PART ONE

CHAPTER 1 — INTRODUCTION

1.1. Introduction

The close relationship between the disciplines of music and mathematics has been explored consistently throughout the history of Western music. An early commentator on this relationship was Pythagoras in the 6th century BC. In the teachings of Pythagoras and his followers:

> the understanding of numbers was thought to be the key to the understanding of the whole spiritual and physical universe, so the system of musical sounds and rhythms, being ordered by numbers, was conceived as exemplifying the harmony of the cosmos and corresponding to it (Grout 1980, p.6).

In the Middle Ages, music was considered as a branch of mathematics, one of four branches comprising the Quadrivium: arithmetic, geometry, astronomy and music (Grout 1980, p.24). From within this period of Western music there are many examples of compositional formalism, that is, the systematic ordering and organisation of musical parameters, examples of the embodiment of proportion and abstract numerical relationships (Dodge and Bahn 1986, p.185). In the 20th Century, composers continue 'the custom of borrowing systems of organisation from outside the musical realm' (Burns 1994, p.2). This is most evident in the works of computer music composers who have, since the mid-1950s, programmed computers with various mathematical and scientific processes for the purposes of generating musical works.

Since the introduction of commercially available music software programs in the mid-1980s, non-computer programmer composers are no longer, perforce, compelled to master programming languages to achieve musical results. To make programs commercially viable, contemporary programmers are incorporating a broad range of algorithms based on scientific or mathematical principles, thus allowing composers access to such principles without the need for programming, and often without the need to know programming languages.

Four example works are presented within this study, works composed specifically to demonstrate a range of extra-musical principles applied with composition algorithms. The four example works span the three primary types of computer music environments: electro-acoustic music; electronic music; and computer-assisted composition of acoustic music. The works are: *Study for Triangles,* for three triangles and computer controlled triangle samples, an electro-acoustic work developed with a software program entitled *M*; *Étude in Memoriam Allan Dagg,* for computer-generated tape, an electronic work developed with the software program *Jam Factory*; *Descendant Lines*, for two piano accordions, an acoustic work developed with the software program *Symbolic Composer*; and *When Cinderella's Monkey Comes Here I'll Feed Him*, an acoustic work developed with the software program *Phrase Garden*. This program was developed specifically for this study using the algorithm development program *MAX*.

1.2. Rationale

Algorithms used in music composed in conjunction with computers are generally developed by composers with computer programming experience as practical solutions to composition problems arising within their own musical works, or are developed in order for the composer to explore musical possibilities of extra-musical principles. Until the advent of the micro-computer in the early 1980s, this composer/programmer paradigm represented the primary induction method of new algorithms into the community of composers concerned with producing music with computers. With the current generation of fast and economically viable personal computing systems, such solutions are made available to composers other than the original developers, increasingly appearing in commercially available software programs as standard composition tools. As discussed in Chapter Two, there is a paucity of studies pertaining to the applications of commercially available algorithms, a paucity that calls for a study providing a detailed examination of approaches to the use of algorithms within the context of commercially available software programs.

The *MAX* program provides a middle-ground in the paradigm, presenting composers with the ability to assemble algorithms in a visually-based programming environment,

without the need to learn traditional programming languages such as C or LISP. This study demonstrates the manner in which composers without traditional programming experience can employ *MAX* to develop algorithms suited to their own compositional styles.

1.3. The Study and Main Elements

The purpose of this study is to examine algorithms within automated composition software programs and to provide descriptions of the application of algorithms in the four example works. The examinations focus on both the historical background of algorithms and technical descriptions of each algorithm, while the applications of algorithms are detailed through the functions each algorithm carries out within the context of an example work.

The study is broadly divided into two main sections. The first section details the application of algorithms available in the commercial software programs of *M*, *Jam Factory* and *Symbolic Composer*. The second section details the development and application of new algorithms using the development program *MAX*. The two sections of the study are divided into five separate parts, the first section containing Parts One and Two, the second, Parts Three to Five.

Part One of the study provides definitions of terms related to the field of automated composition, details the existing literature on automated composition using historical surveys, and provides overviews of the commercially available software used in the study. Part Two details the applications of algorithms in the compositions written with the software programs *M*, *Jam Factory* and *Symbolic Composer*.

In the second section, Part Three details the personal compositional style used in the works composed with *M* and *Symbolic Composer*, along with brief analyses of two further instrumental works composed without computer software. These analyses serve to demonstrate incompatibilities between composition processes provided in commercially available software and those used in the compositional style. Part Three serves as a background to the substantial Part Four, in which the development of the *Phrase Garden* program is detailed with reference to the compositional style, and details an example work

composed with the *Phrase Garden* program. Part Five presents conclusions drawn from the examinations carried out in the study.

In the parts of the study involving the *Symbolic Composer* and *MAX* programs, LISP *functions* and *definitions* within *Symbolic Composer*, and *objects* within *MAX* are, for clarity, differentiated from the main text font with the default fonts of the programs. In *Symbolic Composer* the font used is `Courier`, and in *MAX* the font is Geneva.

1.4. Limitations of the Study

The software programs employed in this study were chosen to illustrate a broad range of algorithms developed for control over pitch and rhythmic structures in music. The study is limited to software systems developed on the Apple Macintosh platform, one of the earliest personal computing platforms to provide a graphic user interface (GUI), an interface ideally suited to on-screen control of musical parameters within the computer composition environment. While graphic user interfaces or GUIs have been subsequently incorporated on various computing platforms, the Macintosh platform, in initially providing a graphic interface, has had developed for it a range of composition software programs that have become significant in the computer music community.

In the field of automated composition there is a diverse range of software offering the composer a wide range of algorithms. In adopting a software program, the composer embarks on a learning curve to facilitate the use of algorithms inherent within a program, and while some programs have moderate curves, others have steep curves involving an investment of many hours. For this study a limit of five software programs was decided on, to facilitate an effective study of algorithms with learning curves appropriate for the use of interactive and algorithmic software, the two primary types of software in the field of automated composition. The initial two example works in the study are developed with interactive software, while the third work uses algorithmic software, reflecting the short learning curves required for the chosen interactive programs and a substantially steeper curve required for an algorithmic program. The fourth example work is developed with the

interactive *Phrase Garden* program, this program itself developed with the fifth program used in this study, *MAX*.

In the development of the example works, a minimal hardware/software configuration was employed so that the study would be relevant to as broad an audience as possible within the computer composition community. Such a configuration forms the core of systems used in the development of interactive and algorithmic works, the specific requirements of which are detailed in Chapter Three of the study.

A further delimitation of the study is its primary focus on deep-level compositional structures of pitch and rhythm as opposed to the high-level area of timbre. In the examples, the area of timbre is seldom developed or controlled algorithmically. Rather, the examples are either performed with acoustic instruments, or synthesised and sampled timbres are used in performances activated with the MIDI (Musical Instrument Digital Interface) protocol (Loy, 1985).

The commercially available software programs used in the study fall into two main categories. *M* and *Jam Factory* are both *non-development* programs in which the user only has access to algorithms supplied with the programs. *MAX* and *Symbolic Composer* are both *development* programs in which the user does have access to the underlying programming languages (C and LISP respectively) and can develop new algorithms within the software programs. Within this study, *Symbolic Composer* was used as a non-development program, primarily to illustrate the wide variety of algorithms supplied with the program, whilst *MAX* was used as a platform for the development of new algorithms. Software versions were current at the time of the initial use of each software program throughout the development of the study. Software versions used were *M* version 2.2, *Jam Factory* version 2.01, *Symbolic Composer* version 2.32 and *MAX* version 2.5.

1.5. Definition of Terms

1.5.1. Algorithm

The term algorithm is a derivative of the term *algorism*, used to describe the rules of calculating with Arabic numerals (Knuth 1973, p.1). The term *algorithm* describes a set of

rules or sequence of operations, an explicit, finite procedure for accomplishing a task (Roads 1985c, p.xv). Knuth (1973, pp.4-6) gives the following criteria for determining an algorithm:

Finiteness	-	an algorithm must always terminate after a finite number of steps.
Definiteness	-	Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
Input	-	An algorithm has zero or more inputs, i.e. quantities which are given to it initially before it begins.
Output	-	An algorithm has zero or more outputs, i.e. quantities which have a specified relation to the inputs.
Effectiveness	-	The algorithm must be sufficiently basic that [it] can in principle be done exactly and in a finite length of time.

Until the 1950s, the term algorithm was traditionally associated with Euclid's algorithm for finding the greatest common divisor of two integers, but since that time has been used for a wide variety of purposes (Guttman 1977, p.135), and in a broad sense can theoretically apply to any sequence of actions taken to achieve a goal. As an example, a set of travel directions qualifies as an algorithm: it is finite, terminating with the arrival at a destination, each step is precisely defined, it has inputs (steps to reach the destination), and outputs (the completion of each step), and is effective (the destination can be reached within a finite length of time). In a strict sense, however, the term is normally associated with computational processes for mathematical problem solving. In relation to computer music programs, algorithms are finite computational procedures that can be applied at any level of the compositional hierarchy via a computer programming language. Algorithms can range in function from the production of a single tone, through to descriptions of entire pieces or entire genres of pieces (Yavelow 1992, p.834). Throughout this study, the strict sense of the term algorithm applies. In each of the example works the algorithms described are those that apply computational procedures to the development of musical parameters at various levels of the musical hierarchy, primarily the parameters of pitch and rhythm.

1.5.2. Algorithm classifications

Compositional algorithms fall into five main categories: stochastic; rule-based; chaos-based; grammar-based; and Artificial Intelligence based. The term stochastic is literally synonymous with the term random, however where random implies a short term

process in which outcomes are unpredictable and patternless, stochastic processes 'typically reflect a concern for the large scale distributions of outcomes' (Ames 1987, p.185). The remaining four categories are broadly classified as deterministic. As opposed to stochastic processes, deterministic processes are those in which outcomes are known or predictable. Rule-based algorithms are confined to one or more steps that simply perform a task:

> An elementary example of a rule-based process would centre around a series of tests, or rules, through which the program progresses. These steps are usually constructed in such a way that the product of the step leads to the next new step (Burns 1994, p.19).

Chaos-based algorithms draw on the scientific inquiry of chaos theory which 'presents a universe that is deterministic, obeying the fundamental physical laws, but with a predisposition for disorder, complexity and unpredictability.' (Hall 1991, p.8). Grammar-based algorithms, derived from the study of linguistics, contain sets of rules determining how the whole of a statement relates to its parts (its phrases, clauses etc.) (Ames 1987 p.185). Artificial Intelligence (AI) systems 'are those in which the computer algorithms are able to "learn" which solutions are retainable/useable by a series of comparisons with previously-stated material' (Burns 1994, p.22).

1.5.3. Automated composition

The term automated composition broadly refers to composition methodologies in which computers and computer programs 'undertake decisions affecting the content of a musical composition' (Ames 1987, p.185). Over a 30 year history two primary types of automated composition have evolved, referred to as interactive and algorithmic composition.

1.5.4. Real-time/Interactive composition

A real-time computer music system is one in which an event sequence is processed by a computer and heard in the same time-space by a listener or audience, a system in which 'behaviour is dependent upon time' (Dannenberg 1989, p.225). By analogy the term *real-time* can apply to an acoustic instrument performance, or the simple playing of a compact disc recording, both of which fit the above criteria. However, the term is normally associated

with performances employing electronic and computer music systems. Real-time music systems can be broadly classified into two primary areas, simple real-time systems and interactive systems. Simple real-time systems are computer-based performances wherein pre-programmed event sequences are controlled by the computer and performed with an output device such as a synthesiser, sampler or sound module.

'Interactive systems are those whose behaviour changes in response to musical input' (Rowe 1993, p.1): systems in which pre-programmed material is subject to real-time manipulation by a live performer/composer, and systems in which the computer generates musical materials in response to a live performer. Algorithms within interactive systems are programmed to respond to human input, transforming, in real-time, pre-programmed musical materials according to inherent computational procedures pre-defined for individual algorithms.

1.5.5. Algorithmic composition

The term algorithmic composition refers to a method of composition whereby musical procedures are expressed within a computer program as algorithms, in the form of alpha-numeric data. Once the procedures are established for all parameters of a composition, the computer compiles the data, converting alpha-numeric information into musically relevant data that is transferable via the MIDI protocol to a notation- or performance-based format (Yavelow 1992, p.864). Algorithmic composition as such represents a non-real-time system of composition, a system in which the processing of computer event sequences is not heard in the same time-space by a listener or audience, but after the computer has compiled and converted alpha-numeric data into a musically recognisable format. Throughout this study the term algorithmic composition represents a non-real-time procedural method of composition wherein abstract specifications for musical process are pre-defined before compilation to a final output.

1.6. Summary

This study originates from a paucity of studies pertaining to the applications of algorithms using software from the field of automated composition, and examines the application of algorithms to deep-level compositional structures of pitch and rhythm. Four example works are presented, each developed within the confines of a minimal hardware/software configuration. The examples include three works developed with interactive composition software and one work developed with algorithmic composition software, each example illustrating various applications of compositional algorithms. The settings for the example works span the three primary types of computer music environments: electro-acoustic music; electronic music; and computer-assisted composition of acoustic music.

CHAPTER 2 — LITERATURE REVIEWS

2.1. Historical Overviews

2.1.1. Introduction

Literature pertaining to the broad historical development of automated composition is primarily documented in three surveys: Lejaren Hiller's 'Music composed with computers — a historical survey' (Hiller 1970); Charles Ames' 'Automated composition in retrospect: 1956-1986' (Ames 1987, pp.169-85); and Gareth Loy's 'Composing with computers — a survey' (Loy 1989a, pp.291-396). A more recent doctoral dissertation by Kristine Burns provides a comprehensive historical account of algorithms in music composition, tracing algorithm developments from 1957 to 1993 (Burns 1994). Brief overviews of automated composition systems are documented in various monograph introductions and chapters (Cope 1991; Dodge & Jerse 1985; Moore 1990; Roads 1985c).

In the surveys of Ames and Loy the main focus is on automated composition systems developed by individual composer/programmers for the production of their own musical works, systems that were generally developed prior to the widespread availability of micro-computers and prior to releases of commercial, MIDI-based software systems developed to run on micro-computers. The survey by Loy, whilst mentioning the 'impressive attributes' of MIDI-based commercially available systems (Loy 1989a, pp.324-5) remains focused on the composer/programmer paradigm. The Burns dissertation expands upon the paradigm to include 'universal algorithmic composition programs' based on the MIDI protocol. The focus of the Burns dissertation is historical, pertaining to algorithms available within commercial automated composition programs. Particular applications of such algorithms in the context of musical examples are not included in the scope of the document. Similarly, further literature pertaining to commercially available software is limited to reviews of individual programs such as those provided by Yavelow (Yavelow 1987, 1992), these reviews detailing algorithms within programs, but without a provision of specific examples of the applications of algorithms. This broad focus on commercially available software programs and algorithms results in the paucity of studies pertaining to the applications of algorithms that this study seeks to address. Periodic comprehensive literature searches have

searches have been undertaken during the course of this study, the last in September 1997, and have substantiated the lack of literature pertaining to the applications of algorithms in commercially available composition software.

The following brief historical overviews of algorithmic and interactive composition summarise the cited surveys and include literature from the composer/programmer paradigm pertaining to the development of automated composition. The overviews provide a background to the development of commercially available automated composition software programs, leading directly to the specific interactive and algorithmic composition programs that are used in the current study.

2.1.2. Algorithmic composition

Initial instances of computer composition with algorithms arose in the United States in the 1950s, most importantly in the work of Lejaren Hiller and Leonard Isaacson (Hiller 1970, pp.42-96). In Hiller's *Illiac Suite* (1957), algorithms developed by Hiller and Isaacson generated musical parameters with stochastic processes, the results printed in alpha-numeric code and subsequently transcribed for string quartet (Dodge & Jerse 1985, p.292). Following the *Illiac Suite*, Hiller in association with Robert Baker, developed *Musicomp*, a programming system employing stochastic processes similar to those used for note generation in the *Illiac Suite*, but also providing libraries of algorithms that allowed, for example, the manipulation of pitch series via the normal serial transforms of retrogrades and inversions, and various permutation techniques (Loy 1989a, p.368). Hiller and Baker's *Computer Cantata* (1962) is based on stochastic procedures derived from probability theory, a theory in which the uncertainty of isolated random incidents is circumvented by predicting distributions of outcomes from many similar incidents taken together. Probability numbers indicate the likelihood that a random incident will produce a specified outcome, and range from zero (impossibility) to unity (certainty), with a continuum of gradations in between (Ames 1988, p.186). Probabilities in the *Computer Cantata* are derived from Charles Ives' *Three Places in New England* and are imposed upon pitch, duration, dynamics, notes versus rests and playing styles (Ames 1988, p.176). In addition to probabilities, serial procedures

derived from Pierre Boulez's *Structures* for two pianos are implemented within the work (Dodge & Jerse 1985, p.293-4). *Musicomp*, and the system's algorithm library, was further developed during the 1960s by Hiller, Herbert Brün and John Myhill. Brün's *Soniferous Loops* (1964) for example, included a rest/play algorithm in which the activity of each instrumental part was controlled by percentages of rest/play probabilities: in the opening measures of the piece the flute part rests 26% of the time, while unpitched percussion rests 68% of the time (Ames 1987, p.172).

While the above American systems were developed to generate and realise particular musical works, more general systems were developing in Europe, geared towards the development of numerous and varied works (Ames 1987, p.173). Iannis Xenakis developed his 1962 stochastic music program for the purposes of employing statistical procedures to deduce musical works, to use the computer as an aid to composition, as opposed to the computer simulating composition processes, the latter concept a focus of Hiller and Isaacson in the *Illiac Suite*. While these two approaches to the use of the computer are markedly different, the algorithms employed were similar: on each continent the use of stochastic processes formed the basis of the compositional style. In the case of Xenakis, the composer controls means and variances in probability distributions to shape a work. In Hiller's case, the composer controls rule sets that are applied to random sequences (Loy 1989a, p.311).

Another European composer, Gottfried Michael Koenig, in 1964 began to develop his *Project I* and *Project II*, systems that again employed combinations of stochastic and serial procedures (Loy 1989a, p.314). Koenig's systems furthered the use of algorithms to include control over various musical parameters. Within them, elements could be brought together to form a musical context (Roads 1985a p.570). *Project I* and *Project II* were used by Koenig to develop numerous compositions throughout the 1960s, his *Übung für Klavier* composed in 1969 demonstrating various applications of algorithms to pitch, duration, dynamics and register (Ames 1987, pp.176-7; Laske 1981, p.57).

Developments in algorithmic composition in the 1970s led to applications of deterministic procedures, initially with musical applications of formal grammars, primarily adopted from Noam Chomsky's *phrase structure grammars*:

> This approach enables a composer to describe musical forms economically, by first providing a general *archetype* (or *axiom*) of the form and by further listing a set of *productions* for deriving details from generalities. The full power of such an approach can be attained only if the productions are capable of acting upon their own results (i.e. capable of recursion) (Ames 1987, p.180).

Grammars relate to recursive computer programming techniques wherein a computational procedure is defined in terms of itself (Jones 1989, p.185). Initial theoretical work associated with grammars arose in the work of Curtis Roads (Roads 1985a, p.403), Steven Holtzman (Holtzman 1981, p.51) and Kevin Jones (Jones 1981, p.45). Roads, Holtzman and Jones each applied various Chomskian classifications of productions, while an initial application of grammars occurs in Charles Ames' *Crystals* (1980) in which recursive grammars are applied within the context of *Gestalt* hierarchies, derived from Gestalt psychology, a precursor to cognitive psychology (Ames 1982, pp.46-64).

Further developments with recursive techniques arose in the early 1980s with the application of procedures derived from chaos theory. A branch of chaos theory is fractal geometry, categorised in the work of the French mathematician Benois Mandelbrot. Fractals exhibit the phenomenon of self-similarity. That is, characteristics of a fractal at a macro-structural level are reflected in similar characteristics at a micro-structural level. Such self-similarity is classified as statistical, 'where characteristic features are perceivable as being generally the same, but not identical', or literal, 'where the detail precisely models the shape characteristics of the overall structure' (Jones 1989, p.180). Initial works employing fractal procedures include Larry Austin's *Canadian Coastlines* (1981) which employs statistical self-similarity (Dodge & Jerse 1985, pp.301-3), and Charles Dodge's *Profile* (1984), which employs literal self-similarity (Dodge 1988, pp.10-14).

Ongoing developments in computer science have been incorporated into algorithmic composition programs. An important example is the application of Artificial Intelligence techniques which rely on recursive programming to 'implement decision making processes that employ searches to discriminate actively between multiple options' (Ames 1987, p.181). Several branches of Artificial Intelligence have been developed and have had subsequent applications in music systems. Examples of these are Connectionist systems that 'employ

"brain-style" computation, capitalising on the emergent power of a large collection of individually simple interconnected processors operating and co-operating in a parallel distributed fashion' (Todd & Loy 1991, p.ix), and Neural Networks in which 'knowledge is represented by the connection strengths between processing elements in a network, and the mutual reinforcement or inhibition of elements by other elements' (Loy 1989b, p.25).

While a detailed survey of composition algorithms is not within the scope of this study, two primary areas of development in algorithmic composition are related in the above overview. In essence, these areas include algorithms that are based on stochastic processes, in the systems of Hiller, Xenakis and Koenig, and deterministic processes, in the formal grammars described by Roads, Holtzmann and Jones, the fractal processes used by Austin and Ames, and AI techniques. In practice there are many convergences of these two primary areas, for example stochastic processes are applied in the formal grammars described by Jones (Jones 1981, pp.45-61), and in the fractal methodologies used by Austin (Dodge & Jerse 1985, pp.301-03).

The implementation of the MIDI specification in the early 1980s led to a realisation by commercial companies of algorithmic compositional possibilities inherent in the combination of MIDI-linked computers and digital synthesis hardware (Loy 1989a, p.372). Commercial MIDI-based software programs began to appear in the mid-1980s. These incorporated both stochastic and deterministic processes as models to represent musical data. The commercial programs were based on general purpose programming languages rather than the music specific software employed within the composer/programmer paradigm, programs such as Hiller's *Musicomp* and Koenig's *Projects I* and *II*. Examples of commercially available algorithmic software systems developed in the 1980s include Charles Ames' *Compose* written in the C programming language (Ames 1992, p.57), and Tonality Systems' *Symbolic Composer*, written in the LISP programming language (Sica 1994, p.107). Such systems offer the composer a diverse range of both stochastic and deterministic algorithms, implementing and extending an extensive range of algorithmic compositional techniques developed since the 1950s.

2.1.3. Interactive composition

Interactive systems were originally developed for real-time manipulation of timbre during the late 1960s. These were 'hybrid' systems in which 'the computer provided control signals for analog sound synthesis equipment' (Risset 1985, p.130). An example of such a system is *Groove* (Generating Real-time Operations On Voltage-controlled Equipment) developed by Max Mathews and Richard Moore (Mathews & Moore 1970, p.715-21). The concept of real-time processing of high-level musical structures progressed through the 1970s in conjunction with the obsolescence of batch-oriented computers using punched cards, the development of on-line computers with increased computational speeds, and 'a desire to facilitate rapid interaction between composer and sound' (Ames 1987, p.178). While initially devoted to high-level sound synthesis structures, real-time systems began to be applied, in the mid-1970s, to deep-level musical structures in the works of Emmanuel Ghent and Laurie Spiegel. In the works of both composers *Groove* was used to manipulate pitch and rhythmic motives in real-time, applying inversions, retrogrades, augmentations, diminutions and transpositions to Spiegel's pre-composed melodic and rhythmic patterns, and to Ghent's pitch and rhythmic patterns originally produced with random functions (Ames 1987, pp.178-9)

By the late 1970s, digital synthesis techniques were available and attention was increasingly focused on deep-level structures. In American composer Joel Chadabe's *Solo* (1978) both timbre and duration were controlled by the position of the performer's hands in relation to proximity-sensitive antennae. His *Rhythms* (1980) took the interactive process further, with the performer, via the computer keyboard, controlling transpositions, melodic variation, rhythms and orchestration (Chadabe 1984, pp.22-7).

Both *Solo* and *Rhythms* were conceived following the development of *Play*, a 1977 program by Chadabe and Roger Meyers for control of an analog synthesiser. This program was described as functioning in two stages:

> (1) a design stage, where the composer designs a specific composition process, using any of the modules available in the program, and (2) an operation stage, where the composer interacts with the playback according to the design (Chadabe & Meyers 1977, p.12).

This design/operation paradigm forms the basis of Chadabe's interactive composing, a process that involves firstly, the programming of the computer to respond in real-time to performer actions and generate materials not controlled by the performer, and secondly, the simultaneous composition and performance resulting from human interaction with the program.

Following his success with initial experiments in interactive systems, Chadabe founded Intelligent Music, a company providing a commercial outlet for interactive composition software employing the MIDI protocol. The first two software systems, *M* and *Jam Factory,* were released in 1986. In both systems the design/operation paradigm is employed. The difference between these programs and earlier interactive systems is that pre-programmed algorithms 'provide a wide and flexible range of controls that give the user the opportunity to personalise the results produced by the program' (Zicarelli 1987, p.13).

In addition to *M* and *Jam Factory,* numerous commercially available interactive software systems using pre-programmed algorithms have been released. Laurie Spiegel's *Music Mouse* represents one of the earliest examples, in which the mouse of the computer (i.e. the spatial location input device, usually with one or more on-board switches, which on the Apple Macintosh, is attached to the Analog Data Bus) 'controls the location of points of intersection of the x y axis, which are used to control pitch and tempo respectively' (Loy 1989a, p.372). A more recent example is Miller Puckette's *MAX*, released commercially in 1990. Within this program algorithms are realised by manipulating graphic objects on screen and making connections between them (Rowe 1993, p.25). As shown in Chapter Eight of this study, *MAX* provides an array of pre-programmed algorithmic objects, enabling manipulation of musical structures in real-time. In all of these interactive systems, as in algorithmic composition systems, a broad range of both stochastic and deterministic algorithms are implemented, providing the composer with a diverse array of composition tools.

2.2. Software Overviews

2.2.1. Introduction

The specific software systems employed in this study are *M*, *Jam Factory, Symbolic Composer* and *Phrase Garden*, developed with *MAX*. A software overview of *Phrase Garden* is not given in this chapter as an extensive discussion of the program is provided in Chapter Eight. Primary literature sources pertaining to the remaining software systems are the manuals supplied with each software program, and descriptive articles from software authors relating their system configurations and details of their software. Secondary sources are in the form of software reviews, monograph overviews and surveys. Both primary and secondary literature sources provide detailed information on inherent system features and the following overviews summarise the existing literature on each of the software systems.

2.2.2. *M*

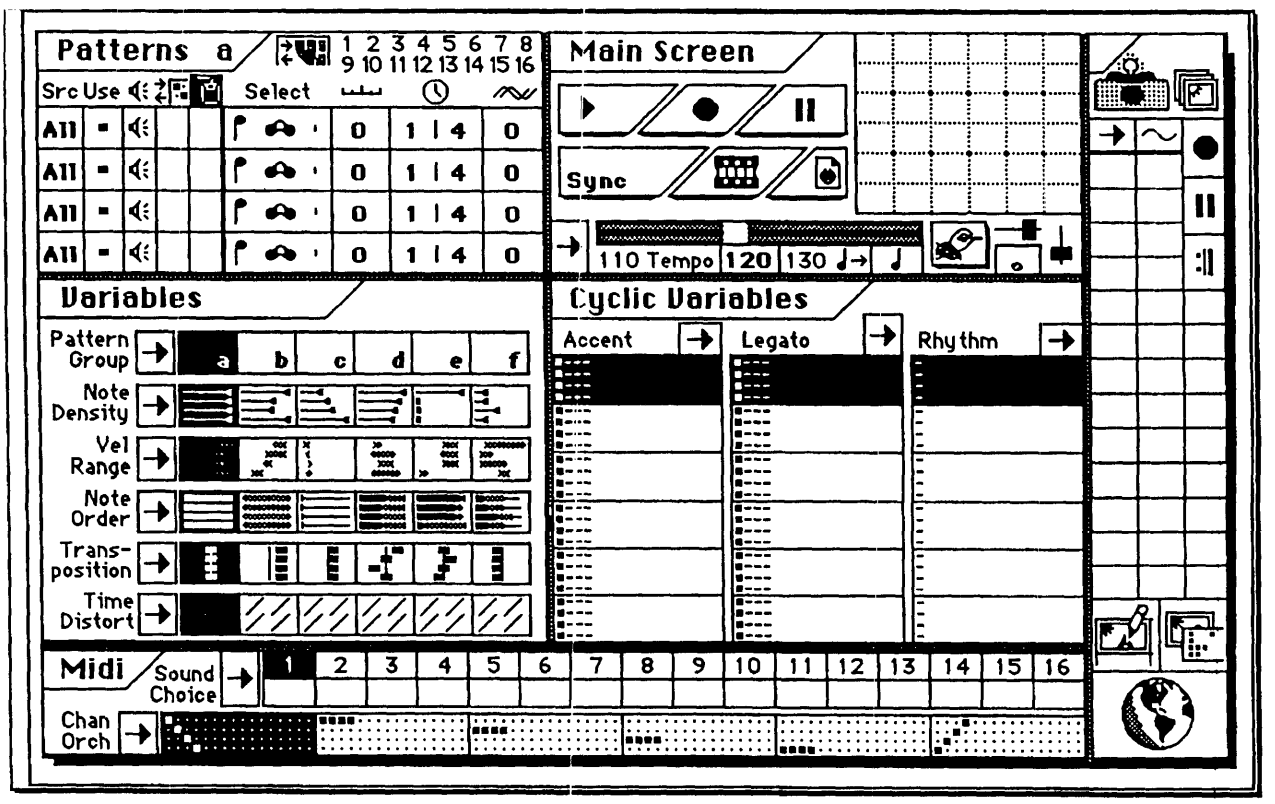
M was developed in 1986 by David Zicarelli, Joel Chadabe, John Offenhartz and Antony Widoff (Zicarelli 1987, p.13). Following on from Chadabe's previous work, the program is based on the design/operation paradigm developed in the *Play* program (Chadabe & Meyers 1977, p.12). In *M* however, the process is elaborated, providing an ability to modify the design of a work so that the distinction between the design and operation stages becomes blurred. 'At certain times the user is in the design stage, at others the user is in the performance stage' (Zicarelli 1987, p.13).

The procedure in using *M* is to enter musical ideas such as melodies, chords and rhythms, and then work with the program to transform those ideas into finished compositions, the real-time environment of the program allowing the composer to 'explore musical ideas quickly, efficiently and with immediate satisfaction' (Chadabe & Zicarelli 1986b, p.29). In an initial design stage:

> the user determines the basic musical material for four separate parts — e.g., melodic patterns and/or pitch distribution (input can be in step or real-time, monophonic or polyphonic), rhythmic patterns, accent patterns, articulation patterns, intensity ranges, orchestration (program numbers as well as MIDI channel assignments), transposition configurations, proportions of pattern-variation methods (original order/permutation/random), and tempo range (Yavelow 1987, p.221).

The four separate parts are collectively labelled as a pattern group, the software allowing six individual pattern groups that can each contain four different pitch patterns. Similarly, six separate positions contain alternative configurations for each of the remaining parameters (Zicarelli 1987, p.19). Figure 2.1 shows the main screen of *M*, the six configurations for each variable are shown in miniature on the main screen, while enlarged versions of the configurations (as shown in Figures 2.2-2.4) are used for editing configurations.

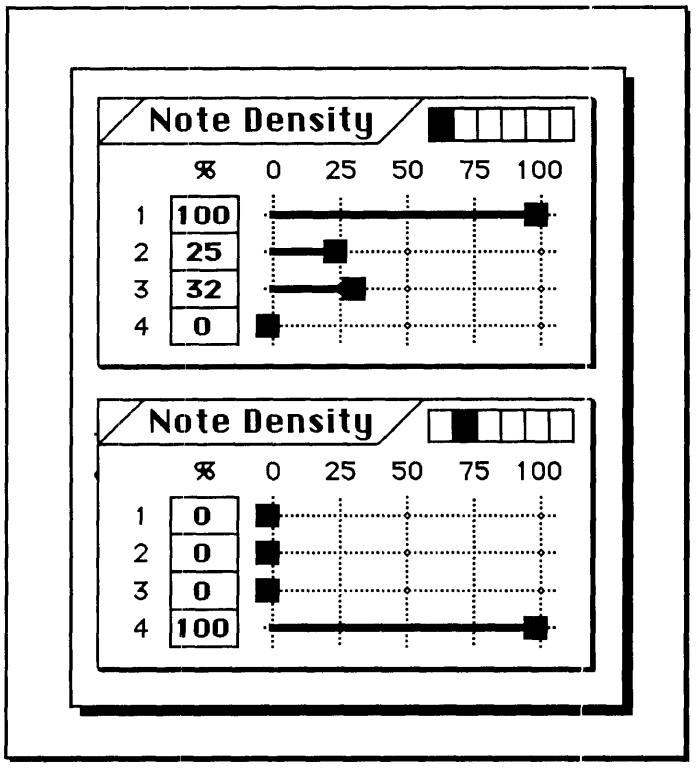
Figure 2.1 *M,* Main screen



In the operation stage the settings for each of the above parameters are alterable in real-time, thus effecting an immediately audible design alteration and blurring the distinction between design and operation stages. As an example, four pitch patterns can be established in the four parts of a pattern group and subject to two different settings of *M's* Note-density algorithm (or variable) 'which controls the percentage of random skips that occur in playing a pattern' (Zicarelli 1987, p.20). In Figure 2.2:

position 1 can be interpreted as follows: pattern 1 plays all the time, patterns 2 and 3 play very rarely, and pattern 4 doesn't play at all. Thus switching to position 1 of the note-density variable causes pattern 1 to play much more than all the other patterns. Switching to position 2 effectively cuts off all patterns except pattern 4 (Zicarelli 1987, p.20).

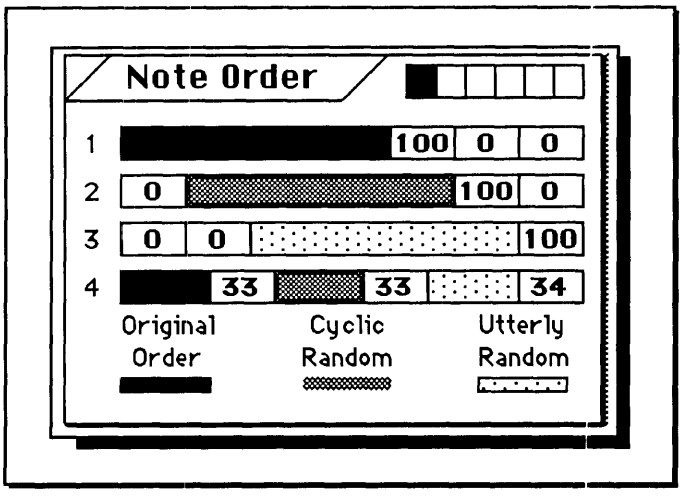
Figure 2.2 *M*, Note-density settings (Zicarelli 1987, p.21)



The variables that affect musical output in *M* are classified within the program as either variables or cyclic variables, and are implemented as stochastic and/or rule-based deterministic algorithms. The Note-density algorithm in Figure 2.2 is classified as a variable and implements a stochastic algorithm that, when the value is less than 100%, makes a probabilistic decision whether to skip a note based on the percentage (Zicarelli 1987, p.20). The Note-order variable shown in Figure 2.3 implements both deterministic and stochastic processes. The pitch pattern in Part 1 is deterministic, continually presenting the pattern assigned to the part in its original order. The pattern in Part 2 is stochastic, with a single re-ordering of pitch material occurring before a more deterministic repetition of the re-ordered pattern. The pattern in Part 3 is stochastic, presenting continuous re-orderings of the Part 3

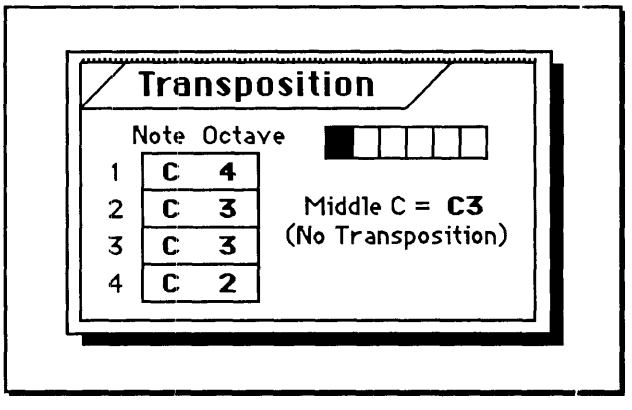
pitch material. The Part 4 pitch pattern is subject to all three re-ordering processes, combining both stochastic and deterministic processes.

Figure 2.3 *M*, Note-order variable



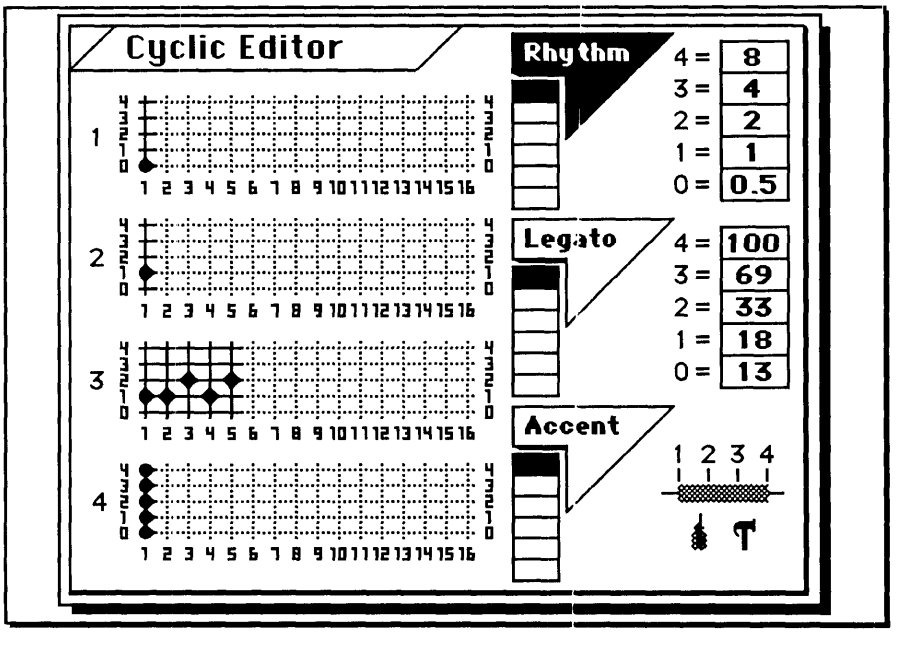
The Transposition variable in Figure 2.4 is an example of a purely deterministic algorithm, containing a fixed offset, 'expressed as a note above or below middle C, which is added (or subtracted) to the note values of each pattern' (Zicarelli 1987, p.20). The pitch pattern in Part 1 is always performed one octave higher than the original, Parts 2 and 3 are at pitch, and Part 4 is played one octave lower.

Figure 2.4 *M*, Transposition variable



Cyclic variables in *M* are data structures that have some number of scalar or random ranges, and are applied to duration, articulation (*legato* versus *staccato*), and accent (MIDI velocity) (Zicarelli 1987, pp.20-1). Cyclic variable settings are applied to each of the four parts independently as determined by the selection of each part's x and y matrix within the Cyclic editor. The x and y matrices, as shown in Figure 2.5, are positioned vertically on the left side of the Cyclic editor. On the x axis the numerals 0 to 4 correspond to numerals in the multiplier vertical array in the upper right of the Cyclic editor. The numerals in this array multiply or divide a clock speed which is set on the main screen. On the y axis of the Cyclic editor matrices the numerals 1-16 represent time steps, the user selecting up to 16 time steps to be cycled through by highlighting a desired number of steps. Figure 2.5 illustrates rhythmic settings for the four pitch parts, the array at the upper right indicating, for example, 1/16th notes as 0.5 through to whole notes as 8. Following this example, Parts 1-3 are deterministically based: Part 1 will sound as continuous 1/16ths (0 on the x axis), Part 2 as continuous 1/8ths (1 on the x axis), and Part 3 as 1/8th, 1/8th, 1/4, 1/8th, 1/4 (1,1,2,1,2 on the x axis). Part 4 however has the full range of rhythmic values (0 to 4 on the x axis), representing a stochastic process in which any of the defined rhythmic values are randomly used within the part. Cyclic variables, like the variables, also have six available configurations.

Figure 2.5 *M*, Rhythm Cyclic variable editor



The array and combinatorial possibilities of parameter settings in *M* provide an interactive composition environment in which users 'can impose arbitrary distinctions between composing and performing by restricting their actions at specific times' (Zicarelli 1987, p.14). As in the Note-density variable, variables for velocity range (dynamics), note ordering, transposition, rhythm, articulation, and tempo are each independently configurable. In the operation stage, adjustments to any variable configuration may be made at any time to affect the musical output of pitch material within pattern groups. As such, the user of the program is free to manipulate an array of variable parameters, 'conducting an orchestra of ideas and transformational processes' (Yavelow 1987, p.222).

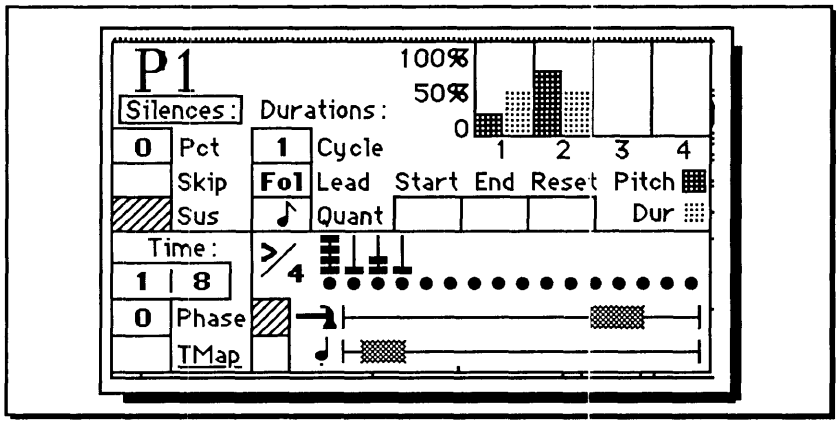
2.2.3. *Jam Factory*

Jam Factory was developed by David Zicarelli in 1986, and like *M*, the program is based on the design/operation paradigm developed by Chadabe (Zicarelli 1987, p.13). Similar to *M's* four part patterns, *Jam Factory* implements 'four polyphonic sequencer modules, called *players*, which the user "teaches" by playing MIDI data into them' (Yavelow 1987, p.222). Each of the four players has a variety of controls for pitch, rhythm, dynamics and articulation, implemented with both stochastic and deterministic processes.

Pitch and rhythmic controls for each player are stochastically based. Each player 'holds an input stream of pitches and durations, each with its own set of transition tables' (Zicarelli 1987, p.23). 'The transition table is a way of storing the probability that a certain action happened given some number of previous actions' (Zicarelli 1987, p.24), and allows the computer to "improvise" on pitch and rhythmic materials stored in the table (Burns 1994, p.114). The number of previous actions is referred to as an *order*. As an example with pitch, in a 1st- order transition table the probability that one pitch will occur is based on the occurrence of an immediately preceding pitch. In a 2nd-order table, the probability that a pitch will occur is based on the occurrence of two preceding pitches. A 3rd-order 3 table is based on the occurrence of three preceding pitches. *Jam Factory* allows 1st-order through to 4th-order transition tables.

Figure 2.6 shows the controls for one of *Jam Factory's* four players, the graph in the upper right referring to entries in transition tables. The darker portion of the graph refers to pitch transition tables, the lighter portion to durations. Numerals beneath the graph refer to the orders. The graph indicates that, for pitch, a 1st-order table will be used 25% of the time, and a 2nd-order table used 75% of the time. For durations, a 1st-order table will be used 50% of the time and a 2nd-order table used 50% of the time. The probabilistic processes for pitch and rhythm control in *Jam Factory* are known as Markov chains, named after the 19th century Russian mathematician, Andrei Andreiëvich Markov who conceived the process as 'a means by which decisions could be made based on probability' (Burns 1994, p.15). Markov processes are further detailed in Chapter Five of this study.

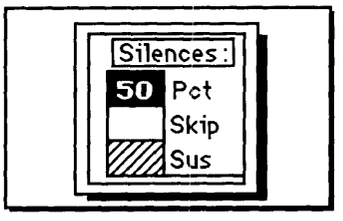
Figure 2.6 *Jam Factory,* Player 1 settings



Durations in *Jam Factory*, apart from being controlled by transition tables, can be stochastically controlled using the Silences algorithm shown in Figure 2.7. This algorithm implements the same process as the Note-density algorithm in *M*, but in reverse, 'the percentage is of silence, not playing' (Zicarelli 1987, p.25). More deterministic controls add variety to a performance:

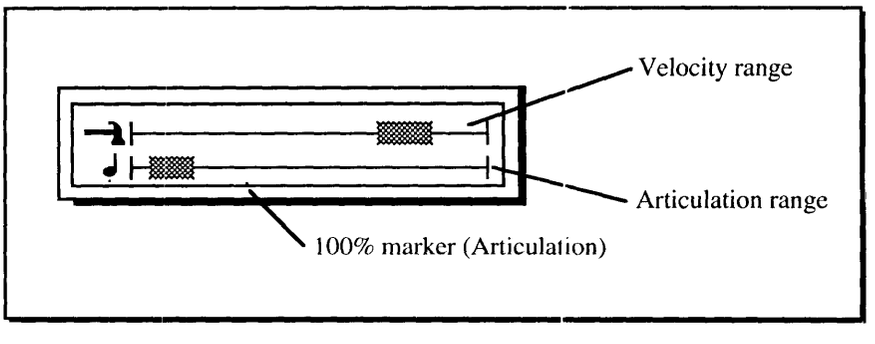
> Skip, when enabled, causes the next note in the sequence from the transition tables to be "thought of" when it isn't played (producing the effect of skipping a note in the melody). Sustain, when enabled sustains any notes that might be playing through a random silence, giving the impression that the notes have a longer duration (Zicarelli 1987, p.25).

Figure 2.7 *Jam Factory,* Silences algorithm



Similar to the variables and Cyclic variables in *M*, further algorithms within *Jam Factory* allow combinations of stochastic and deterministic processes for control over numerous musical parameters. The Velocity-range (dynamics) and Articulation-range algorithms shown in Figure 2.8 allow settings for both of these parameters to be determined as a singular setting, or can be set to include a range of values that vary randomly within a chosen range. In Figure 2.8 the overall range of velocities is from 0-127. The bar graph is set to include random velocities from an approximate range from 70 to 104, as indicated graphically by the width of the grey rectangle. The articulation range is expressed as a percentage from 0 to 800% of time between pitches, the small vertical marker on the border immediately below the articulation range line indicating 100%. If the articulation range were set deterministically as a singular value (by narrowing the width of the grey rectangle down to a single line) from 100% to 100% (immediately above the 100% marker) the result would be that each pitch would last the full duration until the following pitch. Higher percentages result in overlaps of pitches (*legato*) and lower percentages result in shortened pitches (*staccato*). The setting in Figure 2.8 is approximately between 20% and 60%, resulting in *staccato* articulations within the set range.

Figure 2.8 *Jam Factory,* Velocity/Articulation algorithm



As in *M*, each algorithm in *Jam Factory* can be manipulated in real-time. The initial input of pitches, chords and durations in each of the four players is subject to degrees of variation depending on the player settings, the provided algorithms allowing the user to interact with the four players as they "improvise" on the initial materials (Chadabe & Zicarelli 1986a, p.3,1). The combinations of stochastic and deterministic algorithms in both *M* and *Jam Factory,* along with their implementation using the MIDI specification, led to a 'phenomenal response' to the programs, the first time any effort at automated composition had met 'with the slightest commercial success' (Ames 1989, p.175).

2.2.4. *Symbolic Composer*

Symbolic Composer was developed by Peter Stone of Tonality Systems, and is described as 'a powerful modelling and development system suitable for all kinds of music' (Morgan 1990b, p.4). The program implements an alpha-numeric interface which allows symbolic and numeric expressions of musical parameters, and the manipulation of such expressions:

> with both common and experimental routines of composition... The software contains a full set of mathematical, symbolic and conversion functions facilitating the creation of any internal mathematical or symbolic model[s] and their mapping onto musical parameters (Morgan 1990a).

Symbolic Composer's manual tutorials present a series of examples, the first of which is presented in the following overview of the software, tutorials that methodically expand in conjunction with skills gained by the user. While some examples in the numerous *Symbolic Composer* tutorials may be considered as compositions in their own right, they are didactic in nature and serve to expose the user to a small proportion of algorithms available within the program. The full extent of the program's algorithmic capability is only visible after considerable time spent with the tutorials, papers and a *Hypercard* 'stack' supplied with the program (Sica 1994, p.108).

The philosophy of *Symbolic Composer* is that composition involves symbol manipulation. In Western music notation, symbols are traditionally used deterministically to represent musical parameters. With computer-based alpha-numeric languages, 'the meaning

that a symbol assumes can vary according to the representation to which it is bound' (Sica 1994, p.108). As an example, the symbols a b c and d may be mapped onto a C major scale-based note series of C D E and F. Alternatively, these same symbols may map to a chromatic-based scale of F# G G# and A. Similarly, symbols may be used to represent rhythmic structures, for example a b c d may represent rhythmic values of 1/8 1/16 1/16 1/4, or 1/4 1/4 1/8 1/8. Simple representations such as these form the basis of *Symbolic Composer's* system of mapping symbols to musical parameters. A process of conversion provides an ability to 'transform one representation into another without loss of meaning in the data (i.e. converting a symbol pattern into a numerical pattern or vice versa)' (Sica 1994, p.108).

A wide variety of stochastic and deterministic algorithms within the program allow symbolic representations to form the seeds of a musical composition. The processes of mapping and conversion are used to 'generate, process, analyse and reprocess any kind of object' (Sica 1994, p.108). Generators, for example, allow symbolic and numeric patterns to be generated using a variety of processes including recursive, fractal- and random-based algorithms, while processors allow various transformations of symbolic and numeric patterns such as retrogrades and inversions. Further functions within the program include Artificial Intelligence-based neural networking facilities, libraries of symbol patterns, chords and scales and a 'Visualizer' which presents visual interpretations of symbolic and numeric data.

Figure 2.9 is a *Symbolic Composer* 'score' taken from the manual (Thomas & Morgan 1990, pp.102-03), and provides an overview of the *Symbolic Composer* layout. Alpha-numeric data is presented in *chunks*, enclosed in parentheses according to LISP syntax rules. Each chunk in the example represents a definition of symbolic or musical parameters, and the order of the chunks has its own particular logic:

- variable structures
- instrument definitions
- tonality mapping definitions
- compilation instructions
- timesheet

(Thomas & Morgan 1990, p.103).

A semicolon at the start of a line disables any compilation of information on that line, enabling the user to write in memos, titles, guides etc. that are not required for the running of the program.

Figure 2.9 *Symbolic Composer,* Tutorial example (Thomas & Morgan 1990, pp.102-3)

```
; tutorial example 1 -  mctest1

(setq symbols '(a b c d e f g))

(def-instrument-symbol
  test1 symbols
  test2 symbols
  test3 symbols
  test4 symbols
)

(def-instrument-length
  default 1/16
)

(setq tonals (activate-tonality  chromatic c
6) (blues1 c 4)))

(compile-song "ccl;output:" 1/4 separate

; BARS                |---|---|---|---|
changes tonals      " .. "
test1   changes     "-   "
test2   changes     " -  "
test3   changes     "  - "
test4   changes     "   -"
)
```

The first chunk '`(setq symbols '(a b c d e f g))`' is a variable structure and simply defines a set of symbols used within the example. The defined symbols at this stage could apply to any number of musical parameters such as pitch, rhythm or dynamic level. The activation of symbols is incorporated within later chunks that define separate parameters individually. The function `setq` allows a variable to be created. The word `symbols` is user defined and may be any word. If the defined symbols are applied to pitch, the word used can refer to the symbols' control over that parameter: the word `melody` for example, would be appropriate. While the Figure 2.9 chunk defines a single symbol pattern, any number of

symbol patterns can be defined with the `setq` variable, allowing, to follow the previous example, definitions for a `melody1`, a `melody2`, a `melody3` etc. Each definition may have a different symbol pattern, as shown in Figure 2.10.

Figure 2.10 *Symbolic Composer*, Alternative symbol patterns

```
(setq melody1 '(a b c d e f g)
      melody2 '(h i j k l m n)
      melody3 '(a z b y c x d)
)
```

The second chunk in Figure 2.9 defines a name for each instrument (in this example instrument names are replaced with `test1` through to `test4`) and a symbol pattern associated with each instrument's pitch structure. While each of the four instruments in Figure 2.9 uses the same symbol pattern, instruments can adopt any defined symbol pattern. For example, if the symbol patterns defined in Figure 2.10 were used, `test1` could be assigned to either `melody1`, `melody2` or `melody3`.

The third chunk in Figure 2.9 defines the rhythmic material of the example. Here the rhythmic unit is a 1/16th note value. The word `default` is a shortcut to defining the 1/16th note value for all four instruments, saving the user the more lengthy writing of each instrument name and its corresponding rhythmic unit. Instruments are not limited to singular rhythmic values, as they are in Figure 2.9, but may have multiple rhythmic values and differing rhythms from other instruments, as shown in Figure 2.11.

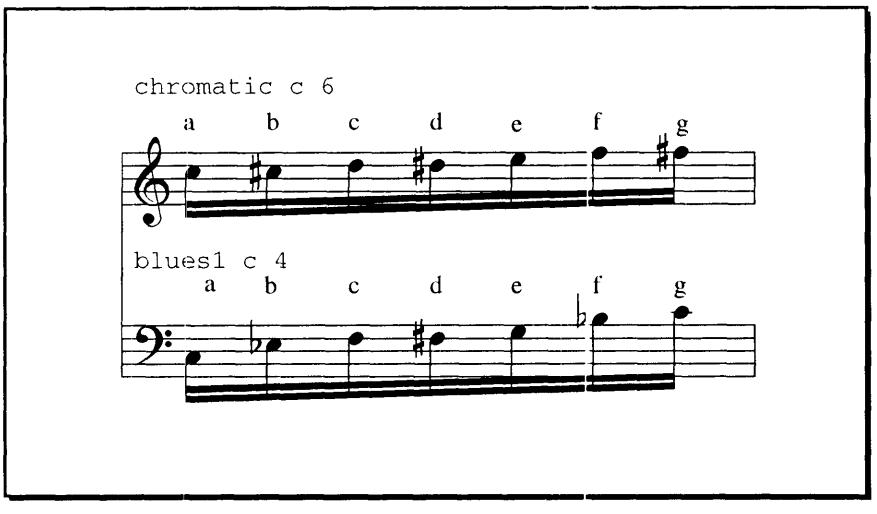
Figure 2.11 *Symbolic Composer,* Alternative length patterns

```
(def-instrument-length
test1 '(1/8 1/16 1/16)
test2 '(1/16)
test3 '(1/4 1/4 1/8 1/8)
test4 '(1/2 1/4)
)
```

The fourth chunk in Figure 2.9 defines the variable `tonals` which describes a particular tonality consisting of two separate scales: a chromatic scale starting on the C one octave above middle C (middle C in *Symbolic Composer* is defined as `c 5`) and a blues scale starting on the C one octave below middle C, (`c 6` and `c 4` respectively). The function `activate-tonality` calls up the two scales from *Symbolic Composer's* pre-defined scale library.

At this stage of the *Symbolic Composer* score, the defined symbols (`a b c d e f g`) are mapped to either the blues scale or the chromatic scale with a continuous 1/16th note rhythm, as shown in Figure 2.12. Where there are more symbols than scale pitches, *Symbolic Composer* restarts the scale up one octave as shown with the `blues1` scale and the symbol `g` in Figure 2.12.

Figure 2.12 *Symbolic Composer,* Symbol to pitch mapping

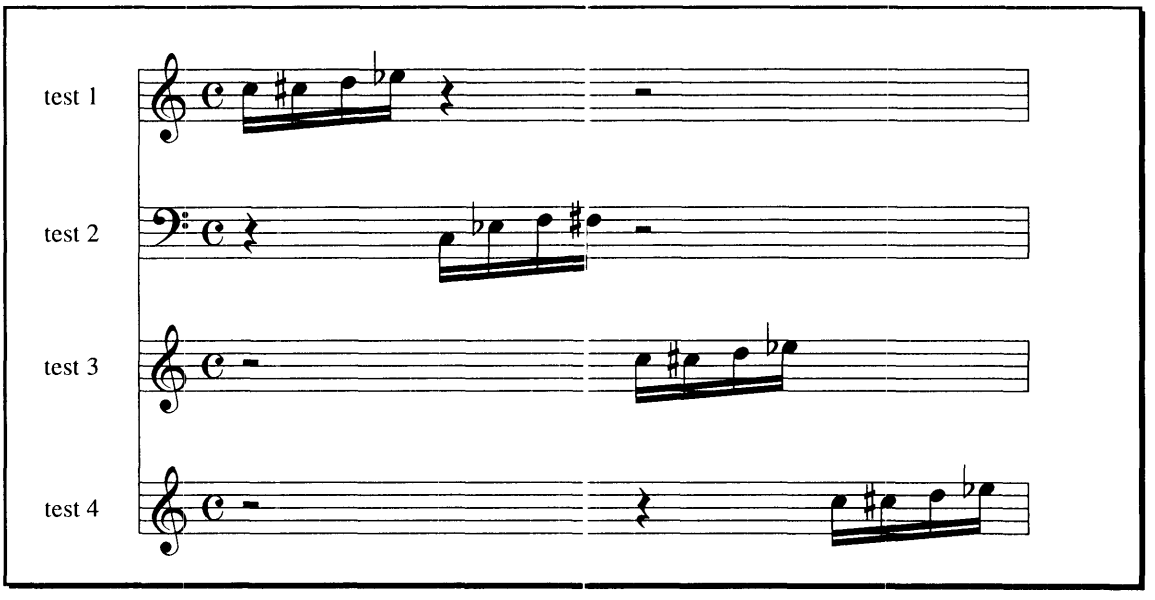


The fifth chunk in Figure 2.9 begins with directions for storing MIDI files on compilation of the score. In this example, the MIDI file will be stored in a folder named 'output'. Next in the chunk is a *Symbolic Composer* timesheet that shows when each instrument plays or is muted. The vertical and horizontal lines within the timesheet represent time periods according to a defined resolution which is stated after the previous compilation directions. In this example this resolution is `1/4`, indicating that each line within the timesheet lasts for a 1/4 note duration. The vertical lines in the timesheet represent the first

1/4 note in measures of 4/4 time, the following three horizontal lines represent the remaining three 1/4 notes in each measure. An absence of lines (i.e. blank space) indicates muting of instruments. With the previous rhythmic definition of `1/16` for each instrument, each line in the timesheet will represent a statement of four 1/16th notes. The result of the fifth chunk is that `test1` plays four semiquavers on the first beat of the measure, `test2` plays four semiquavers on the second beat, `test3` plays on the third beat and `test4` on the fourth.

The word `changes` is a variable that relates to the two scales defined in `tonals`. This variable functions between the double quotation marks that follow the words `changes tonals`. The period marks, which in Figure 2.9 occur in relation to the second and third beats of the 4/4 measure, indicate where changes occur from one scale to another, whilst blank spaces, which occur on the first and fourth beats, indicate that no change of scale occurs on those beats. In Figure 2.9 the initial tonality is the chromatic scale, the first scale defined in the previous chunk. The first change point occurs on the second beat of the measure and activates the blues scale. The second change point occurs on the third beat and returns the tonality to the chromatic scale. The blank space on the fourth beat indicates that the chromatic scale remains in use. Figure 2.13 shows the result of the example in common notation.

Figure 2.13 *Symbolic Composer,* Output in common notation



Stochastic and deterministic algorithms available within *Symbolic Composer* are applied to musical parameters by inserting algorithms as alpha-numeric data into the chunks that define individual parameters. Within the program:

> music can be expressed as anything that can be presented by data or algorithms; an atom, a genetic code, the planetary system, temperature curves, stockmarket figures, program structures, language constructs, mathematical functions, theories, brain waves, text or any other information that can be interpreted as data or algorithms by a computer (Thomas & Morgan 1990, pp.45-6).

'In short, this is a program for transforming any kind of data or algorithm into a musical structure' (Sica 1994, p.108).

2.2.5. *MAX*

MAX was initially developed in 1986 by Miller Puckette at the *Insitut de Recherche et de Coördination Acoustique/Musique* (IRCAM) in Paris. 'Originally developed as a non-graphical language intended to control IRCAM's powerful 4X synthesiser, *MAX* was later implemented as a graphical environment for MIDI on the Macintosh' (Dobrian 1990a, p.1). The program was prepared for commercial release by David Zicarelli for Opcode Systems in 1990. *MAX* is based on object-oriented programming languages, 'in which programs are realized by manipulating graphic objects on a computer screen and making connections between them' (Rowe 1993, p.25). Object-oriented programming languages (OOP) originated with the programming language SIMULA in the 1960s and were subsequently developed in the early 1980s with the programming language SMALLTALK (Blaschek 1994, p.11). Object orientation:

> is a programming discipline that isolates computation in *objects*, self-contained processing units that communicate through passing *messages*. Receiving a message will invoke some *method* within an object. Methods are constituent processing elements, which are related to each other, and isolated from other methods, by virtue of their encapsulation in a surrounding object. Depending on the process executed by a method, a message to an object may enclose additional arguments required by that process as well (Rowe 1993, p.26).

Whilst *MAX* has an object-oriented programming language basis, it does however lack the object-oriented programming concept of inheritance, 'by which objects can be defined as specializations of other objects' (Rowe 1993, p.26).

Unlike *M, Jam Factory* and *Symbolic Composer*, which are programs that offer the composer numerous pre-defined algorithms, *MAX* is a *development* program that allows the user to create his or her own algorithmic processes to manipulate numerical data. Within the program:

> a composer specifies a flow of MIDI or other numerical data among various objects representing such operations as addition, scaling, transposition, delay etc. A collection of interconnected objects is called a *patch* (Rowe 1993, p.26).

As a development program, *MAX* represents a middle-ground between commercially available composition software and the composer/programmer paradigm. With the relatively clear programming language and the ease of programming offered, composers using *MAX* are able to design algorithms based on scientific and mathematical principles, and use the output of such algorithms as MIDI data. The following *MAX* patches illustrate the programming style used in *MAX* and show the manner in which a simple patch for pitch transposition (Figure 2.14) can be expanded to incorporate delays and harmonisation (Figure 2.15).

Figure 2.14 shows a *MAX* patch that transposes music played on a synthesiser. The notein object receives, via a Macintosh serial port from an external MIDI device, MIDI note-on data (a pitch value and a velocity value). The notein object sends the pitch value out an outlet which is represented by the black portion on the bottom left of the object, and sends the velocity value out the middle outlet. If a middle C were played on the synthesiser at a moderately loud dynamic level, the actual MIDI information received by the notein object would be: 60 64 60 0, wherein the first two numbers represent the pitch value of middle C (60) and the velocity of 64 in the MIDI scale of 0-127. The second two numbers indicate MIDI note-off information, transmitted when the middle C pitch is released, wherein the velocity value for middle C (60) is zero. The note-on data (60) is then received by the + object which in this patch is set to add 7 to any incoming data, thus changing the 60 to 67. In terms of pitch, the + object transposes the middle C note up seven semitones to the G above middle C. This new pitch value is then sent out the + object outlet.

The noteout object works in the same way as the notein object but in reverse. It transmits data it receives to a serial port of the Macintosh and then on to an external MIDI device. The inputs to the object are like the outputs of the notein object, the left input receives pitch data to transmit to the synthesiser, the middle input receives velocity data. In Figure 2.14 the velocity values are transmitted directly from the notein object to the noteout object without change.

Figure 2.14 *MAX,* Simple patch for note transposition

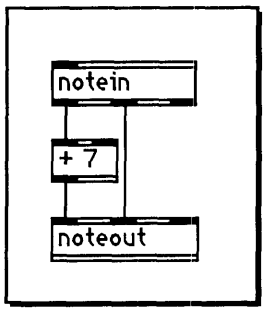
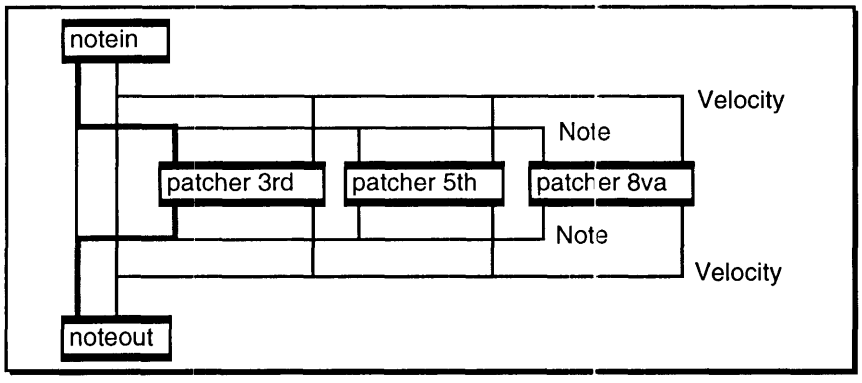


Figure 2.15a shows a patch that functions in the same way as the Figure 2.14 patch but is expanded with three sub-patches. Sub-patches in *MAX* are embedded in a main patch and are defined by the user with the *MAX* patcher object. A sub-patch may contain any number of objects. The content of the first two Figure 2.15a sub-patches is shown in Figures 2.15b and 2.15c. In Figure 2.15a, pitch data is passed simultaneously to the noteout object, as shown by the direct lines (connections) from the notein objects pitch and velocity outlets to the noteout objects pitch and velocity inlets, and also to the sub-patcher objects labelled patcher 3rd, patcher 5th and patcher 8va. For clarity, the pitch data connection from the notein object to the patcher 3rd object is shown in bold in Figure 2.15a. The direct connections from the notein to the noteout signify that pitch data sent from the synthesiser will be passed through *MAX* via notein and noteout and returned to the synthesiser without change. This same pitch data is also sent to inlets of the sub-patcher objects via branching connections from the notein object, and is processed by further objects within the sub-patchers.

Figure 2.15a *MAX,* Patch with sub-patches



At the top of Figure 2.15b there are two *MAX* inlet objects (with shaded downward triangles) that correspond to the two inlets of the top of the Figure 2.15a subpatcher object labelled patcher 3rd. The left inlet object receives pitch data sent from notein while the right receives velocity values. The +4 object in the patcher 3rd sub-patch functions in the same way as the +7 object in Figure 2.14, except that pitch data is transposed up 4 semitones (a major 3rd) instead of the 7 semitone transposition of the former patch. The direct connection from the +4 object to the left-most outlet object (with a shaded upward triangle) signifies that the transposed pitch will sound simultaneously with the original pitch passed from the notein to the noteout in the main (Figure 2.15a) patch. This results in a harmonisation of the original pitch played on the synthesiser (if C were played, an E above would be added). The Figure 2.15b sub-patch also serves a second purpose by sending the pitch data received by the sub-patch to a *MAX* pipe object. The pipe object delays data it receives by a number of milliseconds specified by the user. In Figure 2.15b this delay is of a 250 millisecond duration. From the pipe object outlet, the delayed pitch data is sent to the sub-patcher's outlet object and then, in the main patch, to the noteout object. Velocity values that enter the patcher 3rd sub-patch take two paths. They are passed from the sub-patcher's inlet object directly to the outlet object to correspond to the non-delayed notes that pass through the sub-patch, and are also sent to a pipe object, to correspond to delayed pitch data. The presence of the pipe object in the velocity data path ensures that each pitch value entering the sub-patch retains a positive velocity signifying a MIDI note-on message, and a zero velocity signifying a MIDI note-off message.

The Figure 2.15c sub-patch functions in the same way as the Figure 2.15b sub-patch with the exception of the transposition value which transposes incoming pitch values up seven semitones (perfect 5th), and the delay value which delays a pitch value by 500 milliseconds or a half second. The third sub-patch (patcher 8va) in Figure 2.15a again provides the same function, but transposes pitch values up 12 semitones and delays pitch by 750 milliseconds. Figure 2.15d shows, in common notation, the result of playing a single middle C on a synthesiser and routing the MIDI data through the Figure 2.15a patch.

Figure 2.15b *MAX,* Transposition patch (patcher 3rd)

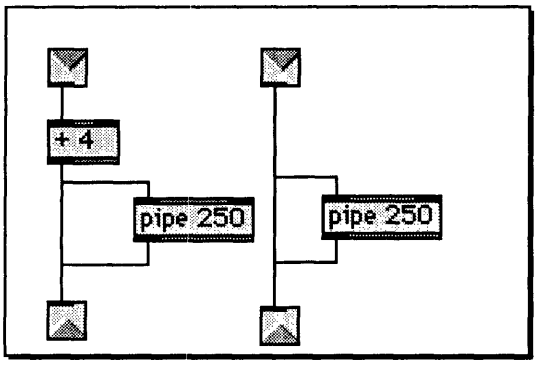


Figure 2.15c *MAX,* Transposition patch (patcher 5th)

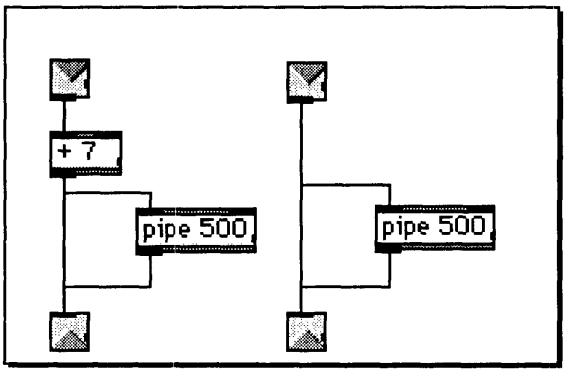
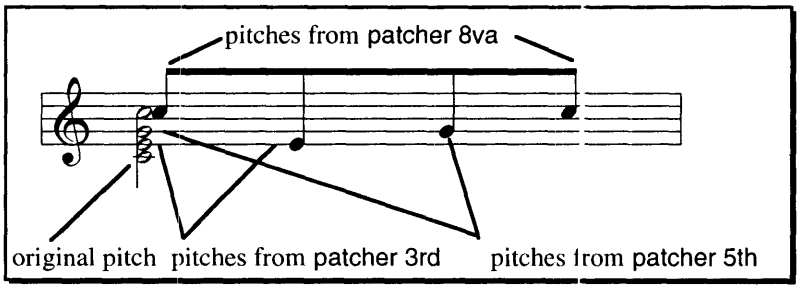


Figure 2.15d *MAX,* Patch result



The preceding patch examples illustrate a working methodology in *MAX*, the user specifying where data arrives from in each patch with a MIDI object such as notein or, in the case of sub-patches, from connections made to the sub-patch inlets. Combinations of *MAX* objects within patches and sub-patches then allow the user to develop a wide variety of stochastic and/or deterministic processes with which incoming data can be manipulated. Within the *MAX* environment there are over 200 low-level objects (like the + and pipe objects shown above) that the composer uses to extend the musical capabilities of the software 'to higher levels of abstraction which permit larger scale musical concepts to be realised' (Henderson-Sellers & White 1996a, p.1). Use of further MIDI objects such as noteout allows manipulated data to exit the program to a MIDI device. Developed patches can be saved either as *MAX* documents or within a stand-alone application that combines a *MAX* reader called *MAXPLAY* with user developed patches.

Unlike the small amount of literature concerning the software programs of *M*, *Jam Factory* and *Symbolic Composer*, documentation on *MAX* is extensive. Many users of the program publish their work with the program in journals, for example Lindsay Vickery (Vickery 1996, pp.9-10) and Stuart Favilla (Favilla 1996, pp.27-9) in a recent *Sounds Australian Journal*, whilst the patches they develop are regularly made available for downloading from the internet, for example Henderson-Sellers and White with their *LorenzSA MAX* patch (Henderson-Sellers & White 1996b). For the purposes of this study, the preceding discussion of *MAX* suffices to provide an overview of the program, whilst in Chapter Eight, a more extensive discussion of *MAX* objects and functions is given in the context of the *Phrase Garden* program that was developed with *MAX*.

2.3. Related Literature

Literature pertaining to historical and technical aspects of algorithms used in music is broad, drawn from a diverse array of both primary and secondary sources. In the examination of any one algorithm, various sources provide firstly, the historical background of an algorithm, secondly, technical descriptions of an algorithm's application within mathematical or scientific contexts, and thirdly, applications of an algorithm within musical

contexts. As an example, the historical background of Markov chains is documented in numerous sources (Ames 1988; Borowski & Borwein 1989). Markov applications in mathematical and scientific contexts are documented in a wide variety of fields, for example, from geography (Collins 1975) through to computer sciences (Knuth 1973). Within the composer/programmer paradigm, documentation of musical applications of Markov processes are similarly varied (Ames 1988; Dodge & Jerse 1985; Hiller 1970; Jones 1981; Olson & Belar 1961; Pinkerton 1956; Xenakis 1971). In the following chapters, numerous citations of literature in this area provide relevant historical and technical backgrounds to specific algorithms used within the example works provided in this study.

2.4. Summary

While many applications of algorithms have been documented from within the non-commercial composer/programmer paradigm, literature pertaining to algorithms within commercially available composition software is limited to documentation of software specifications by software authors, and reviews such as those summarised in the previous software overviews. Specific applications of algorithms are limited to didactic examples provided within software programs. In the tutorials of both *M* and *Jam Factory* the applications of inherent algorithms are limited to 'guided tours' of the software programs, the results of which are ultimately defined by the user, while in *Symbolic Composer's* tutorials the example works, though they resemble complete compositions, are essentially didactic works that guide the user toward the skills necessary to operate the software.

Conversely, the documentation provided with *MAX* gives over 40 tutorials in which many of the program's objects are discussed at length, while the reference manual for the program provides examples for each *MAX* object. *MAX* is, however, a development program and whilst *MAX's* low-level objects are extensively detailed in the manuals, higher level abstractions (i.e. algorithms built from these objects) fall more into the category of the non-commercial composer/programmer paradigm, with composers detailing their individual work with *MAX* in journals, monographs and via the Internet.

PART TWO

CHAPTER 3 — METHODOLOGY

3.1. General Methodology

In each of the example works composed for the study, algorithms were selected according to specific requirements of the individual compositions. In general, the algorithms selected are in accord with pre-defined guidelines for musical parameters and/or programmatic associations developed for each work.

The four example works are discussed individually in separate chapters, each chapter providing a compositional overview that details pre-defined musical and programmatic materials employed in each work. Following the compositional overviews in each chapter are sections detailing compositional structures. These sections are comprised of sub-sections detailing pitch and rhythmic structures used in each work, and a sub-section detailing timbre when the example works are in an electronic or electro-acoustic format. Where applicable, further sub-sections are given to cover details of compositional structures not provided in the pitch, rhythm or timbre sub-sections.

Following the sections on compositional structures in each chapter are sections detailing the applications of algorithms within the example works. In these sections are examinations of the specific algorithms used in each work, incorporating historical and technical aspects of algorithms, and the applications of algorithms in the contexts of the example works. In Chapters Four and Five, interactive software programs are used in the composition of the example works, sub-sections within these chapters detailing deterministic algorithms, stochastic algorithms and the applications of algorithms in real-time respectively. In Chapter Six, algorithmic composition software is used in the composition of the example work. In this chapter, the section detailing the applications of algorithms is comprised of sub-sections detailing grammar-based algorithms and chaos-based algorithms respectively.

Chapter Four of the study provides an examination of the work composed with *M*, an electro-acoustic work entitled *Study for Triangles*, Chapter Five provides an examination of the work composed with *Jam Factory*, an electronic work entitled *Étude in Memoriam Allan*

Dagg, and Chapter Six provides an examination of the work composed with *Symbolic Composer*, an acoustic work entitled *Descendant Lines.*

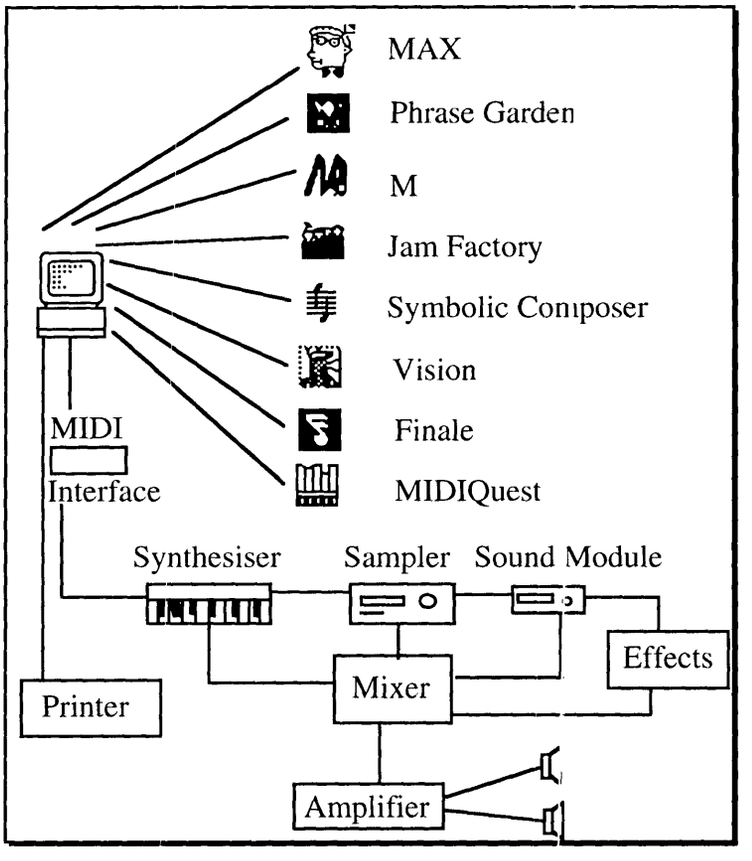
In Chapter Seven the personal compositional style employed in the initial three example works is discussed, and brief analyses of two instrumental works composed without computer assistance are given. The analyses are used to demonstrate incompatibilities between composition processes available in commercially available software and those used in the compositional style. Chapter Eight details the *Phrase Garden* program and demonstrates the manner in which *MAX* is used to develop this program to be closely aligned with the compositional style employed in the two instrumental works analysed in Chapter Seven. Chapter Nine provides an examination of a work composed with the *Phrase Garden* program, an acoustic work entitled *When Cinderella's Monkey Comes Here, I'll Feed Him.* Chapter Nine is structured in the same way as Chapters Four, Five and Six, providing an overview of the example work, a section on composition structures and sections detailing the application of algorithms in the composition of the work.

3.2. System Configuration

The system used for the development of the four example works is shown in Figure 3.1. This is a simple MIDI-based system centred on an Apple Macintosh computer, software for the Macintosh platform, and various devices for sound output and printing. The software component comprises the interactive composition programs *M, Jam Factory* and *Phrase Garden,* the algorithmic composition program *Symbolic Composer*, the development program *MAX*, *Vision*, a sequencer used for playback and editing of MIDI data, *Finale*, a notation program used for printed output of acoustic instrument parts, and *MIDIQuest*, a MIDI patch editor/librarian program used for the development and editing of MIDI instrument patches. For sound output, two external hardware devices for the generation of FM (Frequency Modulation) synthesised sound are employed (Chowning 1973), and combined with a sampling module. Final sound output is controlled with a digital effects unit, a mixing console, and amplification.

This configuration forms a fundamental MIDI system, a minimum in regard to developing works with automated composition systems, and is accessible on a relatively small budget (Yavelow 1992, pp.241-4). Whilst it is a fundamental configuration, it represents the core of more sophisticated and expensive systems, systems in which interactive and algorithmic software are used to control composition processes, and systems in which various types of sound synthesis and sampling modules are employed for sound output. As such, this fundamental configuration is familiar to composers using automated composition systems, from those with similar configurations in small-scale home studios to those with access to more sophisticated systems.

Figure 3.1 System configuration



CHAPTER 4 — *M -- Study for Triangles*

4.1. Overview

The *Study for Triangles* (Track 1 on the accompanying audio CD) was initiated after simple sound editing experiments were carried out on two triangle samples. The results of the initial experiments yielded a variety of timbres that were conceived of as the basis for a study featuring triangle samples in combination with acoustic triangles, in which triangle samples could be manipulated in real-time and used for live performance interactions with quasi-improvised textures produced by a set of three acoustic triangles. Various timbral possibilities arising from combinations of samples and acoustic triangles resulted in the formation of a three part form (A B C) for the work, each section focusing on differing timbral possibilities available within the ensemble.

Within the short gestation period of the work, various aspects of triangles in general were considered, leading to an incorporation of simple triangle theorems into pitch and rhythmic structures devised for the work. Based on the Pythagorean *Holy Tetractys* (shown in Figure 4.1), pitch structures used for the edited triangle samples are visually complemented in live performance by the equilateral shape of the acoustic triangles, while the edited samples are complemented by rhythmic structures derived from Pythagoras' theorem on the right-angle triangle.

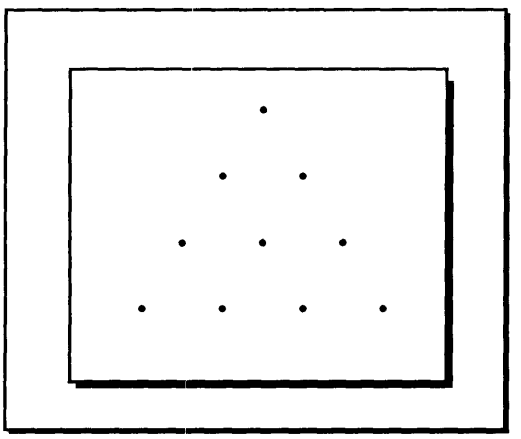
The *Study for Triangles* in general represents a simple work. However, the realisation of the work in live performance requires the implementation of numerous algorithms, both deterministic and stochastic, that enable real-time processing and transformation of the pre-defined musical materials. Using the interactive *M* software, pitch and rhythmic sequences for the composition are stored in the program in a design stage, while in the operation stage, algorithms are called upon for real-time transformation and processing of the pre-defined sequences. The following sections describe the pre-defined materials used in the work, and the manner in which the materials are defined and stored in *M*.

4.2. Composition Structures

4.2.1. Pitch structures

Pitch structures within the *Study for Triangles* are initially derived from the Pythagorean *Holy Tetractys*, a vertical arrangement of the numeric sequence 1 + 2 + 3 + 4, as shown in Figure 4.1. This numeric sequence is taken as a basis for an interval sequence using semitones, and results in a five note sequence of C C# D# F# A#, wherein C-C# is an interval of one semitone, C#-D# is an interval of two semitones, D#-F# is an interval of three semitones, and F#-A# is an interval of four semitones.

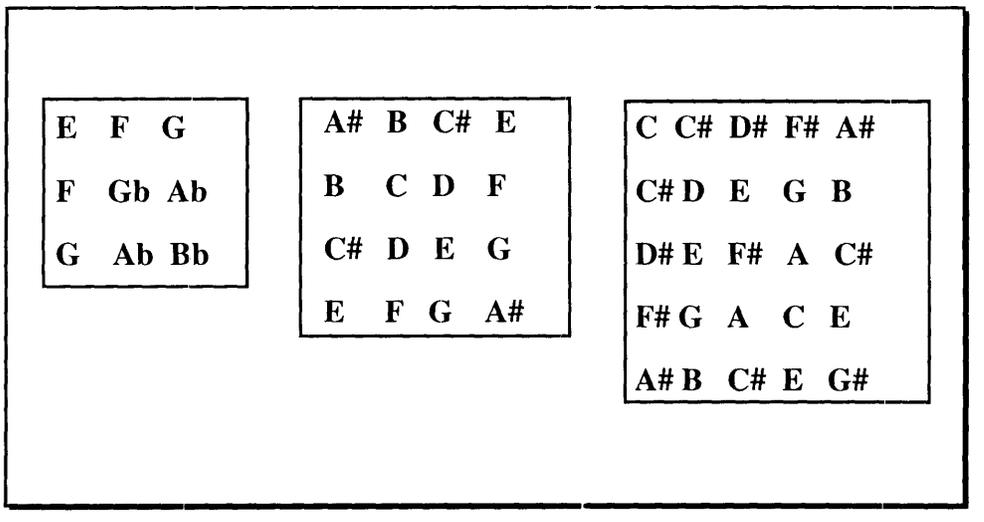
Figure 4.1 Pythagorean *Holy Tetractys*



Further pitch generation from the note series is derived from Pythagoras' equation for the right-angle triangle: $m^2 + \{\frac{1}{2}(m^2 - 1)\}^2 = \{\frac{1}{2}(m^2 + 1)\}^2$, wherein *m* is any odd number other than one. In keeping with the simplicity of the work, *m* was assigned the number 3, the smallest odd number other than one, and results in the numerics 3, 4 and 5 as the square roots of the numbers generated in the equation. With the five note series derived from the *Holy Tetractys*, the number 5 is used as a basis for a note matrix containing 25 notes, as shown in Figure 4.2. The penultimate four notes of this matrix are taken as the starting point of a second matrix based on the number 4 and generating 16 notes. The penultimate three notes in this second matrix form the start of a third matrix based on the number 3 and generating nine notes. The three matrices provide all note material employed within the

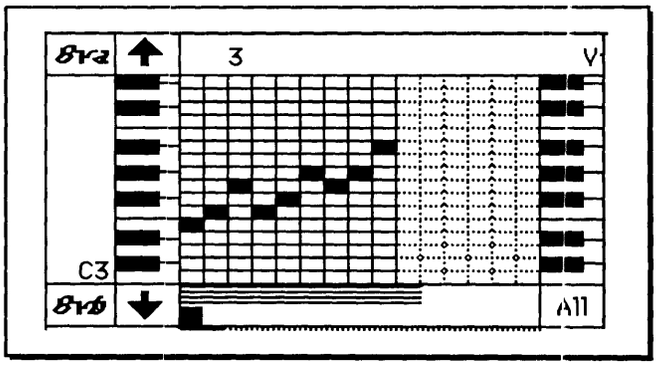
work, the nine-note matrix used in the opening A section, the 16-note matrix used in the central B section and the 25-note matrix used in the final C section.

Figure 4.2 *Study for Triangles,* Note matrices



Pitch materials used in the *Study for Triangles* are, in the design stage, stored in *M* as three separate patterns, one pattern for each of the three main sections of the work. Notes from each matrix are entered sequentially into a pattern and result in a cyclic iteration of notes from the upper left to the lower right of each matrix. Within *M*, representation of stored pitch is in the form of a piano scroll and grid. Figure 4.3 shows the *M* piano scroll containing the nine notes from the A section matrix, each inverted (blacked out) square indicating the tessitura placement and the sequence of individual pitches. The indication C3 denotes middle C.

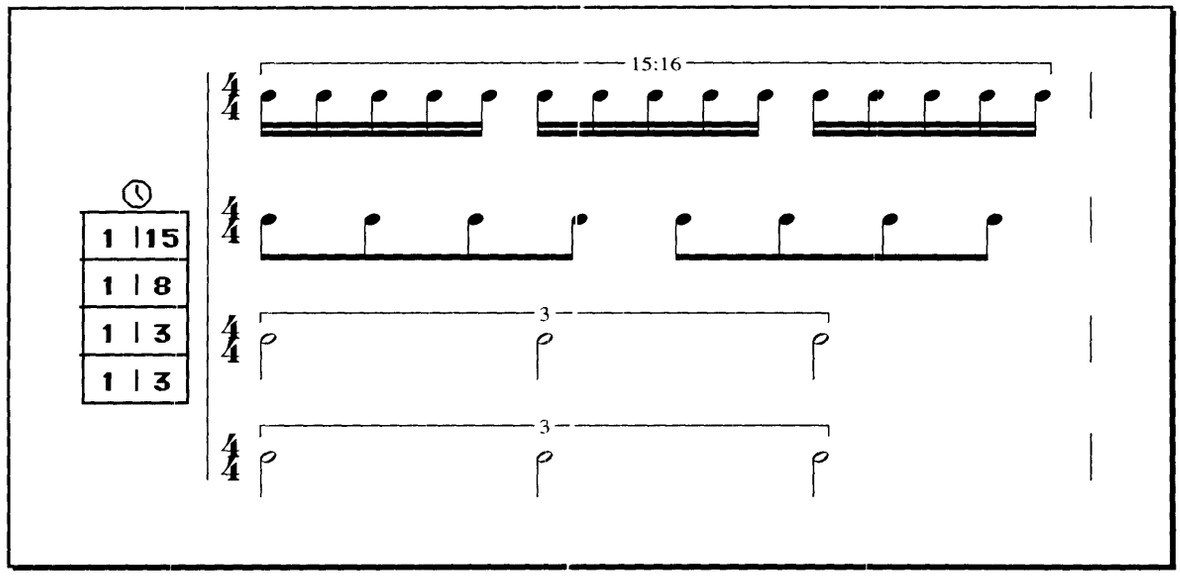
Figure 4.3 *Study for Triangles,* Piano scroll, nine-pitch series



4.2.2. Rhythmic structures

Rhythmic structures within the electronic part of the *Study for Triangles* are, like the note matrices, derived from the Pythagorean equation for the right-angle triangle, and based on the numbers 3, 4 and 5. Within the work, these numbers are initially applied as subdivisions of a 4/4 meter, as triplet, straight, or quintuplet rhythmic values. In the design stage of the *Study for Triangles*, rhythmic values are assigned within *M* with a Time-base algorithm which allows the setting of rhythmic values using numerator and denominator values, as used in standard time signatures. Figure 4.4 shows the numerator/denominator settings for *M's* four parts in the opening section of the work, and the resulting rhythmic values in common notation. Part 1 contains 1/15th notes, in effect representing quintuplet subdivisions of 1/2 note triplets, Part 2 contains straight 1/8ths, Parts 3 and 4 contain 1/2 note triplets.

Figure 4.4 *Study for Triangles,* Time-base algorithm and resulting music notation



Further definitions and manipulations of rhythmic materials are carried out with *M's* editor for Cyclic variables. Within the rhythmic editor of the Cyclic variables, the numerator value set within the Time-base may be multiplied or divided by any of five separate values using the editors multiplier vertical array. Figure 4.5 shows the Cyclic editor rhythm settings for the opening section of the work with the multiplier vertical array settings in the upper

right of the Figure. In Figure 4.5a the resulting rhythmic durations are shown in common notation: Part 1 is set to correspond to the Time-base, position 1 in the multiplier vertical array corresponding to the Time-base numerator. Parts 3 and 4 have multiplications of the Time-base numerator, the multiplier vertical array position 4 indicating the multiplication of the numerator value by 5. As the Time-base denominator for Parts 3 and 4 is a triplet value, the results of the Cyclic editor settings are durations of five 1/2 note triplets. Part 2 is assigned three different settings of its Time-base 1/8th value. In the Cyclic editor matrix for Part 2, the left to right order of the settings results in sequential durations equal in length to four x 1/8th notes, five x 1/8th notes, three x 1/8th notes and five x 1/8th notes.

Figure 4.5 *Study for Triangles,* Cyclic editor. Section A

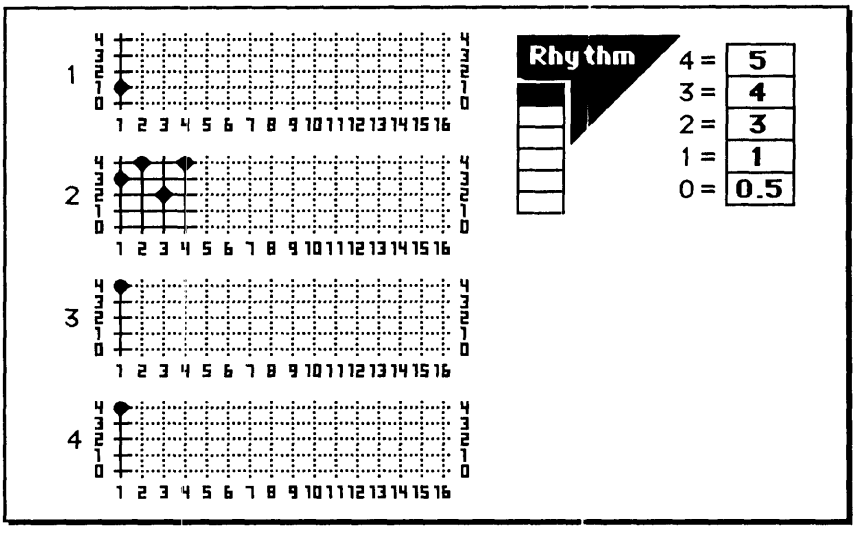
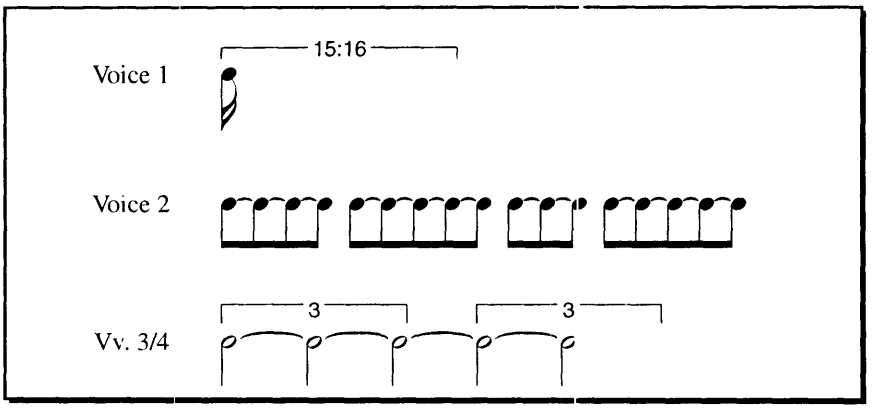


Figure 4.5a *Study for Triangles,* Cyclic editor result in common notation



The Cyclic editor settings for the central section of the *Study for Triangles* are identical to those in the opening section, however the Time-base settings in Parts 1 and 2 are altered and result in an entirely different set of rhythmic values in those parts. In the final section of the work, both the Time-base and Cyclic editor settings are altered. Figures 4.6 and 4.7 show the Time-base and Cyclic editor settings for the sections.

Figure 4.6 *Study for Triangles,* Time-base, Cyclic editor settings, Section B

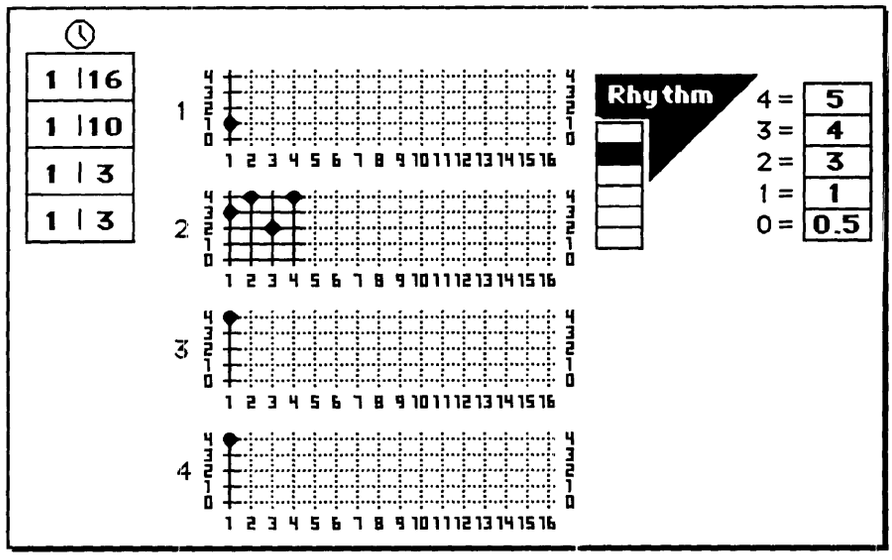
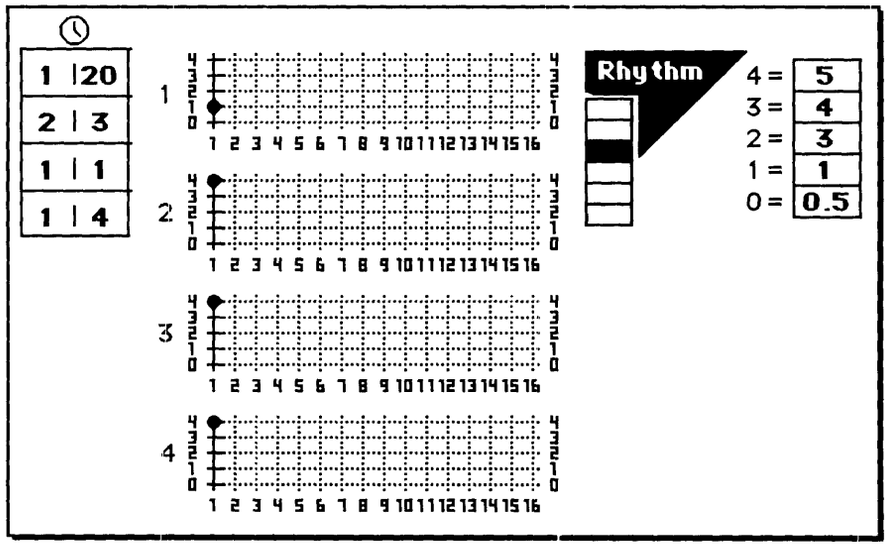


Figure 4.7 *Study for Triangles,* Time-base, Cyclic editor settings, Section C



4.2.3. Timbre

The *Study for Triangles* is primarily a study based on timbre, the timbres available from the three acoustic triangles complementing timbres developed for the electronic part, and the three sections of the work each exploring the various timbral possibilities available with the combination of acoustic and sampled triangles. Timbres in the electronic part, as previously mentioned, are derived from two triangle samples, edited to provide four distinct timbres. Each timbre is complemented with a similar or contrasting timbre by the acoustic triangles:

- Timbre #1 — a sample of a normally played triangle transposed to a low register to give gong-like timbres. This timbre is complemented in the acoustic parts with the swinging of the triangles in full circles, resulting in a doppler effect in which apparent changes of pitch are produced by the triangles' rapid movements toward and away from a stationary audience.
- Timbre #2 — the normally played triangle sample is reduced to the initial attack portion of the waveform, resulting in a *staccato* attack with minimal sustain and decay. This timbre is complemented with a similar *staccato* attack in the acoustic parts, achieved by dampening the triangles with the hand.
- Timbre #3 — The reverse of the above sample, contrasted with the normal playing of the acoustic triangles.
- Timbre #4 — A sample of the triangle when scraped with the beater, complemented with the scraping of the acoustic triangles.

The various combinations of the acoustic and electronic timbres dictate the overall form of the work. The opening A section focuses on Timbres #2 and #3, the central B section focuses on Timbre #4 before combinations of Timbres #2, #3 and #4 are presented, and the final C section focuses on Timbre #1 before combinations of all four timbres are presented. The following table specifies the allocation of the four parts within the three *M* patterns for sections A and B (both use the same timbres), and section C of the work to the above timbre numbers.

Table 4.1 *Study for Triangles,* Timbre to part allocation

Pattern Group a and b	(sections A and B)
Part 1	Timbre #2 (attack only portion of sample)
Part 2	Timbre #2 (attack only portion of sample)
Part 3	Timbre #3 (reverse sample)
Part 4	Timbre #4 (scraped sample)
Pattern Group c	(section C)
Part 1	Timbre #2 (attack only portion of sample)
Part 2	Timbre #1 (normal triangle sample)
Part 3	Timbre #3 (reverse sample)
Part 4	Timbre #4 (scraped sample)

4.2.4. Acoustic triangle parts

Instrumental parts within the *Study for Triangles* were not composed with the aid of composition software, but were notated using the software notation program *Finale* (refer to Appendix 1 for the notation of the parts and performance notes). Quasi-improvised textures within the acoustic parts are achieved with the freedom given to the performers in their tempi (MM 1/4 = 76-96 is given as a guide only), the specification in the performance notes that 'synchronisation of parts within the score is unnecessary throughout the piece' and the indication to repeat any material within a system freely.

As in the rhythmic materials used for the electronic part, the rhythmic materials in the acoustic parts are based on the numbers 3, 4 and 5. In the opening section of the work, the acoustic parts are numerically based on groups of rhythmic attacks. In each part, attack groups based on one of the numerics 3, 4 or 5 are presented in each system, the three numerics presented simultaneously by all three parts in each of the section's three systems. In the opening system the six-inch triangle presents attack groups of 5, the eight-inch triangle presents groups of 4 and the 10-inch triangle presents groups of 3. In the first system of the central (B) section, the numerics are assigned as multiples of 1/4 note durations (1/4 =

1), the numeric 3 equalling a dotted 1/2 note, the numeric 4 equalling a whole note, the numeric 5 equalling a whole note plus a 1/4 note. The remainder of the work simply applies combinations of attack groups and multiples of 1/4 note durations.

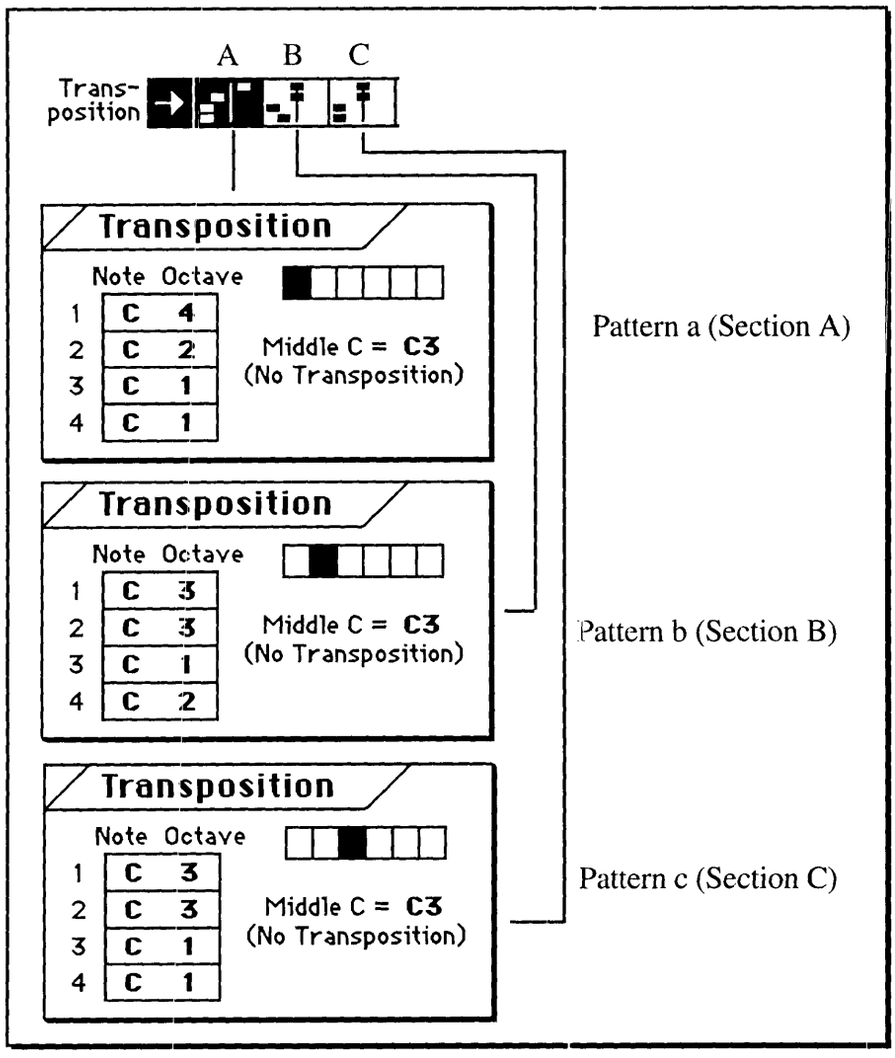
4.3. The Application of Algorithms

4.3.1. Deterministic algorithms

With the exception of Note-order and Note-density algorithms, algorithms employed within the *Study for Triangles* are deterministic, that is, where algorithms (for example the Cyclic variables) can use either random or deterministic processes, purely deterministic processes are employed. The primary algorithms in this deterministic category are the Time-base/Cyclic variables rhythm editor as described above, the Transposition algorithm, the *Legato* algorithm and the Accent algorithm. The purely musical nature of these algorithms precludes an historical account. However, in the following discussion, technical aspects of the three latter algorithms are covered, along with their applications in the context of the work.

The Transposition algorithm sets the pitch of each part in a pattern. Within this algorithm pitch is defined in relation to middle C. If a pitch pattern begins with an E, as in the first note matrix in the work, setting the Transposition algorithm to C3 (middle C) will result in the pattern beginning with E, (i.e without any transposition). Increments in the algorithm are by semitone and octave (either down or up). Throughout the *Study for Triangles* only octave increments are used so that the original pitch structures derived from the Pythagorean theorems are retained. Figure 4.8 shows the Transposition algorithm settings for the three sections of the work as they appear in miniature on the main screen, and as they appear in the variable editors.

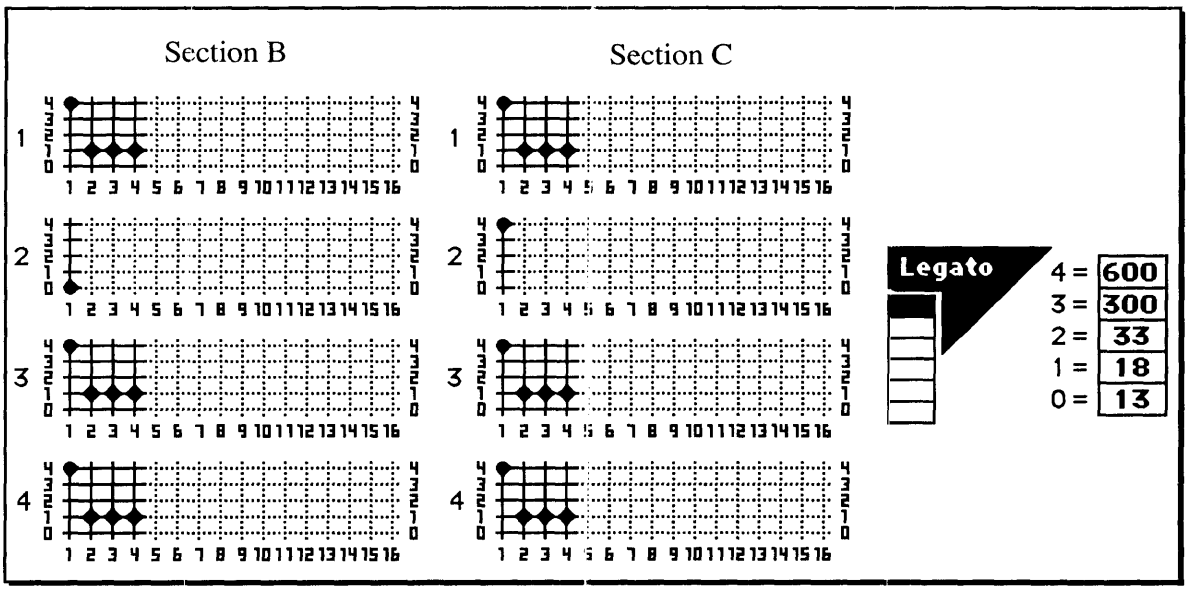
Figure 4.8 *Study for Triangles,* Transposition settings



The *Legato* algorithm implemented in *M* is similar to the Articulation-range algorithm in *Jam Factory* (see section 2.2.3), wherein percentage values indicate note lengths. Settings of 100% or more indicate that notes will be played for the full duration or longer than indicated in the Time-base setting and in the Cyclic editor rhythmic settings. Smaller percentages result in *staccato* articulations, larger percentages result in *legato* articulations. Figure 4.9 shows the *Legato* algorithm settings in the opening section of the *Study for Triangles*. Part 1, assigned the *staccato* attack portion of the normal triangle sample, has a repeated or cyclic range of small percentage settings (0 = 13%, 1 = 18%, 2 = 33%) resulting in a more *staccato* effect than that given by the original sample. Conversely, Part 2, which is assigned the same timbre, is set to 600%. While the original *staccato* effect

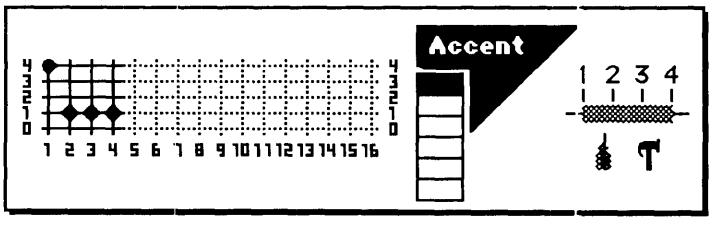
section to very *staccato* in the middle section, and the return to a very *legato* setting in the final section that coincides with a change of timbre in the part, from the attack portion of the normal triangle sample, to the gong-like transposed triangle sample. The remaining parts are all assigned the same setting (the *M* default setting), resulting in a mixture of *legato* and *staccato* articulations.

Figure 4.10 *Study for Triangles, Legato* settings, Sections B and C



The Accent algorithm in *M's* Cyclic variables provides a means of accenting musical events in a repeated cycle by increasing MIDI velocity values. Accent cycles may be of various lengths with up to 16 events, and individual cycles are applicable to any *M* part. Throughout the *Study for Triangles* the program's default setting (shown in Figure 4.11) is used in all parts. This setting indicates a strong accent on the first of every four events and weak accents on the following three.

Figure 4.11 *Study for Triangles,* Accent algorithm settings



of the sample remains unchanged with this setting, the sample is played in full, resulting in a contrast between the very *staccato* setting of Part 1 and the full *legato* sample in Part 2.

The setting of the Transposition variable has a further effect on the *legato* and *staccato* nature of the Part 1 and 2 timbre. This timbre is from a single sample assigned to middle C (i.e. when middle C is played the sample sounds as it was originally recorded). In a performance of sampled sounds 'playing different [keyboard] keys speeds up or slows down the rate the sample is played back, changing its pitch and length' (De Furia & Scacciaferro 1987, p.16). The upward transposition used in Part 1 results in a shortening of the wave form so that the sample is sped up and appears to be more *staccato*, while the downward transposition, as used in Part 2, results in a lengthening of the wave form so that the sample is slowed down and appears to be more *legato*. The setting of 600% in the *Legato* algorithm for Part 2 thus maximises the effect of the lengthening of the waveform by downward transposition, ensuring the full duration of the transposed sample is played. Parts 3 and 4, assigned the reverse and scraped samples are, in the opening section, assigned large *legato* settings (300% and 600%), similarly maximising the lengthening of waveforms on downward transposition.

Figure 4.9 *Study for Triangles, Legato* settings, Section A

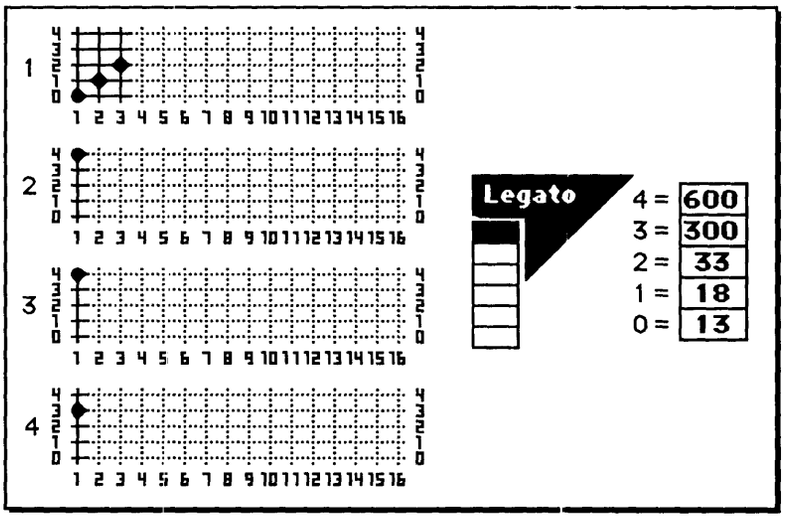
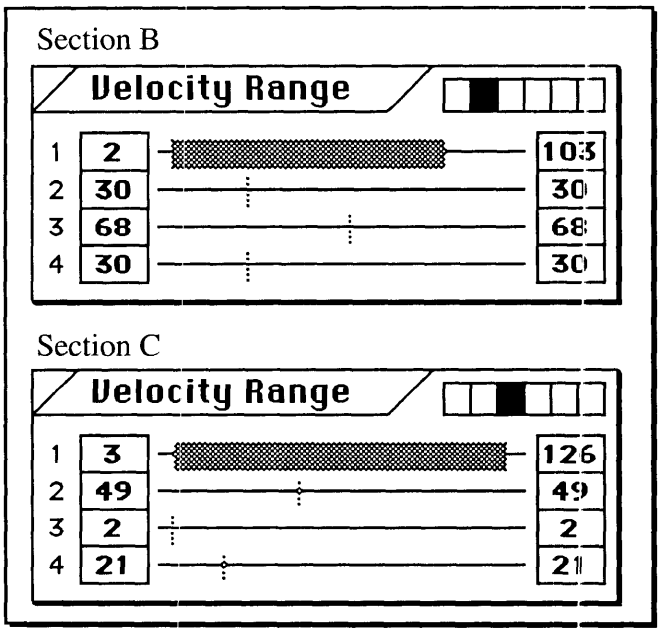


Figure 4.10 shows the *legato* settings for the middle and final sections of the work. Notable changes in the Part 2 settings are the change from very *legato* in the previous

The strength of accent is defined using *M's* Velocity-range algorithm in which a range of velocities (speeds with which, for example, a MIDI synthesiser key is struck) is set according to MIDI values from 1-127. If the full range is selected, the first of the four events in Figure 4.11 will receive the maximum velocity of 127, the remaining three events receiving a value of 1, resulting in a maximally accented first event and minimally accented following events. The Velocity-range algorithm may also be set as a single value, in which case all events receive the same accent. In the *Study for Triangles*, Part 1 is the only part that receives a range within the Velocity-range variable, set in sections B and C of the work. The remaining parts all receive a single velocity level, resulting in the same accent level for every event. Figure 4.12 shows the Velocity-range variable for the B and C sections, the Part 1 range in the B section indicating a velocity level of 103 on the first of every four events, and a level of 2 on the remaining three events.

Figure 4.12 *Study for Triangles,* Velocity algorithm settings, Sections B and C



4.3.2. Stochastic algorithms

The Note-order and Note-density algorithms within *M*, while broadly classified as stochastic, are primarily concerned with short term processes in which outcomes are unpredictable and patternless to varying degrees. As such they represent algorithms based on

random processes, as opposed to stochastic processes, which are concerned with large scale distributions of outcomes. The distinction between the two process types is further outlined in the following examination of *M's* Note-order and Note-density algorithms.

In relation to computer science, random processes relate to random numbers. In order for a computer to generate an apparently random number sequence an algorithm is used, the algorithm itself however must be pre-determined and well-defined (Moore 1990, p.408). The result of using such an algorithm is that the algorithm itself does not produce actual random numbers, but sequences of apparently random numbers, known as *pseudo-random* numbers. Numerous algorithms have been developed for the generation of pseudo-random number sequences, the first by the Hungarian mathematician John von Neumann (Shiflet 1987, p.176), and have been applied in a range of mathematical and scientific fields. 'One significant application of random numbers is in the area of simulation where the computer is used to imitate and study such occurrences as nuclear reactions or the workings of a large corporation' (Shiflet 1987, p.176).

One of the most popular techniques for generating pseudo-random numbers is known as the *linear congruential method,* introduced by D.H. Lehmer in 1948:

> The basic idea of this method is to generate the next number in a pseudo-random sequence from the last one according to a recurrence relation such as
> $X_{n+1} = (aX_n + c) \bmod m \quad n \geq 0$
> where $X_n \geq 0$ is the n th value in a sequence, $a \geq 0$ is the multiplier, $c \geq 0$ is the increment, and $m \geq X_0$ is the modulus (Moore 1990, p.409).

X_0, as an initial value (or seed number), can be derived by various methods. One common method is to use the computer system clock to return a seed number based on the time of day. Figure 4.13 shows a simple application of the linear congruential method using the time 8.00 as the seed, and the resulting, repetitive, four-number pseudo-random sequence derived from simple numbers assigned to the remaining variables. While the resulting four-number sequence in this example is limited and not particularly random, various values may be assigned to X_0, *a, c* and *m,* all of which will result in sequences of varying lengths and apparent randomness.

Figure 4.13 Linear Congruential Method

$X_0 = 8$
a = 7
c = 6
m = 10

$8 \cdot 7 = 56 + 6 = 62 \text{ " } 2(\text{mod}_{10})$
$2 \cdot 7 = 14 + 6 = 20 \text{ " } 0(\text{mod}_{10})$
$0 \cdot 7 = 0 + 6 = 0.6 \text{ " } 6(\text{mod}_{10})$
$6 \cdot 7 = 42 + 6 = 48 \text{ " } 8(\text{mod}_{10})$

8,2,0,6,8

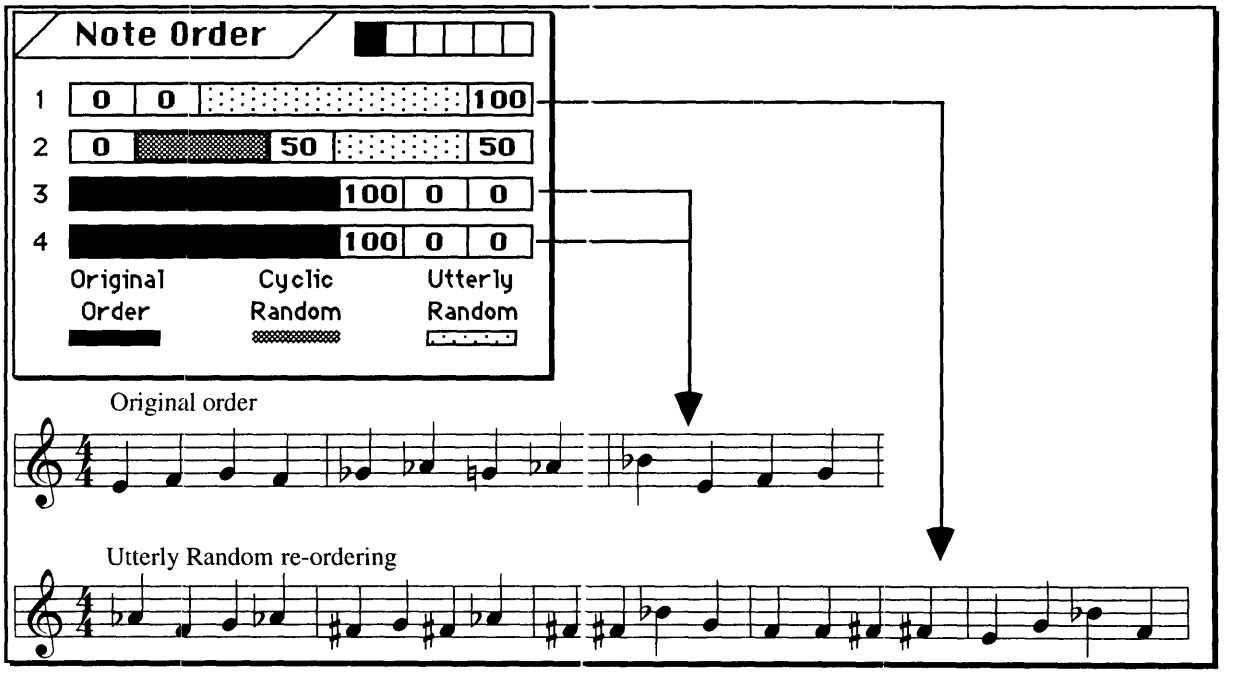
Once a process such as the above is defined for generating pseudo-random numbers, further computer algorithms provide methods for truncating and mapping generated numbers to achieve specific numeric results. As an example, for the musical parameter of pitch a sequence of numbers can be truncated to a range of 0-11, each successive number then mapped to form a melody using pitches of a chromatic scale. More sophisticated algorithms may then be used as *random sieves*, in which numbers are accepted or discarded according to predefined rules. To follow the previous pitch example, a rule $x_n \leq 2$ may be invoked, limiting the random melody to either semitone or whole tone steps. If the seed number is 5 for example, it is mapped to an F in the C chromatic scale. The following number (X_0) according to the rule must be either 6 or 7, mapped to either F# or G. Any number generated other than 6 or 7 is discarded.

Within *M's* Note-order algorithm, such processes allow the mapping of pseudo-random number sequences to pitches stored within *M's* pitch patterns. The three types of ordering within the algorithm are 'original order', where no random processes or pitch re-orderings occur, 'cyclic random', where there is a single random re-ordering of pitch patterns before the new pattern is repeated, and 'utterly random', where the stored pitch pattern is subject to continual random re-ordering. In the *Study for Triangles* random note order settings are employed in Parts 1 and 2, while Parts 3 and 4 adhere to original ordering

throughout. Part 1 in the opening section of the work employs the 'utterly random' setting. Figure 4.14. shows the Note-density algorithm settings for the opening section, the output of the pitch pattern in original order and a version of re-ordering using random processes.

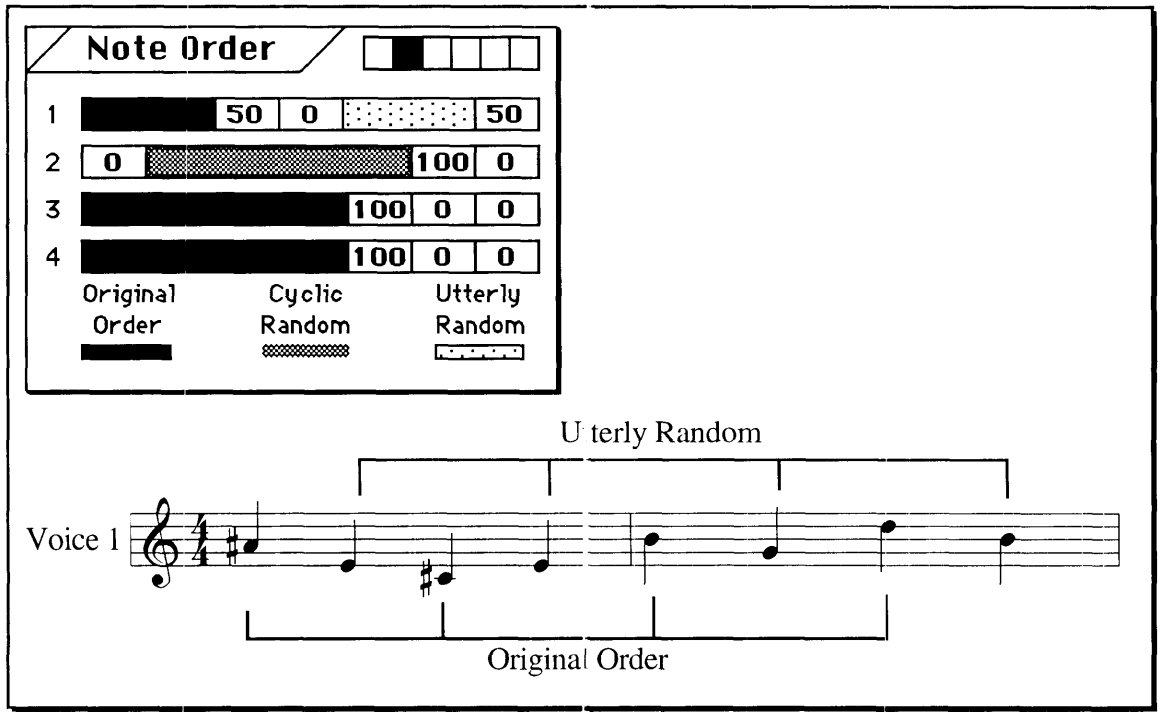
As shown in Figure 4.14, Part 2 in the opening section has a combination of 'cyclic random' and 'utterly random' settings. In relation to the nine-note series used in the opening section, the 'cyclic random' setting provides a single re-ordering of the nine notes, the re-ordered set then repeated throughout the section in combination with 'utterly random' re-orderings. The effect of the Part 1 and 2 Note-order algorithm settings within the opening section is that three different melodic versions of the same note sequence are presented, versions with varying degrees of stability, where stability is defined as note consistency by continuous repetition of a note pattern. Parts 3 and 4, with their original order setting, are the most stable, presenting the same nine-note sequence repeatedly. Part 2, with its combination of cyclic and utterly random settings, presents a less stable version of the note set, the repeated re-ordering of the set continually interrupted by 'utterly random' re-orderings. Part 1, with its 'utterly random' setting, presents the least stable version of the note set with continual re-orderings of the nine notes within the set.

Figure 4.14 *Study for Triangles*, Note-density settings, Section A, random output, Part 1



While the above processes for pitch variation are classified as random (i.e. short term processes in which outcomes are unpredictable and patternless), the possibility of combining settings within the Note-order algorithm represents a further process called a *biased choice*. Biased choices lean closer to stochastic processes in that they are concerned with larger scale distributions of outcomes, providing an ability 'to specify that some choices are more likely than others within a range of possibilities' (Moore 1990, p.418). Biased choices 'are the defining characteristic of so-called stochastic processes, especially when the biases are chosen to model the statistical behaviour of some observable phenomenon' (Moore 1990, p.418). In a biased choice process, 'the likelihood that a particular event will occur can be expressed as a probability — as the ratio of the number of occurrences of that event to the total number of results of the random process' (Dodge & Jerse 1985, p.266). In the *M* Note-order settings the range of possibilities is limited to the three types of note ordering as indicated by the three different types of variable areas on each voice's bar graph. Solid black, grey and dotted areas correspond to 'original order', 'cyclic random' and 'utterly random' algorithms respectively, while the ratio of occurrences is represented with percentages. The Part 2 settings in Figure 4.14 indicate a 50% probability that a note will be taken from either the 'cyclic random' ordering or the 'utterly random' ordering. The only other part in the *Study for Triangles* that receives combinations of note orderings is Part 1 in the central section, which employs the 16-note matrix shown in Figure 4.2. In this section, Part 1 receives settings of 50% original order and 50% 'utterly random'. Figure 4.15 illustrates a possible output, the 50% settings resulting in the replacement of alternate notes from the original order with an 'utterly random' note choice.

Figure 4.15 *Study for Triangles,* Part 1 example output, Section B



The Note-density algorithm in *M*, as previously described in Chapter Two, 'controls the percentage of random skips that occur ir playing a pattern' (Zicarelli 1987, p.20). Like the Note-order algorithm, this algorithm is controlled with a biased choice process in which the range of possibilities is the notes within a pattern, and silences. While there are essentially no actual biased choices employed in the Note-order algorithm in the *Study for Triangles* (50/50 settings preclude any bias toward one note ordering or another), settings in the Note-density algorithm do illustrate clear biased choices. Figure 4.16 shows the Note-density algorithm settings for the central and final sections of the work, the opening section is precluded due to settings of 100% throughout. Figure 4.17 illustrates the effect of the Note-density algorithm setting on Part 3 in the central section of the work. With the setting of 67%, the output, as shown in Figure 4.17, will include the original ordering of notes with a bias toward the part playing two thirds of the time, and silences for the remaining third. Over the 10 1/4 note period given in the example, this relates to seven 1/4 notes played, and three 1/4 notes resting.

Figure 4.16 *Study for Triangles,* Note-density settings, Sections B and C

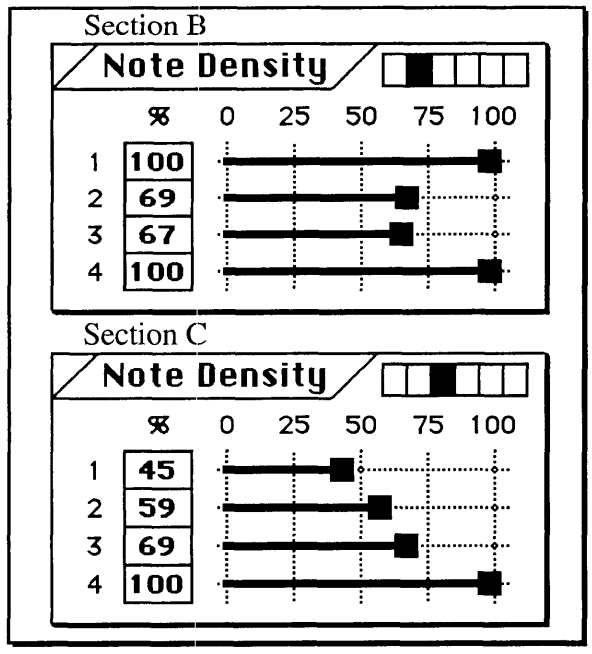
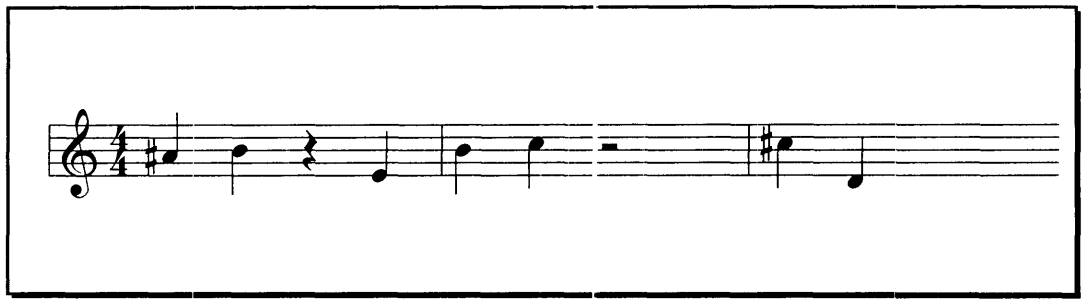


Figure 4.17 *Study for Triangles,* Example output, Part 3, Section B



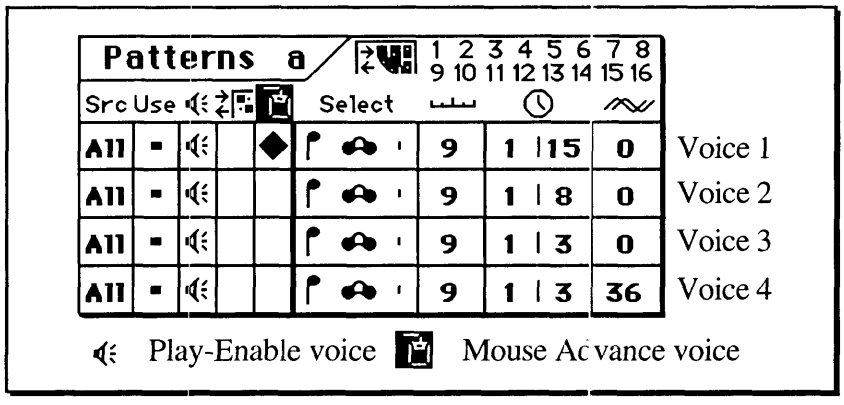
4.3.3. The application of algorithms in real-time

While most of *M's* algorithms are controllable in real-time, only two algorithms, the Velocity-range and the Note-density algorithms, are manipulated in performances of the *Study for Triangles.* Such manipulations are based entirely on textures and dynamic levels presented by the acoustic triangles, and as these parameters are variable from one performance to the next, differing degrees of algorithm manipulation similarly occur. The freedom allotted to the acoustic performers in tempo, and the repetition of scored materials, results at times in dense textures, in which case electronic parts with settings of less than 100% in the Note-density algorithm are adjusted upward to complement the acoustic texture. At other times the same freedom in the acoustic parts can result in sparse textures, in which

case the Note-density settings are decreased. Similarly, dynamic levels in the acoustic parts vary, and degrees of dynamic variation are complemented in the electronic parts by increasing or decreasing settings in the Velocity-range algorithm.

Within *M* there are on/off switches called Play-enable and Mouse-advance, both of which enable real-time control over when parts are playing. Both are represented by icons as shown in Figure 4.18. The Play-enable switch allows a part to be heard when the icon appears next to the part in the patterns window on the main screen. When the icon is absent the part is silent. This switch allows control over texture through having one or more parts playing at one time. The Mouse-advance function is activated when a black diamond appears next to the part in the patterns window of the main screen, directly below the computer mouse icon. When this function is activated the user can control the playing of a part by moving the mouse. If the mouse is kept still the part is silent. Throughout the *Study for Triangles*, Part 1 is always operated in conjunction with the Mouse-advance function, this function allowing a simple complement to varying densities of dampened triangle attacks provided in the acoustic parts.

Figure 4.18 *Study for Triangles,* Patterns window, Part-enable icons



4.4. Summary

As shown in the above examinations, the majority of algorithms in the *Study for Triangles* are deterministically employed, and relate to the implementation and manipulation of rhythms, transpositions, articulations, and velocity (dynamic) levels. The algorithms

relating to pitch control within the work are based on random procedures, allowing re-ordering of pitch either randomly, or with the more stochastic-based biased choice process.

In the interactive operation stage of the *Study for Triangles,* manipulations of the Play-enable, Mouse-advance, Note-density and Velocity-range algorithms provide the primary real-time method for interaction between the three acoustic performers and the computer operator, interactions that occur within the musical parameters of texture and dynamics. In the design stage, the remaining algorithms available in *M* provide a means of establishing materials and processes for control over the remaining musical parameters of pitch, rhythm and articulation. In this way, algorithms available in *M* provide a medium in which musical parameters may be controlled in real-time in the operation stage of the program, or may be established in the design stage and left to work as part of the program without real-time manipulation. Within the interactive environment of the *Study for Triangles*, the possibility of limiting real-time control in *M* to the Play-enable, Mouse-advance, Note-density and Velocity-range algorithms allows the computer operator sufficient space in which to concentrate on the parameter of timbre, the primary focus of the work.