The Subsumption Strategy Development of a Music Modelling System

Joanna Joy Bryson

Department of Artificial Intelligence
University of Edinburgh

September 14, 1992

Abstract

This dissertation describes an attempt to apply a behaviour-based robotics design methodology, subsumption architecture, to a software project which emulates a human competence. The project produces a chord structure for a song when presented with its melody as audio input, with the goal of providing real time accompaniment without the benefit of a score. The implications of behaviour-based distributed intelligence paradigms for both the engineering and the explanatory aspects of artificial intelligence are discussed.

Acknowledgements

My deepest gratitude goes to my supervisors, Alan Smaill and Geraint Wiggins not only for their guidance and assistance, but for their enthusiasm, support, patience, and tact.

Thanks also to the following individuals for taking the time to share their expertise with me: Michael Aieta, (AI systems for musical performance); Peter Cooke, (Music theory); David Corne, (Connectionism applied to music); Beau Lotto, (Neurophysiology); Maja Mataric, (Representation in subsumption architecture); and especially David Willshaw, my part-supervisor in neural networks.

Special thanks to all the musicians who contributed to this project: Helen Lowe, (penny whistle); Colin Macnee, (vocals); Manuel Trucco, (Scottish small pipes); and particularly Sheila Tuli for being my human-accompanist research subject.

Kudos for technical assistance to: Alan Black, Andrew Fitzgibbon, and Kostis Kaggelides, (C++); Ian Frank (latex); Rocio Aguilar-Chongtay, and Edith Rodriguez-Camarena (recording); and Ilona Bellos, and Bernadette Noon, my xfig and bibtex mercenaries;

I am grateful to all the people who have expressed an interest in (or just tolerated the noise of) my project. In particular, I would like to credit the following individuals for making substantial contributions through their suggestions and discussions: Gill Kendon, Philip Kime, Ian Porter, and especially William Smart who also assisted with the dissertation.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Purpose

The purpose of this project is to examine the application of a novel architecture to the problem of emulating a human competence. Specifically, my aim is to design a system for deriving the chord structure for a song from its melody. This requires creating a new model from a set of data, which involves adding to, removing from, and retaining intact various features of that data. In other words, it is a form of *interpretation*, one of fundamental tasks of intelligence.

The approach used for developing this program is a system of distributed intelligence called *subsumption architecture*. This is a methodology developed by Rodney Brooks at MIT for designing and controlling robots[Brooks 91b]. The strategy is based on a principle of developing individual *competences* which run in parallel with each other, each acting on its own inputs and creating its own outputs without regard for the existence of the other modules.

The basic principle of a distributed intelligence is this: that a complex thing is built out of a number of relatively simple parts. The complexity of the overall is a consequence of how we as observers perceive the interactions of the simple parts; the complexity is said to be *emergent* from those parts. For an example of this, think of four people singing “Frere Jaques” as a round. The melody sung by each individual is simple, but the song sounds much more complex and interesting when heard overall as a round. Similarly, an intellectual task which seems complex may be actually be executed by smaller parts capable only of much simpler work but whose efforts are coordinated into the appearance of an intricate whole.

In the course of this project, a system composed of basic musical competences has been constructed, using a methodology close to the principles of subsumption architecture. Though the system's output is unlikely to be confused with an experienced musician's, it does show some of the same competences as a naive human musician. The development process was very successful, and it is my belief that it could be continued to bring this project much closer to a reasonably competent accompanist without using traditional symbolic processing.

Besides producing an interesting piece of artificial intelligence, this project is significant in two other respects. First, subsumption architecture does not seem to have been applied to a software project before. A new technique for software design, particularly one that could be applied to parallel, real time tasks, would be useful. Second, some researchers support distributed processing as a model of biological intelligence. But many of those who do believe it can only be applied to low-level behaviours, such as reflexes. Music is generally considered to involve higher level cognitive functioning, so a distributed program that successfully modeled a musical competence would be very interesting theoretically.

1.2 Thesis Layout

This chapter is the introduction. It starts with a section describing the projects purposes, followed by the thesis layout, a section describing the origins of the projects conception, and finally a glossary of terms used in this project.

Chapter 2 describes the theory behind this work, including an in depth description of distributed intelligence and of subsumption architecture. It also describes related work.

Chapter 3 describes the general structure of the system of programs implemented within the course of this project. This includes a list of all the behavioural units, or *modules* used, and how they interact.

The next several chapters treat of these modules individually, as each is essentially a mini-project in itself.

Chapter 9 evaluates the overall success of this project. This is not only a critique of my implementation, but also contains discussions of the subsumption strategy as a development tool, and of distributed intelligence as a paradigm for human cognition.

Chapter 10 describes further work that could be done on this project.

Finally, Chapter 11 concludes the paper with a summary.

1.3 Origins and Motivations

The following is a chronology of the development of my concept for this project. It attempts to show not only the history, but also some of the challenges addressed.

1.3.1 Fundamental Issues

My interest in this project stems originally from two concerns I had before beginning this course. The first was an issue of system design. I had been programming progressively larger systems on progressively more distributed hardware platforms. It occurred to me that much of our software was more informed than it needed to be. This lead to unnecessarily large and complex programs, which costs in implementation time, maintenance, and in communication time with shared data. On the other hand, programs that are too uninformed allow incoherent inputs which will produce useless outputs and possibly break the system. The problem is one of design: *how do you know how much knowledge is needed where?*

The second was an issue of human cognition. I was aware from my undergraduate course work that perception and memory are largely constructed on demand of access, and can differ greatly from the actual real-world events we think they are representing.[1] But I noticed that this was also true of thought processes, such as planning and decision making. Of course, Sigmund Freud proposed a competitive (multi-agent) theory of consciousness to explain why we might be unaware of our own contradictory behaviours. But his theories concerned deep psycho-social needs and desires, while these same kinds of inconsistencies between described methodologies and actual actions can be seen in such banal tasks as cooking, driving, or performing office routines. Often a person asked to explain such tasks will not remember the actual sequence of events, or may even "remember" them inaccurately. These acts are usually considered "voluntary", which is by definition "under conscious control". I became curious about the actual role of consciousness in this kind of action. This is a question of cognition: *how do we plan and control our daily behaviour, particularly those aspects we don't really think about?*

1.3.2 A Design Strategy

As part of my course work here at Edinburgh I took a robotics class called "Intelligent Sensing and Control." [Hallam 91b] This course covered different control strategies for the goal oriented behaviour of robots, one of which was a new strategy being developed at MIT called "subsumption architecture." [Brooks 91b] Where traditional robotic control systems were based on the idea of a central planner that directed activity, this system is based on individual behaviours which determine for themselves when they should act. These behaviours, or competences, are developed by the designers one at a time, starting with the most basic. As higher level behaviours are added, they subsume the primary control of the robot. Since this behaviour-based methodology has been developed, there have been a number of successful robots built using it, including here at Edinburgh. These robots tend to be developed more quickly and to behave more dependably than robots controlled by traditional methods. [Maes 90][Hallam 91b].

This new paradigm, though designed for robotics, seemed quite applicable to other areas. In fact, the lecturer, John Hallam, made a point of the way human

[1] See [Brooks 91a] for a brief overview of recent work in this area, especially. [Dennett & Kinsbourne 90]

actions seem to be encoded into functions higher than what we consider reflexes, but lower than conscious control. For example, when you go to pick up an object, you don't "think" about the exact position your hand should be in, though you may think of where your hand should be on the object. The idea is that grasping any normal object is identical as far as your higher control structures are concerned, while somewhere there is a competence which actually takes care of moving the hand into the proper shape.

It occurred to me that this strategy might be applied to answering both the issues raised in the previous section. But this raises several questions:

- Could this strategy be applied to general software design, particularly of large systems?
- How far can this strategy go in terms of cognitive capacities? How abstract of problems can be approached?
- How easily can an engineer decompose a problem into this kind of a solution?

I began thinking about exploring these issues in my M.Sc. project by attempting to model an interesting human competence using the subsumption architecture technique.

1.3.3 Music as a Domain

The next step was choosing a competence. On the basis of other course work (particularly Knowledge Representation and Inference) interpretation, or symbolic abstraction, seemed the most interesting problem. This can be seen particularly in the area of natural language. Many of the problems in natural language research seem to parallel the problems of research in robotics. Both areas seem tractable, but in fact tend to get bogged down in details on the low levels. In both the solutions that are derived tend to be too fragile to work in real-world situations. And in both it seems quite unlikely that people can be going through the complexity of planning each individual act (such as individual word choice and intonation) but rather that they concentrate on higher level goals.

However, building even a subset of language competence from basic principles to a sufficient level that it could be tested seemed beyond the scope of a five month project to me. But when accompanying some folk musicians on the guitar one day I became aware of the analysis process I was trying to do, and realised this might be the competence I was looking for.

Marvin Minsky describes music as "our only chance to make comparative studies of linguistic cognition."

> Is there another activity where you have temporal patterns that satisfy overlapping constraints in a symbolic way? [Roads 80]

Rodney Brooks attributes much of the trouble roboticists have gotten into in the past to their tendency for running their robots in oversimplified environments[Brooks 91b]. He explains that this leads to misjudging which parts of a problem are difficult, or even important. I strongly suspect the same could be said about natural language in artificial intelligence. It tends to concentrate on written language, which seems to me both simpler with its well-formed, completed sentences and words and lack of inflection, and more complex with its more ritualised and deliberate structures. Music of course has also been transcribed into written formats, and pieces of music are developed in written format. Many Artificial Intelligence projects have been performed on these. I use as my input the actual sounds produced live by musicians playing traditional instruments, and performing music which was developed on instruments. Consequently, my project could be seen as emulating a folk musician.

1.4 Definitions of Terms

Following is an informal glossary of terms used throughout this dissertation. The first part refers to terms used in describing the main program implemented in this project. Some of these terms are my own convention, while others are derived from Brooks. For a more thorough discussion of the concepts behind the latter, see the next chapter, Theoretical Background. The second part explains the musical terminology as it is used in this thesis. For a better grounding in these concepts I recommend the paper "The Perception of Music" by Longuet-Higgins [Longuet-Higgins 87].

1.4.1 ...with Respect to the Program

This program is an example of **distributed intelligence**, which means that the intellectual activity takes place in multiple separate units which are *not* coordinated by any central control. In particular, it is an example of a **behaviour-based** system, which means that these units each represent a behaviour of some type, rather than a unit of information processing, as in functional systems; or a generic, undifferentiated processor, such as in neural networks.

These behavioural units are called either **competences** or **modules**, these terms coming from the works of MIT and Edinburgh respectively. To a slight extent I am more inclined to use "competence" when discussing what the behaviour does, and "module" to describe what does the behaviour, but I admit to occasionally using the terms interchangeably. Brooks groups competences into **levels** of competency. This is an external, intellectual distinction, used for talking about related behaviours as a unit, but they do not exist as physically distinct units in the implementation. This is in contrast to Marvin Minsky's **agents**, also units of behaviour, but which can be recursively, hierarchically *composed* of more simple agents. Competences are defined completely independently of each other.

Each of these individual behaviours is represented on three different levels within this project. The core, theoretical concept of the behaviour is a competence or module, as described above. Each module is associated with a stand-alone program which implements it and any lower level modules it depends on, and shares the same name. I refer to these executable programs as **robots** to emphasise that these are mechanisms for interfacing with the "real world" — for generating output so that users can evaluate the modules. The output of these robots is *not* equivalent to the output of the modules: it is often simplified or processed for ease of evaluation.

The third representation of a behaviour is on the code level. This program is implemented in an **object-oriented** language, **C++**[Stroustrup 86]. Such languages are designed to facilitate the use of data structures as the central conceptual units of the program. Thus functions associated with a data structure are defined as part of it. These super-structures are referred to as **objects**, or, in the case of **C++** as **classes**. Each module in the conceptual structure of the program is associated with an object class in the code, which is often itself composed of other object classes. This dissertation seldom goes to a significant level of detail about the code, but I do occasionally use these terms.

1.4.2 ...with Respect to Music

Neither this program nor this dissertation refer to anything very technical or sophisticated with respect to music theory. Most of the knowledge required is an understanding of the terms I use to describe the units into which my program decomposes music.

Music can be described using four parameters: **pitch**, the perceived highness or lowness of the sound, **volume** or **gain**, the loudness or softness of the sound, **threshold**, the time when the sound starts, and **duration**, its extent with respect to time. The volume parameter is largely ignored in this dissertation. A **note** is a discrete unit of music, and for the purpose of this program it can be defined as having a discrete pitch and duration, and having started at a specific point in time.

Sound is actually vibration in a medium, and the rate of these vibrations is called their **frequency**. Pitch is dependent on frequency. However, most events which cause sustained vibrations of a consistent frequency also result in vibrations at related frequencies, such as twice the frequency, three times the frequency, and so on. (See figure 4–1 for a logarithmically scaled graph of this effect.) These related frequency vibrations are referred to as **harmonics**, while the first frequency is called the **root**, **principle**, or **fundamental**. Notes are identified in terms of their principle harmonic, but humans can identify them even if this root component is missing if the rest of the harmonics are present.

A note at the frequency of the first harmonic of another note, that is, twice its frequency, is considered to be one **octave** above it. For many musical purposes, notes that share this relationship are considered to be interchangeable. They are said to be of the same **pitch class**. The number of different pitch classes *within* an octave is discrete, and defined by convention. Actual frequencies for a note may range around the technical frequency of the class, but if they are too far away they are considered "out of tune" or "off key".

Although the program is built for flexibility with respect to number of pitch classes, it was run on the Western standard of 12 pitch classes, and its interfaces were built to use the standard labels of A to G♯. However, these are internally represented numerically, and the interfaces know nothing of key or harmonic convention. This is significant because the second pitch class, for example, may either be referred to as A♯ or B♭, there being an actual slight difference in pitch in some tonal systems because the harmonic series that construct these two terms are different. (see [Longuet-Higgins 87]) However, within the context of this program, it will always be referred to as A♯[2].

The most important unit of time within a song is the **beat**. The beat is the regularly occurring point of emphasis in a song, where listeners are most likely to clap, or dancers are most likely to step. A beat is composed of underlying **pulses**. Pulses can best be understood as the smaller units of time at which faster notes may change. Pulses are of regular length, and in Western music beats tend to be composed of a uniform number of pulses. There are also higher levels of time classification, for example beats are clustered into **measures** or **bars**, and measures into **phrases**.

The primary goal of this program is to analyse a "melody". In this project I took this to imply that only a single note is present at any given point in time, in which case the input is called **monophonic**. If more than one note is present at one time, the music is said to be **polyphonic**.

A **chord** is a group of notes. In general, the elements of a chord are played simultaneously. There are conventional chords where the component notes are chosen on the basis of their harmonic relations. Such a standard is called a **chord type**, for example *major* or *minor*. A complete description of a chord requires naming the pitch class of the root frequency for the series of harmonics used in the harmonic relation, for example A major or C♯ minor.

[2] because "#" is available in ASCII.

Chapter 2

Theoretical Background

This chapter explains the theoretical background for this project, and surveys some of the previous work done along similar lines. It begins with a discussion of recent theories of distributed processing with respect to explaining human cognition. It follows with descriptions of two methodologies which use distributed intelligence, neural nets and subsumption architecture. These approaches have been successfully used for creating intelligent behaviour. Neural Networks are the core of several of the behavioural modules which compose this project. Subsumption Architecture is the overall approach used and analysed by this project, and as such is the central theoretical issue of this paper. This chapter's final sections discusses related research.

2.1 Distributed Intelligence

Since its inception earlier this century, Artificial Intelligence has mostly followed the same logical/sequential approach to problem solving used in Computer Science.[1] Although significant and interesting progress has been made in this way, for sev-

[1] For a thorough review of the history of AI paradigms in relation to Psychology and Computer Science, see [Brooks 91a].

eral reasons some researchers have begun to doubt this as a model for most of human cognition:[2]

- Sequential processing often takes hundreds or even thousands of steps for the kinds of things people do very quickly, such as recalling a person's identity from their face. Because the chemical processes we use for thinking are very slow, humans couldn't possibly be using more than about 100 sequential steps for these kinds of tasks.

- Symbolic systems tend to be very fragile — if anything happens to break the sequence of steps, the results tend to be completely useless. Animal intelligence, by contrast, seems to be extremely robust. A great deal of a brain can be destroyed and an animal still function fairly well. Further, the functioning is more likely to gracefully degrade than become totally meaningless.

- People tend to be fairly bad at tasks that necessarily require explicit symbolic processing. It takes us much longer to work through a math problem that requires relatively few steps than it does to do something as highly complex as facial recognition. Also, we are much more likely to make mathematical mistakes.

- It is very hard to make a symbolic/sequential system learn or modify itself. This is probably tied to the fragility issue mentioned above: learning involves making mistakes, and mistakes in sequential code are often disastrous.

- Symbolic systems cope best with absolute, quantified data, but most real world data is uncertain, relative, and continuous. A great deal of effort has been spent in the field of Knowledge Representation on these issues, but the solutions tend to be complex and difficult to generalise; essentially, they are unnatural.

[2] This summary of the difficulties with symbolic processing is taken from a lecture by Nick Chater [Chater 92].

There are two main alternatives to the symbolic/sequential approach. One is Parallel Distributed Processing, or Neural Nets. The other behaviour-based, multi-agent systems, such as Subsumption Architecture.

2.1.1 Neural Nets

The neural network paradigm was originally inspired by research into animal brains. The brain is composed of millions of neurons — tiny single cells. Each neuron receives signals from many others, and under the right set of circumstances fires a simple signal itself. The messages a neuron receives may be either excitatory or inhibitory, and are essentially just summed until a threshold is reached. Neural Nets imitate this using a single simple computational unit, the perceptron. When the sum of a perceptron's inputs reaches its threshold, it too fires a signal.

Neural networks and perceptron theory were heavily researched in the sixties, and enjoyed a rebirth of interest in the eighties when new training methods opened further possibilities for their uses. Though some researchers still use them for biological modeling, much current work in this field, such as the popular back-propagation technique of error reduction, is not biologically tractable.

One of the main interest in neural networks is that rather than programming them explicitly, they are "trained". That is, they are exposed to a large number of inputs, and their behaviour is modified towards a certain response for these inputs. This modification is achieved by altering the extent to which each signal coming in to a perceptron affects its sum. Every connection coming into a unit is said to be "weighted", this weight is multiplied by the signal before being summed by the perceptron. There are many methodologies available for achieving this training, but it remains the largest area of research within neural nets.

2.1.2 Subsumption Architecture

The other alternative to sequential processing has been to extrapolate this distributed approach to computation into larger distributed units of intelligence, behaviours. The best known work written on this principle is Marvin Minsky's "The Society of Mind"[Minsky 85], in which Minsky argues that human behaviour can only be truly explained and understood if intelligence is considered to be broken into hierarchical units of separate, competing agents. The appeal of this approach is not only in explaining human behaviour, but in that he has explained a tractable way in which such behaviour can *develop*, and, more importantly to Artificial Intelligence, can *be* developed.

Some of the most interesting and successful research done so far in implementing behaviour based intelligence has been done in robotics. In particular, the Subsumption Architecture approach developed at MIT in the late Eighties by Rodney Brooks has resulted in several robots capable of robust and interesting behaviour with relatively short development times and simple processing units.

Brooks believes that true intelligence cannot be achieved without addressing the fundamental issues that most biological intelligence addresses, issues like moving and surviving in an unpredictable environment.[Brooks 91b] He criticises most AI programs with having as their input "a restricted set of simple assertions deduced from the real data by humans." In particular, Brooks attacks the approach of trying to construct an internal model of the world, and then constructing plans with that model which are re-applied to the external world. This is the central paradigm not only for most of robotics, but for nearly all Artificially Intelligent reasoning systems. Brooks states,

> We have reached an unexpected conclusion (C) and have a rather radical hypothesis (H)
>
> C — When we examine a very simple level intelligence we find that explicit representations and models of the world simply get in the way. It turns out to be better to use the world as its own model.
>
> H — Representation is the wrong unit of abstraction in building the bulkiest parts of intelligent systems.

Brooks' preferred "unit of abstraction" is individual behaviours, or competences. In order to make coherent robots out of independent behaviours, he has developed a strategy called *subsumption architecture*. Under this approach, the desired necessary functionality for the robot is broken into its component behaviours.[Brooks 89] These behaviours are then implemented one at a time, starting with the most basic competence. A successful implementation of a behaviour is a functional robot with the new competence and also any other competences that were already present. Each new behaviour is layered on top of previous ones, and thus are referred to as "higher level". Higher level competences may be aware of lower level competences, but lower level competences have no knowledge of higher level ones.

For example, since Brooks deals with mobile robots, his lowest level is composed of competences involved in avoiding objects that are in the robot's way. After having constructed a robot capable of moving freely, higher level goals and objectives are added via higher level modules. These higher levels do not need to be aware of the problems of avoiding objects, because if there is something in the way, the lower level modules solve the situation (see figure 2–1).

Figure 2–1: Schematic from [Brooks 91b] of Rodney Brooks First Subsumption Robot, Allen.

Requirements for an Intelligent System

In his seminal paper,[Brooks 91b] Brooks sets out four criteria for a successful "Creature", whether biological or mechanical, and explains how subsumption is used to meet these criteria.

1. Creatures need to be able to respond quickly and appropriately to changes in their environment. For example, if a wind comes up, a creature might need to modify its movement to compensate, and not doing this quickly enough might cause the creature to fall. Subsumption addresses this by having simple, low level behaviours which are dedicated to tasks like movement or balance, which are in continual operation and frequently sense their environment.

2. Creatures should be robust with respect to their environment. If the environment is in some way not ideal for the creature's operation, the behaviours of the creature should not totally collapse. Instead, you would expect gradual performance degradation as the environment becomes more hostile. This has been a significant problem with most artificial systems. Subsumption developed creatures are more resistant to such problems, partly because they have multiple behaviours operating in parallel which can address them, and partly because they work without a central representation with its incumbent expectations.

3. A creature should be capable of pursuing multiple goals. Further, it should be opportunistic in this pursuit, it should be able to take advantage of situations useful to one of its goals even if it hadn't actively been pursuing that goal when the situation occurred. Under subsumption, each of the competences can be thought of as having its own goal. And each competence activates itself when it recognises an opportunity to fulfill that goal.

4. A creature should do something, it should have a "purpose of being." Under subsumption, the purpose of being is defined in the goal of its higher-level functions.

Principles of Subsumption Architecture

In the same paper[Brooks 91b], Brooks outlines several criteria for a subsumption strategy approach.

- Each layer should be engineered separately, and tested and debugged until flawless before proceeding. Debugging a multilayer system is difficult, Brooks advocates this approach because it reduces the problems to being either in the current layer or the interface between the current layer and the previous one.

- A layer should be a fixed topology network of finite state machines possibly augmented with a few registers (for data storage) and timers. It sends and receives all messages over wires. It should be data driven, with no global data and no dynamic communication. "No dynamic communication" means that layers do not engage in two way communication — when a message is sent, there is no indication whether it has been received.

- Layers may only interact with other (lower) layers via their input and output wires. A higher layer may suppress the input of a lower layer, and it may inhibit the output. "Data driven" means that the system is *reactive*. Activity by a competence is a consequence of events or opportunities in the environment, not of instructions from another competence. "Suppression" of the input may include replacing it, but output may not be altered, only left unaffected or discarded.

- There should be no simplified test environments. As mentioned above, Brooks believes that most AI programs have their intellectual work done for them by the people creating their input. Subsumption applications should cope for themselves with all the noise of inaccurate sensors and all the unpredictability of the real world.

- There should be no representation. In his 1990 paper, he not only advocates against central representation but says there should be no variables instantiated, no rules matched, and no choices made.

These criteria have been used as the main guidelines for implementing this project.

Summary

Brooks gives a very telling summary of the difference between traditional AI approaches and these behaviour-based approaches in [Brooks 91a]. In traditional AI, programs are broken into information processing units, such as perception, planning, modeling, and learning, and intelligent behaviour such as avoiding obstacles and moving are expected to emerge from the interaction of these components. In a behaviour-based system, intellectual tasks like planning and learning are expected to emerge from the interaction of the basic behaviours.

2.2 Related Work

Most research using subsumption architecture has taken place in robotics. [Maes 90] gives an overview of projects, while [Malcolm *et al.* 89] provides a good discussion of the impact of this new paradigm on robotics in general. One particularly well known implementation is Ghengis [Brooks 89], a six-legged cockroach-inspired robot that trained its own neural nets to learn to walk. It was eventually capable of pursuing people over rough terrain.

Most robotics projects written outside of Brooks own laboratory tend not to use a strict subsumption, but to use a modified or "hybrid" approach. Frequent modifications include the introduction of planners and representations for higher level tasks.[Man 92] What many of these projects find most useful from subsumption architecture is the rules for communications between distributed agents.

I was unable to find any software projects previously written under subsumption architecture. However, much work in AI is done using distributed systems. Planning and scheduling is one area heavily using multi-agent distributed intelligence.[Durfee 88] For example, a planner may have different modules or "agents" dedicated to such tasks as releasing jobs into the system, load balancing, and error monitoring.[Gomes 92] Their interaction results in the required scheduling. The task itself is a good example of planning being emergent from appropriate behaviours.

There has been much research in applying connectionist architectures (neural nets) to various aspects of musical cognition.[Todd & Loy 91] However, these are largely unrelated to this project because

- they tend to apply a single net to an entire problem, rather than work on cooperating behaviours, and
- most of them work with pre-transcribed scores.

The most closely related work in terms of unscored accompaniment I have found reference to is Roger Dannenberg's jazz accompanist system. It is designed to listen to a few bars of improvised jazz and to then to join in on "piano" and "drums". [Dannenburg & Mont-Reynaud 87][Dannenburg & Mukaino 88] Unfortunately I was not able to track down the references in time to discover the approach he uses.

There are many programs that involve having a computer follow a score (rather than improvise) in order to play along with live musicians. One of the most interesting of these is Machover's Hyperinstruments. What makes this interesting with respect to this project is that Machover has chosen to use an approach inspired by Minsky's "Society of Mind" for the section of the program responsible for recognising where in the score the other musicians currently are. Another is Matsushima et al.'s organ playing robot [T. Matsushima & Ohteru 85].

Other related systems worth mentioning are Longuet-Higgins music tracking program[Longuet-Higgins 87], and Peter Desain's rhythm tracker[Desain 92]. The former is purely symbolic, while the latter is implemented as a neural net. Both are tackling a quite different problem than this project. Treating only the rhythm aspect for a moment, both of these projects are trying to find the exact rhythm of musical melodic events. Both do this by puzzling out the relationships between adjacent notes in a time frame known to be a fixed and significant length, a measure. In Longuet-Higgins program, for example, the measure is counted out before the beginning of the piece in beats. My project is unconcerned with the exact rhythmic events of the melody, but has to discover the beats.

Chapter 3

Project Overview: The Modules

The project as it stands consists of six modules. In order of implementation, they are:

1. NOTE: derives pitch class from the raw input,
2. CHORD: derives a chord from a group of pitch classes,
3. THRESH: seeks the beginning (threshold) of notes,
4. BEAT: finds the most probable time intervals between thresholds,
5. CHANGE: guesses when a chord change has occurred, and
6. TIMED: synchronises the output of BEAT with the melody to anticipate the beat thresholds.

As each competence was developed incrementally, there are in fact six "robots" or programs, each typified (and named for) its highest level competence. It is important to understand the differences between a robot and the competence of the same name.

- A robot contains not only its namesake competence, but all the competences that module subsumes. Individual modules are NOT composed of previous competences, but are separate and distinct objects.
- A robot is an interface between the modules and the user. It does not necessarily display all the output information generated by even the highest level modules; in general it greatly simplifies this data. It may also perform

simple extrapolations on the data for the benefit of observers, using heuristics extrinsic to any of the competences.

This chapter begins with a basic description of the overall program. Next I sequentially describe each of the robots in terms of the interaction of their competences and their user level output. The following chapters explain the modules in more individual detail. To see the complete diagram of the interactions of the modules, skip ahead to the section on TIMED and diagram 3.2.6 on page 26. For a discussion of the the programs' input, see chapter 4. For examples of the robot's output, see appendix **??**.

3.1 Overview

3.1.1 The Task

The purpose of this program is to derive chord structure from a melody in real time. This emulates the human competence of providing chord accompaniment to unfamiliar music. The melodic structure consists of a series of pitches of specific time duration arranged sequentially in time — that is, a monophonic melody. A chord is a group of pitches occurring simultaneously. The chord structure for a song is the sequence of chords, also associated with durations, which is considered to be a harmonious accompaniment to the melody.

Just what is "considered harmonious" is subjective. Extensive systems of formalised rules for harmonisation exist, but even these are not deterministic. Human accompanists will often choose different chord patterns for the same melody, either from other accompanists or even from their own previous interpretations. In addition, Celtic music did not develop polyphony until recently, so some of the melodies in this project were created with no knowledge of the chord types currently applied to them. Nevertheless, acceptable accompaniments of such simple melodies are fairly standardised. The success of this program can be evaluated in two ways — by comparing it to the chords derived for the same input by several musicians, and by having musicians judge whether the chord structure of the piece "sounds reasonable".

3.1.2 The Approach

The pitches that compose the chords are derived from the pitches that compose the melody. And the time-wise occurrences and durations of the individual chords is a part of the same rhythmic schema as the time events in the melody. Thus, two of the most important tasks for a robot deriving chord structure are interpreting the pitches and the time schemes of the melodic input.

In my implementation, I chose to make Pitch recognition the most basic level of competence. This level outputs possible pitch classes with associated probabilities. The next layer is Chord recognition, which transforms the output from pitches into chords. The following level is Time recognition. Although the basic competence of this level is to recognise the beats, the output of the program at this level is not just rhythmic information, but the modified output of the chord layer. Now instead of a single chord, it outputs chords in a timed sequence.

SONG
STRUCTURE
TIME
CHORD
PITCH

Figure 3–1: Basic Layers of a Complete Machine.
Bold face script indicates implementation.

Should this project be continued, the next layer would try to recognise Structure in the piece. That is, it would look for recurring patterns within the piece. This has already been partly implemented with respect to time in the final module — TIMED tries to find and anticipate basic rhythmic structures. The final level of competence would probably be Song recognition, whereby the program would learn to recognise song types. This would subsume Structure in its ability to anticipate the number of times a pattern would repeat, and what the next pattern is likely to be. (See figure 11.2 on page 90 for schematic version of the proposed final robot.)

This last level raises two important points about the nature of this project. First, assuming the program could recognise (or was told in advance) the exact song and thus already "knows" the chord sequences and duration, this would not mean the program is finished. It simply means that the accompaniment is more likely to be accurate. The highest levels do not *replace* the lower levels, they *refine* them. The robot must still "listen" for changes in time, key, and even melody in order to effectively accompany the piece. Second, within the context of this program as I have implemented it, recognising a "song" does not mean recognising a melody. This program at no point attempts to transcribe the actual melodic input. Consequently, melodies human listeners would consider to be different would be considered the same "song" by this system if they have the same chord structure.

3.2 The Modules

The following is a record of the modules in the order of their completion, and a description of their associated robots. The purpose of this section is to describe how the modules relate to each other, both in terms of functionality and development. For fuller descriptions of the modules implementations, see the chapters bearing the same names.

The robot descriptions are accompanied by diagrams of the flow of information through their system. These consist of

- heavy arrows indicating input and output interfaces with the external world,
- large circles representing the modules,
- smaller arrows indicating the internal flow of information,
- small open circles indicating where a module taps information from lines between lower level modules, and
- small filled circles indicating where a module affects these lines.

3.2.1 Note

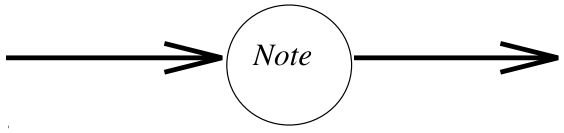

Figure 3–2: The Note Robot

The NOTE competence was the first developed. It takes audio input (microphone input run through the sound board of a SUN SPARCstation) and outputs a complete, weighted list of pitch classes. The note robot prints out a list of the highest weighted pitch class for each of a series of overlapping time frames.

3.2.2 Chord

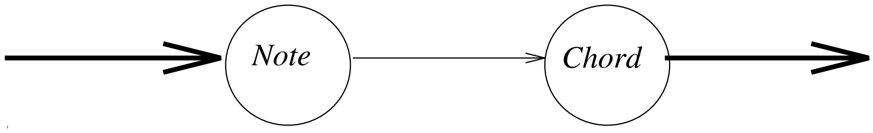

Figure 3–3: The Chord Robot

The CHORD competence takes as input weighted pitch class information such as the output of NOTE. Its output is a similar weighted set of chord types. The chord robot compiles all of the outputs from NOTE for an entire sequence of input, and feeds them into the chord competence. Its output is a list of possible chords and their weights. If run over an entire song, the result is the key of the input melody.

3.2.3 Thresh

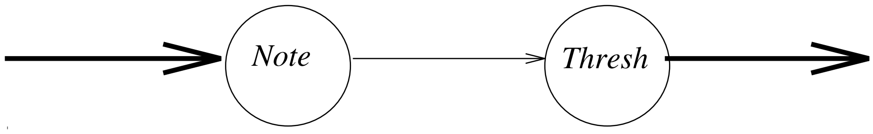

Figure 3–4: The Thresh Robot

The THRESH competence also takes as input a sequence of pitch class output like that of the NOTE competence. It searches for a stable pitch following a period of transition, which it considers to be the threshold of a new note. Its output is a boolean True or False for each input, where True signifies a threshold. Notice that since it looks for stability, this signal necessary comes after a time lag linked to how stability is defined. The thresh robot prints out the time frame numbers and the output of the thresh competence, 1 for true, 0 for false. Notice that this a second subsumption of the NOTE competence with no relation to the previous one, CHORD.

3.2.4 Beat

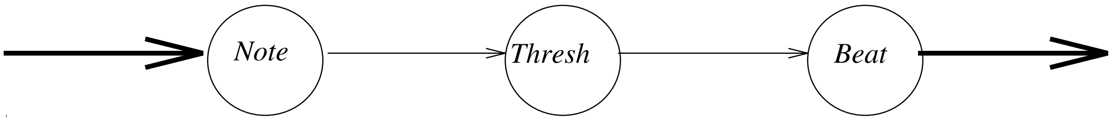

Figure 3–5: The Beat Robot

The BEAT module looks for common intervals between note thresholds. It takes as an input boolean output like that of the THRESH module. It records the amount of time between positive signals, and creates an array of possible time intervals, with associated weights for those most likely to occur. The existing weights decay over time, so that time changes can be tracked. The beat robot prints a list of time frame numbers, and when a threshold is perceived it prints the three most heavily weighted duration figures next to the frame number.

3.2.5 Change

Module CHANGE takes as input both a sequence of booleans representing note thresholds in time and a sequence of weights representing notes. Its purpose is to subsume the CHORD competence to find locations within the song where the chords change. If CHORD is run without interference, it soon stabilises to reporting the key for the melody.

The CHANGE module actually modifies the inputs for two separate incarnations of CHORD. One represents the expected value of the chord, which at this level is its previous value. The second represents the possibility that a new chord has

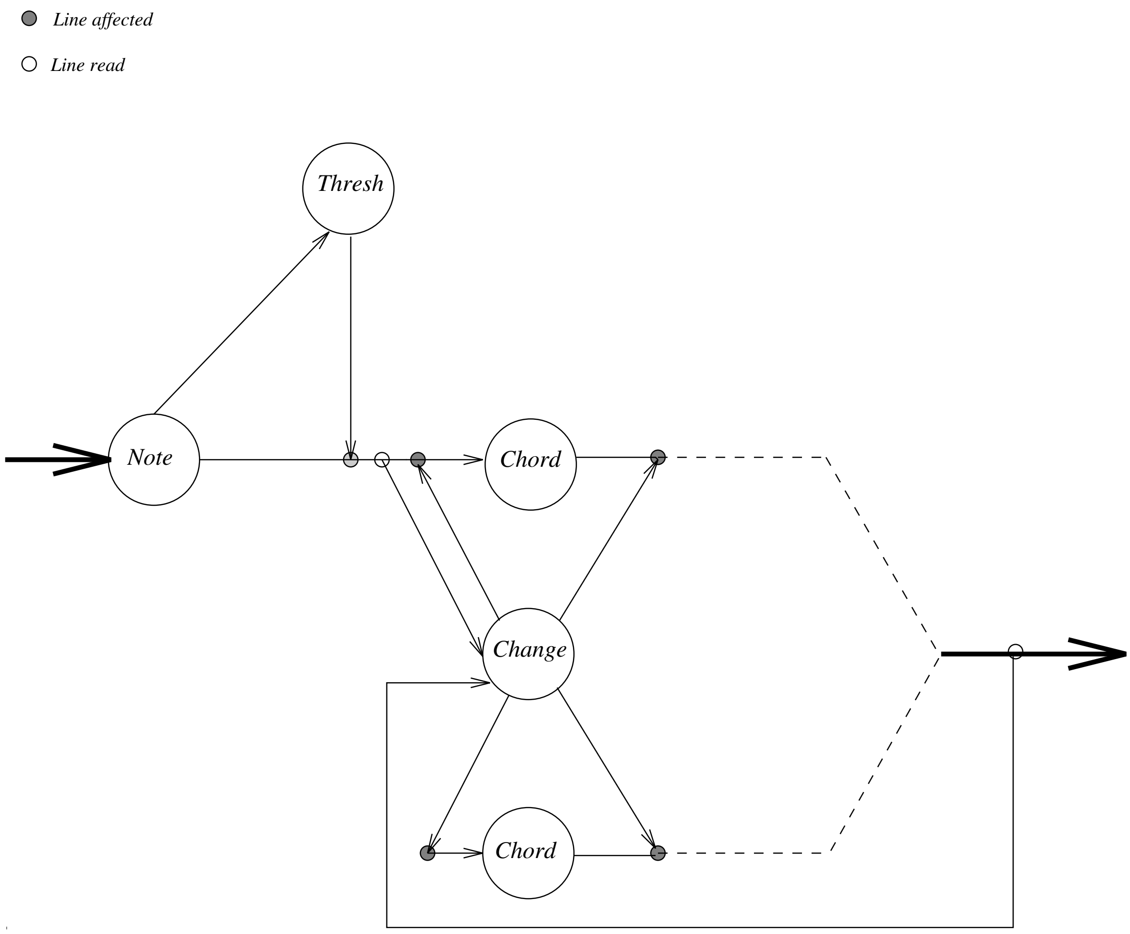

Figure 3–6: The Change Robot

started. CHANGE's output is the output of the CHORD module that seems most strengthened by the current trend in the melody. It uses the threshold signal as a possible time when a new chord should start. The change robot uses THRESH for its time input, and prints out the time frame, the leading chord theory, the secondary chord theory, and, if a THRESH signal was received, the word "beat".

3.2.6 Timed

The TIMED competence takes as input both a sequence of booleans representing note thresholds in time and a sequence of weights such as are outputted by the BEAT competence. It uses the BEAT information in order to build and maintain a guess at what the dominating time intervals are, and the THRESH input to try to determine when these events occur. Its output is a boolean stream similar to

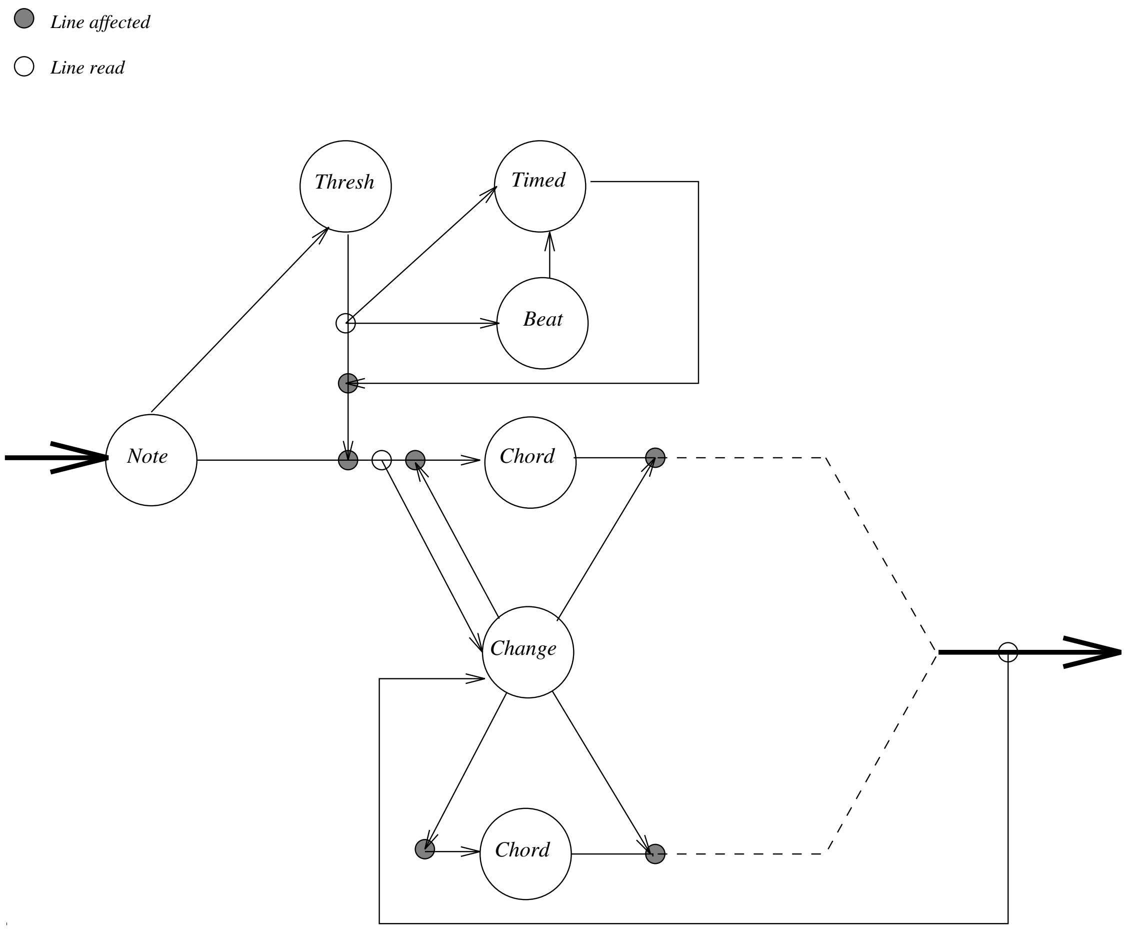

Figure 3–7: The Timed Robot

the output of THRESH, but it is corrected for the lag and reports regularly at the duration proscribed by the interval being reported. The format of the output of the timed robot is identical to that of the change robot, though its content differs substantially, because TIMED triggers CHANGE at regular and significant intervals. There is also another, alternate robot for TIMED called "noise" which produces auditory output.

This module is the last completed, and as such its robots represent the final state of the project.

Chapter 4

The Note Competence

The NOTE competence derives a pitch class for a given sound from the system-level input. The term "pitch class" refers to the concept of pitch modulo the octave. By "system-level input", I mean the input which comes from outside this program: the sensor data.

This module is essentially a neural net. More precisely, it is a competitive net, with each competing output representing a pitch class. The reaction of all of the output units is recorded as the real output of this module, though the note robot reports only the winning output.

4.1 Design Motivation

The decision to make the output in terms of pitch class rather than just pitch was based on the goal of the overall project — to derive a chord structure in real time. Since melodic transcription is not required, octave information is superfluous. Further, limiting the range of the NOTE module's output (and consequently any dependent module's inputs) reduced the number of calculations required, saving processing time.

The choice of a neural net over a conventional pattern recogniser for this competence was an engineering decision, though based partially on the biological model. Sound input for a single note produced by any conventional means consists of a large number of component frequencies. These frequencies are permutations of the base frequency with which we identify the pitch. Even if this base frequency is missing, a person will still hear the tone as being that frequency if a sufficient number of the related frequencies (or *harmonics*) are present in their correct proportions. Further, people tend to identify tones within some spectra as a certain pitch, but spectra between adjacent pitches are not continuous - some notes are considered "out of tune" if they are too far from their ideal.

Training a neural net to make the appropriate classifications seemed the easiest way for handling the complexity of these many variables. If necessary, the net can be retrained quickly for any scale — so long as it has the same concept of octave and the appropriate number of pitch classes is defined (a compile-time constant). The exact thresholds between pitch classes were considered irrelevant, since any tone too close to a threshold would sound out of tune to a human listener as well.

As explained in section 2.1.2 of the Theory chapter, the idea of working with real-world data is fundamental to Brooks' approach. For this reason the net was neither operated nor even trained on artificially generated "pure" data, such as MIDI input or computer generated wave forms. If time allowed it would have been interesting to see whether a net trained on such input could have performed in comparison (see the Further Work section of this chapter).

An underlying assumption crucial to the nature of this module is understanding that we do not assume that the melody is either correctly received or perceived. Errors of a few notes by either the performer or the listener should not have a severe effect on the interpretation of the piece. This criterion is found not only in the robustness Creature requirement, but in the musical behaviour being emulated.

4.2 Description

4.2.1 The Input

The first stage of the input is a microphone. This is run through a Sun SLC audio input using their standard audio board. These signals are recorded into Sun format audio files using their standard *soundtool* utility.

The Sun header sound files are then translated into another format (IRCAM) so they can be used as input to a utility called **pvanal** that comes as part of the **csound** digital audio processing package[CSO91]. Pvanal translates this into Short-Time Fourier Transformation frames. Each frame contains 129 frequencies that occur over a 256 millisecond time interval and their associated gains, in roughly ascending order ranging from 0 up to about 4000Hz. These Fourier frames are the actual input to the network.

4.2.2 The Net

Morphology

The network is a two dimensional matrix. The input units are each associated with a specific frequency. These frequencies, and the overall number of input units are both set by reading a file on execution and so are easily varied for experimentation. The outputs represent the pitch classes; their number is set by a compile-time constant, `PITCH_CLASS_NUM`.

Each input line is connected to every output line, and a weight associated with it. For a trained net these weights are read from the same input file as the frequencies. In training, the file may contain only frequencies, in which case all the weights are initially set equal to each other. The sum of all the weights for any particular output unit is always 1.

The frequencies used for the current version of the robot are scaled evenly across octaves. There are currently 96 frequencies per octave (8 per pitch), and they run from 220Hz through 4000Hz. The reason for using scaled gaps between frequencies instead of even ones was we felt this would help bias the net towards correct relations between harmonics and octaves. (see Design section below)

Execution

In execution, a set of frequency/gain pairs are presented one by one to the net. Each frequency is assigned to an input unit. Once an input channel has been selected, the gain from input is multiplied by each of the weights for that channel, and each result is summed into the output node associated with that weight. (See

Figure 4–1: The Conversion of Sound into Pitch Class
Soundtool captures the microphone input, Pvanal does a Fourier transformation into frequency/gain pairs, NOTE clusters them into input lines and passes them through a neural net, the output units of which represent pitch classes.

figure 4–1.) This process is run for the entire set of frequencies and gains for a single Fourier time frame. The result is an array of summed weights, one for each pitch class. NOTE also provides a proportional version of this array, where the weights sum to 1.0. Because of difficulties with the relevance of the absolute output to the gain level (see the Analysis section in this chapter) it is this proportional information which is used by higher competences.

Training

The training program associated with this net may be run using either supervised or unsupervised learning, the choice being a command line option. Supervised training was used for generating the frequency/weights files actually used by the NOTE module.

For unsupervised training, a simple winner-take-all strategy of training standard to competitive nets is used. Conceptually, this strategy encourages pattern discrimination by reinforcing the winning output for each trial, so that it is more likely to respond again to the same signal. Practically, each weight associated with the given output is decreased by 10%, then the weight thus gained is redistributed to the to the weights that had input, in proportion to the amount of input they saw. Or, in pseudo-C

```
for (i = 0; i < number_of_frequencies; i++) {
    /*generate communal pool of weights by taking 10% of each*/
    weights[winning output][i] *= 0.9;
    /*redistribute the weights based on relative input*/
    weights[winning output][i] += 0.1 * input[i] / input_total;
    }
```

Supervised training uses essentially the same algorithm, except that instead of reinforcing the winner, it reinforces the output that *should* have won. If the winner is a different node, then the bad winner is *negatively* reinforced by applying the above algorithm in reverse.

The actual supervision requires two files: one of the standard audio input files described above, and a key file stating the pitch class appearing at a given time.

(See ??.) The audio training file used for generating the frequency/weights files used for the operational NOTE module consists of the first twelve frets of each string of a mandolin being plucked repeatedly for five seconds each. The training program selects random frames within this file (though avoiding the thresholds between notes to buffer timing inaccuracies).

The original file delimiting the frequencies for the input units is generated by a PERL script fw-gen.prl. This program essentially generates and counts the scaled frequencies over a range which can be changed by editing the script. By my own convention, these frequency/weight files end with the .fw tag, regardless of whether they actually contain weights. The training program is set up to produce an output file and exit when a certain level of convergence is reached. It can also generate files during its run, which allows you to see the course of development of the training. But this requires more run-time interference by the user as these files overwrite each other.

4.2.3 The Program Code

Much of the code for this module is a hodge-podge of signal processing and binary file formatting. These programs are written in C, two of them (as mentioned above) are external to this project. There are also several PERL scripts for creating parameter files and running statistics on files to evaluate them. A full listing of the programs and the call parameters used in my experiments is in the appendices, along with samples of all the files used.

The neural net used for determining pitch class (hereafter referred to as the "note net") is written in C++, as are all the modules. There are four object classes defined — Infft for inputting the raw audio input; Freq_ins, for storing and passing the input; Range, for representing and manipulating arrays of size `PITCH_CLASS_NUM`; and Note_net, for representing and running the entire net. The actual network is an array of type Range, sized as the number of frequencies from the frequency/weight file. The raw and proportional outputs from the net are also of type Range; consequently so are the inputs to any dependent modules. (See figure 5.2 on page 48)

Training the net requires the addition of just two derived classes. Intrain, derived from Infft, adds the features of skipping randomly around the input file and finding the appropriate keyed response if training is to be supervised. Train_nn, derived from Note_net, adds learning, that is, it can adjust the weights of the net after execution.

4.3 Development History and Analysis

NOTE was the first module developed, as I considered it the most basic competence to the project. The bottom level rhythm module is technically equally low level, but since pitch and chord recognition at least aesthetically seem more important to chord structure, I chose to work on this first.

The module took about two months to develop. This can be roughly broken down into (in chronological order):

1. one week of learning C++,
2. two weeks writing the code (slowly because still learning C++),
3. two weeks getting the input in a usable form
4. one week writing the net's input routines and debugging the code
5. one week running experiments and debugging supervised training.

For the sake of coherence, the rest of this section is organised by concept, rather than giving an over-all chronology of development.

4.3.1 The Audio Input

The initial step for the input — recording the sound into a file — was a simple matter of acquiring hardware then running the SUN soundtool utility. The next link in the chain was to be pvanal, the Fourier transformation program associated with csound. However, pvanal would not accept SUN sound files as input. Consequently, I eventually wrote a C program **sun2ircam** that transforms them into IRCAM format.

Once pvanal was operational, I had to decipher its output. The format of the output of the pvanal program was under-documented, and the code I in the csound libraries provided for reading the file corrupted the header. I wound up writing several programs for viewing and analyzing the Fourier Frame files, and determining their actual format from a combination of their documentation, their source code, empiricism, and trial and error. The programs I wrote for this analysis include:

- **fft-look**, a C program that converts the binary data to ascii.
- **max-freq.prl**, a PERL script for finding the maximum frequency in the file. This turned out to be around 4000Hz, which is far below the upper threshold of human hearing, and could therefor be a cause of problems with identifying high frequency pitches, since there would be less harmonics to classify with.
- **freq-dom.prl**, also a PERL script. This program shows the five dominating (highest gain) frequencies for each frame. This was written on the suggestion of my supervisor, Alan Smaill, because I was concerned that I couldn't tell the difference between a file of recorded silence and the training file by looking at the fft-look data. Freq-dom showed some appropriate frequencies for known notes in the training set.

The final work on the input was comprehending the meaning of the header variables relating to the time period represented by each individual frame. This information was necessary for correlating the files to the supervision keys, which are time indexed. This was achieved partly though the documentation, but largely through the use of the **gdb** gnu C++ debugger[GDB91], a calculator, and a knowledge of the actual time extent of the over-all file. For the results, see the declarations and constructor functions for the Infft and Intrain class objects.

While I was able to complete this project using audio input in this format, it is really inadequate for anything but a prototype. The biggest problem with it is that the Fourier transformation has to take place entirely before the rest of the project runs. Thus the parameters of the transformation (frame size being the most important) are out of the control of the rest of the project, and the project is out of even a semblance of real time.

The other problems with the audio input have less impact on the structure of the overall system. It is inaccurate for low frequencies and non-existent for high, and the gain information is normalised rather than absolute. The frequency problem can be worked around by constraining the range of the melodic samples. As for the gain, I decided to ignore volume and concentrate only on duration for the significance of notes. Though this is a loss of information, it is also a reduction in complexity, which is good both for a real time system and a time constrained project.

4.3.2 Net Topology

The pitch class recognition network was originally conceived as being essentially a classic competitive net [J. Hertz & Palmer 91] with three differences:

1. the training would be at least partially supervised,
2. the output would not be binary, but quantitative, reflecting the loudness of the actual note.
3. the outputs of more than just the winning unit would be considered significant.

The first point is discussed in the following section on training. In the end, full supervision was used (see the Implementation section above).

The latter two changes were trivial, it actually involved just *skipping* a step in the competitive net and leaving the output signals as they appeared. The standard neural net algorithm does not guarantee these values to be significant, but empiricism has shown that they they are.

Originally, I had hoped to use the quantitative output as a measure of the loudness of the pitch. This would have been very interesting, because it would have meant the net was actually processing two related but separate kinds of information simultaneously. However, the initial quantities seemed to more reflect the certainty of the note (being very low when the input had actually been noise or silence) than it's volume. It was not initially clear whether this could be resolved with further training — in fact it turned out to be a feature of the input data.

As mentioned in the previous section on input, I decided that loudness was not necessary for the project to proceed, as duration would provide much the same information. For determining the importance of a pitch over a period of the melody, the number of frames it is present for is sufficient.

Maintaining the quantified output in relative form has proven useful. Often if the correct pitch class is not the winner, it is still a leading contender, and thus has a positive contribution to pass on to the chord competences. Similarly, polyphony in input is successfully transcribed. Experiments run under the chord competence testing the scaled quantitative output vs. simple winner information showed better results for the scaled output. (See section 5.3 of the Chord chapter)

4.3.3 Frequency Perceptors

One of the basic parameters that needed to be set for the network was the frequencies for the input units. The range the units would cover and the density of the units were the first obvious questions. It then occurred to us that distributing the units in terms of octave instead of regularly on the basis of frequency might help the net make its generalisation across octaves we were looking for. Distributing the input lines logarithmically allowed the input to be perceptually uniform. (See figure 4–2.) This arrangement is also found in the frequency perceptors in the human ear.

4.3.4 Training

Training is central to use of a neural net. Training this net to be "good enough" proved amazingly easy; in fact it tended to take only on the order of 100 sample input frames. However, training to optimal was never really achieved. I just used a few of the best resulting weight sets for my experiments, and for different tasks different ones performed better than others.

I initially saw three ways to train the network — unsupervised training, partially supervised training, and fully supervised training. The idea of *partial* supervision instead of complete supervision was particularly interesting. The idea was this - since there are natural harmonic relations between various pitches, en-

Figure 4–2: Input to Note as it Appears in the Frequency Perceptors
Height corresponds to gain. Input is the first 20 seconds of the training file. There are only 12 frequency perceptors per octave for the purpose of this graph, though in the robot there are 96. The bands represent the harmonic series, see section 1.4.2.

forcing a given pattern of output might prevent the network from exploiting these natural organisations and make training unnecessarily difficult or even nonconvergent. However, totally unsupervised training would have no particular reason to choose pitch class as its discriminating point. Under partial supervision, one node would be chosen to represent a particular pitch class, and its behaviour would be supervised as described in the Implementation section above. At the same time, the other nodes would be trained using the unsupervised method when they won. Once the chosen node was trained to a certain performance level, the other nodes would be checked for performance, and a tentative map made of pitch classes to output nodes. The next student node would be the node with the most consistent performance still below the target level, or if only a group of non-discriminating nodes is left, one would be chosen at random and assigned one of the remaining pitch classes.

This strategy was abandoned in favour of full supervision for both theoretic and pragmatic reasons. First, I realised this approach would make more sense in a neural net morphology where individual weights impacted on more than one output unit. For a grid, the effects of a weight on the outputs it is not directly connected to are only indirect — they compete with their equivalent input weight for those units, and fairly equivalent across all the other units. Also, when I spent more time than I expected on the audio input, I realised I was running short on time for the other modules, and decided that I was more committed to a study of subsumption architecture than of neural nets in this project.

4.3.5 Results

Unsupervised

I chose to run some experiments with unsupervised training for two reasons:

- out of curiosity, to see if the net would classify pitch classes naturally, and
- as a development tool, so that the nets execution and weight training could be debugged without the more complicated error correction routines added on.

Initially the unsupervised training only affected one or occasionally two of the output units. Examining the trained weight files, I decided that this might be because of all the unused weights higher than the threshold the pvanal output. Whichever unit which first won would be at a significant advantage as a large number of units would necessarily be trained to favour that node again, whatever the input. When I removed these extra input lines, the training expanded to five or six output units.

At this point I could examine the output of the net. I found that —

- Frames were often classified the same as their neighbours — there did seem to be patterns.
- The thresholds between given units dominating the output were sometimes also pitch class thresholds, but often they were not.
- The patterns and thresholds were largely consistent over more training cycles. Even between separate training sessions the patterns were similar, although the output units representing different parts of them changed.

My initial hypothesis was that the output units were picking up on something I considered unimportant, such as volume, or electronic noise generated in the room during recording. But I found producing fairly reliable output very encouraging.

Supervised

My first problem with supervised learning was that I forgot to include a method for testing when training is completed. The first method I tried applying was to look for stability in the weights. However, I found that the weights never stabilised, a common problem for competitive nets. This is because the training system is based strictly on competition, where in other neural net training strategies the amount of training is proportional to the error. My next attempt was to train for a period, then to freeze training and run a randomly generated test across the data. Using this data I found that the net trained extremely quickly to a maximum accuracy of about 86%, though it fluctuates in accuracy around a point of about 75%. The curve to these ratings is extremely steep, with accuracies around 66%

being achieved when a mere 80 samples have been demonstrated, and performance tending to peak by 200 samples.

I was concerned that the correctness level seemed a bit low, but when I examined the output sequentially, I found that within a single note which is being correctly identified, there were regular brief periods of misidentification. I checked against the audio input, and found that these periods actually corresponded to the plucking of the string, which apparently created too much noise to be classified correctly. Other errors I found were periodic reporting of a strong harmonic for note, generally its second harmonic, or locations in the file when a second string had accidentally been left ringing. Consequently I decided that much of the error may be in the training file itself, rather than the network.

Unfortunately, this method of training and evaluation left me open to sloppy errors. For example, in one of the weight files, a seldom used pitch class' output unit was only 50% accurate, a problem I didn't notice until I was trying to use it for chord analysis. Certainly a more reliable method of finding optimal weights should be developed. (see Further Work section below).

Generalisation

Once I had acceptable results from the training set, I was able to test the note net on various other inputs. I found that it generalised completely successfully to other inputs with the same mandolin, and that it could also be used for penny whistle, and Scottish small pipes. On the other hand, it did not work successfully on the human voice, and on only some parts of the range of a guitar and a recorder. I suspect that some of the difficulties with these instruments were not with any intrinsic harmonic problem, but with the ability of the hardware to accurately record the instrument. NOTE seems to have a lower tolerance for noise and distortion in the input than the human ear, but this may well be a consequence of the Fourier transformation program rather than the network.

My experimenting with the guitar showed that the net could successfully identify pitches from octaves lower than it had been trained on.

One other interesting result was that I compared the output of the supervised and the unsupervised training, and found that they were in fact highly corre-

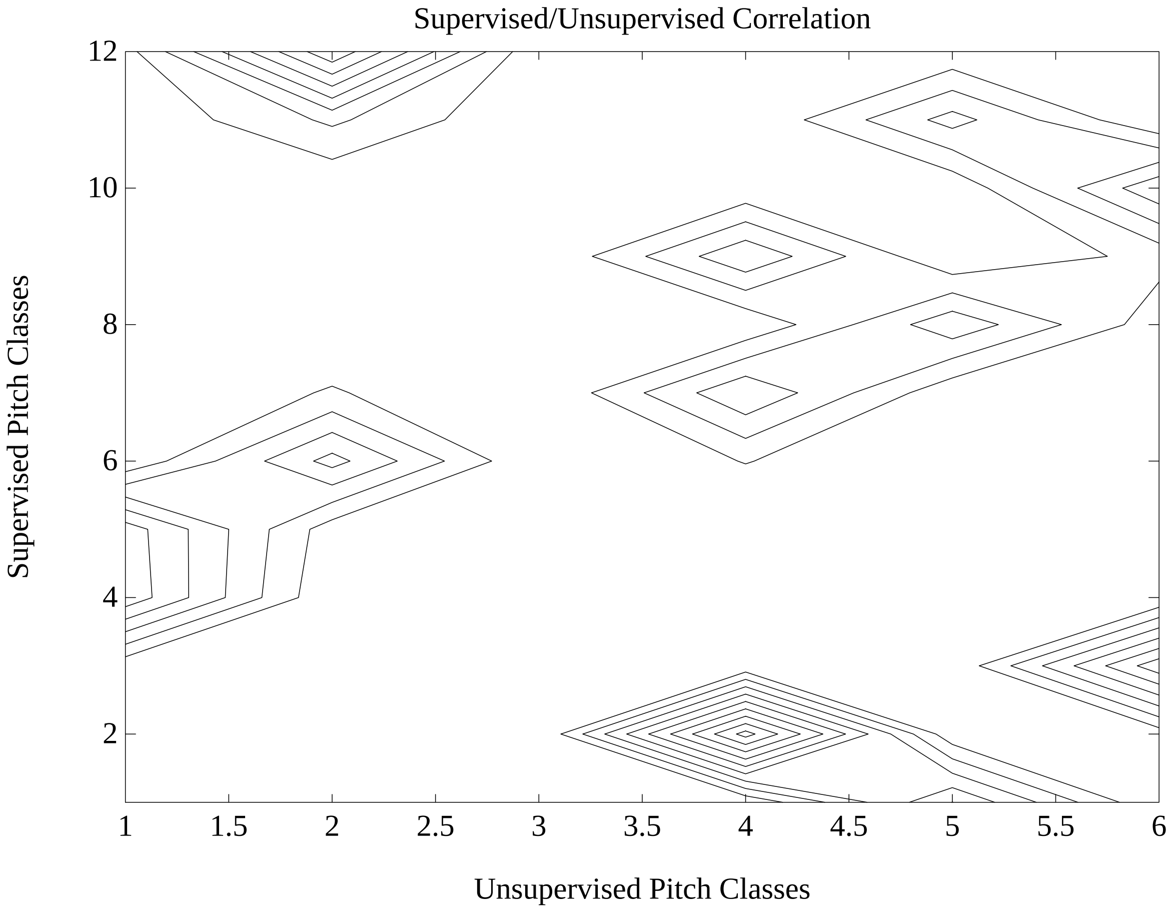

Figure 4–3: The Correlation between Supervised and Unsupervised Training
Note: Unsupervised training only used six output units.

lated. Some pitch classes were classified nearly entirely by a single unsupervised classification, while others were split across just two. (see figure 4.3.5)

4.4 Further Work

The following are suggestions for variants and extensions specific to this module. Larger scale suggestions for further work can be found in the Further Work chapter, chapter 10.

- *Optimise the weight training for the neural net.* It would be nice to have the training program declare optimal convergence, or at least preserve the best performing weights until asked to dump them. As explained in the Development section above, full convergence on an imperfect training net would actually be a bad thing, as would over-specification to a particular instrument, though these may be correlated with the next item.

- *Experiment with more training files.* I only used one training file since it evidently worked well enough. It would be interesting to try:
  - Machine generated "pure" tones. I worked off of Brooks recommendation to use realistic data. However, it's possible that pitch recognition is so mathematical that a good weight file could have been derived from absolute inputs. One concern would be robustness with respect to intonation - slight variances in pitch. I would rather the output level not be correlated to this.
  - Files composed of more than one type of instrument. This might aid generalisation.
  - Files containing more balanced input. Training may be skewed by the fact that the pitches in my file are biased towards the open string pitches of a mandolin. It might be better to use an instrument such as a piano. Again this is an issue of accuracy over robustness.
- *Experiment with more frequency sets.* It would be useful to know the training performance with more or less frequencies per octave. It would be nice to know the exact effect of having frequencies scaled to octave rather than by standard increments. It would interesting to try to actually model the human ear, though note this would involve not only matching frequency mappings, but could involve having distributed input signals. Also, clarify the effects of having frequencies beyond the range of csound.
- *Develop the note recognition net for input directly from SUN audio files.* Bypassing the Fourier Transformations would save a lot of time, and could preserve more information. This would involve making the note net much more complicated, at the very least recurrent. It would also have ramifications for much of the project. Therefor the complete discussion of this is in the overall project chapter on Further Work.

4.5 Conclusion

The NOTE module effectively transforms audio input into pitch classes, and as such is a successful first layer for the chord structure robot. The output is not totally reliable; it varies depending on the training session at which the weights were generated and tends to be random for noise. Some training sessions result in weaknesses in particular classes, or files that generalise less well to other instruments than the one trained on. However, the fact that it does generalise between octaves, recording sessions, and instruments makes it a considerable success.

Chapter 5

The Chord Competence

The CHORD competence converts a weighted array of pitch classes, such as the output of NOTE, into a weighted array of possible chords. A "chord" is denoted by a pitch class and a chord type, such as major or minor. The possible chords are stored in a text file which is named as a command line argument and read in by the program.

CHORD is not only a pattern matcher, it's an interpreter. Its purpose is to convert the information from its fairly mathematical input into symbols recognised by musicians.

CHORD is another neural net based competence — this time a hybrid between an associative and a competitive net. It executes like a competitive net, nearly exactly as NOTE executes, but its training is explicit, done in one instance from weights created from the definitions of the chord types.

5.1 Design Motivation

5.1.1 Approach

There were two major decisions to be made about the approach to take on a chord recognition behaviour.

1. Should the module learn to identify and classify chords itself, or should it be given the rules for chord identification?

2. Should the chords be recognised absolutely, i.e. C minor, F major, or should they be recognised in relative terms, i.e. I, IV, V?[1]

Taking the idea of self learning to an extreme, I could conceivably have had the module try to self organise a system of harmony, that is, choose its classifications of chords. While an interesting idea, I am more interested in producing output in terms of human conventional harmonic systems.

On a more realistic level, I could have chosen to attempt to train a neural net to recognise chord types. This would have been the rough equivalent of having the NOTE competence trained to recognise pitch classes. However, I felt the two tasks were qualitatively different.

All of the frequency space is divided into a fixed number of pitch classes in a relatively well defined way. Further, all pitch classes are essentially equally important. The number of chord types one uses is less well defined, some musicians tend to use more than others. In particular, a novice musician will tend to use less chord types than an expert, which certainly is not true for pitch classes.[2]

One of the reasons this difference is significant is that a problem with a fixed set of equally probable solutions lends itself much more easily to a competitive net. To use nets for chord recognition would have required either

- Determining in advance a complete set of potential chord types to be used. This strikes me as either being too constraining or a waste of time, depending on how large a set of chords you use. As in most cases where you try to make such assumptions in advance, it would probably be both.

[1]The components of an absolute chord are defined as particular pitch classes. The components of a relative chord are defined in relation to the overall key of the piece.

[2]Actually, one could find counterexamples for most of these arguments. For example, there are instruments which are biased for only certain scales so that a novice would likely miss out on some pitch classes, where experts might use more complex melodic devices. Also, some styles of music use extremely constrained chord sets. However, I don't think exceptions cases disprove my general case.

- Retraining the entire net every time a new chord type needed to be added. Also time consuming, and I suspect this would become increasingly difficult as the more obscure chords would tend also to be harder to discriminate.

- Having separate nets for each chord or chord type and training them just to recognise their own target.

I felt that both of the first two options would not only be time consuming but were poor software engineering from a maintenance standpoint. If you apply the third system by chord rather than by chord type, then you have essentially the system I developed, except for the additional overhead of training.

So why not train a separate net for each chord type? Basically, I felt it was unnecessary. You would still need to have a method of identifying the absolute key, which wouldn't be very far from the chord competence anyway. The basic purpose of the CHORD competence is to do a direct translation of the information from the NOTE competence into an arbitrary form of expression. NOTE derives information from the environment, it has to be robust and do a great deal of classification. CHORD already has the benefit of this processing, so its task is more straightforward.

5.1.2 Morphology

The inspiration for the actual morphology of CHORD's network was both an associative net and the competitive net such as was used in NOTE. I thought an associative net would be appropriate, since I saw this as the kind of pattern matching task they are particularly suited for. That is, one with partial (missing some of the pattern) and noisy (having data that is not part of the pattern) cues.

An associative net is normally a large grid, which is trained by presenting it with pairs of binary patterns. Wherever on the grid an input wire crosses an output wire, a weight is assigned of value one. In use, when an input pattern is presented, all of the weights which are activated for each output wire are summed, and if this number is more than a certain threshold, that output wire is considered to be on. (See [J. Hertz & Palmer 91].)

The associative net appealed to me because it is particularly well suited for recognising the sorts of inputs I would have, and I thought the single-act training appropriate for this task. What I didn't like about this kind of net was the necessity of having a large pattern for the output, and the fact that if many patterns are stored in the net, there is a good chance this output may contain some noise itself. It seemed to me this was creating a recursive problem.[3] Also, I realised that for many segments of music the possible chord interpretations would be ambiguous (see section 7). For these reasons I wanted a competitive net with single output units to represent each chord and with weighted results as I have with NOTE.

My end solution was to use the general form of a competitive net, but the weights for each output class resemble those of an associative net, as does the training. The one further modification I made was that rather than leave the values for nodes that do not correspond to a chord at zero, I made them negative. This is necessary to distinguish between some chord types

5.2 Description

The CHORD network can be thought of as an array of single layer nets, each of the size `PITCH_CLASS_NUM` — the same size as the input array. Each of these little nets represents a single chord, and is associated with both a type and a pitch class. (See section 1.4.2.) The training information is read from a file which specifies which pitch classes are a member of which chord. The file lists each chord explicitly, but only one example of each chord type need be generated by hand, I have a utility program written in PERL which shifts this pattern for each additional pitch class.

[3] I later read in [Brooks 91a] Brooks' criticism of neural nets under subsumption on the basis that "they do not have simple translation into time varying perception... They need extensive front and back ends to equip them to interface with the real world." I think I have nearly entirely avoided this criticism in this project, with the possible exception of the back end of BEAT (chapter 6).

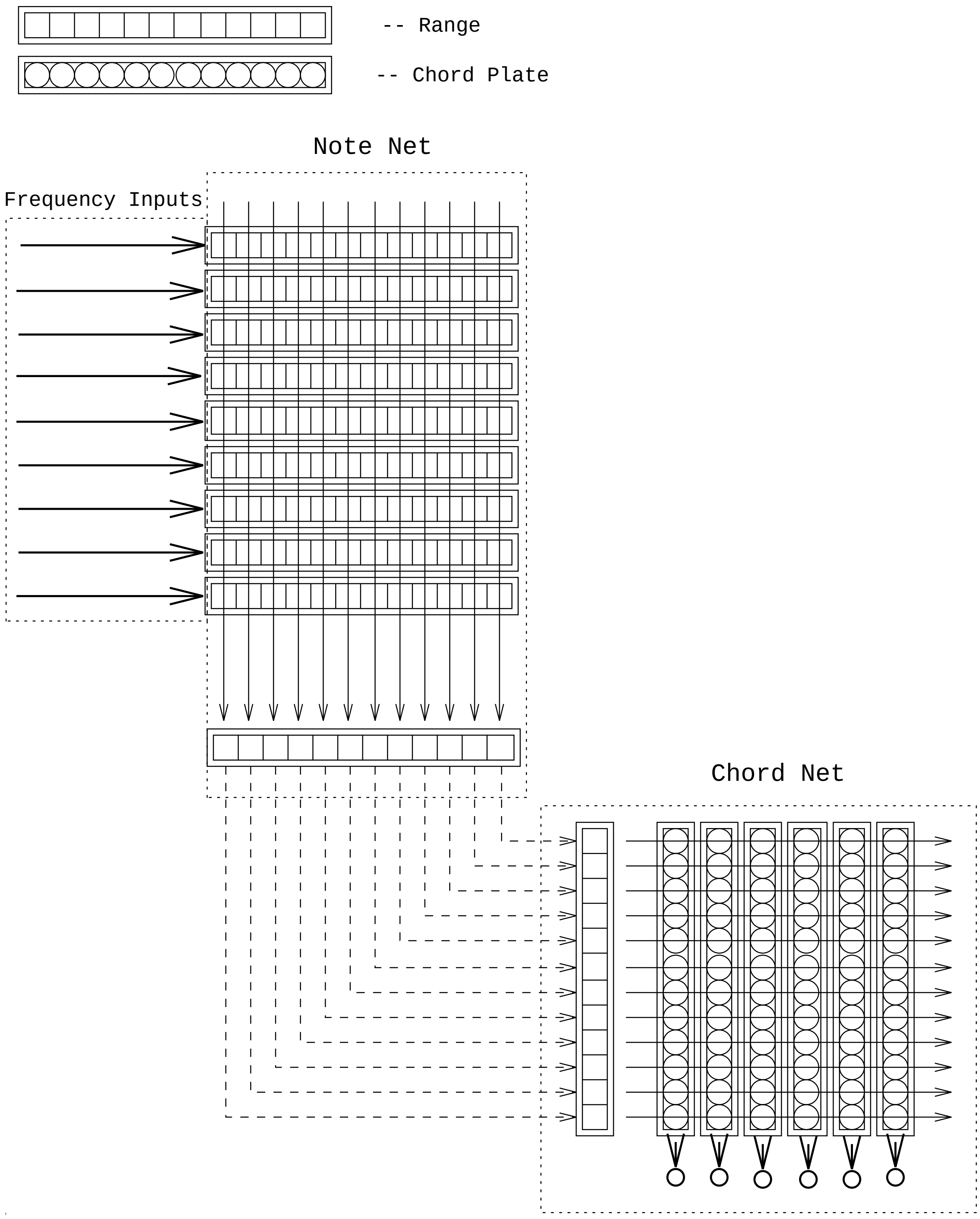

Figure 5–1: The Structures used in the Note and Chord Competences

These individual chord nets are called "Chord_plates" in the code because they are essentially templates. The weight for any pitch class present in the defined chord is one, the sum of all the weights in a Chord_plate is zero.

The chord robot prints out however many chords have weights greater than zero (or any other threshold set on the command line) in order of weight ranking along with the weights associated with each chord and the total weights for each pitch class of the input.

5.3 Development History and Analysis

CHORD was the second module written. It took about two weeks of effort from design through testing. The initial coding took one week, then the debugging took slightly over half a week due mostly to problems I had with the C/C++ library interface. Most of the interesting development history for the software is in the Design Motivation section above.

For testing the program, I initially used small files corresponding to single chord sections of songs I had edited out using the soundtool program. [Sun89] Once the program was working it occurred to me to try running the entire songs through the program. The result of this was to identify the key for the song.

Having CHORD actually made it easier to test some of the features of NOTE that I had not previously checked. I was interested in the significance of the weights reported for the pitch classes other than the winning one, since the NOTE net was only trained with respect to the winner. One straightforward way to research this was to check the result of playing in a single polyphonic instance rather than a monophonic melodic stream. CHORD successfully identified such chords.

Another test was to compare the output of CHORD if only the "winner" from a NOTE input was taken into account rather than the entire array. The output was similar for the small melodic input samples mentioned above, but substantially worse for picking out keys of the overall song. This demonstrates that for more complex input the retention of full information is better than early abstraction.

5.4 Further Work

- *Have the chord competence learn chord types.* Although, as stated above, I don't consider this necessary either to subsumption architecture or the engineering requirements of this project, it would certainly be an interesting exercise. The program would not only need to learn to recognise generalised chord types, but then also find their root. Alternately, the project as a whole could be rewritten to have the chords recognised in relation to the root at this level, instead of having that abstraction done at the Song level as I am proposing. My reasons for not doing this are discussed above in the analysis section, more about this also appears in the Further Work Chapter, chapter 10.

- *Make the chord recogniser smarter by altering weights* One could increase the probability of guessing the correct chord with less notes if one gave additional weight to the roots of the chords. Also, it may be that the performance of this module could be improved a bit by increasing the amount that non-chord members strike against the chord.

5.5 Conclusion

The CHORD module derives the chord from a set of input such as that produced by the NOTE competence. It identifies both by collecting and summing continuous input and by analyzing polyphonic input. The CHORD robot illustrates this process for any segment of song which has been saved into a file. When executed over an entire melody, its output is the melody's key.

Chapter 6

The Rhythmic Competences: Thresh, Beat, and Timed

The rhythmic competences operate as a whole to find the regular intervals at which chord changes are likely. The basic strategy is to do a form of time series prediction on the most likely time for the next occurrence of a note.

The lowest module, THRESH, finds note thresholds by looking for pitch stability. The next module, BEAT, times the intervals between thresholds to find a list of the most likely durations. The highest module, TIMED exploits this duration information and coordinates it with the THRESH output to produce signals in anticipation of significant intervals.

THRESH and TIMED are both reasonably straightforward finite state machines. BEAT is another simple neural net — a single-layer competitive net.

It should be noted that these are not the only competences in the Time layer of the overall architecture (figure 3–1, page 21). CHANGE is the module which actually subsumes CHORD to move it into time. The rhythmic competences could actually be seen as subsuming CHANGE.

6.1 Design Motivation

The primary inspiration for using this method of beat recognition and prediction came from the work of Peter Desain. In [Desain 92] he describes systems which seek to identify the exact rhythmic nature of a melody, essentially by allowing neighbouring notes to compete for their proportions of a measure. In the course of his research, Desain experimented with predicting rhythmic events. He found that, regardless of the rhythmic patterns that were being evaluated, the most probable anticipated thresholds tended to be the beats, with lesser peaks of probability at the underlying pulses. (see figure 6–1)

There were a number of other possible heuristics for finding the rhythmic features of the melody, but none seem as well suited to robustness requirement for a subsumption approach. Looking for loudness cues was the most obvious candidate, but it had several problems, such as the tendency for artists to "accent" unexpected positions in the time in order to add rhythmic interest, or the fact that some instruments are restricted with respect to volume changes. The deciding factor in not even trying this approach, however, was the fact that volume information was not available to the program. (see chapter 4)

Another possible method would have been to look for at the duration of the notes, because musicians tend to emphasise important beats in this manner. However, this approach seemed too complicated for our purposes, as it involves the establishment of a patterned norm *before* it can be applied. It also would have required too much precision for this system, as the end of a note is less distinct than the beginning.

The probabilistic approach based on general note durations, on the other hand, is extremely well suited to the needs of this system. For one thing, no individual notes are crucial. Longer values actually composed of several notes are probably equivalent to other possible notes in the system. At the same time, the extra emphasis given by musicians to main beats does make it more probable that they will be recognised and held.

Figure 6–1: Desain's Rhythmic Predictions[Desain 92].

6.2 Description

6.2.1 Thresh

THRESH is a very trivial finite state machine - it simply looks to see if a pitch has been "stable" for a certain interval (`THRESH_LAG`) after a period when it was not. Stable is defined as being true when none of the relative values of the pitch classes have changed more than another value, `THRESH_DIFF`. `THRESH_DIFF` has a default value but can be changed on the compile line for the actual thresh robot. The output is a boolean value for each Fourier frame, where "true" indicates a threshold.

The reason I called the stability interval "THRESH_*LAG*" is because, since this system is operating in real time, THRESH sends its positive response `THRESH_LAG` frames behind the actual onset of the note.

Defining stability in terms of all the pitch classes instead of just the winner strengthens the robustness of the module in two ways:

1. It copes with harmonic bouncing. The output of NOTE occasionally flips between a pitch class and the pitch class of its second harmonic. This does not necessarily involve a significant change in the output.
2. It copes with polyphony. If there is more than one note playing, the pitch that happens to dominate the output may not necessarily be the most rhythmically significant of the pitches.

Comparing always with the immediately previous input rather than the input from the beginning of the stability period allows for instruments whose intonation tends to change over the course of a single note.

6.2.2 Beat

The BEAT module consists primarily of a data object which is an array of weights. The module watches its input, a string of booleans presumably from THRESH, and counts the interval between the note thresholds. On a "true", it updates the array at the index value of the number of frames since the previous true value.

The array starts out with a length of one and a fixed weight. It is lengthened as required to represent the durations that will be added into it, but the total weight remains constant. When a duration is added, it is actually added as a normalised curve, so that neighbouring indices are also updated. The weight added to these frames is derived from the entire array, using the same kind of algorithm as used in the competitive net training in note (page 31). This allows for coping with time changes, as old high-points gradually flatten if they are not reinforced.

For output, BEAT returns an array of integers representing frame increments which is produced by sorting its indices with respect to their relative weights.

6.2.3 Timed

TIMED is also a finite state machine, though a much more complex one than THRESH. It builds a small array of what it believes to be the critical time intervals. Any interval considered by BEAT to be important is compared against this array. If it is within 20% of an existing value, it updates that value. Otherwise, it is sorted into the array.

TIMED also tries to coordinate these values with the output of THRESH. For whatever interval it is being polled for, TIMED watches for an interval between THRESH events which is equivalent to that distance. When it finds one, it sets its internal clock to synchronise with the actual rhythmic event having occurred `THRESH_LAG` frames before. Whenever this internal counter/clock reaches the value of the target interval, TIMED sends a true value as output and resets the clock. Thus its primary output is of the same type as THRESH, but much more regular and hopefully at an appropriate frame.

6.3 Development History and Analysis

The time competences were developed rather late in the project, not only after NOTE and CHORD, but after a period of writing up. Although these three competences are clearly strongly related, in fact CHANGED was developed between BEAT and TIMED for the simple reason that I couldn't think of a good way to do the time synchronising, so I moved on to the next bit. Since all of these modules are within the same layer (Time), this does not strictly violate Brooks' "independent debugging" principle. (Section 2.1.2.) Overall the development of all four time modules took approximately three weeks, at least half of which was developing algorithms rather than actually coding.

6.3.1 Thresh

The most time spent working on the THRESH module was spent looking for strategies that successfully located the beginning of a note. I ascertained that there really was no way to get volume information from the input files during this stage. Also, I spent some time examining the noise which occurs between notes, with a thought of building a module that could discriminate noise from actual music. But this also proved to be a less successful predictor.

One mistake I made was starting with the `THRESH_LAG` too short. I was too concerned with catching very short notes I saw in the input, and forgot that such values would be insignificant rhythmically. Consequently there was a good deal of spurious results.

Getting spurious results is of course much worse for the correctness of the beat prediction than missing out notes, so choosing a default `THRESH_DIFF` was simply a matter of reducing the value until the spurious results stopped. Actually, this was not totally trivial to judge, particularly on an instrument with many harmonics such as the mandolin. Something that looks from the note robot like a sequence of three harmonically related notes could qualify for "stable" under the most stringent THRESH standards. This is not a problem, however, because such a series really is as legitimate a rhythmic event as a solid note.

6.3.2 Beat

I originally intended to use a recurrent neural net for time series prediction for determining the beat thresholds, as I had heard this method was easier and more accurate than the standard statistical methods. However, I abandoned this course for several reasons:

- The statistical theory underlying such nets tends to be unsound. [Willshaw 92]
- Recent work by Brooks explicitly condemns complex, non-biologically tractable neural nets. [Brooks 91a] (See footnote page 47)
- I realised I would not have time to develop and train so complex a net myself and was reluctant to include more external software into the system.

I decided using standard statistical methodology would probably be faster running and easier to program. However, on examination the statistical methods did indeed seem even more objectionable than the nets. Consequently, I developed my own qualitative approach inspired by Desain's figures. (See figure 6–1.) The concept of this method is to produce similar looking curves based solely on the output of thresh. Compounding normal curves affect an otherwise straight line, building each other up to peaks. A constant weight strategy similar to competitive net weights follows change over time. Notice this approach doesn't exactly duplicate Desain's graphs — the pulses are not evenly tracked through the beat with lower probabilities than the actual beats as reported by Desain. Instead, the first pulse will often be the highest peak for any rapidly moving piece.

The largest problem with BEAT is that an important peak will generally be spread across several neighbouring indices. This is hardly a difficult problem; it simply requires a little smarter pattern recognition in routines seeking the peaks in the data.

6.3.3 Timed

TIMED was written and designed in considerable haste, and as such has two major flaws.

First, it does a very naive evaluation of the BEAT data — it looks only at the current *highest* peak. This strategy actually works fairly well, however, because of BEAT's time decay. The highest peak is likely to be the one most recently updated, if the current signal from thresh was actually in a statistically significant location. Thus TIMED does manage to stay updated fairly well.

The second problem is more serious. TIMED has built into it the assumption that the smallest unit it finds is the pulse, the next largest is the beat, and it only keeps track of three. Also, it only tries to synchronise for one of these time values, which is chosen externally. TIMED *should* keep track of both the size and the synchronicity of any distinct size it finds, and a higher level module could make any necessary decisions about which of these is more significant than another. I do not expect that one would need an overly informed, dedicated module such as CHORD to perform this task. Rather, it would be ideally handled in CHANGE, with competing CHORD modules being generated for each time unit, and their significance being judged by the success of their chord pattern production.

Having said this, TIMED still significantly improves on THRESH in providing reasonable places for time changes. It does successfully follow both gradual changes in time and even inconsistently timed performances. The timed robot gives much more reasonable output than the changed robot which it subsumes. However, it currently tends to result in time changes on too short of intervals, generally half beats.

6.4 Further Work

The main suggestion for further work on these rhythm modules would be to implement the changes suggested just above in the TIMED analysis. I actually suspect that trying the other threshold-finding or time-series-analysis possibilities mentioned would not prove either significant or interesting for the reasons mentioned in their respective sections.[1] More information on further work on the time section in general is mentioned in the next chapter on CHANGE.

6.5 Conclusion

The rhythmic perception competences work successfully as a unit to provide information crucial to finding more accurate chord structures more easily. They offer reasonable locations to break off and look for a new chord, and this results in much more key-compatible chords being produced as output from the CHANGE module.

The development history of these modules, along with the other time module CHANGE brings up an issue with the subsumption architecture approach, in that modules were not debugged and implemented totally independently of each other. This issue is discussed in the Analysis chapter (chapter 9). On the other hand, these modules do provide particularly clear examples of several subsumption issues:

- Not all modules subsume something, some just provide additional behaviours. This is true of THRESH and BEAT.
- the subsumption by TIMED of THRESH is a particularly nice example of adding a significant competence completely transparently to the affected module (CHANGE).

[1] With the possible exception that a threshold-finding function based on gain might be able to follow the conventional "count in" so that the robot could start on time with other musicians. Right now the count would have to be played on an instrument.

- TIMED, particularly in its proposed refinement, provides the best example implemented in the overall program of a task considered to be complex pattern recognition, but done in a non-symbolic way.

Chapter 7

The Change Competence

CHANGE subsumes CHORD to move it into real time. It does this by controlling its flow of input rather than just allowing it to absorb the entire melody. Actually, CHORD could be said to operate in real time on its own, but in practice its output rapidly stabilises to the chord which is the primary key of the melody. CHANGE makes the possible underlying chord changes visible by applying CHORD to smaller segments of the melody.

CHANGE operates by taking rhythm information as a cue to search for a possible chord change. At the threshold of a significant time interval, CHANGE starts a competing CHORD module from scratch, and then checks to see if it seems to be creating a stronger match to the melody than the prevailing theory coming into the time interval.

In the diagram of the change robot (figure 3.2.5), CHANGE appears to be extremely complex because of its interactions with the other units. It appears, in fact, to be in a situation of control. One should realise that intrinsically CHANGE is a simple finite state machine, as shown in figure 7.

It should be noted that CHANGE is the goal-producing module referred to in section 2.1.2 not only for its own robot, but for the timed and noise robots as well. Although these modules in some way subsume and improve on CHORD's

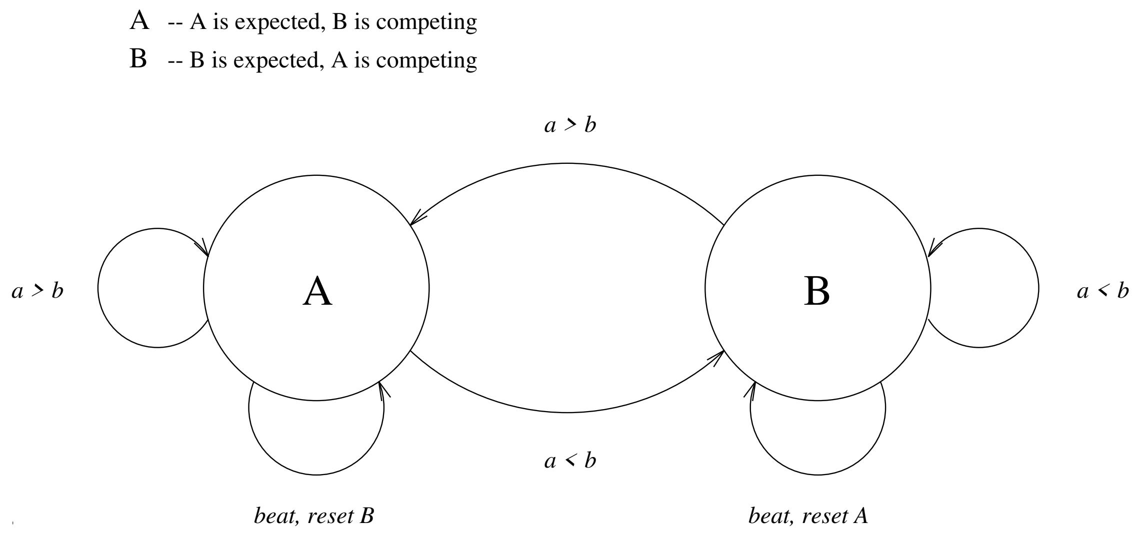

Figure 7–1: Change as a Finite State Machine.
The Output of the "Expected" Chord Becomes the Output of Change.

behaviour, they do not subsume its purpose in the overall robot: to find the chord changes.[1]

7.1 Design Motivation

The main design decision in the CHANGE module was to use more than one instantiation of CHORD. This is because there are two possible states at any given point in time —

1. a new chord is beginning, and the notes previously seen are irrelevant, or
2. the previous chord is continuing, and the notes previously seen can help identify it.

Notice we want to have as many notes as possible that belong to a single chord evaluated together, because fewer notes can leave ambiguities. For example, both

[1]Noise actually partly subsumes this goal by smoothing the chord output — see chapter 8.

A minor and C major contain pitch classes E and C. If either or both of these notes were the only pitches present, any decision from CHORD would be arbitrary.

Another consideration is that this module always reports a running output of its best theory. This may be early in the beat and thus severely underinformed, and the output may shift rapidly between several chords as a consequence. For the purposes of providing information for a higher level pattern matcher, only the best guess at the *end* of the time interval would need to be reported.[2] However, one of the stated goals of this project is to mimic a human accompanist, and so the entire "thought process" is available and printed via the change robot. This way, were the project in real time, the best guess chord could be played for some fraction of the beat, though this means a few mistakes would be made. One could imagine that a finished version of this robot might have a flag for performing either as a timid or an aggressive accompanist. The timid robot would not produce output until it thought it had a song it recognised, while the aggressive one played its best guesses from the beginning, and one could presumably hear its performance improve as the pattern recognition took control.

CHANGE actually had many other important design decisions, but many of these were unforeseen and took place over the course of its development. Consequently, these are detailed in the following section.

7.2 Description

CHANGE consists only of structures for keeping track of each of the chords it uses, and the functions for comparing them and controlling their input.

As input is generated by the NOTE module, the CHANGE module passes it unaltered to its chord modules, except when it is signaled by its rhythm module that there is a beat. In that case, CHANGE resets both its CHORDS' input. The leader is reset to a single instance of the chord it last reported; this is the CHORD

[2]The robot "noise" demonstrate this idea by taking the project out of real time. See Chapter 8.

called "expected". The other CHORD is reset to totally empty registers — this CHORD is called "potential" or "competing".

The change robot reports the output of both of these modules for every time frame, with whichever module is currently the strongest being reported in the first column. If this program were real time and auditory, it would be playing the chords in the first column in an attempt to accompany its input.

7.3 Development and Analysis

CHANGE was developed over a period of about five days, between the modules BEAT and TIMED. The change robot only uses THRESH for its time event signals, but CHANGE was written with the expectation that it would be subsumed by something more informed about likely chord thresholds.

7.3.1 The Chord Comparison Routines

The first and largest problem with CHANGE was finding a comparison routine for the two chords. My original plan was that if the competing chord seemed to be strengthening the resolve of its winning chord while the expected chord was being weakened, the competing chord would become chosen. That is

```
IF ( (expected.weight-of-winner / expected.total-weight) < it was before)
     AND
     (competing.weight-of-winner / competing.total-weight) > it was before) )
        THEN
        swap(competing, expected);
```

But I was concerned about the validity of these numbers, so I included another heuristic which compared simply the relative strength of the chords. I called the former strategy "gradation" and the latter "rating".

On the initial test, the gradation comparison was totally ineffective after a short ways into the song. The key chord came to dominate, and the competing chord never won. The rating system was similar, but occasionally there would be periods when chord changes did occur. For this reason I stuck with the rating comparison for the rest of development, although I still suspected gradation was more desirable. The reason for preferring gradation was that the absolute strength seemed too open to factors such as the number of notes within an interval - two or three notes defining a chord would be much stronger than many fast notes including some passing tones, especially since the space between notes tends to be interpreted by NOTE fairly randomly. But the overall trend for a time interval should be more reliable.

The problem with both comparison routines was that the expected CHORD gained too much weight momentum. As it was exposed to more frames, new input became less relevant, and unlikely to significantly affect the weights. The reason this affected the rating comparison less that the gradation was that occasionally an interval happened to contain notes that nearly completely defined a chord without contradiction. Since the expected CHORD would have been absorbing a large number of contradicting values, it would finally loose out and be reset at the next time event. Then neither CHORD would have much momentum, and the swapping would work for a period until one CHORD found a dominating chord again.

Once this problem had been resolved (see "Resetting the Expected Chord" below) I again examined the two comparison strategies. The results are the same about 80% of the time, but where there were differences I found the gradient strategy nearly always chose chords that were more consistent and believable.

7.3.2 Resetting the Expected Chord

The way I first recognised the momentum problem was by observing the actual values of the rating comparison through a debugger [GDB91]. I focussed on a time interval between THRESH signals where the output of NOTE clearly defined the competing chord and not the expected one, but the competing one didn't win. I saw that while the strength of the competing value grew consistently, the rating for the expected one never shrank significantly, and that for this particular interval there wasn't time for the competitor to overtake it.

This problem was solved by resetting the expected CHORD's value. When the competing CHORD is set to empty, the expected CHORD has its input array reset so that it appears it has been exposed to exactly one perfect instance of the chord that was previously winning. In this way, the expected CHORD is still biased to interpret any pitch which is appropriate to its chord as what it expects to be the correct chord. At the same time, competing chords win out when they have sufficient contrary evidence.

This approach also has the advantage of making CHANGE easier to subsume. I had from the beginning expected that at some point CHANGE would be subsumed by a pattern recognition module which would send more informed guesses about the upcoming chord than just that it was the same as the previous one. (See figure 11.2.) Previously, I had worried about how to make a suggested chord from a pattern matcher look like an existing chord in the expected CHORD's data array. The solution to the momentum problem also resolves this issue by making the expected CHORD's data array look like a suggested chord.

7.4 Further Work

- *Further refine the comparison routine.* The comparison routine is really the crucial element of this module. Further experiments should probably carried out to make sure that the gradient method is truly optimal.

- *Pay attention to more rhythmic intervals.* As the Analysis section of the Rhythm chapter explains (section 6.3.3), there are many possible time threshold intervals. Evaluating chord changes with respect to each of them would obviously necessitate more competing copies of CHORD, and possibly competing modules of CHANGE. Since CHANGE has the same output as CHORD, it is even possible to envision tree hierarchy of CHANGE modules, with CHORDS as the leaves. This idea needs more development.

7.5 Conclusion

That a song has a chord structure implies a melody is broken up into sequences of time-consecutive pitches, each of which has a single chord held as applying throughout its duration. If one interpreted a melody only as a string of notes, one would have an astronomic search problem looking for all the possible chords applied across all the possible sequences (of all possible lengths) of the melody. A musician trying to find a chord sequence knows that there are certain regular intervals of time at which a change is more likely to occur, and would not consider changes at points that were not on these cusps. CHANGE applies such time information to the CHORD competence, and evaluates whether the making a change at such a break seems to be a "good idea". In doing this successfully, it allows the creation of the first robots which actually suggest chord structures.

Chapter 8

Noise

Noise is not really a competence, it's just a robot. Its purpose of existence is to put the output of the highest level robot (timed) into auditory form, so that it can be more easily judged by people [1] not sufficiently trained in music to derive significance from the written pitch classes and chord names alone.

The noise robot translates the information from NOTE, TIMED and CHANGE into input for the csound program [CSO91]. It reports a note if its value has been consistent for at least two frames. This output is actually fairly confusing because the melody has been transposed entirely into a single octave, but if played simultaneously with the actual input the results can be compared.

There are two possible outputs for the chord information. One reports the output of the leading CHORD in the CHANGE module the same way as the melody is reported. The other reports the winning value of the chord at the end of a threshold interval as the chord for that entire interval. This is the information that the next higher level module would be using for pattern recognition, as it is the most informed guess for the value of that interval (see section 7.1). This is only possible in the noise robot because the robot operates outside of real time, so the chords can be set retroactively at the *end* of the beat. Thus the first version more closely represents the project, and reflects the way the final project would sound before it had found a pattern to follow. The second version is more useful

[1]Like the author...

for evaluation, however, because it demonstrates the “conclusions” drawn by the chord module, rather than its “thought process”.

Chapter 9

Analysis

The aim of this project was to attempt to implement a human competence, the ability to derive chord structure from melodic structure in real time, using a behaviour-based development approach, subsumption architecture. In this chapter I discuss the extent to which this project succeeded on each aspect of this objective. First I discuss the performance of the system from a musical perspective. Next I discuss how well the project conforms to subsumption architecture, analysing not only the project but also the approach. Then there is a commentary on whether this project and this approach actually model the way people perform this task. Finally, there is a brief discussion of the success of this project as an intelligent agent.

9.1 Performance as a Musician

The performance of the individual musical competences that make up the final program are analysed in their respective chapters. This section addresses the overall program.

The program was evaluated on the basis of its ascii output by three musicians with varying levels of knowledge of classical harmonic theory. It was evaluated on the basis of its auditory output by three musicians and three lay people. In both cases, one of the musicians was the author, and the evaluations were carried out fairly informally. Since the system was never brought into real time, all the evaluations were performed on several samples which were recorded live in the computer laboratories. The samples consisted of at least two songs each on the mandolin (which is the instrument the NOTE module's neural net was trained on), the Irish penny whistle, the Scottish small pipes, and the human voice.

The performance of the program varies widely depending on the input. For some samples it is considered quite plausible, where for others it seems random. When the outputs of all the modules are examined together, however, the results are generally considered reasonable, if sometimes naive. I believe then the sources of error can be reduced to three basic problems.

1. *The input is not successfully interpreted by the lowest level modules.* This is particularly a problem with NOTE. For example, it was completely incapable of picking out the pitch classes for the human voice input. Obviously chords generated for this input *were* truly random.

2. *The program fails to find the true beats.* If it works on too short of a time interval, the consequence is that the program is under-informed. Bizarre chord choices can be based on a single note. Evidence that this is a source of trouble can be found in two places:

   - performance is generally best on fast songs, where several notes are likely to fall within the TIMED event gaps, and
   - performance is best on instruments with polyphony, such as the mandolin, where often more than one string is ringing at a time, and the smallpipes, which have drones. In fact, one piece was recorded twice on the smallpipes, with and without drones. In this case it is very clear that the drones eliminated the most inappropriate chord choices, though at the same time they confused the THRESH module because of the continuity of the sound.

3. *The program lacks the higher level competences that would make it likely to repeat patterns.* This is another issue of the contribution of context. For example, most phrases end with one protracted incident of the note that is the root of the key, but this is directly preceded by a contrasting chord. With the next level of competence, the program would have access to the fact that the previous time this contrasting chord was played, it was followed by the root chord, and could be biased to recognise the root note as part of

> the root chord. But under the current program, the chord anticipated is the previous, contrasting chord, and the resultant chord can be quite incorrect. To a human observer, this is one of the most easy chords to anticipate, so this sort of performance seems particularly uninformed. We *expect* learning and consistency from a musician.

Despite these factors, on inputs compatible with its biases the performance of the program is quite impressive. This is especially true when compared with a human accompanist approaching a new song for the first time. It tends to use the correct chords about half the time, and to make changes where changes are required. "Incorrect" chords are seldom totally dissonant, but rather are *unexpected* because they fail to fit into a pattern. Time changes are compensated for successfully, even when they involve dropped beats. In general, the program's rating as a musician was "promising"

9.2 Performance as an example of Subsumption Architecture

As a structure for evaluating this project in terms of the subsumption architecture approach, I am using the list of criteria set out in section 2.1.2 of the Theory chapter. For each criteria, I discuss both how well this project conformed to it and evaluate its utility in terms of software development.

9.2.1 Independence in Engineering

> *Each layer should be engineered separately, and tested and debugged until flawless before proceeding.*

Each competence was developed, implemented, and tested as its own "robot". This approach was extremely useful in simplifying the task of putting together a complex system.

I cannot claim, however, that I left established modules entirely alone after completing them, or even that they were completely debugged. Most of the changes I made to already "complete" modules took the form of parameter adjustments. For example, while developing the rhythm competences (chapter 6) I did not recognise the problem with the `THRESH_LAG` variable in THRESH being too short until I had heard the output of the TIMED module. This kind of performance tuning does seem in contradiction with Brooks' recommendations[1], and is admittedly complicated when several modules are involved. On the other hand, it is a normal part of most real-time system development, and may pragmatically be unavoidable.

I feel this issue is actually related to the principles of using an unsimplified environment and expecting sliding performance degradation as the environment is less than ideal. Under these criteria one does not expect perfect test results. I personally found it very difficult to judge when a level's performance was then adequate without seeing how it interacted with higher layers. I did not have sufficient time to try to find, or even determine how to recognise, optimal performance.

A less frequent modification I made to earlier levels was to add functions for examining the outputs, or in the case of CHANGE affecting CHORD, suppressing the inputs in different ways than I had originally anticipated. Actually, I think this is a problem with a contrast in modularity between subsumption architecture and object oriented programming. In a robot, these additions would have taken the form of wires being attached, and could have been considered a part of the higher-level module. But within C++, it made sense for routines to be attached to the object they were reporting on. Consequently, I was changing the objects of the lower-level module although I was contributing to the higher-level one.

One advantage of doing subsumption in software instead of hardware was that when I did perform such modifications, I was able to recompile and test the lower-level module's robot. This way I could make certain I had not introduced errors into the lower levels. This is a luxury roboticists do not have, so perhaps sticking strictly to this principle is not as important for a software project as a hardware one.

[1] Though TIMED and THRESH *are* in the same "layer".

9.2.2 Components

> *A layer should be a fixed topology network of finite state machines possibly augmented with a few registers and timers. It sends and receives all messages over wires. It should be data driven, with no global data and no dynamic communication.*

The modules of this project are fairly well "augmented finite state machines" as specified, although their data objects are more complex than "a few registers" would imply. On a functional level, however, even the most complex of these objects is essentially a two dimensional array of floating point numbers. Of course, there are no wires in the program, so the restriction on dynamic communication in terms of handshaking is meaningless. On the other hand, the modules are all data driven - their internal states are only affected by their input. And there is no global data — data objects are all associated with modules and can only be altered by their own module.

9.2.3 Interaction

> *Layers may only interact with other (lower) layers via their input and output wires.*

The competence that most seriously challenges Brooks' principles of modularity is CHANGE (see chapter 7). The question here is whether CHANGE has too much influence over CHORD. After all, CHANGE is able to reset CHORD's input, and even send it artificial chords.

Brooks does state that input messages can not only be intercepted, but replaced with new messages.[Brooks 91b] One could imagine then, in an embodied robot, a module that normally summed its input, but a certain input signal caused it to flush its buffers. A subsuming function could then send this clearing signal, possibly following it by an artificial signal. In this case the functionality of the CHANGE module could be exactly duplicated. However, because CHANGE is actually software, different functions are used for sending conventional and artificial input. This was a choice made for good coding style over explicit modeling of a robotic system.

Similarly, the fact that there are multiple instantiations of the CHORD module under CHANGE may make it look as if CHORD has become a subroutine to CHANGE. In fact, in section 7.4, I suggest that CHORD modules should be created dynamically as needed by CHANGE to entertain multiple theories about the current chord. This situation is unlikely to be duplicable in hardware in the near future. However, none of Brook's explicit warnings have been violated. The instantiations of CHORD are oblivious to the higher modules, and they run continuously once they are created, not just on demand. The fact that only one of the CHORDS outputs is visible from the external is a consequence of perfectly legitimate output suppression on the part of CHANGE.

Thus I would say that CHANGE conforms to the requirements of subsumption architecture. Some of the form has been modified to take advantage of the media, but the design and fundamental operations are consistent with the standard.

In general, I think this principle has been well respected, and could continue to be followed through the higher modules. I had originally expected this to be a problem, because within music so many of the tasks of comprehension seem mutually dependent, I anticipated a situation where it would be difficult to establish a hierarchy.

In fact, this principle strikes me as one of the strengths of subsumption architecture as a development tool. Thinking of previously developed modules only in terms of fixed input and output makes it much clearer how they should be interfaced with. Thinking of a current module in terms of the input and output that higher modules will be able to see helps clarify both the requirements of the module, and whether in fact the module should be reclassified into smaller or larger competences.

9.2.4 Test Environments

There should be no simplified test environments.

I conformed to this principle entirely. In fact, there is some doubt as to whether performance could be improved if I had used more "simplified" environments, such as artificially generated training sets for the NOTE competence, or music written to conform to the standards of traditional harmonics, such as Mozart. These options are discussed in the Further Work chapter (chapter 10).

9.2.5 Representation

> *There should be no representation. No variables instantiated, no rules matched, and no choices made.*

There is a question whether it even makes sense to talk about music, particularly chord structure, in a system that is supposed to have no representation. For example in section 5.1.1 of the Chord chapter, I state that chords are being treated as a language with which the program communicates its knowledge to the human users, as a translation of the information from the NOTE module's output. This implies that chords are *symbols* — that they are a form of *representation.*

My interpretation of this issue is that the problem is not with symbols themselves, but symbolic reasoning. Brooks claims his methodology can be applied to any problem —

> I think the new approach can be extended to cover the whole story, both with regards to building intelligent systems and to understanding human intelligence.[Brooks 91a]

But there can be no other way to communicate arbitrary goals or results to a system except via symbols.

An example of this from Brooks' own lab is Toto, a navigating robot built by Maja Mataric. Brooks quotes Mataric's work as an example of a valid subsumption based robot with a map. What makes it good subsumption architecture is that the map is "decentralised, non-manipulable, [has] no central control which builds, maintains, or uses the map — rather the map is itself an active structure which does the computations necessary for any path planning the robot needs to do."[Brooks 91a]

But Mataric expects that higher level modules will eventually give goals to her map; in the current implementation users do this via buttons on Toto's top.[Mataric 90] These goals take the form of landmark types, so clearly a) these landmarks, which are used as the nodes of the map she builds, are defined prior to the creation of the map and in fact external to the robot and b) they are used for communication. Mataric describes such message passing as necessary, but states that no theory about the organisation of such messages has yet been developed.[Mataric 92]

To return to the original statement of criteria above, it can certainly be said that this program is not a production system, there are "no rules matched" to preconditions. However, it could be argued that two of the largest outstanding issues of the program are violations. The problem with picking the beat out of the various time intervals in TIMED might be called "variable instantiation", and the problem of finding a comparison routine for CHANGE might be considered "choosing". The further work section of these modules' respective chapters suggests ways of doing away with these problems, but these essentially involve deferring these decisions to higher modules. Of course, this strategy works very successfully in the interface between NOTE and CHORD. As stated in the Analysis section of chapter 5 the maintaining of the information on all the pitch classes from NOTE actually improves the performance of CHORD.

9.3 Performance as a Model of Human Cognition

As mentioned at the beginning of the Theoretical Background chapter (chapter 2) one of the appeals of distributed intelligence paradigms is that they present an alternative model of human cognition. Of course, any real evaluation of human cognition is beyond the scope of this project. This section is designed simply to highlight a few of the observations made on this subject over the course of the project.

On the level of the basic competences, it was noted that NOTE not only recognised notes successfully as people do, but made some of the same kinds of errors, particularly confusing a note with its second harmonic. Similarly, CHORD was observed to make some of the same errors that naive humans do, particularly when presented with unusual modalities.

At the accompaniment level, a subject was brought in specifically to observe how a human reacted to the same input and task as the project had been evaluated on. The problem with a human subject is that it is essentially impossible to find one capable of creating reliable chord accompaniment, but with anything approaching the contextual naivete of the system. For most songs, the subject constrained her choice of chords to three or four most likely to be found in folk music, and would strum each after a chord change had occurred to hear which seemed to fit best. However, in one instance she was presented with a piece in an unusual modality, and with an unpredictable time structure. In this case she began rapidly trying a larger number of chords with nearly every pitch change in a way that sounded extremely like the performance of the project on the same piece.

Another particularly interesting behaviour by the subject was on a song that she recognised; she chose a totally different approach to chord fitting. Her level of knowledge of the songs was sufficient that she could anticipate where a chord change would be required. She then picked a chord slightly *in advance* to the expected change, then analyzed whether the notes played fit into her chord. This was very like the strategy developed for CHANGE.

Another point this strategy makes is that she could not possibly have been following the classical sequential harmonisation approach, although she was trained and proficient in it. Classical harmonisation involves transcribing the melody intact, then following a system of rules to derive a chord structure and deduce the changes. The subject clearly was learning this song in a different sequence. Her familiarity with the song was sufficient that she knew an element of the chord structure — where the changes occurred, but she was not sufficiently familiar with the melody to predict which chords should be played.

Further evidence of this point was found when I asked her to derive a polished version of the chords for one of the songs she was unfamiliar with, so that I could use it as an objective basis of comparison with the output of the program. To speed the process she chose to work out the final order by humming the melody herself rather than playing with the recording, and I noticed that she was humming the melody incorrectly. I recorded her playing her final version of the chords along with her humming the melody herself. The chord structure she derived is completely compatible with the original melody, but her melody was radically different. This is even stronger evidence that the competence actually used by musicians for this task differs from the sequential proscribed methodology of music theory.

Rodney Brooks asserts that *any* intelligent task can be performed using a system following the rules of subsumption architecture.[Brooks 91a] For me, this begs the question of what consciousness, an essentially sequential intellectual activity, is for. Brooks' answer is that "thought and consciousness are epiphenomena of the process of being in the world." My personal sense is that human consciousness is too elaborately evolved to be just an epiphenomenon. Neural networks research has shown that training distributed systems to do tasks that need to be in a specific sequential order is extremely difficult. I suspect that a sequential thought process is useful for such training. As mentioned above, the subject accompanist seemed to be using sequential reasoning for short cuts in determining the chord structure, such as limiting chord choice. However, once a chord structure is learned, a person who can play the song perfectly is often incapable of correctly reporting the chords they play if asked to recite them without duplicating the behaviour.

9.4 Performance as a Creature

I have used the term "robot" throughout this dissertation to refer to the compiled programs that implement the modules. I used this term partly to make the clear distinction between the things that generate the output used to evaluate the modules, and the modules themselves which often produced far different output. But technically, the term can be sincerely applied to these programs if one believes this definition from the opening of the Proceedings of the First International Symposium on Robotics Research[Brady & Paul 84][Hallam 91a] —

> Robotics is the intelligent connection of perception to action.

So to what extent does this project conform to Brook's description of a Creature? (see page 15)

> *Creatures need to be able to respond quickly and appropriately to changes in their environment.*

The robot is able to react to changes in time and key, because it is constantly monitoring and reacting to its environment.

> *Creatures should be robust with respect to their environment.*

The robot can cope with changes in time, pitch, key, and to some extent instrumentation. As what it hears becomes further from its trained input, its performance gradually degrades.

> *A creature should be capable of pursuing multiple goals.*

Different modules in the robot are constantly, simultaneously watching for new information about pitch, chord, and time.

> *A creature should do something, it should have a "purpose of being."*

The purpose of this creature is to create chords in accompaniment of its input, as this is the highest level module implemented in this project.

Chapter 10

Further Work

This is a fairly large scale project, which is not nearly complete in fulfilling all its goals and possibilities. Also, it applies a relatively new approach, subsumption architecture to a totally different domain, software development. Consequently, the potential for further work is extensive.

This chapter is broken into three parts. The first discusses potential changes and extensions to the project without adding additional competences. The second discusses potential additional levels of competence to this project. The third proposes a few examples of other projects that could be developed using similar behaviour-based strategies to the software domain.

10.1 Within the Scope of This Project

Each of the modules developed was essentially a mini-project in itself. Because I was most interested in studying the interaction of modules under subsumption architecture, and given that I had limited time, I tended to stop experimenting with a module as soon as the results looked "good enough" for me to continue on. As a result, nearly all of the modules could be further refined, and specific suggestions for this are in the Further Work sections of the chapters dedicated to describing the modules.

However, there are two significant changes that could be made that have a more global effect.

- *Move the project into real time.* This project would be more useful as both a model and a tool if it were in real time. Further, it would be far easier to evaluate and debug. Brooks considers being "situated", that is, actually a part of the environment one is reacting to, as a necessary precondition for intelligence.

  There are two ways to achieve real time: either create a more sophisticated (probably recurrent) net for evaluating the sound input without the benefit of the Fourier transformation, or else find a Fourier transformer that operates in real time. The latter would probably be easier, especially since the next version of csound is supposed to have this capability. The former might actually be more interesting, and would have the advantage of having more information directly available to the program. The partially supervised training method described in section 4.3.4 might be useful here.

  Notice that there would be some redesign involved in the main functions of the robot programs, because in a real-time system one would not sample the *next* input, one would sample the *current* input.

- *Move the system onto a parallel architecture.* Not only should the modules really be running independently in parallel, but the majority of the operations performed in module execution involve arrays and could be done in parallel. If the program were written in parallel it would be a better model of its own functionality, and this is nearly always a good idea in software design. And of course, it would run much faster.

10.2 Extending the Project

- *Add the Structure layer — unanticipated pattern recognition.* This layer would look for and anticipate patterns in the chord structures derived from CHANGE. If the suggested modifications were carried out, this would probably include choosing the time intervals in which patterns seem to occur. This would differ from the chord module in that there would not be templates to choose from.

- *Add the Song layer — learning.* At this layer one would almost certainly have some modules for abstracting the relative chords instead of using the absolute ones. But the most interesting thing would be the idea of "learning", building a repertoire, particularly if it could still be done within subsumption. My current best theory for this would be to build a structure not unlike the mapping module in [Mataric 90]. Since many songs, particularly folk songs, have similar structures, one could envisage developing a net with the nodes being relative chords, and the songs being paths through the net. This module would observe the output of the Structure layer to find the chords which seem to be forming patterns. It would subsume the Structure layer's impact on the CHANGE module, because it would be more informed with respect to when patterns are going to change.

10.3 Using the Project — Musical Cognition

Supporters of behaviour-based architectures often state that their systems are more developmentally/ethologically plausible and better explain human cognition than traditional, logic based, linear approaches. These claims could be analysed in projects that did more rigorous comparisons between this project and human performance than I had time to do. Possible research areas include:

- *Comparing the output of this system, the performance of actual folk musicians, and a traditional, rule-based system.*

- *Examining the errors made by naive musicians at these tasks, and how they change as expertise is gained.* Duplicating or predicting the mistakes made by people is one of the classic tests of an information processing theory. Finding any similarities between the stages of development of musicians and some of the competence levels of this program would be particularly interesting.

10.4 Other Applications

I believe that the subsumption architecture approach, modified to software, is an extremely promising AI technique, which could be applied in many areas. Some suggestions:

- *Face recognition* There is a great advantage to applying subsumption architecture to basic cognitive tasks like this: one can use the psychological data about human development to determine the order of implementation of the modules. For example, the first module for face recognition would be establishing the outline of the head, the next would locate the eyes from this information. This project could also grow along the lines of recognising emotional expressions from the interactions of the modules for the different parts of the face.

- *Monitoring software, for example in medical or industrial applications.* Low level modules could observe and react to every sensor input. As the seriousness of a situation is often dependent on the relation between a number of factors, so the output of the program could be dependent on the similarly modeled interactions of the modules.

- *Computer Operating Systems* As cited in section 2.2, distributed approaches are often used now for planning and scheduling. I think subsumption architecture could probably be applied directly to such tasks, particularly something with discreet, well defined possible operations, such as an operating system. This project would be particularly interesting if this highly parallel architecture were applied to multi-cpu systems. The technique discussed in the Change chapter (chapter 7) of creating modules on demand would be particularly useful for running systems on transputer systems where variable numbers of processors may be assigned to specific processes.

- *Oral Communication* Thinking about applying these techniques to the area of natural language is a bit daunting, but there are many aspects of human communication which could be addressed, such as deriving emotional content from inflection. As suggested above under Face Recognition, modeling the development of children's capacities may be the best way to approach the entire issue of language.

Chapter 11

Conclusions

11.1 Summary of the Project's Goals

The aim of this work has been to build a software project using a behaviour-based, robotics approach called "subsumption architecture" to mimic a human cognitive ability, deriving the chord structure for a song from its melodic structure. This was to be done in real-time, using live performance on conventional, acoustic instruments as input. The primary purpose of this work is to examine the possibility and utility of applying subsumption architecture to a large software project on a domain considered to be a "higher" cognitive function.

11.2 Description of the Project

The project as it now stands consists of six independent modules, each of which embodies a behaviour or "competence".

1. NOTE which derives pitch class from the auditory input.

2. CHORD which derives a chord from an input of pitch class information.

3. THRESH which finds the threshold in time for a new note.

4. BEAT which finds the most likely times for an occurrence of a THRESH event.

5. TIMED which uses information from BEAT and THRESH to anticipate the next rhythmic event.

6. CHANGE which uses this rhythmic information and CHORD to find likely locations for chord changes.

These six modules compose three layers of competency; Note, Chord, and Time; each of which subsumes but does not replace the operation of the previous layer in order to add its particular goals to the overall output of the robot. In order to create a competent accompanist, I anticipate the need for two more layers, Structure which would bias the output from the time layer towards patterns that are repeating, and Song, which would anticipate changes in the patterns Structure was finding. For a diagram of the interaction of the implemented modules and the anticipated layers, see figure 11.2.

Due to difficulties with signal processing of the input, the programs are not actually in real time, but the processing they do assumes that they are; moving the project into real time should not be a problem should the signal processing issues be resolved.

11.3 Results

11.3.1 In Terms of Output

The program has difficulty with some instrumentations and with slow moving songs with few notes per beat. However, given favourable input, each of the modules performs its task, though not with complete reliability, to a sufficient standard that a) musicians consider its output to be reasonable and intelligent and b) that the modules above it can use its output to operate to the same standard. Many of the mistakes which are made are understandable given the "naivete" of the program as it now stands, and correspond to mistakes made by musicians in a similar state of ignorance.

11.3.2 As a Software Engineering Model

The behaviour-based strategy proved to be a valid and useful method of decomposing the task of creating the chord structure producing program. In particular, subsumption architecture provided guidelines that made structuring the interactions between these modules straight-forward and provided a way to manage the complexity of a growing program. Under subsumption, one is only concerned with the workings of one competence at a time.

Some modifications were made to the standard for subsumption architecture delineated by Rodney Brooks in [Brooks 91b] in porting the theory to software. The most significant of these were:

- The modules were implemented in an object oriented language (C++) rather than in hardware.

- More complex data structures were used both within modules and for input and output. These were still reasonably simple structures by software standards, consisting mostly of one and two dimensional arrays of floating point numbers.

- Communication between modules was not constrained to wires, or even models of wires. Rather it was achieved by using trivial functions associated with the module's objects. Clarity of the codes intent was chosen over clarity of the constraints placed on input and output.

- Arbitrary numbers of copies of modules can be created dynamically to cope with new data situations as they arise. This adds to the robustness of the system in unpredictable situations.

11.4 Conclusion

The music robot constructed is a functional and interesting artifact of Artificial Intelligence. I believe this project can be extended to create a competent mechanical accompanist, which could be used not only for entertainment, but for studying the behaviour-based model of human cognition.

The subsumption architecture approach was used successfully in creating this robot. There were some problems in the decomposition of the task, particularly in evaluating the performance of behaviours prior to the development of the higher levels that depend on them. Nevertheless, its strengths in organising incremental development, managing program complexity and exploiting parallel architectures make it a promising methodology for many software development tasks.

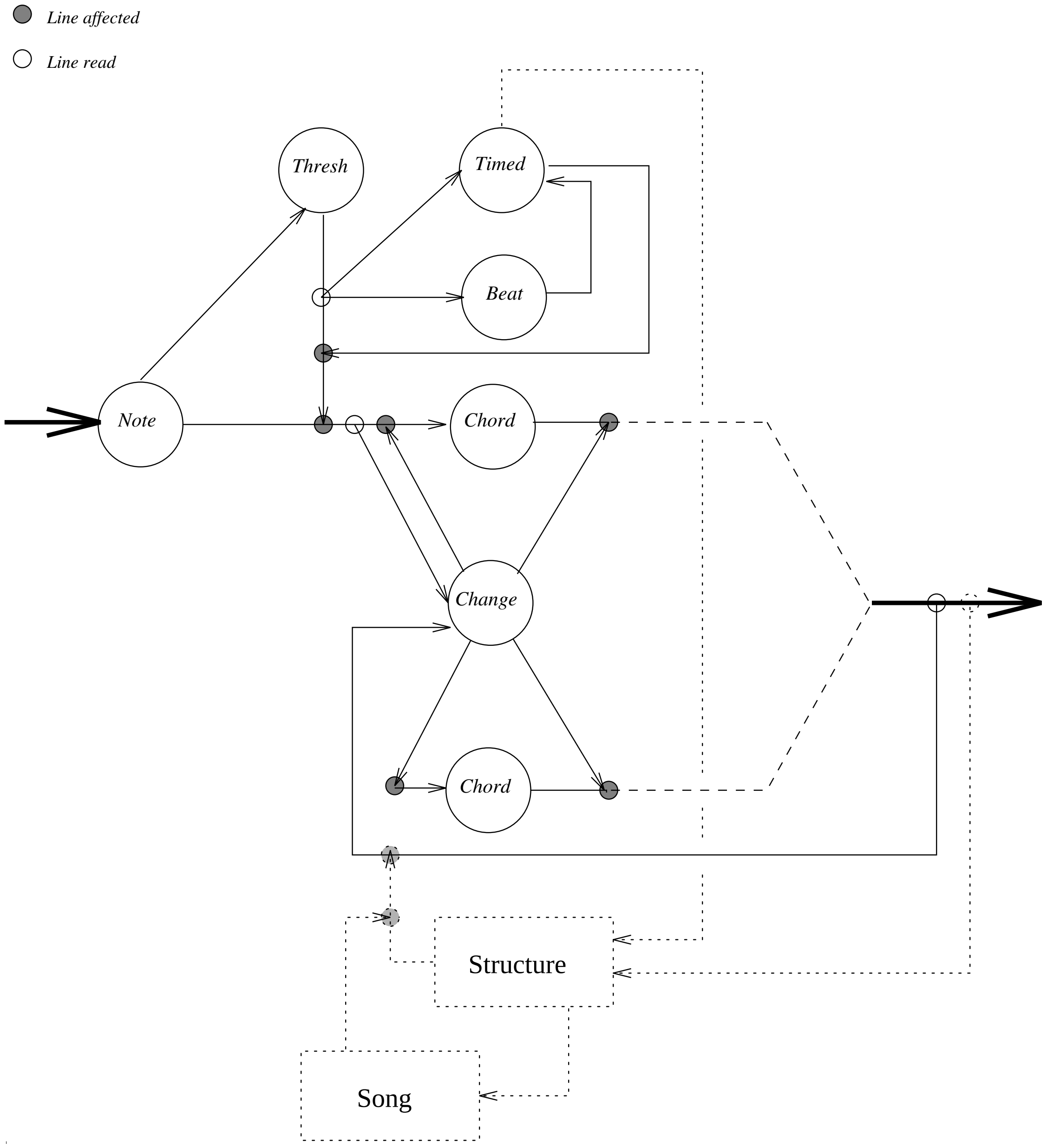

Figure 11–1: The Expected Final Robot.

The rectangles represent layers, as in figure 3–1, the circles modules.

Bibliography

[Brady & Paul 84] Michael Brady and Richard Paul, editors. *Robotics Research: The First International Symposium.* The MIT Press, Cambridge MA, 1984.

[Brooks 89] R. A. Brooks. A robot that walks : Emergent behaviours from a carefully evolved network. A.I. Memo 1091, MIT, Cambridge, Massachusetts, February 1989.

[Brooks 91a] R. A. Brooks. Intelligence without reason. A.I. Memo 1293, MIT, Cambridge, Massachusetts, April 1991.

[Brooks 91b] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[Chater 92] Nick Chater. Neural networks: Motivation from psychology. Lecture notes from the Neural Networks MSc module, Department of Cognitive Science, 1992.

[CSO91] *The Csound Reference Manual.* Cambridge, MA, 1991.

[Dannenburg & Mont-Reynaud 87] R. Dannenburg and B. Mont-Reynaud. Following an improvisation in real time. In *Proceedings of the ICMC*, pages 241–248, 1987.

[Dannenburg & Mukaino 88] R. Dannenburg and H. Mukaino. New techniques for enhanced quality of computer accompaniment. In *Proceedings of the ICMC*, pages 241–249, 1988.

[Dennett & Kinsbourne 90] D. Dennett and M. Kinsbourne. Time and the observer: the where and when of consciousness in the brain. Technical report, Tufts University, 1990.

[Desain 92] P. Desain. A (de)composable theory of rhythm perception. to appear in Music Perception, 1992.

[Durfee 88] Edmund H. Durfee. *Coordination of Distributed Problem Solvers*. Kluwer Academic Publishers, Boston, MA, 1988.

[GDB91] *Using GDB: A Guide to the GNU Source-Level Debugger*. Cambridge, MA, 1991.

[Gomes 92] C. P. Gomes. *Achieving Global Choerence By Exploiting Conflict: A Distributed Framework for Job Shop Scheduling*. Unpublished PhD thesis, Department of Artificial Intelligence, 1992.

[Hallam 91a] John Hallam. Autonomous robots: from dream to reality. Teaching paper, University of Edinburgh, 1991.

[Hallam 91b] John Hallam. Intelligent sensing and control. Lecture notes (revised from Tim Smithers, 1990), 1991.

[J. Hertz & Palmer 91] A. Krogh J. Hertz and R. G. Palmer. *Introduction to the Theory of Neural Computa-*

tion. Addison-Wesley Publishing Company, Redwood City, CA, 1991.

[Longuet-Higgins 87] H. C. Longuet-Higgins. The perception of music. In M. Boden, editor, *Mental Processes : Studies in Cognitive Science*, chapter 13, pages 169–187. MIT Press, Cambridge, Massachusetts, 1987.

[Maes 90] P. Maes. *Designing Autonomous Agents : Theory and Practice from Biology to Engineering and back*. MIT Press, Cambridge, Massachusetts, 1990.

[Malcolm *et al.* 89] C. Malcolm, T. Smithers, and J. Hallam. An emerging paradigm in robot architecture. In *Proceedings of IAS*, volume 2, Amsterdam, Netherlands, 1989.

[Man 92] R. F. Man. Subsumption architecture-based real-time multitasking kernel for programming autonomous robots. Distributed over the Newsnet, 1992.

[Mataric 90] M. J. Mataric. A distributed model for mobile robot environment-learning and navigation. Lab report, MIT, Cambridge, Massachusetts, May 1990.

[Mataric 92] M. J. Mataric. Personal communication, 1992.

[Minsky 85] M. Minsky. *The Society of Mind*. Simon and Schuster Inc., New York, NY, 1985.

[Roads 80] C. Roads. Interview with Marvin Minsky. *Computer Music Journal*, 4(3):25–39, Autumn 1980.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[Sun89] Sun Microsystems, Inc. *soundtool*, 1989.

[T. Matsushima & Ohteru 85] I. Sonomoto K. Kanamori A. Uesugi Y. Nimura S. Hashimoto T. Matsushima, T. Harada and S. Ohteru. Automated recognition system for musical score – the vision system of wabot-2. *Bulletin of Science and Engineering Research Laboratory*, 112:25–52, 1985.

[Todd & Loy 91] P. M. Todd and D. G. Loy, editors. *Music and Connectionism.* MIT Press, Cambridge, MA, 1991. Based on two special issues of the Computer Music Journal.

[Willshaw 92] D. Willshaw. Personal communication, 1992.