Software Systems for a DNA Sequencer

by

Ahmed Ait-Ghezala

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Electrical Engineering and Computer Science and Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Techonology

September 1, 2000

Author _____
Department of Electrical Engineering and Computer Science
September 1, 2000

Certified by _____
Prof. Paul Matsudaira
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, department Committee on Graduate Theses

Massachusetts Institute of Technology

Abstract

Software Systems for a DNA Sequencer
by
Ahmed Ait-Ghezala

Submitted to the
Department of Electrical Engineering and Computer Science

September 1, 2000

In partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science and Master of Electrical Engineering and Computer Science

ABSTRACT

The initiative to complete the sequencing of the human genome is bringing the need for high-throughput sequencing capabilities to the forefront. We at the BioMEMS engineering group at the Whitehead Institute are designing and building a new sequencing machine that uses a 384 glass "chip" to dramatically increase sequencing rates. This thesis describes the design and implementation of two of the machine's software components. The first is a prototype application for the control of a robot used to automate sample loading. The second is a software filter that allows us to generate quality scores from data processed by Trout using Phred. I present the algorithm used to perform the filtering and show that the results are comparable to the processing of data with the Plan-Phred processing package.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

In the Name of God, the Beneficent, the Merciful. I would first like to thank Sameh El-Difrawy for all of his advice and support. Thank you to Professor Matsudaira and Dan Ehrlich for giving me the opportunity to work on this project. Thank you also to Dieter Schmalzing, Aram Adourian and Mark Novotny for the fruitful discussions on the chemistry of base-calling.

Last but not least, I would like to thank my family and most importantly my mother to whom I dedicate this thesis. I thank her for the encouragement that she has always given me, and the sacrifices made to get me where I am today.

Chapter 1

INTRODUCTION

1.1 The Human Genome Project

The Human Genome Project (HGP) is an international effort coordinated by the U.S Department of Energy and the National Institute of Health (NIH). It started in 1990 as a 15 year effort aimed at: identifying the more that 100,000 genes (the human genome) that constitute 0the human DNA, determining the sequence of chemical base pairs that form these genes, storing this information, and building tools to access this information in an efficient manner to allow for data analysis [1]. Researchers working on the HGP have their sights on a completed sequence of the human genome by 2003, and already have a rough draft of the sequence.

The project's early years put a lot of effort into optimizing the existing methods and developing new technologies to increase DNA sequencing efficiency [1]. Work completed in the last 3 years at the Whitehead Institute for Biomedical Research has extended this technology by allowing the use of micro-fabricated electrophoretic devices for sequencing and genotyping [2]. These micro-devices have intricate micro-channels cut into glass, allowing for many more sequencing lanes to exist on one plate than before, in effect increasing sample throughput. We are currently developing a DNA sequencer that utilizes 384 such micro-channels on one plate. We are also developing tools for improved accuracy evaluation.

1.2 DNA, sequencing and base-calling

A human chromosome has around 10^8 base pairs (bp). On the other hand, the largest piece of DNA that can be sequenced in the laboratory with current technology is approximately 800bp long. Each strand of DNA is made up of a helix of complementary base pairs. At present nearly all DNA sequencing is done using the enzymatic dideoxy chain-termination method of Sanger [4]. In this method, a single strand of DNA (amplified using polymerase chain reaction) has a primer reacted to a specific region on it from which the sequencing will begin. An enzyme called DNA polymerase (an enzyme that catalyzes the elongation of any given strand of DNA) is used to react the cloned strands with a mixture of 4 bases A, C, G and T; the bases that make up DNA. This reaction is terminated at varying points along the original DNA template strand with fluorescent dye molecules hence forming a mixture of labeled, single-stranded fragments of varying lengths, each complementary to a segment of the original template strand and extending from the primer to the occurrence of a base.

These samples are then placed into a gel medium and separated along the micro-channels described above using a strong electric field formed by an applied voltage across the ends of the channel. Since DNA is negatively charged the strands migrate along the gel under the application of the electric field with a speed that is approximately inversely proportional to their size. As the fragments reach the end of a channel, a laser beam that scans channels at a fixed frequency excites the dye molecules terminating the DNA fragments and a set of four photo multiplier tubes (PMTs) collect the emission intensities at four different wavelengths.

The raw data obtained from the wavelength reads is processed to generate a sequence of bases that represent the original DNA template.

1.3 Aims of our work

The aim of the Whitehead Institute BioMEMS Engineer Group is to design and fabricate a fully functional sequencer. This thesis implements control software for a robot that automates sample loading in the sequencing machine under development. In addition the design and implementation of a software filter to allow for base-calling of the data obtained from the sequencer using third party software.

Chapter 2 explains the problem at hand. In section 2.1 I discuss the sequencer machine hardware in addition to the software aspects of data acquisition, machine, and robot control. In section 2.2 I discuss the problem of base-calling with confidence estimation.

In chapter 3 I present the robot control software specifications that were needed, then go through the design and implementation process that was undertaken to satisfy these requirements. In short, by following the well-understood concept of the software life cycle I will describe the design and implementation of the control software. In addition, I discuss how this method of development was suited to our project and suggest ways of changing the cycle to better suit software development in a project with similar characteristics (in both time constraints and requirements) as ours.

In Chapter 4 I discuss the design and development of a software filter that maps the output of Trout, a base-calling software package developed at the Whitehead, to a trace that is suitable for processing by another third-party software package, Phred, in order to generate confidence levels.

Chapter 2

PROBLEM STATEMENT

This chapter consists of two sections. The first discusses the sequencer machine and the general requirements of the software necessary for its operation. The second section of this chapter introduces the base-calling software and the problem I am solving.

2.1 Sequencer machine

2.1.1 Sequencing Plate

The 384-lane glass "chip" is at the core of the sequencing technology being developed in our lab. The chip is made of two glass plates. Wet chemical etching creates the channels in one sheet of glass by defining channels with the required geometry using photolithographic techniques developed in the lab [5]. Access holes to the channels are then drilled using a laser-drilling technique in the other plate. The plates are then bonded together to enclose the channels. The channels converge at the base of the chip for scanning by the spinner laser. This sequencing technology is expected to dramatically increase the daily throughput of base-calls as shown in Table 2.1

	Lanes/system	Bases/lane	Run time (hr)	Runs/day	Bases/day
ABI 377 Slab gel	96	800	8	2	153600
ABI 3700 capillary	96	500-600	2	10	576000
Micro device	384	600-800	0.83	26	7987200

Table 2.1. Improvements with the new microchip array for sequencing over older technology.

2.1.4 Robot hardware and software

To automate the process of loading samples into 384 lanes, the group designed and built a robot (refer to Figure 2.2.) The robot is made of a pipette head and four axes: X, Y, Z and θ.

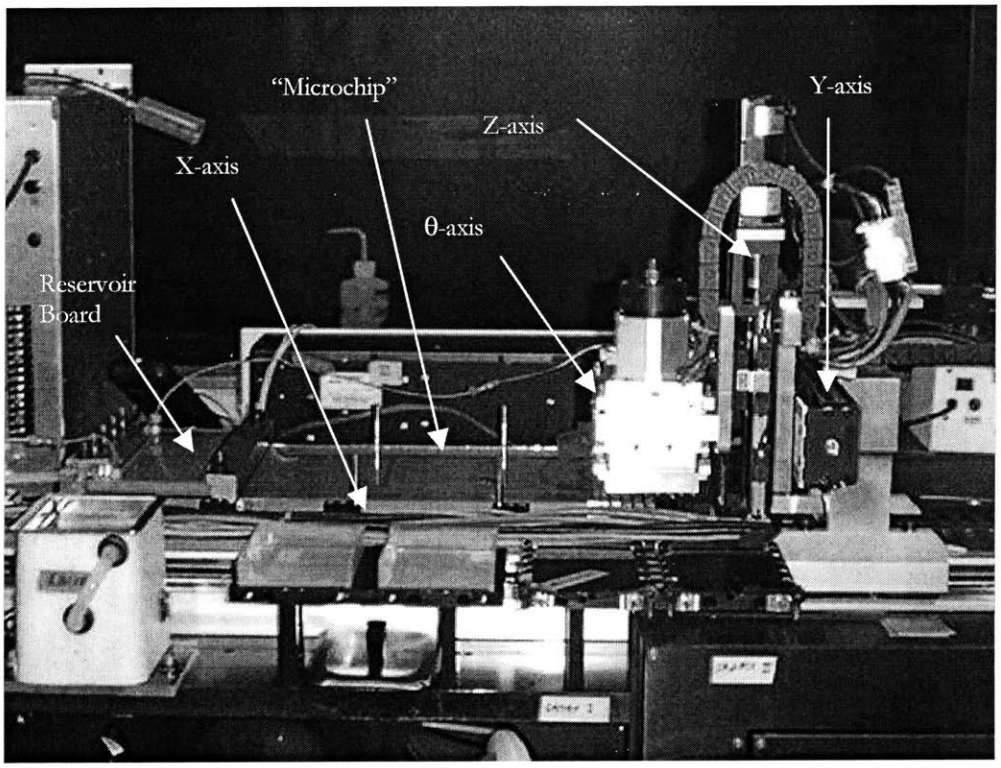


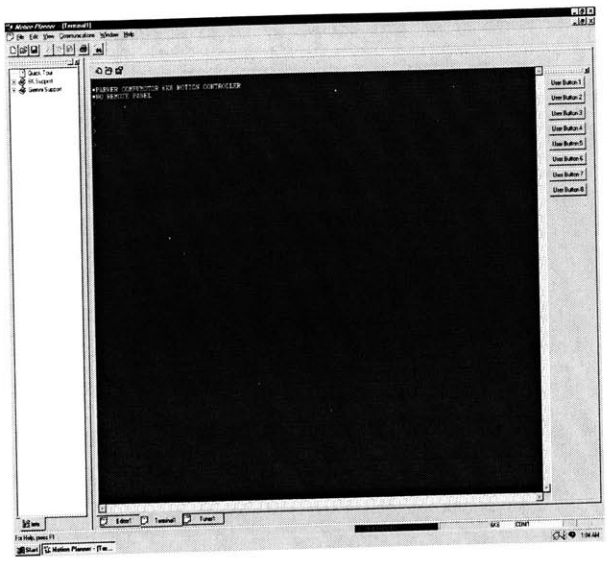
Figure 2.2. Robot.

The X and Y axes are driven by Yaskawa Sigma Series SGMP 3000rpm, 0.13hp servo motors with mounted breaks, while the Z axis is driven by a Yaskawa Sigma II Series SGMAH 3000rpm servo motor also with mounted break. The X and Y servomotors are controlled by a Yaskawa Sigma Series SGDA speed/torque control servo amplifier. The Z axis motor is controlled by a Yaskawa Sigma II series SGDH torque, speed and position control servo amplifier. The servo amplifiers are in turn connected to a Parker Automation 6K 8-axis controller. This controller is connected to a PC through a serial port. Each axis (except for the θ axis) has a home switch and an end of travel limit

switch. Both are Hall effect switches that are triggered by a metal attachment on the movable part of an axis. The home switch is positioned at the desired home location along the axes and the end of travel switch is placed at the points beyond which motion is not permissible on an axis. The θ axis is a custom built pipettor with 96 needles and is the component used to move samples around the sequencing machine.

The X, Y and Z axes move the pipette head in their specified direction. Directing the θ axis allows the 96 needles of the pipette head to aspirate and inject water, buffer and DNA sample into the reservoir base of the sequencing plate. The 6K controller comes with a windows-based graphical interface called Motion Planner, that allows a user to test and control the robot from a command shell interface. The Motion Planner software comes with a comprehensive library of low-level functions that control the robot and set various run motion parameters. However the program does not provide a user-friendly interface and requires extensive knowledge of low-level robot parameter details, making it difficult for a non-specialized user to work with. The program also lacks flexibility for our testing purposes. For example, we must be able to stop and start the robot at different times, to rearrange the steps of the loading protocol and to define completely new protocols on the fly. In other words the notion of a *typical protocol* was not completely defined from the start and testing had to be performed to find out what exactly was required. The Motion Planner interface requires rewriting programs each time we want to change a protocol (Figure 3.2); additionally the interface doesn't give the user the ability to pause the run and restart it at a point further ahead.

User interface provided is command line. Programs can be uploaded but require knowledge of the library functions and are very inflexible for development purposes. A simple run which requires two actions (program shown here) needs to have loops rewritten if the run order is changed making it very impractical.



```
repeat
var1=var1+4.5
var2 = var2+4.5;
;wash station
;reach
d(var1) (var2),,
go11,,
;lower
d,,(var4)
go,,1,
;wait
t20
;raise
d,,(var3)
go,,1,
;Sampleload
;reach
d(var5)(var6),,
go11,,
;lower
d,,(var8),
go,,1,
;aspirate
d,,,(var18)
go,,,1
;raise
d,,(var7),
go,,1
;end loop
until(var19=4)
end
```

```
repeat
var1=var1+4.5
var2 = var2+4.5;
;Sampleload
;reach
d(var5)(var6),,
go11,,
;lower
d,,(var8),
go,,1,
;aspirate
d,,,(var18)
go,,,1
;raise
d,,(var7),
go,,1
;wash station
;reach
d(var1) (var2),,
go11,,
;lower
d,,(var4)
go,,1,
;wait
t20
;raise
d,,(var3)
go,,1,
;end loop
until(var19=4)
end
```

Figure 2.3. Command shell UI of Motion Planner + Sample program to demonstrate need to reorganize loops making it very inflexible for simple reordering of events and useful mostly for testing and calibration purposes.

2.1.5 Automation of run

To automate the entire sequencing run we developed a software package called the Housekeeper. This package integrates all the steps of a sequencing run (Figure 2.4) including the sample loading steps performed by the robot and the operation of different pumps and valves. A second package called the Sequencer was also developed to control the data acquisition operation.

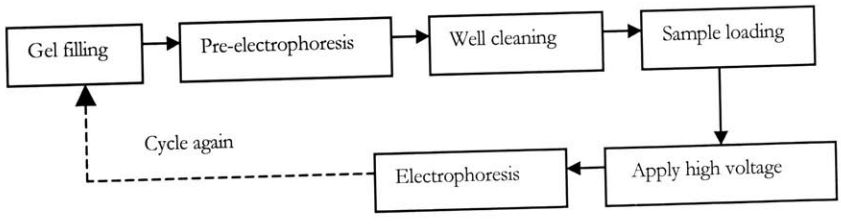


Figure 2.4. Automated run cycle protocol.

Although our initial design incorporated the robot control routines into the Housekeeper design, it became clear that this was not suitable for testing purposes and did not provide an intuitive interface for the robot because it relied on the development of an instruction file offline, which could then be fed into the Housekeeper for processing (thus providing minimal user functionality once a run started with the Housekeeper.) We therefore decided to separate robot functionality from the rest of the software. It was clear that the robot operations were first to execute in the run protocol and did not require coordination with other events. The design of this stand-alone software package is the focus of the following chapter.

2.2 The base-calling effort

The ability to generate base calls with their associated quality scores is an important feature of any base calling software. These quality scores are used by sequence assembly programs to evaluate the relative quality of different segments of a sequence and decide accordingly whether or not to include those segments in the finished product. Currently, the Phred base caller is the industry standard base-caller as it is capable of generating base calls with quality scores. A quality score for a called base is defined to be:

$$q = -10 \times \log_{10}(p)$$

Where p is an estimate of the probability that the called base is erroneous, which means that a higher quality score equates to a lower probability of error at that position [6]. Phred is tuned to work with chromatogram files generated by the ABI373 and ABI377 slab gel machines and preprocessed by either ABI preprocessing software or by *Plan*.

Trout is preprocessing and base-calling software that was developed locally at the Whitehead Institute to base-call ABI373 and ABI377 files. Our group has

successfully been able to tune Trout to work with the output of the micro-channel based system we have developed and we were able to report read lengths of up to 800 bases with only 2% error rate [7]. However, Trout doesn't produce quality scores. The problem therefore boils down to developing a software package that gives meaningful confidence scores specifically for the chemistry we are using and does the preprocessing and base calling at the same time.

Instead of reinventing the wheel, and in the spirit of software-reuse I combined the Phred and Trout software packages as the core of a final product application but with the following proposed modification: linking the preprocessing stage of Trout to a special Trout-to-Phred filter stage that prepares the preprocessed Trout output and passes it to Phred which can then call bases and generate confidence estimation levels. Future work will likely involve the implementation of a separate confidence estimation package for Trout.

A flow diagram of these two approaches is shown below Figure 2.5. It is hoped that implementing such a software tool will maintain a much-needed balance between advancements in sequencing hardware and speed – which our microchip based sequencer is doing – and software.

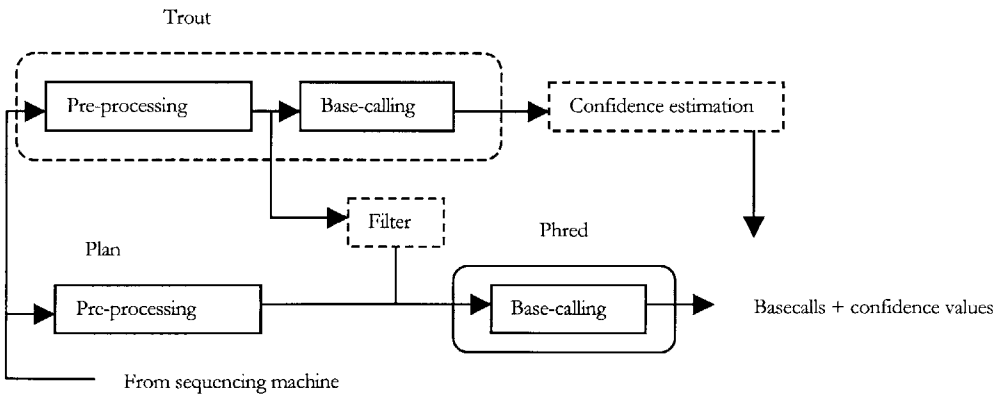


Figure 2.5. Flow diagram of two options of software packaging to base call with confidence estimation.

Chapter 3

ROBOT CONTROL SOFTWARE & PROTOTYPING

Before describing the robot control software, to see the differences in our development cycle I will describe some basic theory of the software life cycle and how prototyping fits into the model.

3.1 The standard software development cycle

The standard software life cycle model states that software development should consist of five distinct stages: requirements analysis, specifications, design, implementation, and verification. Requirements analysis is concerned with deriving, from the intended user, what the expected use and functionality of the system should be. It involves direct contact with the user to shape the idea. Specifications involve stating the functionality and constraints of the system in a very precise and unambiguous way. This may be in the form of a mathematical notation or a textual description. Design is coming up with a solution that satisfies the specifications and then modifying it if the need arises. Implementation is realizing the design in a programming language that can be executed on the target machine. Finally, validation is the process of checking the implementation to ensure that it has fulfilled the specifications laid out by the user.

Software development using this standard model may be disadvantageous however because of the following [8] :

1. The earlier an activity occurs in a project the less we understand about the nature of that activity.

2. The earlier an activity occurs in a project the poorer the notations used for that activity.

3. The earlier in a project that an error is made, the more catastrophic the error is. So for example early requirements and specifications errors have typically cost a hundred to a thousand times more that those made during the implementation and have in fact led to many projects being cancelled completely after years of effort in development.

4. Because a user is unable to visualize the final results of a system that has been formally described in a specification document by the programmer, it often becomes very hard to validate the work that comes very early on by checking with the end user.

5. Finally, an important aspect of the work being done in our group was the need for rapid development as the full functionality of the software empowered the rest of the group to evaluate and modify other important elements of the system.

3.2 Prototyping

The solution to many of the problems mentioned above is software prototyping; a method of development that is usually used at the early stages of full development and considered a learning stage that involves a lot of interaction between the user and developer. For this stage to be successful, feedback from the user about desired features is essential and the interaction with the developer will give the user himself a better understanding of the available capabilities.

Of the available and established prototyping methodologies, I chose to follow the evolutionary model. Evolutionary prototyping argues that the system starts as one entity and slowly evolves into another entity adding and perhaps invalidating original requirements as new information about the system comes to light. In effect, it allows the functionality of a software system to be introduced incrementally with regards to the final version of the software. However the

difficulty with this implementation methodology (which could later in the development cycle become an advantage) is that it requires the design to be flexible enough to allow for continuous changes during and after the implementation stage. Additionally, prototypes in general do not necessarily implement all the features of the final version.

3.2.2 Prototype objectives

In our work the required functionality was not clear and kept evolving with time. The constant changes in hardware and procedures made our selection of a prototype approach a natural one. The specific motivations behind a prototype approach were:

1. Need for rapid development.
2. Specifications laid out by the user (in this case a chemist) who was testing the sequencing machine kept changing with time as changes in hardware and loading protocol occurred. Such changes were expected of a new experimental system of hardware components.
3. Need to analyze the feasibility of features specified in the initial specifications and during the course of development. A sort of complementary tool to the software design and development stages of the software life cycle.

3.3 Prototype

The prototyping of the application ended up being a recursive cycle of development with each part of the cycle consisting of modifications to the original specifications. In the following discussion, a *cycle* or *run protocol* refers to the set of ordered steps (each with an associated location) taken by the robot during a sequencing run.

The first version I developed, v1.0, was based on the following simple specifications:

1. The robot had to, on user demand, be movable to four separate locations on the sequencing apparatus. The user should be able to set each such location.
2. The user needed manual control over the robot, giving him the ability to move it at will to specific locations on the sequencing apparatus.

With these specifications in mind, the UI that was developed is shown Figure 3.1. It provided a simple interface that met the initial requirements mentioned above.

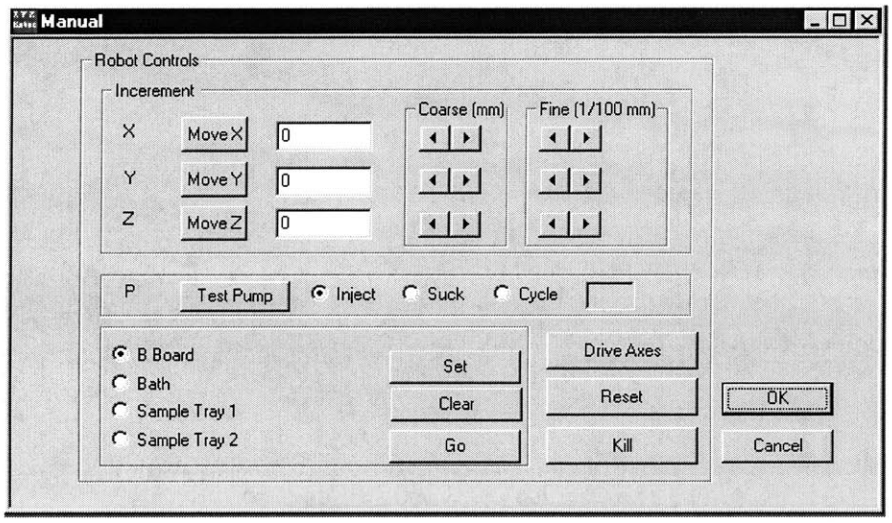


Figure 3.1. Version 1.0 user interface.

After initial testing by the user, the following issues were raised and needed addressing:

1. There were in fact many more locations the robot would need to visit.
2. The notion of a cycle was introduced to represent the idea that the robot can visit different locations in a defined sequence and execute a certain pipette action at each location.

The fastest way to incorporate more robot positions was to introduce the notion of a position cycle which could be made up of any number of uniquely identifiable positions. This foresaw (correctly) the idea that elements of a cycle may have to be removed or added at any position in a run protocol. In addition, it made the idea of many positions more feasible because a list of items is essentially limited only by disk space. However, the ability to create an infinite number of positions to which the robot could move to also meant changes in the GUI because the dynamic nature of the cycle creation meant that unlike the V1.0 GUI, the positions could not be viewed and activated by static buttons and edit boxes.

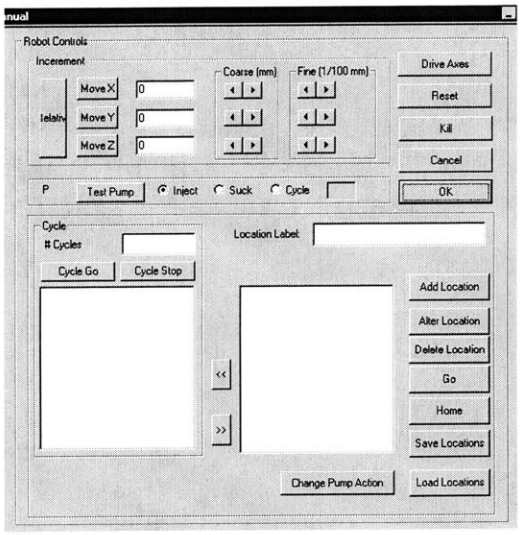


Figure 3.2. V2.0 User Interface.

To allow for differentiation between different positions on the board, each position would have a unique label associated with it (e.g. Waste, Buffer, Sample1 etc.) The GUI was also modified to give simpler manual movement control on all the axes by adding fine control capability. The product of these modifications is shown in Figure 3.2.

The system was then tested for some basic functionality by the user. This testing introduced more required functionality.

1. The pipettor needed to be programmed to aspirate and inject programmable volumes of liquid, then cycle over this operation a user-set number of times.
2. The user wanted the ability to stop and restart a cycle at any time and at any position in the cycle in addition to having a visual indication of the current step of the cycle being executed by the robot.

The second specification required the program to communicate continuously with the robot controller in a master-slave fashion. Requests to execute instructions sent by the master (the program) to the slave (robot controller) had to be synchronized because execution of the master's command by the robot controller often involved motion of several mechanical components. If command synchronization, as illustrated in Figure 3.3, is not accurately achieved commands could be skipped causing mishaps. In Figure 3.3 command B is not processed because it arrived while A is being serviced.

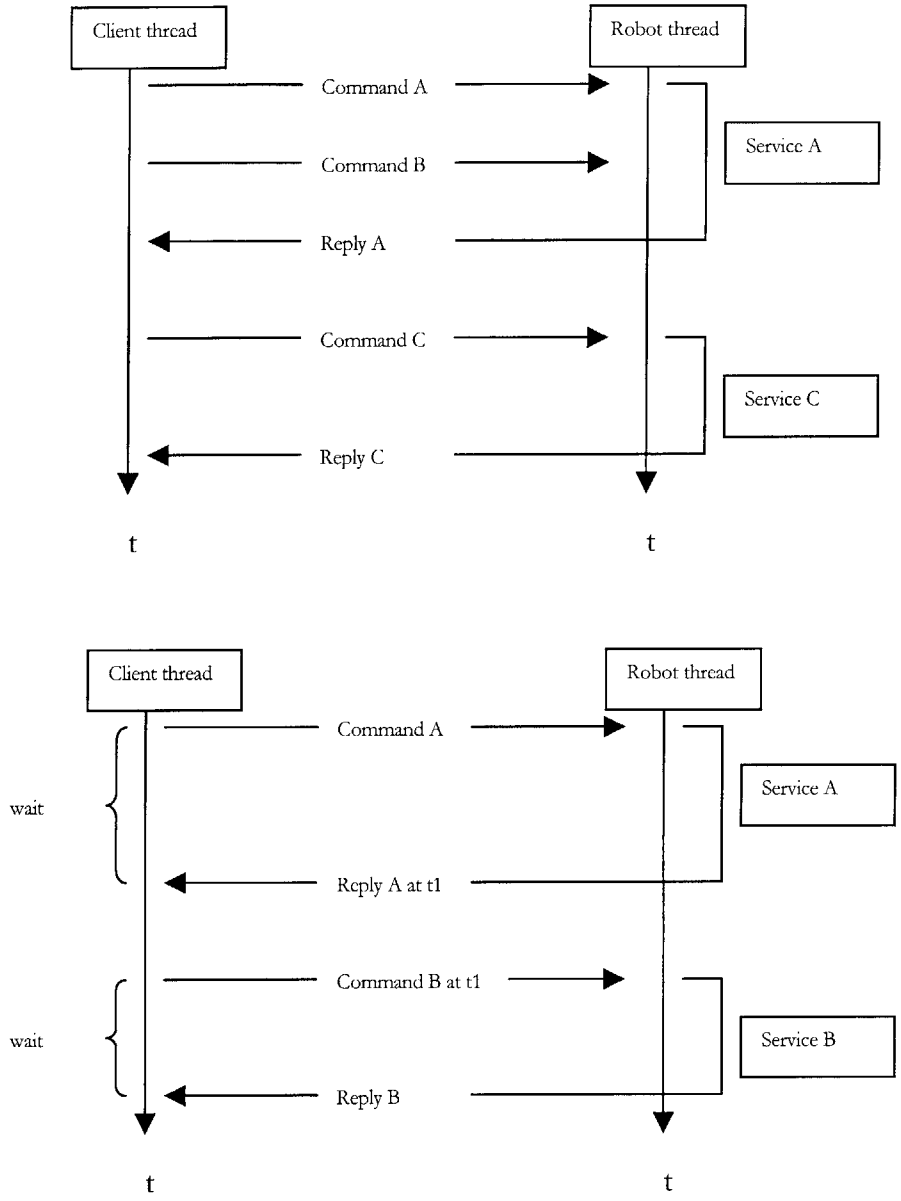


Figure 3.3. Controlling command synchronization.

The final version V3.0 saw the following specifications come to light:

1. Need to store cycles and load them at will.
2. Need to store calculated positions, which could later be used to create cycles.

And had the GUI shown in Figure 3.4.

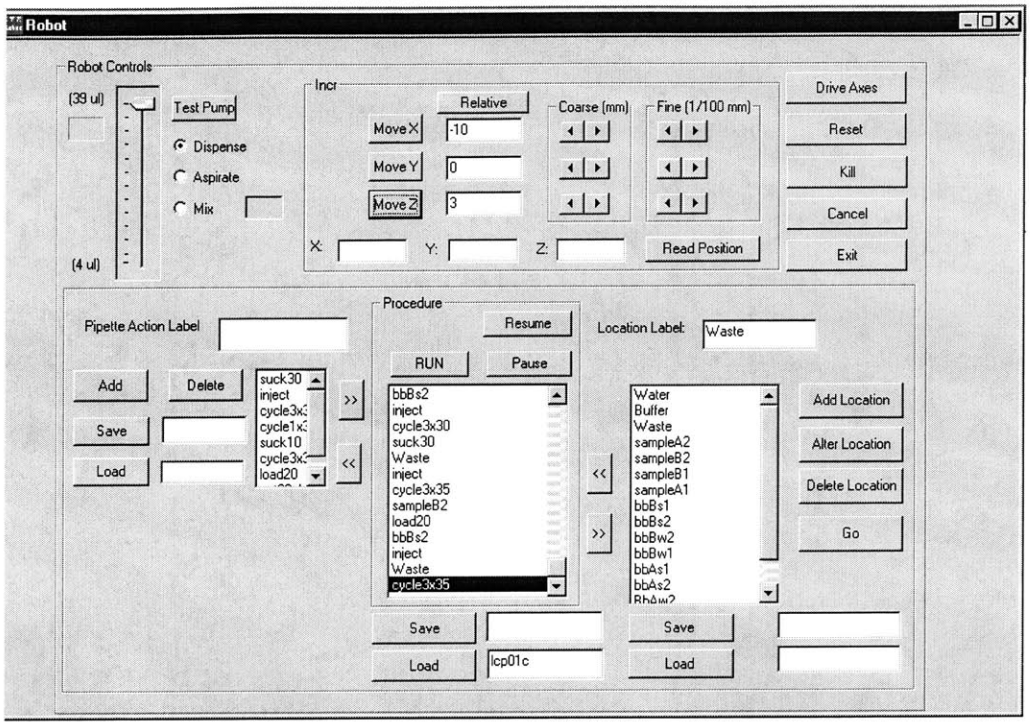


Figure 3.4. Final GUI V3.0.

3.5 Object Model

Having come up with a prototype implementation that was tested and shown to be satisfactory by the user, I developed a design using what was learned during the prototype development to come up with a modular design that is more easily maintainable. The object model that was developed is shown in Figure 3.4. The robot contains four axes, each of which has a pre defined home position that is valid for a run. It is not possible to capture this relationship within the diagram but it is an important consideration because once the robot is reset, the home position must also be reset, i.e. there is no stored state between robot resetting. Associated with the axes are a velocity and acceleration. There are many more parameters that can be set, but these two are the ones I consider first. The home Position object encapsulates the idea that each axis has a fixed associated home location, while the xPosition, yPosition, zPosition and thetaPosition objects

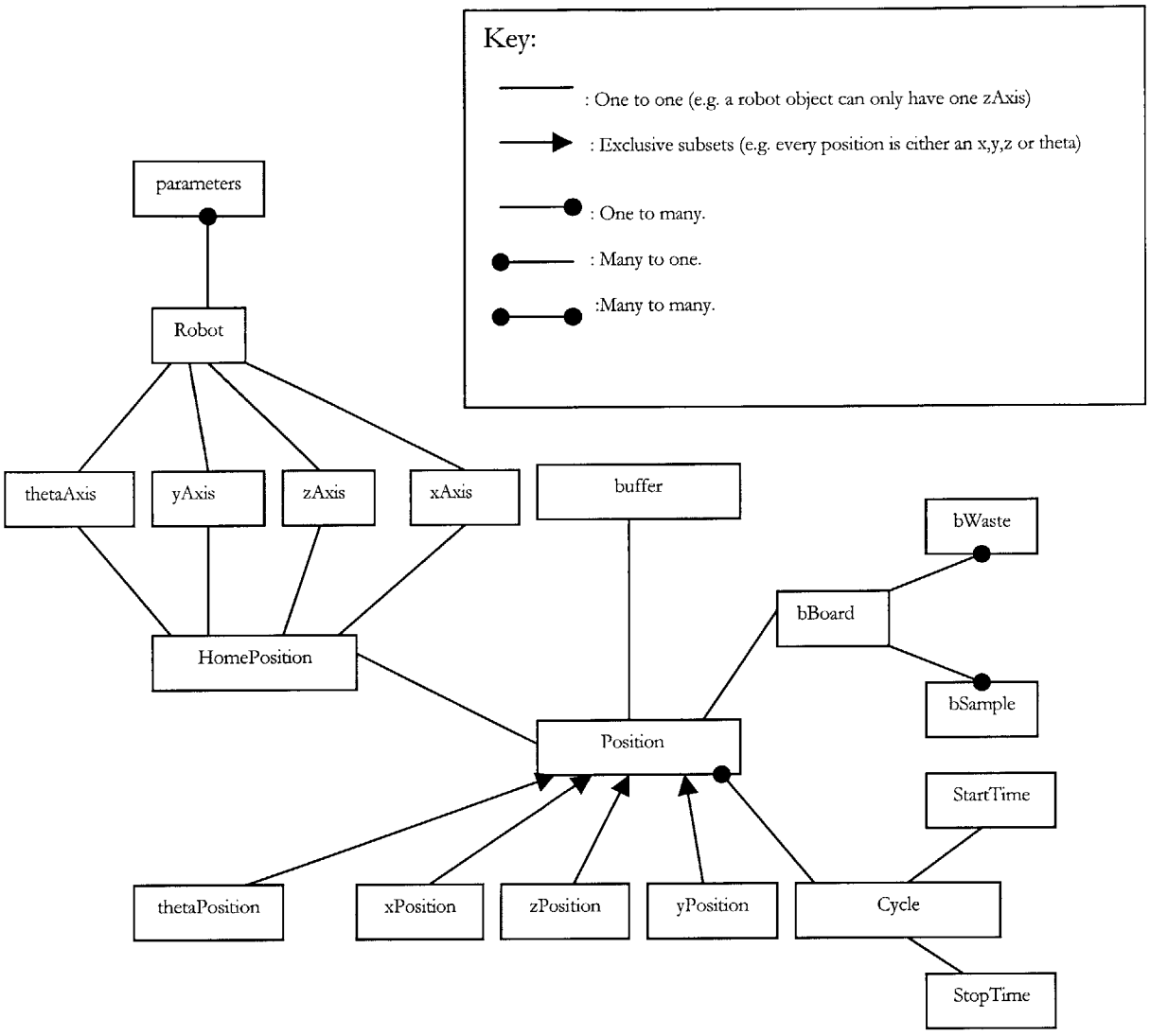


Figure 3.4. Object model for the robot control software.

incorporate the idea of a time variable position (which varies depending on what step in the run protocol we are at also) also associated with the axes.

The Reservoir Board (rboard) into which samples are injected for electrophoresis, the buffer solution container, which holds the buffer that is needed for pre-electrophoresis are also objects that have a position associated with them. The rboard has two types of associated positions; the waste wells positions, and the sample wells positions.

Lastly there has to be a notion of a cycle in the picture of things. A cycle, as described above, is defined by the positions that it must cycle through. Additionally a cycle has an associated start and stop time.

3.6 Code Model

Having generated the object model, which displays the major components of the system it is time to look at the code model, which displays objects from an implementation point of view, i.e. what objects are necessary for the development of a final product. The code model user is shown in Figure 3.5. The code model allows us to look at the interaction between the different components of the system. The discussion that follows is on the main model objects, and why the model takes the form it has.

Apparatus: Introduces the notion of a physical component of the machine. Apparatus is an abstract class, and is sub-classed by Robot, which is the biggest single machine component. The robot itself is made up of four axes, but these are contained within the Robot object.

Runner: The runner is the "main" of the software tool. It is the class that coordinates the action between the different components. The runner receives events from the GUI and acts upon them causing the robot to carry out a manual action or run through a sequence of steps by calling the compiler which generates the program that needs running. **GUI:** The user interface class provides an interface for the user for the system to the system. The GUI communicates with the runner using and event system. So for example any action on the user interface will trigger an event, which is sent to the handling runner. **Cycle:** This class supports the notion of a run a cycle. A cycle has two types of actions: a PositionAction and a PipetteAction. The separation of these two actions was necessary because, as described earlier, it became clear that the movement of the

θ axis was not only associated with a position but also a cycle of aspirations and injections of sample. To reduce the number of Position objects

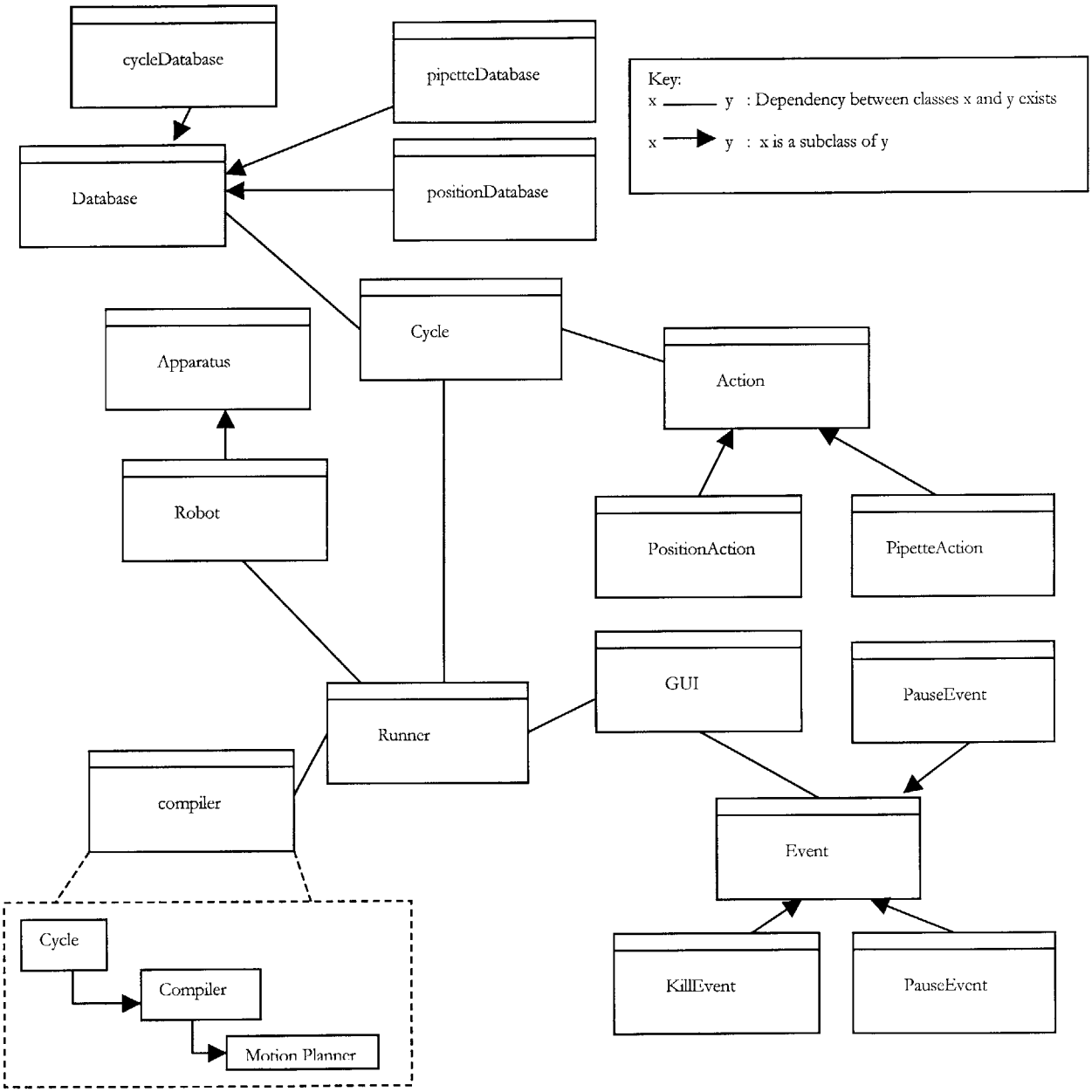


Figure 3.5. Code model.

necessary to encode this movement it was decided to separate functionality by introducing two different classes.

In the following section I formalize the requirements of the code model objects. The *representation* (rep) of an object refers to its representation in software (e.g. the representation of a List object may be an array.) The *abstraction function* describes the contents of this representation (e.g. for a List object this would correspond to a statement about how to access the elements of the List.) The *representation invariant* (rep Invariant) refers to the restrictions imposed on the representation of an object (e.g. for a List object this might translate to a restriction on the number of elements in the List.)

The rep for a cycle is:

$$rep = array[Actions]$$

The abstraction function is:

$$Af(c) = if(c.isEmpty(\ )) \Rightarrow [\ ]$$
$$else$$
$$[c.Actions.getElementAt(1), c.Actions.getElementAt(2), ..., c.Actions.getElementAt(i)]$$
$$where$$
$$i \in [1, c.Actions.getLength()]$$

The rep Invariant is:

$$repI(\ ) = c.Action.getElementAt(i).getLabel(\ ) \neq c.Action.getElementAt(j).getLabel(\ )$$
$$\forall i, j \in [1, c.Actions.getLength(\ )] : i \neq j$$

In other words, the names of the elements of a cycle cannot be the same. This is a restriction that is imposed for safety purposes because the only way of identifying a position from the GUI is by their names, which are listed in the cycle window.

Database: A database is an interface class that describes an object that stores information about a given cycle. There are three classes of database, a

PipetteActionDatabase, a PositionActionDatabase and a CycleDatabase. Since a database is a flat file implementation it was necessary to separate these objects because of the differences described above between a PositionAction and a PipetteAction. The CycleDatabase holds information about a created cycle.

The rep for a database object is:

$$rep = filename, array[Actions]$$

The abstraction function is:

$$Af(c) = [c.Actions.getElementAt(i), c.Actions.getElementAt(2), \ldots, c.Actions.getElementAt(i)]$$
$$where$$
$$i \in [1, c.Actions.getLength(\ )]$$

Also:

$$filename.getLine(1) = c.Actions.getElementAt(1).toString(\ )$$
$$filename.getLine(2) = c.Actions.getElementAt(2).toString(\ )$$
$$\ldots$$
$$filename.getLine(i) = c.Actions.getElementAt(i).toString(\ )$$
$$where$$
$$i \in [1, c.Actions.getLength(\ )]$$

The rep invariant is interesting because it contains a cross class constraint. It is:

$$repI(c) = if(c.isTypeOf(CycleDatabase))$$
$$\exists e, d : e.isTypeOf(PipetteActionDatabase), d.isTypeOf(PositionDatabase),$$
$$(d.filename = (c.filename + "\_pi")) \bullet (e.filename = (c.filename + "\_p")) \bullet$$
$$(d.Actions[i] \in c.Actions) \bullet (e.Actions[i] \in c.Actions)$$
$$\forall\ i \in [1, j.Actions.getLength(\ )] \quad \forall \quad j \in \{c, d, e\}$$

The necessity of this relationship became apparent when evaluation of the prototype was taking place. For example, a user can create any number of different cycles. Before creating a cycle they must first generate a position database (call this database PDB1) that will be used to create the cycle. If one allows the positions of another position database (PDB2) to be incorporated one has the following problems:

1. A position in PDB1 could have the same name as in PDB2 but in fact be associated with a different position. This may be due to:

    a) Small changes in hardware position between runs. A problem that can be solved by a simple calibration step after loading the databases which modifies the database entries to account for movements in hardware.

    b) Differences in naming convention between runs. Creating a naming protocol for different positions on the hardware can easily solve this problem.

2. A position in PDB1 could have a different name to one in PDB2 but correspond to the same position.

Compiler: The compiler was a new addition to the way of thinking about the problem. Because of difficulties involved with getting feedback from the robot, an option deemed feasible and effective was having a compiler that translates a cycle setup to a program that is then run by the robot. The compiler is controlled by the Runner class through events telling it what type of compilation is necessary. The compiler is in effect proxy to the robot; able to generate code on the fly which can then sent to the robot controller for execution.

The full specification of all the objects is shown in Appendix A. The interface between the objects and the outside world is documented here also.

Chapter 4

BASE-CALLING

4.1 Rational

In this chapter I discuss the design and testing of the filter stage proposed in chapter 2.

To approach the problem of designing a filter that allows Phred to generate quality scores from data preprocessed by Trout it is important to understand the differences between the preprocessing steps Trout implements and those of the Plan. Figure 4.1 shows just the pre processing steps taken by both Trout and Plan. Background subtraction, color separation, smoothing filter, mobility correction.

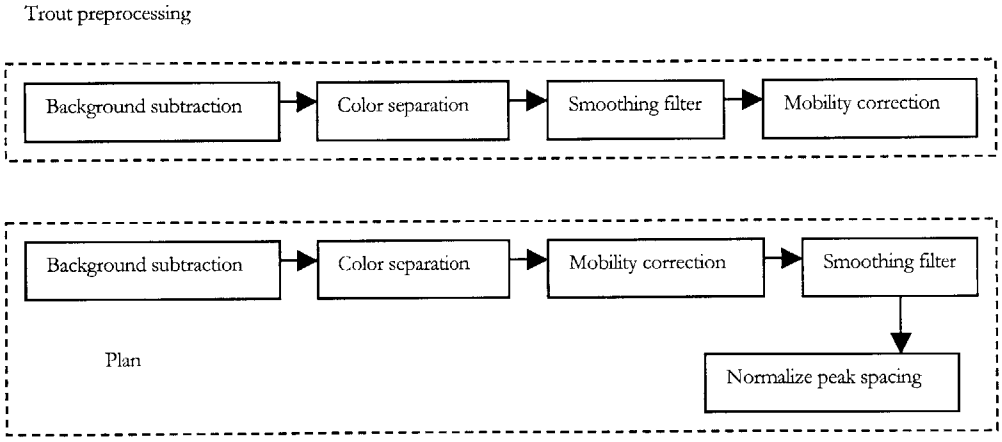


Figure 4.1. Plan and Trout processing steps.

While the general functionalities of each step carry many similarities across the two packages, the underlying algorithms are different. Plan depends heavily on

lookup files to get the color separation matrix, mobility shift correction and peak spacing factors. These lookup files are machine and chemistry specific, which makes it very difficult to adopt Plan for our equipment. Trout on the other hand estimates the color separation matrix and mobility correction factors from the data. Trout however does not have a peak spacing normalization module and the design and implementation of the filter stage depends on understanding how Plan does this function.

4.1.1 How Plan performs space normalization

Normalization of spaces in Plan is done using a predetermined lookup table that assumes certain characteristics of all data. Plan first weights the entries in the lookup table by a factor that it calculates using *average location* and *average spacing* variables found by analyzing the data as follows:

1. Find the peak spacing at a "good point" in the trace. This is accomplished by first dividing the trace into windows and taking the average spacing of the window with the lowest peak spacing standard deviation as the *average spacing.*

2. Taking the midpoint of this window and setting that to be the *average location.*

Using the weighted lookup table, Plan maps, using a linear interpolation, the data to another trace with an average spacing profile defined by the new lookup table.

The normalization of peak spaces by Plan is important for both the Phred base-calling and quality estimation modules. The Phred base-caller first finds idealized peak locations (*predicted* peaks.) To do this it uses the fact that, on average, fragments are locally evenly spaced to predict the idealized evenly spaced locations of bases along the trace. These predicted peaks are then matched with observed peaks leaving *unmatched* peaks. These unmatched observed peaks are

incorporated into the called sequence if it is obvious that they are peaks but were not predicted. However Peaks that are observed but not predicted may be discarded [3]. The quality estimation module then uses an uncalled/called ratio (uncalled bases are those that were observed and discarded), amongst other parameters, to determine the quality of a called base [6].

Normalizing peak spacing and bringing it closer to what one predicts thus improves the base-caller results, in effect reducing the uncalled/called ratio that Plan uses resulting in better quality scores.

4.2 An algorithm for peak normalization

After analysis of Plans' method of normalizing peak spaces, I decided that perhaps a more systematic normalization algorithm would be more suitable. An algorithm that was able to effectively normalize peaks using only features of the data itself and not a lookup table that is chemistry and machine specific.

The philosophy behind our approach was to smooth the average spacing of the trace data. Figure 4.2 shows the peak spacing function, $ps(x)$ of data obtained from an ABI 377 machine. Peaks were found using an algorithm that calculates the second derivative of the function and finds the zero crossings. Notice how peak spacing is initially quite variable yet small, the variation falls between bases 200-500 then rises again. This is a fairly typical profile for the peak spacing of trace data and is explained by the following [3]:

1. Irregular migration of small DNA fragments down the channel caused by the dye used to mark the fragments and unreacted dye-primer or dye-terminator molecules causing the first 50 or so peaks to be unusually noisy and unevenly spaced.

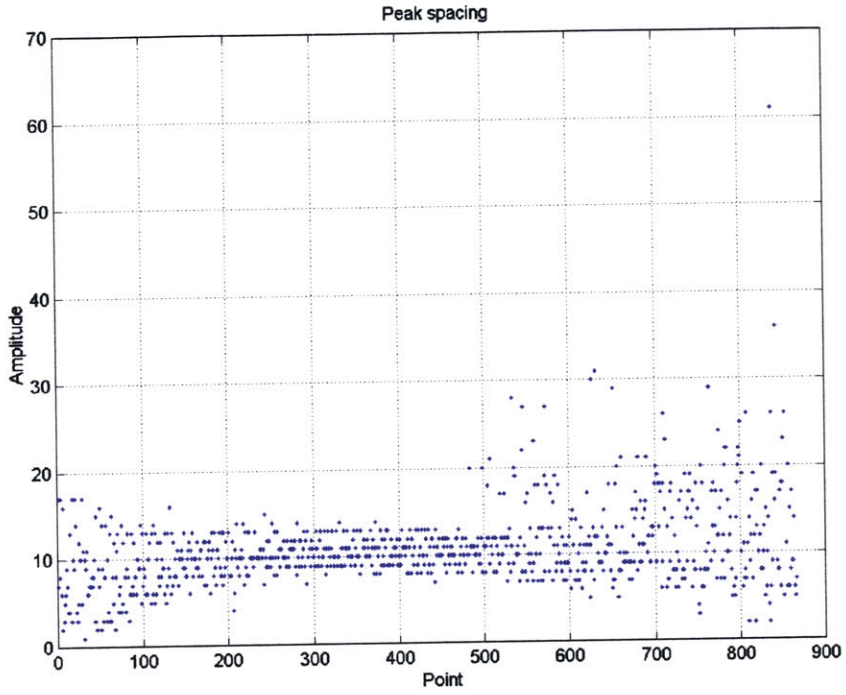


Figure 4.2. Peak spacing function $ps(x)$ from a sample sequenced with an ABI 377 machine.

2. Less well resolved peaks at the end of the trace caused by increased diffusion effects and falling relative mass between DNA fragments causing peaks to be less easily distinguished and leading to the detection of combined and broad peaks in turn leading to bigger and uneven spacing at the end of the trace.

Our initial approach to the problem of smoothing out the peak spacing was to map the old trace to one whose peaks were all spaced equally at some value k. But this proved to be inappropriate for two reasons:

1. The fairly wide variation of spacing along the trace meant that some peaks were disproportionately shifted giving rise to big distortions.

2. Quite often, peaks will overlap significantly even after mobility shift corrections. Trying to separate peaks that have such an overlap could also lead to misleading peak definitions.

I decided to break the original trace up into windows of peaks. Then map each window of data in a manner that took into account the variations in peak spacing from window to window, thus averting the distortions mentioned above.

In the following analysis I will describe the algorithm more formally. A variable with a superscript 'n' refers to the newly mapped trace and one with a superscript 'o' refers to the old trace (i.e. the one being mapped.) The variable *newavgspacing* refers to average spacing desired in the new trace. Now consider the original trace to be a tabulated function with N data points $y^o = f^o\left(x^o\right), x^o = 1,\ldots,N$. If the total number of peaks in y^o is P, we first divide y^o into windows of a peaks, $\lfloor w_0, w_1, \ldots, w_n \rfloor, n = P/a$. Each window has l_{w_k}, $l_{w_k} = 0 \; \forall \; k < 0$ data points. To perform the mapping we traverse and evaluate y^o at incremental steps $\Delta x^o_{w_k}$ (which depends on the window we are currently in.) If $w^i_k, 1 \le i \le a$ is the i th peak in window w_k, we have:

$$\Delta x^o_{w_k} = \left(median\left(\sum_{i=2}^{a} (w^i_k - w^{i-1}_k) \right) \Big/ newavgspacing \right)$$

To find y^n we assume that we have an interpolation function for any $x \in \Re$: $1 \le x \le N$ gives us a value for y^o. We call this function I and it is a function of the original trace. We have:

$$I\left(f^o\left(x_j^k\right)\right) = y_j^n, x \in \Re : 1 \le x_j \le N$$

x_j^k is the point at which we want to evaluate, by interpolation, y^o and it is located in the k th window of y^o. The value of x_j^k is:

$$x_j^k = j * \Delta x_{w_k}^o + (1 - \Delta x_{w_k}^o) \sum_{t=0}^{t=k-1} l_{w_t} \Delta x_{w_t}^o$$

The overall algorithm steps are therefore as follows:

1. Choose an average peak spacing for the new trace, *newavgspacing*.
2. Find peak locations.
3. Break up the original trace into windows of a peaks.
4. Find the median peak spacing of these windows.
5. Map, by interpolation of the old trace to find the new trace values, each window of the old trace to a trace whose average peak spacing is the ratio of the median peak spacing found in step 4 and *newavgspacing* set in step 1.

Interpolation was done using cubic spine interpolation with the natural cubic spline, which has zero second derivatives at both of the boundaries of the curve. Cubic spline interpolation was chosen because it gives an interpolation formula that is smooth in the first derivative and continuous in the second, both within an interval and at its boundaries, thus producing a smooth final trace. The effect of the mapping algorithm on the original peak locations at the window edges is shown in Figure 4.3. Figure 4.4 shows the effects on the trace.

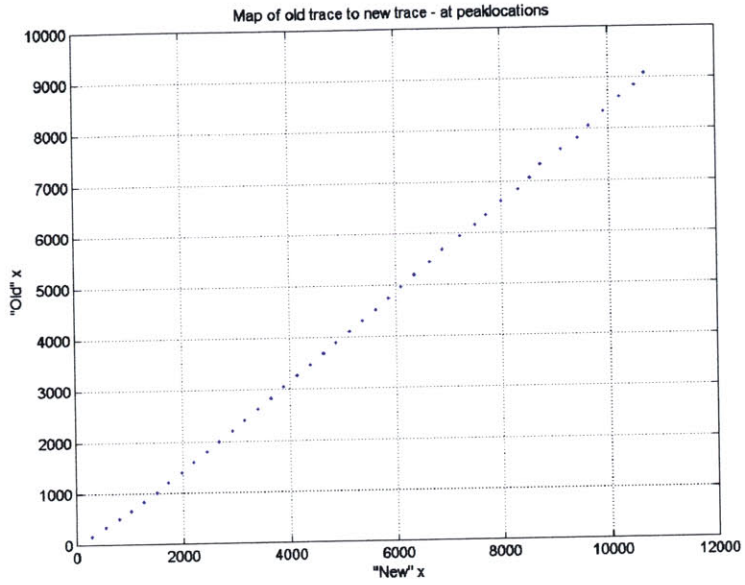


Figure 4.3. Mapping from old trace to new trace. Points shown are the peak locations at the window edges.

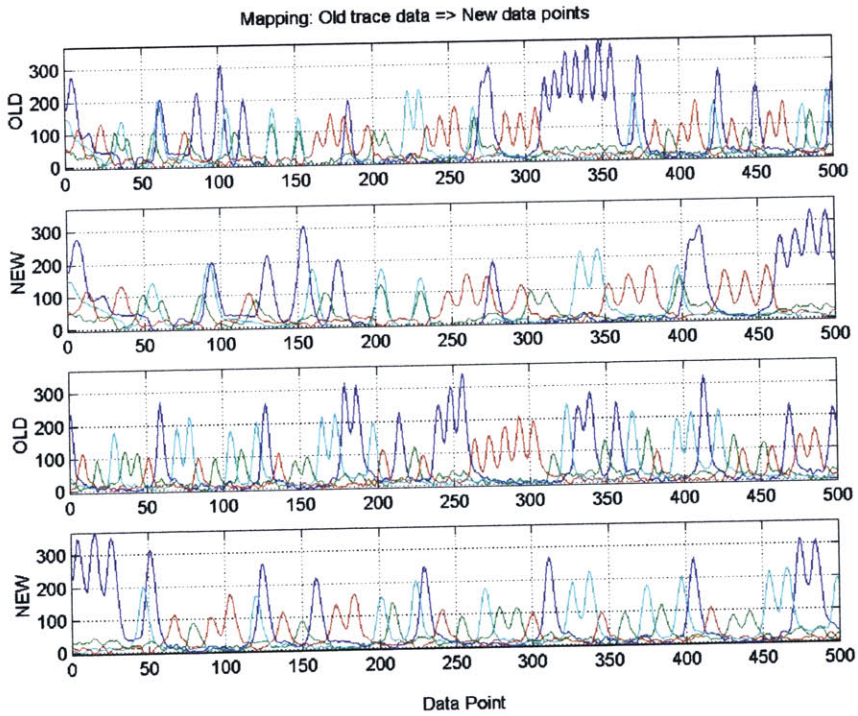


Figure 4.4. Mapping effect on the original trace. Done using window size of 20 peaks and a new average spacing of 12.

The mapping in this case has caused an expansion in the number of points in the original trace which can be observed by the wider peak spacing seen.

4.3 Results and discussion

The testing strategy was straightforward. I wanted to compare the performance of Phred when given the filtered Trout output and when given the Plan preprocessor output. I processed 11 Bluescript sample files produced by an ABI sequencer with the Plan-Phred combination, then with Trout-filter-Phred combination. Comparing the output of the Phred base-caller via these two paths involves two essential aspects:

1. Comparison of the accuracy of the final sequence. i.e. is the sequence called by Phred accurate or not when aligned with the consensus sequence.

2. Comparison of the quality scores produced by Phred. i.e. how confident am I that the called bases are correct.

If the sequence produced has great confidence levels but is not accurate then it is not of much use. Similarly if the sequence is accurate but the confidence values generated are very low, then it is again not very useful. We are thus looking for a balance between good accuracy and good confidence levels

4.4 Quality scores

As previously mentioned the quality scores generated by Phred represent how confident the base-caller is with the called bases. A plot of confidence scores generated by the Phred-Plan combination and the Trout-filter-Phred combination with position is plotted in Figure 4.5.

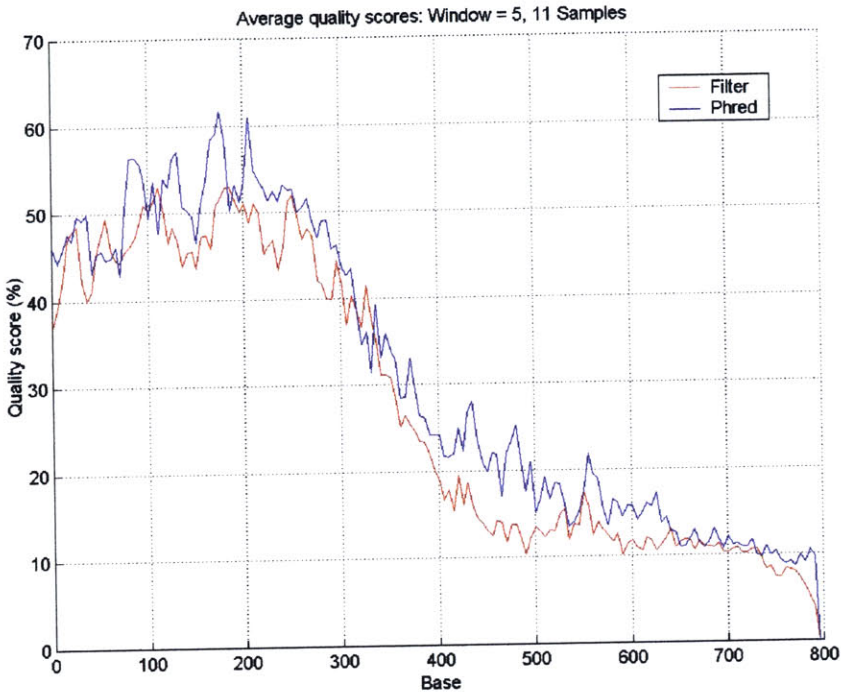


Figure 4.5. Average Trout-filter-Phred confidence scores and Plan-Phred confidence scores.

Note that the filtered trace follows the profile of the Plan-Phred combination closely. The differences can be explained by the following:

1. Decrease in the quality values late in the trace, caused by deterioration of peak resolution and called/uncalled ratio which is one of the parameters used by Phred to designate quality scores. The called/uncalled ratio is itself a function of predicted peaks and located peaks. Because peaks get wider and poorly resolved near the end of the trace, peak prediction becomes less reliable. This error more pronounced in the filtered trace because of poor spacing possibly caused (especially at the end of a trace) by inaccurate peak finding used in the algorithm.

2. In higher quality regions of the trace, the lower than expected quality values may be due to compressions in these regions and the uneven spacing they produce [6]. This may be more pronounced in the filtered trace again because the window of peaks used to determine the new spacing may be too small in a region where compressions may lead us to underestimate the spacing.

4.5 Accuracy

The accuracy of the called sequence is determined by an alignment process. The called sequence is aligned with a consensus sequence that represents the completed know sequence. The alignment algorithm tries to match the called sequence with the consensus as optimally as possible hence minimizing the following types errors:

1) Mismatch: Occurs when a base is aligned with another base that is not the same e.g. A with C.

2) Overcall: Occurs when a base is called at a position but in reality there is no base there e.g. if the consensus sequence is AACG and the called sequence is AACTG, then an alignment where a T is forced into the sequence would be an overcall.

3) Undercall: Occurs when a base is not called, but should actually be there e.g. if the consensus sequence is AACG and the called sequence is AAG, then there is one undercall.

The average total percentage error of the samples and the average of the component errors is shown in Figure 4.6. The figure shows that the filtered data outperforms the Plan-Phred combination up to 600 bases where it has 98% accuracy. If we look at the contribution of the individual error types mentioned

above we see that the filter does not perform as well as Phred through out the trace but staying less than 0.9 errors more than Phred through out the entire trace. On further examination of the processed traces, the errors in overcall can be explained by the following:

1. A high number of peaks appearing like poorly resolved peaks.

2. Poor peak separation especially towards the end of traces where errors were caused because overlapping peaks would both be called.

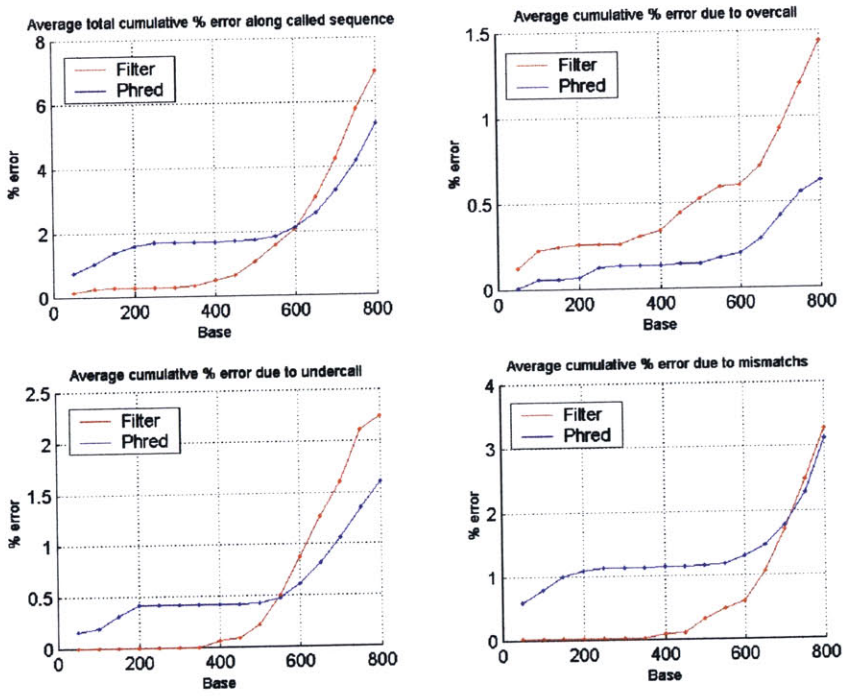


Figure 4.6. Called sequence accuracy.

The filtered trace performed more accurately with respect the number of undercalls up until 550 bases. The rise above the Plan-Phred processed samples error was observed to be predominantly due to:

1. Very poor peak resolution at the end of traces.

2. Poor peak normalization which may have happened even with the window break up of the trace when performing normalization using our algorithm. This is because the window width was chosen based on a count of peaks. However, after 500 bases traces can have very varied peak spacing especially if the peaks are found using a simple algorithm that looks for the zero intercept of the second derivative because many peaks are poorly resolved and tend to appear wide and almost flat creating the illusion of many narrow peaks.

With regards the number of mismatches found, the filtered trace performed more accurately up until ~700 bases.

Having validated the algorithm, I tested it on three sample files that were produced by our machine. The average quality scores for the samples is shown in Figure 4.7. The plot shows a good quality score profile and the high scores between 200 and 550 bases correspond to zero errors in the called bases over this region. On examination of the traces, the dips in quality at bases 110, 290, 480, and 520 can be explained by an inherent weakness in Phred's quality scores module when processing traces that are not produced by ABI machines. This conclusion can be made because the traces at these points were not very noisy.

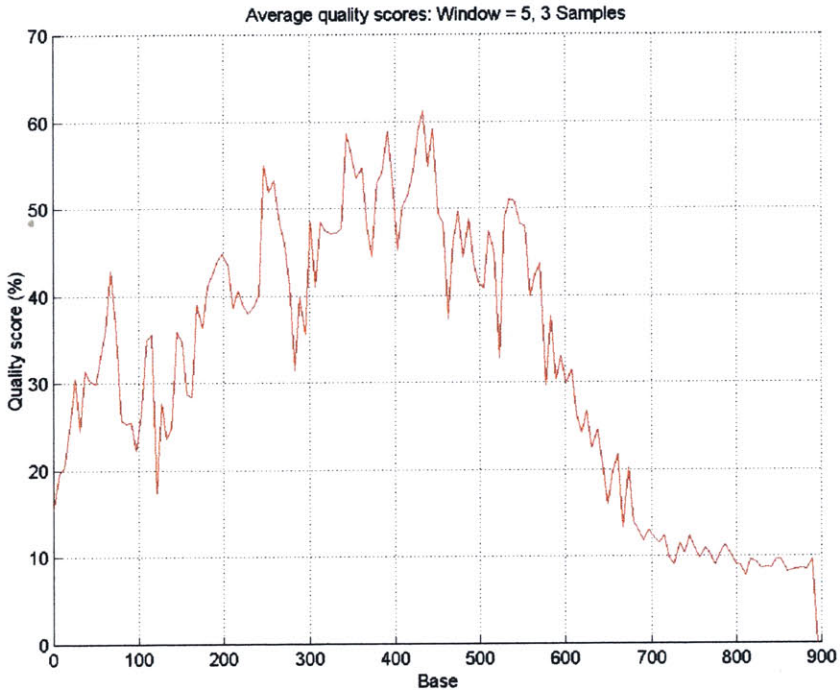


Figure 4.7. Average Trout-filter-Phred confidence scores.

The accuracy traces for these three samples were very good and gave more than 98% accuracy up to 800 bases.

Chapter 5

CONCLUSION

In this thesis I have been able to successfully develop a functional robot control application prototype that is currently in use with the sequencing hardware my group is developing. I found the prototyping approach to be very effective for our purposes because of its short development life. The evolutionary approach to prototyping also meant that I did not have to redesign and rewrite the application each time there were changes to the specifications by the user, changes that occurred very frequently.

In addition to prototyping the application, I have developed a maintainable design for the robot control system that is intended for future implementation.

The second part of this thesis described the implementation of a filter module that allows us to use Trout for preprocessing and Phred for base-calling. With this new filter, I was able to outperform, with respect to accuracy, Phred up until 600bp where I observed 98% accuracy. I found that the degradation in filter performance (compared with Phred) after the 600 bases mark was mostly due to disparities between the number of overcalls and undercalls in the Phred results and the filter results. On analysis, this disparity was attributed to:

1. Poor peak resolution at the end of the data; the source of most overcalls.

2. Poor peak normalization at the end of the data. This was because the algorithm developed relies on the fact that peak spacing within any extended region of a trace is going to be somewhat similar. However, the end of the trace can display very irregular peak spacing. Modifying the algorithm so

that the length of the window of peaks is variable along the trace can reduce the problem.

I was able to realize quality scores with an average profile that tightly shadowed that of the Phred scores all the way through most of the trace, giving credibility to the accuracy I obtained. The differences were most likely due to:

1. Fall in peak resolution at the end of that trace which caused a fall in the called/uncalled peak ratio, one of the parameters Phred uses to generate quality scores.

2. Natural compressions and uneven spacing in peaks again producing problems because the algorithm relies on the spacing in any extended region of the trace to be on average even.

This validation testing was performed using ABI 377 files. Having validated the algorithm I then tested it on three files produced by our machine. The results showed good quality scores with irregularities that I attribute to the fact that Phred was tuned to work with traces produced by ABI machines. In order to obtain better confidence scores, future work will need to train Phred by testing with more data sets from our machine.

Future work will also involve tweaking the algorithm to improved performance in accuracy and confidence for data produced by our machine, in addition to the implementation of a separate quality scores module.

Appendix A

Robot Controller Function Specification

```
*********ACTION*******************************

Class Interface Action Extends CObject
{
Overview:  An action is an abstract function
that is used to represent a Cycle makeup.  An
action has a name in: this.label.

Action()
{
}

abstract CString toString()
{
}

CString getName()
{
//effects: returns this.label
}

CString setName(nname)
{
//modifies: this.label
//effects: set this.label = nname.
}

} //end Interface Action


PositionAction


Class PositionAction
{
Overview:  A position action is an action
associated with the X,Y and/or Z axis.  The
action is made up of a motion along the axes
to (this.x, this.y, this.z.)

//constructor
PositionAction(nname)
{
//modifies: this.x , this.y, this.z are all
set to zero and this.label=name.
}

PositionAction(nx,ny,nz, nname)
{
//modifies: this.x = nx, this.y = ny, this.z =
nz, this.label = nname
}

//access functions

CString getXPosition()
{
//effects: returns this.x
}

CString getYPosition()
{
//effects: returns this.y
}

CString getZPosition()
{
//effects: returns this.z
}

void setXPosition(nx)
{
//modifies: sets this.x = nx
}

void setYPosition(ny)
{
//modifies: sets this.y = ny
}


void setZPosition(nx)
{
//modifies: sets this.z = nz
}

CString toString()
{
//effects: returns a string representation of
this object in the following format:

"this.label this.x,this.y,this.z"

note the string does not contain a carriage
return.

}


} //end Class PositionAction


**********************PIPETTEACTION**********


Class PipetteAction
{
Overview:  A pipette action is an action
associated with the theta axis of the robot
(the pipettor.)  A pipette can either dispense
or inject depending on this.position  In
addition it can cycle through dispensing and
injecting.  The number of times it does this
is determined by this.numCycles.  A
PipetteAction has a name in: this.label.


//constructor
PositionAction(nname)
{
//modifies: this.x , this.y, this.z are all
set to zero and this.label=nname.
}

PositionAction(numberOfCycle,pos,nname)
{
//modifies: this.numCycles, this.position
//effects: this.numCycles = numberOfCycles,
this.position = pos,this.label = nname
}

//access functions
void setNumberOfCycles(nc)
{
//modifies: this.numCycles
//effects: this.numCycles = nc
}

void setPosition(np)
{
//modifies: this.position
//effects: this.position = np
}

CString getPosition()
{
//effects: returns this.position
}

CString getNumCycles()
{
//effects: returns the number of cycles
assoicated with this pippette action
}

CString toString()
{
//effects: returns a string representation of
this object in the following format:

"this.label this.numCycles, this.pos"

note the string does not contain a carriage
return.
}
```

```
} //end Class PipetteAction


********************DATABASE*****************

Class Interface Database
{
Overview: A Database object is an abstraction
that implements a storage medium for data.
Specifically, data relating to PipetteActions,
PositionActions and RunCycles.  A database is
stored in a this.filename.  A database has a
name this.name.  The elements of a database
are stored in this.actionArray.

//constructor
Database()
{
}

Abstract loadDatabase();
Abstract createDatabase(filename);


void insert(action, i)
{
//requires: i >= 0 , i < actionArray.length()
//effects: inserts "action" into
this.actionArray at position i.
//modifies: this.actionArray
}

void append(action)
{
//effects: appends "action" onto the end of
this.actionArray
//modifies: this.actionArray
}

void remove(i)
{
//requires: i >= 0, i < actionArray.length()
//effects: removes an action from actionArray
located at i
}

CObject getAt(i)
{
//requires: i >=0, i < actionArray.length()
//effects: returns the action
this.actionArray[i]
}


}// end Interface Database


****************CYCLEDATABASE***************

Class CycleDatabase
{
//overview: A CycleDatabase is a database that
stores the actions of a cycle.
//          Each cycle database has an
associated PipetteActionDatabase and
PositionActionDatabase this.pipetteDB and
this.positionDB respectively.

//constructor
CycleDatabase()
{
}

void loadDatabase(fn)
{
//effects: returns an array of actions
associated with the actions stored in the file
fn into this.actionArray.  sets this.filename
= fn
//modifies: this.actionArray, this.filename
}

void storeDatabase(fn)
{
//effects: sets this.filename = fn, stores the
elements of this.actionArray in fn in the
following way:

if(this.actionArray[i] isInstanceOf
PipetteAction) for i st. 0 <= i <=
length(actionArray)
then
"p actionArray[i].toString() <return>"
else
"l actionArray[i].toString() <return>"
If "fn" exists it is overwritten.  If "fn"
does not exist it is created.
}

}//end Class CycleDatabase

****************POSITIONACTIONDATABASE*******

Class PositionActionDatabase
{
//overview: A PositionActionDatabase is a
database that stores positions.

//constructor
PositionActionDatabase()
{

}

void loadDatabase(fn)
{
//effects: returns an array of actions
associated with the actions stored in the file
fn into this.actionArray.  sets this.filename
= fn
//modifies: this.actionArray, this.filename
}

void storeDatabase(fn)
{
//effects: sets this.filename = fn, stores the
elements of this.actionArray in fn in the
following way:

"actionArray[i].toString() <return>"

If "fn" exists it is overwritten.  If "fn"
does not exist it is created.
}

}//end Class PositionActionDatabase

****************PIPETTEACTIONDATABASE********

Class PipetteActionDatabase
{
//overview: A PipetteActionDatabase is a
database that stores the actions of a cycle.

//constructor
PipetteActionDatabase()
{
}

void loadDatabase(fn)
{
//effects: returns an array of actions
associated with the actions stored in the file
fn into this.actionArray.  sets this.filename
= fn
//modifies: this.actionArray, this.filename
}

void storeDatabase(fn)
{
//effects: sets this.filename = fn, stores the
elements of this.actionArray in fn in the
following way:

"actionArray[i].toString() <return>"

If "fn" exists it is overwritten.  If "fn"
does not exist it is created.
}
```

```
}//end Class PipetteActionDatabase


**********COMPILER**********

Class Compiler
{
//overview: a compiler is an object that given
a cycle of actions generates Compumotor
commands in the form of a file and writes the
output to a file.

//constructor
Compiler()
{
}

Compile(e)
{
}

CString createHeader()
{
}

CString createCore()
{
}

CString generate()
{
}

} //end Class Compiler



**********EVENT**********

Class Interface Event
{
//overiew: An event object is an interface
representing the events that a Runner object
receives from the GUI.  An event has name
this.name.

Event(n)
{
//effects: creates a new event object with
name = n.
}

CString GetName()
{
//effects: returns this.name
}

CString SetName(nn)
{
//effects: this.name = nn
  }

} // end Interface Event

**********STOPEVENT**********

Class StopEvent
{
//overview: A StopEvent object is an Event
with name = "Stop".  This event is generated
when "Stop" is pressed on the GUI.
}

**********PAUSEEVENT**********

Class PauseEvent
{
//overview: A PauseEvent object is an Event
with name = "Pause".  This event is generated
when one "Pauses" the cycle on the GUI.
}

**********RESUMEEVENT**********

Class ResumeEvent
{
//overview: A ResumeEvent object is an Event
with name = "Resume".  This event is generated
by the GUI when one "Resumes Cycle" on the
GUI.
}

**********KILLEVENT**********

Class KillEvent
{
//overview: A KillEvent object is an Event
with name = "Kill".  A "Kill" event is sent
from the GUI on the pressing of a kill button
on the GUI.
}


**********SINGLEGOEVENT**********

Class SingleGoEvent
{
//overview: A SingleGoEvent object is an Event
with name = "Single".  A SingleGoEvent is sent
by the GUI when a movement to a single
position is required which is indicated by the
depression of the "Go" Button on the GUI.
}

**********RUNNER**********

Class Runner
{
//overview:  A Runner object handles events
from the GUI and calls the compiler with the
specific reqs then takes the output of the
compiler and exceutes it. A Runner is
essentially the main thread of the softear
package.  It has associated with it a robot
object, this.robot a compiler, this.compiler
and a GUI, this.gui. A runner also has cycle,
pipetteaction and positionaction databases
associated with it.

//constructor
Runner()
{
}

void HandleEvent(e)
{
//effects:
if(event.name == "Kill")
call this.Kill()
else if (event.name == "Pause")
call this.Pause()
else if (event.name == "Resume")
call this.Resume()
else if (event.name == "Start")
call this.Start(e)
else if (event.name = "Single")
call this.Single(e)
}

void Kill()
{
//effects: kills robot motion
}

void Pause()
{
//effects: pauses robot motion
}

void Resume()
{
//effects: Resumes robot motion
}


void Start()
{
//effects: If this is the first time being
called, create the cycle program with special
first-time headers and footers
}
```

```
                                                    }
void Single(e)
{                                                   }
//effects: Causes the action associated with
the single event "e" to be exceuted
```

References

[1] Human Genome Project Information: *http://www.ornl.gov/hgmis/project/about.html*

[2] Schmalzing, D., Tsao, N., Koutny, L., Chisolm D., Srivastava A., Adourian A., Linton L., McEwan P., Matsudaira P., Ehrlich D. *Toward Real-World Sequencing by Microdevice Electrophoresis.* Genome Research, 9:853-858.

[3] Ewing, E. , Green, Phil. *Base-Calling of Automated Sequencer Traces Using Phred, I Accuracy Assessment* Genome Research,8:196-194, 1998.

[4] Sanger, F., Nicklen S., and Coulson A.R. 1977. *DNA sequencing with chain-terminating inhibitors.* Proc. Natl. Acad. Sci. 74: 5463-5467.

[5] Schmalzing D., Koutny L., Adourian A., Linton L., Belgrader P., Matsudaira P. *DNA typing in thirty seconds with a microfabricated device.* Proc. Natl. Acad. Sci. USA Vol. 94, pp. 10273-10278, September 1997.

[6] Ewing, E. , Green, Phil. Doe *Base-Calling of Automated Sequencer Traces Using Phred, I Error Probabilities* Genome Research,8:196-194, 1998.

[7] Koutny L., Schmalzing D., Salas-Solano O., El-Difrawy S., Adourian A., Buonocore S., Addey K., McEwan P., Matsudaira P., Ehrlich D. *Eight Hundred-Base Sequencing in a Microfabricated Electrphoretic Device.* Anal. Chem., 2000, 72, 3388-3391.

[8] Hekmatpour S., Ince D. Software Prototyping, Formal Methods and VDM. Addison-Wesley, 1988.