Programming Models for the Development of Interactive Audio Applications

by

Eric Michael Jordan

B.S.E., Computer Science
Princeton University, 1992

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Author .....
Department of Electrical Engineering and Computer Science
November 9, 1994

Certified by .....
Tod Machover
Associate Professor
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Programming Models for the Development of Interactive Audio Applications

by

Eric Michael Jordan

Submitted to the Department of Electrical Engineering and Computer Science
on November 9, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

There are several systems that use external hardware, stand-alone applications, or special purpose languages to facilitate experimentation with audio processes ([22, 35, 44] and [29, section 3.2]). These packages take advantage of the extra efficiency gained by hardware support and by focusing a complete application on a particular problem. However, as computation becomes cheaper, increasingly more complex synthesis and analysis algorithms can be performed in real time on high end workstations without the help of specialized hardware. Environments that are geared specifically towards manipulating signals might solve that particular problem extremely well, but cannot be used at the level of notes or user interaction; these problems must then be solved independently. Adding support for audio to an already existing programming language would eliminate the need to create and support specialized, ad-hoc syntaxes. A general purpose language would ease the transition from the signal level to the note level, because all of the code would exist in the same framework. An extensible language would allow applications to combine low level audio routines with packages from just about any domain, including networking, artificial intelligence, and scientific computation. Finally, an interpreted language would give the user immediate feedback, and would allow applications to be flexible and configurable.

We extended one portable and available language that meets these requirements, Tcl/Tk [30], with the framework for a traditional patch-based audio system. The resulting package, "swish," lets the user create applications that implement, control, and interact with audio processes at a high level. Since Tk provides access to a widget set, the interaction can be either textual or graphical.

In order to incorporate a unit generator patcher into Tcl/Tk, it was necessary to reconcile the programming model presented by the audio system with that presented by the language. Using ideas found in [3], we replaced the traditional graphical dataflow machine with a more general "task" model. In the task model, the temporal relationships are specified independently of the functional relationships. We show

that this model is still amenable to both graphical and textual representations, fits naturally into the Tcl/Tk framework, and is more general and flexible than several other existing systems.

Finally, after examining the way that a typical patch is executed, we discuss the need for a language that is capable of performing incremental compilations. A typing system, lexical scope, and a functional programming style are other language features that might be useful, but are not found in Tcl/Tk. We use these observations to examine several languages that may become available in the near future.

In the conclusion, we hypothesize about the role that audio will play in the near future in the developing networked multimedia environments, and argue that the applications that are going to have the most influence on the way that we use and view computers are going to be the ones that make portability a central concern.

Thesis Supervisor: Tod Machover

Title: Associate Professor

Acknowledgments

I would like to thank the members of my late night support group, Michael Wu and Upendra Shardanand. Mike deserves further credit for introducing me to John Woo films and for teaching me how to use the fax machine.

I would also like to thank my fellow roadies, Mike, Eric, Eran, and Joe, for exposing me to the wild side of St. Paul.

Alex, Damon, Dave, Josh, Tom, and Suzanne created an exciting and dynamic environment to work in, and I am grateful to them for their friendship. In addition, I'd like to thank Professors David Tennenhouse, Barry Vercoe, and Neil Gershenfeld not only for making themselves accessible, but also for their feedback and encouragement.

Once again, I find myself owing my parents and my brother for their tolerance, not to mention ongoing support.

Finally, I'd like to thank Professor Tod Machover for his support, feedback, and general nonstop enthusiasm, as well as for helping me explore this fascinating and exciting field.

Contents

Chapter 1

Introduction and Overview

A large amount of research has been done by the computer music community to determine the data types and programming paradigms that are effective in music and digital audio projects. Many of these concepts have been demonstrated through the use of specially designed languages or special purpose hardware. However, now that many synthesis and analysis algorithms can be performed in real time on high end workstations without the use of a special signal processing chip, the efficiency hit taken by using a higher level, more flexible environment to develop audio applications is more acceptable. Recently, the programming systems community has shown that code can be made modular and reusable through the use of "very high level languages." We propose that extending a high level language with the tools needed to do music and audio would benefit both communities: Designers of computer music projects would have access to a system that integrates the tools that they are accustomed to with code developed for other aspects of computation and interaction, such as graphical user interfaces or networking, while application developers would be given the ability to interact with the user through audio and music.

We illustrate this point with a prototype system, "swish," that extends the Tcl/Tk scripting language [30] with a framework that supports the patching together of "unit generators" to describe synthesis and analysis algorithms in a manner that is extremely familiar to the computer music community [5, 28, 29]. Tcl is a scripting language that meets all of our requirements: it is portable, freely available, high

level, extensible, and interpreted. Tk is an extension to Tcl that provides easy access to a widget set.

It is common for patch editors to provide graphical interfaces. Tk is particularly useful because it enables swish to distinguish itself from these applications by providing both graphical and textual representations of the patches. Unit generators can be manipulated using an intuitive graphical user interface, and then represented textually for use in an application. Alternatively, since all of the unit generators can be manipulated using Tcl procedures, the user has the option of bypassing the restrictive graphical framework altogether, and creating patches directly through textual scripts. Finally, the user can write applications that use audio patches together with Tcl procedures that provide other forms of interaction and computation. Now, interesting synthesis or analysis algorithm can be implemented, without having to write a specialized application that can only read from and write to sound files. Instead, by implementing it as an extension to swish, it can be used in conjunction with other audio processes, and controlled in real time using an intuitive and flexible graphical interface.

Another distinction between traditional systems and swish is that swish uses a task-based programming model [3, 4] instead of a dataflow machine. The basic idea behind the task-based model is to divorce the temporal dependencies of the unit generators from their functional dependencies. The user then has the flexibility needed to handle special cases that the traditional "dataflow machine" approach has trouble with. The task-based model also facilitates a smooth transition from the signal rate to either a synchronous or asynchronous control rate, and leaves open the possibility of developing sets of tasks that work in domains besides signal processing. After describing several existing computer music programming environments in chapter two, we present this model in chapter three, and show that it can be represented graphically in a manner that fits elegantly into the Tcl/Tk framework. Chapter four provides further discussion about swish, the task model, and some important implementation issues.

Swish was developed on an DEC 3000 model 400/400S running OSF 2.0, using a

J300 Sound and Motion card to provide audio, and a KEY Electronics Midiator-124W box to establish MIDI capabilities through the serial port. AudioFile [25], a transparent, portable audio server written by DEC at Cambridge Research Laboratory, was used to interface with the J300 Sound and Motion card. Ak [32], also written at CRL, serves as the interface between AudioFile and Tcl.

Since the Alpha was a relatively new architecture at the time of development, many languages available on other platforms had not been ported yet. At the time, Tcl/Tk was the only free, available, extensible, interpreted, high level language with easy access to a widget set. Furthermore, the designer of Tcl/Tk did not orient his language towards proving a theoretical point about a new and improved way of specifying computation or a better method for creating data abstractions. Instead, he realized that developers of contemporary applications need to be able to specify interaction with the user at a high level. Because Tcl/Tk was designed to support interesting interactive behavior, it has a powerful combination of features that are missing in many popular languages. Its self-inspecting facilities made the current state of the interpreter available to scripts, and aided the development of a prototype of a graphical task browser and editor. Its support for variable traces made it possible for Tcl variables to asynchronously control task parameters. Its concept of tags raised an interesting idea about how to organize dynamically created structures, like notes or scores. Finally, the "configuration variables" routines enabled the user to configure an audio task using the same intuitive syntax used to configure a Tk widget. However, because it is a scripting language, it lacks several important properties, including incremental compilation, lexical scope, and a typing system, that can be found in more abstract and theoretical languages. In chapter five, we discuss the potential utility of incremental compilation and a typing system for an application like swish, and consider several appropriate languages that may be available in the near future on the platform in question.

In recent years, our conception of the nature of a computer has started to change. Users are beginning to expect to be able to view, edit, and communicate data representing a variety of media at a high level. One possible outcome of this is that

the computer music community will be greatly broadened, as more people will have access to the tools needed to investigate this form of art. However, another possibility is that audio will be incorporated into multimedia systems as merely another form of interaction, and play a "voice-mail" kind of role, deriving its purpose and motivation from video, graphics, and networking applications, rather than having any independent meaning. It is crucial that the computer music community influence the path that is being taken, by exporting the tools that they have developed to an audio environment accessible to the general computer community. In chapter 6, we hypothesize about the roles that audio might play in a typical computer system in the near future, and conclude that portability will be an essential feature in any successful development environment for handling audio.

Chapter 2

Related Work

The history of languages and tools for specifying computer music is incredibly extensive. It seems that almost every concept studied in computer science, from context free grammars [20], Petri nets [19], and Markov models [29, section 5.5], to expert systems [23], neural networks, and genetic algorithms [21], has been the basis for some algorithmic composition, sound synthesis technique, or musical listening tool. However, many of these projects have been implemented by developing stand-alone applications, making it impossible to work with them in an environment where all of these mechanisms can interact with each other and with the outside world. To address this issue, several special purpose languages or language extensions have been developed that give the user high level access to a variety of algorithms for handling both signal and note level abstractions. Here, we look at just a few of the more popular systems. Keep in mind that this is just a small sampling of a vast literature, and that we are looking specifically at the programming model used by these packages, and how they integrate music into the programming system as a whole, rather than, for example, at what they have to say about the nature of musical structure.

2.1 Csound, cmusic, and the Music-N family

The Music-N family of computer music systems [34, 45] uses a two-tiered approach: One syntax is used to describe how to produce signals or instruments given a set of

unit generators, and another is used to describe scores. The user divides the code he writes into two parts, based on the type of the object that is being worked on. The structure of the two-tiered approach creates a model of computation that provides the composer with a simulation of the hardware found in a traditional music studio, where MIDI instruments or sequencers send triggers to synthesizers and effects boxes. Csound [44] and cmusic [29, section 3.2] are two popular members of the Music-N family.

Csound provides the user with a powerful library of audio routines that have been proven to be effective in computer music pieces. The parameters for the synthesis algorithms have been tuned, over time, to give the user the correct level of control. Also, Csound can be extended, so that new synthesis algorithms can be added as they are discovered. Finally, its portability has helped make Csound a standard for specifying computer music pieces. For example, there are extensive online archives of musical compositions, specified as Csound input files. Miller Puckette, author of MAX (discussed below), has pointed out that musical compositions should be reproducible for decades, if not centuries. If a musical piece is specified using a particular computer platform or is tied to a popular computer language, then the composition will not outlive the technology or the language. MAX and Csound are appropriate as standards for specifying computer music compositions, because they are independent of any particular architecture or computer language. Of course, this also means that they can not take advantage of the architecture of a given machine, or of the abstractions available in some newly developed programming language. It is clear, however, that portability, longevity, and availability are extremely important features of any system that is intended for artistic use.

In [29, page 174], Moore writes

> "Because the computations involved are arbitrarily complicated and therefore arbitrarily time-consuming, programs such as cmusic do not run in real time, which would allow them to be played by live performers like musical instruments."

Here, there is an implication that real-time interaction is merely a question of

efficiency: if computers were fast enough, then cmusic could be used to specify interactive audio applications. However, it is not at all clear how to specify real-time temporal behavior using the programming model presented by the cmusic score language. It is our contention that real time interaction introduces new questions about how languages should be structured, and is not merely an issue of efficiency. Furthermore, even if the proper language structure was determined, many non-audio capabilities, such as networking, accurate scheduling, and graphical user interfaces, would have to be added to cmusic to support different aspects of real time interaction. Finally, developing a stand-alone audio and music system only for the specification of computer music neglects the systems programming community, who would like to use this technology in their applications. The Music-N family fits very cleanly into the classic UNIX "file-oriented" philosophy: Pipelines of soundfiles and programs that output score files are very common when writing non-realtime compositions. However, the challenges that networking, multimedia, and graphical user interfaces bring to the UNIX world raise many interesting questions about how applications should be structured so that modularity can be maintained as code to support new forms of interaction is developed.

2.2 MAX

Several systems, in part as a reaction to the two-tiered approach, go to the other extreme, and make a point of having a completely unified approach. These systems include MAX [35] and Nyquist [11] (discussed below). MAX is a popular system for specifying note and control level interactions, primarily through MIDI, although signal processing facilities are available on some platforms. It distinguishes itself by using a purely graphical interface. The user creates MAX "patches" by making connections between "boxes." Note, however, that the data being passed between boxes is often MIDI information, and not an audio signal. This makes the interpretation of a MAX patch different from the interpretation of a graph that represents a patch of unit generators, because in a MAX patch, data flows at irregular intervals through different

parts of the graph, while in a patch that represents a signal processing synthesis algorithm, data flows continuously through the whole graph.

In a MAX patch, the connections describe how data is passed from one box to another. The lines are also interpreted, however, by a set of conventions that determine when a box is activated. For example, many boxes execute when data appears on their leftmost input. It is also often possible to trigger execution by passing around "bang" messages. Its intuitive graphical interface and conventions for how boxes are executed make MAX easy to learn and independent of any particular computer architecture or programming language. Sliders, buttons, text boxes, and other important elements of graphical user interfaces can be created easily, and can be connected to ports controlling MIDI devices or audio processes. However, because it insists on remaining independent of any particular environment (as described in the previous section), it cannot easily interact with code that was not written explicitly for use as a box in a MAX patch. Conversely, the graphical interface provides one intuitive and high level way to manipulate MAX code and create patches. However, because MAX insists on isolating itself, it is difficult to create and manipulate patches in any other way.

2.3 Nyquist

Nyquist [11, 9] is another system that unifies the two tiered approach. That, however, is where the similarity with MAX ends. Nyquist extends xlisp with a set of unit generators. However, instead of describing a patch graphically, it is described textually, using lazy evaluation. Each unit generator is a function of several parameters. Some unit generators have parameters that are usually the output of another unit generator. For example, the multiply function takes, as input, the outputs of two other unit generators. When a multiply is defined, two other unit generators are taken as input, and an unevaluated function is returned. In this way, patches are built up. When a patch is finally evaluated, the remaining parameters are taken from the "transformation environment."

This brief description may sound complicated, but the result is a level of data abstraction and a sense of scope that makes this language very powerful. First class functions give us the ability to manipulate sounds through the functions that describe how the sounds are going to be generated, rather than specifying at each step how buffers in memory should be manipulated. Lazy evaluation ensures that each of these functions is evaluated only when the patch is completely described, and that each function is evaluated at most once. Some of the potential as well as a few difficulties offered by functional languages in general are discussed further in chapter five. We note here, however, that specifying interactive temporal behavior and creating a graphical representation of a program written in a functional language are both interesting but open research problems [2, 36].

2.4 The NeXT Music Kit

The NeXT Music Kit[22] is an extension to NeXTSTEP that gives the user access to a library of unit generators, and the ability to make patches out of them from within any language that can link to object files, including Objective C and Common Music [42]. Objective C provides an object oriented environment, so that musical constructs, such as notes or scores, can be described using object classes. Common Music provides an interpreted, portable environment, so that the user can interact with the system dynamically, and then execute his compositions later on other platforms (presumably using a different library to generate audio). The resulting access to low level audio patches from within a high level, complete language creates a flexible and powerful environment. The Music Kit is part of the NeXTSTEP environment, so that the programmer can use all of the features of that operating system when developing applications that handle music or audio. This includes the interface builder (which greatly simplifies the creation of graphical user interfaces), Display PostScript, and access to the plethora of music-oriented applications that have already been created for NeXT workstations (There is a large library of software available, partly because NeXT workstations have been used very heavily by several centers for computer music

research, including CCRMA, Ircam, and Princeton).

Because the NeXT Music Kit is an extension to existing languages, it is easy to use it to write applications that use audio patches and musical abstractions together with other libraries. For example, the application kit gives the programmer access to widgets that can be used to build a graphical user interface. This has been demonstrated by the author's undergraduate research project grasp, the "GRAphical Synth Patch" editor, an implementation of a program proposed in [28]. Grasp is a standalone application that provides both a graphical and a textual interface to the patch creation aspect of the Music Kit. Grasp and other Music Kit interfaces demonstrate that the Music Kit is amenable to a graphical representation.

The Music Kit provides reliable scheduling and extra efficiency by writing patches directly onto the signal processing chip that comes with the NeXT workstation. In chapter five, we will discuss the importance of incremental compilation. Note, for now, however, that generating code to be executed on external hardware is potentially useful independently of the gain in efficiency, simply because it allows us to create our patches on the fly. [5] provides a good description of another system that dynamically generates code for a DSP chip based on a graphical patch of unit generators. Using a DSP chip greatly simplifies the programming model, because it automatically determines the proper order of execution for the patches, and avoids several scheduling issues that arise when patches have to be executed as separate processes or threads (see chapter five). There are two small disadvantages to the approach taken by the Music Kit. First, patches must written in the assembly language used by the chip. Second, our programming model has been fixed: Our audio patches must be static, the dataflow machine simulated by the signal processing chip can only be used to execute audio patches, and, conversely, the audio patches can only be executed by the dataflow machine.

Unfortunately, the Music Kit's general approach has hurt its portability and availability. It is dependent on a language (Objective C), on a platform (NeXTSTEP), and on external hardware (the signal processing chip in the NeXT) that, because of the status of the NeXT workstation, are no longer accessible to the typical computer

user. Although it is now possible to use Objective C on many platforms, NeXTSTEP is only beginning to become available, while the status of the Music Kit is not clear. The Music Kit will either have to remove its dependence on the signal processing chip, or these chips will have to be built into more machines. This may require changes to the programming model presented by the Music Kit, and to the way that the Music Kit interacts with the operating system.

The importance of the longevity of an environment used to specify musical compositions was discussed above. Portability is also important for other reasons. The development of the world wide web and increasingly available audio capabilities has the potential to change the computer music community from being concentrated in several centers of computer music research to being a large distributed network of people who interact and dynamically exchange musical ideas and applications online. This is only possible, however, if the applications that are developed are capable of running on a variety of platforms. In general, users will not want to gear their hardware or their programming environment towards getting one particular mode of interaction with their machines highly optimized, yet they will want to have audio and music available as an important part of a more general multimedia machine. Furthermore, application developers will want to be able to use audio and music, without being forced to limit their applications to run on a particular platform. NeXT is addressing this issue by porting NeXTSTEP to several platforms, and by creating the OpenStep standard. If they are successful, then it is possible that, at some point, the Music Kit will be considered to be portable. It would be interesting to see an application change its status from being available, to being neither available nor portable, to being portable, without ever having changed the code itself! This phenomenon underlines the shades of grey involved in the question of the definition of portability, and shows that it is really tied to our constantly changing, almost arbitrary perception of what a computer consists of.

Smalltalk is another extensible language that has been extended for use in computer music [33]. Smalltalk provides many of the language abstractions that we will discuss as being essential. It is an interpreted, object oriented, multi-threaded lan-

guage capable of performing incremental compilations, that has been shown to be appropriate for the development of specific musical installations [33, 46]. However, because of its large class library, and because it takes over many of the duties of the operating system, implementing the Smalltalk language itself requires too much development time and code to be freely available and portable, particularly on newer, more powerful platforms. [33], for example, describes a system based on Smalltalk that shares many of our goals:

> "The motivation for the development of IDP is to build a powerful, flexible, and portable computer-based composer's tool and musical instrument that is affordable by a professional composer (i.e., around the price of a good piano or MIDI studio.)"

Here, we see a clear emphasis on flexibility and portability. However, by limiting their goals and allowing themselves to rely on commercial software, by assuming that the computer system will be configured and developed from day one with the goal of creating a tool for a professional composer, they have made their system inaccessible to people who are not willing to dedicate their whole computer environment to the composition of computer music. In particular, complete versions of Smalltalk are not expected to be available on the Alpha for at least another year. Even if a user has access to Smalltalk, an application that relies on this language forces the user to dedicate the whole machine to that environment. [33] has achieved the goals it set out for itself, and has proven to be influential within the computer music community. However, it is not the solution for the typical workstation user, who would like to have access to audio and music as one tool in an interactive multimedia programming environment, but is not necessarily willing to design his whole machine configuration around this particular component. It is possible that the workstation designers will realize that Smalltalk provides a much better development environment than Unix and C, and that at some point, Smalltalk will be available as part of any high end workstation that is capable of the computational power needed to handle real-time interactive audio. For now, however, we consider these problems in the context of the world as it exists.

Note that we are guilty ourselves of drawing a rather arbitrary line in our definition of portable, by restricting our interest to Unix workstations. Our only justification for this is that commonly available computers that use other operating systems are only recently becoming powerful enough to handle sophisticated real-time interactive audio processing. However, as time progresses, it will be important to maintain portability. This requires that we use a small system, with no reliance on specific hardware or on a commercial language.

2.5 VuSystem, Pacco, and Tcl-Me

Recently, several portable, real-time, interactive multimedia packages have been developed by the programming systems community, providing the ability to patch together video and graphics modules. They address many of the implementation and scheduling issues facing the developer of an interactive audio package. They include the VuSystem [43], Pacco [7], and Tcl-Me [16]. All of these projects were developed in UNIX, using Tcl.

The VuSystem divides its code into "in-band code" and "out-of-band code." The presented model of computation could be described as a client/server model: in-band code consists of modules that run continuously in the background, responding to asynchronous demands made on it by out-of-band code. Out-of-band code, responding to user input or other external events, starts, controls, and destroys these modules. [13] discusses similar concepts, using different terminology, from the computer music perspective. It introduces the term "resource" for in-band code, and contrasts it with the concept of an "instance," which is a procedure that is "independent and isolated." An example of a resource is given by the AudioFile server [25], which runs in the background, processing play or record requests. An example of an instance is a procedure that plays a MIDI note: it starts up, it executes, and then it goes away, without responding to any events that other procedures might try to give it. [13] proposes the use of a hierarchy of modules, each module treating ones below it in the hierarchy as resources, and the one above it as an instance. This hierarchy provides

a view of the world where control starts at the top (perhaps with the user or the operating system), and is passed down to the lowest level, gaining more detail as it goes down, until it reaches the output. The process can be thought of in terms of decoding. For example, the user might push a button on a controller that sets off a score; a sequencer will decode the reference to the score into a list of notes, and send that list of notes to the synthesizer resource, which decodes each note into a signal, and sends the signal to the mixer resource. This model suggests that for computer music applications, a two level in-band/out-of-band hierarchy provides the right kind of abstraction, but may have to be extended further.

With the VuSystem, modules automatically become active and process data as it arrives on their input ports. The user can think of the modules as running continuously and in parallel, leaving it to the system to provide a scheduling algorithm that supports this view. Each module makes itself available through receive and send procedures. Receive procedures are called when data is made available to a module, while send procedures are used to invoke the next module in the pipeline. Also, it seems that receive and send procedures can use timestamps to determine if it is appropriate for them to execute. In this fashion, as in MAX, each module can use the status of its input ports to enforce a convention that describes when the module is to be executed.

To contrast, modules in Tcl-Me make themselves available through an activation procedure, and give the user the ability to specify activation through higher order modules. In the VuSystem, modules activate themselves when data is present on their input ports. In Tcl-Me, modules are activated, either by other modules or by the user. The Tcl-Me approach is similar to the interaction between swish's invocation procedures and its higher order tasks. We discuss this issue further in chapters three and four. In particular, we will argue that the latter approach is more flexible, and results in a more natural specification of temporal behavior.

Chapter 3

Swish

3.1 Framework

Swish was not designed just to let the user patch together unit generators, and listen to the result. Instead, emphasis is placed on the utility of making audio processes available in a high level language. Natural and elegant integration of the audio processes into the high level, interactive language was the overall goal in the design of swish's basic framework. The result is that every unit generator and audio process can be configured using Tcl scripts. Rather than having one way of dealing with operations that work on signals, and then another for operations that work on notes, the two problems were properly defined, and then merged into the same setting. Instead of creating a special purpose object system with its own syntax and support routines within Tcl, we used the already standard and generalized Tk "parameter configuration" routines, so that the user can interact with swish using the same syntax that is used to create and configure widgets. With these principles in mind, the toolkit has been structured into three layers of abstraction, each one intended as a general purpose tool, instead of just as a means for implementing the next layer.

The first layer is the description of the types useful in a music system. A type provides a description of a format that can be used to pass information from one task to another. It specifies how a section of memory should be interpreted. Swish has three types at the moment, but more could easily be added. Short buffers are buffers

of short integers, used for linear 16 bit signals. They can be sent to the speakers, used to store audio input, or written out to disk to create standard sound files. For more accurate, internal use, there is also support for buffers of long integers. Tasks that operate on long buffers follow the convention that one fourth of the bits (this will usually be one byte, however recall that the Alpha has long integers that are 8 bytes long) are considered to be after the decimal point. The buffers of long integers provide a representation of floating point numbers that can be manipulated using integer arithmetic. This kind of trick is extremely common in inner loops of sound synthesis programs. Our use of it is loosely modeled after code found in Nyquist [11]. Whether or not it is truly necessary anymore, given the improved performance of floating point arithmetic, is an interesting question. Finally, we provide support for instances. From the point of view of the tasks, an instance is a constant double; if the task is expecting a buffer, then it treats the constant as if it were a buffer of long integers, where all of the elements have the same value. However, any instance can be tied to a Tcl variable, so that every time the Tcl variable changes, the value of the instance is changed as well, and vice versa. This enables the asynchronous control of parameters. For example, the user can control an input parameter through a slider widget, and the value will only be updated when the user makes a change. If the user doesn't touch the slider, then the code will run as efficiently as if it were not there.

The importance of the proper specification of types can not be overemphasized. The way to truly extend the domains that the system works in is to add support for new types. A description of how memory is to be used is at the heart of any set of unit generators. Each type should have at least one visualization and editing tool associated with it, as well as a textual representation in Tcl. Although swish currently emphasizes support for signals, its status as a prototype tool would be elevated to that of a complete music and audio synthesis system simply through the addition of note lists, filters, and envelopes as types. Later, we will see, in particular, that adding tasks themselves as a type to the system would actually create a more powerful and dynamic programming model.

The second layer consists of the tasks that act on the types described in the first

layer. These are the traditional unit generators. Swish currently provides support for addition, multiplication, audio input and output, interpolation, bandpass filters, topological-sort based apply tasks, and script tasks.

The types and tasks described above are implemented in C, using the Tcl/Tk C API (Application Programmer's Interface). The third layer provides the graphical and high level textual representations of the tasks and types. The textual representation of the tasks has the same high level syntax that is used to represent Tk widgets. Currently, swish provides its own graphical representation of the tasks through a set of browsing and editing tools. However, it would be worthwhile to investigate incorporating the graphical representations of the tasks and types into a more sophisticated, previously existing browser and editor. It has already been shown that a simple program, tkInspect [40], written independently of swish, can be used to examine and modify the current configuration of the tasks. Xf [14] and hierQuery [37] are two possible candidates for browsing and editing tools that might be extended to support the manipulation of swish types and tasks.

It is important to emphasize that, in a sense, we are using audio and music as an excuse for investigating programming models, to see how well they handle side effects, user interaction, and real time processing with different rates of control. The resulting model, therefore, ought to be general enough to work in other domains as well. Examples that come to mind include scientific visualization, physical modeling, and robotics. Support for these domains could be added, simply by creating appropriate sets of types and unit generators.

3.2 The Task-Based Programming Model

The use of dataflow machine simulators to patch together audio processes is extremely common. It provides an interface that dates back to the configuration of analog synthesizers. As a simple example, frequency modulation synthesis can be described by connecting the output of one sine wave generator (generator "A") to the frequency (or, more commonly, phase) input of another (generator "B") [29](See Figure 3-1).

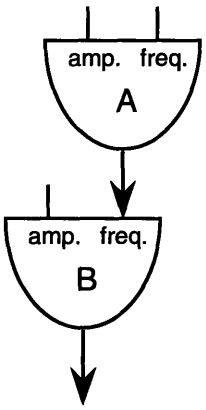

Figure 3-1: Patching together unit generators to describe FM synthesis.

In an analog world, the meaning of the line connecting the two sinusoids is completely clear: the output signal of A is used as one of the input signals for B. However, in a computer, computation is discrete and (in this case) serial, and suddenly, the line connecting the two sinusoids has two completely separate meanings. First, it represents a functional dependence between the two unit generators: The buffer used as the output of one sinusoid is to be used as the input for the other. Second, it represents a temporal dependence between the two unit generators: Every time A is executed, B should be executed as well. The functional dependence statically describes how data is passed, while the temporal dependence actually causes execution to occur.

In the analog world, the temporal dependency is not an issue, since the unit generators execute continuously and in parallel. However, if a computer tried to simulate the patch in figure 3-1 by repeatedly executing the sinusoids as often as possible and ignored the temporal restriction imposed by the line, it could end up executing B before its input buffer has been updated, or it could execute A too often, overwriting the buffer before B has had the opportunity to process it.

Many systems compensate for this discrepancy by making reasonable assumptions about temporal dependencies based on the existence of functional dependencies. For example, [35] describes how MAX uses the convention that one unit generator A will cause another unit generator B to execute (a temporal dependence) if the output of A

is connected to the leftmost input of B (a functional dependency). A more common assumption is the one made by dataflow machines. The "dataflow assumption" is that the temporal dependencies are the same as the functional dependencies. This means that a unit generator A will automatically be executed after each of the unit generators that have their outputs connected to A have been executed. For example, in Nyquist, "behaviors" (functions that produce sounds) can be combined through function composition (using the dataflow assumption) to create other behaviors. At some point, however, behaviors are instantiated into sounds; from that point, only temporal relationships can be specified.

These assumptions are often, but not always, correct. If the user is trying to describe the computation of an expression, for example, then the dataflow assumption is generally applicable. However, graphical dataflow machines have traditionally had difficulty describing sophisticated programming constructs, such as control flow, conditionals, and loops ([1] and [39, chapter 7]). Methods have been developed to handle these situations. In particular, an apply node can be used to elegantly provide many of these facilities [1, 6, 39]. However, when you add the issues that arise because of the use of audio, such as time delays, user events, side effects (audio output, for example), different rates of execution, and asynchronous interactions, the dataflow assumption breaks down very quickly.

Rather than having the designer of the language decide when a unit generator should execute, a language should give the programmer the ability to specify temporal behavior independently of functional behavior. Then, if a particular user of the language decides that he wants to apply the dataflow assumption, he can configure the system so that every time he specifies a functional relationship, a temporal dependency is inferred as well.

The task model gives the user this flexibility. It was originally introduced in [3], and a description of a language that uses it is found in [4]. We describe our use of it in the next few sections. Note that we strongly deviate from it in our use of higher order tasks to specify temporal dependencies.

3.3 Higher Level Tasks

In programming languages terms, a task serves the same purpose as a closure: It provides a description of a function to be executed, together with the environment that it should be executed in. They differ from a closure, however, in the way that the execution environment is described. The key to understanding the task model is understanding how the tasks specify the execution environment. In other words, we must look at how the tasks are represented in memory.

Each task has a list of input and output ports, a "timing" structure that describes when a task executes, and the function that the task executes when it is invoked. Based on figure one of [3], this might look like:

```
task 1
    function
    timing
    ports
task 2
    function
    timing
    ports
...
task n
    function
    timing
    ports
```

[3] proposes that each task specify its temporal behavior through a set of conditions based on the status of its input port. As discussed in chapter two, MAX and the VuSystem use this idea, and create modules that enforce conventions that describe when the modules should execute, based on when data becomes present on input ports. However, the result is a two-tiered system that is not quite an accurate representation of how a typical user thinks about the processes that occur in a computer. Although it is natural to specify "task x receives its first input from port y," a similar temporal statement is harder to formulate. Even [3] says that "[the timing information] is the behavior of the task seen from the outside." Therefore, we propose that the information about the timing behavior of a task should be removed from the

description of the task itself, and that all of this information should be collected into data structures outside of any individual task:

```
task 1
    function
    ports
task 2
    function
    ports
...
task n
    function
    ports

task 1
    timing
task 2
    timing
...
task n
    timing
```

This representation captures the idea that a task actively performs operations on memory, but is executed by some other agent. A task does not specify when it should be executed; rather, some other task causes it to be executed. Of course, in order for the whole process to be initiated, it is necessary for some event external to the system to cause some task to execute. Typically, either the user or the system clock will perform an action like pressing the mouse button or hitting a MIDI note; execution then flows from the user, to the operating system (through an interrupt), to the application (through an event or signal), to the task (through a procedure call). In this sense, temporal dependencies are completely distinct from functional ones: temporal dependencies provide a dynamic description of a chain of causality that starts at the user and ends in a domino-like triggering of tasks, while the functional dependencies provide a static description of where in memory the tasks should read from and write to when they are executed.

In swish, each task has an invocation procedure that can be executed by sending a command consisting of the name of the task, followed by "invoke." Rather than having a single system-wide data structure containing all of the temporal dependencies

for all of the tasks, some of the dependencies are represented as callbacks for events using Tcl's bind command, while others are implemented through the use of special higher order tasks. For example, the "dataflow" task contains a dependency graph. When invoked, it uses a topological sort [8, pages 485-488] of the graph to invoke each of the tasks in the graph in an order that respects the dependencies. Since this is precisely the behavior that is desired when describing an expression, the dataflow task is used quite frequently. However, other models of temporal behavior could easily be implemented as well.

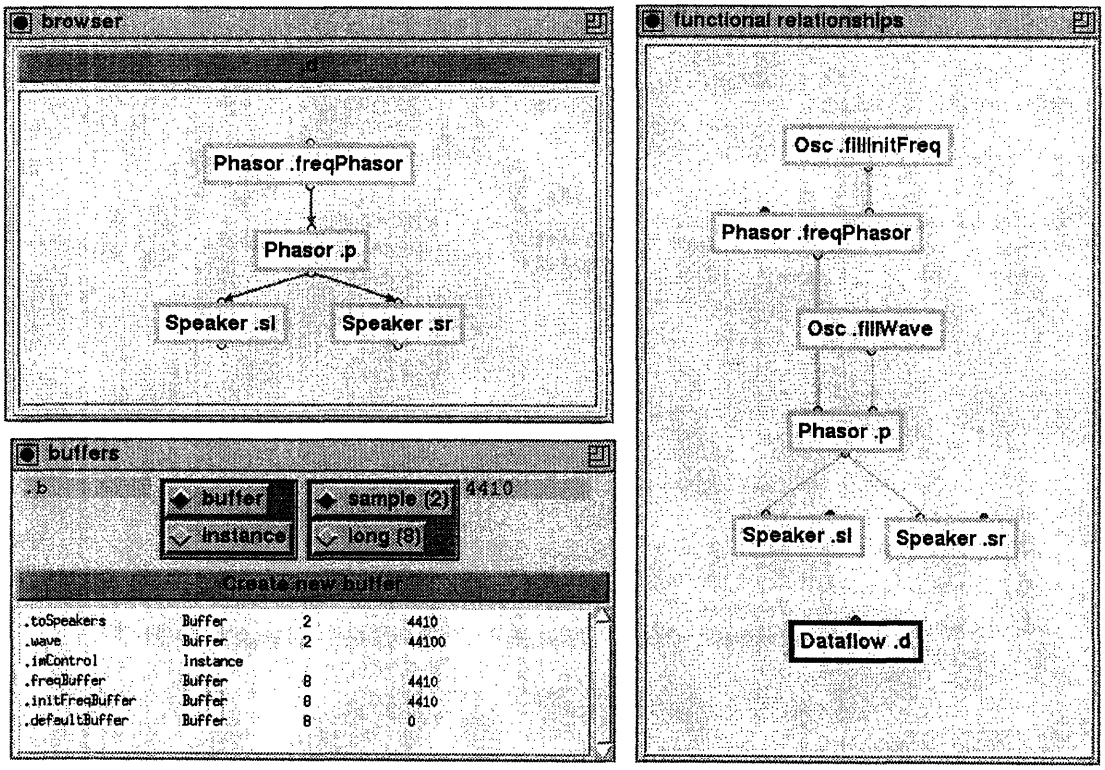

Figure 3-2: The graphical representation of FM synthesis in swish. Note that there are two graphs, one for temporal dependencies, and one for functional dependencies.

For example, a metronome task might invoke an FM task at regular intervals, to update the buffers read by the speakers. Petri nets [19] and Markov models [29, section 5.5] have both been found effective for specifying the algorithmic composition of computer music. Higher order tasks are powerful abstractions, partly because they

are allowed to depend on delays or on user interaction. In swish, each of these examples could be implemented as a task that has other tasks as parameters, and invokes those tasks in a way that exhibits some interesting behavior. Then, we would have three different ways of generating interesting behavior for algorithmic compositions all available in the same environment. This would not be possible in an environment that forced all temporal behavior to be described using the "dataflow assumption." Finally, it would not be hard to extend the current framework, so that higher level tasks could be completely dynamic. By adding tasks themselves to the system as a type, we could create patches that passed tasks themselves around as data. This means, for instance, that the dependency graph for the dataflow task could be treated as an input port, connected to the output of some other task. The dependency graph could then change over time, without requiring any interaction with the user.

The idea of having tasks invoke other tasks is similar to constructs found in other languages. For example, the "apply" node in a dataflow language takes a function and arguments as input, and invokes them. In non-procedural programming languages like ML, "functionals" take functions as inputs, and compose them both functionally and temporally to produce the result [31, chapter 5]. In swish, higher order tasks perform a role similar to both of these, except that they only work in the temporal domain.

A similar abstraction is conceivable in the functional domain. In swish, each task keeps track of its own functional connections. The user sets aside buffers in memory, and then configures the tasks to read from and write to specified buffers. In a more sophisticated system, however, it would be possible to have higher order functional tasks that were responsible for maintaining the functional relationships for a set of tasks. This would give the system a sense of scope. Higher order functional tasks would fill the same roll as structures in imperative languages, meta-widgets in Tcl/Tk, and subpatches in a traditional sound synthesis application.

3.4 A Simple Example

Here, we go through a simple "sine tone" example to show how tasks are created and how dependencies are described. In swish, tasks can be created, represented, and manipulated both graphically and textually. For the sake of simplicity, we present our example initially using the textual representation.

Tasks are created and configured in swish using the same intuitive syntax used to create and configure Tk widgets. This lowers the learning curve, integrates the audio routines into the Tcl/Tk environment very cleanly, and lets the implementor of new tasks take advantage of Tk's support for "parameter configuration."

First, we create one buffer to hold the sine wave, and another buffer to store the input to the speakers:

```
sw_buffer .wave \
    -numBlocks [expr int($sw_samplingRate)] \
    -blockSize $sw_sampleSize

sw_buffer .b \
    -numBlocks $bufferSize \
    -blockSize $sw_sampleSize
```

Next, we create an oscillator task to initialize the wavetable:

```
sw_osc .fillWave -min 0 -max $sw_maxshort -output .wave
```

and a phasor to go through the table:

```
sw_phasor .p \
    -output .b \
    -input .wave \
    -frequency \
    [expr cps2pi(500, $sw_samplingRate, $sw_samplingRate)]
```

and, finally, the speakers:

```
sw_speaker .sl \
    -input .b \
    -update $update \
    -lookAhead $lookahead \
    -device roomLeft

sw_speaker .sr \
    -input .b \
```

```
    -update $update \
    -lookAhead $lookahead \
    -device roomRight
```

Now, our memory system is completely configured. All of the functional dependencies have been described. In particular, the phasor is set up to write to the same buffer that the speakers read from. However, the temporal dependencies have not been specified! In order to do this, we create an apply node, and configure it so that the phasor always executes before the speakers:

```
set d [sw_dataflow .d]
$d add .p {.sl .sr}
```

Finally, we have to create the sine wave:

```
.fillWave invoke
```

Now, if we say

```
.d invoke
```

.p, .sl, and .sr will each be invoked, and we will hear a beep whose length is dependent on the size of the buffers being used and on the sampling rate. A metronome task could be used to keep invoking .d so that the sine tone does not stop.

The value of separating the functional dependencies from the temporal dependencies can be seen even in this simple example. Although many of the functional dependencies also appear as temporal dependencies, note that .fillWave isn't even in the dataflow machine. This makes it possible to fill the wavetable at initialization time, and then not have to refill it every time we want to use it. Conversely, it means that when the wavetable is filled at initialization time, the rest of the tasks do not get invoked. If we had used the dataflow assumption, then the system would have assumed the phasor should be invoked every time that .fillWave is, since .fillWave is writing to the same buffer that the phasor is reading from. The task based model allows us to specify that .fillWave should execute independently of the rest of the tasks, even though it is connected to them functionally.

Tcl-Me hides the existence of the buffers and instances. The programmer specifies connections in terms of the ports associated with tasks, and Tcl-Me automatically creates a correspondence between connections and buffers in memory. Swish is set up

so that every task, buffer, and instance is visible to the user through both a graphical representation and a textual representation. If the above code is typed in using the swish command line, and a graphical representation of the system is active as well, then the graphical representation will be updated automatically as each command is executed. Currently, we use notification procedures to enforce the correspondence. For example, a dataflow task can be given the name of a command to execute every time one of its parameters is changed. Typically, this command will update the graphical representation of the task. A more elegant implementation would raise an event every time a parameter is changed; then, any graphical representation of the task could respond to that event. Since Tcl/Tk does not allow the user to create his own kinds of events, an extension such as userevent [18] could be used. Alternatively, a rule-based system, such as that provided by Rush [38], could provide an elegant solution: We could enforce rules that describe how the graphical representation depends on the state of the tasks.

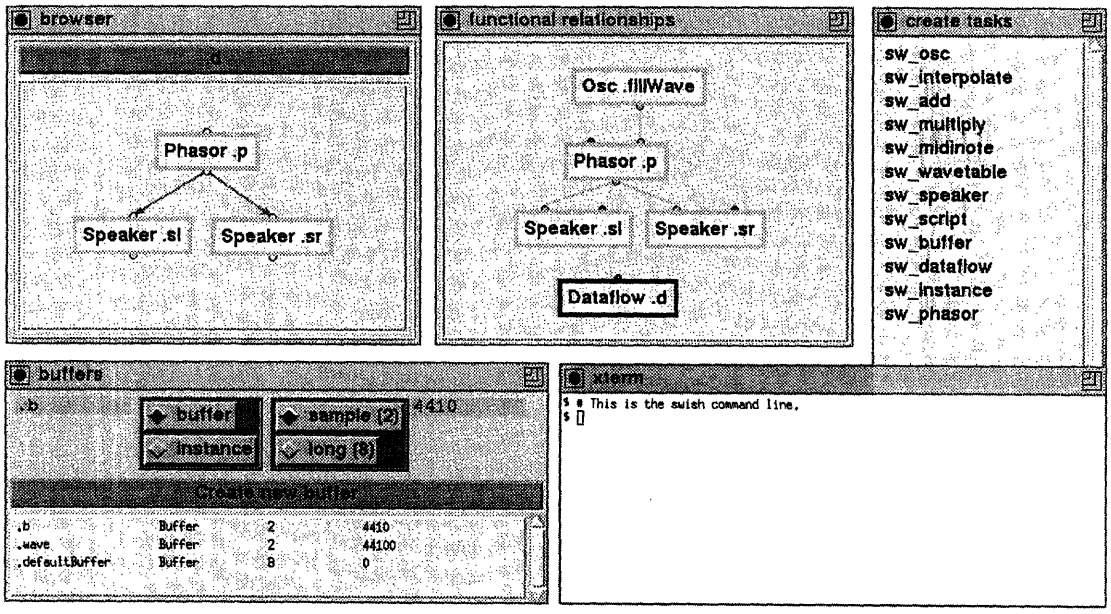

Figure 3-3: The graphical representation of the simple example.

Swish's graphical representation is a prototype of a typical Tcl/Tk browsing and

editing tool. It makes the distinction between types, functional relationships, and temporal relationships extremely clear. In one window, the functional relationships are shown: Each task is displayed in a box, and if the output of a task writes to the same buffer that another task reads from, then a line is drawn. The size and color of the line indicates the type of the data that is being passed. If a task is selected, then another window is opened to show its parameters. In particular, the temporal connections can be viewed in our simple example by selecting the dataflow task, and then looking at the dependency graph. In another window, we have a listing of all of the instances of each type. Each window is completely editable, so that almost every operation that can be done textually through the command line interface can also be done using the graphical interface.

The swish interface illustrates that a program written using the task model can be represented graphically as a Tcl/Tk browser, and can be represented textually using the same interface that is used to represent Tk widgets. Because of this, the audio modules fit very cleanly in the Tcl/Tk environment, and can interact easily with other extensions. To demonstrate, we show how a Tk slider can be used to control the amplitude of the sine wave in our simple example. We will also demonstrate a clean transition from the signal rate to the control rate through the use of an asynchronous interaction: Changing the slider will cause a Tcl variable to change; the same Tcl variable will be the instance that is being read by a multiplication task.

First, we create the slider.

```
set ampControl [sw_instance .ampControl]
pack [fscale .f \
    -variable $ampControl \
    -resolution 0.01 \
    -digits 4 \
    -to 1.0]
```

Next, we add a multiplication task to the system, to scale the signal. Note that the input of the multiplication task is the same variable that is being controlled by the slider.

```
set c [sw_buffer .c \
           -numBlocks [expr int($update * $sw_samplingRate)]\
```

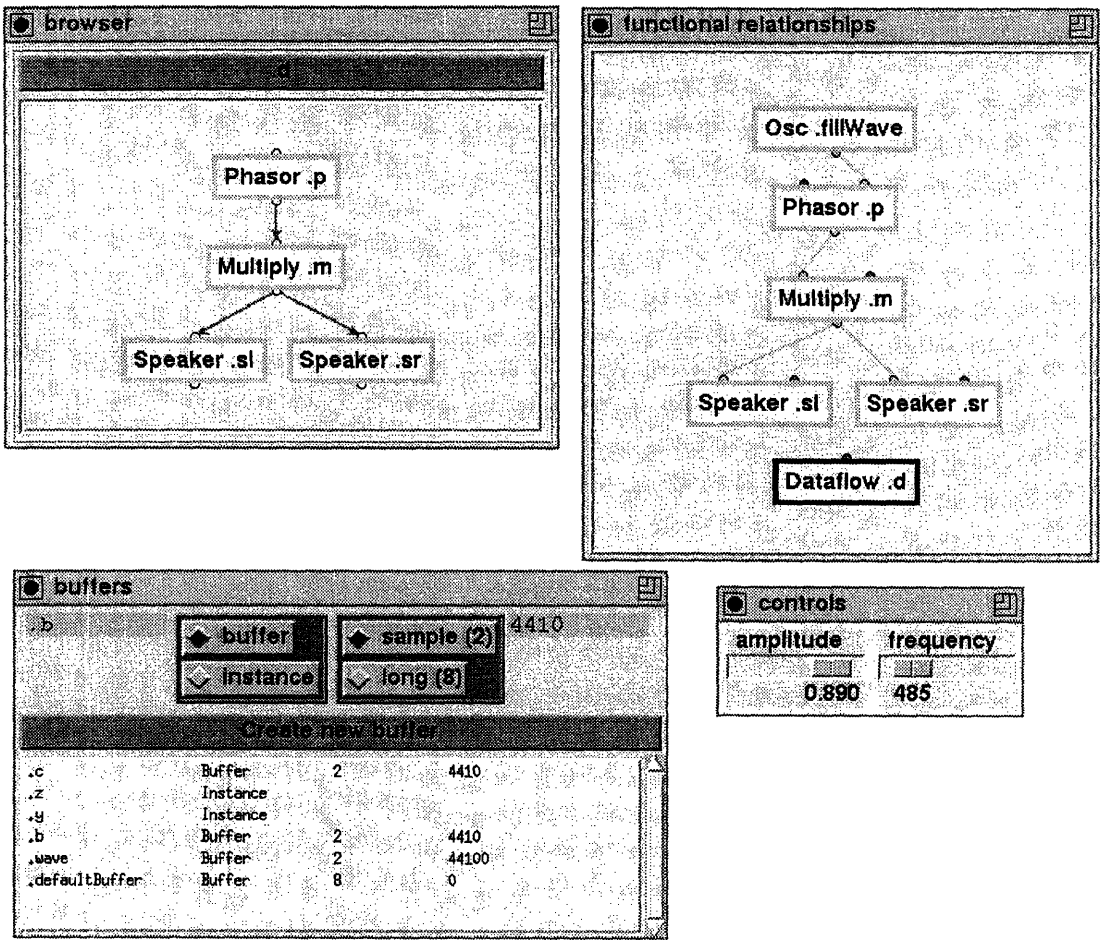

Figure 3-4: Using sliders to control the frequency and amplitude of a sine tone.

```
          -blockSize $sw_sampleSize]
sw_multiply .m -input2Variable $ampControl -output $c -input1 .b
.d delete .p {.sr .sl}
.d add .p .m .m {.sr .sl}
.sl configure -input $c
.sr configure -input $c
```

Any Tcl code executed at this point has immediate, high level access to the amplitude of the sine wave through the global variable $ampControl. The simple fact that both the signals level and the control level code are written using a single interpreter provides a benefit that is missing in many existing packages, where the note level code is written using one interpreter and one syntax, while the signals level code is

written using another.

Chapter 4

Discussion

4.1 Comparison

We argued in the last chapter that modularity was an important principle in the design of swish. Several systems exist and have been discussed that use graphical dataflow machines to describe computation. However, here we argue that the task based model is more powerful than any particular kind of dataflow machine, since the task model can be used to implement them. Furthermore, the particular dataflow machine model that has been implemented in swish allows for a clarity in the graphical representation that is missing in other systems. Finally, swish differs from purely graphical dataflow languages, because it can be manipulated textually as well.

Higher order tasks can be used to implement a variety of different kinds of dataflow machines. For example, the dataflow task that swish currently supports implements a static dataflow machine as efficiently as possible, by performing and caching the result of a topological sort. However, a demand driven dataflow machine could also be implemented, simply by writing a higher order "wrapper" task. The wrapper has another task as one of its parameters. When the wrapper is invoked, it checks to see if all of the input is present for the task it is in charge of; if not, it invokes the tasks that are supposed to provide the data. Once all the input is present, the task can be invoked.

Demand driven dataflow machines are useful, because they provide a graphical

language capable of lazy evaluation. To implement them properly however, some effort must be made to handle the case where a task sends its output to more than one other task. A naive implementation might cause the same computation to be performed more than once, as each output would demand input from the task in question. An efficient implementation would store the result of the computation the first time it was demanded, and have it ready for the future. This "automatic caching" serves the same purpose in a graphical language that assign-once variables do in a textual language.

The other major kind of dataflow machine, data driven machines, can be implemented using a similar structure. This time, the wrapper checks to see if the input is present; if it is, then it first invokes the task that it is in charge of, and then invokes any task that is connected to the output; if not, then it does nothing. Data driven machines correspond naturally to the way that the analog world works. Also, unlike demand driven dataflow machines, they don't tend to build up a large stack before performing any computation.

All of these models are capable of handling expressions well. Given an expression, the precomputed topological sort will be the most efficient, since it does not have the overhead required to determine the order of execution for the tasks at runtime. In this case, each of the dynamic approaches degenerates into an algorithm used to compute a topological sort in linear time. However, more sophisticated programming structures are handled differently by the three models. Conditionals provide a good basis for examining the differences. In particular, if the expression given to a conditional is true, then it is important that the false clause not be evaluated, because the is only going to be thrown out.

With demand driven dataflow machines, conditionals are automatically implemented efficiently. The conditional node simply evaluates the predicate, and only makes a demand on the appropriate input.

Even though the precomputed topological sort task built into swish is static, it can still perform conditionals efficiently. By making tasks a type, a conditional task could be written that has an expression and two other tasks as input, and invokes the

appropriate one after evaluating the expression. This is also typically the solution taken by data driven machines through the use of an apply node.

The advantage to the precomputed topological sort technique is that when you look at a dependency graph, you know that data really does flow through all of it, while when you look at a dependency graph intended for a demand driven machine, it is not clear which parts of it are going to be executed. By bundling up the tasks into "subpatches" that can be fed into apply nodes, the topological sort approach forces the user to divide up the graph into basic blocks. Also, by examining the apply nodes, it is easy to see how the execution stack is going to be built up. With a demand driven approach, however, it is possible to have no subpatches, and have the whole program, control structure and all, in one dependency graph. With text, indentation is used to indicate the conditions under which any particular piece of code is executed; with a dependency graph for a demand driven dataflow machine, however, these clues are missing. Also, in a demand driven system, the execution stack is built up implicitly. These objections could be applied to MAX as well. There, the order of execution depends on the way that the dependency graph is drawn: patches are invoked from right to left. Execution can also be caused by passing "bangs" from one patch to another. Both of these rules obscure the temporal behavior of a patch, simply in order to make it possible to display the behavior of the entire patch using a single graph.

4.2 The Need for Threads

One of the advantages to developing and using your own external hardware to do signal processing is that the question of reliable scheduling can be solved independently, and then built into the design of the chip, which runs continuouly and in parallel, to ensure that any audio patch is executed often enough to keep the output buffers full and avoid glitches. Unfortunately, because of our desire for portability, external hardware was not an option. This forces us to work from within the architecture presented by a contemporary workstation. Here, we discuss some of the options when

trying to ensure glitch-free real-time audio on a Unix machine.

When running a real-time audio process, it is necessary to execute the process often enough to allow the process to refresh the output buffers in order to avoid glitches. One option is to increase the size of the buffers. However, for an interactive application, increasing the buffer size causes delays in response to user input. There is a trade-off between the granularity of response to the user and the amount of time the audio process has when it is invoked, as well as a tradeoff between the delay in response to the user and the frequency with which the audio process must be invoked.

In theory, handling scheduling is the job of the operating system. Ideally, we would like to have one light weight thread for each active patch. That way, we could start up our Tcl interpreter, and every time a metronome task was invoked, we could start a thread. The thread would ask the operating system to call it back at regular intervals. Since the thread is running in the same memory space as the Tcl interpreter, its behavior can still be controlled asynchronously by the user, since the Tcl interpreter will be able to change the values stored in locations in memory that the thread reads its control parameters from. However, since it is an independent thread, it can schedule its own execution; the interpreter doesn't have to worry about invoking it at regular intervals.

Unfortunately, real-time interrupts and multithreading was not part of the original Unix philosophy. The designers clearly felt that threads were unnecessary, since processes could communicate through files (The first sentence of chapter two of [24] proclaims "Everything in the UNIX system is a file."), using pipelines when appropriate, and that the purpose of scheduling was to ensure that system resources were distributed equitably when several users wanted to use the computer at the same time. UNIX has evolved beyond this, and the developing POSIX standards will hopefully turn it into an environment that has complete support for real-time, interactive applications. In the mean time, however, even the POSIX threads on the DEC Alpha don't support per-thread callbacks: Both the operating system and the X windowing system provide per-process callbacks that execute from within the main thread.

This leaves us two options. If we execute each patch in its own process, then

scheduling is easy, but interprocess communication becomes difficult. If we don't, then interprocess communication is trivial, but we still have to address the question of scheduling.

The first option is to create a new process every time a patch is executed. This process executes at regular intervals, modifying its behavior according to the input it receives on some socket. One big advantage to this approach is that patches that could be controlled by sockets could easily be used in a distributed, networked environment. The prime disadvantage is that in order to control a patch through a socket, we would have to create our own protocol for interpreting the data sent across the socket, since we would no longer be able to communicate through Tcl scripts that were executing in the same environment. Depending on how flexible we wanted to be, we could end up creating a complete language for our protocol.

To contrast, the AudioFile server runs in its own process, and responds to requests made by other processes. It implements its own scheduler, so that audio output can be reliably planned for some time in the future. By imagining extending the AudioFile domain to include MIDI streams, or FM synthesis streams, or video playback streams, it becomes clear that the AudioFile server could be expanded to provide scheduling support for a complete multimedia system. AudioFile has, in effect, been forced to implement an abstraction that should really be supplied by the operating system. It has compensated for the lack of a scheduler in the operating system by providing its own. Our second option is to do what AudioFile has done: have only one process, and implement our own scheduler. The process keeps track of all requests for timed callbacks made by any patches. However, it cannot tell the operating system about all of them, because if it did, then the parent thread might be interrupted when it was busy, and because the parent thread must keep track of which interrupt is associated with which patch. Instead, the process waits until it is idle, and then figures out which of the requests is the next one to occur. It schedules an interrupt with the operating system for that time, and go to sleep. When the interrupt occurs, the process will know which patch should be invoked. The problem of handling a set of timed callback requests and dynamically determining which is the next one that is

going to occur is exactly the problem that the operating system is supposed to solve with its scheduler. For musical applications, it is addressed in [10]. Fortunately, it is also part of the functionality of the Tcl/Tk main loop. For now, we rely on the Tcl/Tk main loop to handle the scheduling. But, having an system with three independently written schedulers is clearly inelegant, and will hopefully be addressed by the POSIX standards at some point, through per-thread (and not just per-process) interrupts.

4.3 The Need for Objects

Recently, several systems have proposed using object oriented programming models for computer music projects [22, 33]. These systems make the point that an object oriented system provides a facility for abstraction that is extremely important in an environment that has to support the dynamic creation and naming of any kind of module. Any abstraction, such as a widget or an audio task, that can be said to "exist" in a computer, must have its state represented in memory.

Swish clearly needed support for some kind of object mechanism. From a practical viewpoint, tasks are the bundling together of functions (invocation procedures) with data (parameters), while instances of types (buffers and instances) are just structures in memory. Both are amenable to being implemented as objects.

Since Tcl is not object oriented, many extensions, like swish, that require the dynamic creation of any kind of abstract module must provide their own special-purpose instantiation and control facilities. This accounts for the abundance of Tcl extensions that provide some sort of "meta-widget," "object-oriented," or "namespace" facility. However, having a separate data encapsulation scheme for each extension makes it more difficult for the extensions to work together. Furthermore, Tk does provide support for the mechanisms that it uses to create and configure widgets. Since the Tk "parameter configuration" routines seem to be the only facility for providing data abstraction that has any chance of becoming standard, we decided to use them to instantiate and configure swish objects. There are fairly popular packages that add object oriented facilities to Tcl, but in the long run, none are likely to be as widely

available or as consistently supported as the mechanisms already built into Tk. Furthermore, because we use Tk routines, the syntax for instantiating and configuring swish objects is the same as for Tk widgets. This makes swish tasks easier to implement and cleaner to work with.

Chapter 5

Future Work

5.1 Incremental Compilation

One of the advantages to working in an interpreted environment is that the user has the same power as the developer. Most commercial applications are given to the user as an inflexible, compiled binary. An interpreted language, however, allow the user to configure applications using the same language that the application itself was written in. There is no longer a clear distinction between the role of the developer and the role of the user; both are capable of extending the code at the same level. The user can change the front end using the same tools that the developer used to create it in the first place. Industry is gradually giving users tools and interface agents that allow the user to configure their applications at a high level. As computer literacy increases, people will start to challenge the business driven "developer/user" paradigm, and will demand the ability to write scripts to interact with their applications.

The fact that code written in C can be executed at a lower level than Tcl code introduces a small barrier between the developer and the user. The initial compilation required to incorporate C extensions into the interpreter allows the developer to write code that is executed using the native machine code interpreter, while the user can only write code that is executed using the Tcl REPL (Read-Eval-Print Loop). For example, in swish, audio tasks are implemented by the developer as an extension, because of the need for efficiency. The user has to be able to dynamically combine

these tasks into patches. However, because of the barrier, he can only manipulate them through the Tcl REPL. To handle this barrier, tasks have to be bundled up into procedures that are only available to the Tcl REPL as "black boxes." The resulting patches are not as efficient as if the patches were combined using the same language that they were written in.

At the moment, Tcl scripts and extensions written in C are executed using two different REPL's, one on top of the other: C code is executed by the loop implemented by the hardware that interprets machine code, while Tcl code is evaluated by the Tcl main loop. Code written in Tcl is distinct from code written in C in two ways: It uses Tcl's high level syntax, and it is interpreted using using the Tcl REPL. If we could somehow eliminate the Tcl REPL, then Tcl and C could be thought of as merely two different syntaxes for representing the same language. That way, rather than having code written in Tcl execute on top of code written in C, they would both execute at the same level.

If Tcl had the ability to perform incremental or partial compilations down to native machine code, then Tcl code would execute in the same loop that C code executes in, and Tcl scripts could manipulate tasks much more directly. The tasks created by the developer could be written much more cleanly, and the patches created by the user could be implemented much more efficiently.

In the next two sections, we discuss two specific problems that arose because of Tcl's inability to perform incremental compilation. We then discuss a few languages that have this capability that may become available in the near future.

5.2 The Generalization Phenomenon

In the first chapter of his classic monograph [15, page 4], Dijkstra writes

> "When asked to produce one or more results, it is usual to generalize the problem and to consider these results as specific instances of a wider class."

He goes on to say that the basis for deciding how to generalize a problem should be how easily the correctness of the resulting code can be mathematically verified. In doing so, he under-represents the importance of the question of efficiency. For instance, consider a function isEven(x) that determines if x is even. This is the same as code that determines GCD(2, x). So, is it appropriate to generalize isEven(x) into code that computes the GCD of two integers? It is inelegant to have both isEven(x) and GCD(x, y) in the same system, since they are redundant. If isEven and GCD differ textually by only a small amount of code, then the resulting "cut-and-paste" programming would violate the principle of modularity. However, it is simply very inefficient to compute isEven(x) using GCD(2, x).

This dilemma confronted us repeatedly while designing swish. To illustrate it, we take a closer look at the phasor inner loop.

A phasor task is at the heart of a signal synthesis system. It adds the notion of time or state to a buffer. The code for the phasor in swish is based on code found in Nyquist [11]. In its most basic form, it looks like this: (Recall that long integers are really representations of floats, with SB bits after the decimal point.)

```
register AF_INT16 *inBuf, *outBuf;
 ...
register long wraparound = bufferSize << SB;
register AF_INT16 *bufEnd = outBuf + bufferSize;
register long phaseIncrement = phasorPtr->freq * (1<<SB);
register long phase = phasorPtr->phase;

while (outBuf<bufEnd)
{
  *outBuf++ = inBuf[phase >> SB];
  phase = (phase + phaseIncrement) % wraparound;
}
phasorPtr->phase = phase;
```

This code is fairly straightforward; it just indexes through an input buffer, and writes the result to the output buffer. Note that this code is incredibly compact, and actually does very little on each iteration. This means that even the smallest performance hit in the loop will be a large percentage of the time required to do each iteration. Yet, this code is the inner loop during signal synthesis, and its efficiency

determines where the limit on how sophisticated our interactive, real-time synthesis algorithms can be. It is clearly unacceptable to have any unnecessary code inside this loop at all.

The problem is that this code is not nearly general enough. There are many cases where this is almost what we want to do, but not quite. For example, swish is supposed to support both buffers of 16 bit samples, and buffers of long integers. The code for a phasor that works on buffers of long integers is almost the same as the code written above, except for the declarations of the variables. Also, in the sample code, frequency is a constant. However, if we don't also provide code that can treat frequency as a buffer, then we can't even implement FM synthesis! Once again, the code to do this is very similar to the sample code; we pretty much just have to change "phaseIncrement" to "*phaseIncrement++".

These variations are necessary just to get the most basic functionality out of the system. However, it would be easy to continue to add desirable features by generalizing further. For instance, the buffers can be generalized into delay lines, by turning them into arrays along with an associated index that indicates where the beginning of the buffer is. In fact, the buffers that we are using are themselves a generalization of the buffers found in Nyquist: Nyquist forces the buffers to be a power of 2, so that it can use mask (&) instead of mod (%) in the inner loop when determining if the phase has wrapped around the end of the buffer.

All of the examples of generalization that we have given so far take an input parameter, and make the task work when that parameter has a different or more general type. Many languages address this problem. Some, like ML [31], allow operators to be overloaded, so that they work on many different types. They check the type of a variable at runtime every time it is used. Some object oriented systems would let you define a different phasor for every possible combination of input types. This requires the programmer to write code for a different phasor for each combination of input types. Both of these ideas, and the problems associated with them, are discussed below in a more general context.

We would also like to generalize the functionality of the task in ways that are not based on the types of the input parameters. For example, the sample code determines which sample to use by rounding off to the nearest integer. Many systems, however, provide the option of using interpolation. As another example, the phasor task classically has an amplitude input as well. This could be implemented simply by multiplying the output by the amplitude in the sample code. Once again, the generalization is less efficient but more elegant and more powerful. Note also that many of the generalizations make the original behavior available as a special case.

It should be clear that the issue at hand really is a serious one of modularity versus efficiency, and is not just a question of bit twiddling or implementation issues. Here, we consider the options.

Dijkstra's solution would probably be to forget about efficiency, and write the most concise, most elegant code. For the generalizations that make the original behavior available as a special case, that would mean replacing the special case code with the more general, less efficient code. An example of this is found in the sample code. In Nyquist, all of the buffers have lengths that are a power of two, so that we can use masking instead of mod when determining if the phasor wraps around. The swish code has been generalized, so that it works on buffers of any length. It is still capable of working on buffers whose lengths are a power of two, but when restricted to this kind of buffer, it is less efficient than the Nyquist code. This approach could also be used to generalize any task that is written to operate on scalars into a task that operates on vectors: The original, scalar case can be treated as a vector of length one. However, the resulting code, when restricted to scalar input, will be less efficient than the original code, because of the overhead required to test for the end of the vector. For generalizations that switch between two kinds of similar behavior, Dijkstra's solution forces us to put a conditional inside the loop. For example, if we wanted to allow interpolation, we could write "if (interpolating) *outBuf++ = ...; else *outBuf++ = ..." A conditional can be used also to handle the "mod instead of mask" generalization as well: we could write "if the length of the buffer is a power of two, then use mask; else use mod". This introduces a severe inefficiency because the

test will be performed during each iteration, even though the result of the test will always be the same. However, it is one possible solution.

Well then, why not put the conditionals outside of the for loop? This could be done by making two copies of the loop, one for the original case and one for the generalization, and selecting the proper one. There is a very serious problem with this approach. Before, we simply had to write one conditional for each generalization. Now, every time we have a generalization, the number of loops that we have to write doubles! For example, in order for the phasor to work on both buffers of 16 bit samples and buffers of long integers, it was necessary to write two distinct (but very similar) loops. However, in order to further generalize and make the phasor work both when the frequency is a scalar and when it is a buffer, it was necessary to write a total of four loops. Consider the prospect of writing 125 very similar loops, just so that the "interpolation" task's four inputs and one output can each accept any of swish's three types! This kind of cut-and-paste programming is inelegant and non-modular, and creates code that can't be changed or reused.

Another option is to put the conditionals inside of the for loop, use a really good optimizing compiler, and hope that it figures out that they can be moved outside of the loop. There are several practical problems with this approach. First, we must verify that the compiler that we are using actually performs this optimization. Second, there would be an exponential blow up between the size of the source code and the size of the object code. And, finally, in practice, some generalizations actually can't be written elegantly as conditionals inside of the for loop, because of C's scoping laws. For example, to change the phasor so that it operates on buffers of long integers instead of on buffers of 16 bit samples, it is necessary to change each of the declarations in the sample code from AF_INT16 to long. This particular generalization clearly forces us to have separate, almost identical loops for each case.

Unfortunately, there seems to be no other options when working in a traditional programming environment. Most systems, like csound, Nyquist, or swish, handle the situation through a combination of these possibilities, writing compact code for all but the most critical cases. Swish, for instance, uses five very similar loops to implement

the phasor, seven for interpolation, and five for multiplication. Fortunately, the problem is with the limitations imposed on us by our programming environment, and not one inherent with computation. We investigate some solutions based on changing our programming model after describing how a very similar problem arises when trying to execute user defined patches.

5.3 Implementing Higher Order Tasks

The factor that determines if a particular synthesis algorithm described by a swish patch can be performed in real time is the efficiency with which it is executed. As we have seen, each task in a patch performs very fast iterations over elements of a buffer. The reason that efficiency is even an issue is because of the number of iterations that must be performed to produce 44,100 16 bit samples each second, and not because of the length of each iteration. In the last section, we described some of the difficulties that are encountered when trying to make the code inside the loop as efficient as possible. Here, we describe efficiency issues that arise when trying to glue loops together to execute a patch. In particular, we show that we are taking a large performance hit because the only way that we can manipulate these loops is to execute one after the other – we don't have access to the internal structure of each loop, so we can't combine them at a low enough level to take advantage of our machine's architecture.

To illustrate the problem, we examine the way that the higher order dataflow task operates. Recall that this task performs the topological sort needed to create patches that compute expressions.

The dataflow task maintains a dependency graph for a collection of other tasks. Based on a "lazy evaluation" philosophy, it only performs a topological sort when it has to, but when it does, it stores the result away for the next time it is invoked. This means that the first time it is invoked, the dataflow task performs a topological sort on the graph, and stores the resulting ordered list of tasks in a "cache." From then on, every time it is invoked, the dataflow task uses the cached list to invoke the

tasks in an order that respects their temporal dependencies.

The process of computing a topological sort and then caching the result away as a list of tasks can be thought of as a compilation process. The dependency graph specifies a program that is to be executed, while the cache specifies the same program at a lower level that is easier for the machine to understand. Taking this line of thought further, one could easily imagine the compilation process continuing all the way down to the native assembly code. Unfortunately, because the invocation procedures for the tasks that the dataflow task invokes are "black boxes," it is not possible for the compilation process to go any further than specifying the order in which tasks can be invoked. This is a large source of inefficiency. Not only is there the overhead required to interpret the instructions in the cache, but we are missing out on many important, very basic compiler optimizations. For example, a typical "pipeline" patch consisting of a series of tasks might do something like this when executed:

```
#define A(x) ...
#define B(x) ...
...
#define C(x) ...

for (i = 0; i<bufferSize; i++)
    aOutput[i] = A(input[i]);
for (i = 0; i<bufferSize; i++)
    bOutput[i] = B(aOutput[i]);
...
for (i = 0; i<bufferSize; i++)
    cOutput[i] = c(bOutput[i]);
```

Clearly, a much more efficient solution would be:

```
for (i = 0; i< bufferSize; i++)
    {
        aOutput[i] = A(input[i]);
        bOutput[i] = B(aOutput[i]);
        ...
        cOutput[i] = C(bOutput[i]);
    }
```

This is a large improvement, just because of the overhead associated with performing the extra loops. We also have the more subtle, but just as substantial improvement

of being able to keep the intermediate values of our computation in the cache, if not in registers. Rather than computing the first phase of each computation and then storing the results away in buffers in main memory until we are ready to compute the second phase, the improved code follows a computation all the way through, keeping the relevant variables in registers until it is finished with them.

Unfortunately, we can't use the improved code, because there is no way to dynamically create it at run time. Even if we had access to A, B, and C as functions or as cases in a switch statement, the extra overhead required to perform the function call or switch test at every iteration would cancel out any improvement in efficiency. Basically, all this would do is execute "for each sample, for each task, invoke task" instead of "for each task, for each sample, invoke task."

We conclude, therefore, that there is no way to design swish so that its patches are executed as efficiently as if they were implemented in C, because swish can only manipulate tasks as black box procedures, and can not generate native assembly code.

5.4 Solutions

It should be clear that the ability to dynamically generate code would solve both these problems, and would make swish much more modular and efficient. First, the internals of each task could be implemented more efficiently and more cleanly. Currently, each task has several sections of very similar code. When the task is invoked, one of these sections is executed. With dynamic compilation, tasks could make themselves available to higher order tasks through procedures that emit the code that should be executed when the task is invoked, instead of through "invocation procedures." These procedures could be written very concisely and cleanly: For every generalization, we would have one conditional. For example, to support both buffers of 16 bit samples and buffers of long integers in the phasor example, the code could start out with "if (longIntegers) emit("register long *inBuf, *outBuf"); else emit("register AF_INT16 *inBuf, *outBuf");". Furthermore, executing the emitted code would be more efficient than executing the loops that the tasks are currently implemented with.

Second, simple optimizations could be performed on the emitted code to transform the structure of patches from the "sequence of black boxes" to the improved code, as described in the last section.

At the moment, the only interpreted, free, and portable languages that support dynamic compilation are functional ones created by the programming languages research community, like ML [31], Haskell, or Scheme. It might be possible to add dynamic compilation to the Tcl/Tk environment in a way that maintains portability across architectures, by creating a Tcl interface to the SMLNJ back end. Then, tasks could emit intermediate code, and the SMLNJ back end would compile it down to native machine code, and optimize it. Although SMLNJ was not available on the platform in question when we started the implementation process, this remains an intriquing option for the future.

Partial compilation is another, related language feature that could be used to implement tasks in a more efficient and modular manner. To implement a phasor task that takes advantage of partial compilation, we would write a function that had, as arguments, in addition to the locations of the input, output, and frequency buffers, parameters that indicated whether or not we should perform interpolation, whether the input buffers and output buffers are instances, buffers of 16 bit samples, or buffers of long integers, whether the buffers have lengths that are powers of two, and so on. This function could be written in a concise style, without regard for efficiency. Then, the first time a task was invoked, we could create a new function that had all of these arguments filled in, and the language would automatically compile and optimize the new function. Similarly, tasks could be combined by creating a new function that invoked one task after the other; When it defined the new function, the language would automatically compile and optimize it, so that the "straightforward" algorithm described in the last section would be replaced with the "improved" algorithm. Once again, the only interpreted, free, and portable languages that support partial compilation are functional ones.

When the project started, there were very few functional languages that were freely available for the Alpha. A more fundamental problem with using functional

languages is that they traditionally have had difficulty specifying I/O, user interaction, and side effects. However, because of their ability to handle functions as first class objects, support for lazy evaluation, and ability to dynamically compile down to native assembly code, they have recently received attention by people developing systems that, like swish, have to handle the concept of time in an elegant manner. [2] describes a project that uses Haskell for computer animation. [36] is the beginnings of a visual interface on top of Haskell, much in the same way that swish provides a graphical interface to the tasks in swish. Nyquist [11] is written in xlisp, and emphasizes the utility of lazy evaluation. Finally, Common Music [42] adds support for objects that represent musical abstractions in CLOS (Common Lisp Object System).

However, there are several important mechanisms built into Tcl that simplify applications that use side effects that are not supported by these languages. First, Tcl makes the current state of the interpreter available to scripts. For example, it is possible to determine all currently defined procedures, the parameters for any given widget, or the names of all global variables. This encourages the development of graphical browsers and interface builders (like the prototype task browser and editor that swish provides). Second, variable traces make efficient asynchronous interaction possible, and were used in swish to ease the transition from the signals level to the note level. Third, Tcl supports the use of tags with canvas items. The concept of tags has potential for use with audio tasks, as well. Tags can be associated with instances or buffers, making it possible for a process to set the value of all variables whose tags satisfy some predicate. Tags have already been found to be useful in computer music systems [12]. A variable can receive a tag because of its lexical properties (where it lies in the code), or because of its dynamic properties (what caused the variable to be instantiated). They can indicate which task a variable is associated with ("I am a parameter associated with a guitar synthesizer"), which parameter of a task the global variable is ("I am a pitch bend parameter"), when the task was created ("I am the fourth note in channel 3 in the notelist set off in the second section"), or why the task was invoked ("I was activated because the user pressed MIDI note C3.") We could extend the tag paradigm even further by suggesting the use of a database of active

widgets, tasks, and parameters. Finally, functional languages are only beginning to support complete widgets sets like Tk at a high level. Without the ability to change state in an imperative manner, however, it is not clear how to provide an interface to a widget like a slider. In Tcl, the slider is implemented by maintaining a widget record located at a fixed position in memory. Every time the user moves the slider, the widget record is updated. This is a very natural way to represent the slider, and it is not clear how it could be as elegantly implemented in a truly functional language, without having to allocate a new widget record every time the slider moved. A proponent of functional languages may argue that recklessly changing the state of the widget record may cause problems because that widget record may "belong" to some other part of the system. However, Tcl's variable tracing facility compensates for this by allowing a component of an application to monitor changes made to a Tcl variable. Before real-time interaction can be specified in a functional language, these issues must be seriously addressed.

One final possibility is a dialect of Tcl/Tk currently under development, called Rush [38]. Rush gives us the same interpreted, imperative approach that Tcl/Tk has shown is extraordinarily powerful when designing applications that interact with the user in real time, but adds to it several of the features found in functional languages, including a sophisticated rule-based system (a powerful replacement for variable traces), lexical scoping, closures, types, and the ability to compile (although it is not known if this will be dynamic). This is a potentially powerful combination of features. First, having a rule-based system in an imperative language gives the programmer the data protection that functional languages make available through assign-once variables. Rules could be used to model the MIDI note abstraction accurately, by allowing the user to create and configure notes using arbitrary parameter values, but then using rules to enforce the restrictions imposed on note parameters by the MIDI standard. For example, a rule could be used to force the pitch bend parameters associated with the notes on any particular channel to be the same. Any time one of the pitch bends changed, the rule would change the pitch bend value for all of them. Second, the absence of a typing system in Tcl forces the parameter configura-

tion routines to superficially create a special purpose convention for specifying types inside of widgets and tasks. One problem that this causes is that typing information is not available at the script level. Rush provides a consistent type system. This could be used to automatically generate graphical interfaces. For example, if the user is examining an integer, then the computer might produce a slider. However, if the user decides to look at a dataflow task, then the system would be able to figure out, solely from the typing information, that it should represent this particular task using a dependency graph. This is a principle already used in existing editors/browsers like [37], but it could be done much more elegantly if types were actually built in to Tcl. [36] proves this point, and starts raising questions about the right way to represent more complicated types. This is an extremely intriquing area. Some interesting questions include: What is the proper way to graphically represent an object whose type is the Cartesian product of two other types (such as int * int), and how can types be used to aid in the graphical creation of a function whose type is known in advance. Rush should make it possible to begin investigating these issues, by giving us the data abstraction capabilities of a functional language, but the basic programming style of an imperative language, along with all of the features that make Tcl so appropriate for real-time interaction.

Chapter 6

Conclusion

The development and expansion of the world wide web in recent years has demonstrated that generally available desktop computers are capable of the computation and communications bandwidth necessary to support distributed multimedia applications, and that typical workstation users are literate enough to use the technology and are willing to support the growth of standards that allow people to interact through a variety of media, including graphics, text, hypertext, animation, video, and audio. The computer music community is very aware of these developments, and is responding with further research into representations of music appropriate for real-time, networked installations [46]. The HyTime/SMDL standards [41] being considered by the American National Standards Institute may provide the SGML (standard generalized markup language) that will allow note-level descriptions of music to be passed around on the internet through textual documents. Prof. Machover has recently proposed the Virtual Brain Opera, an extensive, networked, musical installation to be made available on the internet [27]. The computer music community already has an extensive history of interest in real-time interaction with audio processes (see, for instance, [26]); the transformation of the computer music community from that small group of people with access to the required special purpose hardware to the global community of workstation users connected through the internet will reemphasize the need for environments that are not only capable of performing real-time audio with no special purpose hardware, but are also easily installed, extensible, and, most of all, portable.

In particular, there are three commonly proposed ways to extend the domains that a network, such as the internet, can operate over; it is our contention that each option calls for an interpreted, portable, extensible, high level language.

The first option is to extend the browsers used to inspect the data being distributed over the network. For example, documents that describe music could be distributed over a large network if everybody that wanted to inspect musical data installed a program on their machine that interpreted a textual representation of music. The browser could then be configured to invoke that program every time a document in the new format was encountered. This option clearly requires the development of a program that can be installed by anybody who wants to have access to the new formats, so portability and ease of installation becomes a real concern. It also assumes that the browser is, in some sense, extensible. Perhaps the most serious drawback for this scheme is the simple requirement that each person that wants to examine data that represents music must be willing to take the time, disk space, and security risk needed to install this new program on his system.

The second option is a client/server approach. To create a new standard, you set up a server process on your own machine. Other machines can communicate with this process using a standard network protocol, and give it any documents that have to be interpreted. The process can then respond using another standard client/server protocol. It might use, for example, the X window protocol to respond with an image, or the AudioFile protocol to send back an audio interpretation of the data. The advantage to this is that only one person (the person who designed the representation of music) has to install the new software. Furthermore, he has complete control over the software, and can update it at any time. The disadvantage is that all of the computation takes place on this one machine, creating a potentially serious bottleneck both in terms of computation and in terms of communications bandwidth. Here, we see that an application is going to have to be written that is capable of handling several network protocols, plus document parsing, plus, in our example, musical abstractions. Clearly, this application calls for the modularity provided by an interpreted, high level, extensible environment.

The final option, and perhaps the most intriquing one, describes a network that depends on using rpc's (remote procedure calls) to interpret pieces of code, rather than clients and servers that pass data to each other in some previously arranged, hard-coded, domain-dependent format. At the 1994 Tcl/Tk workshop, Prof. Ousterhout described a network where processes passed flexible and dynamic pieces of code to each other, rather than passing static documents that are known to be in some arbitrary, predetermined format. Implementing this scheme would require the cooperation of the networked community as a whole; however, once it was in place, anybody could then extend the domains that the network could operate in. Clearly, the main hurdle here is security. Equally clear is the importance of portability, extensibility, and availability for the interpreted language. Replacing the transmission of domain-specific documents with the transmission of code to be executed in a secured environment is similar to the way that PostScript has superceded ASCII for the transmission of text to printers, and would make the multimedia environment a more flexible and dynamic arena.

These possibilities for enabling audio to become an integral part of a networked world all involve the development of a portable development environment. The main difference lies in the question of which machines have to execute the application.

Within the next few years, the programming systems community will determine the nature of multimedia interaction over the internet. This decision will be more of a communal settlement into some commonly utilized solution rather than a well thought out, deliberate decision made by a particular person or agency. The result will be based on the popularity and general acceptance of specific portable and available applications that actually exist, rather than on the most technically or musically appropriate solution. It is essential, therefore, that the computer music community begin to develop an environment that makes musical constructs accessible to the typical workstation user, so that an effective approach that reveals the artistic aspects of audio will become part of the standard interaction between people and their computers, rather than one that simply adds audio as a thin layer in between the user and already existing applications like news readers and electronic mail processors.

Bibliography

[1] Arvind and David E. Culler. Dataflow architectures. LCS 294, MIT, 1996.

[2] Kavi Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, January 1994.

[3] M. R. Barbacci and J. M. Wing. Specifying functional and timing behavior for real-time applications. Technical Report 177, Carnegie Mellon University, 1986.

[4] Mario R. Barbacci, Dennis L. Doubleday, Charles B. Weinstock, Steven L. Baur, David C. Bixler, and Michael T. Heins. Command, control, communications, and intelligence node: A Durra application example. SEI 9, Carnegie Mellon University, 1989.

[5] John A. Bate. Unison – a real-time interactive system for digital sound synthesis. In *Proceedings of the International Computer Music Conference*, 1990.

[6] Micah Beck and Kesav Pingali. From control flow to dataflow. Technical Report 1050, MIT, 1989.

[7] A. Biancardi and A. Rubini. Non-string data handling in Tcl/Tk. In *Proceedings of the Tcl/Tk Workshop*, pages 171–175, 1994.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 1991.

[9] R. B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Dennis Baggi,

editor, *Readings in Computer-Generated Music*, pages 25–40. IEEE Computer Society Press, 1992.

[10] Roger Dannenberg. Real-time scheduling and computer accompaniment. In Max V. Mathews and John R. Pierce, editors, *Current Directions in Computer Music Research*, pages 225–261. MIT Press, 1989.

[11] Roger B. Dannenberg. Nyquist.
WWW URL file://g.gp.cs.cmu.edu/usr/rbd/public/nyquist.

[12] Roger B. Dannenberg. Music representation issues, techniques, and systems. *Computer Music Journal*, 17(3):20–30, Fall 1993.

[13] Roger B. Dannenberg, Dean Rubine, and Tom Neuendorffer. The resource-instance model of music representation. In *Proceedings of the International Computer Music Conference*, 1991.

[14] Sven Delmas. Xf: Design and implementation of a programming environment for interactive construction of graphical user interfaces.
WWW URL file://harbor.ecn.purdue.edu/pub/code/xf-doc-us.ps.gz.

[15] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[16] Patrick Duval and Tie Liao. Tcl-Me, a Tcl Multimedia Extension. In *Proceedings of the Tcl/Tk Workshop*, pages 91–96, 1994.

[17] John V. Guttag, James J. Horning, et al. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[18] Michael Halle. Tk userevent 0.95.
WWW URL file://harbor.ecn.purdue.edu/pub/extensions/uevent-0.95.tar.gz.

[19] Goffredo Haus and Alberto Sametti. Scoresynth: A system for the synthesis of music scores based on Petri nets and a music algebra. In Dennis Baggi, editor, *Readings in Computer-Generated Music*, pages 53–77. IEEE Computer Society Press, 1992.

[20] S. R. Holtzman. Using generative grammars for music composition. *Computer Music Journal*, 5(1):51–64, Spring 1981.

[21] Andrew Horner, James Beauchamp, and Lippold Haken. Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis. *Computer Music Journal*, 17(4):17–29, Winter 1993.

[22] David Jaffe and Lee Boynton. An overview of the Sound and Music Kits for the NeXT computer. In Stephen Travis Pope, editor, *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*, pages 107–115. The MIT Press, 1991.

[23] Margaret L. Johnson. An expert system for the articulation of Bach fugue melodies. In Dennis Baggi, editor, *Readings in Computer-Generated Music*, pages 41–51. IEEE Computer Society Press, 1992.

[24] Brain W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, 1984.

[25] Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, and Lawrence C. Stewart. AudioFile: A network-transparent system for distributed audio applications. In *Proceedings of the USENIX Summer Conference*, June 1993.

[26] Tod Machover. Hyperinstruments: A progress report 1987-1991. MIT Media Lab, January 1992.

[27] Tod Machover. Brain Opera and Virtual Brain Opera. MIT Media Lab, August 1994.

[28] Michael Minnick. A graphical editor for building unit generator patches. In *Proceedings of the International Computer Music Conference*, 1990.

[29] F. Richard Moore. *Elements of Computer Music.* Prentice-Hall, 1990.

[30] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[31] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[32] Andrew Payne. Ak.
WWW URL file://crl.dec.com/pub/misc.

[33] Stephen Travis Pope. The Interim DynaPiano: An integrated computer tool and instrument for composers. *Computer Music Journal*, 16(3):1–23, Fall 1992.

[34] Stephen Travis Pope. Machine tongues XV: Three packages for software sound synthesis. *Computer Music Journal*, 17(2), Summer 1993.

[35] Miller Puckette. Combining event and signal processing in the MAX graphical programming environment. *Computer Music Journal*, 15(3):68–77, Fall 1991.

[36] H. John Reekie. Visual Haskell: A first attempt. Technical Report 94.5, University of Technology, Sidney, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, PO Box 123, Boadway NSW 2007, Australia, August 1994.

[37] David Richardson. Interactively configuring Tk-based applications. In *Proceedings of the Tcl/Tk Workshop*, pages 21–22, 1994.

[38] Adam Sah, Jon Blow, and Brian Dennis. An introduction to the Rush language. In *Proceedings of the Tcl/Tk Workshop*, pages 105–116, 1994.

[39] John A. Sharp. *Data Flow Computing.* Ellis Horwood Limited, 1985.

[40] Sam Shen. tkInspect.
WWW URL file://harbor.ecn.purdue.edu/pub/code/tkinspect-4d.README.

[41] Donald Sloan. Aspects of music representation in HyTime/SMDL. *Computer Music Journal*, 17(4):51–59, Winter 1993.

[42] Heinrich Taube. Common Music: A music composition language in Common Lisp and CLOS. *Computer Music Journal*, 15(2):21–32, Summer 1991.

[43] D. L. Tennenhouse, J. Adam, D. Carver, H. Houh, M. Ismert, C. Lindblad, W. Stasior, D. Weatherall, D. Bacher, and T. Chang. A software-oriented approach to the design of media processing environments. In *Proceedings of the International Conference on Multimedia Computing and Systems*, May 1994.

[44] Barry Vercoe. *Csound: A Manual for the Audio Processing System and Supporting Programs.* MIT Media Lab, 1986.

[45] Geraint Wiggins, Eduardo Miranda, Alan Samill, and Mitch Harris. A framework for the evaluation of music representation systems. *Computer Music Journal*, 17(3):31–42, Fall 1993.

[46] Michael Daniel Wu. Responsive sound surfaces. Master of science in media arts and technology, Massachusetts Institute of Technology, September 1994.