The instance variable parameters holds the parameters to the method or *nil* if there are none.

An example of a symbol that is represented as a message is a damper pedal indication on a piano part. Performers who play instruments that use pedal indications implement the method damperPedalDown: for their specific instrument. Performers playing instruments that do not use a damper pedal needn't implement the method, or may implement the method to do some other task. Program changes in MIDI files are represented with the message programChange:. (The MUSE messages for other MIDI events can be found

3. As a space optimization for note symbols that originated from MIDI files that do not have release velocities and pressure values, the articulation instance variable can be the attack velocity rather than an ASDArticulation object.

in *TABLE D.2* and *TABLE D.3* in *Appendix D* on page 111.) The MUSE score file format (described in *§MUSE Score Files* in *Appendix B* on page 83) may include arbitrary messages that are sent to any named object. The score language can be easily extended by implementing methods on the performer, conductor, and other classes and referencing them in a MUSE score.

Time-varying Functions

CMN scores often have symbols that represent time-varying functions. Examples of such symbols are ***crescendo***, ***accelerando***, and ***rubato***. Symbols of this type are also represented with instances of the class **Message**. The methods that implement the message are *regenerative* in the sense that the receiver performs the task and then reschedules the message. Regenerative methods of this type can be used for a variety of purposes, such as changing interpretation symbols such as dynamics and tempo over time. Regenerative methods can also be defined to generate notes using random number generators, or to play repetitive sequences. In addition, regenerative methods can be used to directly control synthesis parameters that vary over time, such as vibrato and timbre changes.

An example of a method that will cause a ***crescendo*** from the current dynamics level to a ***forte*** is shown below. The dynamics level will be increased by 0.05 four times per beat until it reaches 0.95. (This is 95% of the maximum, assuming that the dynamics level is normalized to be in the range 0.0 to 1.0.)

```
Conductor method
crescendoToForte
    | dynamics |
    "Get current dynamics."
    dynamics := interpretationContext dynamics level.
    "If already loud enough, terminate."
    dynamics >= 0.95 ifTrue: [^self].
    "Set new dynamics level."
    interpretationContext dynamics level: dynamics + 0.05.
    "Schedule next increment of the crescendo for 0.25 beats from now."
    self scheduleMessageIn: 0.25
        receiver: self
        selector: #crescendoToForte
        with: nil.
```

Organizing Symbols into Scores

Most digital music representations do not provide any structure beyond simple lists of low level events. In developing MUSE, it was apparent that the representation, as well as systems that use the representation such as ZED, would benefit from an abstraction mechanism for capturing the inherent hierarchical structure found in many musical compositions. MUSE provides *hierarchical* structures for assembling notes that are based on techniques developed for VLSI CAD applications [Whitney, 1985].

Hierarchical composition begins with small building blocks called *cells*. Cells can act as *templates* that can be *instantiated* and combined to create larger building blocks. These larger building blocks can then be combined, and so on. In VLSI CAD, cells are composed in rows and columns. For example, a cell that is a one bit adder can be replicated and composed in a row to create a sixteen bit adder. When cells are instantiated, a variety of transformations can be applied, such as spatial translation and rotation. At each level of the hierarchy, higher-level semantics are defined by abstracting from the details of the structures below. This hierarchical approach has been shown to aid in managing the complexity of large networks of objects, making the understanding of such structures tractable.

Composers often create *motifs*—melodies or phrases consisting of notes and rests—that are used multiple times throughout a composition. Composers often transform the motif by applying pitch inversion, transposition, or different tempi. In MUSE, the discrete symbols for notes, rests, and messages are the cells. Cells can be composed in series by placing them one after the other in time. The resulting melody or phrase is called a *sequence*. Cells can also be composed in parallel, indicating that the symbols are to be played simultaneously. This type of composition is used to create chords or harmony. The resulting structure is a *parallel sequence*. Series and parallel sequences can then be hierarchically composed into larger, more elaborate series and parallel structures. Such scores take on a *tree* structure where the *leaves* of the tree are notes, rests, and messages.

MUSE supports a number of transformations on sequences. When sequences are instantiated, they are translated in time. This means that all times for the symbols in the sequence are expressed relative to the beginning of the sequence. When the sequence is instantiated in the score, all of the times of the symbols will be offset by the location of the sequence within the score. Sequences may also be transformed by setting a private interpretation context for the sequence that only applies to the symbols within the sequence. Another type of transformation is accomplished with a selector representing a method that is applied to each note in the sequence.

At the highest level of the hierarchy are the score and parts. A performer's part can be viewed as a series sequence and the conductor's score as a parallel sequence of parts. The class definitions for objects used for series and parallel sequences are shown below.

```
Object ( )
        MUSESymbol (time)
        MUSEStructure (name events interpretationContext transformation current)
            Sequence ( )
                Part ( )
                CueSheet ( )
            ParallelSequence ( )
                Score (cueSheet)
            RepeatedSequence (numberOfRepeats count)
```

All sequences are instances of a concrete subclass of the abstract class **MUSEStructure**. All sequences have a name that is used to reference the sequence. The instance variable events is a list of symbol objects that are instances of any concrete subclass of **MUSESymbol** except **Part** and **Score**, but including **Sequence**, and **ParallelSequence**. The symbol objects within a MUSE structure have times that are relative to the beginning of the structure. They are located in the score by adding their time to the time of the enclosing structure. This calculation is done recursively when an event is scheduled for each symbol.

The class **RepeatedSequence** is used to encapsulate the sequence stored in the instance variable events. This class is used to create multiple instantiations of the same sequence. Repeated sequences do not require that the actual data be replicated. The instance variable numberOfRepeats defines the number of times that the sequence is to be repeated. The count instance variable is used to keep track of the current iteration during the performance. When the end of the sequence is reached, the count is incremented and the sequence is reset. When the count is incremented beyond the number of repeats, the iteration is terminated.

The instance variable interpretationContext is used to define an interpretation context that only applies to the sequence. An interpretation context can be defined at the beginning of the score for all parts in the score, in a part for all symbols in a part, or in a sequence for all symbols in the sequence. The instance variable transformation is an optional selector that can be used to perform more sophisticated transformations on the sequence, including those that use real-time control inputs.

The score is an instance of **Score** consisting of a collection of parts stored in the instance variable events. Each part is an instance of the class **Part** that is assigned to a performer. The class **Part** is virtually identical to the class **Sequence** and is separated out simply to illustrate that it has two special properties: it is a root class that cannot be in the event list of another MUSE structure object; and its instances are one to one with performer objects. Similarly, the class **Score** is virtually identical to **ParallelSequence** except that all objects in its event list are instances of **Part**. There is exactly one instance of **Score** and this instance is associated with the single ZED conductor object. The class **CueSheet** contains cue symbols and interpretation symbols that are global and apply to all performers that do not otherwise have an interpretation context.

References

Balzano, G. J. The Pitch Set as a Level of Description for Studying Musical Pitch Perception. In Clynes, M. (ed), *Music, Mind, and Brain: The Neuropsychology of Music.* New York, NY: Plenum Press, 1982, pp. 321-325.

Byrd, D. A. *Music Notation by Computer.* Ann Arbor, MI: University Microfilms, 1984, pp. 10-14.

Creston, P. *Principles of Rhythm.* Melville, NY: Belwin Mills, 1961, p. 3.

Dyer, L. M. MUSE: An Integrated Software Environment for Computer Music Applications. In Berg, P. (ed), *International Computer Music Conference at Royal Conservatory, The Hague, Netherlands.* San Francisco, CA: Computer Music Association, 1986, pp. 167-172.

Dyer, L. M. *A Semantic Digital Representation for Music.* Master's Thesis, Unpublished, Caltech, Pasadena, CA, 1987.

Foss, L. The Changing Composer-Performer Relationship: A Monologue and a Dialogue. In Boretz, B. and Cone, E. (eds), *Perspectives on Notation and Performance.* New York, NY: W. W. Norton & Company, 1976, p. 37.

Frydén, A. and Sundberg, J. Performance Rules for Melodies. Origin, Functions, Purposes. In Buxton, William (ed), *International Computer Music Conference at IRCAM, Paris, France.* San Francisco, CA: Computer Music Association, 1984, pp. 221-224.

Grout, D. J. *A History of Western Music.* New York, NY: W. W. Norton, 1973, p. 225.

Helmholtz, H. L. F. *On the Sensations of Tone as a Physiological Basis for the Theory of Music.* New York, NY: Dover, 1954, p. 433. Unabridged, unaltered republication of translation by Ellis, A. J. *Die Lehre von den Tonempfindungen.* Germany: Longmans & Co., second edition, 1885.

Malm, W. P. *Music Cultures of the Pacific, the Near East, and Asia.* Englewood Cliffs, NJ: Prentice-Hall, 1977, p. 148.

Mathews, M. V. *The Technology of Computer Music.* Cambridge, MA: MIT Press, 1969 (1969a, pp. 115-172; 1969b, p. 90; 1969c, p. 130).

Mathews, M. V. Graphical Language for the Scores of Computer-Generated Sounds. In Boretz, B. and Cone, E. (eds), *Perspectives on Notation and Performance*. New York, NY: W. W. Norton & Company, 1976, p. 162.

Pierce, J. R., Mathews, M. V., Reeves, A., and Roberts, L. *Theoretical and experimental explorations of the Bohlen-Pierce scale*. Journal of the Acoustical Society of America, 84(4):1214-1222, 1988.

Rastall, R. *The Notation of Western Music*. New York, NY: St. Martin's Press, 1982, p. 1.

Read, G. *Music Notation*. New York, NY: Taplinger, 1969, pp. 24, 27.

Read, G. *Modern Rhythmic Notation*. Bloomington, IN: Indiana University Press, 1978.

Smith, R. B. *The New Music*. Oxford, Great Britain: Oxford University Press, 1975.

Stone, K. Problems and Methods of Notation. In Boretz, B. and Cone, E. (eds), *Perspectives on Notation and Performance*. New York, NY: W. W. Norton & Company, 1976, p. 13.

Whitney, T. E. Hierarchical Composition of VLSI Circuits. Ph.D. Thesis, *Caltech Computer Science Technical Report,* Caltech-CS-5189:TR:85, California Institute of Technology, Pasadena, CA, 1985.

Chapter 6

Real-time Performance

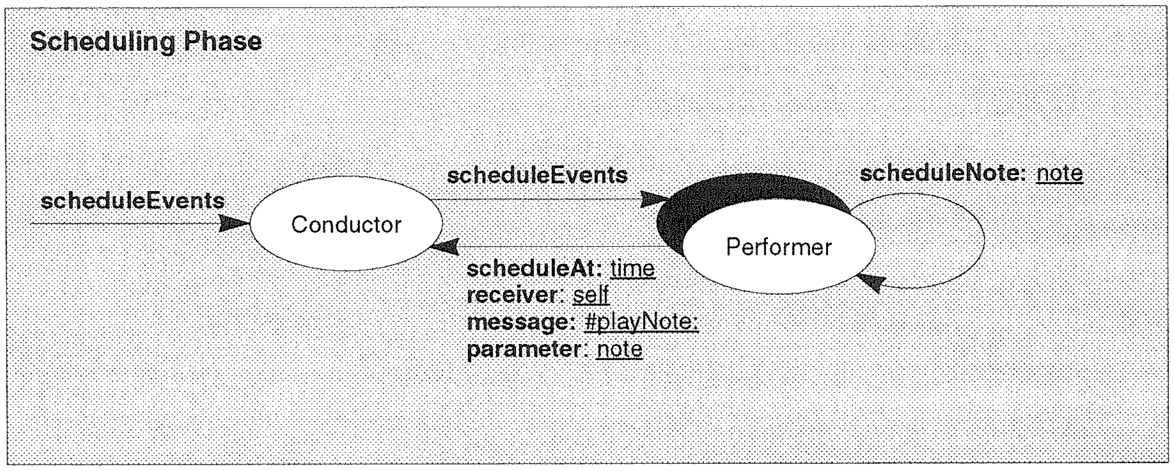
This chapter describes ZED's real-time performance mechanism. First, an overview of the performance is presented. The scheduling phase is outlined and the performer's default schedule method is defined, along with the default method to play notes. The details of the mapping of an abstract MUSE note to MIDI packets are described. The runtime execution loop is discussed and example patches are shown. Finally, several simple tempo tracking algorithms are presented and discussed.

ZED Performance Overview

ZED divides the performance into two parts: a compile time scheduling phase that takes place before the performance; and the actual runtime performance. Before the performance begins, the conductor sends the message scheduleEvents to each of the performers in the conductor's performers collection. The performers iterate through the symbols in their respective parts. For each symbol, a message is sent to the conductor requesting that an event be scheduled for the symbol. The performers schedule their events by sending themselves the message specified by their scheduleMessage instance variable. This message can be specified in a ZED configuration file or, if none is specified, the message scheduleNote: is used. This method schedules the message playNote: to be sent to the performer at runtime. *FIGURE 6.1* illustrates the default messages that are sent during the scheduling phase and during the performance.

Performer Runtime Methods

The details of the playNote: method for a MIDI performer are shown in *FIGURE 6.2*. The parameter to the method is a note and the method uses the interpretation context of the performer in the calculation. The



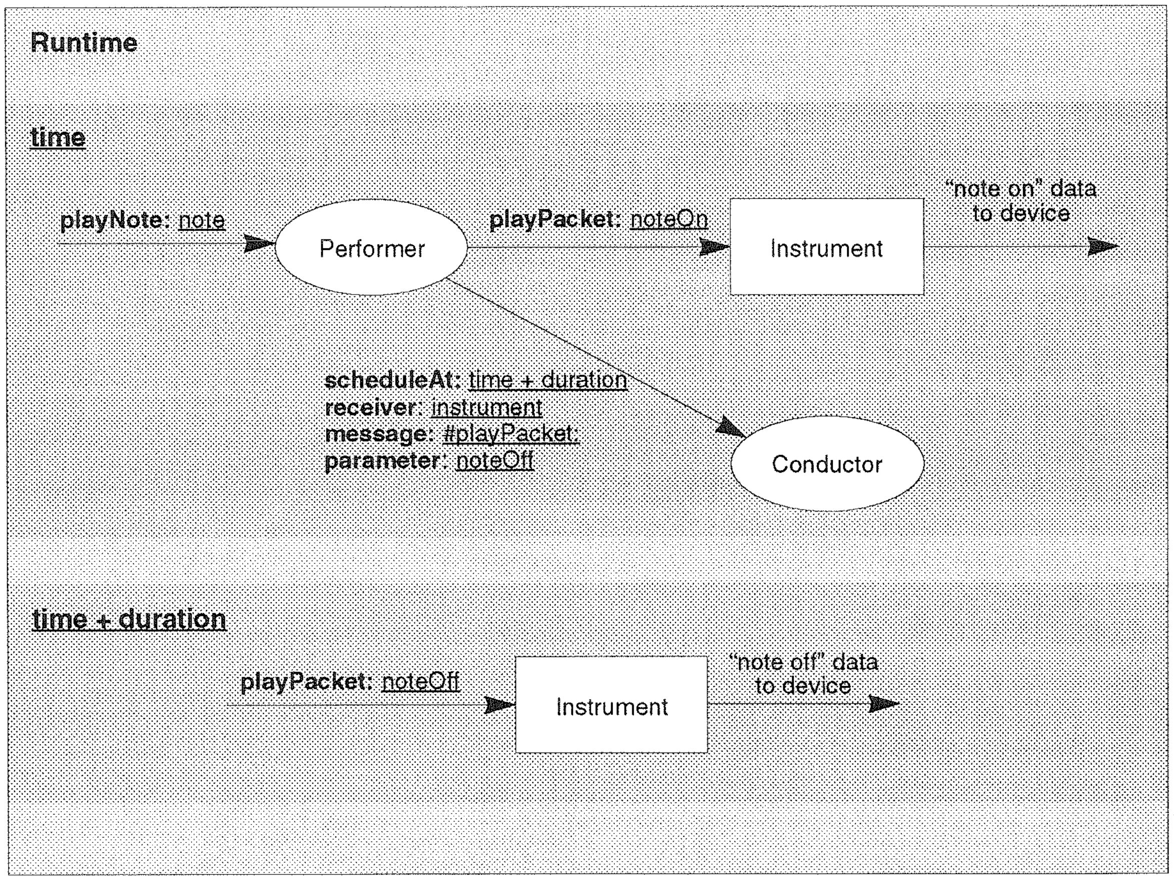


FIGURE 6.1 Default scheduling and runtime methods.
The scheduling phase begins when the message scheduleEvents is sent to the conductor which is then sent to each of the performers. Each performer schedules an event for each symbol in their part by sending the message in their scheduleMessage instance variable. The default is the message scheduleNote:. The method scheduleNote: sends a message to the conductor to schedule an event that causes the message playNote: to be sent during the performance. The runtime performance messages are also shown. The message playNote: is sent to the performer from the conductor at the time the event is to take place, causing a "note on" packet to be computed and sent to the instrument as the parameter to the playPacket: message. After the packet is played, the performer computes a "note off" packet and dynamically schedules a playPacket: message to be sent directly to the instrument at the time of the note plus the duration.

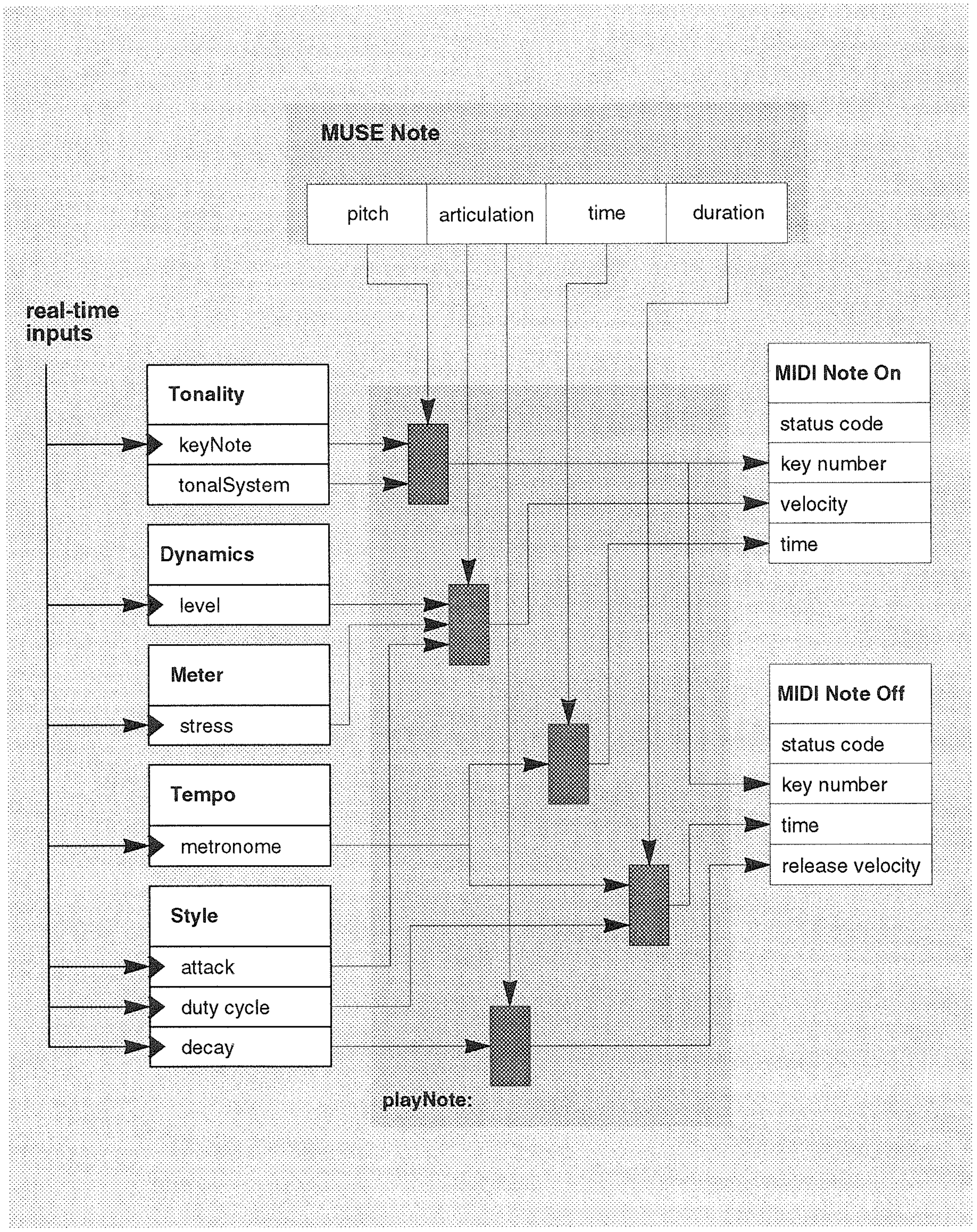


FIGURE 6.2 Detailed MIDI performer runtime method for playing a note.
The small gray boxes represent the computations performed by a MIDI performer object to compute the "note on" and "note off" packets for a MUSE note. The performer object does not actually compute the physical time of the packet, as this is computed during the performance by the scheduler using the tempo. The figure also shows how the real-time control of interpretation symbols affect the interpretation of a MUSE note as it is mapped to instrument packets.

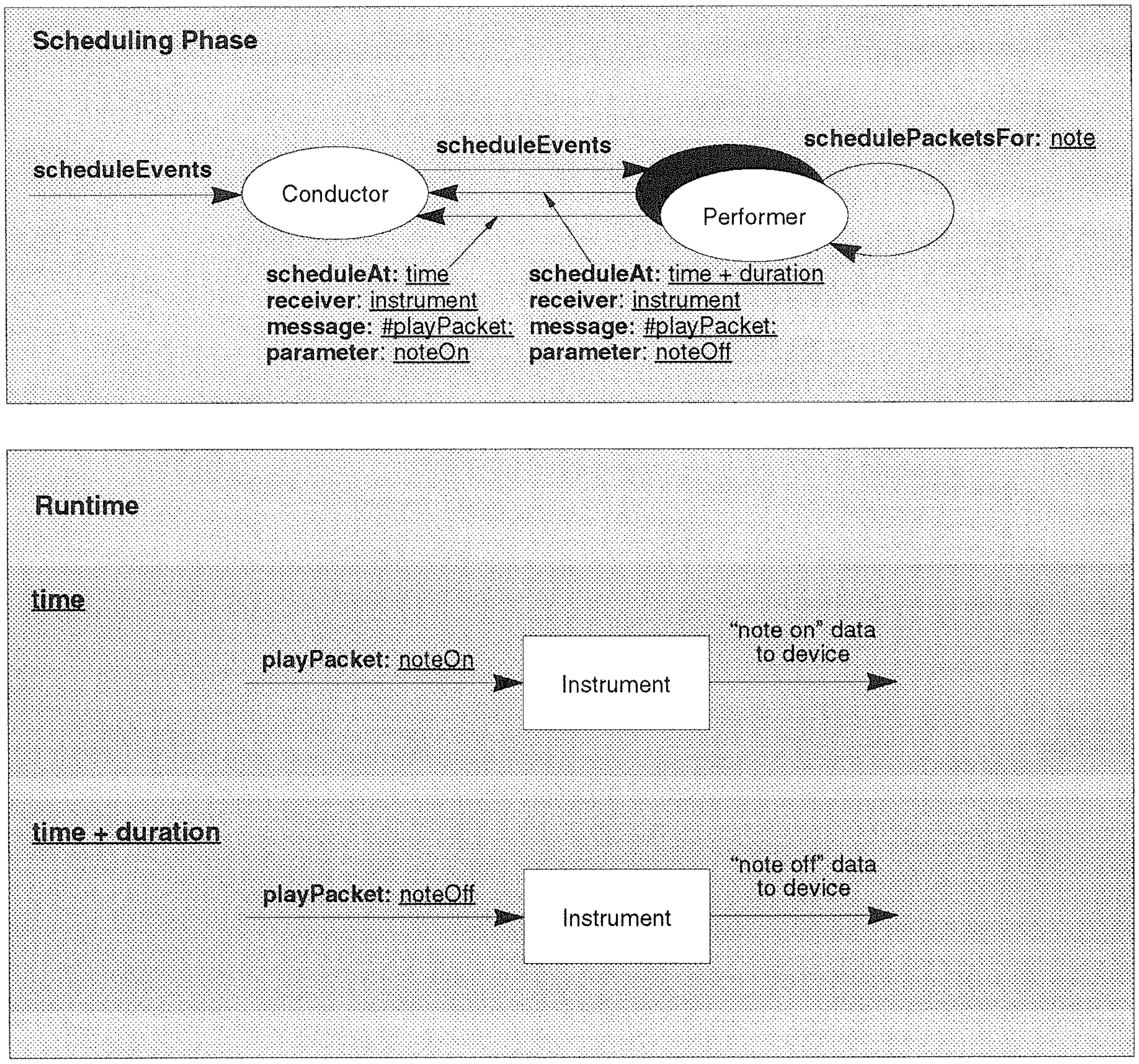


FIGURE 6.3 Scheduling and runtime methods for a static performance.
If a performance only has the tempo under real-time control, the "note on" and "note off" packets can be computed by the performer during the scheduling phase and the performer can schedule the events to play the packet to be sent directly to the instrument. This type of performance has minimum runtime overhead.

method computes a "note on" packet that is sent to the instrument immediately, and a "note off" packet that is scheduled for some amount of time in the future. Each of the interpretation symbols can be under real-time control.

Computing all of the data for the MIDI packets at runtime could potentially effect timeliness. Depending on which interpretation symbols are under interactive control, various values in the instrument packets can be precomputed. This static data can be precomputed by changing a performer's scheduleMessage to do the static computations and then schedule a play event that computes the remaining values. For example, if the tonality and tuning are not under interactive control, the key number can be precomputed. If the duty cycle in the style is not under interactive control, then the "note off" event can be scheduled during the scheduling phase. At the opposite extreme of the default situation is one where the performer's schedule message computes the entire packet and schedules an event that sends the packet directly to the instrument at runtime. In this case the performance is not affected by real-time input and the performance has maximum runtime efficiency, but is completely static. The messages for such a performance are shown in *FIGURE 6.3*. Thus, each performance can be optimized for maximum runtime efficiency without loss of flexibility.

Performance Execution Loop

FIGURE 6.4 shows the basic performance execution loop and the details of the alarm calculation. When the performance begins, the conductor object gets the time of day from the system clock. The physical time for the next node is computed and an alarm is set. The function for setting the alarm takes as a parameter the amount of time *from when the function is called.* The physical time and alarm time calculations are shown below for the *i*th node in the scheduler. The variable *physicalTime* is a time of day representing when the next event is to take place. The *systemclock* represents a function that returns the time of day when the function is called. The variable $physicalTime_0$ is the time that the performance began. The *abstractTime* is the time (in beats) that the event is scheduled to take place.

$$\begin{aligned} physicalTime_0 &= systemclock \\ physicalTime_i &= physicalTime_{i-1} + \left((abstractTime_i - abstractTime_{i-1}) \times \frac{60}{mm} \right) \\ alarmTime_i &= physicalTime_i - systemclock \end{aligned}$$

An important property of the calculation of the alarm's time is that it is computed as a function of the system clock at the moment that the alarm is set. This takes into consideration the overhead required to respond to the previous alarm, play the events, and increment the physical time. Thus the performance is realigned every time an alarm is set, preventing the accumulation of error that would otherwise result.

Once the alarm is set, ZED goes to sleep. When the alarm goes off, a wake up function is executed. (This is the same as the semantics defined by the UNIX™ functions *signal* and *ualarm* [Bell Labs, 1982].) The wake up function plays the current events, gets the next node from the scheduler, sets the next alarm, and goes back to sleep.

Example Patches

Any aspect of the performance can be controlled interactively by the user by defining a patch that will select particular inputs and act on them by sending messages. The most common way to control the performance is to directly update the interpretation symbols with real-time control values. An example of a patch that controls the global tempo and one that controls a single performer's dynamics is shown in *FIGURE 6.5*. A selection of more complicated patches are described in the following sections.

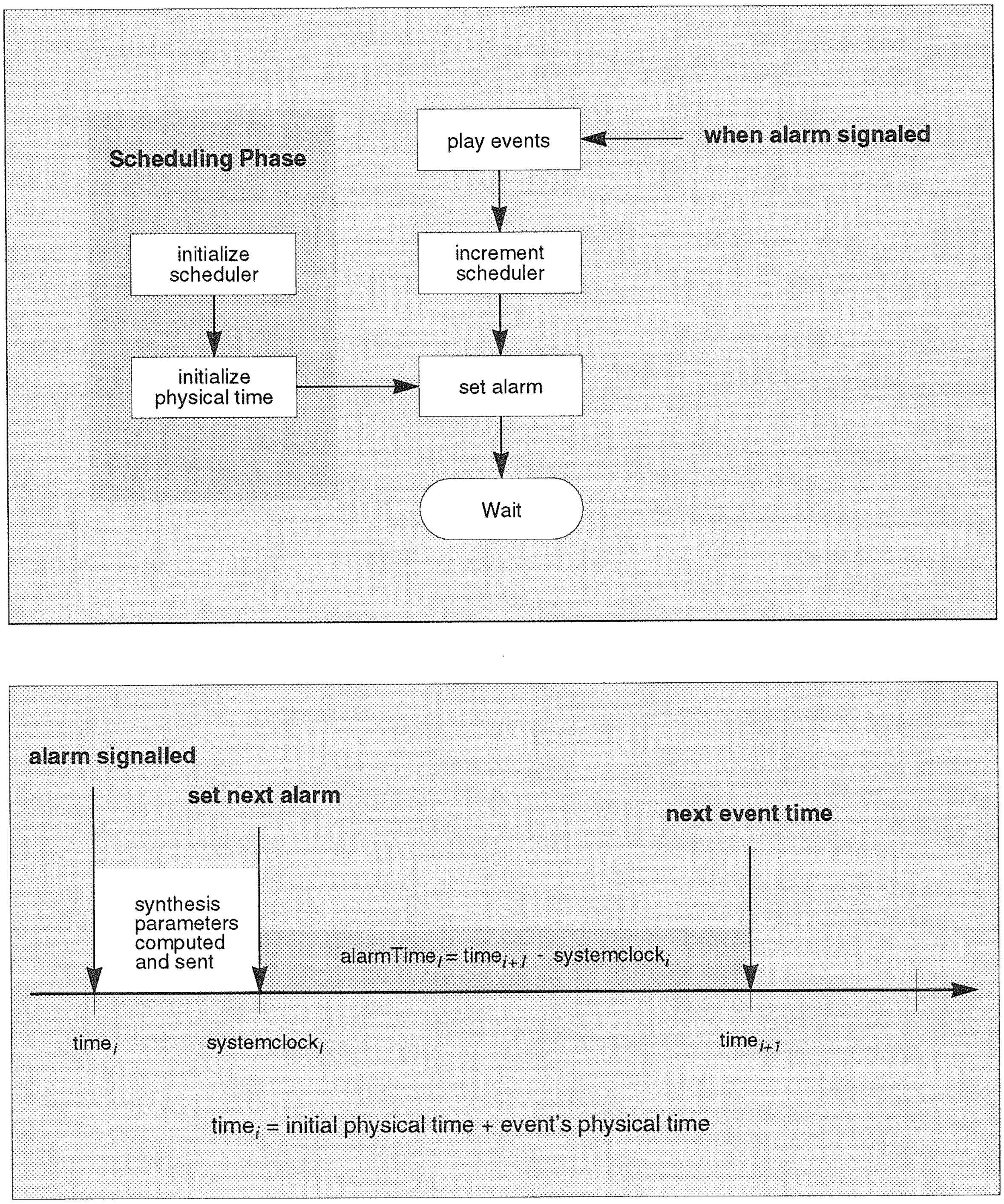


FIGURE 6.4 Real-time performance execution loop.
When the performance begins the time is captured from the system clock. The physical time is incremented by the physical time of the current node (the number of beats from the beginning of the composition scaled by the tempo). Then an alarm is set and the system goes to sleep. The alarm's time is computed as a function of the system clock to prevent the accumulation of error resulting from the time it takes to compute the synthesis parameters and send them to the device. When the alarm is signaled, the events for the current node are played, the current node is incremented, and the cycle repeats. The scheduler is initialized and the current node set before the performance begins.

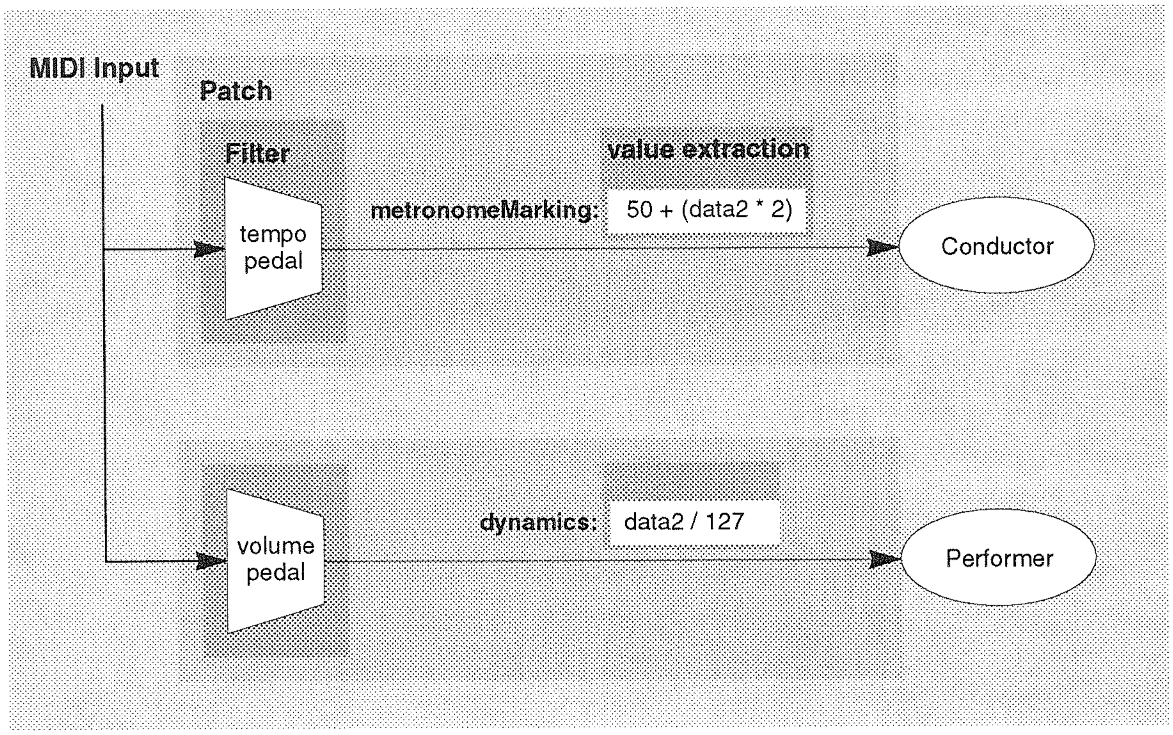


FIGURE 6.5 Example patches for controlling the global tempo and local dynamics.
When a MIDI tempo pedal input is received, the data2 value (in the range 0 to 127) is used to compute the new metronome marking. The message metronomeMarking: is then sent to the conductor, causing the conductor's tempo object to be changed. When a MIDI volume pedal input is received, the data2 value is used to compute a dynamics level for a particular performer.

Triggering a Sequence

A patch can be defined to dynamically start a sequence when a particular MIDI event is received such as a "damper pedal down" input. The sequence can be stopped when a "damper pedal up" input is received. *FIGURE 6.6* shows an example of such a patch. The method start, implemented by the sequence, adds the corresponding sequence's static queue to the queue list in the scheduler. The method stop removes the queue from the scheduler's queue list. A more detailed description of the implementation of dynamically triggered queues can be found in *§Triggering Sequences in Real-time* in *Appendix A* on page 79.

Sampling a Sequence

A more complicated patch is one that samples incoming MIDI events and stores them in a sequence. To enable this, a patch is defined that causes another patch to be dynamically connected. *FIGURE 6.7* shows a patch that begins sampling MIDI key up and down events when a damper pedal goes down and ends sampling when it goes up. When a "damper pedal down" input is received, the patches #KeyDownPatch and #KeyUpPatch are connected by the method connect. This method connects the patches by adding them to the conductor's patches collection. These two patches filter all MIDI "key down" and "key up" events and

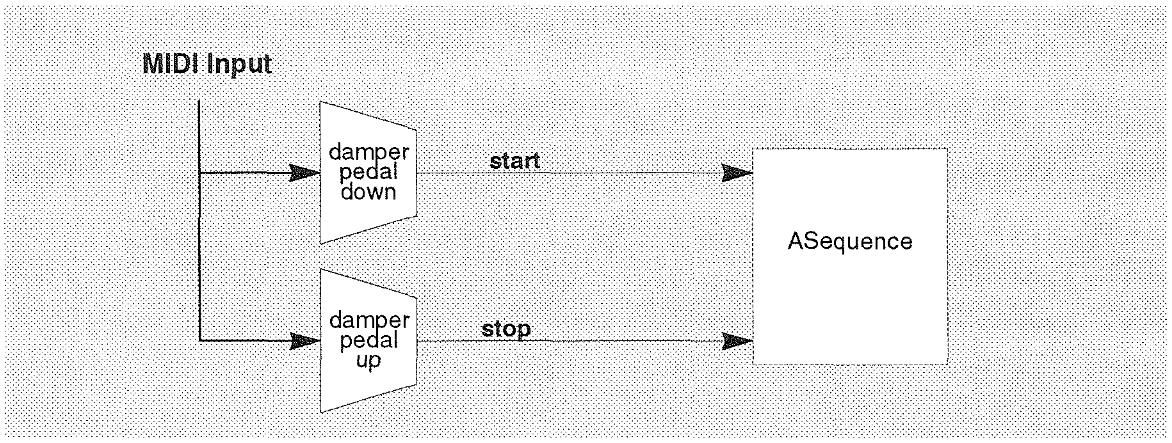


FIGURE 6.6 Example patch for triggering a sequence.
Two patches are shown. The message start is sent to the sequence named #ASequence when a "damper pedal down" event is received. The message stop is sent when a "damper pedal up" event is received.

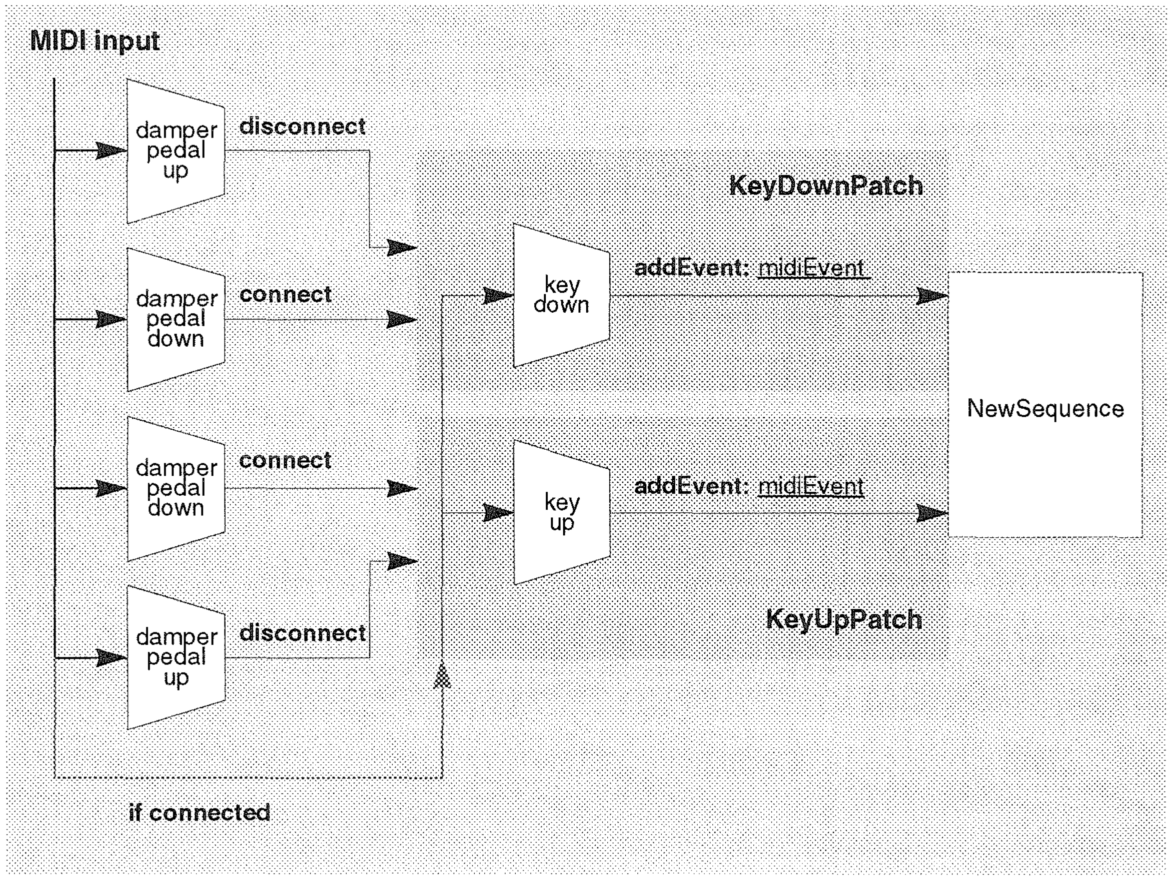


FIGURE 6.7 Example patch for sampling MIDI "key up" and "key down" events.
When a MIDI "damper pedal down" event is received, the two patches named #KeyDownPatch and #KeyUpPatch are connected (added to the conductor's patches collection). MIDI "key down" and "key up" events are added to the sequence named #NewSequence until a "damper pedal up" event is received, at which time the patches are disconnected.

add them to the sequence called #NewSequence. When the damper pedal goes up, the patches are disconnected. This is done with the method disconnect which removes the patches from the conductor's patches collection

Special Tempo Controls

As described earlier in this chapter, the tempo can be controlled by directly updating the metronome marking from a continuous real-time control input. Each time an input is received, a message is sent to the conductor to affect the global tempo, or to a performer to affect a local tempo. If there are multiple independent time references in the performance, (i.e., if more than one tempo object is under real-time control), then a message is sent to the scheduler to notify it of each tempo change. The scheduler in turn cancels the currently pending alarm and computes a new one using the new metronome marking. This allows the system to instantly respond to tempo changes.

In addition to direct control, there are a number of other ways to control the tempo. Two techniques are discussed below. The first is setting a new tempo with preparatory beats and the second is synchronizing the performance with a cue sheet.

Preparatory Beats

In the live orchestra, the conductor sometimes gives the performers a couple of preparatory beats before a tempo change. The conductor indicates the new tempo before it happens, thus preparing the performers so that they are synchronized at the moment that the new tempo takes effect. Controlling the tempo with preparatory beats is a convenient way for a user with a trigger input device such as a MIDI drum or the Stanford Radio Baton to set the tempo in a ZED performance. The user provides two inputs in the new tempo and at the moment that the second one is received, a new tempo is computed. This type of tempo control can be easily implemented with a patch that remembers the time of the first input and computes a metronome marking when the second input is received. (The patch should then reset its state so that it can accommodate more than one tempo change.) The metronome marking is computed during the performance with a value extraction method that implements the function shown below. The variable *physicalTime* is used to denote the physical time in units of seconds (set by the system clock) that the input is received. The abstract time between the two successive inputs is assumed to be one beat.

$$mm = \frac{60}{physicalTime_2 - physicalTime_1}$$

Synchronizing the Performance with Cues

A cue sheet is used to synchronize the events in the score with real-time inputs. The cue sheet contains a list of cue objects, each with a time (in beats) from the beginning of the score. The inputs from a particular input controller are matched one-to-one with these cues. The tempo is adjusted each time a new input is received. If the cue is received earlier than it was expected, then all events before the cue are skipped. If the cue is received later than it was expected, then the system waits for the cue. This mechanism is suitable for controlling a synthesized accompaniment to follow a live player and has been successfully employed with the Conductor Program and other accompaniment systems.

Each time an input is received, a new metronome marking is computed. The computation is similar to the one used for preparatory beats, except that the metronome marking must be scaled by the number of beats between the cues. The calculation for the *i*th cue input is shown below. This calculation uses the variable *cueTime* to represent the cue object's time instance variable (in beat units).

$$mm = \frac{60}{physicalTime_i - physicalTime_{i-1}} \times (cueTime_i - cueTime_{i-1})$$

Rehearsals

This cueing algorithm is adequate when the changes in tempo are small. The algorithm does not effectively track unexpected tempo changes of large amounts. When a cue arrives, a new tempo for the next beat is computed based on the tempo of the previous beat. Therefore, the system assumes that the performance is going to proceed at the same tempo. If the tempo gets faster, the last note before each cue is shortened when the score is synchronized with the input. If the tempo slows down, there may be silence while the system is waiting for the next cue.

Rehearsals and learning can be incorporated into ZED to better predicting the future tempo. The user rehearses a performance before a concert by playing the composition several times. The system follows the user's inputs and retains the timing information of the cue inputs during the rehearsals. (Between rehearsals, the information can be stored in computer memory or in a disk file.) During an actual performance, the stored timing information is used to allow the computer to better predict when each cue will arrive during the performance.

A simple learning algorithm can be implemented to maintain an array of the expected time (in physical time units) between successive inputs during the performance. The array is initialized to the physical time of the durations of the cues based on the tempo with no interactive control. During each rehearsal, the times of the new inputs are incorporated into the array. There are many ways to incorporate these values. The simplest way is to store the values of the previous rehearsal. A more robust approach is to maintain a running average. The equations below show the calculation of a running average. The running averages are stored in the array

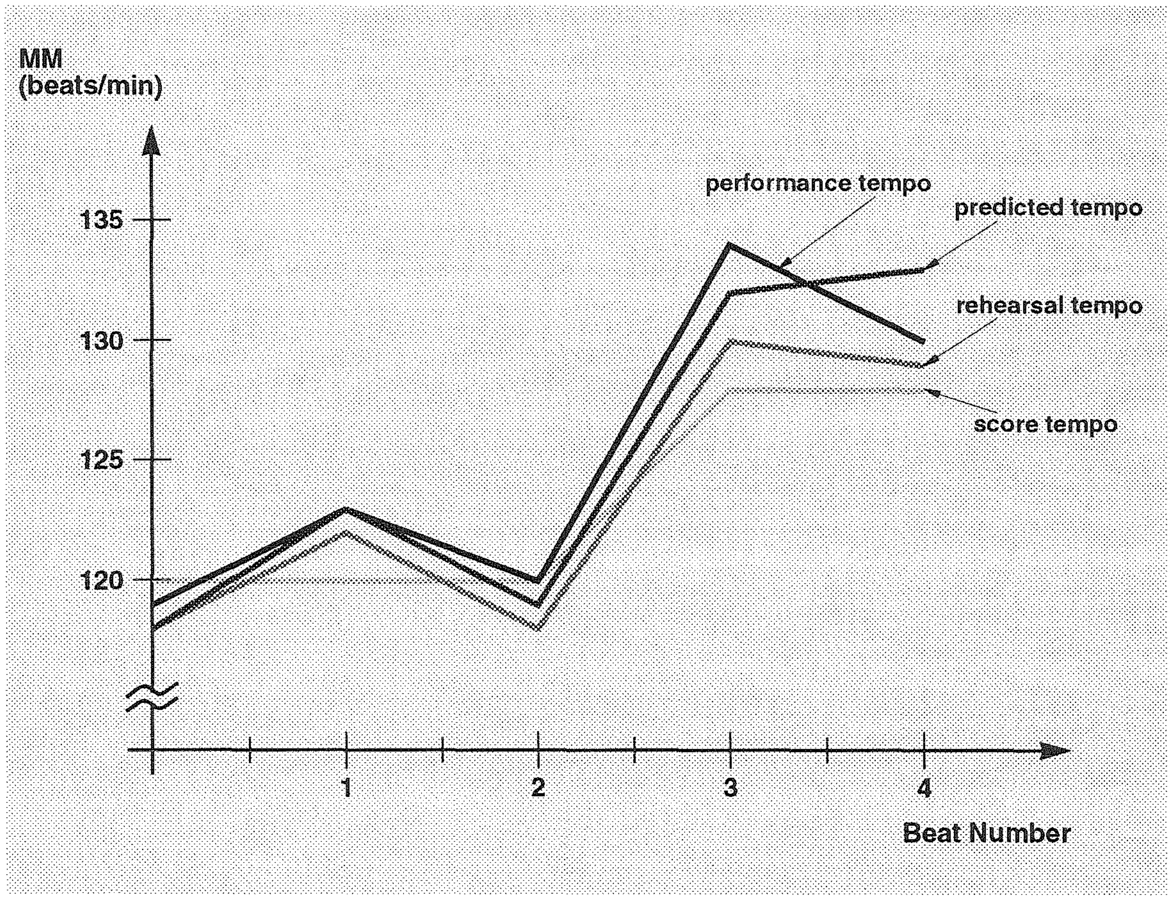


FIGURE 6.8 Tempo prediction for performance that follows rehearsal tempo.
The above graph shows that if the performance tempo has the same basic shape as the rehearsed tempo, the algorithm tracks quite closely. The algorithm breaks down when the performance tempo changes faster than the rehearsed tempo.

Estimate, indexed by the number of the cue in the score. The variable N is the number of the rehearsal and *physicalTime* is the time (in seconds) when the real-time input is received. The time in beats between successive cues is represented with the variable *cueTime*. When the ith cue is received, its time is averaged into the previous estimate and a metronome marking is computed based on the estimated time of the next cue.

$$Estimate_i = \frac{(Estimate_i \times (N-1)) + (physicalTime_i - physicalTime_{i-1})}{N}$$

$$mm = \frac{60}{Estimate_{i+1}} \times (cueTime_{i+1} - cueTime_i)$$

A problem with the metronome marking calculation above is that it does not use the data in the current rehearsal, i.e., the inputs *during the performance* do not affect the system. A problem arises if, during the performance, the user plays the basic tempi that were rehearsed, except that they are all played slightly faster (or slower). This is not uncommon because live performers are often slightly anxious during the

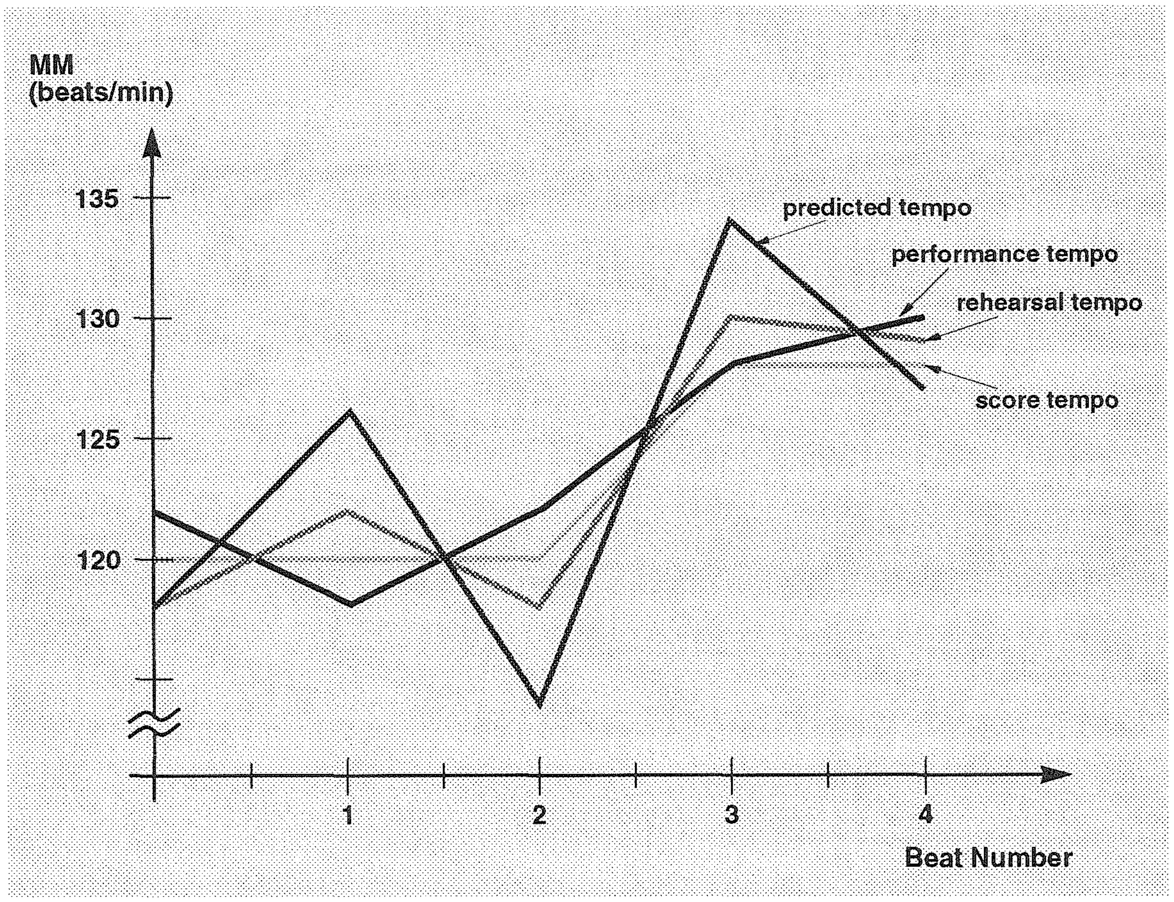


FIGURE 6.9 Tempo prediction for performance that differs from rehearsal.
The above graph shows that if the performance tempo has a slope opposite of the rehearsed tempo, the tempo predictions diverge from those of the performance, thus causing the synthesizers to be ahead or behind the cues.

performance. An improved calculation for the metronome marking is shown below. This calculation offsets the estimated time for the next input by the error in the current estimate.

$$mm = \frac{60 \times (cueTime_{i+1} - cueTime_i)}{Estimate_{i+1} + ((physicalTime_i - physicaltime_{i-1}) - Estimate_i)}$$

The new calculation handles the particular case when the tempo changes are basically correct but the performance is slightly slower or slightly faster than the rehearsals. *FIGURE 6.8* shows how the algorithm tracks a small set of sample data. The algorithm tracks real-time inputs that are correlated with the rehearsal. As the difference between the performance and the rehearsal increases, the predicted tempo diverges from the performance tempo. *FIGURE 6.9* shows the algorithm's poor performance when the performance is completely uncorrelated with the rehearsal data.

Summary

This chapter describes the ZED real-time music performance. A technique was shown for exploiting both the runtime efficiency of static systems, and the runtime flexibility of dynamic systems. This balance is achieved by providing the user with the ability to easily customize the scheduling of events so that all static data can be precomputed before the performance begins without affecting the interactive control of the performance.

The tempo tracking algorithms presented are very simple. It is not possible to define a perfect algorithm that can adjust to sudden and arbitrary changes in tempo. After all, human performers cannot follow such changes when they are not notated in the score and are not rehearsed. It is somewhat surprising that in practice, for many performance situations that do not include improvisation, the simple learning algorithm presented works reasonably well with a very small amount of runtime computation. Clearly, however, more sophisticated tracking algorithms are needed. The primary issue in developing such algorithms is providing better and more robust tempo predictions while still achieving timeliness. In addition, in order for the tracking algorithms to be useful in the context of a system such as ZED, they must be designed to achieve timeliness independent of the particular composition and patches in the performance. Further study is needed to determine how much computation can be devoted to score tracking without jeopardizing timeliness. In addition, careful analysis is required to determine how score tracking algorithms that requires a large amount of computation affect the performance as the number of real-time input controllers and the number of performer objects in the system are increased.

References

Bell Telephone Laboratories, Inc. *UNIX™ Time-Sharing System: UNIX™ Programmer's Manual.* New York, NY: Holt, Rinehart and Winston, 1982.

Conclusions

This thesis presents a new software architecture for real-time music performance under interactive control. The approach is an implementation of a model of live orchestra performance that captures some of the dynamic behavior of the orchestra. A software simulation of music performance was implemented using an object-oriented paradigm in a system called ZED. ZED has objects that are one-to-one with the components of the live orchestra: a conductor, performers, and instruments. In addition, ZED includes an abstract digital score representation that is score file independent and synthesis device independent. A user can interactively control the performance using one or more input controllers. Patches can be defined to control the performance at three levels: the overall ensemble performance is controlled via the conductor object; the expressiveness of a particular part can be controlled via the performer of that part; and low-level control can be achieved by sending messages directly to an instrument. Thus, ZED embodies the many of the features of existing real-time music systems including patching programs, accompaniment systems, and object-oriented tool kits.

ZED's performer objects implement a method that maps the abstract score representation for notes to device specific parameters for their instrument. This design provides the basis for several types of extensibility: performer objects can evolve and become more sophisticated over time (like their human counterparts) by modifying the "play note" method; specialized performer objects can be defined that can control the subtle properties of a specific synthesis algorithm by creating a new subclass and defining the "play note" method; existing DSP synthesis instrument libraries can be incorporated into ZED by defining new performer and packet classes to interface to the instruments; and new synthesis technologies can be incorporated by defining new performer, instrument, and packet classes. Thus, ZED's design inherently supports evolution and specialization. Furthermore, the architecture is general enough to facilitate the control of other non-music devices including audio mixers, lighting boards, and video tape and laser disk players.

ZED also provides a mechanism for customizing each performance to have runtime efficiency without losing runtime flexibility. Methods can be defined on the performer classes and selected by each performer object to schedule events before the performance and precompute all packets values that are not affected by real-time inputs. Thus, the user can easily achieve maximum performance by defining simple methods rather than having to rewrite the real-time performance loop. ZED includes an efficient hybrid scheduler that handles statically scheduled and dynamically scheduled events separately. The scheduler provides a simple memory management scheme that virtually eliminates the need for runtime memory allocations. The scheduler also supports multiple independent time references that can be under separate real-time control.

There are a number of areas of ZED that require further study. The performer objects currently being used are too simplistic. Although it is not possible to completely model human musical virtuosity, models that are more sophisticated than the ones currently in use would improve the expressiveness of the performance. Currently in ZED, the performer objects play each note independently and do not maintain information about adjacent notes in phrases. ZED's design provides the ability to extend the methods that play a note to incorporate a rule-based algorithm or other techniques that take past and future notes into consideration, as well as the parts of other performers. Such changes can be implemented by changing the existing methods or by creating new performer classes and overriding the methods. As a part of such extensions, further investigation is required to better understand the full dynamic range of the sound synthesis algorithms and how they can be controlled to provide maximum musical expression. Furthermore, the MUSE representation of musical style can be extended through subclassing to capture more detailed information representing more subtle musical style.

ZED's tempo tracking algorithms are also very simplistic. An interesting result of the tempo tracking experiments presented in this thesis is how well the system can follow the tempo with very little runtime computation. In practice, however, it is desirable to allow the live performer more flexibility during the performance. More sophisticated tempo tracking algorithms can be developed using techniques such as linear prediction or neural networks. In addition, a score tracker that tracks pitch as well as time inputs could be incorporated to allow ZED to accompany live performers playing musical instrument controllers such as keyboards and wind instruments. The algorithms of other accompaniment systems described in this thesis could be incorporated by implementing them as the data extraction portion of a patch. In extending the system, however, care must be taken to not sacrifice timeliness for better musical interpretation. (Live performers have the same constraint—the amount of attention paid to each note is determined by when it must be played.) The more sophisticated algorithms for score tracking used by accompaniment systems were designed for systems that only control the tempo of the synthesizer. In those systems, there is only one real-time controller, and the accompaniment scores have a relatively small number of parts. Further xperimentation is required to determine if such algorithms are efficient enough for systems such as ZED that may have a large number of synthesizers and input controllers.

There are also a number of additional features that would make ZED more user friendly. To fully optimize a performance, ZED requires that the user define a library of schedule and play methods that correspond to the set of patches being used. All synthesis parameters that are not affected by real-time inputs can be precomputed and all parameters that are affected at runtime are computed dynamically. ZED's semantics could be extended to use a simple "black box" input/output model for each of the individual parameter computations. Such information could be used by ZED to automatically define schedule and play methods that are optimized for the particular patches used in the performance. The specification of such information, as well as the entire patching mechanism could benefit from a graphical user interface similar to that used by MAX. An integrated computer aided composition application could also be added to make it easier for a user to work on the composition and the performance simultaneously.

The implementation of ZED demonstrated that the object-oriented paradigm is a natural basis for developing real-time and simulation applications. The architecture takes advantage of the inheritance and encapsulation facilities of object-oriented languages to allow the user to easily extend and enhance the system to experiment with new music interpretation algorithms and new synthesis technologies. Objective-C proved to be an ideal language for implementing ZED because it has many essential features of Smalltalk (such as unbounded polymorphy and dynamic message lookup) while also providing access to the efficient C runtime environment. In addition, Objective-C provides static binding for additional runtime optimization.

The evolution of workstation technology has finally reached the point where it can deliver the powerful features of object-oriented languages to software systems that need to run in real-time. As VLSI technology continues to evolve, providing faster and more affordable hardware, the problems in software engineering are becoming less concerned with speed and more concerned with managing the rapidly increasing complexity of software. Thus, new software engineering techniques, in addition to object-oriented techniques, are an increasingly viable alternative to addressing the complexity, while still being able to insure sufficient runtime efficiency.

The software architecture of ZED provides a level of extensibility that makes it an ideal platform for the next generation of computer music performance applications. Furthermore, ZED uses a general software architecture that could be applied to other real-time control applications and simulation systems.

Appendix A

Real-Time Scheduling

The scheduling mechanism is the most critical component in simulation and real-time systems. In ZED, the conductor object uses a scheduler to coordinate the performance and synchronize events. A discussion of real-time scheduling for real-time music performance was presented by Dannenberg [Dannenberg, 1989]. This appendix presents a brief summary of this work as a basis for discussing the design of ZED's scheduler. ZED's scheduler is based on a hybrid approach that maximizes both runtime efficiency and flexibility.

Overview of Selected Scheduling Algorithms

A simple algorithm can be defined to schedule events based on the time that they are to take place. The events are put into a *requests queue* via a *schedule* function. The requests queue is actually a *priority queue* that is ordered by the event's time. The priority queue can be implemented with an efficient ordered data structure such as a linked list or a heap. A linked list allows the cost of scheduling events to be proportional to *n* (the number of items in the list) whereas a heap allows the cost to be proportional to *log n.*

An *alarm* function is defined that is called on each clock tick. The alarm function searches the requests queue for events whose time is less than or equal to the current time provided by a *system clock.* Dannenberg introduced a refinement called "Implementation 4" that invokes the alarm function only when there is an event to be executed, rather than on every clock tick. Another optimization is introduced that addresses the overhead in the priority queue by changing the data structure of the request queue from a dynamic data structure (a heap) to a static data structure (an array). At a time resolution of several milliseconds, a table of requests can be maintained as an array with one array element for each unit of time. The table can hold events that are to occur several seconds in the future. This scheme allows an element to be inserted very fast because the time of the event can be used to directly compute the array index holding the list of requests for that time.

The problem with this approach is that events that are farther in the future than the length of the array cannot be scheduled. Dannenberg addressed this problem by introducing a fall back strategy for long term requests. The basic idea is to put all requests that cannot be immediately entered into the table into a simple linked list called *pending*. A background process uses idle processor time to remove items from the pending list and insert them into a priority queue, and also moves events from the priority queue and inserts them into the table when the event's time is within the time span of the table. This algorithm is referred to as "Implementation 6."

Virtual Time

A shortcoming of the above algorithms is that the scheduler uses physical time for scheduling events, thus preventing the *pace* or *tempo* of the events from being changed during a performance. Controlling the tempo is critical for real-time music performance because it allows the synthesized performance to be synchronized with a live musician who interactively changes the tempo. Because all of the scheduling algorithms described above (except "Implementation 4") send an alarm on every clock tick, it is possible to introduce *virtual time* by re-mapping a hardware interrupt occurring at some fast, fixed interval to call the alarm at a slower, variable rate. This approach doesn't work if there are several time references. A separate scheduler for each time reference could be operated if the number of schedulers is small. If the number of schedulers is large this would be potentially very expensive. An improvement can be made by restricting the changes to the speed (or tempo) to allow changes only with some small advance notice. Each virtual time can be mapped to a physical time shortly before the physical time occurs. Dannenberg proposed a modification to "Implementation 6" can be made whereby the background process converts an event's virtual time to a real time when it is put in the table. Once an event is in the table its real time cannot be changed. The worst case advance notice of a tempo change is the table size. Dannenberg presents a final example that removes this latency in changing the tempo by modifying "Implementation 6" to only call the alarm function when it is time for the next event, as in "Implementation 4."[1]

Discussion of Scheduling Algorithms

The scheduling algorithms described above are completely general in that events can be scheduled in any order, for any time in the future. The focus of Dannenberg's optimizations was to minimize the time it takes to schedule a new event. Dannenberg states that in real-time computer music systems, "frequent operations that consume only a small amount of processing time are not as problematic as less frequent operations that involve significant computation." He uses this reasoning to conclude that the savings of calling the alarm function only when the next event is ready to be played ("Implementation 4") is minimal. This assertion may

1. Dannenberg notes that the resulting algorithm has unnecessary calls to a virtual alarm function but this is not perceived as a problem because the real-time scheduler is so efficient.

be true when implemented on a single tasking operating system and dedicated hardware that is devoted to scheduling. In multi-tasking operating systems, however, there may be serious consequences to performing even the smallest task on every clock tick. It is desirable to implement music performance systems on modern workstations that provide a multi-tasking operating system such as UNIX™ [Bell Labs, 1982; Bach, 1986] and object-oriented programming languages.[2] Dannenberg's approach must therefore be questioned in the context of a multi-tasking environment.

Dannenberg's final optimization requires a background process. In a multi-tasking environment, the operating system is responsible for scheduling and running the processes. Dannenberg did not discuss the properties of scheduling this background process or its overhead and impact on the performance of the primary process. It may be difficult (or impossible) to insure that the scheduler's background process is run just the right amount of time to do its job. If the priority of the background process is set too low, it may not run enough, causing events to be late or missed. If the background process priority is set too high, it could run too often, affecting timeliness in real-time input and output event handling. The size of the table used by the algorithm can be adjusted at the cost of memory. It is, however, difficult to analyze the dynamic behavior of a music performance to determine the "right" size for the table and the "right" priority for the background process. It is particularly difficult because the operating system has its own highly sophisticated and complex process scheduler. Furthermore, finding adequate settings empirically for a given composition and performance may not be sufficient for different compositions with different performance dynamics.

Although in principle the background process uses "idle processor time," in reality, it competes with the primary real-time performance process. The background process uses CPU cycles not only while it is executing, but it also introduces management overhead for such tasks as context switching and possibly even swapping. The effects of the background process on the timeliness and responsiveness of the performance could be noticeable. Because of this, a less costly approach is required that provides adequate timeliness and determinism in a multi-tasking environment.

Real-time Scheduling for Music Systems

Dannenberg's algorithms focused on optimizing the average performance of a general scheduler. The generality of the scheduler described by Dannenberg is not required for real-time music performance. The scheduler proposed in the following sections is designed to optimize *the most likely case in music performance.* The approach is based on an analysis of the most likely scenarios in real-time music performance and optimizing those to run very efficiently, at the expense of having the highly unlikely scenarios run slightly less efficiently.

2. Such workstations are readily available and provide the basis for computer music systems with multiple synthesis and controller devices, and software development tools for building complex systems.

Static and Dynamic Components

A score file is in some sense a computer program that contains a set of instructions that, when carried out in order, results in the performance of a musical composition. Computer programming languages fall into two broad categories: *compiled* and *interpreted.* Compiled programming languages translate a source program into an object program that is loaded into memory and executed. Interpreted languages (such as APL [Gilman and Rose, 1976], SnoBol [Griswold et al., 1971], and LISP [Winston and Horn, 1981]) transform a source program into a simplified language (sometimes called *intermediate code*) that is directly executed dynamically at runtime via an *interpreter.* In ZED, MUSE is, in some sense, an intermediate code for source programs defined in score files.

Compilers bind the attributes or properties of each program variable *statically* at *compile time.* Conversely, interpreters allow *dynamic binding* at *runtime* (while the program is running). Static binding has two primary advantages: *type safety*—type checking at compile time—and more efficient execution. The efficiency comes from the fact that the data needn't be examined at runtime to determine the appropriate operation. Although dynamic binding may be less efficient, it is much more convenient for the programmer and provides greater runtime flexibility. Interpreters that utilize dynamic binding can facilitate the implementation of more complex programming language constructs [Aho and Ullman, 1977].

Real-time systems in general have a *static* component and a *dynamic* component. The static portion of the system is inflexible but has low runtime overhead, whereas the dynamic portion of the system has greater runtime overhead but provides more flexibility at runtime. A real-time scheduler is proposed in the following sections that takes advantage of this partition to achieve a balance of minimum runtime overhead and maximum flexibility. The static component consists of the information that can be statically scheduled before the performance, that is, events whose *abstract time* (in units of beats rather than physical time) has been determined. The dynamic component consists of information that is created during the performance to occur in the future and therefore must be scheduled at runtime. (Real-time inputs that cause an action to take place immediately do not need to be scheduled.)

Static Scheduling

An important property of music performance systems without interactive control is that *no events need to be scheduled during the performance.* The events in the score can be *statically compiled* into explicit synthesis parameters and scheduled before the performance begins. This is because if there is no interactive control, the physical time of all events is known before the performance begins. The amount of time it takes the scheduler to insert an event isn't critical because it does not take place during the performance. A simple queue implemented as a linked list and sorted by time serves as an adequate scheduler. A diagram of the objects involved in such a scheduler are shown in *FIGURE A.1.* The class definitions for the objects used by the scheduler are shown below.

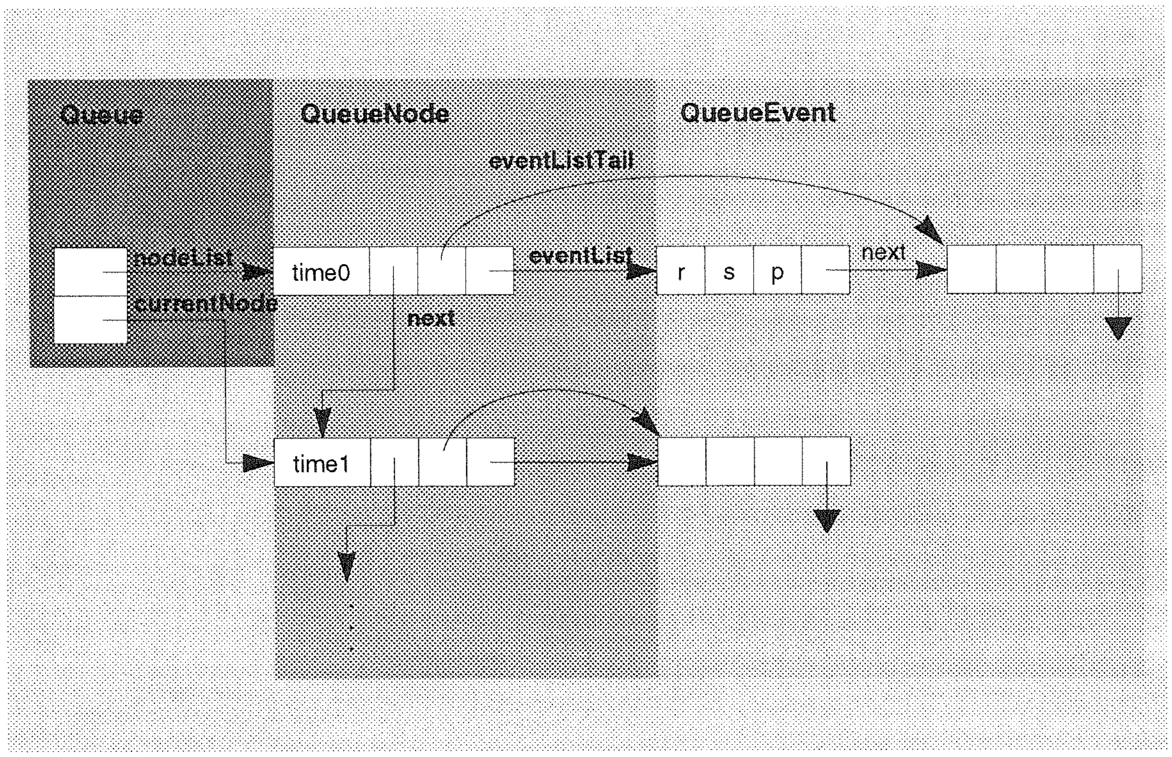


FIGURE A.1 Data structures for a simple scheduler.
Queues have a node list consisting of a linked list of queue nodes, ordered by their time. The instance variable currentNode indicates the current time in the performance. Each queue node has a pointer to the beginning and the end of a list of events that are executed at the node's time. When events are added to a node's event list, they are added at the end to insure a stable sort.

```
Object ( )
    Queue (nodeList currentNode)
    QueueNode (time eventList next eventListTail)
    QueueEvent (receiver selector parameter next)
```

The queue implements an insertion method that uses the standard linked list insert algorithm. Each new event is added to end of the event list of the node with the event's time. This insures a stable sort.[3] If a node does not exist, one is created. If there is no real-time control, the event's time is converted to physical time when the event is scheduled. The physical time is computed by multiplying the abstract time by a *beat length* (indicated by the *metronome marking* of the score's tempo).

This static scheduler has virtually no runtime overhead. The next node in the performance can be returned from the queue a method called next. This method returns the current node and increments the current node pointer to the next node.

3. Although events at a given time theoretically occur simultaneously, in reality the workstation's processor cannot actually execute the events at the same time. Therefore, it is important that a stable sort is maintained. An example of when this matters is if a vibrato event is to be applied at the same time as a new note is played, if the vibrato event is executed before the new note is initiated, it may effect the previous note on the synthesizer.

Interactive Control of Tempo

The benefits of static scheduling can be exploited even with the introduction of interactive control of the tempo. The events can still be scheduled before the performance. Each event has an abstract time expressed relative to a tempo. Because the tempo changes during the performance, the physical time of each event cannot be computed until runtime. The only change to the static scheduler that is required to accommodate interactive control of tempo is that events are scheduled based on their abstract time. The physical time is computed at the last possible moment, namely when the previous event is executed. This allows the system to be maximally responsive to tempo changes during the performance because whenever a real-time input changes the tempo, the next event to be scheduled reflects the new tempo. The multiplication operation required to compute the physical time of an event is simply moved from schedule time to runtime. Thus, the ability to control the tempo of a score in real-time has a runtime cost of one multiplication operation per queue node, <u>a very small price indeed</u>! (This is less than or equal to the number of events in the queue and depends on how may of the events are simultaneous with others.)

Dynamic Scheduling

Real-time music performances under interactive control can also have the scheduling efficiency of static performances *if the abstract time of all events is known before the performance begins.* The static scheduler does not support real-time performances that include events whose time is determined during the performance. The most common patch that dynamically binds an event's time is one that controls the articulation of notes with real-time inputs. In this example, the "note on" event that initiates a note can be scheduled before the performance but the "note off" event cannot be because the duty cycle of the note is not known. Another case that the static scheduler does not handle is when a new event is created during the performance that must be scheduled in the future. For example, a patch can be defined in ZED that creates an arpeggio—several harmonically related notes spaced over time—each time a MIDI "note on" event is received during the performance. This type of patch requires that several events be dynamically scheduled for each input of the type specified by the patch's filter.

Dynamically scheduled events in real-time performance have several interesting properties. Because dynamic events are scheduled on the fly, events are inserted starting from the current node in the queue. (If an event is not in the future, it is ignored.) In the case of dynamically scheduled "note off" events, the events are generated one-to-one with "note on" events. Therefore, the maximum number of "note off" events that are pending at a given time (i.e., the length of the queue) is equal to the number of independent notes that can be played at a given time on the synthesis hardware. Even for sophisticated synthesis environments this rarely exceeds a hundred and is typically a few dozen.

The number of dynamic events that are practical to generate from a static event or interactive control input is limited in several ways. The synthesis hardware has a finite capacity, thus limiting the number of events that can be played at a given time. The speed of the synthesis device interface limits how fast events can be sent and received. In addition, there is a practical limit to what a human audience can comprehend and what is aesthetically pleasing. The technology is fast enough such that the number of simultaneous notes and the rate at that they are played is perhaps limited more by what makes sense musically than by the speed of the hardware and software.

Rather than inserting these dynamic events into the large static queue, a second *dynamic* queue is introduced. The dynamic queue is an instance of the queue class defined for static queues. The dynamic queue has the unique property that its length (the number of pending events) is very small, rarely exceeding a few thousand elements and typically less than a hundred elements. The maximum insertion time for a linked list is a small constant multiplied by the length of the list, and the average time is half that. Therefore, the insertion operation for linked lists with a very small number of elements is extremely fast.

Merging Static and Dynamic Queues

Before the performance begins, the static queue contains all of the events specified in the score file. The dynamic queue is initially empty. The dynamic queue grows and shrinks as elements are inserted and executed dynamically during the performance. The events in the two queues are *merged* at runtime. A *merge queue* is defined that is based on a *merge sort* algorithm. Merge sorts are very efficient [Knuth, 1973] as they simply compare the next elements of each of the queues and return the one with the earliest time. The class **TwoWayMergeQueue** is defined and the method next is implemented to perform this comparison. If the next node of the static queue has the same time as the next node of the dynamic queue, the event list for the dynamic queue's node is appended to then end of the static queue's node and the node from the static queue is returned. The class definition for a two way merge queue are shown below.

```
Object ( )
    TwoWayMergeQueue (staticQueue dynamicQueue)
```

Multiple Time References

Recall that each queue has nodes with times expressed in beats and the corresponding physical time is computed when the previous event is executed. This scheme requires that the events in both the static and dynamic queues have the same tempo or *time reference*. Multiple time references are required when there are two or more tempi under independent interactive control.

In most cases the number of independently controlled tempi is very small, generally one or two, and rarely more than ten. Because this number is so small and because the merge queue is easily extensible, multiple tempi can be handled with a trivial modification to the two way merge queue to handle an arbitrary number of queues at different tempi. The class definition for a scheduler that implements a merge queue for an arbitrary list of queues is shown below. The instance variable queueList is a linked list of queues. The instance variable **currentTime** is used to cache the physical time of the current pending alarm. Also shown is a modification to the class **Queue** that provides each queue object with an instance variable **tempo** to hold the time reference. The class **Queue** has also been modified to include an instance variable **nextQueue** for creating a linked list of queues.

```
Object ( )
    Queue (nodeList currentNode nextQueue tempo)
    ZEDScheduler (queueList currentTime)
```

The class **ZEDScheduler** implements the method **next** to look through the queue list for the next node. Thus, N queues increase the number of comparisons required to find the next node to $N - 1$. Also, $N - 1$ floating point multiplications are required because in order to find the next node the times for the current node in each queue must be converted to physical time based on the node's tempo.

There is, however, a slight problem with responsiveness in this design. After the physical time for the next event is computed, it cannot be affected by tempo changes. Only subsequent events are affected. This problem can be solved by having the conductor notify the scheduler when a tempo has changed. The scheduler then cancels the pending event and recomputes the next node. This calculation is relatively fast and would only cause efficiency problems if the tempo were changing many times per second, but this is rarely the case in most performances. (Live musicians could hardly keep up with such changes either!)

Optimizations

The dynamic queue, because of its small size, does not benefit by introducing complexity to make insertion more efficient. Data structures such as heaps and indexes introduce overhead each time a node is inserted or removed, and therefore do not result in significant gains. A slight performance gain that can be trivially implemented is a pointer cache. Because dynamic events are often inserted in time order (i.e., a dynamic event often has a time that is greater than the previously inserted dynamic event), remembering where the last event was inserted can sometimes reduce the insertion time at the cost of a single comparison operation. Another slight improvement could be achieved by using a doubly linked list, allowing both forward and backward searching from the cached pointer. This optimization introduces a slight maintenance overhead on insertion and deletion.

Memory Management

A hidden cost in the implementation of dynamic schedulers is the cost of allocating memory. In addition, under virtual memory systems such as UNIX™, there could also be a performance degradation due to paging as a result of the addressing space growing larger than physical memory. As mentioned earlier, the dynamic queue can only schedule events in the future. Once a node's events are executed, the node and events are no longer needed. Using the standard technique of a free pool [Knuth, 1968] a *recycling queue* can be implemented that returns the nodes and events to free pools. When new nodes and events are needed they are removed from their respective pools rather than created with memory allocations. To minimize the number of runtime memory allocations required at the beginning of the performance (due to the free pools being empty), the free pools are initialized before the performance to a thousand or so elements (which is greater than the typical length of the queue). If the dynamic queue does not exceed this size at any time, then the amount of memory required by the performance is fixed and there are no additional timing delays due to runtime memory allocation.

Another optimization that can reduce runtime memory allocations is pre-allocation of objects that have their time bound at runtime. For example, if a particular performer object requires "note off" packets to be sent for each note in the part, a packet object for the "note off" event can be created before the performance so that is it ready to be scheduled when the time of the event is known.[4]

Triggering Sequences in Real-time

Often it is desirable in a real-time performance to start playing a *sequence* of notes dynamically as a result of a real-time input. This is called *triggering*. Special consideration must be given to triggering sequences in real-time because they may require a large number of events to be scheduled dynamically, thus violating the assumption that the dynamic queue remains small.

To handle sequences that are triggered, the class **QueueWithOffset** is defined as a subclass of the class **Queue**. The class hierarchy is shown below.

```
Object ( )
    Queue (nodeList currentNode nextQueue tempo)
        QueueWithOffset (offset)
```

Before the performance begins each sequence that may be triggered during the performance is scheduled into its own static queue using event times that are relative to the beginning of the sequence. When the sequence is triggered, the current beat number in the performance is captured in an instance variable called offset. The offset is used to instantiate the sequence at the current time. The queue is then linked into the queue list of the merge queue. The class **QueueWithOffset** is defined as a subclass of the class **Queue**. **QueueWithOffset** overrides the method next to add the offset to the node's time before multiplying by the beat length. Sequences can be stopped under interactive control by sending a message that removes the sequence's queue from the merge queue's queue list.

Repeated Sequences

The principle of a queue with an offset can also be used to save memory on sequences that are repeated. Such

4. There is empirical evidence that these techniques have a noticeable effect on timeliness, although no detailed timing experiments were done.

sequences are very common in most types of music. The class **RepeatedQueue** is defined as a subclass of **QueueWithOffset** with three additional instance variables: numberOfRepeats, counter, and queueLength. The class hierarchy is shown below.

```
Object ( )
    Queue (nodeList currentNode nextQueue tempo)
        QueueWithOffset (offset)
            RepeatedQueue (numberOfRepeats counter queueLength)
```

The numberOfRepeats is the total number of times the sequence is to be played. The counter reflects how many times the sequence has already been played. The queueLength is the number of beats in the sequence. The counter is initialized to zero before the performance begins. When the end of the queue is reached the counter is incremented and compared to the number of repeats. If it is greater than the number of the repeats, the queue is removed from the queue list. If it is not, the offset is incremented by the length of the queue.

Summary

Typically, real-time systems that are completely static are not flexible and do not support interactive control. On the other hand, totally dynamic systems are very flexible but have inherent problems insuring timeliness. The need to heavily optimize runtime scheduling algorithms is alleviated by considering the application domain and partitioning the events into those whose time is statically bound, and those whose time is dynamically bound. The result is not a static scheduler nor a dynamic scheduler, but a *hybrid* scheduler that has all the properties of a completely dynamic scheduler, and most of the efficiency benefits of a static scheduler. Experiments were done that indicate that using free pools for dynamically scheduled objects had a noticeable affect on the efficiency of the scheduler by eliminating runtime memory allocations, eliminating the need to rely on the operating system to free memory, and preventing garbage from accumulating, causing the process size to grow during the performance.

References

Aho, A. V. and Ullman, J. D. *Principles of Compiler Design.* Reading, MA: Addison-Wesley, 1977.

Bach, M. J. *The Design of the UNIX ™ Operating System.* Englewood, NJ: Prentice-Hall, 1986.

Bell Telephone Laboratories, Inc. *UNIX™ Time-Sharing System: UNIX™ Programmer's Manual.* New York, NY: Holt, Rinehart and Winston, 1982.

Dannenberg, R. Real-Time Scheduling and Computer Accompaniment. In Mathews, M. V. and Pierce, J. R. (eds), *Current Directions in Computer Music Research.* Cambridge, MA: MIT Press, 1989, pp. 223-261.

Gilman, L., and Rose, A. J. *APL: An Interactive Approach.* New York, NY: John Wiley & Sons, 1971.

Griswold, R. E., Poage, J. F., and Polonsky, I. P. *The SnoBol 4 Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1971.

Knuth, D. E. *The Art of Computer Programming Volume 1: Fundamental Algorithms.* Reading, MA: Addison-Wesley, 1968, pp. 253-154.

Knuth, D. E. *The Art of Computer Programming Volume 3: Sorting and Searching.* Reading, MA: Addison-Wesley, 1973, p. 160.

Winston, P. H. and Horn, B. K. P. *Lisp.* Reading, MA: Addison-Wesley, 1981.

Appendix B

Score Files

This appendix provides a brief description of the function of score readers. The MUSE score file representation provides the ability to read and write MUSE score objects to and from ASCII files. Then, the mapping of MIDI score files to MUSE objects is described. Finally, a brief discussion is presented outlining how MUSE scores can be defined from multiple score files, and how any of the score files can be converted to MUSE files.

Score Readers

Score files are read by a score reader object. A score reader is implemented for each score file format. The score reader reads the score file and converts the score file data into MUSE object networks in program memory. The score reader class hierarchy is shown below.

```
Object ( )
    ScoreReader (fileName fileHandle)
        MUSEScoreReader ( )
        MIDIScoreReader ( )
```

The class **ScoreReader** is an abstract superclass and the classes **MUSEScoreReader** and **MIDIScoreReader** are concrete subclasses. Each of the concrete classes implements the message readScoreNamed:, which returns a MUSE score object that is an instance of the class **Score**. Other score file formats can be added (such as NeXT score files [Jaffe, 1989]) by subclassing **ScoreReader** and implementing methods that parse the file and transform it into MUSE objects.

Many score files such as MIDI and NeXT are based on note lists and typically consist of device specific values. These are mapped to MUSE message objects. These other score files also have no interpretation context information. ZED configuration and patch files can be used to augment these score files. When this is the case, the MIDI events are mapped to MUSE symbols relative to the *identity* interpretation context that consists of a default tempo, tonality, meter, style, and dynamics. The score file's notion of time is normalized and mapped into the MUSE notion of a duration in beats and a tempo. Similarly the score file notion of pitch is normalized into the MUSE notion of a tonality, and pitches are expressed relative to the tonality. The volume (if any) in the score file is mapped into the MUSE notion of dynamics expressed in terms of a percentage of the maximum. Once the scores are mapped to MUSE objects with an interpretation context, they can be controlled in real-time in the same way as scores from MUSE score files, even though they originated from different score file formats that may not have had interactive control information originally.

The MUSE score file format is an ASCII representation of MUSE objects that is specifically designed for interactive control and mapping to MUSE objects. MIDI files are used as an example of how a score file format that is very different from MUSE score files can be mapped to MUSE objects.

MUSE Score Files

MUSE score files consist of a score, some number of parts, and an optional cue sheet. Parts can be built from notes, rests, and hierarchically with sequences. Parts may also have interpretation symbols that are local to the sequence or part that contains them. Each of the score, parts, cue sheet and sequences are named. The cue sheet consists of definitions for MUSE cue objects and interpretation context objects that are global to the entire score.

In addition to notes and interpretation symbols, MUSE score files can also contain arbitrary messages that invoke methods in the implementation of the performance objects. Thus, the score file and the ZED system are tightly coupled.

Basic Syntax

The syntax of MUSE score files is based on Smalltalk syntax and instance creation semantics whereby an object is represented by a MUSE class name, a set of instance variable names, and values. The MUSE score file format is a simple Smalltalk program that, when executed in the Smalltalk system, returns a score object with part objects consisting of symbol objects. It is therefore possible to allow arbitrary Smalltalk code to exist in the score file because it is simply a Smalltalk program. The score files presented in the following sections have been simplified to only a few specific constructs. Rather than defining complex code in the score file, simple messages are specified that invoke methods that may be of arbitrary complexity. This approach has two important properties: the file format is easier to parse (and is more portable between programming languages), and the system runs faster. The simplified score file syntax, although it is actual

Smalltalk syntax, is easier to parse and does not require an entire Smalltalk parser. As a simplification, all messages are shown with only one parameter, although this is easily extensible.

Although the MUSE score representation allows objects to be created with any class method that is defined in the system, only a few of the standard and most common creation messages are presented. The score file representation can, however, be extended simply by adding new object creation methods and referencing them in the score file. In this way instrument specific or device specific extensions are easily added. Some complex MUSE objects such as interpretation symbols may also be defined with an initialization selector that represents a message that are sent to the class. The method that implements the message creates a new object and initialize its instance variables.

Values are of four basic types: numbers, symbols, strings, and pitches. Numbers include floats, integers, and fractions (rational numbers). Fractions are specified in Smalltalk as two integers separated by a slash character, such as 1/3 for one-third. Method selectors and object names are represented with Smalltalk *symbols* that consist of a hash character followed by an alphanumeric string, such as #new. Smalltalk method selectors may contain a colon, indicating that the corresponding methods have parameters, one for each colon. Pitches can be specified as an instance of the class **Pitch** or as a number (implying the pitch's offset is 0). As a shorthand, a comma can be used between two numbers to specify a pitch. For example, (0,0) is the pitch middle C. (The method "," defined on the class Number creates a pitch object and sets the step and offset of the pitch to the receiver and the parameter respectively.) MUSE score files may also container values that are arrays. Smalltalk array syntax is used. For example #(1 2 3) is a three element array containing the first three counting numbers. Arrays can be recursive, for example, #(#(1) #(2) #(3)) is an array containing three arrays, each having one number.

The MUSE score file syntax also supports a *delta time* representation. In this representation, the time of a symbol is expressed relative to the previous symbol. For simplicity all examples in the sections below are expressed with an explicit time relative to the beginning of the part or sequence.

Interpretation Symbols

Recall the class definitions for the MUSE objects for interpretation symbols as shown below.

```
Object ( )
    MUSEObject ( )
        InterpretationContext (tempo dynamics meter tonality style)
    MUSESymbol (time)
        InterpretationSymbol (name)
            Tempo (metronomeMarking)
            Dynamics (level)
            Tonality (keyNote tonalSystem)
            Meter (beatsPerMeasure referenceBeat stressSelector)
            Style (articulation)
```

In addition to specifying explicit instance variable values, interpretation symbols can be created by specifying an initialization message that is sent to the object to set the instance variables. (The messages are sent when during the scheduling phase when the score is compiled.) A special keyword initialization: followed by a selector represents a method that computes the instance variable values. The optional keyword with: is followed by a parameter specifying the method's parameter.

Interpretation symbols that appear in a part only apply to that part. Interpretation symbols that appear in the cue sheet are considered to be global and therefore are used by all performer objects in the absence of an interpretation symbol within their own part. All interpretation symbols can optionally have a name. The name is important as it allows the object to be referred to in the score, configuration, and patch files, and is the basis of the real-time patching mechanism.

Tempo

Tempi can be defined in a MUSE score by explicitly setting the metronome marking or by setting an initialization message that computes a metronome marking. The syntax for defining tempo objects in a MUSE score file are shown below. Tempo objects can be defined either with a name or without a name. The time t and metronome marking mm are numbers, the initialization s is a selector, the parameter p is a number, and the name n is a symbol.

Tempo Syntax Without Names
- **Tempo** time: t mm: mm
- **Tempo** time: t initialization: s
- **Tempo** time: t initialization: s with: p

Tempo Syntax With Names
- **Tempo** time: t name: n mm: mm
- **Tempo** time: t name: n initialization: s
- **Tempo** time: t name: n initialization: s with: p

Some examples of tempi definitions are shown below.

Tempo time: 0.0 mm: 120
Tempo time: 32 name: #GlobalTempo initialization: #allegro
Tempo time: 64.0 initialization: #accelerando with: 0.10

Dynamics

The dynamics can be defined by setting the *level* to an explicit value or by specifying an initialization message that computes the dynamics level. The syntax for specifying dynamics is shown below. The level l is a number. The time, initialization, parameter, and name have the same types as those for tempo, namely number, selector, number and symbol respectively.

Dynamics Syntax Without Names

Dynamics time: t level: l
Dynamics time: t initialization: s
Dynamics time: t initialization: s with: n

Dynamics Syntax With Names

Dynamics time: t name: n level: l
Dynamics time: t name: n initialization: s
Dynamics time: t name: n initialization: s with: p

Some examples of dynamics definitions are shown below.

Dynamics time: 0.0 level: 0.25
Dynamics time: 32.5 initialization: #crescendo
Dynamics time: 64.25 name: #QuietPhrase initialization: #piano

Tonality

The tonality is specified by explicitly defining a key note and tonal system, or by using an initialization message. Tonality objects may also be named. The syntax for defining the tonality is shown below.

Tonality Syntax Without Names

Tonality time: t keyNote: p chromaticSize: c naturalScale: ns keyScale: ks
Tonality time: t initialization: s
Tonality time: t initialization: s with: p

Tonality Syntax With Names

Tonality time: t name: n keyNote: p chromaticSize: c naturalScale: ns keyScale: ks
Tonality time: t name: n initialization: s
Tonality time: t name: n initialization: s with: p

The key note p is a pitch that is defined as an interval from the tonal system origin (middle C in twelve-tone music). The chromatic size c is an integer. The natural scale and key scale, ns and ks respectively, are arrays. The time, initialization, parameter, and name are the same types as in the other interpretation symbols. An example of a B♭ minor scale with a tempered tuning is shown below.

Tonality time: 0 initialization: #minorWithTemperedTuning: with: 6.-1

Tonality designations in MIDI files result in tonality objects that have a MIDI tuning and the frequencies set to the tempered scale. Tonality objects created in MUSE files have a tuning that is an instance of **MUSEChromaticTuning** and the frequencies set to the tempered scale. When the tonality is defined with an initialization selector, the corresponding method may create a tuning object and set the frequencies in addition to setting the other instance variables of the tonal system.

Meter

The meter can be specified either by explicitly defining the instance variables beatsPerMeasure, referenceBeat, and stressSelector, or by specifying an initialization message. The syntax for defining meter objects is shown below.

Meter Syntax Without Names
Meter time: t beatsPerMeasure: b referenceBeat: r stressSelector: s
Meter time: t initialization: s
Meter time: t initialization: s with: p

Meter Syntax With Names
Meter time: t name: n beatsPerMeasure: b referenceBeat: r stressSelector: s
Meter time: t name: n initialization: s
Meter time: t name: n initialization: s with: p

The time t, the number of beats per measure b, and the reference beat r are all numbers. (The number of beats per measure is typically an integer and the reference beat is typically a rational number, but this is not required.) The stress selector s is a selector for a message computes the stress value for a given beat, and has one parameter, the current beat. This message can be sent by the performer from the playNote: method. The name n is again a symbol as with tempo and dynamics. Some examples for defining meter symbols are shown below.

Meter time: 0 beatsPerMeasure: 5 referenceBeat: 1/8 stressSelector: #twoThree
Meter time: 32 name: #GlobalMeter initialization: #fourFour
Meter time: 256 initialization: #setMeter: with: 6/8

Style

The style is specified explicitly by defining a default articulation or by specifying an initialization message that computes the default articulation. The syntax for defining style objects is shown below. Style objects, like other interpretation symbol objects, may be named. The time, name, initialization, and parameter are the same as for other interpretation symbols. The articulation object a is the specification of an articulation object.

Style Syntax Without Names
Style time: t articulation: a
Style time: t initialization: s
Style time: t initialization: s with: p

Style Syntax With Names
Style time: t name: n articulation: a
Style time: t name: n initialization: s
Style time: t name: n initialization: s with: p

Some examples of style objects are shown below.

Style time: 0 name: #March articulation: (**Articulation** selector: #marcato)
Style time: 32 initialization: #adagio
Style time: 128 initialization: #dolce: with: 0.9

Discrete Symbols

Recall the class definition for MUSE discrete symbols and their instance variable as shown below.

```
Object ( )
    MUSESymbol (time)
        Cue ( )
        Note (duration pitch articulation)
        Rest (duration)
        Message (receiver selector parameters)
    MUSEObject ( )
        Articulation ( )
            ASDArticulation (attack sustain decay dutyCycle)
            SymbolicArticulation (message)
```

All discrete symbols have a time. Rests can be represented explicitly with a time and duration. (Rests are primarily provided for scores that originated in a notation program that expresses rests explicitly.) When MUSE scores are read, rests are unnecessary because each note has a time and duration. They are therefore essentially omitted from parts. Cues are found only in the cue sheet and have only a time. Notes require a duration, a pitch and optionally have an articulation. The articulation is an articulation object that is created by sending a message to the abstract class **Articulation**. The valid creation messages for discrete symbols in MUSE score files are shown below. (In a score file, each of the expressions is enclosed in parenthesis.)

Cue Syntax
Cue time: t

Rest Syntax
Rest time: t duration: d

Note Syntax
Note time: t duration: d pitch: p
Note time: t duration: d pitch: p articulation: a
Note time: t duration: d pitch: p selector: s
Note time: t duration: d pitch: p selector: s with: n

Articulation Syntax
Articulation selector: s
Articulation selector: s with: n
Articulation attack: a sustain: s decay: d dutyCycle: c

All times t and durations d are numbers and are interpreted to mean a number of beats. All pitches p are either a number or two numbers separated by a comma. All selectors s are symbols and if the symbol has a colon, the selector is followed by the string 'with:' and a parameter of any type. The articulation a is an instance of a concrete subclass of **Articulation** that is created with one of the expressions shown.[1] The methods time:duration:pitch:selector: and time:duration:pitch:selector:with: are provided as shortcuts for creating the articulation object. Thus, the following two symbols are identical:

```
Note time: 1.5 duration: 1/2 pitch: 1,0 articulation: (Articulation selector: #accent)
Note time: 1.5 duration: 1/2 pitch: 1,0 selector: #accent
```

When the articulation of a note is represented as a selector, the selector corresponds to a message that is sent to the performer object causing the performer object to in turn compute the articulation. If the performer object does not understand the message, a warning is printed when the score is read and the message is ignored during the performance. If the articulation is not under interactive control, then the message can be sent when the score is read in and the resulting articulation can be cached. If the articulation is under interactive control, then the message is sent at runtime. When the articulation of a note is specified explicitly, the sustain is an array of arrays specifying the time within the note that the envelope is to be updated and the value of the update.

Some examples of individual symbols in a MUSE score file are shown below.

```
Cue time: 0
Rest time: 0 duration: 1/3
Note time: 1/3 duration: 2/3 pitch: 0
Note time: 2.5 duration: 1/2 pitch: 1,1 articulation: (Articulation selector: #staccato)
Note time: 3 duration: 0.5 pitch: 2 selector: #accent
Note time: 4 duration: 0.25 pitch: 3,0 selector: #tenuto with: 0.8
Note time: 2 duration: 1/2 pitch: 1 articulation: (Articulation selector: #downBow: with: 0.2)
Note time: 2 duration: 1/2 pitch: 1,1 articulation: (
        Articulation
            attack: 0.75
            sustain: #(#(0.25 0.7) #(0.5 0.8) #(0.75 0.9))
            decay: 0.5
            dutyCycle: 0.8)
```

Messages

MUSE message symbols are defined by specifying a time, the name of the receiver, the selector, and an optional parameter. The syntax for defining message objects is shown below.

1. As a memory saving measure, if the data originated from MIDI and if the release velocity is not present or is ignored, the articulation can also be specified as an integer, referring to the velocity.

Message Syntax

Message time: t receiver: n selector: s
Message time: t receiver: n selector: s with: p

The object name n is a symbol representing the receiver's name. The name is used to find the object in the object dictionary. An example that shows how a part can be transposed is shown below.

Tonality time: 0 initialization: #minorWithTemperedTuning: with: 6,-1.
Message time: 128 receiver: #GlobalTonality selector: #transpose: with: 2,0.

An instance of **Message** is created and the message is sent at the time 128, with the receiver being the object with the name #GlobalTonality, the selector #transpose: and parameter 2,0. This will cause the part to be transposed up a minor third.

Instantiation and Dependency

Any named interpretation symbol can be used as a *prototype* for other interpretation symbols of the same type. There are two ways to instantiate prototypes: *independently* and *dependently*.

Independent instantiation is used to clone (copy) a prototype object. Cloned objects reflect changes in the prototype but not interactive control of the prototype. If a selector is specified, the message is sent and computes values based on the prototype's values. This operation is only done once when the score is compiled during the scheduling phase. All information relating the instantiated object to the prototype is disregarded. Therefore, interactive control of the prototype does not effect objects instantiated independently.

The syntax for specifying independent instantiation is shown below. The word **anInterpretationSymbolClass** is used to mean any concrete subclass of **InterpretationSymbol** described in the previous sections, namely **Tempo**, **Dynamics**, **Meter**, **Tonality**, or **Style**.

anInterpretationSymbolClass time: t from: n
anInterpretationSymbolClass time: t from: n selector: s
anInterpretationSymbolClass time: t from: n selector: s with: p

The name n specifies the prototype object. The selector and optional parameter are specified by s and p respectively. Thus, independent instantiation operates the same as a MUSE message expression except that a copy is made and the message is sent to the copy, leaving the original object intact.

Dependent instances are so called because they depend on the value of the prototype. Whenever the prototype object's state changes due to interactive control, the selector of the instantiated object is evaluated causing all objects that are dependent on it to also be under interactive control. This mechanism allows interpretation symbols in different parts to be controlled relative to those in another part or in the global state defined in the cue sheet. For example, one part of the score can be defined to be 10% faster than the global tempo and another part to be 10% slower. If a patch is created to control the global tempo, then the dependent tempo follows. If independent tempo were instantiated, they would initially be relative to the global tempo but would not change over the course of the performance.

The syntax for specifying dependent instantiation is shown below. The values t, n, s, and p are the same as those for independent instantiation.

anInterpretationSymbolClass time: t dependentOn: n
anInterpretationSymbolClass time: t dependentOn: n selector: s
annterpretationSymbolClass time: t dependentOn: n selector: s with: p

Some examples of interpretation symbols instantiated from prototypes are shown below.

Tempo time: 0 name: #GlobalTempo mm: 120.
Tempo time: 0 from: #GlobalTempo selector: #twiceAsFast

Dynamics time: 64.25 name: #QuietPhrase initialization: #piano
Dynamics time: 64.25 dependentOn: #QuietPhrase initialization: #times: with: 1.2

Tonality time: 0 name: #BaseTonality initialization: #pentatonicOnKeyNote: with: 2,-1
Tonality time: 0 dependentOn: #BaseTonality selector: #transposeBy: with: 2,0

In the first example above a tempo named #GlobalTempo is created on beat 0 of the score. The tempo named #GlobalTempo is then instantiated with a metronome marking twice as fast as that of the global tempo (as computed by the method #twiceAsFast) and takes effect on beat 0. The part that contains this tempo definition remains at this tempo (240 beats per minute) until the part specifies a new tempo, regardless of whether or not the global tempo is under interactive control.

The second example shows a dynamics object instantiated dependently. If the dynamics specified by the prototype named #QuietPhrase is under interactive control, the instantiated dynamics are always 120% of that specified in the #QuietPhrase object. If the prototype is not under interactive control, the instantiated dynamics are 120% of the dynamics set in the method named #piano. In either case, the instantiated dynamics object is in effect until the next occurrence of a dynamics object in the same part.

The third example above shows a tonality that is transposed by a third relative to another tonality. If the prototype's key note is under interactive control, the instantiated tonality recomputes its key note to be a third above that of the prototype each time the prototype's key note changes.

Organizing Symbols

Symbols in the MUSE score are organized into a score, parts, a cue sheet, and sequences. The simplest score is one that has a collection of parts that are not hierarchically composed but rather, each part has a simple linear list of symbols. The MUSE classes for hierarchical structures are shown below.

```
Object ( )
    MUSESymbol (time)
        MUSEStructure (name events interpretationContext transformation current)
            Sequence ( )
                Part ( )
                CueSheet ( )
            ParallelSequence ( )
                Score (cueSheet)
            RepeatedSequence (numberOfRepeats count)
```

Scores, Parts, and Cue Sheets

A score is created by specifying a set of parts and an optional cue sheet. The syntax for creating score objects, cue sheet objects, and part objects is shown below. It is assumed that there is only one score in the file and that all parts in the file are in the score. There is at most one cue sheet.

```
Score name: n.
CueSheet name: n events: e.
Part name: n events: e.
```

The name n is a symbol. The events e for a part are specified by enclosing each of the discrete symbols and interpretation symbols in parentheses, and separating them with the back-slash character ('\'). The events for cue sheets consist of a similar collection except that the collection includes only cues and interpretation symbols. All carriage returns, tabs, and spaces are treated as simple delimiters as in Smalltalk. The expressions are terminated with a period (as are all Smalltalk expressions).

An example of the syntax for a complete score is shown below.

```
Score name: #SimpleScore.
CueSheet
    name: #Cues
    events: (
        (Tempo time: 0 mm: 120) \
        (Style time: 0 initialization: #allegroConMoto) \
        (Meter time: 0 initialization: #fourFour) \
        (Tonality time: 0 initialization: #minor: with: 1,0) \
        (Dynamics time: 0 initialization: #forte) \
        (Cue time: 0) \
        (Cue time: 1) \
        (Cue time: 1.75)
        ).
Part
    name: #FlutePart
    events: (
        (Note time: 0.0 duration: 1/2 pitch: 0.0) \
        (Note time: 0.5 duration: 1/2 pitch: 1.0) \
        (Note time: 1.0 duration: 1/2 pitch: 2.0) \
        (Note time: 1.5 duration: 1/2 pitch: 1.0)
        ).
Part
    name: #PianoPart
    events: (
        (Note time: 0.0 duration: 1/2 pitch: 2.0) \
        (Note time: 0.5 duration: 1/2 pitch: 1.0) \
        (Note time: 1.0 duration: 1/2 pitch: 0.0) \
        (Note time: 1.5 duration: 1/2 pitch: 1.0)
        ).
```

The score is named #SimpleScore and has two parts named #FlutePart and #PianoPart and a cue sheet named #Cues. The score is in four-four time with quarter note equal to 120. The style is ***allegro con moto*** and the key is D minor. The global dynamics are ***forte***. There are cues on beats 0, 1, and 1.75. The part #FlutePart has four notes and the part #PianoPart has four notes.

The interpretation context of the individual parts can be set by including interpretation symbols within individual parts. The example below shows the flute part transposed up a third and the piano part slightly quieter. The piano part's tempo is relative to the global tempo, therefore it is always 105% as fast as the global tempo.

```
Score name: #SimpleScore.
CueSheet
    name: #Cues
    events: (
        (Tempo time: 0 name: #GlobalTempo mm: 120) \
        (Style time: 0 name: #GlobalStyle initialization: #allegroConMoto) \
        (Meter time: 0 name: #GlobalMeter initialization: #fourFour) \
        (Tonality time: 0 name: #GlobalTonality initialization: #minor: with: 1.0) \
        (Dynamics time: 0 name: #GlobalDynamics initialization: #forte) \
        (Cue time: 0) \
        (Cue time: 1) \
        (Cue time: 1.75)
        ).
Part
    name: #FlutePart
    events: (
        (Tonality time: 0 from: #GlobalTonality selector: #transposeBy: with: 2.0) \
        (Note time: 0.0 duration: 1/2 pitch: 0.0) \
        (Note time: 0.5 duration: 1/2 pitch: 1.0) \
            (Note time: 1.0 duration: 1/2 pitch: 2.0) \
            (Note time: 1.5 duration: 1/2 pitch: 1.0)
            ).
    Part
        name: #PianoPart
        events: (
            (Tempo time: 0 dependentOn: #GlobalTempo selector: #times: with: 1.05) \
            (Dynamics time: 0 initialization: #mezzoForte) \
            (Note time: 0.0 duration: 1/2 pitch: 2.0) \
            (Note time: 0.5 duration: 1/2 pitch: 1.0) \
            (Note time: 1.0 duration: 1/2 pitch: 0.0) \
            (Note time: 1.5 duration: 1/2 pitch: 1.0)
            ).
```

Sequences

MUSE score files allow MUSE sequence objects to be specified in the event list for a part or sequence. To accomplish this, *template sequences* are defined and then *instantiated* in various parts and at various times in the score.

Sequences are defined in the same way as parts and cue sheets as shown with expression below.

```
    Sequence name: n events: e.
```

The symbols in the sequence definition contain times relative to the beginning of the sequence. The sequence can then be instantiated at any point in the score and the times of the symbols are offset by the time of the instantiated sequence. As with other events, the time is specified explicitly. Sequences are instantiated by specifying the class **Sequence**, a time, the keyword play: and the name of the sequence that is to be instantiated. Sequences can also be instantiated with some number of repeats (resulting in an instance of the class **RepeatedSequence**). Sequences are referenced and instantiated by their name. The two ways of instantiating sequences are shown below.

```
    Sequence time: t play: n
    Sequence time: t play: n repeat: r
```

Sequences can reference other sequences. Reference patterns should be acyclic, that is, a sequence should not reference sequences that directly or indirectly reference it as this results in infinite recursion. (Few audiences have the patience to listen to an infinitely long performance!) This is not a requirement, however, as a patch could be created that would cause the sequence to stop playing when a specific input is received. The sequences can be defined in any order in the score file, that is, *forward references* are allowed. Sequences are not actually instantiated until the score is compiled during the scheduling phase after the entire score has been read in. An example of a score that instantiates sequences is shown below.

```
Score name: #AnotherSimpleScore.
Sequence
    name: #Ascent
    events: (
        (Note time: 0.0 duration: 1/2 pitch: 0.0) \
        (Note time: 0.5 duration: 1/2 pitch: 1.0) \
        (Note time: 1.0 duration: 1/2 pitch: 2.0) \
        (Note time: 1.5 duration: 1/2 pitch: 3.0) \
        (Note time: 2.0 duration: 1/2 pitch: 4.0) \
        (Note time: 2.5 duration: 1/2 pitch: 5.0) \
        (Note time: 3.0 duration: 1/2 pitch: 6.0) \
        (Note time: 3.5 duration: 1/2 pitch: 7.0)
        ).
Sequence name: #Descent
    events: (
        (Note time: 0.0 duration: 1/2 pitch: 7.0) \
        (Note time: 0.5 duration: 1/2 pitch: 6.0) \
        (Note time: 1.0 duration: 1/2 pitch: 5.0) \
        (Note time: 1.5 duration: 1/2 pitch: 4.0) \
        (Note time: 2.0 duration: 1/2 pitch: 3.0) \
        (Note time: 2.5 duration: 1/2 pitch: 2.0) \
        (Note time: 3.0 duration: 1/2 pitch: 1.0) \
        (Note time: 3.5 duration: 1/2 pitch: 0.0) \
        ).

CueSheet
    name: #SomeCues
    events: (
        (Tempo time: 0 mm: 120) \
        (Style time: 0 initialization: #allegroConMoto) \
        (Meter time: 0 initialization: #fourFour) \
        (Tonality time: 0 name: #GlobalTonality initialization: #minor: with: 1.0) \
        (Dynamics time: 0 initialization: #forte)
        ).
Part
    name: #FlutePart
    events: (
        (Tonality time: 0 instantiate: #GlobalTonality selector: #transposeBy: with: 2.0) \
        (Sequence time: 0 instantiate: #Ascent)).
Part
    name: #PianoPart
    events: (
        (Tempo time: 0 mm: 132) \
        (Dynamics time: 0 initialization: #piano) \
        (Sequence time: 2.0 play: #Descent)).
```

In the above example there are two sequences, #Ascent and #Descent. The sequence #Ascent is referenced in the part #FlutePart and is transposed. The sequence #Descent is referenced in the #PianoPart and begins two beats from the beginning of the composition.

An example of using repeated sequences is shown below. Each of the sequences #Ascent and #Descent are repeated for a total of two iterations.

```
Score name: #YetAnotherSimpleScore.

Part
    name: #Up
    events: (
        (Tempo time: 0 mm: 132) \
        (Sequence time: 0 play: #Ascent repeat: 2)).
Part
    name: #Down
    events: (
        Sequence time: 2 play: #Descent repeat: 2).
```

Sequences can be constructed hierarchically as well. For example, a new sequence could be constructed from the two sequences #Ascent and #Descent (described above), and then that sequence can be instantiated. The example below shows a sequence #UpThenDown that is created with the sequence #Ascent followed by the sequence #Descent, and a sequence #DownThenUp consisting of the sequence #Descent followed by #Ascent.

```
Score name: #Scales.

Sequence
    name: #UpThenDown
    events: (
        (Sequence time: 0 play: #Ascent) \
        (Sequence time: 4 play: #Descent)).
Sequence
    name: #DownThenUp
    events: (
        (Sequence time: 0 play: #Descent)
        (Sequence time: 4 play: #Ascent)).

Part
    name: #Part1
    events: (Sequence time: 0 play: #UpThenDown repeat: 2).
Part
    name: #Part2
    events: (Sequence time: 0 play: #DownTheUp repeat: 2).
```

Parallel Sequences

Parallel sequences are defined *implicitly* in MUSE score files by simply specifying two objects (including sequences) within the same part with same time. Whenever the score reader encounters this situation, an instance of **ParallelSequence** is created that has as its event list all events with the same time. For example, the sequence #UpThenDown above can be changed to play the sequences #Ascent and #Descent simultaneously rather than serially by setting their time to be the same.

```
Sequence
    name: #UpAndDownTogether
    events: (
        (Sequence time: 0 play: #Ascent) \
        (Sequence time: 0 play: #Descent)).
```

MIDI Files

MIDI files [MMA, 1987] are binary encoded files that have "events" and "meta-events." MIDI events correspond to notes and messages, and meta-events correspond to interpretation symbols. MIDI events are either *channel messages* or *system messages.* Channel messages refer to a particular MIDI channel or voice. System messages refer to an entire MIDI device. The MIDI channel and system messages are shown in *TABLE D.1* and *TABLE D.2* in *Appendix D.*

MIDI files treat "note on" and "note off" events as separate events whereas MUSE notes are single symbols with a time and a duration. All MIDI events associated with a given channel number are a MUSE part. MIDI files are not sorted by channel but are merged into a single time ordered list. Therefore, an important function of the MIDI score reader is to sort the events into parts and to associate MIDI "note on" events with their corresponding "note off" events. MIDI files do not support the naming of parts, so each part object from a MIDI file has a default name of #Part*n* where *n* is the channel number, e.g., #Part1 for channel one, #Part2 for channel two, etc. MIDI files do not support rests or hierarchical construction so the parts are made up of simple lists of note and message objects.

All MIDI files may express a time signature and tempo in the header of the file. If these do not appear in the MIDI file, the default is 4/4 time at quarter note equals 120 beats per minute (the MIDI default).

Time Mapping

MIDI files support two notions of time: metrical time and time code based time. Each of these representations is mapped to the MUSE representation of a time and a duration in beats relative to a tempo. MIDI files represent event times as *delta time*. For a given event, the delta time is the amount of time after the previous event takes place that the event is to take place. A MUSE note event is constructed by pairing each MIDI "note on" event with its corresponding "note off" event and computing the time and duration.

The time in MIDI time units of the *i*th MIDI event can be expressed as a function of the delta time and the accumulation of the delta times of all of the events preceding it as shown in the equation below.

$$midiTime_i = midiTime_{i-1} + deltaTime_i$$

where $midiTime_0 = 0$ and $deltaTime_i$ is the time in the MIDI file for the *i*th MIDI event. The duration of a note is computed as the time difference between the time of a "note on" event and its corresponding "note off" event.

For MIDI files that use the metrical format for time, the delta time is the number of "ticks" that make up a quarter note. This value is expressed as the "division" in the header of the MIDI file. For example, if the division is 96 then an eighth note (one half beat) is represented as 48. The time and duration for a MUSE note in beats for the *i*th "note on" event and its corresponding "note off" event can be expressed as shown below.

$$time = \frac{midiTime_{noteOn_i}}{division} \times \frac{\frac{1}{4}}{referenceNote}$$

$$duration = \frac{midiTime_{noteOn_i} - midiTime_{noteOff_i}}{division} \times \frac{\frac{1}{4}}{referenceNote}$$

The *referenceNote* is the note specified in the meter's time signature (e.g., 2/4 time is a quarter note or 1/4, 3/8 time is an eighth note or 1/8, etc.) The *time* and *duration* are in MUSE abstract units of beats relative to the tempo and are the instance variables of the note object.

For MIDI files that use time codes instead of metrical time, the delta time is expressed as a fraction of a second. In this case the duration and time are mapped in terms of the default MUSE tempo of quarter note equals 120 beats per minute, resulting in each beat being 0.5 seconds in length. The values *framesPerSecond* and *unitsPerFrame* are specified in the header of the MIDI file. The calculation is shown below.

$$midiTime_i = midiTime_{i-1} + \frac{deltaTime_i}{framesPerSecond \times unitsPerFrame}$$

Other score file formats may not have an explicit notion of tempo and time signature and may instead express an event's time in seconds. In these cases the tempo and time signature are defaulted to 4/4 time with a metronome marking of 120 beats per minute. The time and duration for the MUSE events in units of beats can be computed from events expressed in units of seconds by dividing the time by the beat length, namely 0.5 seconds.

Mapping MIDI Pitch to MUSE Pitch

In MIDI files the pitch is represented as a key number. The key number is an integer between 0 and 127 inclusive and is mapped to a MUSE pitch based on the key signature.

The MIDI meta-event for “key signature” is mapped to a MUSE tonality object. If no key signature is specified in the MIDI file, the key of C major is used. The MIDI value “key number” that is found in “note on” and “note off” events corresponds to the MUSE pitch. The MIDI key number 60 corresponds to middle C. The key number representation does not preserve enharmonic pitches, for example C♯ and D♭ above middle C both have the key number 61. There are two ways that the key number can be mapped when the score is read in. The first is to ignore the key signature specified in the score file and use a tonal system that describes MIDI, namely, a chromatic scale size of 128, and natural and key scales of 0. . .127. Under this scheme each pitch would simply be the key number (with no offset) and the tuning would be a MIDI tuning object. The pitch bend events in the MIDI file could be stored in the MUSE score as message events that are forwarded to the device. If the score has a pitch bend event for each “note on” event, the pitch bend could be stored as the pitch’s offset.

An alternative is to use the key signature to construct a diatonic tonal system and map the key numbers into pitch and offset pairs that best fit the key signature. Mapping to step and offset pairs would allow the pitches to be tuned to non-tempered scales that are a function of the key note and key scale. This fine grained tuning is sometimes desirable for synthesis instruments.

Articulation Mapping

For MIDI events the attack portion of the articulation is the velocity of a “note on” event. The calculation for the MUSE attack is normalized to be the percentage of the maximum attack as shown below.

$$attack = \frac{noteOnVelocity}{127}$$

The decay in the MUSE articulation corresponds to the release velocity of the note, also expressed as the percentage of the maximum. If the release velocity is 0, the score reader uses the MIDI value 64. The calculation of the decay is shown below.[2]

$$decay = \frac{releaseVelocity}{127}$$

If the MIDI file has channel pressure (after touch) events, the score reader constructs the varying sustain by creating a list of pairs with the time of the pressure update and the pressure value. The pressure values are normalized in the same way that the velocity is normalized. The time is computed relative to the “note on” event. The equations below show how the time and value pair are computed for the *j*th after touch event for a “note on” event.

$$sustainTime_j = midiTime_{afterTouch_j} - midiTime_{noteOn}$$

$$sustainValue_j = \frac{channelPressure}{127}$$

2. If the MIDI file does not have explicit “note off” events but rather uses “note on” events with a velocity of 0 for a “note off,” the articulation can be represented as the attack value rather than an articulation object to save memory.

Dynamics Mapping

The MIDI file event for changing the "main volume" corresponds to the dynamics event in the global interpretation context. For each MIDI event that sets the main volume a MUSE dynamics object is created that sets the dynamics to the MIDI value as a percentage of the maximum.

Other Events

There are other events in MIDI files that are device specific in that non-MIDI synthesizers, and even some MIDI synthesizers, do not support the functionality. These are primarily the MIDI events for program changes, synthesis parameter control, and MIDI system messages. Non-MIDI instruments may, however, implement these messages. All channel events from the file that are not mentioned in the previous sections are mapped into a MUSE message event whose receiver is the performer object for the part defined by the channel number. All non-channel events are mapped to MUSE message events that have the conductor object as the receiver. All parameters to the messages listed below are normalized from the MIDI values. The MIDI controllers and their corresponding message are listed in *TABLE D.3* and *TABLE D.4* in *Appendix D*.

Scores with Multiple Score Files

As we have seen in this chapter, generic MUSE scores can be built from a variety of different score file formats. Because all score file formats map to a generic representation, a MUSE composition can be defined with multiple types of score files. This capability is important because it allows MIDI files that were created with other applications to be augmented with a cue sheet and interpretation information from a MUSE file. Also, parts from different score file formats (and different applications) can be merged into a common score for performance.

When MIDI files are read the events are sorted into parts based on the channel number and each part is given a default name of the form #Part*n*, where *n* is the channel number. In a MUSE file a part could be created with the same name that contains no note information but only interpretation information for applying tempo changes, style and meter changes, dynamic changes, etc. A cue sheet with global interpretation information could also be defined. Both scores would be read in and merged into a single set of MUSE score objects. Additionally, the MUSE score file may define additional parts to be merged into the MUSE score.

The generic MUSE score representation enables MUSE scores to be created from any supported score file format. For example, MIDI files can be converted to MUSE files by reading them with the MIDI score reader, and then writing the MUSE objects with a MUSE score writer. Each of n score file formats can be converted to any other format by implementing n score readers and n score writers. Without the generic MUSE object representation, it would require n^2 conversion functions to translate all formats to all others rather than $2n$.

References

Jaffe, D. A. *From the Classical Software Synthesis Note-List to the NeXT Scorefile*. Preliminary Draft, Menlo Park, CA: NeXT Inc., 1989.

MIDI Manufacturers Association, *MIDI Musical Instrument Digital Interface Specification 1.0.* North Hollywood, CA: International MIDI Association, 1987.

Appendix C

Other Approaches

This appendix presents a sampling of other approaches to real-time music performance with interactive control. The approaches are then summarized and compared to ZED.

Other Approaches

Three types of real-time music performance approaches are discussed in the sections that follow: applications primarily concerned with patching and sequencing; applications primarily concerned with performance and accompaniment issues; and tool kits that provide a framework for building applications. All of the systems provide the ability to control music synthesizers in real-time, but they are all based on different metaphors. The patching and sequencing applications allow a user to configure the system via a graphical user interface. The user can specify score to play and can route MIDI inputs from controllers to particular channels of MIDI synthesizers. Performance and accompaniment systems have pre-programmed behavior and only allow the user to specify the score. This type of system accepts input from a live performer and attempts to expressively control and coordinate digital synthesizers with the live performer. Tool kits provide a library of functions that can be used by a programmer to create an application. Several example systems are described in the following sections.

NeXT Music Kit

The NeXT Music Kit [Jaffe, 1989a; Jaffe, 1989b] is a tool kit that is implemented in Objective-C on the NeXT Computer. Although the Music Kit contains some of the same classes as the ZED system, the semantics and underlying software architecture vary greatly. The Music Kit is based on Music V [Mathews, 1969] with added MIDI capability. Music V was a very successful non-real-time computer music program, but its design did not incorporate interactive control or an object-oriented paradigm. Conversely, ZED breaks from the legacy of these systems and is designed to take full advantage of the capabilities of object-oriented programming systems. ZED's design is centered around real-time expressive control and a simple conceptual model based on live orchestra performance. The Music Kit has strong roots in playing score files with real-time control as an addition rather than a foundation. Some of the specific differences between ZED and the Music Kit are outlined in the sections that follow.

Score Files

The score file representation used by the Music Kit merges the MIDI event representation with the classical note list representation used by Music V. Neither the Music Kit, nor these other representations provide abstractions for real-time control, making it difficult to control certain aspects of the performance.

Only one basic event class, called "Note," is provided. The note has "parameters" that are implemented in a Lisp property list style rather than an object-oriented style using instance variables. All information is merged into instances of the same note class. (There is no separate interpretation context.) The result is that objects with very different semantics are represented with one class. A "noteTag" property is used to specify what type of event it is. Although implemented in an object-oriented language, this approach does not use the features provided by object-oriented languages, such as data abstraction, encapsulation, and inheritance.

Performance Objects

In the Music Kit, performer objects acquire generic note objects and send them to one or more instruments. There is only one performer class and performer instances send note objects to an instrument that has specific information about the synthesis device and, in the case of DSP synthesis, the synthesis algorithm. Thus, the performer object has no inherent knowledge of the instrument that is being played. The intelligence of how a note object is realized is done by the instrument. (This is the inverse of the real world where an instrument is an inanimate object that doesn't understand notes and the intelligence lies with the human performer.) The interpretation phase provided by live musicians in the real world (and also provided by ZED) is missing from the Music Kit, whereby the symbols written by the composer are mapped to specific gestures that are applied to an instrument. An unfortunate artifact of the Music Kit design is that if a single instrument is to be played differently in different compositions, a new instrument class must be created for each different interpretation. This is because all access to the instrument is done through one message (called "realizeNote:fromNoteReceiver:"). In contrast, ZED simply requires a new method to play events in a different way. In ZED the performer objects may increase in sophistication and capability in a way that is somewhat analogous to live performers developing increasing subtlety and diversity as they mature musically. In ZED the instrument classes remain relatively stable over time, just as the design of orchestral musical instruments has remained virtually unchanged for the last century.

In the Music Kit, the performer objects have a "noteSender," connected to one or more "noteReceivers," each connected to an "instrument." Pipelines of arbitrary length can be created by using an object called a "noteFilter" after the note receiver (in place of the instrument) that can process the note and then send it on to a note sender, etc. The note filter is a subclass of instrument, thus overloading the semantics of an instrument as being something that synthesizes sound <u>or</u> something that manipulates a note and passes it to another object. These pipelines of note senders, note receivers, and note filters do not map into any concepts in the real world and, because the semantics of these objects is unclear, it is difficult to refine and extend the system. Also, these objects have narrow interfaces (only understanding a few messages). Changing functionality therefore generally requires new classes to be created rather than new methods.

Real-time Control in the Music Kit

The Music Kit conductor object provides control over the timing of a performance, but provides no other capabilities to control other aspects of the performance. Note filter objects are used for other types of real-time control. The note filters are linked together into pipelines and each note filter modifies the note and sends it to the next stage in the pipeline. This approach is similar to the basic mechanism that Music V uses for defining synthesis voices. The filtering and pipelining metaphors of Music V are very natural for *sound synthesis*: the configuration of the pipeline and filters is static during the performance and the system is synchronous. The metaphor does not extend gracefully to the interactive, dynamic, and asynchronous discrete-event based world of real-time control.

A message passing paradigm is more appropriate for real-time control for a number of reasons. Although it is possible to do some types of real-time control with the Music Kit's note filters, it is very difficult to use and debug because of the complexity introduced by the interconnection of the filters. No abstraction is provided and often many note filter classes are required to get different types of effects. The note filter pipelines must be carefully constructed in software by creating and linking the objects together. To achieve global control of the performance, a note filter is copied and linked into multiple pipelines.

The most important shortcoming of using filter pipelines for discrete event simulation is that <u>it is not possible to change the real-time control configuration during the performance</u>. This is because the filter pipeline introduces a delay. Not only does this delay cause latency in the response to real-time inputs, but, more importantly, the performance cannot be dynamically reconfigured because the pipeline may not be empty. Doing so could result in timing problems and glitches.

MAX

MAX (formerly known as "Patcher") [Puckette, 1986; Puckette, 1988] is a graphical music programming environment that allows users to gain complete control of the capabilities of MIDI equipment. MAX is similar in spirit to the early block diagram compilers used to specify oscillators and filters for computer music. The power of MAX is that it employs a unified object programming model in a visual programming environment. The underlying idea is to give users complete control of all the possibilities of a synthesizer by providing them with the ability to specify explicitly and independently where all synthesis control parameters come from: the basic pitch and tempo material, timbre changes, articulation, etc. They can all be controlled physically, sequentially, or algorithmically; if algorithmically, the inputs to the algorithm are themselves controllable in any way.

MAX is based on a graphical programming paradigm, whereby boxes are used to represent objects that are connected with lines that represent messages. Objects wait passively for something to happen to them (i.e., they are event driven), at which time they may respond by activating other boxes. The boxes in MAX are of four types: controls like sliders and buttons; indicators that detect specific numeric values or that an event has occurred; objects described with text that do some kind of computation; and messages, also in text, that are to be sent to other boxes. Messages can be symbols, numbers, or any combination of the two. Objects may have inlets and outlets: when an outlet is connected to an inlet, any message the source object puts on its outlet is sent to all the inlets connected to it.

Accompaniment Systems

The area of real-time music performance systems is still relatively young. The first real-time computer performance system was the GROOVE System [Mathews and Moore, 1970]. This system made it possible to perform music in real-time under computer control. The Conductor Program [Mathews, 1989] grew out of the early GROOVE system. A primary goal of the program is to control digital synthesizers in such a way so as to exhibit sensitivity and responsiveness, as well as expressive musical interpretation. The Conductor Program allows a live performer of the Stanford Radio Baton to act as the "conductor" of the synthesizer performance, giving the user direct, explicit control over the performance.

The focus of the Conductor Program is to allow a user to *expressively* control a synthesizer accompaniment time and loudness. The system has often been used with great success to accompany live musicians, particularly vocalists. The system has very successfully demonstrated that, with a sensitive input controller, computers can synthesize *music* rather than just sound.

The Conductor Program defines a score containing the pitches and durations of the notes to be played by the synthesizer accompaniment. The score also specifies exactly which notes the baton inputs fall on, that is, when to expect inputs from the live performer of the baton. (The system has no knowledge of any other live performers.) The program uses the time of certain inputs from the baton to control the tempo of the synthesizer performance, and others inputs are used to control the volume and other aspects of the synthesizer performance. The live performer of the baton listens to the performance and tracks the pitches and times of the other live performer, adapting to changes and mistakes. The baton does not generate sound. The live performer of the baton has *direct* and *explicit* control over a number of different aspects of the performance.

A somewhat different type of real-time performance system is an accompaniment system such as [Bloch and Dannenberg, 1985] and [Vercoe and Puckette,1985]. The primary focus of these systems is to control a synthesizer accompaniment to follow a live performer playing an electronic musical instrument (a wind instrument or keyboard). The system must have a *score tracker* that can adapt not only to tempo changes, but also to pitch mistakes, skipped notes, and other mistakes by the live performer. Bloch and Dannenberg use a pitch based score tracker that employs a time-independent, statistical pattern matching scheme. This score tracker is designed to work for improvised music and that works for monophonic and polyphonic pitch matching. Conversely, Vercoe and Puckette use a time based approach that uses a learning algorithm. The computer accompanist listens and learns the specific interpretation of the live performer. This score tracker was designed for music that has ornamentation and embellishments similar to those found in Baroque music [Grout, 1973].

Summary of MAX and the Accompaniment Systems

All of the systems were successful at solving the particular problem they were addressing: MAX focused on providing a graphical programming environment for sound synthesis equipment; the Conductor Program focused on demonstrating expressive control of sound synthesis in live performances; and the other accompaniment systems focused on controlling a synthesizer accompaniment to follow a live performer playing a musical instrument.

The Conductor Program gives the live performer direct expressive control over the performance. The other accompaniment systems provide only tempo control but allow the performer to play a musical instrument. All three accompaniment systems address expressive timing and the tight integration of a live performer and a synthetic ensemble.

MAX is *configurable* and *extensible*. The user can configure the performance by specifying what aspects of the performance are to be controlled. And the user can *extend* the system by writing software for new objects and functionality. MAX does not fundamentally support score tracking like the other systems, although because it is extensible a score tracker the user could add one to the system [Puckette, 1990].

None of the systems described are *device independent*. The accompaniment systems are designed for particular types of devices. Although MAX supports a variety of devices, MAX is not device independent because no abstraction is provided to shield the user from the specifics of the device. Therefore, MAX patches deal with low level data values rather than abstracted musical concepts.

Comparison with ZED

Both MAX and ZED have a performance configuration stage. MAX performances are configured by creating graphically and ZED uses a configuration file. (It would, however, be easy to add a graphical user interface to ZED.) MAX uses four types of boxes or objects: controls; indicators or filters (that detect specific numeric values or that an event has occurred); computations; and messages. Messages can be symbols, numbers, or any combination of the two. ZED patches consist of controls (from input controllers), filters, messages, and a receiver object that is to act on an input. The Music Kit is configured by writing methods that create and assemble note filter objects into pipelines. The primary difference between the patching mechanisms of ZED and MAX is the *granularity* of the action that is taken when an input is received. ZED abstracts the action to a single message that is sent to and object. The details of the action are implemented in the method (and perhaps other methods). MAX does not make such a distinction.

MAX was not implemented in an object-oriented language, but it has an object-oriented flavor provided by its "objects" and "message passing." MAX is extensible but the user must define new objects and functionality but with a conventional (non-object-oriented) programming language such as C. Although the Music Kit was implemented in an object-oriented language, the design is not object-oriented because very little subclassing, inheritance, encapsulation, and data abstraction is used. The design is difficult to extend because classes are overloaded and do not have clear semantics.

ZED is both configurable and extensible. A particular performance is programmed by creating a set of performance files for the score, configuration, and patches. And, ZED's functionality can be extend by defining new methods on ZED classes or by defining new subclasses of ZED classes and overriding methods. The semantics of the performance objects are well defined and the number of method interfaces for each class is small.

ZED differs from the Music Kit design in three ways. First, ZED does not rely on generic classes with little semantics (such as those for performers and notes) but rather relies on inheritance and subclassing for creating specialized classes with well defined semantics. Second, ZED's real-time control mechanism is based on message passing rather than filter pipelines. And third, ZED provides an abstract music representation that separates the interpretation information from the notes to allow high- level control.

Summary

The primary difference between ZED and other music performance systems is in its approach. ZED's focus is on providing a system that *behaves* in a way similar to that of a live orchestra. Abstractions are provided for the components of a live orchestra. This allows the user to interact with the system at a high- level. The user can cause messages to be sent to the conductor or a performer, and can affect *musical* aspects of the performance such as tempo, key, dynamics, and style. The user could for example, cause a trumpet performer object to "make all staccato notes quieter and shorter."

ZED uses object-oriented design methodology and it was found to be very suitability for discrete event simulation problems. This design approach enabled the system to closely reflect the orchestra model. ZED provides device independence across synthesis devices and input controllers, and provides score file independence.

As a result of modeling the real world orchestra and using an object-oriented design methodology, the resulting implementation had a number of desirable features. Defining an abstract digital music representation that had the properties of Western music notation resulted in the system being able to read a variety of different score files and map them to a single representation. Separating out the interpretation symbols from the note symbols provides an elegant way to provide real-time control of the performance. Using performer objects that hide the specifics of the synthesis device allows the scores and the real-time patches to be device independent. The specialization of performers through subclassing and message overriding provides easy extensibility. In addition, the use of performer objects creates an evolution path for the system so that new technologies and new knowledge of how to control the technology can be incorporated.

References

Bloch, J. J. and Dannenberg, R. B. Real-Time Computer Accompaniment of Keyboard Performances. In Truax, B. (ed), *International Computer Music Conference at Simon Fraser University*. San Francisco, CA: Computer Music Association, 1985, p. 279.

Grout, D. J. *A History of Western Music*. New York, NY: W. W. Norton, 1973, p. 392.

Jaffe, D. A. *From the Classical Software Synthesis Note-List to the NeXT Scorefile*. Preliminary Draft, Menlo Park, CA: NeXT Inc., 1989a.

Jaffe, D. A. Overview of the NeXT Music Kit. In Wells, T. and Butler, D. (eds), *International Computer Music Conference at Ohio State University*. San Francisco, CA: Computer Music Association, 1989b, p. 135.

Mathews, M. V. *The Technology of Computer Music*. Cambridge, MA: MIT Press, 1969.

Mathews, M. V. and Moore, F. R. *GROOVE — A Program to Compose, Store, and Edit Funtions of Time*. Communications of the ACM 13(12):715-721, 1970.

Mathews, M. V. The Conductor Program and Mechanical Baton. In Mathews, M. V. and Pierce, J. R. (eds), *Current Directions in Computer Music Research*. Cambridge, MA: MIT Press, 1989, p. 263.

Puckette, M. Interprocess Communication and Timing in Real-time Computer Music Performance. In Berg, P. (ed), *International Computer Music Conference at Royal Conservatory, The Hague, Netherlands*. San Francisco, CA: Computer Music Association, 1986, pp. 43-46.

Puckette, M. The Patcher. In Barlow, C., Lischka, C., and Pannes, M. (eds), *International Computer Music Conference at GIMIK, Cologne, West Germany*. San Francisco, CA: Computer Music Association, 1988.

Puckette, M. Presentation at CCRMA, Stanford University, Palo Alto, CA, May 1990.

Vercoe, B. and Puckette, M. Synthetic Rehearsal: Training the Synthetic Performer. In Truax, B. (ed), *International Computer Music Conference at Simon Fraser University*. San Francisco, CA: Computer Music Association, 1985, p. 275.

Appendix D

MIDI Specification

TABLE D.1 MIDI Channel Voice Messages

Name	Status	Data Bytes	Description
Note Off	1000nnnn	0kkkkkkk	note number (0-127)
		0vvvvvvv	note off velocity
Note On	1001nnnn	0kkkkkkk	key number
		0vvvvvvv	if 0 then note off else velocity
Poly Key Pressure	1010nnnn	0kkkkkkk	key number
		0vvvvvvv	pressure value
Control Change	1011nnnn	0ccccccc	control #(0-121) (See table 3)
		0vvvvvvv	control value
Program Change	1100nnnn	0ppppppp	program number (0-127)
Channel Pressure	1101nnnn	0vvvvvvv	pressure value
Pitch Bend	1110nnnn	0vvvvvvv	LSB
		0vvvvvvv	MSB

Notes:

- nnnn is voice channel N-1, i.e., 0000 is channel 1 . . . 1111 is channel 15
- kkkkkkk is note number (0-127), kkkkkkk = 60 is Middle C
- vvvvvvv: key velocity, logarithmic scale whereby:
  - 0 is off; 1 is ***ppp;*** 64 is ***mp;*** 127 is ***ff***
  - vvvvvvv = 64 default (when device has no velocity sensors)
  - vvvvvvv = 0: note off with velocity = 64
- vvvvvvv: control value (MSB)
  - for controllers: 0 to 127
  - for switches: 0 = off, 127 = on, 1 to 126 ignored

TABLE D.2 MIDI System Common Messages

Status	Description	Message Name
11110010	Song Position Pointer	songPosition:
11110011	Song Select	songSelect:
11110110	Tune Request	tuneRequest:
11111000	Timing Clock	timingClock:
11111010	Start (from beginning)	startSequence:
11111100	Stop	stopSequence:
11111110	Active Sensing	activeSensing:
11111111	System Reset	systemReset:

TABLE D.3 MIDI Controller Messages

Control Number	Control Function	Message Name
5	Portamento Time	portamentoTime:
8	Balance	balance:
10	Pan	pan:
64	Damper Pedal (sustain)	damperPedal:
65	Portamento	portamento:
66	Sostenuto	sostenuto:
67	Soft Pedal	softPedal:
92	Tremolo Depth	tremoloDepth:
93	Chorus Depth	chorusDepth:
94	Celeste (Detune) Depth	celesteDepth:
95	Phaser Depth	phaserDepth:
all others < 122		controllerNumber:
all others >= 122		channelModeNumber:

TABLE D.4 MIDI Channel Mode Messages

Control Number	Control Value	Description
122	0	Local Control Off
122	127	Local Control On
123	0	All Notes off
124	0	Omni Model Off (all notes off)
125	0	Omni Mode On (all notes off)
126		Mono Mode On (Poly Mode Off, All Notes Off)
126	M	M is number of channels
126	0	number of channels is the number of voices
127	0	Poly Mode On (Mono Mode Off, All Notes Off)