Models of Time in Audio Processing Environments

by

Ivan Neil Burroughs
B.Sc. University of Victoria 2006

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Ivan Neil Burroughs, 2008
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
Models of Time in Audio Processing Environments

by

Ivan Neil Burroughs
B.Sc. University of Victoria 2006

Supervisory Committee

Dr. George Tzanetakis, Co-Supervisor
(Department of Computer Science)

Dr. Nigel Horspool, Co-Supervisor
(Department of Computer Science)

Dr. William Wadge, Departmental Member
(Department of Computer Science)

Dr. Peter Driessen, External Examiner
(Department of Electrical and Computer Engineering, University of Victoria)
Abstract

Time has always been a parameter to minimize in computer programs. It is the stuff that measures our patience as we wait for results. However, for a number of problems, we seek to model a notion of time that can be used to regulate the rate at which things happen. Audio processing is one of these problem areas. It has seen the development of many languages and environments with each one having to adopt a suitable notion of time to support such things as accurately timed events and interactivity while remaining efficient.

In this thesis I will investigate the forms of simulated time within audio processing environments. To this end, I will define a set of properties that shape the construction of a model of time simulated on a computer. We can see these properties in the design of languages and environments that support the scheduling of events. With that in mind, I will provide a survey of the use of time in a number of computer languages and paradigms. The reach of this survey will not be exhaustive but will instead try to investigate different ideas with an emphasis on languages for audio processing. I will also put some of these ideas into practice by presenting two separate audio processing frameworks each with their own model of time.
# Table of Contents

Supervisory Committee ...........................................  ii  
Abstract ................................................................. iii  
Table of Contents ................................................... iv  
List of Figures ......................................................... vi  
Acknowledgements .................................................... viii  

1. Introduction ....................................................... 1  
   1.1 Thesis Overview ................................................ 2  

2. Models of Time in Computation ................................. 3  
   2.1 Clock ................................................................. 3  
   2.2 Event ................................................................. 4  
   2.3 Properties of Time ............................................... 5  
   2.4 Summary ............................................................. 11  

3. Time in Programming Languages ............................... 12  
   3.1 Data containing Time information ............................. 14  
   3.2 Methods for Controlling Time .................................. 15  
   3.3 Scheduling .......................................................... 16  
   3.4 Time Based Programming Languages ......................... 17  
   3.5 Audio Programming Languages .................................. 23  
   3.6 Summary ............................................................ 31
# List of Figures

3.1 Iteratively summing the elements of an array. ........................................ 14
3.2 A simplified scheduler dispatch loop. .................................................. 15
3.3 Definition of a function that generates the powers of 2. .......................... 18
3.4 Using the `whenever` operator to filter every other element of f. ............... 19
3.5 Timing table for the function output of Figure 3.4. ................................. 19
3.6 A sequential function definition for summing values in the stream x. ......... 19
3.7 Using the when operator to change the rate at which values of x are sampled. 20
3.8 Esterel statement for reacting to a button press. ..................................... 21
3.9 Esterel statement to sound a siren in response to an event. ...................... 22
3.10 Time is a behavior. .............................................................................. 22
3.11 FRAn function for cycling colours on left button presses[12]. .................... 23
3.12 CSound Orchestra Specification .............................................................. 24
3.13 CSound Score Specification .................................................................. 25
3.14 Chronic temporal type describing a chord progression[4]. ....................... 25
3.15 Chronic event vec representations. ....................................................... 26
3.16 ChucK program .................................................................................... 27
3.17 Signal analysis in ChucK ................................................................. 28
3.18 Pure Data graphs: a) delaying messages, b) filtering messages. ............... 29
3.19 The outline of a basic MarSystem class. ............................................... 30
3.20 Series and Fanout composite MarSystems .......................................... 31
3.21 Marsyas Example for creating an output file with a loudness of half that of the input file ............................................ 32
4.1 Controls modified during buffer processing may become inconsistent between MarSystems. ................................................. 36
4.2 Timing comparison of repeating event (edges), (A) intended onset timing, and (B) quantized on buffer boundaries. .................. 37
4.3 Marsyas Scheduler class diagram. ........................................ 37
4.4 UML Sequence diagram showing function calls involved in event dispatch. ......................................................... 38
4.5 Event Interface. .......................................................... 39
4.6 Timer Interface. .......................................................... 40
4.7 Scheduler Interface. ...................................................... 41
4.8 Virtual Scheduler Interface. ............................................. 42
4.9 Network to play random sinewaves. .................................... 46
4.10 A network that plays notes on each peak detected. ............... 48
4.11 Marsyas Event to set the frequency of a SineSource using a sequential list of frequencies. ........................................... 49
5.1 Marsyas network creation and control .................................. 53
5.2 Linking two controls .................................................... 55
5.3 Update wrapper function ............................................... 56
5.4 MarsyasOCaml Gain Module .......................................... 58
5.5 MarsyasOCaml Network Construction ................................. 59
5.6 A possible MarsyasOCaml Scheduler function ....................... 60
Acknowledgements

I would like to thank my supervisors George Tzanetakis and Nigel Horspool who have supported me over the course of this work. My research at UVic and this thesis benefitted from both an NSERC PGS-M scholarship and a UVic Fellowship. I am grateful to these institutions for their support.
Chapter 1

Introduction

Time is an inescapable fact in our lives. Our very existence is defined by it. Our thoughts, our actions, our language all have an inherent time based component to them. Computers, whose discovery and construction are born from our thoughts and actions, are naturally governed by the very same temporal phenomena. Computer programs take time to run. Because of this a great deal of effort has gone into ways to optimize the running time of algorithms so as to reduce the amount of time required to obtain the desired results.

For an entirely different class of problems time is not simply a parameter to be optimized but is an integral part of the computation. Results are expected to be produced at particular points in time rather than in a minimum amount of time. For these problems a specific notion of time must be defined that suits the requirements of the application.

Since the range of applications requiring the timed management of results is quite large, this work has concentrated on the area of audio processing. Audio processing deals with the manipulation or analysis of sound over time. Sound analysis applications attempt to determine features of the original sound source in an attempt to answer such questions as whose voice is talking, what musical instrument is being played, or what is the rate of the musical notes being played. Sound synthesis is the process of generating sound by combining different electronic signals. Synthesis is not only used to create new sounds but also to emulate existing sounds like musical instruments or the human voice.
1.1 Thesis Overview

This thesis begins in Chapter 2 with definitions for the important terms *clock* and *event*. It will then discuss four temporal properties that will play a part in shaping a suitable notion of time for a given application area. The mechanics of time has been the subject of great debate for thousands of years. Fortunately for this thesis, there is no need to discover the real properties of time. Instead, these properties will apply to notions of computer-simulated time and will therefore be bounded by the limits of a computer running in real time.

Various notions of time have been developed for use in different programming language environments. Chapter 3 will present an overview of some of these environments and how time may be manipulated in each. The groundwork will be set with a look at general purpose languages for which no notion of time has been built in. Developers requiring actions to be performed with respect to time must build their own structures for this task. A discussion of these structures, usually called schedulers, will also be presented. For the remainder of the chapter, a number of languages and environments that support the scheduling of events with respect to time will be presented.

Chapters 4 and 5 will present two systems for controlling tasks with respect to time in an audio processing environment. Developing a suitable notion of time for these environments will have a direct impact on the types of actions that can be performed and on the amount of control the system will have over their management. Chapter 4 describes a scheduling system designed for an existing audio processing framework. The resultant design incorporates a flexible notion of time based on the audio sample rate that allows for user-definable time references. Chapter 5 describes a new audio processing framework built from the ground up to include its own notion of time. It employs a powerful reactive system for propagating environment parameters. Within this system any parameter can be used to dispatch events.
Chapter 2
Models of Time in Computation

What, then, is time? If no one asks me, I know what it is. If I wish to explain it to him who asks me, I do not know.

Saint Augustine, Confessions[1]

Time is a basic truth that governs our lives and yet is hard to define. It seems to be beyond the grasp of our minds yet its effect on our lives is undeniable. It allows us to reason about our existence by forcing order on the events that affect us. The notions of before and after and cause and effect are only definable in terms of time. They help us to make judgements about future decisions through evaluation of the past. Time is something to be used wisely or wasted yet is also beyond our control as it flows forward without regards to how we make use of it.

In this chapter the notion of time will be explored through the discovery of its properties in the context of time based computation. Clocks and events, the basic components of any discussion about time, will be explained as well. While this work is primarily concerned with the area of audio processing it will prove helpful to include other areas in this investigation where time is an important parameter.

2.1 Clock

Definition 1 (Clock). something that defines the rate of change of a particular time reference.
Wall clocks and wristwatches diligently count the seconds, minutes, and hours that tick by during the day. The standard time reference for these devices has been developed through observation of the day/night cycle. Of course, this is not the only possible time reference.

In a computer application time references can be developed to suit the task. For audio applications the audio sample rate is generally used as the main time reference. Usually, 44.1kHz is used as the sample rate which translates to a clock tick every 23μsecs. Time references need not be synchronized to the standard clock. Orchestra conductors define a time reference through the motions of the baton. If a computer could be made to sense these motions then the baton could be used as the reference for a clock.

The terms ‘Timer’ and ‘Duration’ often find their way into discussions about time. The term ‘Timer’ is often used in place of the term ‘Clock’. In some cases, a clock represents the time reference while a timer is a particular instance of that reference with the same rate but having a starting point. The idea of ‘duration’ is often used when talking about time, particularly when discussing an amount of time that has passed. It is duration that is measured by a timer - the amount of time that has passed since the timer started counting.

2.2 Event

Definition 2 (Event). *something that happens at a point in time.*

An event may result in a change in state or it may not. Consider a Mute event that causes a speaker playing music to become silent. This would be considered a change in state due to the lasting effect. A change in state means that some variable or system switch, under program control, has been set to a value that is different than the value it was previously set to. In order to hear music again an UnMute event would have to follow.

A Print event that emits a message to a computer screen may not be considered to result in a change in state. Quite simply the state of the program does not change because of a
printed message. It could be argued that the state of the world has changed since we now know that a message was printed. However, for the computer program no memory exists so program state remains unaltered. It should be noted that the printing of the message is thought to be instantaneous though in actual fact it is not. Printing could alter how a program runs in extreme cases.

The kinds of events being discussed here are not to be confused with those events that occur in the real world such as a birthday. A birthday begins when the previous day ends and ends when the next day begins. This implies that a birthday event has duration. Clearly, from the definition presented here, the word ‘point’ has no duration. To unify the notion of a birthday with the working definition it can simply be stated that a birthday is a state that begins with a Birthday Begin event and ends with a Birthday End event.

Finally, an event is associated with a point in time. If it happens then it must happen at some point in time. It is possible to talk about an event in an abstract way such as muting a speaker. However, in order to realize the muting, or even to understand what muting means, a point in time with a before and after must be associated with it.

2.3 Properties of Time

Time is a child playing draughts, the kingly power is a child’s.

*Heracleitus, DK 52[5]*

The true nature of time has perplexed those who have attempted to discover its essence throughout the ages. Part of the difficulty in understanding time is the fact that we lack control of it. It is not a thing that can be grasped and investigated. Our lives exist inside of it without any means to step outside and observe it.

It is far outside the scope of this work to discover the true nature of time. Instead, we seek to define a notion of time within the universe of running computer programs. This is quite a limitation indeed as it affords us the role of deity who is able to manipulate the
universe to suit our own needs. We hold great power for we can define our own notions of time that can be stopped, started, and moved at any rate we please. However, before all of this new found power can get to our heads we must remember that we are still not completely free of ‘real’ time.

**Property 1. Direction (Advance/Retreat).**

The early Greek philosophers were troubled by the notion of change. Time seemed intrinsically connected to it: in motion, constantly moving forward. As soon as some action is performed it is relegated to the past. The past is placed behind us and we accept that time moves forward to places not visited before. It might be suspected that to move backward in time would be to revisit past actions or events and to undo them and any memory of them as if they had never happened. There is no way to be sure of this as we could define another time base with which this motion is on a forward moving continuum. Clearly these are not assertions that we can test or even experience.

Naturally this forward motion translates to running computer programs. After all, running programs take real time to run. They perform a number of successive tasks as they move towards solving a problem. However, if it is possible to define a time reference and to simulate it on a computer then surely a notion of reversible time could be realized.

A significant problem with reversible time is that its flow becomes non-deterministic. In computer science we seek deterministic properties of algorithms over non-deterministic properties. If a running program is set to call a function named foo then we would expect that foo would soon execute. We would not expect something else to happen. In terms of time related tasks, if we expect a task to occur after some \( n \) ticks of a clock we should expect that that task is executed after \( n \) ticks. At no point should we find that the clock says we are \( n + 1 \) or more ticks away from the task being executed. A clock that is allowed to retreat eliminates the implicit guarantee afforded by non-retreating time that future tasks will eventually be executed.
The utility of a time reference that could reverse would likely be task specific and difficult to generalize. Depending on how this time reference is implemented, scheduled tasks may never execute. If it truly reverses time then a complete history of state changes will need to be kept so that they may be reversed as well. This can be very expensive in terms of space.

Difficulties also arise if events have been scheduled in the future prior to time being reversed. Consider a scenario where the current time is $t$, an event $E_{1_{\text{t-1}}}$ was dispatched in the past, and a new event $E_{2_{\text{t+1}}}$ has been posted for future dispatch. If time reverses to $t - 1$ then $E_{1_{\text{t-1}}}$ will need to be recalled or undone while $E_{2_{\text{t+1}}}$ is now farther into the future. Now that $E_1$ has been undone the conditions for posting $E_2$ may have changed which would require it to be recalled. Therefore, the dispatch of an event may have dependencies on prior conditions. Being able to reverse time may require a system for recording and managing these dependencies.

Systems that appear to allow a user to reverse accumulated changes and retreat to some prior state are not unheard of. Many graphics programs maintain a history of changes and allow users to ‘undo’ them. These changes are not forgotten and may be reapplied unless the user makes a change to the ‘past.’ The same idea is applied to WWW browsers for the internet. The user views a number of web pages with each subsequent visit entered into the history. A user may go back to previously visited pages but should the user choose another link on a previous page then the future path is lost.

**Property 2. Continuity (Continuous/Discrete).**

Wherefore he resolved to have a moving image of eternity, and when he set in order the heaven, he made this image eternal but moving according to number, while eternity itself rests in unity; and this image we call time. For there were no days and nights and months and years before the heaven was created, but when he constructed the heaven he created them also.
Plato sees the creation of time as seemingly continuous yet sensibly divisible. If time were not continuous then there must be some smallest division of time. Nothing could happen between the boundaries of this quantum as this would imply a before and after and that further division is possible. As for the length or duration of this division, Leibniz stated in his New Essays on Human Understanding[13], "if there were a vacuum in time, i.e. a duration without changes, it would be impossible to determine its length." If it's impossible to determine its length then, from our vantage point within it, it is likely impossible to determine if time is anything other than continuous.

Some computer applications attempt to use a continuous notion of time. When interfacing with the outside world events could happen at any time and not necessarily at predictable intervals. This notion is a convenience for the user as it attempts to hide the necessary discretization of computer input. Problems arise when converting from continuous to discrete time systems. For example, if the quantization of time is too coarse then an event occurring at a point in continuous time appears as an instantaneous spike without duration. Unless this event occurs exactly on the tick of the discrete clock its occurrence may be lost.

Understandably, we don’t discuss time or daily events with respect to continuous time but rather to divisions of it. For computer applications the discrete notion of time is far more obvious. Computers execute instructions based on the system clock. Things happen with respect to the edges of the clock cycle. Any time reference implemented on a computer will ultimately be quantized by the system clock. The minimum time division of this reference will likely be much larger than the system clock cycle due to the processing involved in maintaining the reference.

**Property 3. Relativity (Relative/Absolute).**

Absolute, true, and mathematical time, of itself, and from its own nature
flows equably without regard to anything external, and by another name is called duration: relative, apparent, and common time, is some sensible and external (whether accurate or unequable) measure of duration by the means of motion, which is commonly used instead of true time; such as an hour, a day, a month, a year.

_Sir Isaac Newton, Principia Mathematica_[15]

Time can be either relative or absolute. Relative time is, as the name implies, a point in time relative to some prior point. It is duration: a count of the number of units of time since the starting time. When one says “Lunch will be served in fifteen minutes,” the fifteen minutes is a duration in time relative to the point in which the statement was made. Absolute time is the time of some reference time source to which other means of specifying time are often related. The notion of ‘now’, implied in the lunch statement above, is absolute. The same statement could be restated in absolute terms as “Lunch will be served at 12:15 pm.”

In some sense, absolute time is an illusion. The statement “It is 12:15 pm,” means that twelve hours and fifteen minutes have passed since midnight. “Sally is seven and a half years old,” means that seven and a half years have passed since Sally was born.

Whether or not absolute time actually exists apart from relative time is certainly beyond the discussion here. For the purposes of constructing a notion of time for computer programs a type of absoluteness can be defined. If the relative time statement made above, “Lunch will be served in fifteen minutes,” is considered as the expression $t + 15\text{min}$, then its absolute time counterpart is simply that expression evaluated for a particular point in time $t$. Absolute time is therefore defined by the existence of some implied reference point in time based on the reference clock. In terms of event based computation, an event may be constructed to check for email every ten minutes. This relative time might be stored in the event so that when it is first posted at an absolute time of 12:00pm it prompts the check
for email then reposts itself at 12:00 pm + 10 min = 12:10 pm.

**Property 4. Regularity (Regular/Irregular).**

In isolation a single time reference will advance at a regular rate. This must be assumed for there is no way to determine the regularity of a time reference without some other reference to compare it to. From this observation a definition for the regularity of a clock can be stated.

**Definition 3 (Regularity of a clock).** *a sequence of ticks that continuously repeats at the same rate compared to a reference clock.*

Any clock that is chosen as a reference is automatically regular. Indeed it is necessarily so. By definition, if the reference is irregular then there must be another reference that is regular which was used to determine the irregularity of the chosen reference. The regular reference becomes the master reference since no decisions about regularity can be made if the master is irregular.

The regularity of a time base is important to systems using multiple clocks. If two clocks are deemed regular with respect to each other then these clocks may be collapsed into a single clock by defining a conversion method from the subordinate clock $S$ to the master clock $M$. If an event is to be performed at some time on $S$ then the time of the event may be easily converted to $M$.

If an irregular time reference with respect to the master reference $M$ is to be supported then the system will become more complicated. A perfect conversion function from $S$ to $M$ will not exist as this would require accurate prediction. The only way to know exactly when a time on $S$ will happen with respect to $M$ is to wait until the moment that the event happens.

As an example that will illustrate the problems faced in the work presented later, suppose we wish to support two timers. The master timer is based on the standard notion of
time using milliseconds while the subordinate timer is based on a human tapping a drumstick. The system is dynamic in that the types of events dispatched is not preconfigured. In response to each tap a different drum sound is to be played.

Since there is some randomness to the precise timing of human action there is no way for the system to determine exactly when the next tap, or tick of the timer, will come. Predicting the next tap has the potential for gross error. The system will have to detect the tapping of the drumstick and immediately post an event to the queue based on the current time of the master timer so that it can be dispatched immediately. For accurately timed dispatch, the system is forced to support two timers.

2.4 Summary

For computational tasks involving the management of actions over time some notion of time must be adopted. The properties of time presented here will necessarily help to shape that notion of time whether they are directly considered or not. In most cases time will advance as reversible time introduces a number of complications. While time may be thought of as continuous it is generally discussed and always implemented in terms of discrete units. There are systems that use a notion of continuous time, particularly for external interaction, but ultimately they must provide a translation to time with discrete units. Time is generally thought of as relative to some absolute time reference. Supporting multiple regular timers requires translation functions for each timer. However, combining regular and irregular time references can be difficult to do and may require supporting separate timers.
Chapter 3
Time in Programming Languages

Computer programs have always had time as an implicit parameter. For the most part it is something to be optimized by writing more efficient algorithms in order to minimize running time. Along with storage capacity, running time is an important measure of efficiency. However, for a number of problems, time is an explicit parameter that is used to regulate the rate of computation. For these problems, a specific notion of time must be created that is suitable for the problem domain.

3.0.1 General Purpose Languages

General purpose programming languages are those languages that are not directed at a single problem domain. They include a type system, constructs for creating data structures, and constructs for defining the steps of an algorithm. To be useful in a modern computer setting, these languages must have a way to interface with the host operating system. Within the range of these languages, a number of different programming paradigms exist.

Imperative programming involves defining sequences of instructions for changing program state. These statements can be grouped to form functional units, iterative loops for repeating sequences, and decision making constructs. Languages falling into this category include C, Fortran, Perl, and many others. While languages like C++, Java, and C# are typically referred to as Object-Oriented languages, they are essentially imperative with extended semantics for organizing related data and functions.

These languages tend to be, at least in part, low level meaning that a block of program code will have a similar structure to its compiled machine code counterpart. Running time
is impacted by the number of statements in a block of code and the execution times of those individual statements. A programmer can manipulate the number of instructions in order to modify the running time but not necessarily with predictable results. The compiler may also attempt to modify running time by translating these statements into a set of efficient machine code instructions.

Functional languages are different in structure from their imperative counterparts. They are higher level and as a result more divorced from the computer architecture. Rather than containing lists of statements, functional programs contain expressions to be evaluated. In fact, a program itself is an expression. While these expressions may be written in a particular order this does not necessarily imply that they are evaluated in the same order as they are written. This makes it more difficult to determine actual running time from the code itself.

Both language paradigms support constructs for repetition. Iteration using the imperative **while** loop or a recursive function call are ways to structure a set of instructions for repeated application. Frequently, looping involves incrementing an indexing variable for sequential access of data in some linear structure such as in Figure 3.1. In time based computation terms, the ‘samples’ array represents a finite stream of numbers where each has a time component that is some increment of time later than the previous. The variable ‘t’ represents a point in time and ‘speedsum’ is the sum of speeds up to time ‘t’. This interpretation is not without flaws. The time variable ‘t’ can be manipulated to jump to any point in time and a value may be read from ‘nums’ corresponding to that point in time. However, the running sum ‘speedsum’ will not follow a ‘t’ that jumps to a previous point in time.

While each iteration is an increment of computational time later than the previous iteration, time between iterations is not necessarily uniform with respect to a real clock. Consider the conditional statement **if** x **then** foo() **else** (). If the value x evaluates to true then the time taken to call the function foo() will take longer than doing nothing as in the **else** branch. Placed inside a loop where x may vary on each iteration, the run-
```c
float average_speed(int[] samples) {
    int speedsum=0, t=0;
    while (t < nums.length) {
        speedsum = speedsum + nums[t];
        t=t+1;
    }
    return speed/(float)t;
}
```

Figure 3.1: Iteratively summing the elements of an array.

ning time of the loop may also vary on each execution. Precise, repeatable timing will be
dependent on the variable x and the contents of foo().

Ultimately these general purpose languages have no built-in notion of time beyond the
simple ordering of computation. Generally, a computer program is a recipe on how to solve
a problem without indication of the rate at which it is to be solved or when the result is to
arrive. This problem makes it more difficult to control time in a meaningful way.

### 3.1 Data containing Time information

Digital media formats for audio and video will necessarily have a time component to
them. Each sample in an audio file occurs at some point in time relative to the other
samples. Often these samples occur at a regular rate such as 44.1kHz used to record
audio CDs. Other data formats contain data points that do not occur at a predictable rate
and therefore require time information embedded into the stream. The MIDI[14] control
protocol\(^1\) specifies that events contain a delta time parameter that indicates when the event
is to happen relative to the previous event in the data stream.

The MIDI protocol is particularly well suited to capturing real-time information due to
its non-regular time sampling. For example, it is good at recording a piano performance
as it captures the notes being played, the times they are struck, and dynamics of the strike

\(^1\)Musical Instrument Digital Interface is a control protocol for communicating between digital devices.
such as key pressure. Contrast this with an audio recording where each sample indicates the current audio signal conditions at a particular point in time. The data will then have to be reconstructed in order to play or analyze the original recording conditions. This can be an exceedingly difficult task.

### 3.2 Methods for Controlling Time

While most general purpose programming languages do not include mechanisms to control the program execution rate they can be made to do so. Certainly, most languages provide a means to connect to the host operating system thus allowing them to read the system time or various input devices. Using this input timing information programmers can write their own structures and functional units to manage the rate at which their program executes.

A naive program can repeatedly check the time in an endless loop until some future time at which it can awake and perform its task as in Figure 3.2. More sophisticated systems will incorporate multiple timers or try to reduce dead time during polling by performing other tasks. These program units are called Schedulers.

```c
1 dispatch_time = get_next_dispatch_time();
2 while(true) { // loop forever
3     // poll time
4     while(time() < dispatch_time) {
5         // do nothing
6     }
7     dispatch_events();
8     // assuming there are future events
9     dispatch_time = get_next_dispatch_time();
10 }
```

Figure 3.2: A simplified scheduler dispatch loop.
3.3 Scheduling

Scheduling is the task of managing the execution of events with respect to a time reference. At its most basic a scheduler receives events with individual execution times, sorts them according to those times and dispatches them when their execution times are no longer in the future.

The components of a scheduler will depend on the complexity of the task the system is designed to address. A simple scheduling system may operate on a single timer and dispatch a single event at a regular interval. This kind of system will not require much infrastructure to implement. More complex schedulers may have to manage an unpredictable schedule of events. These systems will require a data structure for sorting events according to their dispatch times. Having to manage events on multiple timers can add an extra layer of complexity to the scheduler.

The simplest method for supporting multiple timers is to convert dispatch times to the master timer before inserting them into the queue. However, as discussed in Section 2.3, there may not be a conversion function from a subordinate timer to the master as might be the case for irregular references. In this case, supporting multiple timers may be the answer. A system of this sort would require multiple schedulers with their own timers and sorted event queues. A master scheduler would decide which queue received a particular event based on the time reference it was posted on. A scheduling system that follows this design is described in Chapter 4.

It seems intuitive to picture the event queue as a list of sorted events. This is not the most efficient structure as insertions into the queue will require worst case $O(n)$ operations on a list of $n$ events. Of course, the advantage is that removal is $O(1)$ — a single operation. A more appropriate structure for handling large numbers of events might be a balanced tree or heap with $\log n$ insertion and removal.

Schedulers must run efficiently. The time taken to organize events and to dispatch them
will have an effect on performance. For a real-time system the rate at which the scheduler can dispatch events places a limit on the smallest time quantum that the system can handle. For non-real-time systems there is no limit on the minimum time division that events can be dispatched at although there may be practical reasons for introducing one.

Having to build a mechanism for managing time by hand can be a laborious task depending on the level of sophistication required. Different tasks require different levels of control and generality. In certain problem domains scheduling is a necessity but having to build in a mechanism to manage the execution rate of a program can obscure the program logic and distract from the original task. In many cases domain specific languages have been developed that take care of the scheduling for the software writer.

3.4 Time Based Programming Languages

In this section a number of different programming paradigms and languages are presented that support event based programming with respect to time. Each language has had to develop a notion of time that is useful to the particular application area as well as methods to specify when tasks are to be executed.

3.4.1 Real Time Programming Languages

Real-time programming languages deal with problems for which time critical performance guarantees are required. Applications like robotics, airplane systems, and computer based music performance require that tasks are completed within a specified time frame and are not delayed by such things as memory allocation or garbage collection. These applications do not allow the software to pause the clock in order to service incoming task requests.

Real-Time Java[3] represents a collection of technologies that make real-time programming possible in Java. Among its components is a set of time related classes. Clocks represent continuous time which is actually relative to January 1, 1970. The Timer class
counts time relative to some point on a clock. Events may be scheduled on a timer through its `schedule` member function and will fire the event at the appropriate time.

Classes such as the AsyncEvent and RealTimeThread are schedulable by a scheduler. Typically the AsyncEvent is subclassed to provide event functionality. It provides a `fire` member function that is called when the event is dispatched by the scheduler. The scheduler and how it works is somewhat open so as not to tie the platform to one method. However, a priority based scheduler is included in the basic system.

### 3.4.2 Synchronous Languages

Synchronous languages do away with the notion of logical units of time. Instead, events occur inside sequentially ordered time slots. The rate at which these slots occur is synchronized to a clock yet is not necessarily connected to any physical notion of time. For example, measuring the outside temperature might represent a clock where each tick occurs every time the temperature changes by a degree. Slots have no duration yet may contain any number of events. These events are executed instantaneously or before the next time slot begins.

The Lucid[21] language has been somewhat influential to languages in this area even though no implementation is available. Lucid is somewhat explicit about the incremental nature of a running computation. Functions may be defined that describe a computation over a stream of data. The “followed by” operator `fby` indicates how a value will change on each iteration of computation along with the `next` operator used to divine the value of a name on the next iteration. As an example the expression in Figure 3.3 will take on the values 1, 2, 4, 8, 16, ... Note that at time $t = 0$, $f = 1$ while `next` $f$ evaluates to 2. These operators are used to describe streams, or values that change over time.

```latex
f = 1 \ fby (2 \ fby 2 * \ next \ f);
```

Figure 3.3: Definition of a function that generates the powers of 2.
Sometimes it is useful to sample the values of a stream at a rate other than the rate of the stream being sampled. Lucid defines the `whenever` operator for this purpose. Whenever the expression on the left-hand-side is true, and the expression on the right-hand-side is true, the value of the left-hand-side is emitted. In Figure 3.4, the value of \( f \) is emitted whenever it is evenly divisible by 4\(^2\). A useful visualization presented in Figure 3.5 is the “chronogram” which is used extensively in the Lucid Synchrone documentation[18].

```
1 \( f \) whenever \((floor(sqrt(f))) ** 2\) eq \( f \)
2 where
3 \( f = 1 \) fby \((2 fby 2 * next f)\);
```

Figure 3.4: Using the `whenever` operator to filter every other element of \( f \).

```
| t  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ...
|----|---|---|---|---|---|---|---|---|---|---
| f  | 1 | 2 | 4 | 8 | 16| 32| 64| 128|256|...
| (sqrt(f) mod 1) eq 0 | T | F | T | F | T | F | T | F | T | ...
| =  | 1 | 1 | 4 | 4 | 16| 16| 64| 64| 256|...
```

Figure 3.5: Timing table for the function output of Figure 3.4.

Lucid Synchrone[8], built on top of OCaml, is a realization of the Lucid language used for experimenting with reactive systems. All basic types including constants are lifted to value streams. Consider the example in Figure 3.6 taken from the Lucid Synchrone Tutorial and Reference Manual[18].

```
let node sum x = s where rec s = x -> pre s + x
```

Figure 3.6: A sequential function definition for summing values in the stream \( x \).

This sum function is known as a sequential function, denoted with the keyword `node`, as it describes an operation on a sequential stream of data. When the function is called it

\(^2\)The floor operator does not appear in the Lucid grammar [21] but is introduced here for convenience. I will assume for this illustration that the result of \( floor(sqrt(f))**2 \), of type float, may be compared \( f \), of type integer.
is supplied an integer stream labeled \( x \). For the first value in the stream of \( x \), \( s \) takes on that value. For all subsequent values of \( x \), \( s \) takes on the value of \( \text{pre} \ s + x \) which means the previous value of \( s \) and the current value of \( x \). Note that, at the current instant, \( s \) is not defined. Therefore the expression \( s + x \) is undefined.

As with Lucid, the sum function has an implicit clock that defines the rate of execution. This clock is defined by the rate of values appearing in stream \( x \). Changing the rate of sampling is performed in a similar way to Lucid. The parameter \( y \) of Figure 3.7 defines a second stream whose rate becomes the sampling rate of \( x \). Like the \textit{whenever} operator of Lucid, the \textit{when} operator defines the new sampling rate of \( x \).

\[
\text{let node sampled_sum} \ x \ y = \text{sum}(x \text{ when } y)
\]

Figure 3.7: Using the when operator to change the rate at which values of \( x \) are sampled.

### 3.4.3 Reactivity

Reactive systems are those that propagate signals by reacting to changes in the environment. These systems adopt a flow model based on connecting individual processing components. Not to be confused with the control flow of imperative languages where individual statements are executed in order, reactive systems use dataflow where elements of the system are executed when the dependencies or inputs change in value. Reactivity is not a notion of time, however it is very useful in defining one. Lucid and Lucid Synchrone employ a dataflow model of computation which is inherently reactive.

As the name implies, components of the system react to their inputs by modifying their state and propagating new signals to each output. Consider the statements
\[
a = b + c; \quad b = a + d;
\]
In a general purpose language the statements are evaluated sequentially to arrive at a value for \( a \) which is then used to find the value for \( b \). Evaluation stops. In a reactive system, the names represent value streams that change over time. The value for \( b \) is used
to compute a value for $a$. This change in $a$ prompts an evaluation of the second statement to find a value for $b$ which prompts an evaluation of the first statement and so on.

A change in input should take zero time to propagate. If the system accepts an input change then the system should finish updating itself prior to it accepting another change in input. In the statements above $c$ may be an input which starts the evaluation of the first statement and the subsequent endless cycle. Care must be taken to avoid this. In Lucid and Lucid Synchrone this sort of endless loop is avoided by making $a$ and $b$ undefined in the current instance and forcing the use of the `pre` keyword to access their values at the previous point in time. The expressions would then become $a = \text{pre}(b) + c; b = \text{pre}(a) + d$.

Esterel[2] is an example of a reactive programming language. It is an imperative language whose functional units resemble Mealy automata. Each unit has a number of inputs and outputs. The rate of computation is controllable by the programmer through the declaration of inputs. This is a ‘Multi-Form’ notion of time where inputs resemble timers, because their signals can be counted, while outputs resemble events. For example, suppose that `Press` and `Second` are inputs and `Candy` and `Siren` are outputs. The statement of Figure 3.8 reacts to the press of a button on a candy machine by outputting a `Candy` message. After receiving the `Candy` message a connected system might respond with a siren that sustains for 3 seconds as in Figure 3.9.

```plaintext
1 every Press do
2       emit Candy
3 end
```

Figure 3.8: Esterel statement for reacting to a button press.

Functional Reactive Animation[12] is a language, built on top of Haskell, for defining graphical animations. FRAn defines two important data types: behaviors and events. Behaviors are polymorphic, reactive values that vary over time. They are equivalent to a
function of type \( \text{Behavior } a : \text{Time} \rightarrow a \). A stream of audio samples could be a behavior of type \( \text{real} \). Time itself is a behavior: its definition presented in Figure 3.10. Events were originally defined as time/value pairs [12] or \( \text{Event } a : \text{Time} \times a \) but were later redefined to be sequences of time/value pairs [11].

FRAn uses a continuous notion of time primarily because the authors feel it is more appropriate for capturing interactive events. Ultimately FRAn is computer based and continuous time must be mapped to the discrete. This poses some problems to an implementation primarily due to the infinitely divisible nature of an interval of time. Events may occur instantaneously and may pass undetected. As an example, the event “light on when time equals 1” might be interpreted as “turn the light on for the instantaneous moment when time is equal to 1 then immediately turn it off.” In response to this issue the authors impose a kind of interval analysis where the values taken on by a behavior can be captured over an interval.

Figure 3.11, borrowed from FRAn[12], shows an example of a declaration of a reactive behavior. The colorCycle function defines a behavior of type \( \text{colour} \) which is red until a left-button press is detected at which point it becomes green. The source code of Figure 3.11 appears more complicated than it is. The \( \text{lbp} \) function represents an external left-button-press event that, when supplied a time argument \( t_0 \), returns the pair \((t_{\text{event}}, \text{event})\)
where $t_{event}$ occurs after $t_0$. The $*=>$ function is an event handler that takes an event and a function from Time to a behavior. This event handler takes the time out of the event pair and calls the supplied function with that time and produces a new event. The 'untilB' function maintains the behavior on its left-hand-side until the event handler returns the result of the left-button-press event. It computes a new behavior which in this case is the switch to the colour green.

```
1  untilB : Behavior a -> Event (Behavior a) -> Behavior a
2  lbp   : Time -> Event (Event ())
3  (*=>) : Event a -> (Time -> b) -> Event b
4
5  colorCycle t0 =
6      red  'untilB' lbp t0 *=> t1 ->
7      green 'untilB' lbp t1 *=> t2 ->
8  colorCycle t2
```

Figure 3.11: FRAn function for cycling colours on left button presses[12].

### 3.5 Audio Programming Languages

A number of audio programming languages and frameworks exist for processing audio. Audio tasks include generating music, synthesizing sound, and audio feature extraction. Each of these tasks has a temporal component to them due to the changing nature of sound. For this reason, each language must adopt a notion of time suitable to the tasks it wishes to accomplish. Trade-offs may have to be made.

#### 3.5.1 CSound

CSound[9] is an audio programming language for sound synthesis and musical performance. A ‘program’ consists of two units: the instruments definition, describing the instruments and the sounds they make, and the score that contains the notes played and
the time at which they are to sound. Each note must specify the time at which it is played relative to the beginning of the performance along with the duration that the note is to sound.

Figure 3.12 shows a basic CSound Instruments definition section. For all instruments the sample rate sr is 44.1kHz and control statements in the Score section will be evaluated at a rate defined using the ksmps value.

```
<CsInstruments>
sr = 44100 ; Sample rate
ksmps = 10 ; Control Sample Period
nchnls = 1 ; Number of output channels

instr 1
  a = 1000 ; Amplitude
  f = 440 ; Frequency 'A'
  p = 0 ; Phase
  myosc oscil a, f, p  ; Oscillator
  out myosc       ; Output.
endin
</CsInstruments>
```

Figure 3.12: CSound Orchestra Specification

A Score section for a CSound program is shown in Figure 3.13. Unless specified elsewhere the default tempo for time statements is one second. The f opcode constructs a table used for generating the audio. The i opcode is used to activate an instrument. In this case the parameters in order specify that instrument 1 is to be played at time 0 and it should be played for 1 second. The next i opcode specifies that instrument 1 should be played for 1 second at time 2 seconds. Finally, the e opcode specifies the end of the last section.

The Instruments and Score sections define the entire piece prior to it being played. Scheduling of events is therefore somewhat straightforward as the events and their order are known at the start. CSound can be used for real-time performance. It has the ability
Figure 3.13: CSound Score Specification

to accept input such as MIDI for controlling parameters of instruments. MIDI messages however are handled immediately and are not scheduled.

3.5.2 Chronic

Chronic [4] is a language for computer music programming that introduces the idea of Temporal Type Constructors. As the name implies, temporal types are types with a time component. The three constructors introduced in Chronic are: $\alpha$ event having type $\alpha \times \text{time}$, $\alpha$ vec is a vector indexed by time, and $\alpha$ ivec is an infinite vector indexed by time. These type constructors can be combined with regular types to describe data that changes over time.

Figure 3.14: Chronic temporal type describing a chord progression[4].

Consider Brandt’s chord progression example[4] of Figure 3.14. Each vertical line represents a set of pitches of type Pitch vec. Each set of pitches (a chord) happens at an arbitrary point in time, indicated by the symbol @, and has type Pitch vec event.
Finally, a succession of chords over time will be contained in a vec and will have the Chronic type Pitch vec event vec.

The difficulty with a vec containing events is that not every slot in the vec will contain an event. What do the empty slots contain? Chronic defines an event vec module to address this problem. Figure 3.15 shows three representations of a series of events. Line 1 shows a vec containing four events at successive points in time. Line 2 shows interpolation between events using the piecewise constant function. An initial value is required, in this case 8, in case the first event does not start at time 0. Line 3 shows a piecewise linear interpolation where the values in between two events migrate linearly towards the next. Obviously other interpolations are possible given the right function.

1 let events = [| 3. @@ 1; 1. @@ 3; 4. @@ 6; 1. @@ 8 |]
2 let pwc = [| 8.; 3.; 3.; 1.; 1.; 4.; 4.; 1. |]
3 let pwl = [| 8.; 3.; 2.; 1.; 2.; 3.; 4.; 2.5; 1. |]

Figure 3.15: Chronic event vec representations.

Chronic’s programming model is distinct from that of languages like CSound. In the CSound model the programmer has control over those constructs that are active at the current point in time. Chronic allows the programmer to step out of the timeline and operate on a period of time. This is more powerful because it makes possible such processes as delay, by looking back in time, or the application of the Fourier Transform for analysis which requires an array of samples over time.

3.5.3 ChucK

Chuck[22] is a strongly timed audio programming language suitable for sound synthesis experimentation and is even used for real-time performance. A running program and therefore the sound output may be modified in real-time with the dynamic addition or subtraction of running code.
ChucK bases its clock on the audio sample rate which is usually $44.1kHz$. Processing is performed a single sample at a time without buffering. This affords ChucK the capability of sample accurate timing at the cost of efficiency.

A left to right syntax is used for assignment using the ‘chuck’ operator $\Rightarrow$. Processing is advanced by manipulating the keyword `now` through ‘chucking’ a time value at it as in $1::\text{ms} \Rightarrow \text{now}$ on line 8 of Figure 3.16. This expression regulates the rate at which the while loop cycles and therefore defines the rate of change of the `sinosc` frequency.

```
1 sinosc s $\Rightarrow$ dac;
2
3 0.0 $\Rightarrow$ float t;
4 while( true )
5 {
6   ( math.sin(t) + 1.0 ) * 10000.0 $\Rightarrow$ s.sfreq;
7   t + .004 $\Rightarrow$ t;
8   1::\text{ms} $\Rightarrow$ now;
9 }
```

Figure 3.16: ChucK program

Until recently audio analysis was not available in ChucK due to its single sample architecture. For example, determining which frequencies occur during a period of time in the audio signal requires a Fourier Transform to be performed on it. The Fourier Transform will transform the time based signal into a collection of frequencies occurring during the period of time being analyzed. Usually this period of time is small, perhaps on the order of 512 samples. No frequency information can be discovered from a single sample.

In order to maintain its precise control over time, ChucK has adopted a more flexible approach to audio signal analysis[24]. Unlike PD[19] and Marsyas[20], which slice the signal into windows of a particular size and perform analysis on each, ChucK waits until the precise point in time that analysis is requested. The new Unit Analyzer building blocks do not interfere with the primary task of synthesis but wait alongside the synthesis stream,
28

```
// our patch
adc => FFT fft =>^ Centroid c => blackhole;

512 => fft.size; // set the FFT window size
Windowing.hann(512) => fft.window; // set hann window
second / samp => float srate; // compute sample rate

while( true ) {
    // get centroid which gets the fft
    c.upchuck();
    // print out centroid
    <<< cent.fval(0) * srate / 2 >>>;
    // advance by the sample window
    fft.size::samp => now;
}
```

Figure 3.17: Signal analysis in ChucK

buffering sample data until a request is made for analysis. An example of analysis in ChucK, taken from the ChucK website[23] is shown in Figure 3.17.

### 3.5.4 Pure Data, Max/MSP

Pure Data[19] and its commercial counterpart Max/MSP are real-time graphical languages for processing media. Processing objects are placed on a canvas and connected by drawing lines from outputs to inputs to describe a flow network. Audio data, successively modified by flowing through the network, and control data are separate flow types. Since the product of the network is sound, time is referenced to the sample rate which is typically 44.1 kHz. Time is advanced by allowing data samples to cascade through the network. Conceptually, a single data sample passes through at a time. In practice, however, data passes through in buffers of 64 or more samples for efficiency. The flow of control data is affected by the size of the buffers passing through as control changes only occur on the boundaries of a buffer.
Control flow and events are unified as messages passed between objects. Consider the PD network of Figure 3.18a. The graph shows two Message objects: bang and stop. When bang receives a mouse click it sends the bang message to the delay object. The delay object holds back the bang message for 2000 milliseconds after which it is printed to the console output.

Time is always in standard units but message flow can be regulated by other means. For example, the ‘select’ object, a kind of conditional statement, passes messages based on its arguments. In Figure 3.18b each time the bang is clicked the count contained in the float object is incremented by one. The result of the count modulo three is passed as a message to the select object which only passes a message on the left output when the input message is 0.

### 3.5.5 Marsyas

Marsyas[20] is a C++ software framework for developing audio analysis applications with emphasis on Music Information Retrieval. An audio application can use the framework to describe a dataflow audio processing network for transforming an input signal and/or extracting information from it.

The basic building blocks of the data-flow network are ‘MarSystem’ objects. Each MarSystem describes a particular processing task on the audio data whether it modifies the data or extracts information from it. For efficiency, data is passed through the network...
in buffers that are typically 512 samples in size. Marsyas can modify the buffer size at run-time which sets it apart from other systems using a similar buffered approach.

```cpp
class Gain: public MarSystem {
    MarControlPtr ctrl_gain_;  
    void myUpdate(MarControlPtr sender);
public:
    Gain(string name);
    void myProcess(realvec& in, realvec& out);
}
```

Figure 3.19: The outline of a basic MarSystem class.

Figure 3.19 shows a basic MarSystem example. MarSystems break down into several main components: controls, construction, update, and process. Controls are the class variables that define the MarSystems behavior. They are accessible outside of the MarSystem using a path notation and task specific function calls. Line 2 defines a control. Construction of a MarSystem involves initializing controls and adding them to the lookup mechanism.

During processing a MarSystem is passed a buffer of sample data, called a realvec, to its myProcess function. It is also supplied an output buffer which will be passed to the next MarSystem in the network. The myUpdate function is called whenever a MarSystem’s controls have been modified. This allows for the updating of dependencies that rely on modified controls.

Composite MarSystems can contain other MarSystems which allows for the construction of more complex networks. Figure 3.20 shows the shapes of the Series and Fanout composites. Each one is essentially a list of MarSystems with a different orientation. It is the task of the Composite MarSystem to manage the passing of data between those MarSystems it contains.

Like Pure Data, control flow is a separate concern from the data flow. Controls are accessible through specific get and set function calls on the network. In order to address a
control, buried somewhere in the network, a path notation is used. The example on Line 14 of Figure 3.21 shows an update call on the MarSystem called `series`. This causes a search within `series` for the Gain MarSystem called `gain`. Once found, a search for the gain control of type `mrs_real` is performed. Once found, its value is set to 0.5.

Because Marsyas processes samples of audio data it is inherently tied to the audio sample rate as a time reference. However, audio passes through the network in buffers of multiple samples. The network must be prompted to process each buffer by invoking the `tick()` function call as on Line 22 of Figure 3.21. This means that time essentially stands still until the controlling function forces a tick. Once the tick has been made the controlling function stands still until the buffer of samples passes through the network. This means that no control calls such as `getctrl`, `setctrl`, or `updctrl` can be made while the network is processing data. The result of this is an effective time resolution of the buffer size passing through the network.

### 3.6 Summary

No matter what the programming paradigm is there exists some notion of time. At the very least, all programming environments and languages are subject to the constraints
MarSystemManager mng;

// construct the network
MarSystem* series = mng.create("Series", "series");
MarSystem* src = mng.create("SoundFileSource", "src");
// let the ser marsystem connect these together
series->addMarSystem(src);
series->addMarSystem(mng.create("Gain", "gain"));
series->addMarSystem(mng.create("SoundFileSink", "snk");

// set the input filename
src->updctrl("mrs_string/filename", "ip.wav");
// set the volume multiplier to half
series->updctrl("Gain/gain/mrs_real/gain", 0.5);
// set the output file name
series->updctrl("SoundFileSink/dest/mrs_string/filename", "op.wav");
// loop while the input file has more samples
while (src->getctrl("mrs_bool/notEmpty")->to<mrs_bool>())
{
    // process another buffer
    series->tick();
}

Figure 3.21: Marsyas Example for creating an output file with a loudness of half that of the input file

of real time where each sequential instruction costs some measure of time to run. Time becomes an antagonist whose influence must be reduced by writing more efficient software. Beyond this notion of time is that of simulated time. Counting loop iterations is a common practice for programs written using general purpose languages. Actions can be run when a variable reaches a certain count. However, more sophisticated timing mechanisms require considerable programming effort to implement.

Many languages and environments have been developed for time aware programming tasks. The notions they have adopted work well for their given application area. Real-time
languages are well suited to interaction with the outside world. Musical languages use sequential time ordered into beats per minute. Sequential programming languages abstract time even further to that of a sequential ordering of events.

In the next two chapters two systems for developing audio processing applications will be presented. Each one has adopted a very general notion of time that does not limit itself to a single clock. Where ChucK schedules events according to the audio sample rate, the Marsyas Scheduler of Chapter 4 allows users to define their own timers and events. This flexibility is carried over into MarsyasOCaml described in Chapter 5. MarsyasOCaml improves on the Marsyas design by making controls reactive. This improvement effectively distributes the timing and scheduling to every control.
Chapter 4
Flexible Event Scheduling for Audio Processing

The first version of Marsyas was primarily focused on extracting information from music. With the introduction of sound synthesis in the second version it was clear that Marsyas was to take a greater interest in sound and instrument experimentation. However, Marsyas still had no built-in method for controlling time leaving this management to the application developer. Changes to control values were immediate and could not be delayed into the future.

The Marsyas Scheduler[6] project was given, as its main goal, the task of adding scheduling capabilities to the Marsyas framework. Originally, this meant scheduling the modification of control values over time. However, it soon became apparent that by keeping the scheduler as general as possible that it could be made to schedule a wider range of event possibilities using a wide range of time references.

4.1 Marsyas Scheduler Architecture

Since no scheduler existed in Marsyas prior to this project it was not known what range of uses a scheduler might encourage. There were the obvious uses of delayed setctrl/updctrl function calls but there could also be calls to the operating system or other external systems not in Marsyas. The choice of time references would necessarily include the sample rate and system time but it could also support interactivity if it could detect user inputs. It became apparent that the most useful scheduling system is one that lets the user define the
limits of time and event.

While a generalized scheduler that places no limits on the notions of time and event is appealing it is not entirely possible within the context of Marsyas. There are a number of practical considerations that limit the scheduler design. The most important consideration is the flow of data through the network. Recall that Marsyas is built around the processing of audio sample data so its primary reference is the audio sample rate. To benefit efficiency, data flows through the processing network in buffers of samples rather than a single sample at a time.

It would be desirable to support sample accurate timing of event dispatch. However, achieving this would be particularly difficult due to the use of buffered processing. Consider a buffer of data from time $t$ to $t + L$ flowing through the network. The sample at time $t$ may be processed up to $n$ times when flowing through $n$ MarSystems. A MarSystem $B$ that processes after another MarSystem $A$ will actually process the sample at time $t$ after $A$ has processed the sample at time $t + L$. In effect, processing jumps back in time each time a MarSystem in the network receives a buffer of data. Complicating the problem further is the issue of state. If $S_t$ is the program state prior to $A$ processing the buffer then that is the state that each MarSystem in the network must see when it receives the same buffer. If the state is modified during processing through a control value change then all those changes must be reset then repeated for each MarSystem that sees the buffer.

Figure 4.1 illustrates the problem. While MarSystem $A$ processes the data Control value $D$ is modified. When MarSystem $B$ receives the processed buffer, time is reset to $t$ but according to Control value $D$, time is $t + L$. In order for the system to remain consistent $D$ will have to be reset and the original event that modified $D$ at time $t + k$ will have to be rescheduled so that the process can be repeated.

While audio samples may be repeatedly processed as they flow through the network, the buffer that carries them will only flow through the network once. That is, once a slice has flowed through the network it will never flow through it again. Therefore, if the MarSystem
Figure 4.1: Controls modified during buffer processing may become inconsistent between MarSystems.

state changes are delayed until after a buffer is processed then the state of the network need not revert to a past state during processing. This greatly simplifies the implementation at the expense of sample accuracy. Time resolution, therefore, becomes a function of the buffer size. Both Marsyas and Pure Data delay processing of control data until after a buffer is processed.

This simplification is not without its problems. Sample accuracy is lost except where state changes happen on slice boundaries. For many situations this loss in accuracy is tolerable as the typical delay will be measured in small fractions of a second. For example a slice of 512 samples at 44.1kHz sample rate would have a worst case loss of $\frac{512}{44100}/2 = 0.0058s$ or roughly 5.8msecs. However if the dispatch time of an event is based on a prior event, such as when repeating an event at intervals, there will be a cumulative error as shown in Figure 4.2. This cumulative error can be reduced or avoided by using it as a parameter to adjust the reposted event dispatch time.

4.2 Scheduler Design

The Marsyas Scheduler design, as shown in Figure 4.3 abstracts the notions of time and event away from the scheduler itself. These abstractions allow for a diverse range of events and reference timers, leaving their definition up to the designer of the processing network.
Events need not be constrained to actions within the framework. There is no reason why an event should not be able to trigger actions external to the network.

As an example, the scheduler Marsyas could be used to control a lighting system where flashing lights are synchronized to beats detected in the audio. Marsyas is capable of recognizing beats in an audio signal. These beats can be used as a source for a timer. An event could then be created that interfaces with an external I/O port to communicate with the lighting system. Finally the event can be posted on the scheduler and set to repeat. Section 4.2.5 describes a similar example and its implementation.
Figure 4.4 shows the sequence of calls performed when dispatching an event. The process begins with a tick call on the Series MarSystem. This call propagates to each scheduler’s timer. If the timer has advanced since the last tick, then the timer calls the scheduler’s dispatch method. If there are events who have dispatch times that are less than or equal to the current time then the scheduler will call the event’s dispatch method. In the figure the particular UpdCtrl event calls the Series MarSystem with an update control message.

4.2.1 Events

As defined in Chapter 2, an event is something that happens at a point in time. Its action could result in a side-effect such as printing a message to the console, or it could make a change to the system state such as updating a control value. In Marsyas an event is split into two classes: a MarEvent and a ScheduledEvent. The MarEvent is an abstract class that allows the user to define their own event actions. The ScheduledEvent class contains the event time information required by the scheduler for ordering the event based on its dispatch time. The interfaces for these classes are presented in Figure 4.5.

The MarEvent interface requires the implementation of a single method called dispatch. This method is called at the point in time the method is to be dispatched and is intended to
```java
interface MarEvent {
    MarEvent();
    void dispatch();
}

interface ScheduledEvent {
    ScheduledEvent(int time, MarEvent e, Repeat r);
    void setTimer(TmTimer t);
    int getTime();
    void setTime(int t);
    MarEvent getEvent();
    void setEvent(int time, MarEvent e, Repeat r);
    bool repeat();
    void doRepeat();
}
```

Figure 4.5: Event Interface.

perform the event action.

The ScheduledEvent class is a wrapper for the MarEvent class. It exports all the timing information required by the Scheduler to order the event by its dispatch time. The setTimer method associates a reference timer with the event. It is used to calculate repeat times as necessary. The methods getTime and setTime are used to access the time property for the event. This value is an integer count based on the associated timer.

4.2.2 Timers

Timers are objects that advance time by counting. This sparse definition allows for a diverse range of timers. Audio sample rate or a system clock are examples of advancing counters that could be used to reference a timer. Sensor input of some real-world action such as tapping a drumstick or opening a door could also be used to reference a timer.
Timers need not advance linearly but they must advance.

Figure 4.6 shows the interface for a Timer. When constructed the Timer should be supplied a reference to the Scheduler with which it is associated through the setScheduler function. During operation the timer is advanced by calling the tick function on each buffer boundary. The tick function calls readTimeSrc to get the current value of the time source which it then uses to update the cur_time value. The tick function then calls the trigger function which simply calls the scheduler’s dispatch function. Granularity is simply a means to reduce the number of calls to trigger to every $g$ ticks.

For a Timer with default behaviour the only function that needs to be implemented is readTimeSrc. A time source could be as simple as reading a control value or something more complicated like a system call.

```java
interface TmTimer {
    string type;
    string name;
    unsigned long cur_time;

    void setScheduler(Scheduler* s);
    void setGranularity(int g);
    int getTime();
    void tick();

    int readTimeSrc();
    void trigger();
    int intervalsize(string interval);
}
```

Figure 4.6: Timer Interface.

### 4.2.3 Scheduler

A scheduler is simply an object that manages events based on a single timer. The Marsyas Scheduler uses a heap to sort events according to the designated event dispatch
time in relation to the scheduler’s timer. Note that the scheduler itself need not know how
the timer or an event works but only how to compare the event time with the timer in order
to be able to sort events and dispatch them.

As can be seen in the interface of Figure 4.7 the scheduler contains a Heap and its own
timer. The post method is used to sort new events into the heap. Ultimately, the heap
expects a ScheduledEvent so the post method that accepts three arguments will repackage
those arguments into a ScheduledEvent and call the other post method. The tick method
simply calls the timer’s tick method so that it may update its count. Once the count is
updated the eventPending method may be called which determines if the topmost event
on the heap has a dispatch time that is less than the current timer’s time. If an event is
pending then the dispatch method may be called. This method will continually pop pending
events off the stack and call their dispatch methods. Once an event has been dispatched, its
ScheduledEvent wrapper will queried to see if it is a repeating event and if so its dispatch
time will be updated and it will be reinserted back into the heap.

```
1 interface Scheduler {
2    Heap<string, ScheduledEvent> queue;
3    TmTimer timer;

4    void setTimer(TmTimer t);
5    void tick();
6    bool eventPending();
7    void dispatch();
8
9    void post(string time, MarEvent e, Repeat r);
10   void post(ScheduledEvent e);
11 }
```

Figure 4.7: Scheduler Interface.
4.2.4 Virtual Scheduler

Multiple control rates may be desired for scheduling different events. However, as the scheduler interface shows, a scheduler has a single timer. Dannenberg[10] has previously outlined a method for converting multiple control rates into a single reference rate so that events may be organized in a single structure. His work implies that for each time reference supported by the scheduler there exists a function, or a very close guess, for transforming a given time reference into the master reference time. It was decided that this was not possible for the Marsyas Scheduler as it would places limits on the generality of the scheduler. Suppose the scheduler receives an event referenced to a sensor on a door. There is no way to accurately guess when the door will be opened or closed. The alternative was to create a Virtual Scheduler that controlled multiple schedulers each with their own timer. It is the Virtual Schedulers task to direct new events to the correct scheduler based on the time reference the event is to be posted on.

```java
interface VScheduler {
    List<Scheduler> schedulers;

    void addTimer(Timer t);
    int getTime(String timer);

    void tick();
    bool eventPending();

    void post(String time, String timer,
               MarEvent e, Repeat r);
    void post(Time t, MarEvent e, Repeat r);
}
```

Figure 4.8: Virtual Scheduler Interface.

The Virtual Scheduler interface is shown in Figure 4.8. The addTimer method checks the list of schedulers for one with a timer matching the requested timer. If one does not exist
then a new scheduler is created with the timer and is appended to the list of schedulers. The first post method creates a Time object from the given time and timer string variables and calls the second post method. The second post method creates a ScheduledEvent object, searches the list of schedulers for one with the timer name corresponding to the name in the Time object, and finally posts the event on the correct scheduler.

4.2.5 An Example

Figure 4.10 shows an application that dispatches events in response to hand clapping or any other loud audio beats. These beats become the countable timer ticks that drive the scheduler’s clock. The network itself is constructed using four MarSystems in series: an AudioSource that reads audio data from the computer input, a PeakHold that detects peaks in the audio signal that are greater than the average, a SineSource for generating the audio tone, and an AudioSink that sends the tone to the computer speakers. The SineSource actually ignores the incoming audio data so that, in this case, the output of PeakHold does not end up in the resulting audio signal.

The PeakHold MarSystem of this example is actually a modified Peaker MarSystem. It detects peaks in the audio signal based on a region of the buffer defined using the mrs_natural/peakStart and mrs_natural/peakEnd controls. The mrs_real/peakStrength control determines the level at which a peak in the signal may be assumed. On Line 25 the mrs_natural/peakValue control is used which will contain the value 1 when a peak is detected and 0 otherwise. The TmSampleCount timer will read this control as its timer source and add this value to the current timer count. One issue to consider when counting peaks is that a loud noise may spill over into multiple buffers. This can have the effect of a peak being found in each buffer when only one exists. The easiest way to avoid this problem is to use a flag to indicate that a buffer has been detected and to reset this the next time a buffer contains no peak.

On Line 22 the TmSampleCount timer is added to the enclosing Series MarSystem.
This timer advances by counting a control value from a supplied MarSystem. In this case it will count the peaks detected with the PeakHold MarSystem. If there are no peaks in the current buffer window then the timer will not advance.

On Line 28 a new EvPlaySeq event, described in Figure 4.11, is created that takes a sequence of note frequencies. Each time the event is posted it updates the frequency of a supplied SineSource MarSystem and then advances to the next note in the sequence. By setting the repeat time to 1, the event will be reposted to one clock tick in the future.

Figure 4.11 shows the EvPlaySeq event. Most of the code is used during the creation of the event. The most important method is dispatch. It simply sets the frequency of the target SineSource MarSystem using the value in the notes array then advances the array position counter.

This example uses one of Marsyas’ strong points — analysis — to drive the main scheduler timer. While this form of analysis may not be very sophisticated it does demonstrate a timer having an irregular rate. Clapping or tapping a drum stick near a microphone could drive the timer. To use sound files as a source, simply change the AudioSource MarSystem to a SoundFileSource and set the file name.

4.3 Is this general enough?

Complete generality of the scheduler design is not achievable without redesigning Marsyas itself. Marsyas is tied to a sample rate that defines the absolute time reference to which all others are subordinate. Normally this sample rate mirrors the sample rate of the audio source whether it is a sound file or live audio input. In addition, the buffers that flow through the network are related to the sample rate where one buffer element corresponds to a single sample. While the sample rate is modifiable, even at run-time, it would be difficult to impose a new timing regime given that all the MarSystems and supporting elements assume a connection between the sample rate and the buffer samples. Also, re-
call that for efficiency each MarSystem processes an entire buffer before the scheduler has
a chance to check for pending events. This forces an even coarser quantization on any
scheduler timers. These limitations are not a flaw of the design of Marsyas but are instead
optimizations for the primary task of audio processing.

It would be interesting to imagine a more generalized system. In such a system, the
relationship between scheduler and processing network becomes reversed. Instead of a
scheduler being a member of a MarSystem as in Figure 4.3 the MarSystem and its network
become parameters of those events that use them. An audio sample event may contain
a reference to the processing network that it is intended to be processed with. When the
sample event occurs the scheduler will run the processing network with the sample then
repost the event to await the next sample. Instead of the Marsyas tick() being called on the
network, as in Figure 4.4, the scheduler would be ticked directly at the outset whereupon it
would begin the cycle of checking for events that come due on their respective timers.

The advantage to a system based on a scheduler is that it moves away from a specific
model of time to a system where any model is possible and all are created equal. In terms
of equality, no model of time is subservient to another. This should allow for more accurate
timing as events can be serviced when they happen. Unfortunately the problem of mixing
sample accurate timing with buffered audio processing in a datafow network would still
exist.

4.4 Marsyas Expression Syntax

While the Marsyas Scheduler is flexible, in that it allows users to write their own events
and timers and then compile them into the framework, is also a little difficult to use. Having
to write C++ code is not exactly a model for ease of use. Also, there may be too much
flexibility as the user who wants to customize their event must either rewrite and recompile
the event or build their own flexibility into the event.
To alleviate these problems a new scripting language called the Marsyas Expression Syntax was developed. The syntax is designed to be short and concise so that expressions can be written in as few lines as possible. These expressions are meant to be used as events through the EvExpr event class.

Figure 4.9 shows the construction of an event that updates the frequencies of two sine waves on each occurrence. The Ex expression parser takes two strings. The first is executed when the event is posted while the second defines the event itself. The first string defines two aliases for the frequency controls to aid readability. The second changes the frequency of each control then prints out the new frequencies to stdout.

```c
1  EvExpr* e = new EvExpr(series,
2      Ex("SineSource/src1/mrs_real/frequency >> @freq1, \n
3          SineSource/src2/mrs_real/frequency >> @freq2 ",
4            "freq1 << 120. + 3000. * R.rand(), \n
5               freq2 << 120. + 800. * R.rand(), \n
6                 'src1=' + freq1 + ' src2=' + freq2 + '\n'
7                     >> Stream.op"),
8      Rp("true"));
```

Figure 4.9: Network to play random sinewaves.

MESy contains a number of libraries for dealing with the basic data types, math functions, stream I/O, and timers. Additional functions and libraries may be added, particularly if the user wishes to call functions external to Marsyas. Once defined, these functions must be compiled as classes into the Marsyas Framework. They then become available in the same way as the built-in library functions.

The advantage to using MESy to define events is obvious. Without it, a user would have to define a custom event each time they wish to try something new. Once defined the event needs to be compiled into the Marsyas framework which may take a considerable amount of time to do. Using MESy the new event is defined in application file rather than the framework. There is even a facility to read a MESy file at run-time.
4.5 Summary

The Marsyas Scheduler was presented at ICMC in 2006 [6] with a focus on the integration of scheduling into an audio processing framework, and at OOPSLA 2006 [7] where the focus was on the object-oriented design of the scheduler itself. Both of these conferences were informative to the design of the scheduler as they forced me to design engaging examples that showed off its power. Only those who attended will be able to say if I succeeded. However, it was these demonstrations that motivated the design of the MESy syntax. It had become clear that, while the scheduler is certainly powerful, not everyone has the patience to write C++ code. Systems like Pure Data and ChucK require no recompiling and yield much more immediate results which, in turn, promotes experimentation. In order to fully exploit the power of the scheduler tools must be developed to make it easier for the average person to use.
```cpp
void peak_detect_clock() {
    MarSystemManager mng;
    // these marsyms will be accessed directly later
    MarSystem* pkr=mng.create("PeakHold","pkr");
    SineSource* sinewav =
        (SineSource*)mng.create("SineSource","ssrc");
    // build the network
    MarSystem* ser = mng.create("Series","series");
    ser->addMarSystem(mng.create("AudioSource","src"));
    ser->addMarSystem(pkr);
    ser->addMarSystem(sinewav);
    ser->addMarSystem(mng.create("AudioSink","dest"));
    // update network marsym parameters
    ser->updctrl("PeakHold/pkr/mrs_real/peakStrength", 1.0);
    ser->updctrl("PeakHold/pkr/mrs_real/peakSpacing", 1.0);
    ser->updctrl("PeakHold/pkr/mrs_natural/peakStart", 0);
    ser->updctrl("PeakHold/pkr/mrs_natural/peakEnd", 500);
    ser->updctrl("AudioSink/dest/mrs_bool/initAudio", true);
    ser->updctrl("AudioSource/src/mrs_bool/initAudio", true);
    // use the sample count timer
    ser->addTimer("TmSampleCount","tmr");
    ser->updtimer("TmSampleCount/tmr/MarSystem/source", pkr);
    ser->updtimer("TmSampleCount/tmr/mrs_string/control", "mrs_natural/peakValue");
    // construct the event using predefined musical pitches
    mrs_real notes[] = { C4, G4, E4, F4 };  
    EvPlaySeq* ev = new EvPlaySeq(notes,12);
    ev->setTarget(sine);
    // repeat every tick
    ev->set_repeat(Repeat(1));
    // post the event for dispatch on first tick
    ser->updctrl(TmTime("TmSampleCount/tmr","0"), ev);

    while(true)
        ser->tick();
}
```

Figure 4.10: A network that plays notes on each peak detected.
class EvPlaySeq : public MarEvent {
    protected:
    SineSource* sine_source_;  
    mar_real* notes_; 
    mrs_natural notes_len_; 
    mrs_natural position_; 

    public:
    EvPlaySeq(mrs_real notes[], int len) {
        setSeq(notes, len); 
    }
    virtual ~EvPlaySeq() {};

    void setTarget(SineSource* src){
        sine_source_=src; 
    };

    void setSeq(mrs_real notes[], int len) {
        notes_=notes; 
        notes_len_=len; 
        position_=0; 
    };
    // change frequency of sine source, advance note position
    void dispatch() {
        if (notes_!=NULL && sine_source_!=NULL) {
            sine_source_->updctrl("mrs_real/frequency", 
                                  notes_[position_]); 
            position_++;
            if (position_>=notes_len_) 
                position_=0; 
        }
    };
};

Figure 4.11: Marsyas Event to set the frequency of a SineSource using a sequential list of frequencies.
Chapter 5

Reactive Marsyas

The MarsyasOCaml project was initiated as an investigation into how Marsyas might be redesigned in a functional language. The primary goal was to create a framework that would allow the construction of audio processing networks using functional programming techniques. While a pure functional design was not realized, MarsyasOCaml did introduce a number of ideas that have influenced the current design of Marsyas.

5.1 Design

Casting Marsyas’ dataflow network construction as a functional programming problem is a natural fit. Function composition, central to functional programming, is analogous to dataflow network construction. Unfortunately, the efficient syntax and semantics of high-level functional languages do not usually translate into an efficient practical implementation, at least in comparison to lower level languages like C or C++.

An early goal of the project was to demonstrate that the reimplemented system could be usable in real processing situations and could possibly be made faster than Marsyas. Perhaps this goal was somewhat misguided as it was unlikely that MarsyasOCaml would be able to compete with Marsyas given its well established community of developers and users. At one point MarsyasOCaml showed faster processing speeds for a complex network. However, a subsequent Marsyas redesign eliminated any advantage.

The result of this desire for efficiency had a considerable impact on MarsyasOCaml. The overall design bears considerable resemblance to Marsyas rather than being a radical departure. It is unclear just how much of a departure would have been possible since many
of the features of Marsyas such as network construction, MarSystem controls, and dynamic control modification to name a few were to be maintained in the new implementation. However, there are a number of areas where the redesign offers advantages over Marsyas. The propagation of control changes using a reactive model is more efficient and immediate than the method used in Marsyas.

In reality, MarsyasOCaml became an investigation into the reimplementing Marsyas in a different programming language paradigm where functional techniques were available. It still retains the dataflow network approach and uses functional techniques for network construction. However, a considerable amount of imperative code is used within the MarSystem definitions simply for efficiency. This does not seem unreasonable. Audio processing is a processor intensive application requiring efficient software techniques. Employing the dataflow programming model does not, generally, lead to more efficient software. Instead, it can be a great convenience for the programmer as it is quite natural to think about a processing task as the composition of some smaller set of ordered tasks. In MarsyasOCaml the MarSystem processing tasks use imperative code while the dataflow network construction uses function composition.

MarsyasOCaml benefits greatly from expanding the use of reactivity to the control flow. The dataflow model is reactive in nature. Within a dataflow network the connected objects react to the availability of data on their input by producing processed data on their output. The same idea is applied to controls where a change in a control is propagated to other dependent controls without affecting those controls that are not dependent. Control reactivity is explained in greater detail in Section 5.1.2. Not only can reactivity improve efficiency but it can also be used as a mechanism for scheduling. Scheduling in MarsyasOCaml is explained in Section 5.3.
5.1.1 Implementation

OCaml[16] was chosen as the implementation language due to familiarity with the ML family of languages, its efficient numeric processing (floating point numbers in arrays are not boxed ¹) being similar to C, and the availability of an optimizing compiler. OCaml also has a reasonably good interface for connecting with C code which proved necessary for audio I/O. Additionally, the availability of ML references simplified the addition of reactive controls.

5.1.2 Controls

The most significant departure from Marsyas is the design of the controls system. Controls represent the attributes that define the behaviour of a MarSystem. Some controls are common to each MarSystem. The output buffer dimensions are defined by the \texttt{/natural/onSamples} and \texttt{/natural/onObservations} controls.

In MarsyasOCaml controls are polymorphic in type. This allows for the definition of essentially any type as a control. Added to the basic control types of \texttt{real}, \texttt{natural}, \texttt{string}, and \texttt{bool} are \texttt{Sink} and \texttt{Source} for I/O and \texttt{realvec} representing the buffers used to pass data between MarSystems. Polymorphic control types have since been added to Marsyas through the C++ template mechanism — a significantly more complicated approach though perhaps necessary due to the implementation language.

In Marsyas, controls are members of the C++ class that defines a given MarSystem. Controls that are common to all MarSystems are inherited from the base MarSystem class. Prior to recent changes, controls were accessed by traversing the network using a path notation. This can be seen in Figure 5.1 on line 4 where the \texttt{filename} control of type \texttt{mrs_string} is updated in the \texttt{SoundFileSource} object. On line 10 the \texttt{notEmpty} control of type \texttt{mrs_bool} is read on each iteration of the loop.

¹Boxing means to allocate memory on the heap for storage of a value. This requires an additional pointer dereference in order to access the value which would be very costly for audio processing.
MarSystemManager mng;
MarSystem* pnet = mng.create("Series", "pnet");
pnet->addMarSystem(mng.create("SoundFileSource", "src");
pnet->updctrl("SoundFileSource/src/mrs_string/filename",
    sfName);
pnet->addMarSystem(mng.create("Spectrum", "spk");
pnet->addMarSystem(mng.create("Gain", "g2");

while(pnet->getctrl("SoundFileSource/src/mrs_bool/notEmpty"
    )->to<mrs_bool()>) {
    // tick network and do something
}

Figure 5.1: Marsyas network creation and control

MarsyasOCaml departs from the Marsyas control model by moving the controls from their respective MarSystems into a separate control library. This change is mandated by the fact that MarSystems are functions in MarsyasOCaml. Control values are ‘curried’\(^2\) into the function as parameters and are therefore inaccessible outside of the function scope. It is not possible to ‘walk’ the composed function tree and query or update function parameters as it is with the Marsyas class architecture.

Once the parameters are curried into a function, a closure is made and the parameters can no longer be changed. This poses a problem for the dynamic update requirement for controls. To overcome this, ML reference types are used which resemble pointers in the C language. ML references are always guaranteed to contain a value so that null pointer errors, often seen in C code, are not a possibility. Using references, function parameters can track the current value of a control without having to rebuild the processing function or

---

\(^2\)Currying is a term used often in functional programming. It involves filling in the parameters of a function to create a new function. For example, define the \texttt{add} function with two parameters \texttt{a} and \texttt{b} as \texttt{let add a b = a + b}. This declaration has type \texttt{val add : int -> int -> int} which means \texttt{add} is a function that takes an \texttt{int} as a parameter and returns a function of type \texttt{int -> int}. This type allows the definition of \texttt{add3} by ‘currying’ in the first parameter as in \texttt{let add3 = add 3} to realize the type \texttt{val add3 : int -> int}. \texttt{add3} is now equivalent to \texttt{let add b = 3 + b}.\]
entire network.

The control library mirrors its functional dataflow network counterpart — controls are stored along the same paths as their respective processing functions relative to their neighbours. For example, a frequency control might be stored along the control path `/Series/ser/Oscillator/osc/real/freq`. This design is appealing from a structural point of view though it poses a problem for efficient access to values.

A significant price is paid for looking up a control and reading its value. A naive implementation would involve string parsing and multiple function calls as the request passes through the control tree structure. This can become very expensive if a control value is read each time a buffer of data that passes through the network as on line 10 of Figure 5.1. This process can happen thousands of times for even small data sets resulting in a very negative impact on processing time.

To avoid repeated lookup in the control library, the notion of readers and writers is introduced. Readers and writers are essentially functions with direct access to a particular control value. A program need only access the control library once when requesting a reader or writer. Apart from being far more efficient, readers and writers offer another advantage — control value accesses can be monitored.

Reader and writer functions can contain other tasks such as counting the number of accesses or updating dependencies. This dependency updating is at the heart of reactivity and enables control value changes to flow through a dependency tree. Another advantage is that access to values or permissions can be enforced such as ignoring a write call. The tradeoff for this level of control is a minimum of a function call on each read or write though this is far better than the extensive processing required to access controls in a library.

5.1.2.1 The MarControl

A MarControl is structured as the tuple

(value, listeners, links, reader, (wrt_flag, writer, wrt_bak, wrtwrp_bak))
The first element is the current value of the control stored as an ML reference. This allows the value to be updated without rebuilding the MarControl’s structure. This value is never read or written to directly in order to prevent unmonitored access.

Listeners are a list of value references that are to be updated each time the control changes. Rather than using a reader function which results in a ‘pull’ of the value each time it is required, the listener is ‘pushed’ a value each time an update occurs. Listeners are more efficient. Reading a value reference requires a dereference rather than a function dereference plus a function call when using a reader. However, listeners avoid the read monitoring mechanism.

```
1 MarControls.linkctrl_natural env
2   (prv^"/natural/onSamples")
3   (nxt^"/natural/inSamples")
```

Figure 5.2: Linking two controls

Links are references to functions of type `a -> unit where the polymorphic type `a takes on the same type as the control value. Using the linkctrl functions as in Figure 5.2 the MarControls module will retrieve the writer function for the onSamples control and place it in the links list of the inSamples control.

The reader is a function of type val rdr : unit -> `a that returns the value of the control when called. A reference to it may be retrieved using the MarControls.getrdr function. Customized reader functions that perform a recalculation of the retrieved value or have some side-effect can be created and set with MarControls.setrdr.

The last tuple of the MarControl represents the elements required for writing to a control. Each control has a default writer function of type val wrt : `a -> unit. The writer can be retrieved using the MarControls.getupd library call. It can be changed to another using the MarControls.setupd library calls.
A write wrapper function wraps the writer with another function that performs some side-effect. That side-effect might even block the call to the writer. Decoupling the wrapper function from the actual writer allows the writer to be changed without affecting the wrapper functionality and vice versa.

```ocaml
let upd_onSmpls_wrp (u_outvec:(Realvec.t->unit)ref)
  rd_obsvs
  r_val
  (f_upd:int->unit)
  (new_value:int) =
  f_upd new_value;
let smpls_val = !r_val in
let obsvs_val = !rd_obsvs in
!u_outvec (Realvec.make smpls obsvs);;
```

Figure 5.3: Update wrapper function

The most common use of the wrapper is to keep different controls in sync. As an example, the dimensions of the outvec control are defined by the onSamples and onObservations controls. Figure 5.3 shows a wrapper function to be used with the onSamples control. On line 6 the wrapper calls the onSamples writer function to update the control value. Line 7 reads the updated value rather than rely on `new_value` as there is no way of knowing how `f_upd` actually updated onSamples using `new_value`. Line 8 reads the current onObservations control using its reader function. Finally, on line 9 the outvec control is updated by calling its writer function with the new dimensions.

To allow updating of both the writer and the wrapper, backups may be made to allow rebuilding the composite writer function when changes occur. The `wrt_flag` is used to indicate whether the function is wrapped or not to allow for the correct rebuilding of the writer on updates.
5.2 Control Reactivity

The reactive nature of controls in MarsyasOCaml appears to offer considerable advantage over a non-reactive approach. Only those controls that have inputs or dependencies that change will be updated. In contrast, a non-reactive system would either update all the controls or perform considerable checking to avoid unnecessary updates.

The Marsyas method for updating controls involves setting a “dirty flag” when control values change. While a control can be changed instantly its dependencies cannot. Marsyas has no facility for dynamically linking controls of different types such as between "int/on-Samples" and "realvec/outvec". These must be hard-coded into the MarSystem into the myUpdate function.

On the next network tick after a control value update, the dirty flag prompts a call to the MarSystem update function which checks for and updates all dependencies. This is not a reactive system since the updating of values is dependent on the control flow of flag checking. The fact that the updating of dependencies is delayed is not necessarily a problem. Recall that the Lucid Synchrone language, which is reactive, delays value changes to the next clock tick. This could even be considered an advantage of Marsyas over MarsyasOCaml since it affords some consistency to a control value. If MarsyasOCaml changes a control value during processing then previous MarSystems will have read a different value from a control than later MarSystems will read.

5.2.1 MarSystems

MarSystems are the basic processing objects of Marsyas. In Marsyas, MarSystems are classes with two basic tasks: object creation and buffer processing. In MarsyasOCaml, MarSystems are modules that contain at least two functions: create and process. Figure 5.4 shows an example of a MarSystem module for adjusting the gain of an audio signal.

The create function performs two basic operations. Firstly, it initializes and adds all the controls required for the new MarSystem to a new controls library. Secondly, it creates
```ocaml
let process (g:float ref)
  (outvec:Realvec.tref)
  (name:string)
  ((invec,env):MarSystem.t_vec_env)
: MarSystem.t_proc_ret =
Realvec.pair_iter (fun x -> x *. !g) invec outvec;
(outvec,env);;

let create name parameters =
  let env = MarSystem.create name parameters
  [(name^"/real/gain", MarControls.Real 1.0)] in
  (* register listeners *)
  let outvec = MarControls.getlstnr_realvec
    env (name^"/realvec/outvec") name in
  let gain = MarControls.getlstnr_real
    env (name^"/real/gain") name in
  (name,process gain outvec,env,[]);;
```

Figure 5.4: MarsyasOCaml Gain Module

A new process function by currying in the necessary parameters. The new process function
and environment are returned. In Figure 5.4 the create function also retrieves listeners, ML
references, for the output realvec array and the gain control which are then curried into the
process function.

The process function is called on each network tick with a buffer of data and the
control environment. It then processes that data and copies it into the output buffer before
returning. The input buffer is always the output buffer of the previous MarSystem. In
Figure 5.4 each sample is multiplied by the gain control amount. Note that in OCaml the
syntax !g indicates that g is to be dereferenced to obtain its value.

5.2.2 Network Construction

Networks are built by composing process functions. The preferred way to accomplish
this is by using the MarSystem Manager module which acts as a MarSystem factory. Two
basic function calls are supported. The `MSMng.marsys` function is used to create basic MarSystems and takes as strings the MarSystem type and name as well as initial control values as parameters. `MSMng.composite` is used to create composite processes such as Series and Fanout. It takes a MarSystem type and name, initial control values list, and a list of MarSystem processes that it should contain. An example of network construction can be seen in Figure 5.5.

```ocaml
let (process, env) =
  MSMng.construct [
    MSMng.composite "Series" "fnet" [] [
      MSMng.marsys "WAVSource" "src"
      (["/string/filename", MarControls.Str "ip.wav"]);
      MSMng.composite "Series" "spectralShape" [] [
        MSMng.marsys "Hamming" "hamming" [];
        MSMng.marsys "Spectrum" "spk" [];
        MSMng.marsys "PowerSpectrum" "pspk" [];
      ]
      MSMng.composite "FanOut" "specDesc" [] [
        MSMng.marsys "Centroid" "cntrd" [];
        MSMng.marsys "Rolloff" "rolloff" [];
      ]
    ]
  ];
```

Figure 5.5: MarsyasOCaml Network Construction

### 5.3 Scheduling in MarsyasOCaml

Unlike Marsyas, MarsyasOCaml does not contain a scheduling unit. However, this does not mean that it is not capable of scheduling events. In fact, the reactive control value design of MarsyasOCaml supports most of the functionality that a scheduler would include. Control values can become timers while access to control values is through reader and writer functions that can be used for event dispatch. If a scheduler were added it would
simply provide the functionality of a factory for producing and posting events.

5.3.1 Timers

Some controls are already timers. An audio input MarSystem will have a sample count control. Still other controls, particularly those that do not always increase, may be used as part of a timer. For example, the number of times a control is accessed can be counted as a side-effect of the writer function. Adding the ability to dispatch events means implementing a function to manage the event queue and adding it to either the link functions or the function writer itself.

Consider the example scheduler of Figure 5.6. The timer function, supplied an event list, can be added to a control using MarControls.reg_fun. Each time the control is written to the timer function will be called. On each call the event queue is checked and those events that have a count of less than one will be dispatched. The timer count is therefore implicit and no timer variable needs to be created.

```ocaml
1 type sched_evnts = (int * (unit -> unit)) list
2
3 let timer (xs:sched_evnts ref) value : unit =
4   xs := List.fold_right
5     (fun (t,f) xs ->
6      if t<1 then (f(); xs)
7      else (t-1,f)::xs)
8     !xs []

Figure 5.6: A possible MarsyasOCaml Scheduler function
```

It would be most likely that we would use the sample count control of the MarSystem supplying the input signal. Usually this is an AudioSource in the case of a wired input or a SoundFileSource when reading from disk. This control is updated on each realvec buffer boundary. It would be fairly straightforward to modify the timer function of Figure 5.6 to dispatch events according to the sample count.
Controlling events according to the real-time system clock becomes a little more difficult since no control keeps track of system time. There are two ways that a system timer could be added. A new `timer` function could be created to read the system time. It would then have to be registered with a control that is updated on each buffer passing through the network. An alternate method would be to create a new MarSystem with a system time control that could be updated on each buffer passing through the network. A similar function to the one above could then be registered on the new control.

### 5.3.2 Events

What good are timers if there’s nothing to do when they tick? As defined earlier in 2.2 an event is simply something that happens at a point in time. In the Marsyas scheduler events are represented as objects with a single dispatch method that takes no parameters (refer to Figure 4.5). In MarsyasOCaml there is no need for an explicit event abstraction. Events are completely user-definable but will most likely be represented by a function call. In the simple scheduler of Figure 5.6 the events are functions of type `unit -> unit`. In MarsyasOCaml function pointers can be passed in the same way as objects may be passed in Marsyas.

### 5.4 Summary

While the MarSystem dataflow design of MarsyasOCaml is similar to Marsyas the control flow described here is an important departure. Reactive controls unify data and control under the dataflow model. Control changes propagate through linked controls and interested listeners as they happen. Non-dependent controls are unaffected.

An important byproduct of the reactive design presented here is the ability to do event scheduling with minimal effort. This becomes possible by enforcing the rule that control value updates must go through the writer function. Control values can then act as timers by responding to value updates. A timer can then be based on the control value itself or on
a count of the number of read or write accesses. Event management requires some basic implementation but is easily added to the system through the same method as for control value propagation.
Chapter 6

Conclusion

This thesis has presented a discussion of computational time with a focus on audio processing. In any system where actions are to be performed at specific points in time an appropriate model of time must be defined. Decisions about the shape of this model will ultimately have to consider the temporal properties discussed in Chapter 2.

These temporal properties that govern time-based computation were discovered by surveying the related literature and environments presented in Chapter 3 and through experimentation with the two systems designed as part of this thesis. For each of these properties I have described a number of issues related to their implementation. Understanding and balancing these trade-offs is key to developing an appropriate temporal model.

Two systems were developed during this thesis work. The Marsyas Scheduler presented in Chapter 4 was designed to fit within an existing software framework. Given that the mature Marsyas design forced certain choices to be made the resulting scheduler has proven to be very flexible. The key to this flexibility is that it avoids placing unnecessary limits on the kinds of timers or events that the user wishes to define.

MarsyasOCaml, presented in Chapter 5, departs from the Marsyas design by incorporating a new model of time based on reactivity. The resulting framework is more powerful for time-aware tasks than Marsyas because each control has the inherent ability to act as a timer upon which events may be scheduled. This effectively distributes event scheduling to the source of the timer on which the event is scheduled.

Both of the systems presented here offer advantages over other audio processing systems with respect to the timed dispatch of events. Pure Data and ChucK use a single time
reference based on the audio sample rate. While it is possible to create other references within those environments, they will ultimately be second-class citizens compared to the master reference. In contrast, all timers in Marsyas are created equal and each may define their own methods for specifying time.

6.1 Future Work

Implementing what is essentially a programming environment on top of an existing programming language has its limitations. In the case of both the Marsyas Scheduler and MarsyasOCaml, interacting with the framework requires formulating statements within the rules of the implementation language. For example the expression ‘12 milliseconds’ has no analog in C++ or OCaml. Instead the time value must be defined using the available language domain constructs such as a string value "12ms" which must be parsed at runtime, a C++ class type Time(12,"us"), an ML type type time = Time of int * string or by some other means.

Ultimately, it would be desirable to design a new Marsyas programming language with its own built-in model of time. Clear and concise syntax could be chosen for expressing dataflow and task scheduling. Time itself could become a type along with the programmatic means to define timers and perform actions on these timers. One of the more significant motivations for such a project would be the ability to utilize domain specific knowledge within the compiler to affect areas such as type checking and improved program optimizations. Efficiency optimizations are important for supporting higher resolution timers.

As outlined in Chapter 3 many languages already exist for time aware computation. The Lucid language shows how carefully chosen keywords for describing computation order can benefit code readability. The ChucK language adds time as an integral type and benefits from a clear method for specifying time and advancing computation. However, the audio languages surveyed only supported a single notion of time based on the sample
rate. Sequential languages like Lucid Synchrone and Esterel are more flexible in terms of supported timers but are not necessarily efficient choices for audio or media processing. There may be room for yet another programming language.
Bibliography


