GENERATIVE GRAMMARS AND THE COMPUTER-AIDED COMPOSITION OF MUSIC

Steven R. Holtzman

Ph. D.

University of Edinburgh

1980

ABSTRACT

The application of computers in music has focused almost exclusively on problems of sound synthesis. The application of computers in the process of music composition, ie. the generation of sound structures, remains largely unexplored.

This thesis describes a computer software system designed to usefully aid composers in the process of music composition by automating part of the composition process. The composition system described uses generative grammars to automate the generation of music structures.

The core of the system described is two facilities. These are 1) a facility for formally and explicitly defining the grammars of music languages, ie. the GGDL programming language, and 2) a facility for using GGDL language definitions to automatically generate utterances in the specified languages, ie. the GGDL-Generator.

An implementation of these facilities has been integrated with programs to enable sound synthesis and the graphic editing of music structures. The system, implemented on a network of computers at the Department of Computer Science, Unversity of Edinburgh, is described.

The thesis presents and evaluates some of the practical results obtained using the GGDL computer aided composition system. It is shown how the system may be used to compose macro- and micro-sound structures. An automated digital sound synthesis instrument developed using generative grammars is also described.

ACKNOWLEDGEMENT

I should like to thank R. Thonnes for his advice concerning the implementation of work reported in this thesis and Professor Sidney Michaelson for his helpful comments on a draft of this thesis. Special thanks are due to my supervisors Jeff Tansley and G. M. Koenig for their support and encouragement, and to Professor Bernard Meltzer for initially encouraging me to undertake the research reported in this thesis.

The work reported in this thesis was carried out between October 1977 and December 1978 in the Department of Artificial Intelligence, University of Edinburgh, and between January 1979 and September 1980 in the Department of Computer Science, University of Edinburgh. The work was supported by a University of Edinburgh Studentship.

Some of the work presented in this thesis has also been reported in a number of publications. These are listed in the bibliography: Holtzman 1978, 1978b, 1979, 1980, 1980b, 1980c, 1980d.

DECLARATION

I declare that this thesis has been composed by myself and that the work reported is my own.

CONTENTS

CHAPTER 1: Introduction

At present, the application of computers in music has focused almost exclusively on sound synthesis. Research in computer music has mostly been concerned with the development of hardware and software models for the synthesis of sound [1] and the description of instrumental and instrumental-like timbres using these models. [2] The application of computers in the process of music composition, that is, the generation of sound structures, has, for the most part, been neglected. With the exception of the work of a few composers, notably Iannis Xenakis and G. M. Koenig, this area of computer music research remains largely unexplored.

This is possibly due to the greater subtlety of the problem. Whereas in synthesis, the machine is used as a powerful calculator using algorithms based on formal acoustic theory which are explicit and straight-forwardly programmed; for composition it is a great problem even to formalise the rules of a music language. Indeed, though composers of very different compositional technique can describe instruments to perform their music using the same acoustic model, such as the widely used frequency modulation synthesis model, each would require the formalisation of different composition languages to use a computer as an aid for composition

---

1) For example, Fourier synthesis, Frequency Modulation Synthesis (Chowning 1973), Vosim synthesis (Kaegi 1978).

2) Research in synthesising instrumental timbres includes J. C. Risset's (1966, 1968) extensive work using frequency modulation synthesis, J. Beauchamp's (1979) and D. Morrill's (1976) synthesis of trumpet timbres, and J. Gray's (1975) 'exploration of musical timbre'.

in their style. Programs for sound synthesis, such as Mathews' (1969) MUSIC V, have been widely implemented and are used by many composers, though to date, computer programs for music composition have been designed by individual composers to compose in their particular style or music language. A composer interested in computer composition has been required to design and implement what would most likely be a large and sophisticated software package in order to use a computer as an aid in the process of composition.

An Aid for the Generation of Music Structures

This thesis describes a computer software system designed to usefully aid composers in the process of music composition by automating part of the composition process. Recent years have seen the development of software systems to aid designers, ie. computer aided design (CAD) systems, in a variety of disciplines - eg. architecture, electronic circuit layout, VLSI design. Like a CAD system, a "CAC" (Computer Aided Composition) system should act as a tool to facilitate the 'design' of large music structures. Just as a CAD system may permit the definition of building blocks, such as 'cells' for VLSI, and provide facilities for defining structures built from these blocks, a CAC system should permit the definition of a variety of musical building blocks and provide facilities for the construction of larger structures from these minimal music units. These facilities should, of course, be designed around the types of structures that are likely to be defined when composing music.

In trying to determine whether computers can be a useful aid to composers for composing music, it is first necessary to consider what functions a computer might usefully perform to help a composer. Since the turn of the century there has been an increasing formalisation of the composition process. This was, for example, manifest in serialism and the music of the Viennese school - Schoenberg, Berg and Webern. In the post-war era, the most significant influence on young European composers was certainly that of the serialist movement, and, in particular, the ideas of Anton Webern (Die Reihe 1955). In the Cologne school (Stockhausen, Koenig, Kagel, Ligeti) of composition in the '50s, much of the composition process had become a process of applying explicitly defined and well formalised rules, the rules of serialism, to the generation and manipulation of music structures. It was perhaps in some of the serial music of the French composer Pierre Boulez that explicit definition and deterministic control of the composition process reached its culmination, eg. "Structure 1A" (Boulez 1955). This trend towards formalisation is also apparent in the music of Iannis Xenakis. Xenakis rejected serialism arguing that the correspondance between the rules of serial composition and the compositions generated was inaudible, from which he concluded that serial technique was invalid (Xenakis 1955). As an alternative, Xenakis formulated other rules for the generation of music structures. These rules, which he called 'stochastic composition rules' (Xenakis 1971, 1971b) were equally explicit and formalised.

The significant point to be taken for the design of a CAC

system is that, for many contemporary composers at least, the process of music composition can be formalised and explicitly defined. A computer scientist might be led to conclude that if the rules of the composition process can be sufficiently formalised, then part of the composition process can be programmed and automated.

Looking more closely at these formalised processes of composing, they are processes in which composers are defining the syntactic rules of a music language and then generating music structures by applying these syntactic structuring rules. A composer may possibly apply these rules a large number of times to generate complex structures. A useful aid to composers would be to automate the laborious process of applying these syntactic structuring rules to generate compositions. The process of composition then becomes one of defining a set of compositional rules which the computer then automatically applies to generate compositions. The role of the composer becomes that of 'selector': he selects and defines a music language, and then may select from among the structures generated from that language definition.

In fact, during the past 25 years, a number of composers have programmed compositional rules to automate their composition process. Perhaps the first composition realised in this manner was L. Hiller's (1957) "Illiac Suite for string quartet", using random composition procedures programmed on an Illiac computer. Since 1962, Xenakis has used composing programs based on his

stochastic composition principles to compose a number of pieces (Xenakis 1971). Also notable, G. M. Koenig has, since 1963, developed composing programs based on his compositional ideas (Koenig 1970, 1970b).

These composition systems are systems where a specific set of rules has been programmed and the process of composition in a specific language has been automated. However, the rules available are limited and it is unlikely that more than a few composers will be able to share such a specific set of rules for composition. Each composer using such an automated system will most likely want to specify different types of rules.

This thesis describes a flexible CAC system designed to automate part of the composition process. The system has been designed to permit the specification of a large number of different types of compositional rules. Unlike other programs that have been designed to automate the composition process, this is the first system which provides a powerful facility for a composer to define an arbitrary set of compositional rules which can then be used as the basis for automatic composition. A composer may define an arbitrary compositional language, and the computer will then automatically generate utterances in the defined language.

Generative Grammars and Automated Composition

The composition system described in this thesis uses generative grammars (Chomsky 1957) to automate the generation of music structures. An obvious basis for a general rule based system for language generation is grammars. However, though it is fairly clear that generative grammars may be used for the generation of (at least some types of) music, it is not obvious that using generative grammars for the generation of music might be helpful to composers. With reference to results derived by implementing a CAC system based on generative grammars, it is shown in this thesis that generative grammars can be a useful aid to composers in the composition process. In addition to being an aid to composers, it is proposed that such a system could be a valuable aid for musicological and Artificial Intelligence research concerned with the representation and modelling of composition processes.

The formal description of musical processes has been a part of music theory for centuries. More recently the use of grammars for the purpose of music analysis is found in the work of Ruwet (1972), Nattiez (1975), Laske (1972,1973), Winograd (1968) and others. A survey of grammars applied to music studies may be found in Roads (1979). However, though a number of composers have automated their composition processes by programming their compositional rules, the composition system described in this thesis is the first implemented system to use grammars for the generation of music structures.

The basis of the CAC system described in this thesis is the Generative Grammar Definition Language (GGDL) compiler. GGDL is a language with which one can formally define the components of a generative grammar with sufficient explicitness that the definition may be used to drive a mechanism which will generate automatically utterances in the language. A formal specification of GGDL was begun in 1977, and prototype composing systems using generative grammars were implemented for designing the 'automated non-standard digital sound synthesis instrument' (Holtzman 1978b, 1979) and for the composition of an electronic composition, 'After Artuad' (Holtzman 1978, 1978c). In an independent manner, a language similar to GGDL was proposed by C. Roads (1978). However, his 'Tree' is a considerably less powerful than GGDL and has never been implemented.

The basis of GGDL is linguistic. Much work has been done in the field of linguistics for formally representing language processes. This is not to say that the rules one actually defines to describe a music language will be similar to those which might describe some other natural or formal language; rather, linguistic experience with representing language processes is exploited to provide a versatile 'language definition language'. The linguistic facilities for representing language rules, ie. grammars, by rewrite or production rules has been enhanced with certain features which are especially convenient for describing common composition processes. It may still be the case that it is extremely difficult to represent certain types of composition processes with the rules provided.

Though GGDL is particularly designed for defining grammars of musical languages as an aid to the composition process, because music is approached as a formal system of relationships between sound objects (Holtzman 1978) GGDL would also be suitable for defining other 'structural' or formal languages. For example, a set of phonological rules could be defined to produce appropriate phonetic data for a speech synthesiser.

The GGDL-CAC System

At an extremely general level, one could perhaps say that, in the process of composition, most composers generate some sort of working material, fragments of a musical structure, and then evaluate them, rework them and build larger musical structures from these smaller components. A computer system to aid a composer in this process may help with the generation of musical structures, as this thesis proposes the GGDL grammar system may do; but also should facilitate the evaluation of generated structures. For composers to actually use such a system it would need to be interactive and, ideally, allow the material generated using the system to be quickly and easily evaluated, for example, as sound or in the form of a score.

The generative grammar system, therefore, should be seen as one part of a suite of programs to aid a composer. Generating structures, other programs might, for example, permit the easy inspection of the results. Programs for synthesis, score editors (Smith 1972, Buxton, et al, 1979), and so on, could complete the

suite. The GGDL composition system implemented at Edinburgh has been integrated with programs to enable sound synthesis and the graphic editing of music structures. The complete suite of programs, referred to as the GGDL-CAC system, has been implemented on a network of computers; a VAX 11/780 supports the GGDL composition software for the generation of music structures and a graphic editor, and a PDP-15/40 equipped with the appropriate hardware is used for the performance, ie. synthesis, of music structures.

In addition to integrating the GGDL composition software with other composition aids in the implementation described in this thesis, the GGDL composition software system has been designed in such a way as to permit a flexible interface with other programs. A mapping process in the act of generating compositions permits the definition of an output in an arbitrary format independent of the actual generation of a composition's structure. Thus, given a set of compositional rules, one can generate output compatible with different synthesisers, score editors or other alpha-numeric representations of the score.

Composition with the GGDL-CAC System

Results obtained using the implementation of the GGDL-CAC system demonstrate that generative grammars can usefully automate the composition process to aid composers. The GGDL-CAC system has been used by several composers to generate music structures. Notably, D. Hamilton used the system over a period of two months

to realise a BBC commission, and G. M. Koenig experimented with GGDL to help him conclude whether grammars could be a useful way of automating the composition process. The author of this thesis also has used the GGDL-CAC system for the generation of a number of music compositions.

In addition to demonstrating that the system can be used to generate 'macro-compositions', it is shown that the system can be used to automate the composition of 'micro-sound structures'. By macro-composition is meant the generation of sound structures described in terms of complex sound-objects, such as notes or noises. By the composition of micro-sound structures is meant the description of sound structures in terms of the minimal units of a sound description, such as samples or synthesis parameters. The objects manipulated when composing at the micro-level are not themselves sounds, but, when used to form a complex structure of such objects, may define a sound. The generation of macro-music structures such as those discussed in Chapter 4 is, in a sense, what one would expect, or what one might minimally want from a composing program. In Chapter 5, it is shown that using GGDL, a composer, as he may define the objects he wishes to compose with, may use the system with equal facility to compose at the micro-level.

Though Koenig and Xenakis have experimented with using their composition principles to organise micro-sound structures, they designed their programs in such a way that it was necessary to write different composing programs to apply the same rules to

different objects. Applying computer science design principles of modularity and exploiting concepts developed in generative linguistics, GGDL is designed to permit the definition of compositional rules independent of the objects that are to be used to realise structures generated with the defined rules. The flexibility of GGDL permits a composer to define not only an arbitrary set of compositional rules, but also an arbitrary set of elements to be composed with. No other composing programs permit such flexibility. The advantages of such flexibility are demonstrated in the examples of Chapters 4 and 5.

The thesis also describes what is called the 'automated non-standard digital sound synthesis instrument' (Holtzman 1978b). This instrument is the basis for an innovative approach to digital sound synthesis and the composition of sounds and was developed as a direct consequence of using generative grammars to automate the composition process. It demonstrates that the use of generative grammars and an automated CAC system may, in addition to facilitating composers in the composition process, open new possibilities in composition that could not have been arrived at without such a system.

Summary

In summary, this thesis investigates whether it is possible to automate parts of the composition process to usefully aid composers. It is suggested that generative grammars could be used

to automate the process of the generation of music structures. However, though it is clear that music may be generated with grammars, it is not obvious that generative grammars will be able to usefully aid composers. A computer aided composition system based on generative grammars was designed and implemented to investigate if, in fact, generative grammars could usefully aid composers by automating part of the composition process. In addition to the automatic composing system, the system includes a suite of programs to permit the inspection and evaluation of generated music structures. Presenting results obtained using the system, it is shown that generative grammars can usefully aid composers by automating part of the composition process.

The thesis begins with a review of research work concerned with computer composition. The basis of the CAC system described in this thesis, that is, the GGDL programming language, is then described.

The thesis presents and evaluates some of the practical results obtained using the GGDL computer aided composition system. In Chapter 4, Steve Reich's 'Clapping Music' (Reich 1972), Schoenberg's TRIO from the Piano Suite, Op. 25 (Schoenberg 1925), and David Hamilton's 'Four Canons' are used to demonstrate how GGDL might be used for describing and generating complex compositions. Though these examples demonstrate the considerable power of GGDL and the possible sophistication of results, these sort of results are, in a sense, what one would expect from a CAC system. However, the facility in GGDL to define arbitrary

compositional objects makes it equally possible to compose micro-sound structures and examples of composing sounds are presented in Chapter 5. In addition, using GGDL and concepts developed for composing with such a system, eg. conceiving of music as a hierarchical system which may be described in terms of formal relationships between objects, an innovative approach to digital sound synthesis and the composition of sounds was developed in what is called an 'automated non-standard digital synthesis instrument'. This 'automated instrument' is described in Chapter 6. It is perhaps these new possibilities that will be of most interest in computer aided composition.

These examples of using generative grammars for the description and generation of music compositions are followed in Chapter 7 with a description of the implementation of the GGDL-CAC system. In Chapter 8, the conclusions that may be drawn from the research reported in this thesis are discussed and possible directions for further work considered.

CHAPTER 2: Review of the Literature

In this chapter, related work in automating the composition process is discussed. In particular the composition programs of Iannis Xenakis and G. M. Koenig are described and compared to the GGDL composition system. A language similar to GGDL proposed by C. Roads is also discussed. Roads independently proposed using grammars for describing and automating the composition process. It is shown that the CAC system described in this system provides a facility which is not available in any other CAC systems, by which a composer may automate the composition process. The approach described in this thesis to the problem in terms of generative grammars is also shown to be original.

"Computer Aided Composition"

W. Buxton's (1977) paper, "A Composer's Introduction to Computer Music", surveys computer music. Apart from synthesis programs, he discusses what he calls 'composing programs' and 'computer-aided composition programs'. As examples of the former, he refers to programs developed by Hiller (1959), Xenakis (1971) and Koenig (1971,1971b). The brevity of the list indicates how little research has been done in automated composition. Buxton refers to several programs as computer-aided composition programs: Score (Smith 1972), Musicomp (Tanner 1972), Groove (Mathews et al 1970), and the POD programs (Truax 1973). Buxton's (Buxton et al 1978) more recent Structured Sound Synthesis Project would also belong to the category of 'computer aided composition programs'.

These programs are essentially score editing systems that permit composers to define, edit and manipulate representations of music structures and, in most cases, listen to and possibly view the results.

In this thesis, 'computer-aided composition' (CAC) is used to refer to any facilities which may aid a composer in the process of composition. CAC facilities therefore are understood to include both 'composing programs' and 'computer-aided composition programs' as distinguished by Buxton. Both aid composers in the process of composition. It is felt, rather, that the fundamental difference between these two types of programs is that 'composing programs' are automated compositional aids, and that Score, Musicomp, etc. are non-automated compositional aids.

Another difference between these two types of programs in the case of the programs cited by Buxton, is that the 'composing programs' have been designed by individual composers primarily as aids to themselves for composing in their particular styles, whilst the various music editing programs are intended as general facilities to aid composers of possibly very different compositional method and style. However, this distinction is not a necessary difference. It just so happens that none of the automated compositional aids have been designed for general application. The GGDL-CAC system fills this gap. It is an automated compositional aid intended for general use.

Hiller's "Illiac Suite"

In 1957 Hiller composed the "Illiac Suite" for string quartet, using data generated by an Illiac computer (Hiller 1959). To generate the data, Hiller programmed the machine to generate randomly distributed notes and durations. In 1963, Hiller and Robert Baker developed a number of composition routines which formed the basis of Musicomp (Hiller 1969). Hiller has used such composing routines to write several pieces since the "Illiac Suite". These routines are mostly based on 'stochastic' processes which are embodied in the composition techniques of Iannis Xenakis.

The ST-programs of Iannis Xenakis

In 1954 Xenakis originated what he called 'stochastic music', music composed by means of formalised composition techniques 'largely based on mathematics and especially the theory of probability' (Xenakis 1971). Using such techniques, musical 'textures' are described by probability distributions of 'sonic events'. For example, 'the composition of the orchestra could be stochastically conceived...during a sequence of a given duration it may happen that we have 80% pizzicati, 10% percussion, 7% keyboard, and 3% flute class' (Xenakis 1971). Stochastic techniques could also be used to generate a number of pitches over a range of frequencies with a given probability distribution, durations could be distributed over time, and so on.

Xenakis has composed many pieces using such techniques. In one of his early stochastic compositions, "Syrmos" (Xenakis 1959), Xenakis described eight different textures in terms of probability distributions. Written for string orchestra, these textures were, for example, descending bowed glissandi, pizzicato clouds, and 'atmospheres' made up of notes struck col legno. In his compositions, Xenakis calls these textures 'screens' and uses Markov chains to describe transition probabilities from one screen to another. A matrix defining the transition probabilities between eight textures is given in Figure 2-1.

Xenakis programmed his rules in a series of programs known as the 'ST', ie. STochastic, programs. However, his formalised composition techniques and the use of mathematics in the process of composition were not a consequence of computers.

> "Computers are not really responsible for the introduction of mathematics into music; rather it is mathematics that makes use of the computer in composition...the advantages of using electronic computers in musical composition (are) 1) the long laborious calculation by hand is reduced to nothing 2) freed from tedious calculations the composer is able to devote himself to the general problems that the new musical form (ie. that described by a set of rules) poses and to explore the nooks and crannies of this form while modifying the values of the input data" (Xenakis 1971).

Xenakis also suggested that, using computers, techniques used to compose macro-compositions could also be applied to the composition of micro-sound structures. To investigate this, he implemented a program to calculate waveforms using stochastic techniques.

> "Solutions in macro-composition can engender simpler and more powerful new perspectives in the shaping of micro-sounds than the usual trigonometric (periodic)

	A	B	C	D	E	F	G	H
A	0.021	0.357	0.084	0.189	0.165	0.204	0.408	0.096
B	0.084	0.089	0.076	0.126	0.150	0.136	0.072	0.144
C	0.084	0.323	0.021	0.126	0.150	0.036	0.272	0.144
D	0.336	0.081	0.019	0.084	0.135	0.024	0.048	0.216
E	0.019	0.063	0.336	0.171	0.110	0.306	0.120	0.064
F	0.076	0.016	0.304	0.114	0.100	0.204	0.018	0.096
G	0.076	0.057	0.084	0.114	0.100	0.054	0.068	0.096
H	0.304	0.014	0.076	0.076	0.090	0.036	0.012	0.144

Figure 2-1: A Xenakis 'matrix of transition probabilities (MTP)' (Xenakis 1971 p. 89). Eight 'screens', each with a transition row, are represented by A,B,C,D,E,F,G,H.

functions can. Therefore, in considering clouds of points and their distribution over a pressure-time plane, we can bypass the heavy harmonic analyses and syntheses and create sounds that have never before existed. Only then will sound synthesis by computers and digital-to-analogue converters find its true position" (Xenakis 1971).

The Composition Programs of G. M. Koenig

Koenig was, in the 50's, a leading member of the Cologne school of composition. It was during this time that Koenig formulated the concept of what he was later to refer to as 'programmed music', influenced by the serial and formal composition techniques that were then prevalent. Koenig explains programmed music as music composed by following a set of formal and explicit instructions (Koenig 1971, 1978). The process of composition consists of defining a set of compositional rules and then applying the rules to generate a composition. The realisation of a composition, given a set of compositional rules, could be done by anyone, or anything, capable of following the formalised instructions.

The 'programme' by which a work is composed is not necessarily a computer program. For example, the score for his composition, "Essay, composition for electronic sounds" (Koenig 1960) written in 1957, is a 'programme'. It consists of all the instructions required to realise the composition. What is to be recorded, what length tapes should be cut, how they should be ordered, and so on. Given the programme, anyone could follow the instructions to realise the composition.

It was not until some years after Koenig formulated the idea of programmed music that he began research into programming compositional rules. It was in fact his idea of programmed music that led him to using computers.

> "Between 1957 and 1963 I composed as well as electronic pieces two piano pieces, a wind quintet, a string quartet and three orchestral pieces, applying composing methods which could one and all have been performed with the aid of a computer" (Koenig 1978).

Koenig's interest in developing composition programs is not only to generate compositions, but to 1) study the process of composition by trying to formalise and program it 2) study the consequences of 'programmed music'. Koenig has been interested in making a systematic study of 'form-potential'. Compositional rules define formal relationships which may be realised in music; these possible realisations are what Koenig calls a 'form-potential'. In terms of grammatical description, it is suggested that the 'form-potential' of a set of rules could be likened to the 'language' defined by a grammar.

Koenig has written three composition programs, 'Project 1' (Koenig 1970), 'Project 2' (Koenig 1970b), and the 'SSP' (Berg 1978, Berg et al 1979) sound synthesis program. The earliest of his three composing programs was 'Project 1'.

> "I had the idea of collating my experience with programmed music at the desk and in the electronic studio to form a model which would be almost fully automatic. Faithful to the fundamentals of the nineteen-fifties, all the parameters involved were supposed to have at least one common characteristic; for this I chose the pair of terms 'regular/irregular'. 'Regular' means here that a selected parameter value is frequently repeated: this results in groups with similar rhythms, octave registers or loudness, similar harmonic structure or similar sonorities.

> 'Irregularity' means that a selected parameter value cannot be repeated until all or at least many values of this parameter have had a turn. The choice of parameter values and group quantities was left to chance, as was the question of the place a given parameter should occupy in the range between regularity and irregularity. A composer using this program only has to fix metronome tempi, rhythmic values and the length of the composition...all details are generated by the automism of the program" (Koenig 1978).

'Project 1' was designed as a means of investigating periodicity (regularity) and aperiodicity (irregularity) in music structure.

Whereas 'Project 1' offers a composer little opportunity to influence the structures generated by the composing program, 'Project 2' permits a composer to influence the way in which structures are generated.

> "On the one hand, the user is expected to supply a lot of input data not only defining the value-ranges in eight parameters but also making the parameters interdependent; on the other hand, the individual decisions within the form-sections (that the program generates) are not made to depend on chance, as in Project 1, but on selection mechanisms specified by the composer" (Koenig 1970b).

In 'Project 2', Koenig defined five selection mechanisms, or procedures, each based on a selection principle. The user could not actually define selection procedures, but could specify which selection procedures and what parameters would be used for selections.

The five selection procedures in the 'Project 2' program are 'alea', 'series', 'ratio', 'tendency', and 'group'. These selection procedures permit the selection of elements from a defined set according to specified rules. 'Alea' does this randomly; 'Series' does this ensuring that no element is selected a second time until all other possible selections have been chosen

at least once, in accordance with the traditional serial selection principle; 'ratio' uses a specified weighting for each element to bias its selection; 'tendency' permits the specification of a dynamic mask over the elements; 'group' selects an element from the specified set and then repeats that selection a specified number of times.

In 1963, Koenig suggested that "the question arises as to how instrumental experience in macro-time could be transferred to micro-time" (Koenig 1963). Serial composers producing electronic music in Cologne wanted to unify the macrostructure of a composition with the microstructure of the sound(s) from which it was composed. The same principles should produce both. This was the basis, for example, of Koenig's (1960) "Essay". The sounds do not result from a preconceived acoustic idea but rather from serial manipulation of basic material specified in accordance with the overall form of the piece.

Koenig therefore designed a computer sound synthesis program (SSP) (Berg 1979) that permits the application of his 'selection principles' to manipulate samples. Samples may be organised to form 'segments', and, in turn, using the same selection principles, these segments may be manipulated to form larger sound structures.

Grammars and automatic composition

Both Xenakis and Koenig turned to computers as an aid in the composition process. The computer programs they designed to automate part of their composition process were designed around particular concepts of music composition. Neither the ST-programs nor Koenig's programs permit a user to define his own compositional rules. In the ST-programs, variables can be changed by a composer to alter the probability distribution of components of a sound texture. In Koenig's 'Project 1', a composer has little control over the output that the program will generate though, in 'Project 2' and the 'SSP' program, the composer can program the generation of utterances using a limited virtual machine instruction repetoire.

The CAC system described in this thesis is designed to automate the generation of musical structures whilst permitting a composer to specify his own compositional rules. This is the first system which provides a specially designed facility, the Generative Grammar Definition Language, for a composer to define an arbitrary set of compositional rules which can then be used as the basis for automatic composition. The basis of the 'language definition language' is the 'rewrite rule' (cf. 3.1.1), a concise method of describing language constructs that has been widely used in linguistics and formal language research. Rewrite rules with various properties may be used to describe different types of languages.

Language systems using rewrite rules (or similar formalisms such as Backaus-Naur Form (BNF)) and implemented on computers have been designed for the purpose of defining the syntax of a language and have also been used for recognising, or parsing, language constructs. In certain respects, when implemented, language generation poses problems that are not encountered in parsing. For example, a problem in generation not encountered in parsing is how, given a number of possible alternatives that may be generated, selection between them is to be made. That is to say, in parsing, the right-hand side alternative which actually matches the string being parsed, is determined by the string; in generation, one of the possible right-hand alternatives must be selected by the generative mechanism.

In Xenakis' ST-programs, the matrices of transition probabilities and probability distributions, and in Koenig's programs, the selection procedures, are used to select one element from a number of possible selections at any given time. The GGDL automatic composition system provides some system procedures similar to those available in the ST-programs (cf. Finite-State Rewrite Rules, 3.1.213) and 'Project 2' and 'SSP' (cf. 'Blocked Generation', 3.1.212). These system procedures are provided because they are widely used selection techniques in music composition. However, it is not possible to predict all the selection procedures that composers may wish to use. Therefore, in addition to permitting a composer to specify the syntax of a compositional language in the form of rewrite rules, a simple high-level programming language is provided with which a composer

may specify his own selection procedures. These selection procedures may then easily be integrated into the rewrite rule specification of the syntax of a compositional language.

Though grammars have recently been used for music analysis, their use for the specification of generative mechanisms for composition has not been explored. The CAC system described in this thesis is the first implemented system to use grammars as the basis for specifying mechanisms for the generation of music structures. However, at the same time as the GGDL language was specified, C. Roads independently described a language called 'Tree', also based on grammars, for the specification of generative mechanisms for music.

In 'Composing Grammars', Roads (1978) suggests that grammars could be a useful way of specifying the syntax of a music language.

> "If it is the task of one kind of composition to structure the syntax of a lexicon of sound objects, it is natural to think of clarifying and extending this process by means of a notation system and with the aid of a computer program. The structure of music expressions, and of the grammar behind them, can be described by means of concepts and notation developed in formal language theory."

After a discussion of the use of grammars in music analysis, Roads proposes a language for the specification of a music language that may then be used for the generation of compositions. He proposes that the language should permit the specification of syntax using context-free rewrite rules. He also suggests that the rewrite rules should permit the specification of more than one right-hand side production for a given left-hand side and that a 'control

procedure' could arbitrate between the alternatives available. Such a control procedure, he suggests, could be written in a high-level language.

Though Roads' 'Tree' shares a number of features in common with GGDL, for example, the use of rewrite rules and selection or 'control' procedures, 'Tree' is considerably less powerful than GGDL and has not been implemented. In addition to context-free rewrite rules, GGDL also permits the use of context-sensitive rewrite rules and includes a transformational processing stage in generation (ie. properties associated with Type 1 and Free Grammars). It is, however, with reference to 'control' procedures that Roads' specification is extremely vague, which is probably due to the fact that 'Tree' remains an unimplemented language. Though Roads suggests that 'control' procedures could arbitrate between alternative choices, he does not clearly specify how this is to be done. GGDL, in addition to providing system procedures for selecting between alternatives, permits the definition of functions in a high-level programming language. During the process of rewriting a function may be called and the result it returns may be used as an index to the possible selections.

Other limitations of Xenakis' and Koenig's programs are that the objects composed with in the programs, and the format in which output is generated, are fixed. Though both Xenakis and Koenig have experimented in applying their compositional techniques for generating micro-sound structures, in both cases composed of samples, they have had to design new programs for this purpose as,

in their implementations, the process of structure generation and the objects in terms of which the structure is defined are closely dependent on one another. GGDL is designed to permit the definition of compositional rules independent of the objects that are used to realise structures generated with the defined rules. This approach is based on a distinction made in generative linguistics between the generation of abstract structures and their mapping or morphological realisation. Thus, in the GGDL generation process, there is a structure generation process and a mapping process. There is also an intermediate transformational process. Similarly, Roads, in his description of 'Tree', proposes a distinct 'lexical mapping' stage. However, whereas with GGDL it is possible to generate a structure and then use different mapping definitions to map the same structure to different formats using different mapping definitions, it is not clear in Roads' description of 'Tree' whether one can actually separate these processes in such a manner. Nor does Roads' 'Tree' include a Transformational processing stage.

Summary

In summary, automatic composition programs have been designed around predefined compositional rules and have thereby been limited in their generality. They have also been defined in terms of a specified compositional object - ie. notes or samples. GGDL is a system which permits the specification of composition rules and may therefore be used as a compositional aid with more general application. Furthermore, by introducing modularity into the

generative design, GGDL may be used to generate structures independent of how they are to be realised. That is, the compositional object and the process of composition may be specified independently permitting the same compositional rules to be used to compose with different objects, or, vice-versa, the same objects to be used with different compositional rules. The GGDL-CAC facility described in this thesis provides facilities that may be used by a composer to automate, and thereby aid him in, the process of composition.

Chapter 3: The GGDL Generative Grammar Definition Language

A problem common to both the composer wishing to automate part of his composition process and the researcher wishing to investigate a composition process is to formally and explicitly represent that process. In this chapter, the use of grammars for the description of composition processes is discussed. In particular, the facilities available in the GGDL generative grammar definition language for defining a grammar are described.

GGDL may be used to define a Transformation Grammar with three components: a set of phrase-structure rules, a transformational component and a morphological mapping component. In the second part of this chapter, the formalism of the rewrite rule is defined and the types of grammars that may be described using rewrite rules with different properties are reviewed; the transformations available, and lastly, the process of morphological mapping, are described.

Generative Grammars

A grammar gives rules for combining the elements of a language, such as, notes, words, or noises, to form utterances (or sentences) in the language. A 'generative grammar', introduced by Chomsky (1957), is a system of explicit and formal rules with an associated lexicon. These rules are formulated in such a way that they may generate a set of sentences, that is, combinations of the lexical elements, which constitute a language. The language

defined by a set of rules consists of all the combinations of the elements that the rules can produce and only those combinations, though this is not necessarily a finite number.

GGDL is a language with which one can formally define the components of a generative grammar with sufficient explicitness that the definition may be used to drive a mechanism which will automatically generate utterances in the language.

A Transformational Grammar

A grammar may consist of a number of components. In GGDL one may define a special type of grammar, a Transformational Grammar, which consists of a set of Phrase-Structure Rules, a set of Transformational Rules, and a set of Morphological Rules. Phrase Structure Rules generate sequences of terminals (cf. 3.1.13) which are the names of morphemes; Transformation Rules may manipulate these morpheme strings based on special characters in the string which initiate structural changes - these may be called "Structural Change Markers"; Morphological Rules map the surface-structure of the morphemes, generated by the Phrase-Structure and Transformational Rules, into sound or possibly some other representation.

In GGDL one can define an arbitrary set of Phrase-Structure Rules, a set of Transformation Rules are provided by the system, and an arbitrary set of Morphological Rules may be defined.[1]

1) It is interesting to note that, whereas for spoken language

Describing Music Languages with Grammars

A traditional music score is a representation of a collection of utterances in a musical language. The language theoretically could be formally described by a set of rules. Typically, the grammar by which a musical structure is derived in the process of composition is not easily, and certainly not completely, detectable - it is oblique but may be inferred. When confronted with a machine, all of the processes involved in composition must be made explicit and be unambiguously represented. Anything left unsaid will not be taken for granted by the generator program. A composer must make explicit:

> "a knowledge needed for producing syntactically well-formed, semantically interpretable, and sonically intelligible structures...rules of the syntactic component concern the grammatical well-formedness of musical structures, rules of the semantic component determine the interpretations associable with well-formed musical strings; sonological rules, together with universal sonic constraints, determine the relationship between the syntactic-semantic structure of a music and its acoustical representation in as far as this relationship is controlled by grammatical rules" (Laske 1973).

GGDL is a language that may be used to formally represent this 'knowledge'. That is, in GGDL, one can formally and explicitly represent the syntax of a language to generate 'well-formed' musical structures; one can also define functions to control which

---

the transformational rules are different for different languages and the phrase-structure rules may possibly be common, for different music languages it is suggested that the same transformations may be used though the phrase-structure rules will differ. The musical transformations available in the transformational stage of generation in GGDL, such as inversion, retrograde and transposition, have been exploited by composers of very different music languages: Machaut, Bach, Beethoven, Bartok, Schoenberg, Messaien, Stockhausen, for example.

well-formed structures are generated, ensuring that they are not simply grammatically correct but are musically understandable to the composer; and one can formally define the relationship between the abstract representation of a structure and its realisation in a set of 'mapping rules'.

Using GGDL the generation process is divided into three distinct stages. First an abstract structure is generated representing relationships between musical objects. This process consists of the generation of utterances in a language where the language syntax is defined by rewrite rules (cf. 3.1.1) and selection procedures control the process of rewriting. In GGDL, a generative grammar definition program may define an arbitrary set of phrase-structure rules at two levels, a syntactic and a control level.

Second, given a structure generated by such rules, it then undergoes a transformation process. In this process, special characters in the generated string indicate that certain structural transformations should occur. These transformations are predefined in GGDL. The transformational process results in a string which should, at this stage of generation, consist only of terminal strings, though this is still an abstract representation of the music structure.

Third, the abstract structure is 'mapped' into a 'concrete' musical representation, such as sound or a score. This process may also be defined formally in a mapping program which consists

of instructions for mapping abstract objects to real ones.

The division of the generative process into these three stages was somewhat arbitrary. Experience in computer science has shown that such modularity, the basis of structured programming (Dahl 1972), facilitates the formalisation of programs. However, though it has been suggested that these three stages provide a useful way of modularising the process of the generation of natural languages, it is not obvious that such a division will be a useful one for the generation of music structures. This is especially so when one considers that the phrase structure rules of natural and music languages may be very different and that the notion of transformations in music is very different from that in natural language.

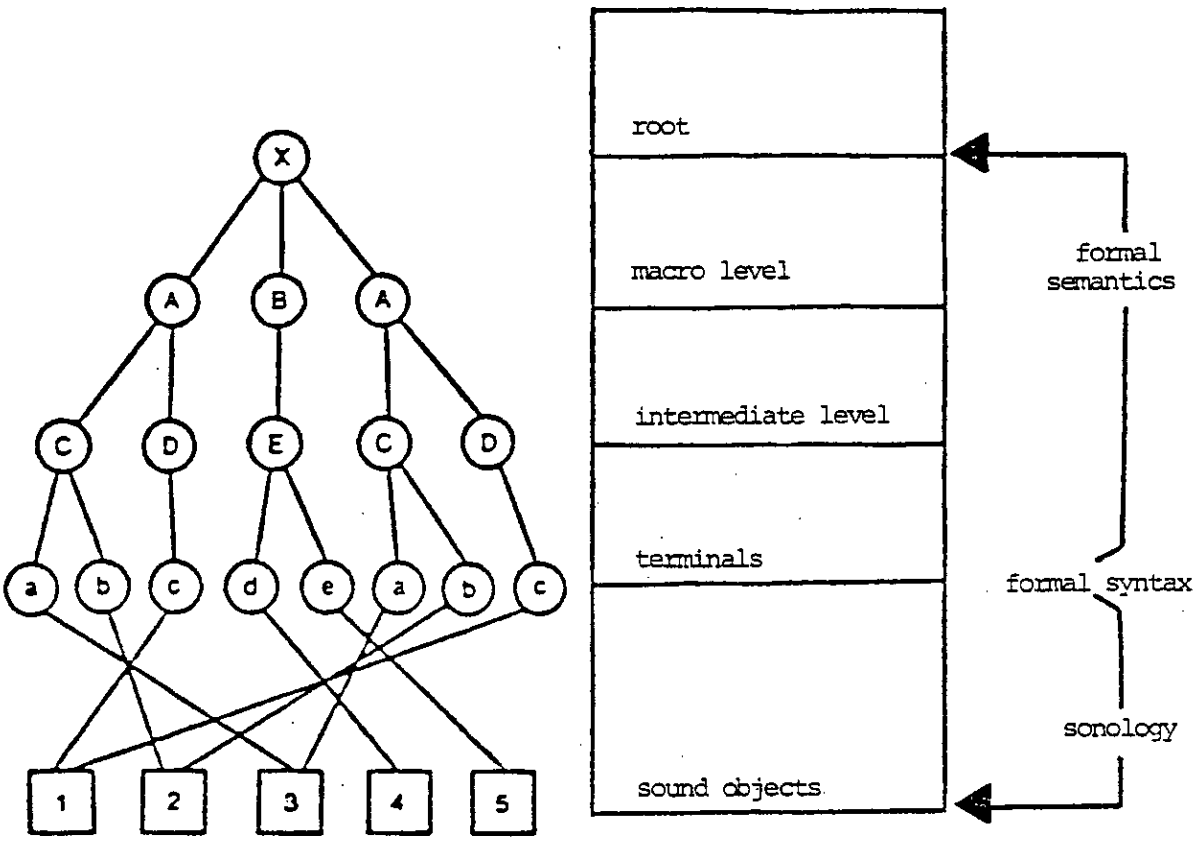
It is suggested that such a division of the generation process for music is, in fact, a useful way of modularising the process. For example, the notion of transformations in music structures is a familiar one, viz. transposition, inversion, retrograde. Furthermore, evidence from previous research in automated computer composition, ie. the programs of Xenakis and Koenig, suggested that it might be advantageous to separate the process of structure generation using a set of rules from the definition of the objects of that structure. Whereas in the work of Xenakis and Koenig, different programs were developed to apply the same rules to different compositional objects, a separation of the generation of a structure from the definition of the objects used to define it would enable one to change only one module of a language

definition without influencing other modules.

In trying to investigate whether generative grammars could be of aid to composers, it was necessary to choose a specific implementation of a generative grammar, and, using that implementation, make an assessment. Though the choice of the type of grammar to use was somewhat arbitrary, it is felt that there were reasonable grounds to suggest that the division of the generation process into the three distinct stages used would be useful and results gathered using such an implementation, presented in Chapters 4-6, support the conclusion that such a division would prove a useful one.

The phrase-structure rules of a grammatical description of a language may be thought of as manipulating only abstract objects. Objects are related only one to another (see Holtzman 1978) and are undefined except to the extent that they are related to each other. What the objects actually are, or will be, when they are mapped into, for example, sounds is completely arbitrary. The processes of structural generation and the structure's realisation are independent.

In approaching a grammatical system, the rules of the grammar may be thought of as working on two orthogonal axes which describe two different types of relationships amongst the abstract tokens which the grammar manipulates (see Figure 3-1). The relationships on one axis are known as 'syntagmatic', with the other the relationships are 'paradigmatic'. J. Lyons explains that a



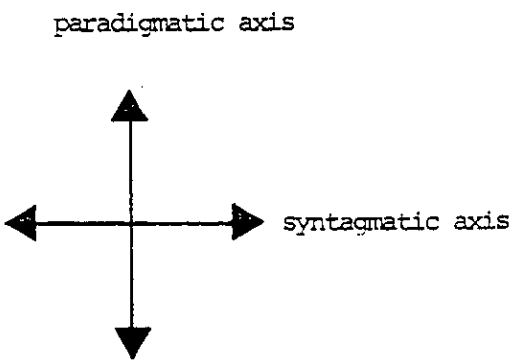


Figure 3-1: Paradigmatic and syntagmatic relations amongst the objects of a structure. (After C. Roads (1978))

linguistic unit

> "...enters into paradigmatic relations with all the units which can also occur in the same context; and it enters into syntagmatic relations with the other units of the same level with which it occurs and which constitute its context" (Lyons 1968).

Syntagmatic relations have to do with the relations of tokens at any one level in a structure; they are the result of combinations of tokens on the same level. Paradigmatic relations occur when tokens may be substituted for one another and still perform the same function in the system as a whole. Paradigmatics is the study of the derivation of a token showing the relationship between the surface and deep structures (Jakobson 1970).

These relationships may be expressed using rewrite rules. Rewrite rules may be used to define what legal sequences of terminals and non-terminals may be generated. That is, rewrite rules may be used to define syntagmatic relations. Rewrite rules may also be used to define paradigmatic relations. For example, the following rewrite rule, where "." indicates alternative right-hand side substitutions,

[A -> B . C . D]

describes a paradigmatic relation where functionally the tokens 'B', 'C' and 'D' are equivalent, all being derived from the non-terminal 'A'. This implies a hierarchical relation, as the category 'A' includes the tokens 'B', 'C', and 'D'.

The representation of syntagmatic and paradigmatic relationships in a grammatical definition are clearly separated in a GGDL definition. Syntagmatic relations are defined in the

rewrite rules, and though from the rules it is visible what paradigmatic alternatives there are for any substitution, how the choice of substitution is to be made is defined apart from the rewrite rule. The definition of the syntax of the language in the form of rewrite rules and the control of paradigmatic selections are distinguished processes in a GGDL language definition; there is a clear distinction between the rules that define a language's syntax and the procedures that control the process of generation given the syntax specification.

The separation of the specification of a language's syntax and the control of generation using the syntax definition is a natural way to separate the definition of a mechanism to generate utterances in a language. It can be likened to separating the two problems of 'what to say' at any given time, and 'how to say it'. The syntax of a language consists of a fixed set of rules which define 'how' things may be generated, ie. in what forms. However, the process of generating an utterance which conforms to the syntax is dependent on 'what' should be generated at any given time. It is dependent on the context in which it is generated. This context is possibly always changing.

In the transformational processing stage, parts of the structure may be subjected to complex transformations. The phrase-structure rules generate an abstract structure with special characters marking structural transformations that should be applied and the transformational processor generates from this string a structurally transformed string (see Figure 3-2). These

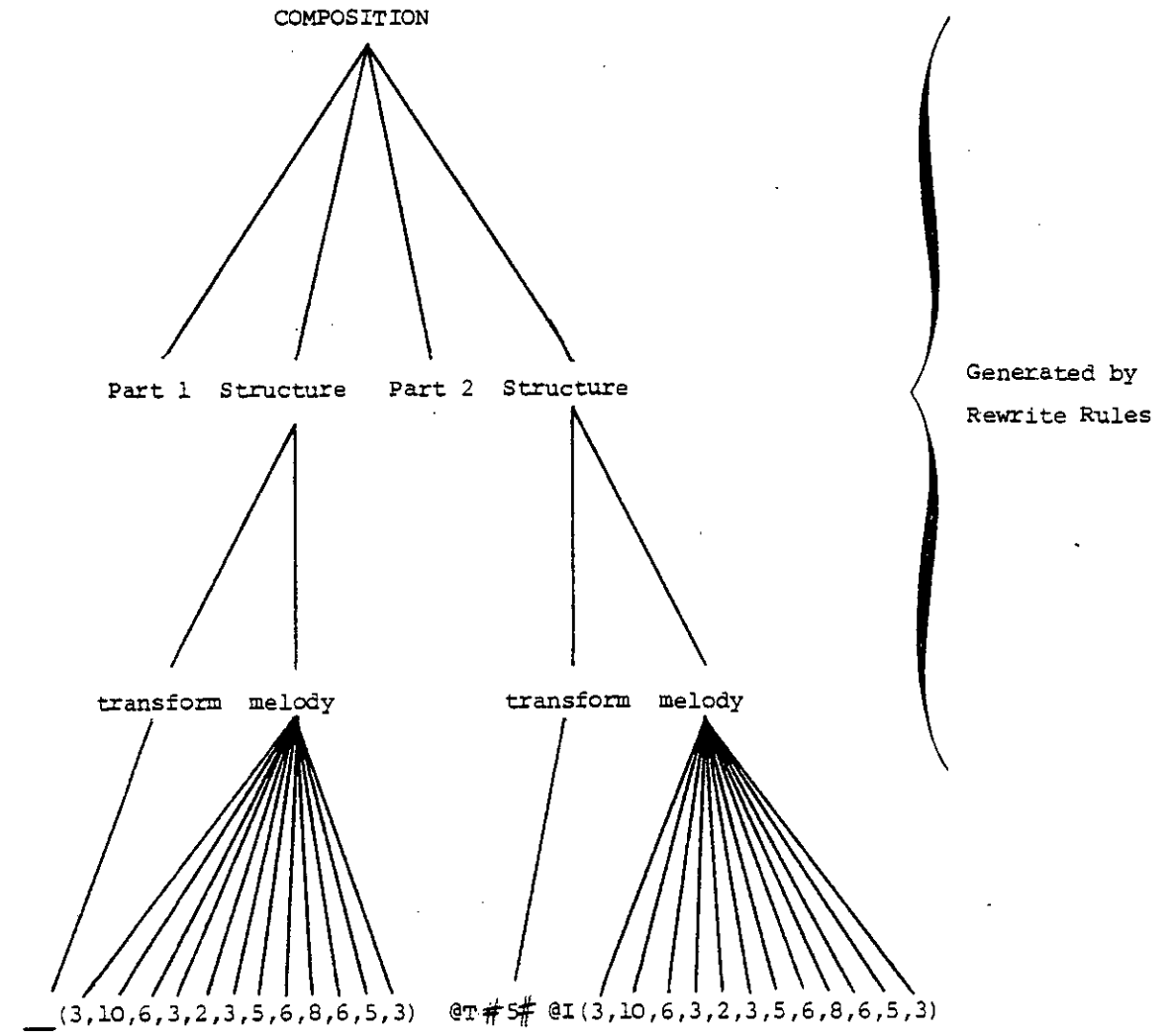


Figure 3-2: A parse of the generation of the beginning of CONTRAPUNCTUS XII from J.S. Bach's 'Art of Fugue'.

First, a terminal string is generated by applying rewrite rules.

[COMPOSITION → Part 1, STRUCTURE, Part 2, STRUCTURE]
[STRUCTURE → TRANSFORM, MELODY]
[MELODY → 3,10,6,3,2,3,5,6,8,6,5,3]
[TRANSFORM → __ (ie. null) . @T#5# @I (i.e. inversion transposed a fifth)]

Second, parts of the string are transformed. Note that '@' indicates a transformational change marker.

transformations correspond to commonly used music transformations such as transposition, inversion and retrograde.

Recent analytical research has suggested that compositions may often be described as consisting of a small number of basic units or strings, eg. two and three note patterns, which are subjected to a number of transformations to generate complex music structures. Rouget (1961) analysed African chants and found that they could be described in terms of a very few 'minimal units' which are repeated and transformed. Similarly, Bertoni, et al (1978) found that Bach could be analysed and represented with a few musical units and the transformations of inversion, transposition and retrograde.

This suggests that the process of composition consists, at least in part, of defining minimal units and selecting transformations that should be applied to them to generate larger structures. In GGDL one can define such units and transformational structures using the phrase-structure rules and the process of applying the transformations is automated. However, the complexity of structures described in these terms still may be rather great. An analysis by Goguen (1975) of 'Three Blind Mice' in terms of three minimal units and some transformations suggests that the problem of formally representing the structure of what would ordinarily be considered a trivial musical example need not be non-trivial (see Figure 3-3).

Lastly, a morphological mapping stage takes the abstract

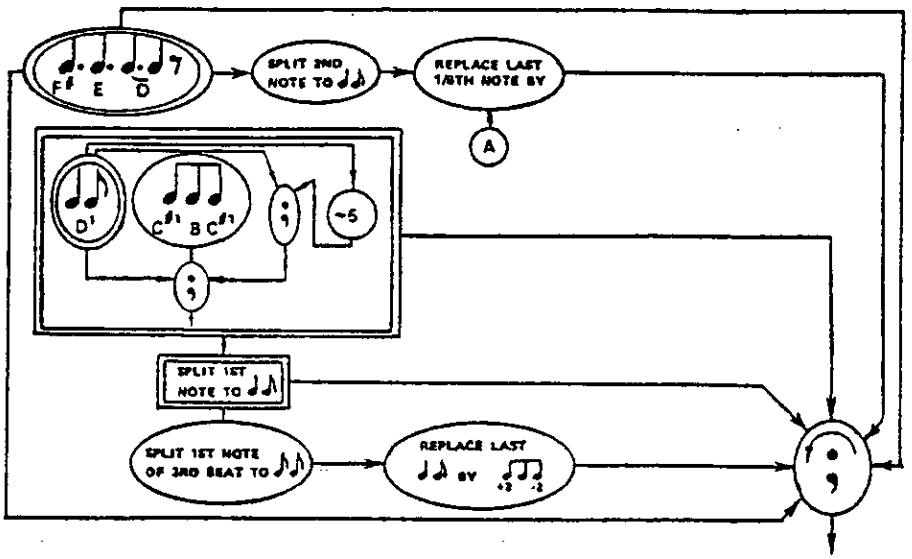


Figure 3-3: Goguen's Analysis of "Three Blind Mice".
The piece is described in terms of the minimal thematic units - i.e. repeated strings of notes - which can be found. There are three, each encircled in the above diagram. These units are sequenced in various juxtapositions and with various transformations applied to form the complete piece; ; indicates succession.

representation of the music structure, generated by the phrase-structure and transformational processing stages, and translates it to an interpretable format. This process of mapping may also be formally defined in GGDL. One could define rules to map an abstract representation of a music structure into, for example, data formatted for a synthesiser, or to a score format (see Figure 3-4).

The remainder of this chapter describes the facilities of the GGDL language for defining a generative grammar. It is divided into three sections dealing in turn with the three stages of processing of the GGDL generative system:

1) generation with Phrase-Structure Rules
2) Transformational processing
3) Morphological Mapping.

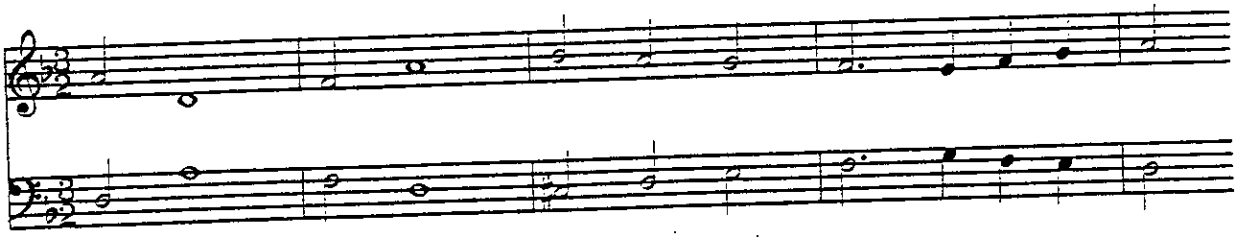


Figure 3-4: A structure generated from the rewriting and transformational processors (see Figure 3-2) shown after mapping for a graphics program.

1 Phrase-Structure Rules

3.1.1 Rewrite Rules

A common way of formally representing the rules of a grammar are 'rewrite rules'. These rules take the form:

[X -> Y]

where the arrow may be interpreted as an instruction to replace or 'rewrite' the character string occurring to the left of the arrow with the character string to the right of the arrow. Often a rule allows more than one possible substitution of the characters found on the left of the arrow. An example of this in GGDL is:

[X -> A, . B, . C,D, . E, . F, . G,]

where there are six alternative choices for the substitution of 'X' separated by periods ('.').

To initiate the generation of an utterance in the language defined by a set of rewrite rules, a string of characters is input to the generative system. All the sequences of these characters which match the left-hand side (LHS) of one of the rewrite rules is substituted by one of the sequences on the right-hand side (RHS) of that rule. If a string of characters can match more than one LHS, a RHS string from the rule with greater priority is substituted (cf. 3.1.12).

3.1.11 Rewrite Conventions

GGDL uses notational conventions for rewrite rules that require all rules to be contained within square brackets, ie. '[' and ']'; the period ('.') character will separate alternatives on the right-hand sides of rules; in terminal (3.1.13) strings the comma (',') is a separator for character strings which are morpheme names, ie. the names of objects; parentheses may be used to group morphemes.

The '_' character is interpreted in the context of rewrite rules as meaning "any string of characters" and can be used on the left-hand side to indicate that a match is found if the separated LHS characters occur with any string of characters in between them in the input string. If the '_' character is also used in the RHS character string, the replacement strings on the right-hand side of the rule will replace each of the separated character strings of the left hand side. For example, the rule

```
[ STRING1 _ STRING2 -> STRING3 _ STRING4 ]
```

will result in the substitution of STRING3 for STRING1 and STRING4 for STRING2. The same number of '_' characters must occur in the RHS substitution as on the LHS. Alternatively, if the '_' character only occurs on the left-hand side of the rule, then everything between and inclusive of the characters matched on the left-hand side will be replaced by the right-hand side substitution. Alternative RHS substitutions may use, in the same rule, either format. For example:

```
[ Q _ F, X _ P -> V _ A,B, W _ H,G . ZZ ].
```

3.1.12 Rewrite Rule Priority

The priority of application of different rules is implicit in their ordering - the first rules are applied first. The different ordering of rules, it should be noted, may result in a different rewriting.

In rewriting, first all matches with the LHS of the first rule are searched for, each match resulting in a substitution with one of the RHS character strings. The search for a match with the LHS of the rule begins at the start of the string. By default, the search for any remaining matches with the same rule continues from the point in the string immediately after the substituted string. This makes it possible to have RHS substitutions which include the same character string as the LHS without causing infinite recursion.

After all matches with the LHS of the first rule have been found and substitutions have been made, matches with the LHS of the second rule are then searched for, beginning again at the start of the string. This process is repeated until all the rules have been tried with successful matches resulting in substitutions. If any matches have been found in the cycle through the rules the cycle is repeated and the process of rewriting continues by searching for matches with the LHS of the first rule, and, after applying the first rule, continuing with the remaining rules. The process of rewriting is ended when no matches are found with the LHSs of any of the rules. That is, the

rewriting process cycles through all the rules in order repeatedly until a complete cycle without matches is made. The string will at this point consist only of terminals and be output to the Transformation process (cf. 3.2).

It is possible, however, to set flags in a GGDL grammar program to indicate that generation should not be done in the manner described above, ie. the default. One flag may be set to indicate that the rules should be cycled through in order, from first to last, but that on completion of the cycle generation should terminate. That is, the rules should be cycled through once only. A second flag may be used to indicate that, after a match, searching for further matches with the same rule should continue from the beginning of the input string so that any significant changes may be considered and recursion may occur within a single rule. These flags may be altered during generation by user-defined functions or procedures (cf. 3.1.24). It is the responsibility of the user to ensure that generation will not lead to infinite recursion and that the string finally generated consists only of terminals.

3.1.13 Terminals and Non-Terminals

Terminals and non-terminals are represented by strings of characters which are their names. Non-terminals are sequences of characters which will ultimately be replaced by other characters, or possibly a null string, in the process of 'rewriting' the string. Terminals are sequences of characters which may be found

in the final output string after rewriting has been completed and for which there must exist a morphological definition that permits them to interpreted at the morphological stage of generation. Therefore, non-terminals must be found on the left-hand side of rules whereas terminals need not be. All space-characters and newlines are ignored in rewrite rules. The names of terminals and non-terminals, ie. character strings, are separated by the ',' separator character in rewrite rules as well as in the string that is rewritten.

In GGDL, no distinction is made between upper and lower case characters; all characters are read as upper case. However, for clarity, the convention for the examples presented in this thesis is that terminals are represented by lower case character strings and non-terminals are represented by upper case character strings.

3.1.14 Types of Rewrite Rules

Rewrite rules may be distinguished by certain properties of their left-hand side (LHS) and right-hand side (RHS), rules of different types having a greater or lesser degree of power in expressing grammatical relationships. In Chomsky (1957), four different types of grammars are defined where each is qualified by the type of rewrite-rules that may be used in defining a language. These are, beginning with the weakest in expressive power, Type 3 (Finite-State), Type 2 (Context-Free), Type 1 (Context-Sensitive) and Type 0 (Free) grammars. Each of the lower grammar-type rules may be used in the definition of a more powerful grammar. For a

full discussion of these grammars and the differences in their expressive power, Lyons (1968), Chomsky (1957) or any introductory text dealing with generative linguistics should be consulted.

GGDL permits the use of all of these types of rules.

3.1.141 Type 3 Grammars

Type 3 Grammars are characterised by rewrite rules which have only one non-terminal on the LHS and at most one terminal and one non-terminal, always to the right of a terminal, on the RHS of the rule. A Type 3 rewrite rule would be:

```
[ X -> a, Y ]
```

where 'A' is a terminal and 'Y' is a non-terminal. By using a number of such rules an infinite variety of sequences may be generated, eg.:

```
Grammar-Type-3:
      Rule 1: [ X -> a, Y ]
      Rule 2: [ Y -> b, Y . b, Z ]
      Rule 3: [ Z -> c, Z . d, X . d ]
```

('a','b','c', and 'd' are terminals as they do not occur on the LHS of any rules.)

If such a grammar were initiated with the non-terminal 'X', a string of terminals would be generated consisting of strings with an 'a' followed by an arbitrary number of 'b's followed optionally by an arbitrary number of 'c's and then a single 'd'. An example of a state of the string during generation is:

```
1) a,b,b,c,d,a,Y...
```

By application of Rule 1 to [X] the string [a,Y] is generated.

The application of Rule 2 produces [a,b,Y] and again, produces [a,b,b,Z]. Applying Rule 3 produces [a,b,b,c,Z] and again, produces [a,b,b,c,d,X]. Then applying Rule 1 (cf. 3.1.12), [a,b,b,c,d,a,Y] is generated. The process of rewriting would continue until Rule 3 was applied and generated only the terminal 'd'. Other examples this grammar would generate are:

2) a,b,b,b,b,b,c,c,c,c,c,c,c,c,c,d,a,b,b,c,d,a,b,b,b,d

and:

3) a,b,d,a,b,b,c,d,a,b,b,b,c,c,c,d,a,b,b,b,b,c,c,c,c,d

and:

4) a,b,b,b,c,c,c,d,a,b,b,c,c,d,a,b,c,d

3.1.142 Type 2 Grammars

Type 2 Grammars are characterised by rewrite rules which have only one non-terminal on the LHS and any number of non-terminals and/or terminals on the RHS of the rule. A Type 2 rewrite rule, for example, is:

[Q -> c, G, f, Z]

where the RHS elements may be either terminals or non-terminals. Thus, any non-terminal may generate in one production a string of non-terminals and terminals.

Looking again at the fourth example of a string generated by 'Grammar-Type-3', the number of 'b's and succeeding 'c's is always the same. This however, need not be the case as can be seen from other possible strings the same grammar can generate. It is not possible to define a Type 3 grammar to only generate strings where

the numbers of 'b's and succeeding 'c's is always the same, ie. strings of 'b'(n)'c'(n) for all values of 'n'.

Instead of Grammar-Type-3, a set of rules which will always generate sequences where the number of 'b's is equal to the number of 'c's can be defined using context-free, ie. Type 2, rewrite rules.

```
Grammar-Type-2:
        Rule 1: [ X -> a, Y ]
        Rule 2: [ Y -> Q, Z ]
        Rule 3: [ Q -> b, Q, c . b, c ]
        Rule 4: [ Z -> d, X . d ].
```

Examples of strings generated by Grammar-Type-2 are:

```
        a,b,b,c,c,d,a,b,b,b,b,c,c,c,c,d
```

and:

```
        a,b,b,b,b,b,b,b,b,c,c,c,c,c,c,c,c,d,a,b,c,d
```

3.1.143 Type 1 Grammars

The rewrite rules of a Type 1 Grammar allow one or more non-terminals and any number of terminals on the left side of the rewrite arrow. There should be at least as many tokens in each right-hand side alternative. This makes for a considerable difference in power over rules of Type 2 Grammars as now it is possible to specify the context in which a non-terminal should be rewritten. With Type 1 rules it is possible to define different rewritings of a non-terminal in different contexts.

In a Type 2 Grammar, as there can be no more than one non-terminal on the LHS of the rule, any occurence of the LHS

characters is rewritten without consideration of the context in which it is found. Should one want to rewrite the non-terminal 'P' as 'Q, T' only if the preceeding non-terminal is 'S', there is no way to express this with a Type 2 rule. As Type 1 Grammars allow more than one non-terminal or terminals on the LHS of the rule, this might be represented as:

[S, P -> S, Q, T].

The rewriting of 'P' in other contexts would be represented by other rules.

In another example, a cadence [I,V,I] may be generated by the rule:

Grammar-Type-1:
Rule 1: [CADENCE -> I, V, I]

In order that the occurence of the tonic note of the tonic chord preceeding the dominant chord does not weaken the effect of the final tonic chord, a second rule may be added which considers the context of the tonic chords:

Rule 2: [I, V -> I6, V].

Now the rules will generate [I6, V, I].

3.1.144 Type 0 Grammars

Type 0 Grammars allow rewrite rules of any form - ie. with any number of non-terminals and terminals on either side of the rule, null productions (strictly speaking the RHS of the other grammar types should expand the LHS or at least not decrease it), not completely specified contexts, and so on.

For example, Type 0 rules may be used to represent a Sonata as an ABA form.

```
Sonata-Grammar:
      Rule 1: [ SONATA -> A, B, A ]
      Rule 2: [ B -> DEVELOPMENT ]
      Rule 3: [ A -> theme1 in KEY, theme2 in KEY ]
      Rule 4: [ theme1 in KEY -> theme1 in tonic ]
      Rule 5: [ theme2 in KEY _ DEVELOPMENT -> theme2 in dominant
                              _ DEVELOPMENT ]
      Rule 6: [ DEVELOPMENT _ theme2 in KEY -> DEVELOPMENT _
                            theme2 in tonic ]
      Rule 7: [ DEVELOPMENT -> modulation of themes ]
```

This grammar will, if initialised with 'SONATA', generate

```
[ theme1 in tonic, theme2 in dominant, modulation of themes,
      theme1 in tonic, theme2 in tonic ]
```

It should be noted how the '_' character is used to say, for example, in Rule 5, "If 'theme2' occurs before the DEVELOPMENT" and in Rule 6, "If 'theme2' occurs after the DEVELOPMENT". Though Rules 6 and 7 specify a context in which KEY should be rewritten, the context is not completely specified. This distinguishes this 'Sonata-Grammar' from a Type-1 grammar as in the latter, the context in which a non-terminal is to be rewritten must be completely specified by terminals and non-terminals. It also should be noted that the order of the application of the rewrite rules in this example 'Sonata-Grammar' is significant. If Rule 7 were to be applied before Rules 5 and 6, the matches of these latter rules would not occur and improper strings would be generated, ie. strings that still included non-terminals. The priority of the application of different5 rules is implicit in their ordering - the first rules are applied first (cf. 3.1.12).

It is also possible in a Type 0 Grammar to rewrite terminal strings. This may be done by defining rewrite rules which will

rewrite terminals or sequences of terminals. Alternatively, a Type 0 Grammar also permits the inclusion of a third type of character string - ie. neither terminal nor non-terminal - the 'Structural Change Marker'. These initiate transformations, special rewritings, of the string in a distinct transformational stage which follows the process of rewriting using the Phrase-Structure Rules. Formally, a Transformational Grammar is a Type 0 Grammar with special type rewrite rules - the Transformational Rules (see Section 3.2).

Though theoretically, 'Structural Change Markers' are seen as distinct in type from terminals and non-terminals, in practical terms, using GGDL where the process of generation with Phrase-Structure Rules and the process of structural transformation are separated, these markers may be seen as terminals in the Phrase-Structure Rules. That is, after the process of rewriting is completed, the generated string may include such markers as well as the defined terminals of the rewrite rules.

3.1.2 Control of Rewriting

Rewrite rules may generate more than one string:

[A,B -> D, . Q,C . X].

Rewrite rules define paradigmatic relationships in a grammatical definition and where there is more than one possible generation, ie. more than one RHS string, from a given rule, the question arises as to which one should be selected and how selection should

be controlled. Apart from the rewrite rules, a separate level may be defined in the grammar expressly for the purpose of controlling selection in the rewriting process. To control selection, the system provides a number of 'system functions' which may be applied for determining selection, or alternatively, functions may be defined. A function for selection control is essentially a set of rules by which selection from several choices is determined. In GGDL programs, a high-level programming language is provided in which routines and functions may be specified precisely, a more rigorous definition of which may be found in Appendix 1. Examples of GGDL programs also can be found in the Appendices.

3.1.21 System Control Functions

The GGDL language has three predefined selection functions that may be used to control the process of rewriting. These permit selection from a number of possible selections to be made randomly, by invoking a control mechanism simulating serial selection, and by invoking a mechanism which simulates a finite-state machine.

3.1.211 Random Selection

Where there are several RHS strings and no explicit selection control, as in the rules above, a random choice is made from all the possible strings. By default, selection is random. An example of a rewrite rule where selection of a RHS string is random is:

[X -> G . X,WZ . NOTE . NOISE]

3.1.212 'Blocked' Generation

A special system control function can be invoked by the '!' character immediately following the rewrite arrow or invocation number (cf. 3.1.3):

[Q -> ! A, . B, . C, . D, . E,].

This will select randomly one of the possible RHS strings and then 'block' the generation of this string until all the other possible selections have been chosen. No possible selection can occur twice until all others have occured at least once. For example, given the string [Q,Q,Q,Q,Q,Q,Q,Q,Q,Q,], the repeated application of the above rule for rewriting "Q" might generate:

[C,A,E,D,B,D,E,C,B,A,].

The selections from the RHS are made randomly, but none of the RHS possibilities are selected a second time until all the possible RHS strings have been selected at least once.

This selection principle is a familiar one in contemporary music based on the serial law of Schoenberg that 'no note may be repeated in the series until all notes have occured once'. It is also used in the programs of G. M. Koenig (1972, 1978) where it is called 'Series'.

3.1.213 Finite-State Generation

The second provided system control function allows for the control of generation by a finite-state transition matrix. It is invoked by an asterisk ('*') immediately following the rewrite arrow or the invocation number (cf. 3.1.3):

[A -> * ...

Following the asterisk there must be 'arrow-bracketed' (ie. '<' and '>') control-information and a matrix consisting of rows of transition values for each of the strings that may be generated.

The bracketed control-information consists of two variables. The first, which is obligatory, is the number of possible strings that may be generated by the rule. This must be a constant value. The second is an optionally defined value for the initial state of the finite-state machine of this rule only, ie. the state from which the first transition using the rule is to be made. As this value is optional, by default, the initial state is randomly determined from among the possible states. The state-value will automatically be updated by the system when the grammar is invoked.

The control values are separated by a comma (','). Examples of bracketed control-information for a Finite-State rewrite rule are:

< 3 >

and:

< 2, 1 >

The matrix of transition values is defined as a set of rows of transition values with each row enclosed in parentheses. Each row is headed by the character string associated with a transition to that state, ie. the string that will be generated upon transition to that state. The string is followed by constant values which give the weights of the possible transitions from the state associated with a row to other possible succeeding states. All the values are separated by periods ('.'). There must be the same number of rows and the number of transition values for each row as the value given to the first of the control-information variables.

A row may take the form:

```
( A . 21 . 357 . 84 . 189 . 165 . 204 . 408 . 96 )
```

where the string generated will be 'A' and there are eight possible transition states, ie. there are eight possible strings that the rule may generate. In the case of this row, the probabilities of the transitions to the respective states, on the next invocation of the grammar, are 2.1%, 35.7%, 8.4%, 18.9%, 16.5%, 20.4%, 40.8%, and 9.6%.

An example of a complete Finite-State rewrite rule is:

```
[ X -> *  < 8, INITSTATE >
    ( A, .  21 . 357 .  84 . 189 . 165 . 204 . 408 .  96 )
    ( B, .  84 .  89 .  76 . 126 . 150 . 136 .  72 . 144 )
    ( C, .  84 . 323 .  21 . 126 . 150 .  36 . 272 . 144 )
    ( D, . 336 .  81 .  19 .  84 . 135 .  24 .  48 . 216 )
    ( E, .  19 .  63 . 336 . 171 . 110 . 306 . 102 .  64 )
    ( F, .  76 .  16 . 304 . 114 . 100 . 204 .  18 .  96 )
    ( G, .  76 .  57 .  84 . 114 . 100 .  54 .  68 .  96 )
    ( H, . 304 .  14 .  76 .  76 .  90 .  36 .  12 . 144 ) ]
```

The finite-state rewrite rule in GGDL could be used for the description of 'stochastic composition rules' such as those used

by Iannis Xenakis. The finite-state rewrite rule above, for example, is an equivalent representation in GGDL of the matrix of transition probabilities defined by Xenakis and shown in Figure 2-1.

3.1.22 Non-System Rewrite Control

If the system control functions are found inadequate for the type of control desired, in GGDL a simple high-level language is provided for the definition of non-system rewrite control functions. Non-system control functions are called by name, enclosed in arrow-brackets immediately after the rewrite arrow or invocation number (cf. 3.1.3). The possible strings which may be selected by the function follow separated, as usual, by periods ('.'). A rewrite rule with control by a non-system function is, for example:

```
[ X -> < FUNCNAME > SEL1 . SEL2 . SEL3 ].
```

The control functions are written apart from the rewrite rules. They are functions to the extent that they must return an integer as a result, the number being interpreted as the selection of the Nth possible RHS string. In this thesis non-system control functions are defined as sets of English statements defining the rules for choosing a RHS element. Each set of such statements will be given a name and this name will be the name of the function which is referred to as described above - ie. it is called for making a selection where the name occurs between arrow-brackets.

3.1.3 Multiple Invocation

Any rule may be invoked to generate several of the RHS strings concatenated in a single rewrite-generation. This may be done by placing immediately after the rewrite arrow an 'invocation' number, which may be any legal arithmetic expression, enclosed in '#'s. For example, a 'blocked' generation rule (cf. 3.1.212) could be defined which, for each invocation, ie. whenever a match with the rule's LHS "Q" is made, would generate a concatenation of five possible RHS strings. That five strings should be generated is indicated by the invocation number, " #5# ", just after the rewrite arrow.

[Q -> #5# ! A, . B, . C, . D, . E,]

As there are only five possible RHS strings and blocked generation is used, ie. no RHS string may be selected a second time until all other possible selections have been chosen once, each time the rule is invoked a complete series of the RHS elements will be generated. Example strings are:

A,B,C,D,E

B,E,D,A,C

D,C,A,E,B

3.1.4 Metaproductions

Metaproductions are a special type of production or rewrite rule by which added context-sensitive control may be gained in a grammar. Originally they were introduced by Aad Van Wijngaarden (1965) and were later used in the description of the, at the time,

new programming language ALGOL-68. The use of metaproduction rules in 'W-Grammars', ie. grammars with metaproductions, is discussed by Uzgalis (1977). Using metaproductions in a grammar, one defines two sets of rewrite-rules, a set of 'metaproduction rules' and a set of 'hyper-rules'. From these a third set is generated - the set of rewrite rules used to generate the language.

Metaproduction rules may have the same format as any rewrite rules (cf. 3.1.2) excepting that, in GGDL, they are enclosed by the '"' character rather than brackets '[' and ']'. An example of a metaproduction rule is:

" Q -> ! A, . B, . C, . D, . E, "

Essentially, the LHS of a metaproduction rule acts as a variable in the 'hyper-production rules'. Hyper-production rules are the bracketed rewrite rules with uninstantiated variables, ie. character sequences which match those of the LHSs of the metaproduction rules. The metaproduction rules are each invoked once and once only with the initialisation of a program and a string is generated from the RHS possibilities. Only one of the possible strings that the rule may generate is generated. Subsequently, the LHS of the metaproduction is matched against the character strings, both LHS and RHS, of the hyper-rules, the ordinary rules enclosed by brackets '[' and ']', and where a match is found the one generation of the metaproduction rule is substituted for the metaproduction rule's LHS. This substitution process is slightly different from Van Wijngaarden's "Universal

Replacement Rule".

For example, two metaproduction rules are defined:

```
" SERIES1 -> #5# !  A, . B, . C, . D, . E, "
" SERIES2 -> #5# !  A, . B, . C, . D, . E, "
```

and there is a rewrite rule with the metaproduction-LHS-strings (ie. a hyper-rule):[1]

```
[ COUNTERPOSE -> SERIES2, SERIES1, @I (SERIES1),
        @B (SERIES2) ]
```

With the initialisation of the program, generation with the metaproduction rules will be invoked. Let us say that SERIES1 generates the string "A,C,E,D,B" and SERIES2 generates "B,E,D,C,A". The LHSs of the metaproductions will then be matched against strings of the rewrite rules and these generated strings substituted where matches are found. The rewrite rule "COUNTERPOSE" will now look like:

```
[ COUNTERPOSE -> B,E,D,C,A,A,C,E,D,B, @I (A,C,E,D,B)
          @B (B,E,D,C,A) ]
```

It is with this new rule that rewriting and generation of strings in the language will now take place.

The added facility of metaproductions allows one to more easily express certain relationships. With hyper-rules, one can describe, for example, a compositional structure as a relationship between objects where the objects are undefined. By instantiating the hyper-rules the structure is 'filled' with a different set of objects without affecting its definition. As in the above example, one could describe manipulations of a series or theme and then use a different series or theme as the basis of the structure, for example, by altering the rule by which the series

1) In GGDL, the character '@' indicates a 'structural change marker'. See Chapter 2.

or theme is generated.

Another example of the additional expressive power with metaproductions would be the case where one wanted to generate the same number of 'a's as 'b's and 'c's. With Grammar-Type-2 (cf. 3.1.142) it was possible to generate the same number of objects if they succeeded one another and then only with two objects. It would not be possible with either Type 2 or Type 3 Rules to generate the same number of several objects, and though it is theoretically possible with a context-sensitive grammar, ie. Type 1, it is extremely difficult and impractical to express with such rewrite rules.

M-Grammar:

```
Mrule 1: " X -> Y . X,Y "
Rule 1: [ Q -> X,a, X,b, X,c ]
Rule 2: [ Y,a -> a,a ]
Rule 3: [ Y,b -> b,b ]
Rule 4: [ Y,c -> c,c ]
```

Strings that this grammar will generate if initialised with 'Q' are:

a,a,a,a,b,b,b,b,c,c,c,c

a,a,b,b,c,c

2 Transformational Rules

3.2.1 Structural Change Markers

'Structural Change Markers' initiate transformations of the character-string generated from the Phrase-Structure Rules. In GGDL, there are four transformations which can be initiated by the occurrence of a Structural Change Marker. The transformations take strings of characters enclosed in parentheses as arguments, these are what Chomsky (1957) calls 'kernel strings'. The transformations are initiated by the markers '@B', '@I', '@T' and '@M'. The first three transformations reverse the order of (@B), invert the relative relationships of (@I), or transpose (@T) the relative relationships of one parenthetically enclosed string of morphemes, ie. one of the kernel strings. The fourth transformation (@M) 'merges' two or more sets of morphemes. The inversion and transposition transformations require that the morphemes be declared in ordered sets. The argument or kernel string of a transformation may include markers (with their kernel strings) indicating further transformations; such transformations are said to be 'nested'.

In a string where several of these markers are to be found, first the transposition transformation is applied, and if nested within an outer transposition transformation, the innermost transformation is applied first. Likewise, next the inversion transformation is applied, and then, when all transposition and inversion markers have been removed by applying the appropriate

transformations to their arguments, the backwards transformation is applied, though where the markers are nested, this transformation works from the outermost parenthetically enclosed string to the innermost. Lastly, the merge transformation is applied working innermost application outwards.

For the following examples, an ordered set of morphemes is defined:

[a,b,c,d,e,f,g,eg,eg2,k,l,sk,er,df]

Each morpheme is named by an arbitrary sequence of characters and is associated with an index which is its ordinal position in the set. For example, the index of 'a' is '1', and the index of 'eg' is '8'. Morphemes may only belong to one set. The inversion and transposition transformations are applied modulo the number of elements of the morpheme set in which a given morpheme is declared.

During the process of generation, a string is passed from the Phrase-structure Rules rewriting process to the Transformational process. The latter process removes markers from the string by applying the appropriate structural transformations.

3.2.11 @I - Inversion Transformation

The inversion transformation is based on the concept of the 'interval' between a pair of morphemes. Given an ordered set of morphemes, the 'interval' between two morphemes is equivalent to the difference between their indices. For example, given the ordered set of morphemes defined above, the interval between 'a',

the first element in the set, and 'e', the fifth element in the set, is '+4'.

The inversion transformation 'inverts' the intervals in a sequence of morphemes beginning with the first element in the sequence. The inversion of a string of morphemes, then, is the string in which the first element, the 'base' element, is the same and the intervals between all of the succeeding elements are the 'inverse' or 'negative' of the intervals in the original string. To generate the inversion of a string of morphemes, the interval between the first element and its successor is calculated. Then, the interval is subtracted from the index value of the base element and the result is used as an index to select the next element of the inversion string from the morpheme set. The inversion of the next interval in the kernel string is then subtracted from the index of the second element of the inversion string and used to select the third element, and so on until the complete inversion string has been generated. All arithmetic is carried out modulo the size of the morpheme set.

For example, if the kernel string were:

[@I (a,e,er,b,k)]

the inversion string that would be generated when the inversion transformation was invoked would be:

[a,l,c,df,f]

In the kernel string, the interval 'a,e' is '+4'; the element four steps below 'a' is 'l', so, in the inversion string 'a,e' has become 'a,l'. Likewise, the interval 'e,er' is an '+8'. This

inverted becomes '-8' and the element in this relation to '1' is 'c'. The inversion is applied to all the intervals successively to produce the transform string.

For this transformation of inversion to correspond to what is understood in music to be inversion, all the elements of the kernel string must belong to the same set. The result of applying the inversion transformation to a kernel string including morphemes from different sets is undefined.

3.2.12 @T - Transposition Transformation

The @T transform is a shifting process - the intervals between all the elements (morphemes) are maintained but shifted either upwards or downwards in position in the ordered set.[1] The @T transformation may be followed by a 'transposition interval', a number enclosed by the '#' characters which is the interval by which the argument is shifted or transposed, by default the shift is one position in the scale upwards.

As an example, if the string input to the transformation processor were:

[@T (a,g,eg,l,c,df)]

the process would produce:

[b,eg,eg2,sk,d,a]

By default, the transposition interval was '+1'. If the input string were:

[@T # -5 # (a,g,eg,l,c,df)]

the result would be:

1) Ie., the ordered set in which the morpheme is declared. See page 67.

[k,b,c,f,sk,eg2]

3.2.13 @B - Backwards (Retrograde) Transformation

The @B transformation reverses the order of the morphemes enclosed by the set of parentheses. Given the string:

[@B (g,c,eg,er,a)]

the following would be produced:

[a,er,eg,c,g]

3.2.14 @M - Merge Transformation

The @M merge transformation allows one to superimpose one generated structure on another. It requires as an argument at least two sets of morphemes - each with the same number of morphemes. The two sets are reordered so that the first morpheme of the first set is followed by the first morpheme of the second (and other sets), the second, by the second of the other sets, etc. This allows one to generate independently a number of structures - for example, by a different set of rules - and then superimpose them upon one another where each morpheme might represent a different component of the final sound. A string of pitches and a string of durations may each be generated and then combined (merged). For example,

```
[ @M (pitch1, pitch2, pitch3, pitch4) (dur1, dur2, dur3
    dur4) ]
```

is transformed by the merge transformation into:

```
[ pitch1, dur1, pitch2, dur2, pitch3, dur3,
    pitch4, dur4 ]
```

The numbers of elements in the sets must be the same.

The '@T' and '@I' transforms replace a given morpheme with another from its set. If one applies an inversion to a group of morphemes from different sets the notion of inversion as it is normally understood in musical contexts will be lost, similarly for transposition.

The '@B' and '@M' transformation take groups of morphemes enclosed in parentheses as single objects and will move the complete group.

Strings may be embedded and mixed. Where this occurs priority is @T (innermost first), @I (innermost first), @B (outermost first), @M (innermost first).

3 Morphological Rules

3.3.1 Morphological Definition

Each morpheme is represented by a string of characters, its name. The rewriting and transformational processes generate a string of morphemes, representing 'objects' such as sounds or durations, separated by the ',' character. The morphemes are all terminals in the generative grammar. The morphemes are mapped during the morphological stage of processing the string into a representation that is suitable for interpretation - eg. a score.[1]

1) Pedantically, if the score is in the form of a description of

Using morphological rules, morpheme strings generated by the GGDL grammatical definitions can be mapped into a format acceptable to a given synthesis program. The morphological mapping process is defined by a set of rules separate from the Phrase-Structure Rules. This allows the substitution of a different mapping process with the same generative mechanism, ie. the latter only generates an 'empty' structure and how it is 'filled' in, as far as the system is concerned, is arbitrary. The same structure may be filled by alternative sets of objects. Similarly, the same objects may be used for the mapping of utterances generated by different grammars.

Mapping routines describe a process for rewriting the morpheme character strings into a different format - eg. one acceptable to a synthesis program such as MUSIC V (Mathews 1969), or the non-standard digital synthesiser (Holtzman 1978b,1979), or a speech synthesiser, etc.

---

actual sounds for synthesis, rather than 'morphological', one would have 'morphophonemic', or 'sonemic', as in 'sonology', rules, and, in the case of written output, 'orthographic' rules. What are here called morphological rules are more precisely 'morphological realisation' rules as morphological rules still work on an abstract level transforming morphemes into 'morphs'. In a transformational grammar designed for natural language one can distinguish, at the morphological stage, grammatical morphemes and lexical morphemes (lexemes). In a language where the objects processed by the morphological processor of a transformational grammar are mapped independently of each other, as in GGDL (or Chinese!), the distinction between morphemes and lexemes and the transformation into 'morphs' is lost. In the context of a transformational grammar, the term 'morpheme' is preferred though 'lexeme' or 'morph' could also arguably be used. A process similar to the morphological mapping process is discussed in Roads (1978) as lexical mapping.

The morphological mapping routines can generate data in the form of characters, or in the form of binary (1-byte) numbers, or a mixture of the two. It is up to the user to ensure that data is of the correct type for use with a synthesis program or otherwise.

Summary

The GGDL grammar definition language may be used to define the grammars of many different music languages. The rules of 'stochastic' music composition could be straight-forwardly represented using the system selection procedure for finite-state generation (cf. 3.1.213). This selection procedure could also be used to represent, for example, Koenig's 'Ratio' selection procedure by assigning the same values to each row of the transition matrix. In addition, Koenig's 'Alea' selection procedure is equivalent to the default random selection (cf. 3.1.211) made by GGDL where a number of possible selections are given but no selection procedure has been indicated; and his 'Series' selection procedure is the same as the system procedure of blocked generation (cf. 3.1.212).

In addition, a composer may, should these provided procedures not be satisfactory for selection, define his own procedures using the high-level language programming facilities available. These procedures may then be integrated with the syntax specification of the composing language represented by the rewrite rules. Though it would also be possible to program such procedures in other programming languages, the GGDL language has been specially

designed to facilitate the expression of music language constructs. By separating the definition of a language syntax from the control of selection, it permits an elegant and clearly interpreted representation of a language definition. Furthermore, by providing a framework and conceptual foundation, the GGDL system should facilitate the process itself of formalising the rules of a language.

The tranformational processing of GGDL permits the simple representation of music structures with components which are different only to the extent that they are transformations of the same object. Music structures can often be described in terms of a small number of basic components which are transformed and juxtaposed to form complex structures. The formal description of such structures is greatly facilitated by being able to simply 'mark' such transformations.

Lastly, the independence of the mapping process introduces an elegant modularity to a grammatical description. This results in flexibility when defining a generative grammar for music. The objects of the composition process may be defined independently of the structure generation process and thus, different objects may be used with the same structure generation process or, alternatively, the same objects may be used with different generation processes.

GGDL is a powerful language which may be used to aid composers in the composition process. Using GGDL, composers may define

music languages. Composition, or the generation of structures, in the language may then be done automatically by a computer.

Chapter 4: The Generation of Music Structures

GGDL is a programming language designed especially for the definition of generative grammars for music languages. A definition in the form of a program is sufficiently explicit that a virtual machine can execute the instructions of the program to generate utterances in the described music language. In this chapter, it is shown how GGDL may be used to describe the grammars of, or at least give grammatical interpretations of, three compositions.

The first two examples, Steve Reich's "Clapping Music" (Reich 1972) and Arnold Schoenberg's "Trio" from the Piano Suite, Op. 25 (Schoenberg 1925), demonstrate that GGDL can be used to grammatically describe and generate the compositions of established composers. That is, these composers could have used GGDL to generate their compositions.

The third example is drawn from the work of D. Hamilton, a composer who used GGDL to compose "Four Canons" (1980) for a BBC commission. It is shown that GGDL and the process of describing a composition's structure by means of a grammar in order to automatically generate a composition, was a useful aid to D. Hamilton.

Steve Reich's "Clapping Music"

Steve Reich is one of the foremost proponents of 'process music', a method of music composition associated with the 'new simplicity' movement which was an influential school of compositional thought in the 1970s. In process music, a composition is defined by a set of formal rules which describe a way of transforming a sound structure - the structure that results from this tranformation process is the composition.

Steve Reich's "Clapping Music" is based on the transformation of a hand-clapped rhythmic pattern.

> "One performer claps out an unchanging rhythmic pattern, the other, starting in unison, then displaces the downbeat, a beat at a time, until after twelve changes of the rhythmic position both performers end in unison" (Reich 1980).

That is, the second performer slowly moves out of 'phase' with the first until, after 13 times through the pattern, the two voices are back in phase with one another, ie. they are again synchronised (Figure 4-1A).

This is a composition where the rules have been made explicit by the composer and are easily formalised. In a sense, the rules themselves are the composition. In GGDL, the composition may be defined as two sub-structures each of which is performed by a separate voice.

```
[ COMPOSITION -> voice1,STRUCTURE1,voice2,STRUCTURE2 ]
```

The first structure consists of 13 repetitions of the rhythmic cycle. In the rhythmic cycle, an attack (ie. clap) may be represented by the terminal '1' and a rest by the terminal '0'.

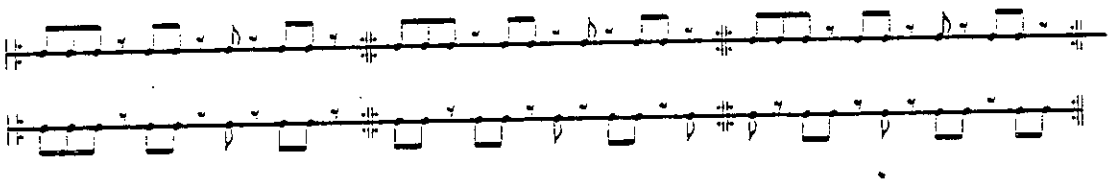


Figure 4-1a: 'Clapping Music' (Reich 1972); bars 1-3.

```
VOICE1
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
1,1,1,0,1,1,0,1,0,1,1,0
```

```
VOICE2
1,1,1,0,1,1,0,1,0,1,1,1
1,1,0,1,1,0,1,0,1,1,0,1
1,0,1,1,0,1,0,1,1,0,1,1
0,1,1,0,1,0,1,1,0,1,1,0
1,1,0,1,0,1,1,0,1,1,1,1
1,0,1,0,1,1,0,1,1,1,0,1
0,1,0,1,1,0,1,1,1,0,1,0
1,0,1,1,0,1,1,1,0,1,1,1
0,1,1,0,1,1,1,0,1,1,0,0
1,1,0,1,1,1,0,1,1,0,1,1
1,0,1,1,1,0,1,1,0,1,0,1
0,1,1,1,0,1,1,0,1,0,1,0
1,1,1,0,1,1,0,1,0,1,1,1
```

Figure 4-1b: 'Clapping Music' in a representation generated using GGDL. The terminal 'bar' has been replaced by newlines.

Each rhythmic cycle may be represented as a 'bar' of music.

```
[ STRUCTURE1 -> # 13 # bar,CYCLES1 ]
[ CYCLES1 -> 1,1,1,0,1,1,0,1,0,1,1,0, ]
```

For the second structure, given the basic rhythmic cycle, it is necessary to 'phase' its performance with each repetition. To do this a selection-function "SELECT" has been defined. In the function, two variables are used: one to 'point' to the presently performed event, either a clap (1) or a rest (0), and the other to keep track of where the phase-shifts should occur. Rewrite rules for generating the second STRUCTURE of the composition are:

```
[ STRUCTURE2 -> # 13 # bar,CYCLES2 ]
[ CYCLES2 -> # 12 # <SELECT> 1, . 1, . 1, . 0, . 1, .
      1, . 0, . 1, . 0, . 1, . 1, . 0, ]
```

The function "SELECT" could be defined as:

```
function select

    ! "objno" points to present beat of rhythm
  objno=objno+1
  if objno=13 then      ;! modulo 12 beats
    objno=1
  finish

  count=count+1
  if count=12 then
     ! a complete cycle of the rhythmic pattern has been
     ! completed so a 'phase-shift' is required, ie.
     ! "objno" is incremented.
    objno=objno+1
    if objno=13 then
      objno=1
    finish
    count=0
  finish

  result=objno

end
```

These rules generate the string in Figure 4-1B, which may be easily transcribed or mapped, for example, for synthesis. The composition could also be generated with other rewrite rules and

the function could also be defined in a number of other ways to produce the same result. The above example is one way of describing and generating "Clapping Music". The process, however, is entirely deterministic and will generate only one result. One could alter the clapping pattern by changing the terminal string generated by CYCLES, and, perhaps, study the 'form-potential' of Reich's 'process'.

A Schoenberg Trio

The Steve Reich composition is a straight-forward deterministic process that is readily formalised and programmed. The next example formalises the rules of a considerably more complex compositional process. The rules defined in fact give a formal, explicit description of one interpretation of the note-duration structure of a Schoenberg composition. (That is to say, there may be a number of other ways of describing the Schoenberg composition.) However, the rules also allow for the generation of other compositions which could be said to share the same set of rules as the Schoenberg. A language may include a large number of utterances - in this case, the language described includes among many utterances, the Trio from Schoenberg's Piano Suite, Op. 25 (Figure 4-2).

The following five rules may be used to generate the note structure of the Schoenberg Trio.

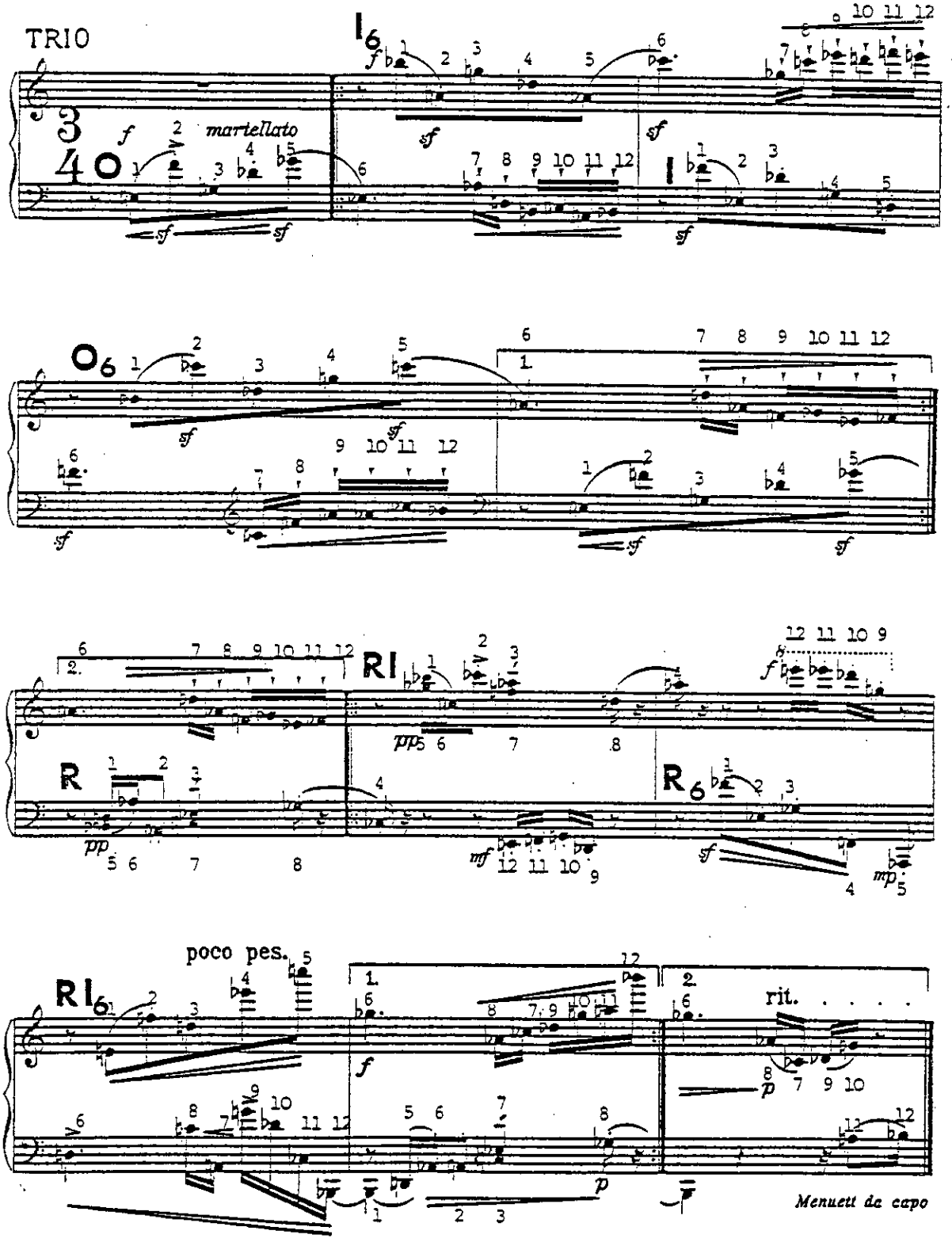


Figure 4-2: An analysis of the "Trio" from Suite fur Klavier, Op. 25 (Schoenberg, 1925).

Figure 4-2: An analysis of the "Trio" from 'Suite fur Klavier, Op. 25' (Schoenberg, 1925). Each version of the series is identified

O - Original
I - Inversion
R - Retrograde
RI - Retrograde Inversion
6 - Tranposed six semitones

The ordinal values of the members of the series have also been marked; note that in R_6 and RI_6 the 7th and 8th notes of the series are reversed.

Compositional Rules for the Pitch Structure of Schoenberg's Trio from the Piano Suite

```
! Meta-production Rule:
  " SERIES -> #12# ! obj1, . obj2, . obj3, . obj4, . obj5, .
     obj6, . obj7, . obj8, . obj9, . obj10, . obj11, . obj12, "
! Rule 1:
  [ COMPOSITION -> CANON ]
! Rule 2:
  [ CANON -> voice1, STRUCTURE, voice2, STRUCTURE ]
! Rule 3:
  [ STRUCTURE -> # 4 # VERSION-TYPE (SERIES) ]
! Rule 4:
  [ VERSION-TYPE (_) -> !  _ . @I (_) . @B (_) .
   @B ( @I (_)) . @T #6# (_) . @T #6# ( @I (_)) .
   @T #6# ( @B (_)) . @T #6# ( @B ( @I (_))  ]
```

The Schoenberg Trio is a canon. In Rules 1 and 2 it is decribed as a given structure ("STRUCTURE") occuring in parallel in time with another structure. The structures begin at different points in time, indicated by the 'entry times' of the different "VOICEs" (which may be defined in the mapping definition). Rules 3 and 4 are used to generate the actual structures.

The STRUCTUREs of the canon consist of four different groups of objects (ie. SERIES) where each series is qualified by a VERSION-TYPE, ie. a set of transformations to be performed on the series (Rule 3). The series may occur in eight different versions: the null transformed version " (_) ", the inversion transformed version " @I (_) ", the backwards (or retrograde) transformed version " @B (_) ", and the backwards inversion transformed version " @B (@I (_)) ", and each of these four transformed versions of the series transposed up six semitones. However, none of these versions of the series occurs twice. It would appear that Schoenberg has used serial selection - ie. none of the versions of the series occurs a second time until all other

versions have occurred at least once. In the rewrite rule (Rule 4), that serial selection should be used is indicated by the '!' just after the rewrite arrow.

The series is generated using a meta-production rule. By this means, every occurrence of the SERIES in the structure is substituted by the same generation of the meta-production rule. If this were not the case, for each occurrence of SERIES, a new series would be generated and the 'sense' of the composition's structure would be completely lost - it would be impossible to recognise what transformations had occurred as there would not be a static reference.

The rewrite rules deal only with 'abstract tokens'. At the transformational stage, in order for the inversion and transposition transforms to be applied, a relative ordering of the objects will need to be defined. In the following examples an object's number suggests its relative position, ie. 'obj1' is the first, 'obj2' the second, and so on, though this would be explicitly stated in a morphological definition program.

The duration structure of the Schoenberg Trio could similarly be described by a set of rules. For example, in all but the third SERIES of each STRUCTURE: the first six objects of the SERIES have the same duration, the last (ie. sixth) of the sub-group being tripled and the rest of the objects in the SERIES share the same duration, half the duration of the first six objects, with, again, the last object's duration being tripled. Thus, there are four

different possible durations: 'duration 1' (♪), 'duration 1 * 3' (♩.), 'duration 2' (♫), 'duration 2 * 3' (♫.). The actual values to be attributed to these duration-morphemes would be defined as part of the mapping process; for example, 'duration 1' might be defined as an eighth note, 'duration 2' as a sixteenth note, and so on. A function will need to be defined to select from these duration values to generate the correct rhythmic pattern: five 'duration 1's, one 'duration 1 * 3', five 'duration 2's, 1 'duration 2 * 3'.

To generate the duration structure of the first, second and fourth occurrences of the series, the following rewrite rule and selection function ("SELECTDURATION") could be defined.

```
[ DURATIONS OF SERIES ->  < SELECTDURATION >
    duration 1 . duration 1 * 3 . duration 2 . duration 2 * 3 ]
```

```
function selectduration
   objcnt=objcnt+1    ;! a global variable counting objects
                      ! in the series
   if objcnt <= 6 then
      if objcnt = 6 then result=2 else result=1 finish
   else
      if objcnt = number of objects in series then
          result=4
      else
          result=3
      finish
   finish
end
```

A set of rules for generating the duration structure of the third SERIES in each STRUCTURE could also be defined. Using a finite-state transition matrix with four duration values, the below meta-production rule includes in the strings that it generates the duration pattern of the third series in the Schoenberg composition. (Using a meta-production rule ensures that

the generated string (rhythm) is the same for both parts of the canon.)

```
" DURATIONS FOR THIRD SERIES ->
   ( duration 2 (♪), . 3 . 1 . 0 . 0 )
   ( duration 1 (♩), . 0 . 0 . 2 . 2 )
   ( duration 3 (𝅗𝅥), . 0 . 1 . 0 . 0 )
   ( duration 4 (𝅝), . 1 . 0 . 0 . 0 ) "
```

The object structure generated by the grammar consists of 'abstract tokens'. The objects must be defined. For example, one could map each of the note-objects onto one of twelve notes - eg. 'obj1' = 1 (C), 'obj2' = 2 (C#), and so on. For the duration structure, one could define the smallest duration value (eg. 'duration 2') as a 1/16th note. Using a 'time-counter' an 'event' could be represented as:

OBJECT-NUMBER (MEASURE-NUMBER, ENTRY)

If the time-signature is 3/4, then "ENTRY", a variable for 'entry-time', is modulo 12, for 12 16ths per measure where the entry-time is given as the Nth 16th note of a measure, and "MEASURE", for 'measure-number', is incremented every 12 16ths. An event description could be generated by calling an object-mapping routine and then a duration mapping routine.

For the third series' durations, a special mapping routine which performs manipulations on the rhythmic values is defined. As in the Schoenberg, the 12-note "SERIES" is divided into four sub-groups. The first four values are output in the normal manner. For the second sub-group the values of "MEASURE" and "ENTRY" are reset so that the first two sub-groups occur simultaneously in two parts. For the third sub-group, the values

of "MEASURE" and "ENTRY" are manipulated so that the order of the four notes' performance is retrograded - and the last, the 9th of "GROUP", note's duration is tripled.

This grammar generates a large number of structures including Schoenberg's Trio. The GGDL grammar program and the MDL morphological mapping program used for this example may be found in Appendix 2. The first composition generated using this grammar and mapping program, ie. essentially a random selection from possible generations, is given in Figure 4-3a. It is transcribed in Figures 4-3b and 4-3c.

The structure generated by the grammar may be mapped onto any defined set of 'real' objects. It would be quite possible to map the generated structure to a representation other than that given in Figure 4-3a. For example, it could be mapped to a format that could be used for synthesis. Alternatively, rather than map the 12 objects of the structure onto the 12 pitches of the chromatic scale, a set of noises could be used. In Figure 4-4 is the score of a composition generated using the same grammar that generated the the composition in Figure 4-3, though with only nine objects and mapped for performance by the non-standard synthesiser.

VOICE1	VOICE2
12(1,3)	12(2,3)
10(1,5)	2(2,5)
4(1,7)	8(2,7)
1(1,9)	11(2,9)
8(1,11)	4(2,11)
2(2,1)	10(3,1)
5(2,7)	7(3,7)
11(2,8)	1(3,8)
6(2,9)	6(3,9)
7(2,10)	5(3,10)
3(2,11)	9(3,11)
9(2,12)	3(3,12)
S(3,1)	S(4,1)
9(3,3)	3(4,3)
3(3,5)	9(4,5)
11(3,7)	5(4,7)
12(3,9)	6(4,9)
7(3,11)	1(4,11)
1(4,1)	7(5,1)
4(4,7)	10(5,7)
10(4,8)	4(5,8)
5(4,9)	11(5,9)
2(4,10)	8(5,10)
8(4,11)	2(5,11)
6(4,12)	12(5,12)
S(5,1)	S(6,1)
6(5,3)	3(6,3)
4(5,5)	9(6,5)
10(5,6)	1(6,6)
7(5,10)	12(6,10)
2(5,3)	5(6,3)
8(5,9)	11(6,9)
11(5,11)	8(6,11)
5(6,1)	2(7,1)
12(6,7)	7(7,7)
S(6,9)	S(7,9)
1(6,6)	10(7,6)
9(6,2)	4(7,2)
3(6,1)	6(7,1)
S(7,1)	S(8,1)
9(7,3)	6(8,3)
3(7,5)	8(8,5)
7(7,7)	2(8,7)
6(7,9)	5(8,9)
11(7,11)	10(8,11)
5(8,1)	4(9,1)
2(8,7)	1(9,7)
8(8,8)	7(9,8)
1(8,9)	12(9,9)
4(8,10)	11(9,10)
10(8,11)	3(9,11)
12(8,12)	9(9,12)
S(9,1)	S(10,1)

Figure 4-3a: Data generated by the computer.

Figure 4-3: A structure generated from a grammar that includes in the 'language' it defines the Schoenberg Trio of Figure 1.

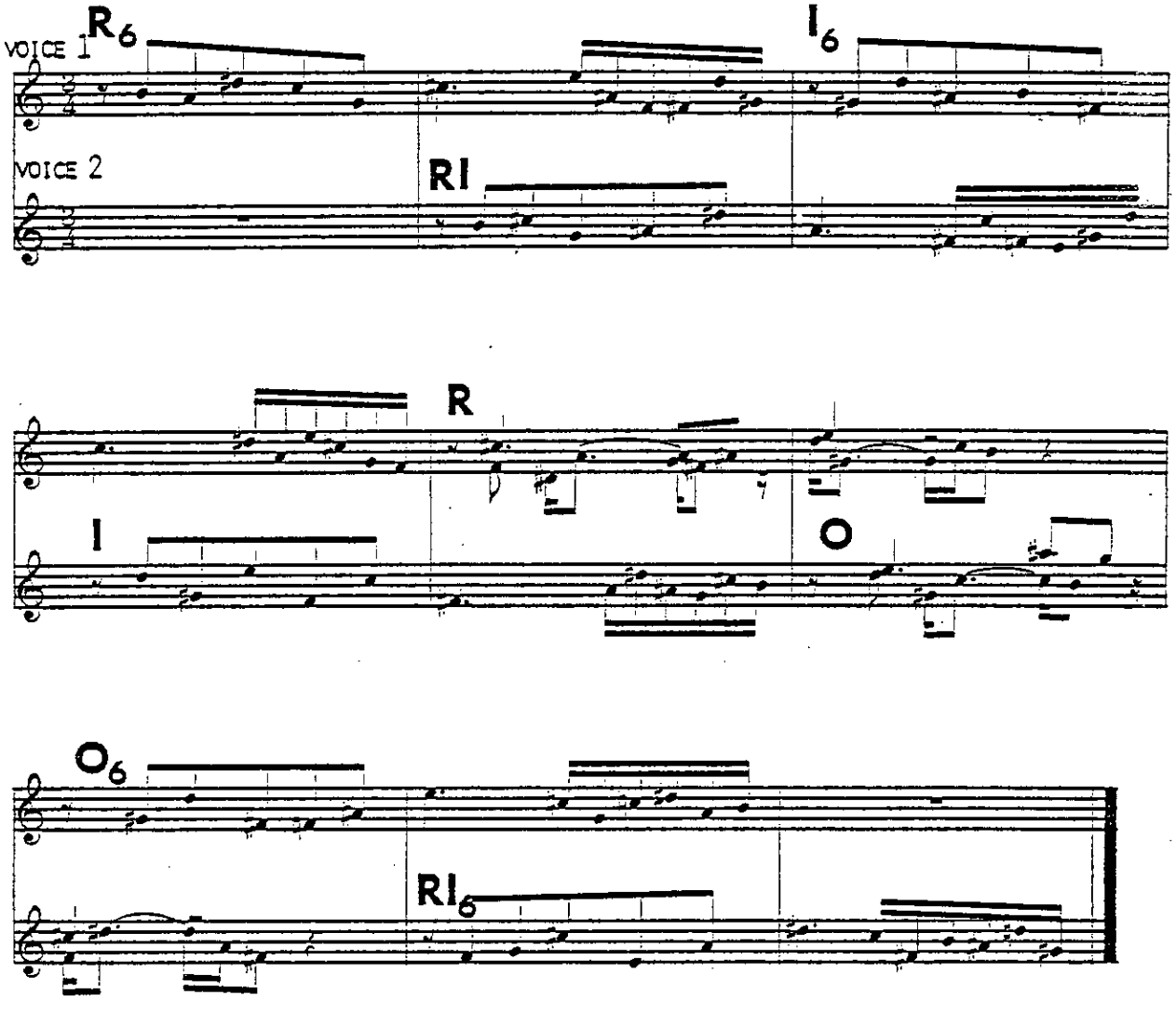


Figure 4-3b: A transcription of the data generated by the computer.

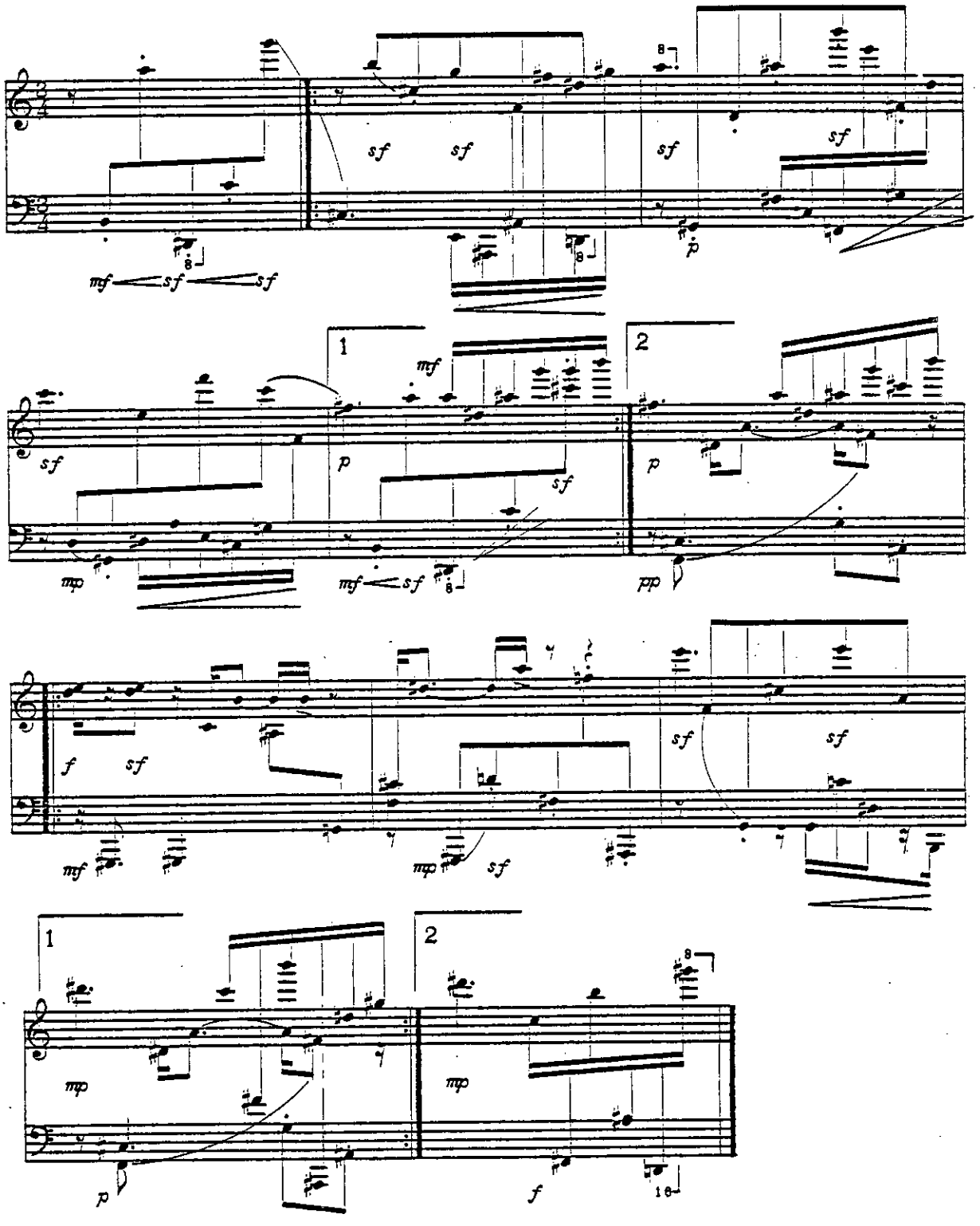


Figure 4-3c: An alternative transcription of the data. 'After Schoenberg' by S.R. Holtzman.

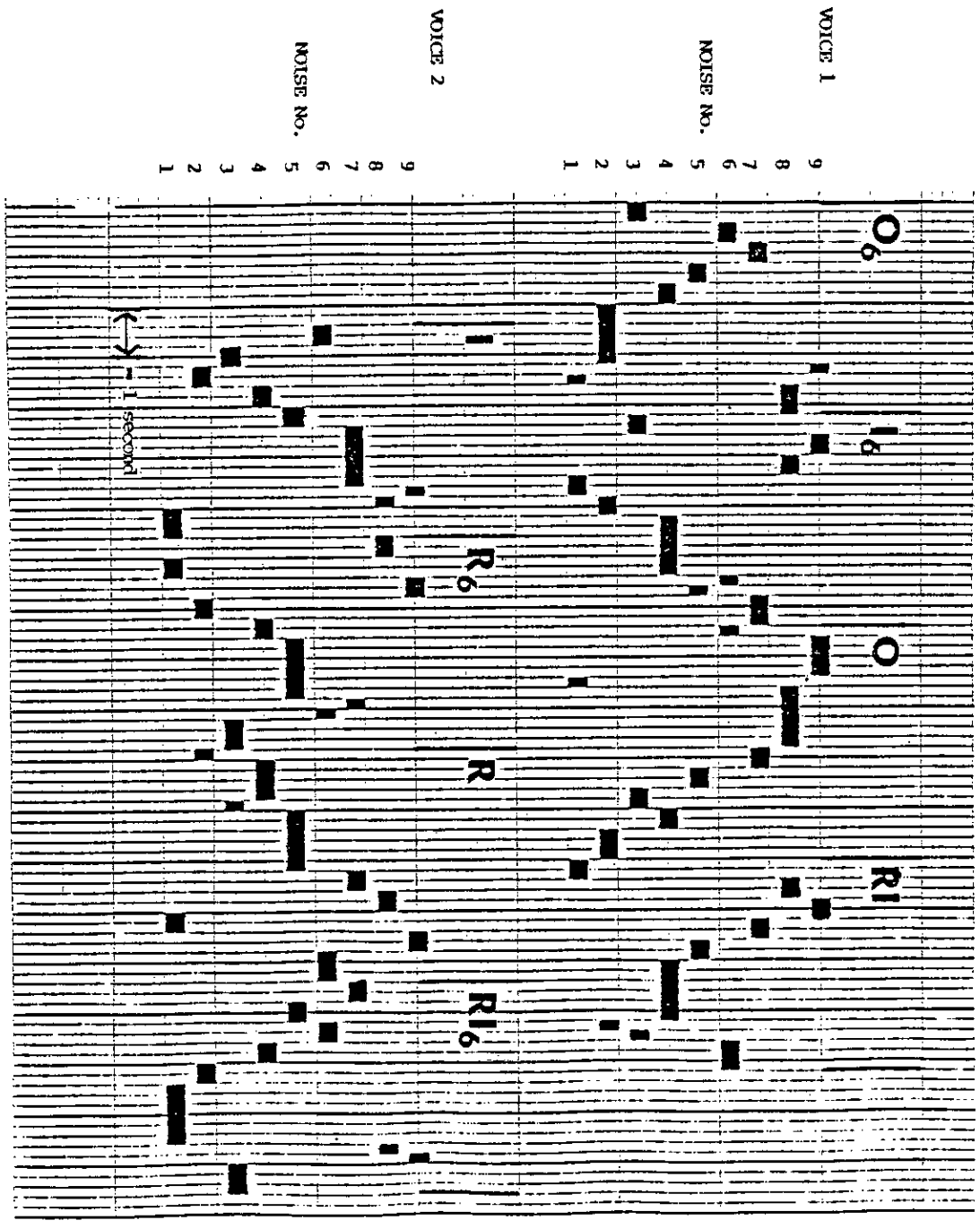


Figure 4-4: Score for structure generated for PDP-15 Non-standard Digital Synthesis Instrument (see Chapter 6) using the grammar used to describe the Schoenberg Trio (Figure 4-2).

David Hamilton's "Four Canons"

In the case of this example, rather than suggest how GGDL may have been used by composers to compose their compositions, the work of a composer who used GGDL to compose is discussed. David Hamilton composed and synthesised four compositions using the GGDL composition and synthesis suite of programs at Edinburgh University, Department of Computer Science for a BBC commission during April-June 1980. "Four Canons" is one of the four compositions. The other three compositions were realised in a similar manner to "Four Canons" though using material generated with different grammars.

In "Four Canons", David Hamilton explores the 'modalities' created by using alternative interval systems to the chromatic 12-tone tempered scale traditionally associated with western music. Hamilton, long interested in alternative systems of proportions for relating frequencies, had had no experience of computers. The use of the computer offered Hamilton the possibility of the extremely accurate performance of arbitrary frequencies for very short time intervals unattainable by other means.

Intervallic systems are founded on proportionally related frequencies. For example, the traditional untempered western melodic scale is based on a series of proportions:

1:1 unison

2:1 octave

3:2 fifth

4:3 fourth

5:4 major third and minor sixth

6:5 minor third and major sixth.

For "Four Canons", a number of morphemes were used in grammars to represent frequencies related by a proportion to the previous frequency. These morphemes were labeled as 'pro1', 'pro2',...'proN', representing such proportions. A string of frequencies could then be defined in terms of a starting frequency and a series of proportionally related frequencies. Hamilton defined rewrite rules to generate such frequency strings as the basic material of "Four Canons". For example,

[FREQUENCY STRING -> STARTNOTE, pro1, pro2, pro3,...proN]

The strings of basic material were completely specified in Hamilton's work. In the case of "Four Canons", the thematic material was based on carefully worked out predetermined intervallic patterns. Hamilton called such a pattern a "SPIRAL".

A complete "SPIRAL" consisted of a starting frequency, indicated by a morpheme 'note1, note2,...noteN', an overall duration, indicated by a morpheme 'dur1, dur2,...durN', and a string of 25 proportionally related frequencies. The proportions were inverted in certain patterns in order to ensure that the frequencies generated stayed within certain frequency ranges. The inversion of a proportional frequency relationship, eg. 5:9 to 9:5, results in the same though inverted interval. Spirals with

different characteristics could be defined by manipulating the inversions of interval proportions in different ways and were indicated by spiral-type morphemes ('spiral1, spiral2,...spiralN'). A rewrite rule to generate a spiral could be defined.

[SPIRAL -> noteN, durN, spiralN, FREQUENCY STRING]

A simple crab canon structure is defined by the following rules.

Rules to Generate a Crab Canon of Spirals

```
[ COMPOSITION -> PART1, PART2 ]
[ PART1 -> VOICE1, MELODY ]
[ PART2 -> VOICE2, @B (MELODY) ]
[ MELODY -> (dur1, note1, OBJECT7)
            (dur2, note2, OBJECT3)
               . . . . . .
            (dur1, note6, OBJECT1) ]
[ OBJECT1 -> spiral1, FREQSTRING ]
[ OBJECT2 -> spiral2, FREQSTRING ]
   . . . . . .
[ OBJECT7 -> spiral7, FREQSTRING ]
[ FREQSTRING -> pro1, pro2, pro3, pro1, pro4,...pro2 ]
```

Using these types of grammars, Hamilton generated various strings of spirals. The first notes of the spirals were used to play a melody as a 'cantus firmus'. For this, Hamilton used the melody of a hymn tune. The durations (ie. speed of execution) of the strings of spirals based on the cantus firmus were then related to each other in terms of the same proportions that were used to define frequency strings.

Hamilton used grammars to generate extended but completely determined structures. By varying the rules, the orders of proportions in the frequency strings, the basic note pattern of the cantus firmus, and so on, different types of structures could be generated and explored. However, these structures are still

abstract. Though the generated strings of morphemes represented structures with carefully calculated proportions, what the proportions actually were defined as remained, in the grammatical definition of the structure, undefined. Given generated structures, it was possible in the mapping program to define the values of the proportions independently of the structures and their generation. Hamilton experimented with trying different proportions. He tried, for example, generating strings of frequencies using the proportions used in the untempered western chromatic scale (to generate untempered melodies rather than the traditionally tempered scales used in the performance of western classical music), as well as using proportions based on other numerical relationships, such as Fibinacci series (eg. 3:1, 4:3, 7:4, 11:7, etc.), a series of odd 'harmonics' (3:1, 5:3, 7:5, 9:7, 11:9, etc.) and so on. Each alternative definition required changes only to the values of variables in the mapping program.

In Hamilton's "Four Canons", the hymn tune elaborated with spirals required, for synthesis, the specification of 286 notes and 286 durations. In fact, Hamilton generated strings of somewhat more complexity. Thus, though he used the grammar in an entirely deterministic fashion, it was possible for him to express complex structures in a compact and convenient representation. Variations of the structure could be easily explored by altering the rewrite rules and variations of the structures' realisation could be explored by changing the mapping definitions of the morpheme-objects.

GGDL provided David Hamilton with a useful tool for aiding him in composing "Four Canons". In the process of composition he was able to investigate the potentiality of his compositional ideas by manipulating the grammars and mapping programs, a task easily achieved by editing the GGDL grammar and mapping definition programs. The calculations necessary to generate a composition based on his compositional rules could be quickly carried out by the computer and a representation of the structure which could be immediately performed for evaluation generated. David Hamilton did not find the use of grammars a conceptually alien way of representing his ideas, that is, his ideas could be readily adopted to the GGDL representation of composition processes. He managed, in the short period, to develop a sufficient understanding of grammars to define structures himself using rewrite rules, though, the programming of mapping and other routines was done by the author with variables defining proportions easily altered and edited by Hamilton.

Compositional Considerations

A problem for any composer or researcher using a generative system such as GGDL is making explicit a set of rules to limit the systems generation. The system potentially can generate an infinite number of different compositions, most of which will be of little interest to composers. For example, a problem David Hamilton had was determining which of all the possible types of proportions he could use, made compositional sense to him. The actual effect of using certain intervallic systems was often

difficult to imagine and Hamilton selected between systems by generating structures and subjectively evaluating them, rejecting certain systems and accepting others. It was a 'trial and error' process dependent on feedback.

David Hamilton used the grammars in an entirely deterministic manner. In the case of the rules used to describe the Schoenberg Trio, the grammar was not deterministic. For example, both the compositions in Figures 4-2 and 4-3 are generated from the same grammar. A large number more will also be generated. The question a composer must consider is how to limit the generation to those compositions which he will consider acceptable. This can be done by experimentation with a grammar, checking its output and modifying the rules until an acceptable grammar is written - or by rejection of output from the machine. In the former case it is necessary to ask, and formalise, what the differences between the two compositions are, which is the more interesting and why. Additional rules could be used to further specify, and limit, the structures defined by the grammar.

The composition in Figure 4-3 is in fact the first work generated by the grammar; it is a random selection from the possible output of the grammar. The series generated as the basis of the structure by chance has repeated occurences of semitone and diminished fifth intervals, intervals a serial composer might well compose with. This is a coincidence rather than determined by the rules of the grammar. As an alternative one could define a transition matrix, for example, which ensured that only certain

intervals occurred.

There are in the Schoenberg composition a number of idiosyncracies. For example, the Bb tied to the last measure is the only tied note; and in the fourth occurrence of the series in each part of the canon, the eight note of the series is played before the seventh. The problem arises as to the extent to which such irregularities should be formalised. That is, should such irregularities be considered idiosyncratic to the particular realisation or should they be considered stylistic: are they 'ad hoc' improvisations or irregularities consistent with a set of compositional rules? For example, could the tied B-flat of the R6 series (last measure) be derived from a rule or was it some idiosyncratic decision that caused Schoenberg to write it as it is? Would it be equally suitable to tie the B natural of the last measure of the composition in Figure 4-3? The occurrence of the eighth element of the series before the seventh in the last series of each voice could be described in rules -

```
IF GROUPCNT=4 THEN
  IF OBJCNT=7 THEN
     et cetera
```

But should this be done for all structures generated, ie. would the rule be applicable with all series? - or does Schoenberg want to avoid the fifth in this particular context, and why just here? Some of these problems are problems of 'recreating' what Schoenberg has written and are problems for a musicologist. However, they are also problems of the composer in formalising what may be his seemingly irregular compositional method.

Another problem in formalising the composition process is deciding to what extent to formalise it. In the case of Hamilton's "Four Canons", the composition generated was to be directly performed by a synthesiser. All data necessary for the performance of the piece had to be generated from the grammar and mapping definitions. However, in the examples generated using a grammar that included the Schoenberg Trio in its 'language' (eg. Figures 4-2 and 4-3), only the note-duration structure was specified. The output of the machine leaves considerable room for interpretation - the octave positioning, instrumentation, dynamics, phrasing, etc. are not specified in the output. One could possibly add these as part of the grammar, if rules can be formalised, or leave the data 'interpretable'.

GGDL may be used by composers as an aid in the process of composition. However, the system does not specify what needs to be defined and attempts to impose as few restrictions on a composer as possible. However, it is this very lack of restrictions, ie. leaving the problem open ended, that may make the use of the system difficult. Unlike in Koenig's and Xenakis' systems, the composer must define his own orientation, specify what it is the computer will be used for and how, and so on. The GGDL-CAC system is intended as a general aid. However, this generality itself may pose problems for a composer.

Summary

It is clear that generative grammars can be used to describe

and automate the generation of music structures. Generative grammars may act as a powerful and useful aid to composers, permitting the description and generation of many different music languages. The separation of the generation of an abstract structure from its definition during the process of mapping can also be usefully exploited. Structures generated using the same grammar may be mapped to representations permitting performances by different means (cf. Figures 4-2, 4-3 and 4-4). Perhaps in a more subtle manner, the mapping process may be used as in the example of David Hamilton's "Four Canons", where a structure of proportions was defined in the abstract independently of the definition of the proportional values. The mapping process allows the composer to define his compositional object.

Though grammars have been presented which describe and generate three compositions, this is not to suggest that any compositions could be described and generated using GGDL, certainly not easily. Certain types of relationships are not easily described using GGDL. For example, as the process of generation works only with abstract objects, temporal relations, ie. relations in the structure after mapping, are not easily accounted for during the process of generation. Thus, though one can, as in the above examples, express linear relationships, and even generate countrapuntal textures where the separate voices are either 'self-synchronised' or are not strictly synchronised during generation, it would be difficult to generate certain types of contrapuntal textures with independent linear and vertical harmonic relationships.

Generative grammars may be used as an aid by musicologists as well as composers. Musicologists studying the formal structure of a composition can use grammars for its description; given such a description its adequacy can be tested by using it for generation and comparing results against the original. An interesting musicological question arises when the same structure may be represented and generated by more than one grammar; which representation is a correct interpretation of the structure becomes a question of reconstructing the actual composition process used to generate the work.

Chapter 5: Composing at the micro-level

An interesting possibility with a computer aided composition system is that of organising micro-sound structures by techniques similar to those used for macro-structural organisation. Just as one can define a set of rules for generating relationships between objects which are 'whole' events, one can define sets of rules for generating descriptions of events at a micro-level. Both Koenig and Xenakis designed programs which, using the compositional rules they had programmed for macro-structural organisation, generated micro-sound structures, structures of digital samples of very short durations. The grammar system described in this thesis manipulates abstract tokens. These can be complete sections of a work, the notes or objects that make up a section, or the micro-components of those objects.

In the previous chapter, it was shown that generative grammars can be used to describe and generate different types of music structures and that the object of the composition process can be defined independent of the generation process. In this chapter, the possibility of extending the application of compositional rules to manipulate objects to describe sound structures at the micro-level is considered. Using GGDL, one need only define micro-objects in the mapping process. In this chapter, the considerable advantage of the possibility of defining the object of the composition process in a mapping definition is further demonstrated.

In this chapter, how grammars were used to generate the sounds of a composition by the author, "After Artaud" (Holtzman 1978), is discussed. G. M. Koenig, during a visit to the Department of Computer Science, University of Edinburgh to experiment with GGDL (June 30 - July 12, 1980), developed some ideas on how grammars might be used to generate symmetrical waveforms. These ideas are also briefly looked at.

After Artaud

Rather than describe sounds in terms of sample relationships, as in the experiments of Xenakis and Koenig's SSP program, one could use a parametric synthesis model for the generation of sounds. For the composition of "After Artaud", the frequency modulation model (Chowning, 1973) was used for describing sounds. Using frequency modulation, a number of parameters may be dynamically assigned values which control the generated sound. These are the frequencies of the carrier and modulating oscillators, the amplitude of the carrier wave, and the modulation index, which determines the 'depth' of modulation. The dynamic values for each of these parameters may be represented by envelopes.

Using GGDL, one could define a set of rules for generating such envelopes. An envelope, for example, could be described as consisting of an attack, a steady-state, and a decay.

```
[ ENVELOPE -> ATTACK, STEADY-STATE, DECAY ]
```

The components of the envelope could then be defined. The

envelope could, for example, be represented by a set of points (x,y) in a Cartesian plane, representing the turning points of the envelope. Different qualities of 'ATTACK' could then be described in terms of the gradient between the starting point of the envelope, eg. (0,0), and the point at which the steady state of the envelope begins. For example:

[ATTACK -> steep . moderate . shallow]

Initialising the first envelope point to "X=0" and "Y=0", a steep attack could be described as 'X + a small increment' and 'Y + a large increment':

X = X + random(200,400)

Y = Y + random(2000,3000)

with an envelope window of, say, 4000 by 4000. 'Steep' refers to a class of attacks which may be realised within a range of gradients (see Figure 5-1). Similar rules could be defined for generating different types of steady-states and decays.

For the composition of "After Artaud", a grammar was defined for generating different types of envelopes and then, at a higher level, rules were given for combining different envelopes for the different parameters of a frequency modulation representation of a sound. Different types of envelopes were defined by their quality of attack, steady-state and decay. For example, one type of envelope might have a steep attack, short steady-state and slow decay whilst another might have a shallow attack, long steady-state and rapid decay. Different types of sounds are generated with different combinations of envelopes. For example, bell sounds have for both their modulation index and amplitude

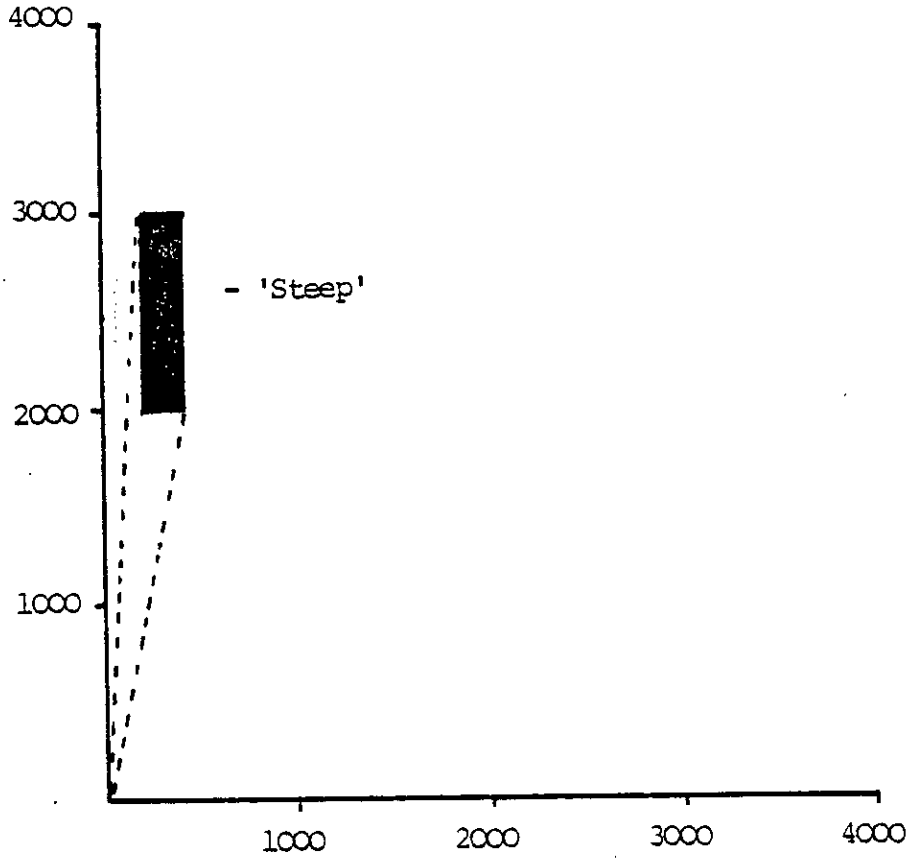


Figure 5-1: 'Steep' refers to a class of attacks which may be realised within a range of gradients.

envelopes, envelopes with very steep attacks and very gradual decays.

An 'abstact-structure', the structure of "After Artaud" derived from an Artaud poem, was represented as a string of non-terminals each representing a single sound. The non-terminals were rewritten as a string of terminals representing a set of envelopes, defined as (x,y) turning points, defining a sound that could be synthesised by frequency modulation synthesis. For each unique sound-object in the macro-structure, ie. non-terminals that occurred only once, the computer generated a distinctive set of frequency modulation characteristics - this was a unique set of carrier frequency, modulation to carrier frequency ratio and amplitude and frequency modulation index envelopes. For sounds that were repeated the computer generated the same terminal representation of a description of the sound. However, during the mapping of the terminals, different values which would still have the same characteristics were generated. If a sound recurred in the macro-structure (eg. 'obj1' in 'obj1, obj2, obj3, obj1'), <u>each</u> occurrence of the sound was realised differently. For example, if the modulation index of 'obj1' was described as 'steep' attack, 'short' steady-state and 'shallow' decay, the actual increment (ie. steepness) for the envelope <u>for each occurrence</u> of 'obj1' was recalculated each time 'obj1' was mapped - ie. realised - as a 'real' frequency modulation sound object. One could therefore get two modulation index envelopes with the same characteristics (perceptually identifiable as the same) yet slightly differing (see Figure 5-2). Every time a trumpet plays a C#, every time I

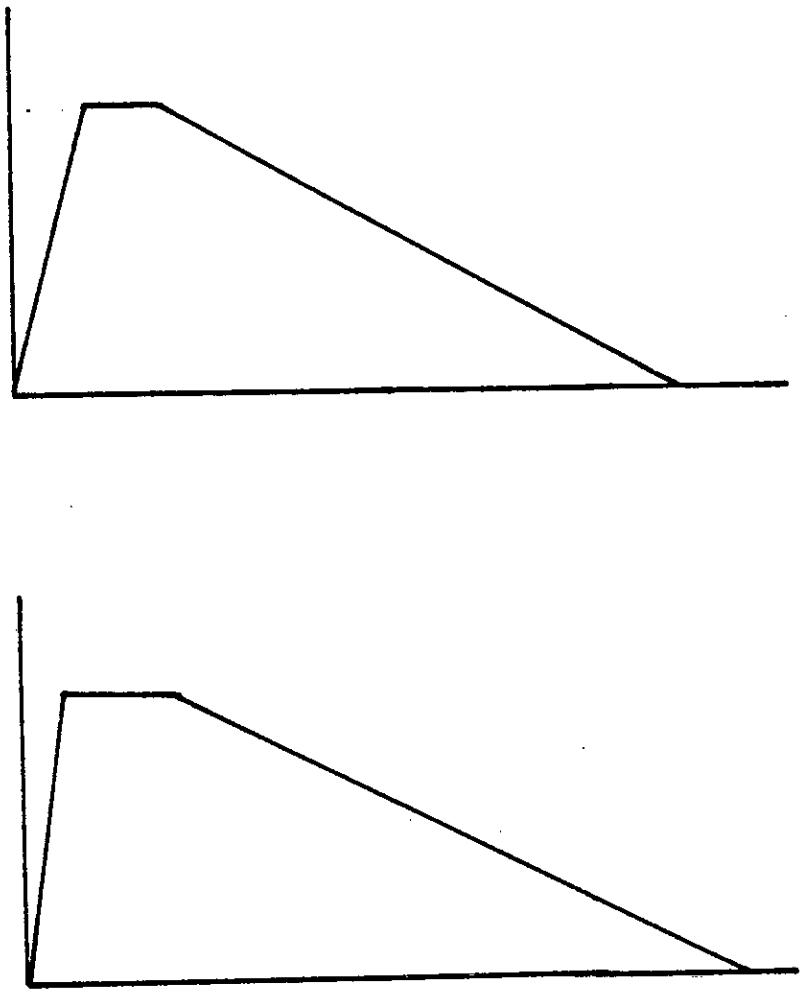


Figure 5-2: Two envelopes with the same characteristics yet slightly differing: e.g., steep attack, short steady-state, and shallow decay.

pronounce the phoneme [ae], and every time the machine synthesises 'obj1', slightly varying envelopes for parameters are generated though the object still remains distinctive. It is still heard as C#, [ae] or 'obj1'.

Generating Symmetrical Waveforms

During his experiments with GGDL, Koenig was interested in seeing what sorts of structures could be easily expressed using grammars. In some of his work, he tried to generate descriptions of sounds in terms of sets of amplitude and time points. His SSP program (Berg 1979) also describes sounds in terms of amplitude and time points, though symmetrical waveform structures such as those Koenig defined using GGDL could not easily be generated using SSP.

Using meta-production rules, he noted that one could generate a fixed object from a set of rules and then perform transformations on the object. The generation of a symmetrical wave could then be represented as a generated structure of time and amplitude points, followed by the same structure pivoted (ie. inverted) around a zero-crossing point with the order of the points reversed. In GGDL, a period of a symmetrical waveform could be represented as a structure followed by itself inverted (@I) and reversed (@B).

To generate symmetrical waveforms using GGDL, Koenig independently generated an amplitude structure and a duration structure for a period of the wave, where each of these structures

was a fixed object followed by itself inverted and reversed. The amplitude and duration structures were each generated using a metaproduction rule. The amplitude structure was generated using a selection procedure that permitted the definition of a dynamic mask over possible selections similar to the 'Tendency' procedure Koenig defined in his own composing programs (cf. Chapter 2). A set of rewrite rules defined by Koenig which could be used for generating symmetrical waveforms follows.

Rules for Generating Symmetrical Waverforms

```
" RANGE -> # N # < TENDENCY >
  p1, . p2, . p3, . p4, . p5, . p6, . p7, . p8, . p9, . p10, . p11, .
  p12, . p13, . p14, . p15, . p16, . p17, . p18, . p19, . p20, "

" SET -> # N #
   t1, . t2, . t3, . t4, . t5, . t6, . t7, . t8, . t9, . t10, . t11, .
   t12, . t13, . t14, . t15, . t16, . t17, . t18, . t19, . t20, "

[ PERIOD -> @M(AMP)(TIME) ]
[ AMP -> PO, GROUP, @I(PO, @B(GROUP))]
[ GROUP -> RANGE]
[ TIME -> t1,SET,@B(t1,SET) ]
```

Examples of some of the waveforms generated using this grammar are given in Figure 5-3.

Summary

GGDL may be used for the definition and generation of micro-sound structures as well as macro-sound structures. The components which describe a micro-sound structure, like those used in the previous chapter for the description of macro-sound structures, may be arbitrarily defined. One can compose envelopes, sound descriptions in terms of frequency modulation, waveforms, and so on.

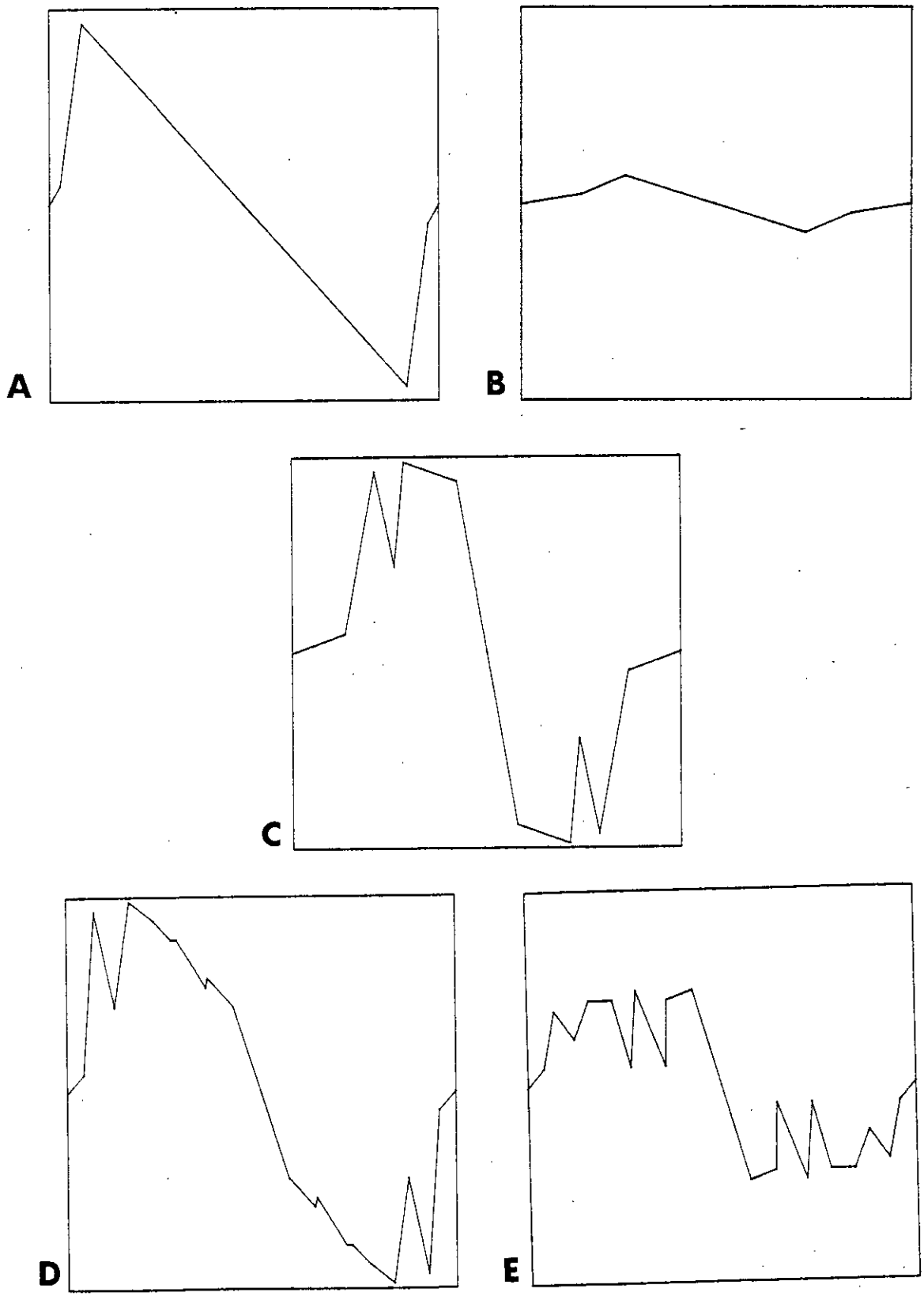


Figure 5-3: Symmetrical waveforms generated using GGDL.

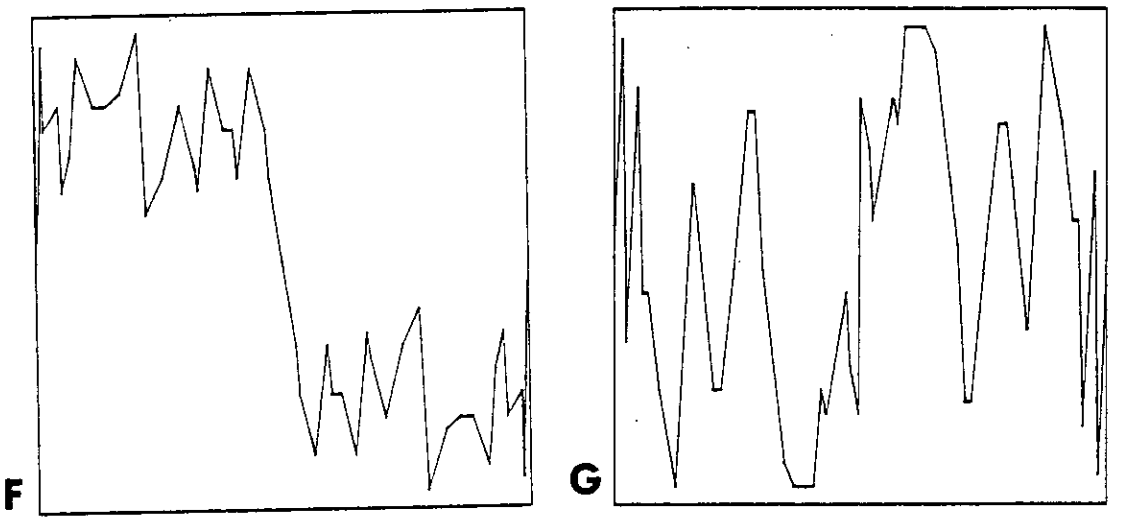


Figure 5-3: Symmetrical waveforms generated using GGDL.

If half of a symmetrical waveform is defined by a 'GROUP' of points in a cartesian plane, the other half may be generated by 'reflecting' the points around a zero crossing. The qualities of only one half of the waveform need to be defined.

```
[WAVEFORM AMPLITUDE POINTS  → PO, GROUP
          @I(PO, @B(GROUP))]
[GROUP → ...]
```

In the examples, one half of the waveform has been defined by

A. two (positive) points
B. two points in a restricted range
C. five points
D. ten points
E. ten points in a restricted range
F. twenty five points
G. twenty five points over full positive and negative range

Using GGDL it is possible therefore, to use the same generation rules independent of the compositional objects defined in mapping. This has interesting compositional possibilities. For example, both a macro-structure and the micro-components that define that structure may be composed by the same rules. Such possibilities are of interest to many composers and a CAC system clearly may have a greater attraction if it is sufficiently flexible that composers may explore such possibilities.

The work of Koenig, in this respect, produced some interesting results. It demonstrated that symmetrical structures could easily be expressed using GGDL. In the examples in this chapter, such structures were generated with a selection procedure that could also have been used to generate macro-compositional structures. Similarly, the representation of a waveform as a structure and its retrograde inversion is a commonly used macro-structural object, eg. a series or theme and its retrograde inversion. In GGDL, either could be generated from the same rules, for example, those defined by Koenig, and only the mapping program would need to be changed to redefine the compositional object.

Chapter 6: An Automated Digital Sound Synthesis Instrument

In this chapter, an 'automated non-standard digital sound synthesis instrument' (Holtzman 1978b, 1979) is described. The instrument is a digital synthesiser which generates sounds by a 'non-standard' method of synthesis. It may be seen as functioning on two major levels: one, as a sound synthesis process, and two, as a syntactic structuring process. The synthesiser, supported by a PDP-15/40 (DEC 1969) with facilities for D-A conversion, was implemented in software, including a 'programmable' grammar for the automatic generation of non-standard sound descriptions and a special operating system for executing such descriptions concurrently - the descriptions are in the form of computer programs.

In the same manner as standard descriptions of sounds were generated by grammars in Chapter 5, the 'automated synthesis instrument' generates non-standard descriptions of sounds using a grammar. In most cases of digital synthesis, ie. standard digital synthesis (see below), sound descriptions are used to assign variables to a function expressed as a computer program; the execution of the program generates data-samples representing that sound. In the case of the non-standard synthesis method described in this chapter, the description of the sound and the program to generate data-samples are one and the same. That is, sounds are described in terms of computer programs generated automatically using a grammar.

Chapters 4 and 5 demonstrate that generative grammars may be used by composers to automate their composing process at the macro- and micro-levels. Generative grammars and computers were used as tools to facilitate an otherwise tedious process. This chapter describes a 'tool' for composers, ie. an automated non-standard synthesiser, developed as a direct consequence of 1) using computers to synthesise sound and 2) using generative grammars to compose music structures.

Standard vs Non-Standard Synthesis

Standard synthesis systems such as Mathews' (1969) MUSIC V provide facilities for working 'top-down' to specify sounds in terms of high-level acoustic models. Complex calculations are used to simulate these models and generate digital samples representing the specified sound. With the exception of cases where these calculations are performed using specially designed hardware (Di Giugnio 1977), calculations are not in 'real-time' and a long list of signal samples is stored; synthesis subsequently consists of the transfer of stored samples to digital-to-analog convertors (DACs).

The sound synthesis instrument described in this chapter rests on a 'non-standard' or instruction synthesis approach to digital signal generation. Based on digital processes, synthesis is built around a technique of applying sequences of virtual machine instructions to samples moving through an accumulator register. Instruction synthesis samples are related only in terms of the

virtual machine instructions that have been used to generate the samples. For example, in such a system, one sample may be related to the previous two samples by an exclusive-or instruction. These relationships, ie. the programs of virtual machine instructions, are defined without reference to some acoustic model or function.

The synthesis possibilities are considered 'idiomatic' to the extent that the technique is limited and 'tuned' to the architecture of a particular cpu which supports the virtual machine. The same non-standard description of a sound when executed by different machines with different word sizes, different implementations of multiplication or shifting, etc. may generate a different set of samples representing that sound. If the supporting machine can execute the virtual instructions in 'real-time', the durations of the samples may be dependent upon the instructions required to generate them. The sample time will be a result of the actual program structure, that is, the program text required to generate a sample, and the machine speed.

In the case of the implementation described in this chapter, the PDP-15/40 (DEC 1969) is an 18-bit computer with an instruction set which includes various arithmetic and logical operations. An operating system has been designed and implemented to execute two non-standard functions simultaneously in 'real-time'. There is no external clocking and the sample time is dependent on the program structure of the sound description and the machine speed. When two functions are executed concurrently, the sample times of samples generated by each function are affected by sample times of

the other function.

Instruction synthesis is non-standard in the following senses: first, the noises this technique tends to generate differ greatly from those of the traditional instrumental repertoire and even from much electronic music; second, in this technique, sound is specified in terms of basic digital processes rather than by the rules of acoustics or by traditional concepts of frequency, overtone structure, and the like.

Sound synthesis programs such as MUSIC V exemplify the standard approach. With the instruction synthesis methodology, research is neither abundant nor well-documented. Paul Berg's ASP (1975) and PILE2 (1979) implement an instruction synthesis approach, as does the system described here. From a slightly different approach, relating samples to one another by means of a hierarchy of virtual machine instructions is Koenig's SSP (Berg 1978b, Banks 1979). Yet another approach which departs from standard frequency/overtone acoustic models has been implemented by Xenakis (1971) using stochastic techniques to generate samples.

A premise fundamental to Berg's and our experimentation is that the programs should

> "explore the idiomatic capabilities of the computer in the realm of sound synthesis...systems where the computer is essential for a reason other than the magnitude of the task. Where it could contribute to production of new sorts of sounds. Or processes for producing sound. Or at least new representations of sound" (Berg, 1975).

Berg's ASP is implemented on a PDP-15/20 (DEC 1969) at the Institute of Sonology, Holland. Using the machine instruction set of the PDP-15, Berg wrote MACRO-15 (DEC 1969b) assembler programs in which sequences of words, ie. samples, are generated to produce noise timbres. Each string of samples so produced will, given certain constraints discussed later, produce a different waveform, ie. sound, which may be either periodic or aperiodic. Given small programs to generate sounds, they are executed in different orders to produce a continuum of juxtaposed timbres and silences. Rather than starting with an idea of a sound and then simulating it, in the ASP program it is by programs of instructions that a sound is described.

The ASP program was written 'manually' in MACRO-15 assembly language. Using a generative grammar, a synthesis process was developed in which small program texts for the synthesis of sound are automatically generated.

The Program Generator

The Program Generator is a set of GGDL programs, ie. a grammar program and a mapping definition program. In the grammar program (see Appendix 3), a set of values is assigned to variables in the control mechanism of the grammar and programs of virtual machine instructions are then automatically generated. A mapping program 'compiles' the virtual machine code to executable machine code for a PDP-15/40. By changing the mapping program, the virtual machine instructions used in this specific system could be simulated

(possibly not in real-time) by other actual machines; the same virtual program executed on different machines might generate different sounds.

The grammar and mapping definition programs of the Program Generator together generate texts (in compiled machine code) which, when executed, create distinctive sounds. These texts are called 'Functions': a series of machine instructions which will access and manipulate data structures (via the accumulator) and place computer words (numbers) in the accumulator which are to be sent to a digital-to-analog convertor. The series of instructions in a Function is finite and of a fixed ordering. In the generation of samples, the sequences of instructions in a Function will be repeated, since the last instruction of a Function is always a 'jump to' the first instruction of the Function. This is ensured by a rule in the grammar by which a well-formed Function is defined as ending with a jump to the first instruction of the Function; this is necessary if the execution of the code generated is to yield continuous sounds of unspecified duration. Depending on whether the instructions include assignments to internal variables or whether the instruction RANDOM is used, a Function may produce either periodic or aperiodic waveforms.

In generating sound-producing text the Program Generator first determines the number of constants and variables a Function will use. Given these data objects, a Function, consisting of sequences of instructions which will produce samples, is generated by a grammar. In the grammar of the Program Generator, 12

primitive instructions, terminals in the grammar, have been defined. These are:

(1)	+	
(2)	-	arithmetic operators
(3)	*	
(4)	/	
(5)	LAC	memory retrieval operator
(6)	RANDOM	random number generator operator
(7)	CONJUNCTION	
(8)	ANTIVALENCE	
(9)	DISJUNCTION	
(10)	EQUIVALENCE	logical operators
(11)	IMPLICATION	
(12)	EXCLUSION	

The grammar is used to generate sequences of these instructions to form 'statements'. Statements may be of two types: 'assignment statements' and 'conversion statements'. Functions consist of sequences of statements.

In an assignment statement, an expression calculates a binary value which is assigned to a variable in memory. In a conversion statement, a value calculated by an expression is sent out as a sample to a DAC. An example, given the variables V1, V2, V3 and the constants C1 and C2, of an assignment statement is:

V1 = C1 * C2 + V3 CONJUNCTION V2

where operators are applied left-to-right. A conversion statement for sending a sample to a DAC is:

DAC <- V3 + C2

Memory and random instructions are in themselves expressions, eg.:

DAC <- C2 (loads the DAC with the constant C2)

or

V2 = RANDOM (assigns a random value to the variable V2)

Functions are written in two passes. First, a grammar (see Appendix 3) is used to generate a virtual machine instruction text. Then, the virtual text is 'compiled' into machine code using the mapping definition. As text is generated and mapped, lists of all the Functions, statements and data are maintained and later used by what is called the 'Performance Process' for the execution of the Functions.

The Grammars and Semantic Constraints

The possible programs (Functions) generated automatically using grammars must be constrained if, when executed, they are to make 'sense'. The grammar of the Program Generator manipulates various types of objects with different properties and values. These include, for example, statements, data structures and primitive, ie. virtual, operations. The Program Generator must embody some understanding of the effects of certain syntactic relationships to write syntactically correct and semantically 'intelligent' programs.

For example, if in a Function an assignment statement assigns a new value to the variable V1:

```
V1 = V1 + 1
```

such an assignment may be said to have semantic value only if the assignment has some resulting effect in the program. If, for example, V1 was assigned another value before it was applied elsewhere, eg.:

```
V1 = 32
```

the previous assignment would be semantically 'senseless'. The generation of such cancelling statements can be prohibited by the rules of the grammar. Constraints may, for example, be applied to the selection of the variable to which an assignment is made in a selection procedure. Though the above sequence of statements may be syntactically correct according to the rewrite rules of the grammar, the generation of the semantically senseless constructs may be avoided using a control function. (See the discussion of syntagmatic vs. paradigmatic relationships in Chapter 3.) Similarly, control functions may be used to prevent the assignment of an object to itself:

V1 = V1

Rules of the grammar may also impose a semantic consistency when generating the expressions of statements. For example, rewrite rules may prohibit certain sequences of logical operations. The expression,

A CONJUNCTION B DISJUNCTION A

results in A, ie. the same value it began with. This sort of 'senseless' instruction sequence could be prevented by prohibiting logical operators to follow one another, though this also prohibits many sensible strings. A better rule, but considerably more difficult to implement, would prohibit only the sequences of complementary logical operators with certain arguments in common, ie. those that will cancel the results of the first operation.

There are also questions of a different sort that must be considered. 'Sensible', in final analysis, may be interpreted in

terms of the perceptible results of the program texts. The perceptual processes which allow intelligibility of communicative systems are an overall constraint on all parts of a sound producing system (Holtzman 1978). For example, given a structure of relationships between a set of noises described by Functions, the Functions, when executed, must sound distinctive if the relationships are to be heard. For each Function to generate distinctive sounds, it must have distinctive features in the program text by which it is described. Each function is generated by a grammar and may be considered as an utterance in the language defined by the grammar. Ideally, the grammar from which a Function is generated would define a language whose utterances would be disjoint from any other Function-generating grammar. Provided a different grammar is used for the generation of each Function, these will have unique syntactic features which generate distinctive waveforms or sounds. Depending on what distinctive qualities the grammars have, the different waveforms generated may also be perceptually distinct.

In the case of the non-standard synthesis instrument described in this chapter, a 'skeletal' grammar has been defined for which a set of values for its variables must be defined to complete the grammar. If different values are assigned to the variables, distinct languages may be defined by the different 'complete' grammars. The variables of the grammar control a number of syntactic features. These include: the number of statements in a Function, permissable sequences of operations, which operations are to be used, the ratio of variables to constants (which

determines the periodicity of the waveform), and so forth. The grammar, written in GGDL, may be found in Appendix 3.

The Performance of Functions

What has been described so far is a system that composes individual sounds. These sounds are described in a non-standard manner in the form of Functions. The execution of these Functions produces sound.

An operating system for the PDP-15/40 was designed and implemented to permit the non-standard synthesis instrument to synthesise two 'voices' simultaneously (see Figure 6-1). Each voice is synthesised by running separate 'performance processes' concurrently, each of which controls the execution of Functions. In addition, processes for clocking the performances of the voices and communications with the control processor, the VAX-11/780 (see Chapter 7), are run concurrently with these.

A 'dispatcher' determines which process should be executed. Initially running an idling process, other processes are activated given instructions received from the control processor which are handled by the communications process. The communications process may be instructed to activate a process to write Functions or the processes required to perform Functions. If the communications process is unable to execute the instructions it responds to the control processor with an appropriate error message which will be conveyed to the user (see Appendix 4).

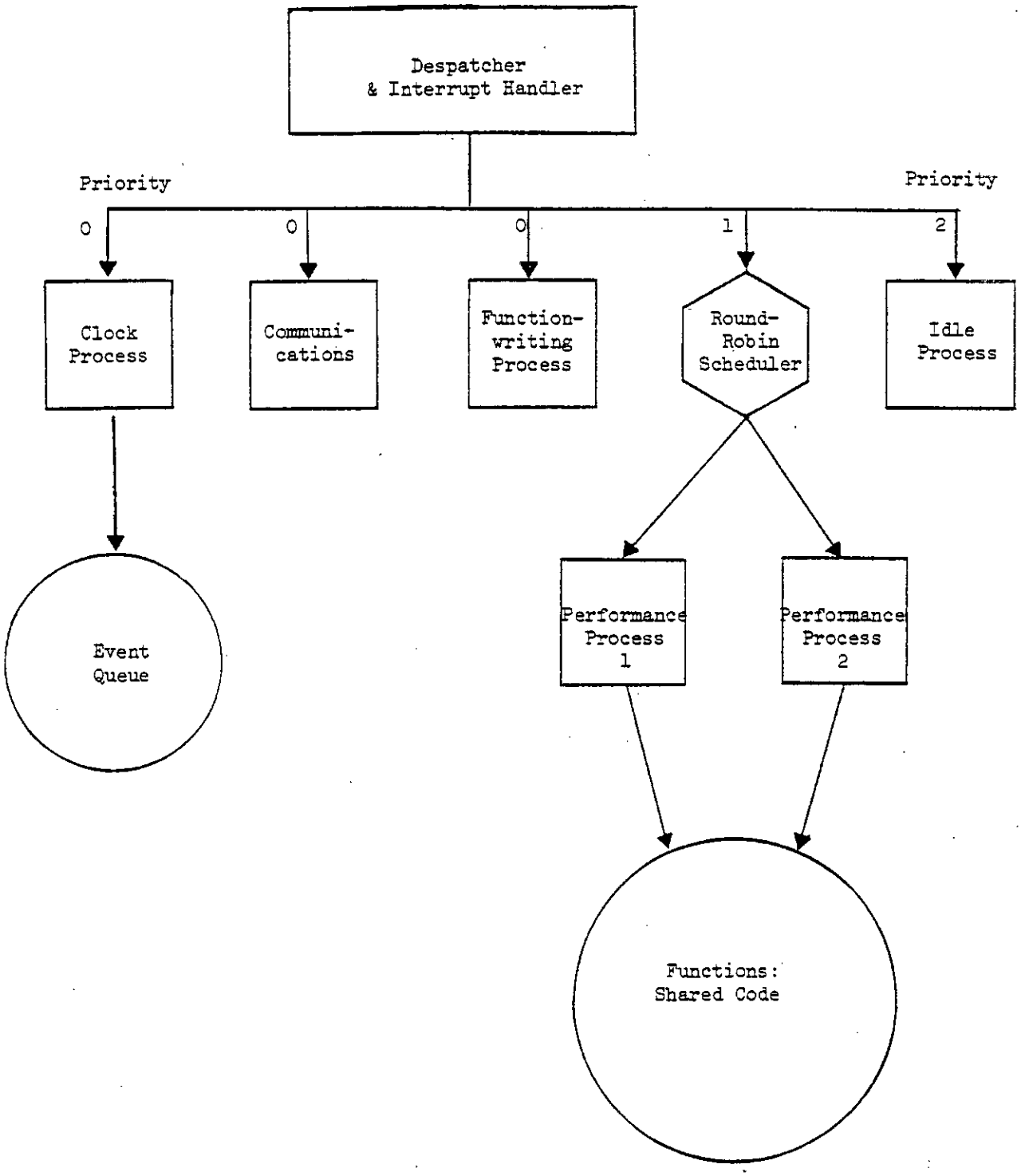


Figure 6-1: Operating System for PDP-15/40
Non-standard Synthesis Instrument

Performance processes (Figure 6-2) control the execution of Functions and are indirectly controlled by the clocking process. The clocking process monitors the 'real-time' elapsed with reference to an 'event-queue' which specifies which Functions are to be performed in which voices, ie. by which performance process, for a given duration. The clocking process checks the time of the next event in the event-queue every 50th of a second (ie. there is a clock interrupt every 50th of a second) and, if necessary, will inform the performance processes of required changes in the execution of the Functions. At the end of the event-queue, the clock process releases, ie. terminates, the performance processes and itself.

Clocking and communications processes are activated by interrupts, suspending the execution of whatever process was being executed at the time. Interrupts are handled serially; that is, interrupts are disabled whilst an interrupt is being handled.

The performance processes carry out a number of steps illustrated in Figure 6-2. When executing the two performance processes concurrently, the dispatcher switches between them using a 'round-robin' or 'flip-flop' scheduler. The performance processes are switched between whenever a sample is generated. A performance process will execute a number of assignment statements and one conversion statement (generating a sample). The time required is dependent on the execution time of the instructions in the statements. When a sample is generated, the performance process is 'put to sleep' and the other performance process is

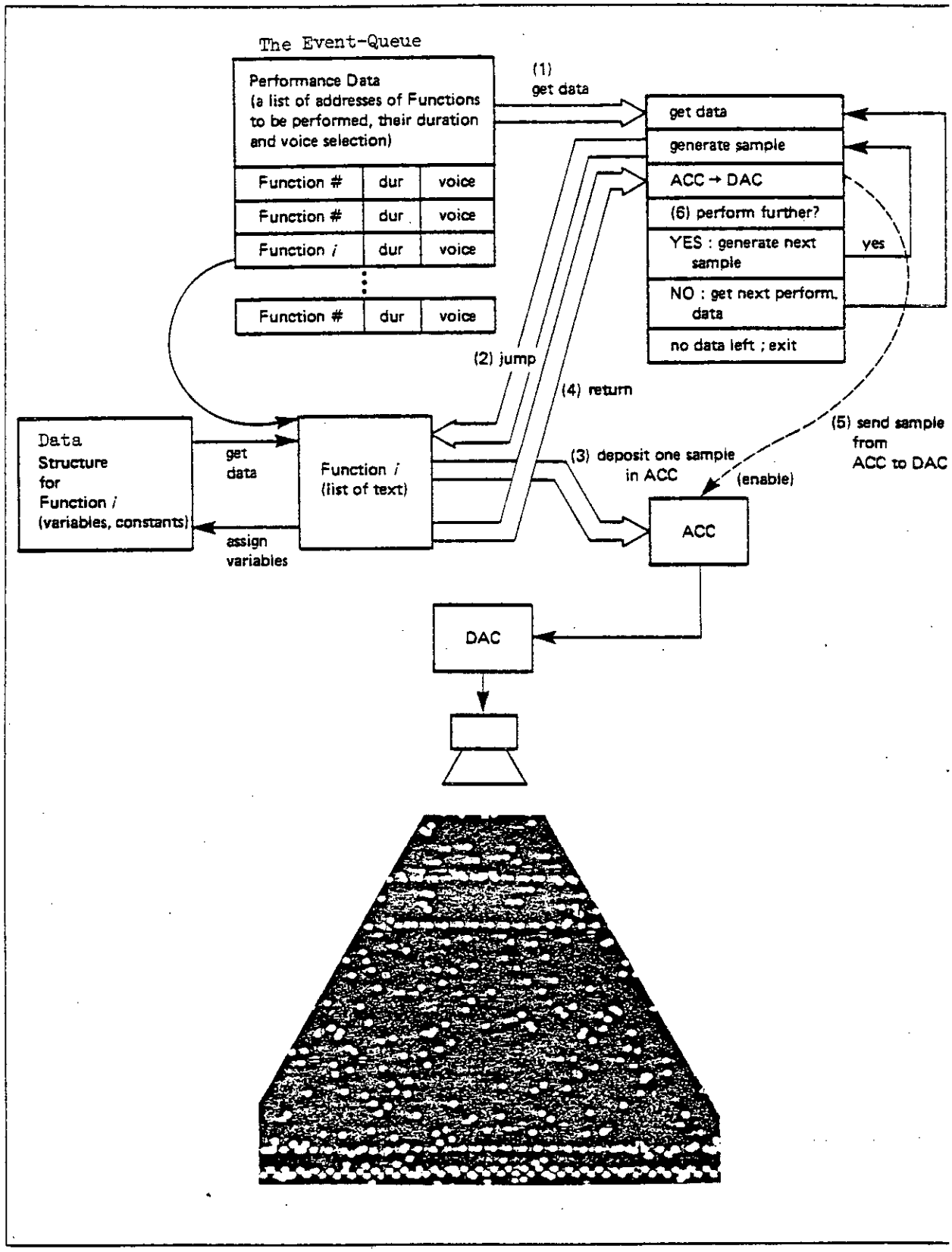


Figure 6-2: The execution of Functions by Performance Process

Figure 6-2: The execution of Functions by Performance process.

1) An event-queue defines a performance by the instrument. On significant clock ticks, the clocking process passes data for the execution of a Function to the relevant performance process.

2) The performance process passes control to a specified Function. The Function is executed until it generates a sample.

3) The sample is left in the accumulator.

4) The Function returns control to the performance process.

5) The performance process sends the sample to a D-A convertor. Each performance process is associated with a convertor.

6) The performance process determines whether the same Functions is to be further executed. It is at this point, in this implementation, that the performance process puts itself to sleep and the despatcher activates the next process on the queue. (See Figure 6-1)

activated. When reactivated, the execution of the Function continues with the statement following the previous executed conversion statement unless the process has been informed by the clocking process that execution of another Function should begin.

Clocking and communications processes are activated by interrupts, suspending the execution of whatever process was being executed at the time. Interrupts are handled serially; that is, interrupts are disabled whilst an interrupt is being handled.

The Context of Functions

In a composition, the sounds described by Functions would be integrated into a hierarchical system of syntactic relationships. Given some number of sound-producing Functions, these may be used as the 'content' of an arbitrary structure. This structure is defined by performance data, an event-queue to be used by performance processes, which determine the relationships between Functions (ie. sounds), their durations and so on.

The non-standard synthesiser may be directly interfaced with the automatic composition software described in Chapters 3 and 7. The composition process using the 'automated non-standard synthesiser' may then be seen as part of a hierarchical semantic-syntactic structuring process (see Figure 6-3). Specifically, by the definition of a grammar giving semantic and syntactic rules for ordering virtual machine instructions, a collection of Functions for producing sounds are generated. At

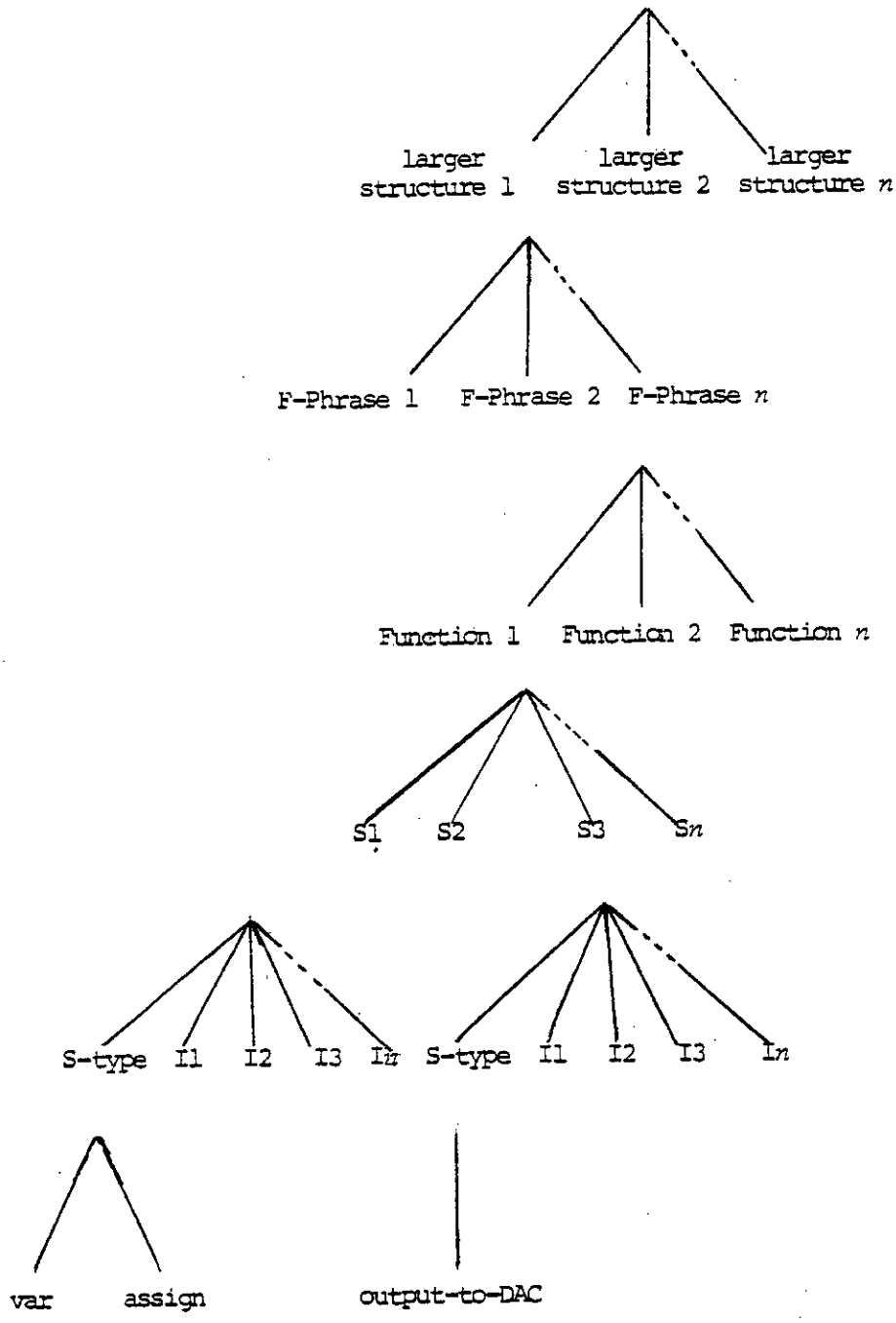


Figure 6-3: The description of a sound structure performed by the non-standard synthesis instrument is hierarchical. The sounds themselves are described by Functions which consists of statements (S) which in turn consists of strings of instructions (I). Statements either assign values to variable locations or output them to D-A converters. The sounds are structured to form larger musical structures.

the lowest level, individual instructions are ordered to form expressions, at a level higher, expressions are used in statements. Statements are ordered to define Functions. Given some number of sound-producing Functions, they may be used as the 'content' of an arbitrary abstract structure.

Using, for example, grammars, Functions are ordered to form phrases of Functions; the phrases, each generated by a distinctive grammar to create a perceptually distinct pattern, are in turn ordered to form large-scale syntactic structures. Using a grammar defined in GGDL for composing a structure consisting of nine distinctive sounds, the tokens (morphemes) representing these sounds could be mapped to non-standard synthesis generated Functions (see Figure 4).

At all levels of the system, the generation of structures may be determined by a definition of a "language". In the case of the 'automated non-standard synthesis instrument', the user controls the process of sound (synthesis code) generation by initialising the control variables of the grammar. At the level of instruction synthesis code generation, variables determine what virtual machine instructions may be used as 'terminal tokens' and how they may be ordered. For a fuller description of the control of the synthesis code generation, see Appendix 3.

Summary

Chapters 4 and 5 demonstrate how GGDL may be used to generate

both macro- and micro-sound structures. The 'automated non-standard sound synthesis instrument' is a synthesis technique working at the micro-level and is a direct consequence of using grammars to describe micro-sound structures. It demonstrates that the use of generative grammars and an automated CAC system may, in addition to facilitating composers in the composition process, open new possibilities in composition that could not have been arrived at without such a system.

Chapter 7: The System Configuration - An Implementation

It was proposed that generative grammars could usefully aid composers by providing a means of automating part of the composition process. A computer aided composition system based on generative grammars was designed and implemented to investigate if, in fact, generative grammars could usefully aid composers. This chapter describes the system implemented at the Department of Computer Science, University of Edinburgh.

The core of the system is two facilities. These are 1) a facility for formally and explicitly defining the grammar of a music language, ie. the GGDL programming language 2) a facility for using GGDL language definitions to automatically generate utterances in the specified language, ie. the GGDL-Generator. Using the GGDL language and GGDL-Generator one may generate compositions.

However, even if the process of composition is automated, if using the computer and evaluating and transcribing compositions automatically generated remains a tedious process, composers will not find such a facility a useful aid. On its own, the GGDL composition generation software would be awkward to use. When composing, most composers generate working material, fragments of a musical structure, and then evaluate and rework the material, using it to construct larger music structures. The composition process involves feedback and iteration. A computer system to aid a composer should therefore provide an interactive environment

which not only facilitates the generation of music structures but facilitates the evaluation of generated structures as well.

A suite of programs was implemented to support the GGDL compositon software. The suite, referred to as GGDL-CAC system, was designed and implemented to permit the generation of music structures using the GGDL composition software and the evaluation of those structures through performance or, in some cases, visual inspection. The suite of programs was designed to permit the editing of structures as well so that the structures generated with grammars could, if desired, be altered independent of the grammar generation and mapping processes.

The GGDL-CAC system was implemented on a network of computers. Facilities at the Department of Computer Science were such that certain tasks could be best performed by different processors, due to hardware and software demands of the different tasks. In this chapter, the suite of programs that have been implemented to form the GGDL-CAC system are described. These include the GGDL compiler, and GGDL-Generator, the synthesis facilities and graphics editor. The configuration of the implementation is also described.

The GGDL Programs

The core of the GGDL-CAC system is the facility to automatically generate music structures in a language defined by a generative grammar. A compiler was implemented, the GGDL

compiler, for the compilation of both GGDL generative grammar definitions and morphological definitions (cf. Chapter 3). The compilation of a GGDL program generates an object code for execution on a virtual machine, the GGDL-Generator. A GGDL language definition is sufficiently explicit and formal that the instructions of the program may be executed to generate statements in the defined music language.

Like a compiler-compiler (Feldman 1966, 1968), the GGDL-Generator accepts a definition of a language and then translates statements made in that language to another representation. Just as a compiler-compiler may be used to translate a programming language to a machine executable format, ie. an object or machine code, the GGDL-Generator uses a language definition to translate statements consisting of terminals and non-terminals to statements consisting only of terminals. The string of terminals generated may then be mapped to a specified format. However, compilation is (generally implemented as) a deterministic process which, given a program to be compiled or translated, will always generate the same translation of a program. With the GGDL-Generator, the rewriting of non-terminals by a set of rewrite rules which define the syntax of the object language is controlled by user-specified control-functions and procedures which need not be deterministic. The control procedures are defined in a high-level programming language which the GGDL compiler translates to the machine code of a virtual machine. The execution of the code is simulated by the GGDL-Generator.

Thus, a GGDL program defines a language. The 'language generator' accepts the language definition and generates utterances in the defined language. To generate structures using the GGDL language, one must go through several stages.

1) One must define a language and a set of mapping rules in GGDL. Files will be prepared with a standard editor. The language and mapping definition files will be compiled. The compiler may then be run with a GGDL definition file as input and will generate, if there are no compilation errors, an object code that may be executed on a virtual machine, the GGDL-Generator.

2) An initialising string must be prepared. This again will be done with a standard editor. The file prepared will consist of a string of non-terminals and possibly terminals (in the grammar defined in the GGDL grammar program).

3) The GGDL-Generator program may be run. This program requires three inputs: 1) the string initialising generation, 2) the compiled GGDL language definition, 3) the compiled GGDL morphological mapping definition. (See Figure 7-1)

The GGDL-Generator passes through three distinct phases of generation. Firstly, the string for initialising generation is rewritten. If there are no non-terminals the string will not be altered. Secondly, the rewritten string undergoes

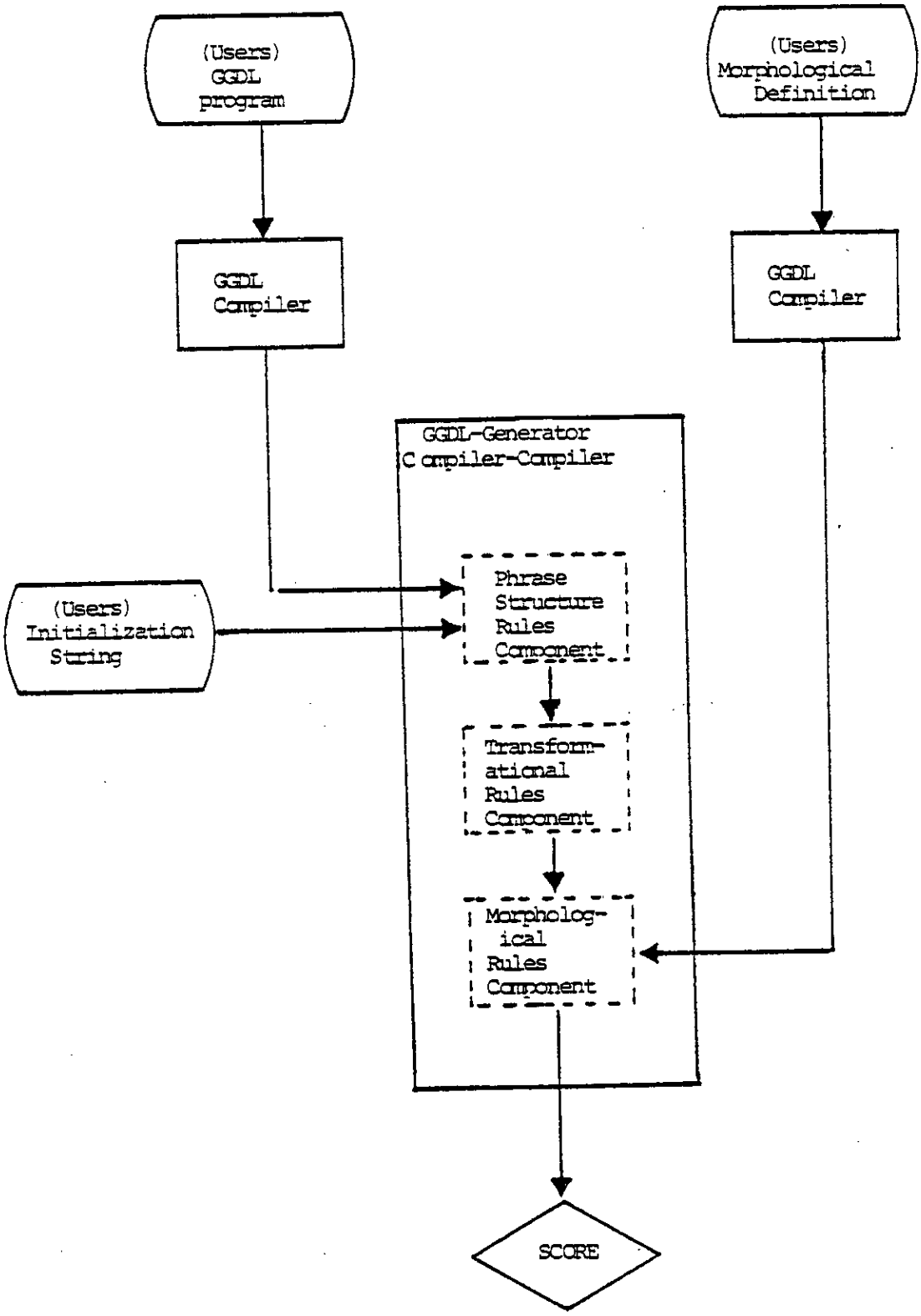


Figure 7-1: The GGDL Programs

transformational processing where structural change markers will initiate transformations. Lastly, the string, transformed, is mapped into its final format according to the mapping rules. The output of the GGDL-Generator program is the generated structure (in the defined language) in the data-format described by the mapping program; output may be formatted for Music V (Mathews 1969) note cards, data for frequency modulation oscillators, a score editor, the non-standard synthesiser, transcription, etc.

Alternatively, the GGDL-Generator may be run with an empty input stream for either the 2nd (GGDL grammar) or 3rd (GGDL mapping definition) input stream. In the case where no mapping program is given, the representation of the structure by abstract tokens generated with the phrase-structure rules is output to the first output stream. The string is output before transformational processing as, if the '@T' or '@I' transforms are indexed, the mapping program is required for the definition of their relative ordering. Where no GGDL grammar program is given, it is assumed that the first input stream will consist of a string of terminals and structural change indexes only. The GGDL-generator, in this case, performs transformational processing and maps the string to the specified format, generating the mapped string as output. By these means, one may generate an abstract structure using a given grammar and then map the same structure by different morphological rules, for example, for different synthesis instruments.

The GGDL-Generator allows, in addition to the first output stream reserved for the program output, a second or possibly third

output stream to be indicated as diagnostics files for monitoring the generative process. These consist of the string at various stages of rewriting and transformation and other diagnostic information that may be useful in debugging GGDL programs. Examples of how the programs are invoked on the VAX implementation are given in Appendix 4.

Inspecting Compositions

Using the GGDL language and GGDL-Generator one may generate compositions. These may be written for traditional instruments or, as often will be the case with composers inclined to use computers, for some means of electronic sound synthesis. The user environment for a composition system should permit the easy examination of structures generated with automatic composing software. To this end a configuration of computers was designed and implemented to permit not only the generation of compositions, but sound synthesis for aural feedback and, in some cases, the possibility of visual inspection.

The synthesis facilities available are limited but may be useful even in the case where the composition system is used to compose instrumental music as it provides at least some aural feedback, if only very sketchy. This is especially needed where, for instrumental work, the output of the computer is not in traditional graphic form, ie. a score, that can be immediately interpreted but rather is generated in some alpha-numeric representation that requires laborious transcription. By changing

only the mapping process, but not the composition's structure, it is possible to receive from the system some quick aural feedback about the results, and, when satisfied, to then map the structure to some alternative format.

The hardware available for sound synthesis at Edinburgh was limited. The only available digital-to-analog convertors were on a PDP 15/40 (DEC 1969), which had no high-speed clocks to facilitate timing of sample conversion. This made it impossible to interface the GGDL composition software with a synthesis system allowing the synthesis of very complex sounds.[1] However, as, at present, most complex sound synthesis may require several hours of calculation for the performance of even a few minutes of music, it is done outside of real-time. The dedication of a processor as powerful as the PDP 15/40 does have the advantage that it may be used for real-time synthesis with immediate response. The PDP 15/40 was configured in a system in which one could, whilst composing, conveniently monitor compositions generated. A structure could later be generated in a format suitable for transcription to a score for traditional instruments, or for a considerably slower but more complex form of synthesis permitting the synthesis of more acceptable sound output, such as MUSIC V.

1) Though MUSIC 4BF (Howe 1975) has been implemented on the Department of Computer Science's VAX 11/780 computer, the samples generated by the program cannot be properly converted on the PDP 15/40. Because the Deparment of Computer Science's PDP-15/40 has only 32k words of core memory, which at sampling rates of 20,000 hertz, means only about two seconds of sample data may be stored in core memory, some form of bulk storage is required. However, reading stored samples there is no method of accurately timing samples as, on the PDP-15/40, there is no fast clock to time sample conversion and the sample-times cannot be accurately timed using a fixed loop of instructions as bulk storage devices such as magnetic tape drives or disks steal memory cycles to read data into main memory.

It is possible with the synthesis software implemented on the PDP-15/40 to synthesise polyphonic structures where one may specify the frequency, duration and waveform of each 'sound event' in an 'event queue'. It is also possible, though only with monophonic synthesis due to the overhead required for calculations in real-time, to specify an envelope for each sound event. In addition, one can perform 'non-standard' synthesis on the PDP-15, which is discussed more fully in Chapter 6. The event queue may specify changes between the methods of synthesis.

A number of graphics programs have been written to facilitate the definition of waveforms and envelopes. These use Tektronics terminals equipped with cursors and storage scope displays. Using a graphics editor implemented as part of the system, it is possible to define a waveform or envelope interactively as a set of points on a Cartesian plane using a cursor or text commands. Up to four waveforms and/or envelopes may be displayed and edited at one time. Other graphics programs are used to display waveforms that have been defined by other means and are not represented in a format compatible with the graphics editor.

The suite of programs which comprise a set of complementary aids for composers have been implemented in a fashion which would permit the easy integration of additional programs. Graphics programs and data for synthesis represent waveforms, envelopes and performance data in a shared format so that the programs are compatible. All files are kept as alpha-numeric representations

that may be edited with a standard text editor or easily integrated with other software that may be written.

The System Configuration

The Edinburgh GGDL composition/synthesis system presently consists of two computer processors interconnected to permit the parallel performance of tasks. These include the generation of data required for performance, the performance, ie. synthesis, of sounds, and the control of 'message' transmissions. The system is designed with one of the processors, the VAX 11/780 (DEC 1977), acting as a main control processor and the other, the PDP 15/40, as a subordinate synthesis instrument. The system was designed as a 'star' configured network (Figure 7-2) to permit the parallel performance of generated music structures by a set of subordinate instruments, ie. synthesisers. At present, only one branch of the star has been implemented.

For the CAC system described in this chapter, a special operating system was designed and implemented for the PDP-15/40.[1] The operating system was designed to permit the concurrent execution of several processes including processes for sound synthesis, which, due to real-time constraints have special

1) A detailed description of part of the operating system of the PDP-15/40 is given in Chapter 6 with reference to the 'automated non-standard digital synthesiser'. The design of the complete operating system is essentially the same (see Figure 6-1). There are several more performance processes which may be activated for different types of synthesis, eg. direct-wave synthesis, or synthesis with envelope shaping, and the clocking process is somewhat extended to handle the timing and inter-process communication required by the additional performance processes.

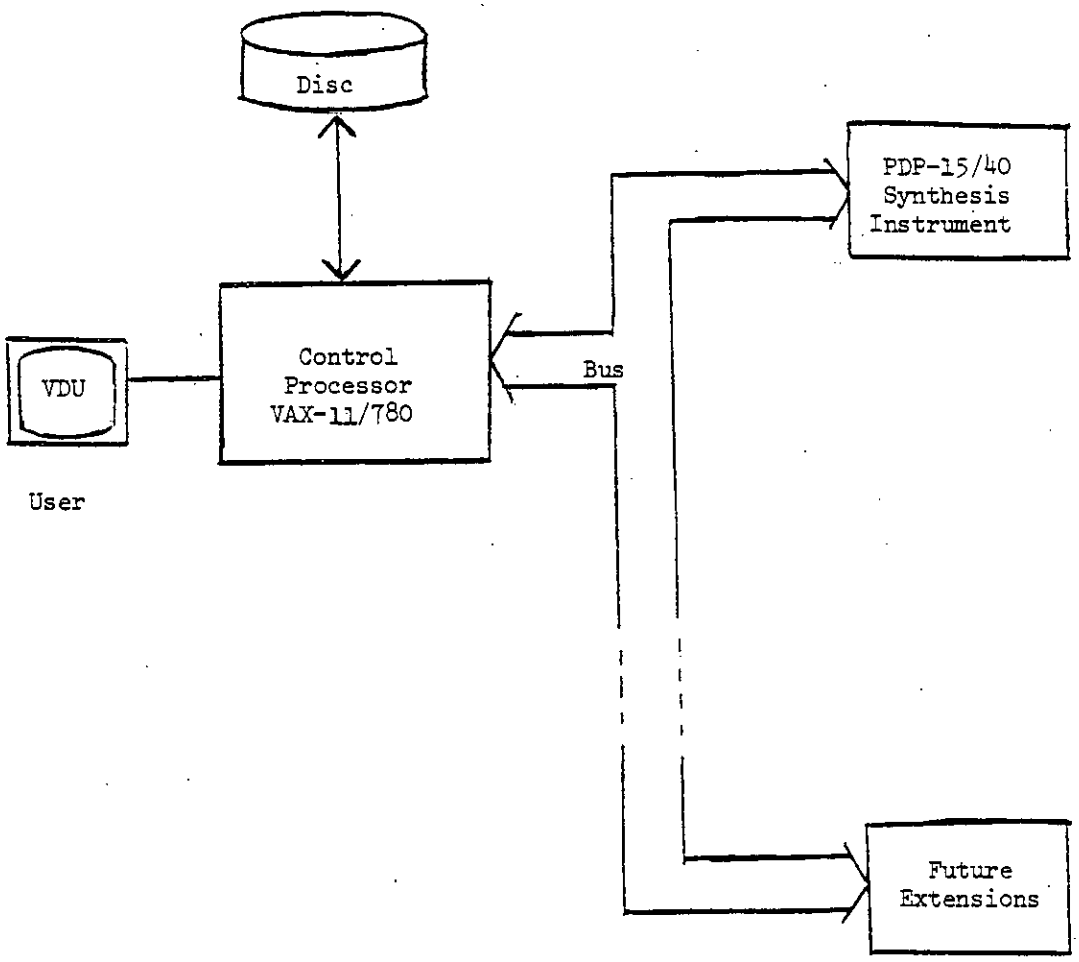


Figure 7-2: The System Configuration

demands, and processes for handling communications with the VAX 11/780. Routines for handling communications protocol were written for the VAX 11/780 using the system I-O, though, given more subordinate instruments, communications and the accessibility of the system would be improved by using a dedicated control processor. The control processor's operating system could be designed to permit the concurrent execution of 'conversations' with several instruments, each conversation associated with a process, as well as the execution of user processes. The communications between the VAX 11/780 and the PDP-15/40 take place over a direct connection using the Edinburgh Department of Computer Science communications links (Tansley 1977). With several subordinate instruments, message switching could possibly use an Ethernet (Metcalfe 1976).

With the present VAX-PDP 15/40 configuration, processes are only run serially. Communications protocol are such that the PDP 15/40 will only accept a message immediately if the message is either an 'abort', which can be determined from the 'request to send message' header transmitted by VAX, or the PDP 15/40 is not occupied with another activity such as performing synthesis. This ensures that synthesis is not audibly disrupted by communications. However, though the operating system designed for the PDP 15/40 allows the control processor (VAX) to interrupt it at anytime, if the PDP 15/40 cannot accept the message body, with VAX it is necessary to wait for the completion of the message transmission before another process may be started as messages are not queued

and another process cannot be run concurrently. This may cause delays undesirable in an interactive environment; such delays could be avoided with a dedicated control processor.

Using the system, a user sends commands interactively to the control processor from a video terminal (see Appendix 4). Synthesis is performed via the control processor (see Figure 7-2). The user may request the control processor to do several things:

1) run the GGDL compiler
2) run the GGDL-Generator
3) run the Graphics Waveform/Envelope editing programs
4) run any user-defined program (high-level languages, etc.)
5) run any of the system utility programs, (eg. the editor) compilers, etc.
6) compile and/or transmit a message to the PDP-15/40 synthesiser

1-5 of the above may be used to generate performance data for the synthesiser. This includes compositions, waveform definitions, control information for the writing of functions on the Non-Standard Synthesiser, and so on. 'Messages' are used to transmit data to and control the performance of the synthesiser.

Message Compilation and Transmission

Messages may be of five types:

1) Performance-data Messages

2) Execution Commands

3) Waveform Definitions

4) Function-writing control data for the Non-Standard Synthesiser

5) Abort Command

Each type of message contains differently formatted information, both for the user and 'on the links'.

'Performance messages' contain information defining the event queue - what pitches are to be performed in which voices, their durations, changes of waveform, the method of synthesis to be used, and so on. 'Execution commands' indicate whether to perform, to write Functions (for non-standard synthesis instrument), or to delete certain previously received messages. Waveform definitions consist of 4096 12-bit samples representing a waveform, and 'Function writing control data' messages contain the variables required to instantiate the grammar of the non-standard synthesis instrument for generating Functions. The 'Abort command' interrupts the subordinate processor and requests it to abandon whatever it is doing and to return to the idle process until further instructions are received.

A user will prepare messages as text files either with an editor or directly typing in a message using the message compilation/transmission program. The user's text must be checked for correct formatting and data types, compiled and assembled into a machine readable representation that can be transmitted (as

bytes) between two processors, and then packed with a correct identifying header and end-of-file character. This can be done using the commands available for instructing the message compilation and transmission program.

To 'perform' the synthesiser, a user must first send the data required to the synthesiser as messages. Commands may then be sent to 'perform' the event-queue to which the synthesiser may respond either by executing the event queue, ie. performing, or by requesting further data, ie. messages, to be sent. For example, if an event queue has not been received, or the event queue specifies that three waveforms are required for execution though only two have been received, the synthesiser generates an appropriate error message. The synthesiser maintains a record of its 'conversations' independent of the control processor. This insulates each system from faults, for example, system crashs that may occur on one of the systems, and enables the control processor to abandon its communications process, losing records associated with the process on VAX, without the loss of information required to maintain a smoothly running system.

Summary

The computer aided composition system described in this chapter was implemented as a suite of programs. The system provides an interactive environment in which music structures may be automatically generated and easily examined. Complementary aids may be used to synthesise, visually inspect and easily edit

structures. For the purpose of the research concern of this thesis, it was used to investigate whether generative grammars could usefully aid composers. The practical results used for the basis of this estimation are presented in Chapters 4, 5, and 6.

Chapter 8: Conclusions and Further Research

This thesis proposed to investigate whether it is possible to usefully aid composers in the process of composition by automating part of the composition process using generative grammars. Considering in what ways composers might be aided by computers, it was suggested that, as many contemporary composers use formalised composition techniques, the process of music structure generation could be automated to aid composers. The process of composition using formal techniques could, in the case of many contemporary composers, be seen as one of defining the syntactic rules of a music language and then generating music structures by applying these syntactic structuring rules. A useful aid to composers, it was proposed, would be to automate the laborious process of applying these syntactic structuring rules to generate compositions. The process of composition then becomes one of defining the rules of a composition language which a computer automatically applies to generate compositions.

It was fairly obvious that grammars could be used for a general rule based system for language generation. But it was not obvious that composers would be able to use grammars as an aid in the composition process. Therefore, a system was implemented, based on generative grammars, to investigate whether, in fact, grammars could be used to usefully aid composers in the composition process.

Other systems had been designed to automate the compositional

process in specific music languages, though not using grammars, and it was clear that, at least to composers composing in these languages, the automation of structure generation was a useful aid. Both Koenig and Xenakis successfully used such systems to automate their composition processes. However, with these systems a composer is limited to a predefined set of composition rules; a composer may not specify his own composition rules to automate his individual composition process. Nor could these systems be used by composers interested in composing with the predefined rules but with a different compositional object. That is, the programs were designed for generating specific types of output, for example, for instrumental composition (Xenakis 1971, Koenig 1970, 1970b) where only a specified set of parameters could be generated, eg. pitch, duration, and dynamic, or for structures of sound samples (Berg 1978, Banks 1979); but an arbitrary compositional object could not be defined.

The system described in this thesis is a flexible CAC system. The GGDL-CAC system set out to provide a facility with general application. The system provides a facility, using generative grammars, that permits the specification of a large number of different types of compositional rules as well as a facility which, using grammatical descriptions of music languages, may be used to automatically generate compositions in a defined language. The implementation of the grammar system was designed in such a way as to permit, in addition to the specification of different compositional rules by different composers, the possibility of the specification of different compositional objects for a given set

of compositional rules.

However, grammars on their own would be difficult to use. It would be difficult to interest a composer to use a facility 'to aid him in the composition process' if its use presented obstacles which obstructed a natural method of composition. Traditional, ie. non-automated, composition tends to rely on a method where music structures are generated, inspected and evaluated in a feedback process. It would not be possible to evaluate if grammars could usefully aid composers unless the facilities with which a composer could use grammars permitted the easy inspection and evaluation of generated compositions. The GGDL-CAC system, therefore, was designed and implemented to provide facilities by which composition languages could be defined and compositions then automatically generated in the language, and facilities for inspecting structures generated.

Results obtained using the implemented system demonstrate that grammars can usefully aid composers in the process of music composition. A grammar was defined to generate, in a completely deterministic manner, Steve Reich's "Clapping Music". The thourough investigation of the possibilities of a composition 'process' such as that used by Reich for "Clapping Music" would be extremely tedious without an automated structure generation system. Clearly, such a composition can be composed without such a system - Reich did so. However, in the case of the Reich composition, it is suggested that with such a system, the complex interactions between the different parts of the deterministic

process could be explored with a number of different rhythmic patterns. The system could reasonably quickly generate different structures using Reich's process and the structures could be synthesised, listened to, and comparative evaluations made. In such cases, a system such as the one described in this thesis could usefully aid a composer.

David Hamilton's composition of "Four Canons" using the GGDL CAC system further supports this. In this case, the system has been used by a composer in a completely deterministic fashion to investigate the possibilities of a composition process. Hamilton generated several 'abstract' structures and, with each structure, defined several different intervallic frequency proportion relationships for mapping the structures. By this method, Hamilton examined, listened to and evaluated the different intervallic systems. Using the material generated, he selected from among the possibilities those structures that were suitable for his compositional aims. However, whereas Reich could possibly have used pen and paper to generate a number of structures using the same process but different patterns, and then examined the differences, in the case of the unusual intervallic relationships that Hamilton was interested in experimenting with, the relationships could not be easily imagined and certainly could not have been performed without the use of a computer.

The example of a grammar that would include the Schoenberg Trio from the Piano Suite, Op. 25, in the language that it generated demonstrates that the system may be used to generate sophisticated

music structures and also suggests ways that the system could be used for musicological research. It is possible with the system to test a formal description of music compositions for their adequacy. A description of 'the language' of a composition can be formalised and, using the description, the machine can generate compositions. If these compositions adequately resemble the composition modelled, then, in a sense, there is some validity to the formal description. The composition generated using the grammar from the Schoenberg example, ie. Figure 4-3, poses a number of interesting musical questions. Comparing the compositions in Figures 4-2, ie. the model, and 4-3 one can ask what the differences between them are, which of them is the more interesting and why? Certainly, if not a useful aid to composers, such a system may, at least, be used to suggest a number of stimulating musical questions.

In these three compositions, Reich's "Clapping Music", Hamilton's "Four Canons" and Schoenberg's Trio, the 'object' of the composition is different in each, ie. clapped notes, computer synthesised notes, and notes to be performed on a piano. The grammar system was designed to perform the process of structure generation in three distinct stages: a rewriting process, a transformation process and a mapping process. Dividing the process into these distinct stages permitted the definition not only of different composition rules, but also different compositional objects. These three compositions demonstrate both that grammars may be used to aid composers and that this type of grammar provides a flexibility which is essential in a system that

is to be of general application. Composers not only compose structures in different ways, but they also compose for different types of performances and may conceive of the compositional object in different ways. Two composers, for example, writing for piano may consider very different aspects of the performance to be significant. One may only be concerned with pitch-duration structures, as in the examples generated using the grammar for the Schoenberg Trio, another only with timbral differences, eg. 'klangfarben compositionen'. The GGDL system is sufficiently flexible to cope with many of the demands different composers may make.

The system, in fact, is sufficiently flexible that a composer may work not only with the composition of 'macro-structures', but also with 'micro-structures'. The author used a grammar system, a prototype of the GGDL system described in this thesis, to compose the sounds of 'After Artaud'. In the case of realising 'After Artaud', grammars were a useful aid. The composition was conceived in such a way that the generation of the sounds and the specific selection of a given sound for a given object in the composition was to be left to a controlled but not completely determined process. In addition, several occurrences of the same sound in the score were to be realised distinctly; this idea was directly derived from experience with grammars and phonetics and the conceptual distinction made between a morpheme and an 'allo-morph', ie. a realisation of the morpheme, a conceptual distinction paralleled in the distinction made in the grammar between structure generation and structure mapping.

Koenig was interested in considering how grammars defined in GGDL could be used to generate structures that could not easily be generated with other composition programs he was familiar with, viz. the SSP sound synthesis program. Symmetrical waveforms are an example of how grammars may be used to express structures not easily described in other generative systems.

The 'automated non-standard sound synthesis instrument' was also a direct consequence of using grammars. It demonstrates that the use of generative grammars and an automated CAC system may, in addition to facilitating composers in the composition process, open new possibilities in composition that could not have been arrived at without such a system.

Thus, the conclusion drawn from this investigation is that, in fact, grammars and the automation of the composition process can be of use to composers. Moreover, the use of a modularised grammatical description was demonstrated to have a number of advantages. Using the CAC system implemented, grammars were shown to be a useful way for describing different types of composition processes as well as for generating compositions by similar processes but using different compositional objects. Not only may grammars be used to facilitate the composition of macro-musical structures, but also micro-sound structures, where often the size of the task of a sound description, for example, where 20,000 digital samples are required for each second of sound, necessitates some form of automation. It was also shown that grammars may open new possibilities in composition.

A conclusion of the thesis is that grammars may be useful for the generation of music structures. In light of this conclusion, perhaps the most significant contribution of the research reported in this thesis is the development of the GGDL language and the GGDL-Generator. If grammars are to be used as an aid, facilities exploiting them will need to be developed. Though, for example, 'After Artaud' exploited grammars using a system especially developed for its composition, the overhead of developing large software systems for each application of grammatical generation is clearly inefficient and would limit the attractions of using grammars, ie. the facility with which they can be applied to certain tasks. Few composers would be prepared to develop such software. The versatility and general applicability of the GGDL system are desirable features in a CAC system automating the composition process.

The GGDL system may be used for investigating in what ways and to what extent grammars may be used by composers to generate music structures. The grammar system could, perhaps, also be used for a disciplined musicological investigation of the nature of the formal structure of music; or to facilitate systematic experimentation and research into 'form-potential', such as Koenig's. GGDL may prove a useful tool with various applications.

Implementing a grammar system was useful for determining some of the benefits of exploiting grammar systems. It also helped to clarify some of the limitations of such systems. Though one may theoretically describe any lanugage in terms of rewrite rules and

grammars, practically, it will be extremely difficult to describe certain constructs using the facilities provided by a given facility.

For example, experience with the GGDL implementation used for the work reported in this thesis made apparent a number of inadequacies of the language. A new version of the GGDL language has been specified in which a number of alterations and extensions have been made. For example, the available data-types have been extended and formatting constraints have been made less rigid.

There are, however, more fundamental limitations which make certain music constructs difficult to describe. The basis of the GGDL language is a linguistic mechanism, the rewrite rule. Natural language is monolinear, but in music, complex linear, ie. melodic, and vertical, ie. harmonic, relationships may occur. Such ortogonal relationships may be difficult to represent using a grammatical system such as the GGDL system.

The partition of the generation process in GGDL facilitates the definition of the compositional object independent of the generation process. In some respects, however, it may prove a limitation. The division makes it is difficult to consider how objects are related to one another 'in time' whilst generating the abstract representation of a music structure. For example, morphemes which represent duration-values, manipulated during the process of structure generation, have no 'real' value. The relationships that are generated in time are therefore difficult

to evaluate. This again makes it difficult to generate contrapuntal textures where both melodic and harmonic relationships need to be considered whilst generating a structure.

In GGDL one can straight-forwardly generate a linear structure and superimpose a counterpoint, as in the examples in Chapter 4. However, in the examples in Chapter 4 resulting vertical or harmonic relationships were not considered explicitly by the generation process. In "Clapping Music" the vertical relationships which resulted from the 'phase' relationship between the clapped pattern in the two voices was not explicitly considered, but rather was an implicit result of the process. Similarly, in Hamilton's "Four Canons", Hamilton was aware of what vertical relationships would result from generating two structures of intervallic relationships in a specified counterpoint but these were not considered and did not influence the generative process. On the otherhand, the grammar derived from the Schoenberg Trio could be criticised for its failing to account for the harmonic relationships that would be created with certain series - a series which resulted in octaves and fifths certainly would not have been acceptable to Schoenberg.

There are ways in GGDL to generate structures whilst considering both melodic and harmonic relations. For example, one could generate the two or more voices simultaneously and make the selection of each voice dependent on the other. However, to generate one voice independently and then harmonise it with a second part after generating the first part, after all, a quite

natural thing for a musician to do, might prove awkward in GGDL. Sufficient experimentation with generating such structures using GGDL has not yet been undertaken to evaluate how the GGDL language might be used to describe such a generation process. In any case, it is not clear what sort of mechanism might be used to automatically generate such structures (with or without GGDL). When a mechanism for generating such structures is 'discovered', GGDL could possibly be extended to provide such a facility.

Directions for Further Research

Given a system such as the GGDL CAC system with its facilities for describing and generating music structures there is scope for experimenting with generating different sorts of structures using such a system. Further extension of the definition language facilities and the development of facilities based on fundamentally musical, rather than linguistic, concepts could enhance facilities for music structure generation.

In any case, the composition of music consists of more than just generating formal abstract music structures. When writing a piece of music, a composer does much more than generate note and duration structures from a set of formal compositional rules. In the Schoenberg Trio example in Chapter 4, the set of rules generates only a note-duration structure. The Schoenberg composition, however, consists of much more than this. The octave distribution of the pitches has not been considered, nor have the dynamics and perhaps most important, the articulation of the work.

Schoenberg's rule of serialism, ie. that no note (or whatever) should occur a second time until all others have occurred once, may be easily formalised and programmed. But it is a much greater problem to formalise the following description of Schoenberg's style: "Though using serial rules to generate notes, rather than late-romantic harmony, Schoenberg's 'Suite fur Klavier' can be said, nonetheless, to be written in a Brahms-like piano idiom".

Most research in computer composition has been concerned with the generation of notes, durations, octave distribution, dynamics, etc. There has been little attempt to try to incorporate in a set of compositional rules 'knowledge' about what instruments and what sounds are used in the composition. A composer writing a piece of music for piano is likely to write a different piece than if it is for violin: phrasing, octave distribution, dynamics, and the pitch structure itself are likely to be influenced by the medium by which the composition is to be realised. The following quotation exemplifies the current concern with musically 'intelligent' programs.

C. ROADS: "One kind of artificial intelligence task is that of a program itself knowing what kind of sounds that it's actually dealing with and altering the program logic according to these sounds. Do you see this as a possibility?"
G. M. KOENIG: "Not only as a possibility, I think it is even a necessity...you need some kind of relationship between the musical language structure and the structure of the sounds produced."
(Roads 1978b)

Further research using the facilities described in this thesis could investigate and formalise the rules of orchestration and how the process of generating a music structure is related to the medium for the generated structure's performance.

Just prior to formally beginning the development of the GGDL CAC system, the author developed a program for the composition of a piece for harp.[1] It was an attempt to design a composition program which could be said to have some musical intelligence. The problem of generating a composition for an instrument is not only to formalise the rules of a music language but to integrate into the compositional rules a description of the instrument that is to perform the generated structure.

To write the composition for harp, it was necessary to constrain the generation of music structures by the limitations imposed by the harp itself. The most obvious problem facing any composer writing chromatic music for the harp is the availability of only seven notes in the octave at any one time and the associated problems of pedalling. But, in addition to the practical details, for example, a harp's pedalling limitations and fingering, or a wind player's breathing, a description of an instrument, ideally, would consider the more subtle possibilities of the different methods of performing an instrument and how they sound and affect the realisation of the composition in musical, rather than practical, terms. That is, how the idiomatic capabilities of an instrument might be exploited to musical effect.

1) It was the difficulties in describing and implementing the program that made clear the possible usefulness of a facility such as the GGDL composition software. The rules of the composition were in fact expressed as a set of nested finite-state transition matrices which could have been considerably more easily implemented using GGDL, rather than designing software to both simulate the finite-state generation mechanism and permit the easy definition of transition values.

In the composition program for writing for harp a number of different ways of performing the harp were listed. The constraints associated with each manner of performance and the ways that one type of performed note could be related to another were also included. For each listed way of performing the harp, different variables could be given for octave distribution, dynamics, and possible pedal changes.[1]

When the computer would generate a note of the composition, it would evaluate the context and determine what modes of performance would be suitable, ie. sound good. Having selected to perform, for example, a harmonic, it would determine what pitches could be performed and which octave would be suitable, which dynamic, and so on. It might consider, from a practical viewpoint, that a harmonic cannot be performed on the wound strings of a harp. From the viewpoint of 'sound quality', it might assess that a harmonic will not sustain if played in the upper octaves or that a full bodied harmonic may be obtained by performing it in the mid-range of the harp. Playing a chord in the higher octaves of the harp, on the other hand, can be used to obtain a very brittle quality. The program was designed to exploit the idiomatic qualities of the instrument to musical effect.

This is not necessarily the order in which a composer would

1) It may be possible to change the available pitches to permit a wanted but not at the time available note to be performed, for example, if one is slowly performing harmonics or 'ordinary' attacked notes. But the limitations increase if one is performing grace notes or chords: there may not be the time to pedal-change.

actually make decisions without the computer. It is, however, one way of beginning to formalise the problems of composing for harp. The harp is a very difficult instrument for which to write and in the program described, the description of performing the harp was not completely succesful. The program generated some unperformable music. Problems relating to the pedalling were the most obvious short-coming of the formalisation of harp technique. It was necessary to 'rearrange' some of the music generated. However, the computer did generate some effective and competently written music for harp (see Figure 8-1).

This points to a possible application of GGDL. Using GGDL one may explore not only the generation of music structures, but rules of orchestration and the relationship between the rules of structure generation and the performance of generated structures.

However, a possibility given such research and fruitful results, would be to further facilitate composers in the process of composition by integrating into a CAC system a data-base listing constraints on instrumental performance and perhaps information on how such constraints need to be considered in the process of composition. Perhaps one could simply add to a set of composition rules, a reference to the data-base's information on the instrument for which the composition is to be performed and the composition system could automatically ensure that a generated composition would be performable. Generated structures could be tested against a set of 'design rules' in a similar way to which integrated circuit CAD systems check a circuit design against

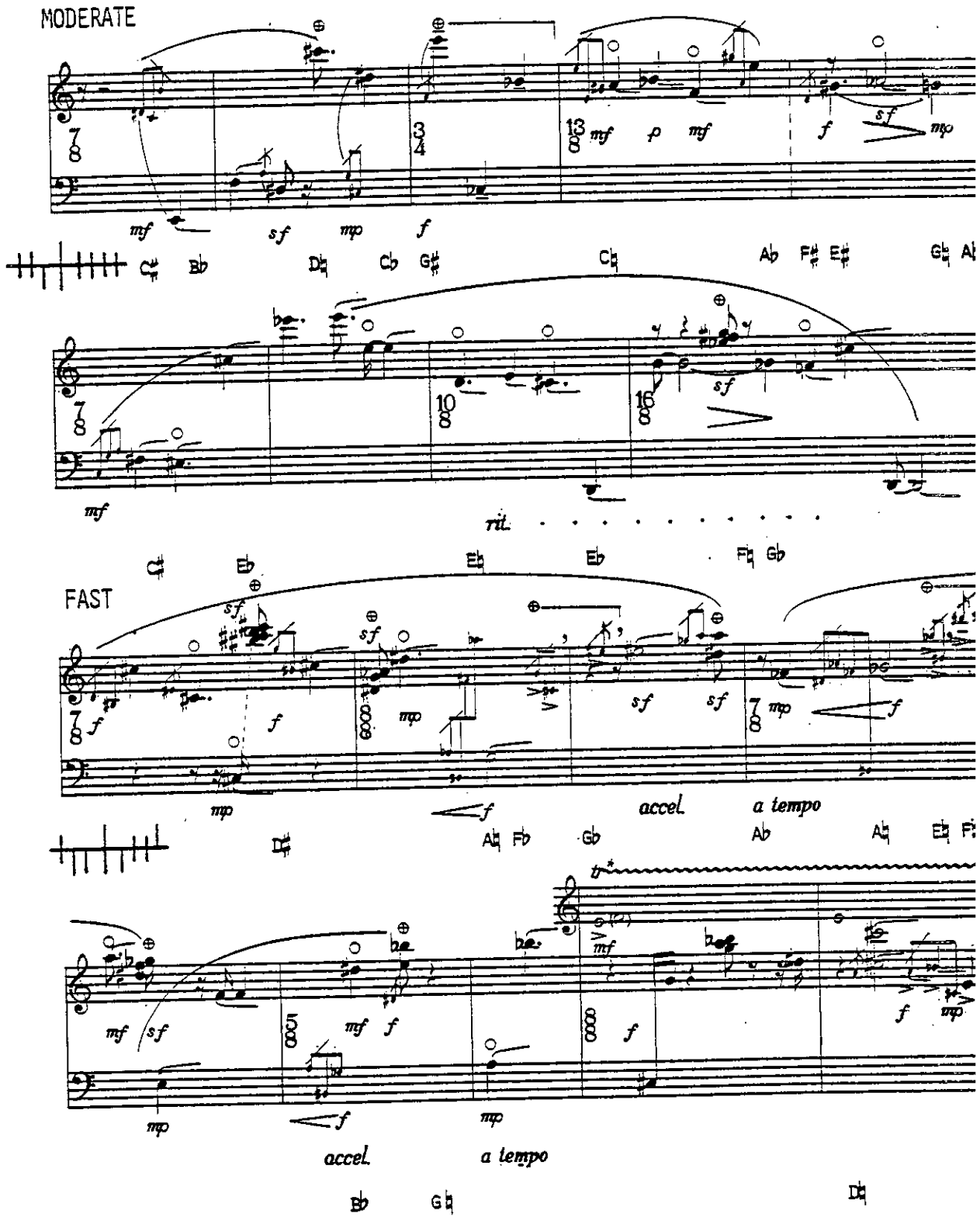
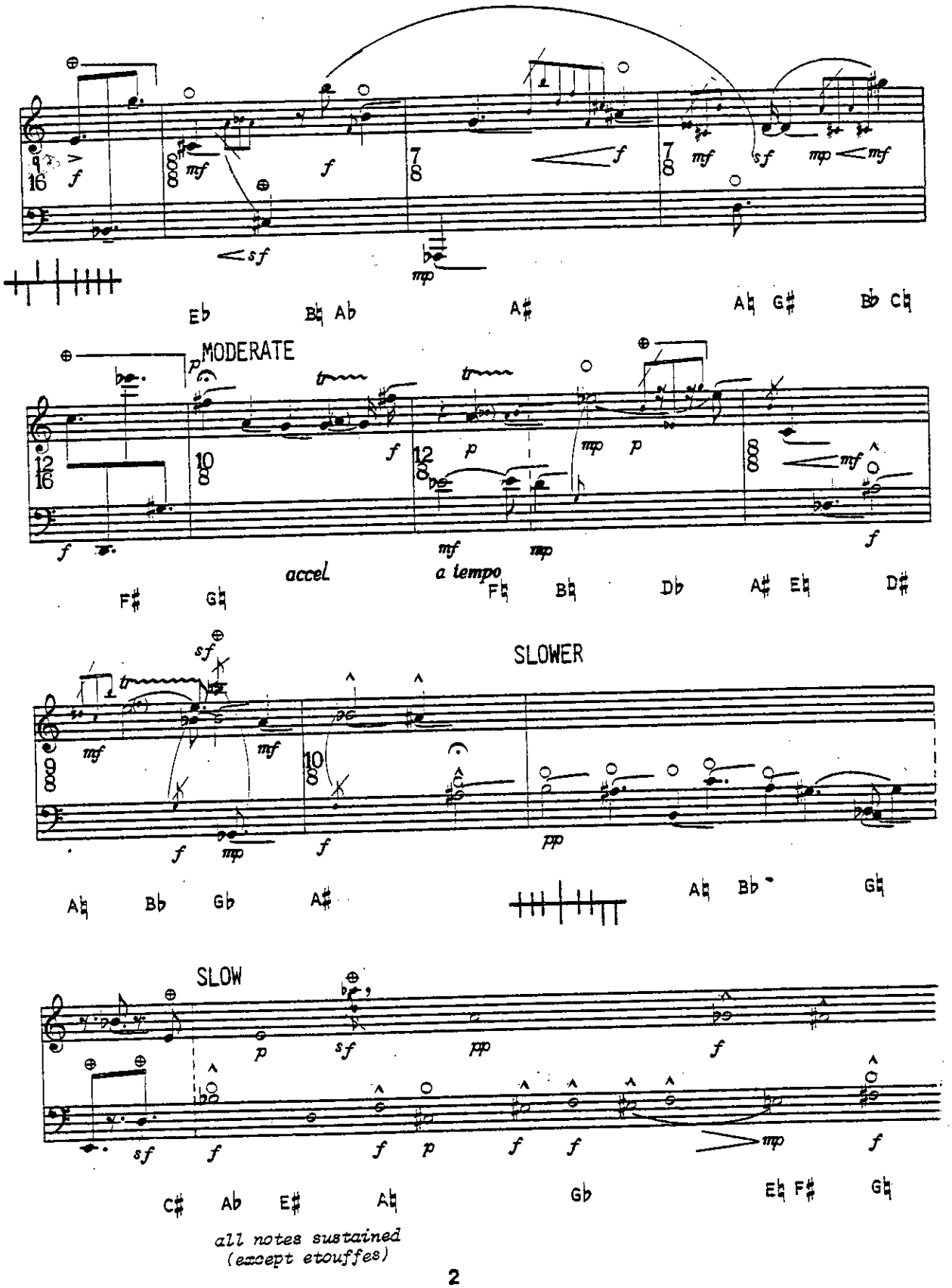
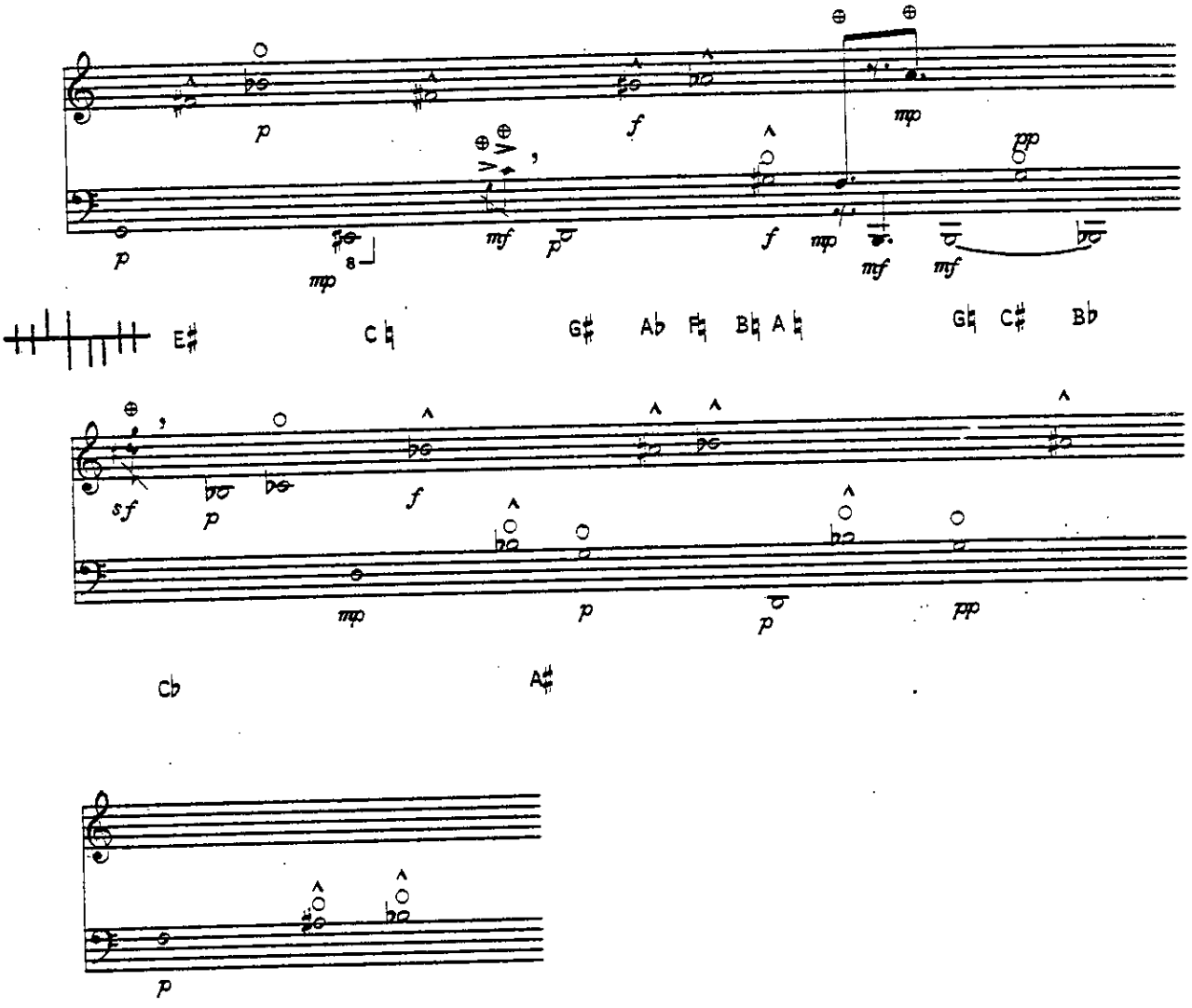


Figure 8-1: 'For Solo Harp' by S.R. Holtzman: a composition composed with the aid of a computer.


MODERATE
accel
a tempo
SLOWER
SLOW
all notes sustained
(except etouffes)



3

specifications of electrical and physical contraints.

There is, however, a considerable difference. In circuit design consistent design rules result in a uniformity that permits a more efficient fabrication process to be designed. Designs may also be checked for logical, ie. functional, consistency. In music, efficiency is not a primary criterion for 'design' or composition. Nor can the 'correctness' of the logic of a musical language be objectively evaluated. In music there is no right or wrong. Some music just sounds better!

BIBLIOGRAPHY

BANKS, J. D., P. Berg, R. Rowe, and D. Theriault
(1979) "SSP - A Bi-Parametric Approach to Sound Synthesis", Sonological Report, Institute of Sonology, Utrecht.

BEAUCHAMP, J.
(1979) "Brass Tone Synthesis by Spectrum Evolution Matching with Non-linear Functions", Computer Music Journal, Vol. 3, No. 2, pp. 35-43, M.I.T. Press, Massachusetts.

BERG, P.
(1975) "ASP - Automated Synthesis Program", unpublished manuscript.
(1978) "A User's Manual for SSP", unpublished manuscript.
(1979) "PILE - A Language for Sound Synthesis", Computer Music Journal, Vol. 3, No. 1, pp. 30-37, M.I.T. Press, Massachusetts.

BERTONI, A., G. Haus, G. Mauri, and M. Torelli
(1978) "A Mathematical Model for Analysing and Structuring Musical Texts", INTERFACE, Vol. 7, No. 1, pp. 31-44, Swets & Zeitlinger, Amsterdam.

BOULEZ, P.
(1955) "Structures, Book 1, for two pianos", Universal Edition, Vienna.

BUXTON, W.
(1977) "A Composer's Introduction to Computer Music", INTERFACE Vol. 6, No. 2, pp. 57-72, Swets & Zeitlinger, Amsterdam.

BUXTON, W. and G. Fedorkow
(1977) "The Structured Sound Synthesis Project (SSSP): An Introduction", Technical Report CSRG-92, University of Toronto, Toronto.

BUXTON, W., R. Sniderman, W. Reeves, S. Patel, and R. Baecker
(1979) "The Evolution of the SSSP Score Editing Tools", Computer Music Journal, Vol. 3, No. 4, pp. 14-25, M.I.T. Press, Massachusetts.

CHOMSKY, Noam
(1957) "Syntactic Structures", Mouton, The Hague.

CHOWNING, J.
(1973) "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation", Journal of the Audio Engineering Society, Vol. 27, No. 3, pp. 526-34.

DAHL, O., E. Dijkstra and C. Hoare
(1972) "Structured Programming", Academic Press, New York.

DEC
(1969) "PDP-15 Systems Reference Manual", Digital Equipment Corporation, Maynard, Massachusetts.
(1969b) "PDP-15 MACRO-15 Assembler Programmer's Reference Manual", Digital Equipment Corporation, Maynard, Massachusetts.
(1977) "VAX11/780 Architecture Handbook", Digital Equipment Corporation, Maynard, Massachusetts.

DIE REIHE
(1955) "Anton Webern", eds. H. Eimert and K. Stockhausen, Universal Edition, Vienna.

DI GIUGNIO, P. and H. Alles
(1977) "A One-Card 64 Channel Digital Synthesiser", Computer Music Journal, Vol. 1, No. 4, pp. 7-9, M.I.T. Press, Massachusetts.

FELDMAN, J.
(1966) "A formal semantics for computer languages and its application in a compiler-compiler" Comm. ACM 9:1, 3-9.

FELDMAN, J., and D. Gries
(1968) "Translator writing systems" Comm. ACM 11:2, 77-113.

GOGUEN, J.
(1975) "Complexity of Hierarchically Organised Systems and the Structure of Musical Experiences", International Journal of General Systems, Vol. 3, No. 4 pp. 237-251.

GRAY, J.
(1975) "An Exploration of Musical Timbre", Stanford University Department of Music Report No. STAN-M-2.

HAMILTON, D.
(1980) "Four Canons", tape composition realised for a BBC commission with the GGDL Computer Aided Composition System at the Department of Computer Science, University of Edinburgh.

HILLER, L.
(1957) "Illiac Suite for String Quartet".
(1959) "Experimental Music", McGraw-Hill, New York.
(1969) "Some Compositional Techniques Involving the Use of Computers", in "Music by Computers", eds. H. Foerster and J. Beauchamp, pp. 71-83, John Wiley and Sons, Inc., New York.

HOARE, C., and N. Wirth
(1973) "An axiomatic definition of the programming language PASCAL", Acta Informatica 2:4, 335-356.
(1974 - revised 1978) "PASCAL User Manual and Report", Springer Verlag, New York-Heidelberg-Berlin.

HOLTZMAN, S. R.
(1978) "Music as System", Interface Vol 7, No. 4, pp. 173-187, Swets & Zeitlinger, Amsterdam.
(1978b) "A Description of an Automated Digital Sound Synthesis Instrument", Research Report No. 59, Department of Artificial Intelligence, University of Edinburgh.
(1978c) "After Artaud", computer music composition for 4-channel tape, realised at the Institute of Sonology, Utrecht.
(1979) "An Automated Sound Synthesis Instrument", Computer Music Journal Vol. 3, No. 2, pp. 53-62, M.I.T. Press, Massachusetts.
(1980) "A Generative Grammar Definition Language for Music", INTERFACE Vol. 8, No. 2, Swets & Zeitlinger, Amsterdam.
(1980b) "The GGDL System Configuration", Proceedings of Computer Music in Britian, ed. S. R. Holtzman, pp. 17-20, EMAS, London.
(1980c) "Using Generative Grammars for Music Composition", Computer Music Journal, Vol. 4, No. 4, M.I.T. Press, Massachusetts.
(1980d) "Grammars and Computer Composition", Proceedings of Computer Music in Britain, ed. S. R. Holtzman, pp. 95-110, EMAS, London.

HOWE, H.
(1975) "Electronic Music Synthesis", Dent, London.

JAKOBSON, R.
(1970) "Main Trends in the Science of Language", Harper and Row, New York.

KAEGI, W, and S. Tempelaars
(1978) "VOSIM - A New Sound Synthesis System", Journal of the Audio Engineering Society, Vol. 26, No. 6, pp. 418-25.

KOENIG, G. M.
(1960) "Essay - Composition for Electronic Sounds" Universal Edition, Vienna.
(1963) "The Construction of Sound", unpublished manuscript.
(1970) "Project 1", Electronic Music Reports 2, Institute of Sonology, Utrecht.
(1970b) "Project 2 - A Programme for Musical Composition", Electronic Music Reports 3, Institute of Sonology, Utrecht.
(1971) "Summary: Observations on Compositional Theory", Institute of Sonology, Utrecht.
(1971b) "The Use of Computer Programmes in Creating Music", La Revue Musicale, Paris.
(1978) "Compositional Processes", presented to the UNESCO computer Music Workshop, Aarhus, Denmark, to be published in the Conference Proceedings.

LASKE, O.
(1972) "On Musical Strategies With a View to a Generative Theory of Music", INTERFACE, Vol. 1, pp. 111-125, Swets & Zeitlinger, Amsterdam.
(1973) "Introduction to a Generative Theory of Music", Sonological Reports, No. 1b, Institute of Sonology, Utrecht.

LYONS, John
(1968) "Introduction to Theoretical Linguistics", Cambridge University Press, Cambridge.

MATHEWS, M.
(1969) "The Technology of Computer Music", M.I.T. Press, Cambridge, Mass.

MATHEWS, M. and F. Moore
(1970) "GROOVE - A Program to Compose, Store and Edit Functions of Time", Communications of the ACM 13.

MATHEWS, M. and L. Rosler
(1969) "Graphical Language for the Scores of Computer-Generated Sounds", in "Music by Computers", eds. H. Foerster and J. Beauchamp, pp. 84-114, John Wiley and Sons, Inc., New York.

METCALFE, R., and D. Boggs
(1976) "ETHERNET: Distributed Packet Switching for Local Computer Networks", Communications of ACM, Vol. 19.

MORRILL, D.
(1977) "Trumpet Algorithms for Computer Composition", Computer Music Journal, Vol. 1, No. 1, pp. 46-52, M.I.T. Press, Massachusetts.

NATTIEZ, Jean
(1975) "Fondements d'une Semiologie de la Musique", Union General d'Editions, Paris.

REICH, S.
(1972) "Clapping Music", Universal Edition, London.
(1980) "Catalogue for Steve Reich", Universal Edition, London.

RISSET, J. C.
(1966) "Computer Study of Trumpet Tones", Bell Telephone Laboratories, Murray Hill, New Jersey.
(1968) "An Introductory Catalogue of Computer Synthesized Sounds", Bell Telephone Laboratories, Murray Hill, New Jersey.

ROADS, Curtis
(1978) "Composing Grammars", unpublished manuscript.
(1979) "Grammars as Representations for Music", Computer Music Journal Vol 4, No. 1, pp. 48-55, M.I.T. Press, Massachusetts.

ROBERTSON, P.
(1977) "The IMP-77 Language", Department of Computer Science Report No. 19, University of Edinburgh.

ROUGET, G.
(1961) "Un Chromatisme Africain", L'Homme 1, Paris.

RUWET, N.
(1972) "Langage, Musique, Poesie", Seuil, Paris.

SCHAEFFER, P.
(1966) "Traite des Objets Musicaux", Seuil, Paris.

SCHOENBERG, A,
(1925) "Suite Fur Klavier, Op. 25", Universal Edition, Vienna.

SMITH, L.
(1972) "SCORE - A Musician's Approach to Computer Music", Journal of the Audio Engineering Society 20, 1.

TANNER, P.
(1972) "MUSICOMP, an Experimental Aid for the Composition and Production of Music", ERB-869, Ottawa, N. R. C. Radio and Electrical Engineering Division.

TANSLEY, J.
(1977) "Multi-Computer Systems", paper presented at IUCC.

TRUAX, B.
(1973) "The Computer Composition - Sound Synthesis Programs POD4, POD5, & POD6", Sonological Reports 2, Institute of Sonology, Utrecht.

UZGALIS, R., and J. Cleaveland
(1977) "Grammars for Programming Languages", Elsevier North-Holland.

WIJNGAARDEN, Aad van
(1965) "Orthogonal Design and Description of a Formal Language", Technical Report MR 76, Amsterdam: Mathematisch Centrum.

WINOGRAD, T.
(1968) "Linguistics and Computer Analysis of Tonal Harmony", Journal of Music Theory, Vol. 12, Spring, pp. 2-49, Yale, New Haven.

XENAKIS, Iannis
(1955) "The Crisis of Serial Music", Gravesner Blatter, No. 1, Ars Viva Verlag, Mainz.
(1959) "Syrmos", Salabert, Paris.
(1971) "Formalized Music", Indiana University Press, Bloomington.
(1971b) "Musique, Architecture", Casterman, Paris.

Appendix 1: The High-Level Language Facilities of GGDL

In the GGDL programming language, high-level language facilities are provided for the definition of control functions in grammar programs, and for the definition of mapping routines in morphological mapping programs, as well as for associated routines and functions.

Non-System Rewrite Control

If, in a grammar program, the system control functions (cf. 3.2) are found inadequate for the type of control desired, the high-level language facilities may be used for the definition of non-system rewrite control functions. Any non-system control functions are called by name (with any parameters) enclosed in arrow-brackets immediately after the rewrite arrow (cf. 3.1.1) or invocation number (cf. 3.1.25), followed by the possible strings which may be selected by the function separated by periods ('.'). A rewrite rule with control by a non-system function is, for example:

```
[ X -> < FUNCNAME (PARAM1, PARAM2) > SEL1 . SEL2 . SEL3 ].
```

The control functions are written apart from the rewrite rules. They are functions to the extent that they must return an integer as a result, the number being used as an index to select a RHS string. A non-system control function may consist of a sequence of keywords, identifiers and constants together with arithmetical operators and various separator characters.

Mapping Routines

Mapping routines are called during the process of mapping a string generated by the rewriting and transformation processes. Beginning at the start of the string, the character string is read until a separator (',') is found. The read string should be the name of a terminal for which a mapping routine is defined. The mapping routine is called; during its execution it may generate appropriate output to represent the terminal, and may alter variables to influence the mapping of other terminals in the string. After returning from the routine, the next terminal is mapped, and so on until the end of the string is reached.

Mapping routines are defined by program text which, like rewrite control functions, may consist of a sequence of keywords, identifiers, constants, function and routine calls, and so on. In mapping programs, procedures for writing to output files may be called; such procedures are not permitted in grammar programs.

IMP and PASCAL Implementations of GGDL

The original implementation of the GGDL compiler and GGDLGEN compiler-compiler (see Chapter 7) was written in IMP (Robertson 1977), an Edinburgh designed programming language. The results discussed in this thesis were all obtained using the IMP implementation.

has suggested

Experience with the GGDL language a number of alterations and extensions that could be made to improve the language.

In addition, as a number of centres for computer music in Europe and the United States have expressed an interest in obtaining the GGDL composition software, a portable version of the system is required. A extended and generally improved implementation of the GGDL-compiler has been designed for implementation in PASCAL (Hoare 1973, 1974). However, the PASCAL implementation is not, at the time of submitting this thesis, yet completed.

The following definition of the high-level programming facilities in GGDL refers to the IMP implementation. Some of the differences in the PASCAL implementation are discussed afterwards.

The definition of rewrite and metaproduction rules is described in Chapter 3. What follows is a description of the programming facilities that may be used for the definition of functions and routines in the IMP implementation of GGDL.

Program Header

A program in GGDL must conform to certain formatting. Before any statements are made a program header must be given. For a grammar definition program, this is the keyword:

%GGDL

Thereafter, GGDL format requires that all global declaration statements are given, followed by initialisation blocks and any routine and function definitions. Then metaproduction rules and lastly rewrite rules may be defined. The program file is terminated by the keyword %ENDOFPROGRAM. The only compulsory statements are the header statement (%GGDL), rewrite rules (of which there must be at least one) and the %ENDOFPROGRAM statement. No routines or functions, metaproductions, etc. need be included in a GGDL grammar program. The morphological rules are defined as a separate file. Morpheme set declarations and mapping routines are illegal in a %GGDL file, as are output routines.

In a GGDL mapping program, before any statements are made the program header:

%MDL

must occur. This must be followed by the morpheme set declarations and then any global declarations. Initialisation blocks and any routine, function of mapping routine defintions may then follow. Metaproductions and rewrite rules are illegal in an %MDL file. The program file is terminated by the keyword %ENDOFPROGRAM. The only compulsory statements are the program header statement (%MDL), the morpheme set declarations and the %ENDOFPROGRAM statement.

Keywords

A keyword consists of a sequence of letters preceded by the character '%'. The keywords permitted in GGDL programs are:

%ARRAY %ARRAYNAME %C %CYCLE

```
%END        %ENDOFPROGRAM        %EXIT        %FOR
%FUNCTION        %FUNCTIONSPEC        %IF
%INITIALIZE        %ELSE        %FINISH
%INTEGER        %INTEGERNAME
%REPEAT        %RETURN        %RESULT
%ROUTINE        %ROUTINESPEC        %THEN
%UNTIL        %WHILE
```

No spaces or non-letter characters may be inserted into the middle of a keyword.

Keywords which may be used in mapping programs that may not occur in grammar definition programs are:

```
%MAP        %OUTPUT        %OUTPUTCHAR
            %MORPHEMESET
```

Identfiers

An identifier consists of a string of letters and digits of less than 20 characters, the first character of which must be a letter. Examples are:

```
IDENTIFIER        X        ID1
```

In subsequent descriptions of the control language syntax, the phrase <NAME> is used to denote the presence of an identifier.

Constants

Constants appear as operands in arithmetic expressions and may be of two forms, decimal and character. Both represent integer valued quantities - the language only has integer valued variables and performs only integer arithmetic. All values are stored as 15-bit binary numbers, with a sign bit. A decimal constant is represented by a sequence of digits:

```
10        25        66
```

No decimal points or powers of 10 are allowed (ie. real numbers or exponentiation).

Arithmetic Expressions

An arithmetic expression consists of a sequence of operands separated by operators. Operands can be array elements, constants, calls on functions or bracketed sub-expressions. The operators are:

```
+       :    addition
-       :    subtraction
*       :    multiplication
/       :    division
<<      :    logical shift left
>>      :    logical shift right
&       :    logical AND
!       :    logical OR
!!      :    logical EXcluxive OR
```

Their precedences are:

```
<<    >>              highest
*    /    &
!    !!   +    -       lowest
```

Precedence is left to right for operators of equal precedence in

an expression. <EXPR> is used to indicate the presence of an arithmetic expression.

Statements in Control-Function Language

Statements in the control-language are separated by either ';' or a newline.

Declarations

VARIABLES and ARRAYS: The names of variables and arrays denoting storage objects are declared either at the head of a GGDL program or at the head of a function or routine, in the former case being global variables and in the latter local to the routine or function within which they have been declared. Declaration statements set aside storage for those variables or arrays. If the variables or arrays are local to a routine or function, when that routine or function is left the storage space is deleted so that it may be re-used for future declarations. When routines and functions are entered recursively new storage is set aside for the variables and arrays declared without the loss of the previously declared variables and arrays. This becomes reaccessible when the recursive activation is left. This is accomplished by means of a stack.

All names must be declared before they can be used. The only objects that may be manipulated in control functions are integer values. Declarations take the forms:

```
%INTEGER <NAME>, <NAME>, ...
%ARRAY <NAME> ( Integer : Integer ), <NAME> ( Integer
        : Integer ), ...
```

Only one dimensional arrays may be defined. The lower and upper bounds of the index of the array must be integers.

Although all the names of variables declared in a single routine or function must be distinct, the same name may be declared in different routines to refer to different storage objects, and if the same name should occur in a routine or function as that of a global variable, the local storage space referenced will take precedence, the storage for the global object of the same name becoming inaccessible whilst within that routine or function. The name will always refer to the most 'local' declaration.

ROUTINES and FUNCTIONS: The name of each routine and function must be declared before it can be called. A specification statement is required before the routine or function is defined. The specification statement consists of the keyword:

```
%ROUTINESPEC
```

or:

```
%FUNCTIONSPEC
```

for a routine or function, respectively, with the name of the routine or function and an optional parameter list definition following in parentheses. The parameters must be type specified and may be of three types - integers, which pass the integer value as an argument, integer-names which pass the address of an

integer, and array-names which pass the address of an array. Only the names of global data objects may be passed and for integers any legal expression may be given. When a procedure is called a list of ACTUAL PARAMETERS must be supplied which must match the formal parameters exactly in number, order and type. In the specification statement the parameter type declarations are made by the keywords for the type, ie.:

```
%INTEGER
%INTEGERNAME
%ARRAYNAME
```

followed by an optional parenthetically enclosed number for the number of parameters of that type - by default, 1. Examples of specification statements are:

```
%ROUTINESPEC RT
%FUNCTIONSPEC FN (%INTEGER, %INTEGERNAME)
%ROUTINESPEC RT (%INTEGER (2), %ARRAYNAME,
          %INTEGERNAME (2))
```

the last of which is equivalent to:

```
%ROUTINESPEC RT (%INTEGER, %INTEGER, %ARRAYNAME,
      %INTEGERNAME, %INTEGERNAME)
```

Initialisation of Global Variables

Declared global variables may be initialised in an 'initialisation block' which is headed by the keyword %INITIALIZE and terminated by the keyword %END. Within this block any global variable may be assigned a constant integer value in statments of the form:

```
GLOBALNAME = CONSTANT
```

For example:

```
VARX = 5
```

An array location may be initialised by a statement in the block of the form:

```
ARRAYX (INDEX) = 7
```

In order to facilitate the initialisation of consecutive array locations, an initial value may be followed by a repetition count in parentheses, or an asterisk ('*') may be used to represent the number of remaining elements in the array. The following declarations are all equivalent:

```
ARRAYX (2:5) = 7,7,7,7
ARRAYX (2:5) = 7(*)
ARRAYX (2:5) = 7(4)
ARRAYX (2:5) = 7,7(2),7(*)
```

The list of constants may extend over one line though no comma should separate constants just before and after the new line.

Routines and Functions

Routines and functions are defined by a set of program statements enclosed by the keywords %ROUTINE or %FUNCTION (followed by the name of the routine or function and parameter types and names) and %END. A routine takes the form:

```
%ROUTINE RT
   ....
```

```
      %END
```
and a function takes the form:
```
      %FUNCTION FN (%INTEGER PARAM1, %INTEGERNAME PARAM2)
        ....
      %END
```
The parameters must match the specification statement parameters for that routine or function exactly in number, order and type.

The difference between routines and functions is that functions produce a value as their result and routines do not. The dynamic (run-time) exit from a routine is achieved by executing the statement:
```
      %RETURN
```
This is also implied by executing the %END statement of a routine. The dynamic exit from a function specifies a result and takes the form:
```
      %RESULT = <EXPR>
```
For example:
```
      %RESULT = I*J-K
```
Should the expression require more than one line it may be continued on the next line by ending the line with the continuation keyword, %C. Execution of the %END statement of a function is invalid since no result is specified.

The program text which defines a routine or function may consist of assignment statements, jumps (ie. redirection of program flow), labels (for jumps), routine and function calls, and IF-THEN conditional clauses, and repetition loops.

Routines and/or functions may not be defined within another routine or function, ie. routine and function definitions may not be nested.

Assignments

Variables and array elements are assigned new values using statements of the form:
```
      <NAME> = <EXPR>
      <NAME> ( <EXPR> ) = <EXPR>
```
For example:
```
      I = 2
      A (I*J) = (K+L)/2
```
Should the expression require more than one line of text it may be continued on the next line by ending the line with the continuation keyword %C.

Jumps

Labels are identifiers followed by the ':' character. For example:
```
      LABEL:
      HERE:
```
The corresponding jump instructions would be:
```
      -> LABEL
      -> HERE
```

Labels are local to a routine or function and jumps can only take place within a routine or function. Labels are not declared and jumps may precede or follow the occurence of the corresponding label without restriction.

Routine and Function calls

A routine is called by executing a statemnt consisting of the name of the routine and a correct list of parameters. The parameters must match in order, name and type exactly with the specification - integers may be any legal expressions. For example:

```
%ROUTINE RT
 ....
%END
```

would be called by the statement:

```
RT
```

Similarly, the call of a function appears as an operand of an expression. For example:

```
%FUNCTION FN (%INTEGER X, %INTEGERNAME Y)
 ....
%END
```

might be called by:

```
I = FN (VAR1+(2*VAR3), VAR2)
```

Routines and functions can be called recursively to any depth.

There is in GGDL a system provided random number generating function. This system function is called like any other function and takes two integer values as arguments - these providing the lower and upper bounds, respectively, within which the random value is generated. The name of the function is "RANDOM" and its specification is:

```
%ROUTINESPEC RANDOM (%INTEGER(2))
```

It may be called, for example, by the statement:

```
RANDOM (LBOUND, UBOUND)
```

Conditional Statements

The general form of the conditional statement is:

%IF <COND> %THEN <TEXT> %ELSE <TEXT> %FINISH

where <COND> represents the condition to be tested and <TEXT> represents program text to be executed. If the condition is true the first block of text (enclosed by the %THEN and %ELSE or %FINISH keywords) of text is executed, otherwise the second block (enclosed by the %ELSE and %FINISH keywords) is executed. The %ELSE clause is optional, it can be omitted, in which case if the condition is false execution proceeds to the instruction following the %FINISH keyword of that conditional clause.

A <COND> is formed by one or more simple relations of the form

<EXPR> <COMP> <EXPR>

where <COMP> represents one of the comparators:

= # <= < >= >

For example:

```
I <= J + K
```

Program text within the conditional statements may be assignments, routine-calls, jumps, labels, nested conditional clauses (in which case the %IF-%THEN clauses and %FINISHes must balance within any routine or function), %RETURN, and %RESULT = <EXPR>.

Repetitions (Loops or Cycles)

A group of statements may be repeated by enclosing them between the keywords %CYCLE and %REPEAT. The statements enclosed by these keywords will be referred to as the 'cycle body' and may consist of any legal statements (eg. assignments, jumps, routine or function calls, IF-THEN clauses, etc.). An unconditional repetition of text would take the following form:

```
%CYCLE
  CYCLE BODY
%REPEAT
```

It is also possible to cycle on a given condition. Conditional cycles may take three forms:

```
%WHILE <COND> %CYCLE
  CYCLE BODY
%REPEAT
```

where the cycle body is only executed while the condition is true;

```
%FOR <CONTROL> = <INIT>, <INC>, <FINAL> %CYCLE
  CYCLE BODY
%REPEAT
```

where the cycle body is repeated until the control variable <CONTROL> is equal to <FINAL>, the control variable being initialised to (when the loop is entered) <INIT> and incremented with each execution of the cycle body by <INC> - the effects of altering the control variable within the cycle body are undefined; and the last form:

```
%CYCLE
  CYCLE BODY
%REPEAT %UNTIL <cond>
```

where the cycle body is executed at least once the condition being tested after execution of the cycle body, repetition of the cycle body ending with the condition becoming true.

The keyword %EXIT may be used as a legal statement and causes the cycle to be terminated and control passed to the statement following the matched repeat.

Comments

Comments may occur in program text and will be ignored by the GGDL compiler. They may occur where any other type of program statement may begin and are indicated by the '!' character. All characters between the '!' character and the next separator will be ignored. A comment in a GGDL program might look like:

```
! This is a comment
```

or:

```
I = I + 2 ;! 'I' is incremented by two
```

Comments may not occur in the middle of other statements, for example, in between the brackets of rules or metaproductions (ie.

'[' and ']' or '"' and '"'), in expressions which may be longer than a single line, in a routine call - where the parameters and the routine name may be over several lines, etc.

Output Routines

Mapping routines will describe a process for rewriting the morpheme character strings into a different format - eg. one acceptable to a synthesis program such as MUSIC V (Mathews 1969), or non-standard digital synthesiser (Holtzman 1979), or speech synthesiser, etc. The mapped representation is output to a new file (cf. Chapter 7) In a morphological definition program it is necessary to indicate what will be sent to this file and this is done by using a system routine called by the keywords:

%OUTPUT %OUTPUTCHAR

These keywords are followed by an argument which is what is to be output by the routine. A call of the output routine may take the following forms:

1) %OUTPUT *argument*
2) %OUTPUTCHAR *argument*

The output file consists of what all the calls of the system output routines send it - in the order that it is sent.

The routine may send out data in different formats. The morphological mapping routines can output data in the form of characters, or in the form of binary (1-byte) numbers - or a mixture of the two. It is up to the user to ensure that data is of the correct type for use with a synthesis program or otherwise.

%OUTPUT

For %OUTPUT, the argument may be either one of four system known words ('SPACE', 'SPACES', 'NEWLINE', 'NEWLINES') or a bracketed expression. The system known words produce as output the character for a space (eg. ASCII 32) or a number of spaces, a newline or a number of newlines. If the plural forms are used then a number must follow in parentheses indicating how many space or newline characters are to be generated. Examples are:

%OUTPUT SPACES (3) producing ' '
%OUTPUT NEWLINE producing '
'

If the %OUTPUT keyword is followed by an expression - the lower byte of the value of the expression is output as an eight-bit binary number. For example,

%OUTPUT ((16*4)+1)

produces the binary number 01000001 as output.

%OUTPUTCHAR

With %OUTPUTCHAR, only character symbols are produced. The argument of the output routine may be either a parenthetically bracketed character string enclosed in quotes ('"') or an expression. In the former format, the character string enclosed by the quotes will be output as written, ie. as a string of character symbols. To output the quote character ('"') as part of a string the single occurence of the character is represented by

'""' in order to distinguish it from the beginning and end delimiters. In the second format, the result of the expression is output as a string of character symbols (rather than as a binary number). For example,

```
%OUTPUTCHAR ("A STRING OF CHARACTERS")
```

produces "A STRING OF CHARACTERS" (without the quotes), and,

```
%OUTPUTCHAR (11+27)
```

produces "38".

Mapping Routines

Mapping routines are defined by program text which may consist of output statements calling the system output routines and any of the statements legal in the control-function language described in 1.24 enclosed by the keywords %MAP and %END. In an MDL file, (ordinary) routines and functions may also call the system output routines. The %MAP keyword must be followed by a morpheme name, ie. a character string, which is the morpheme which the routine is called to map into another representation. By default, any declared morphemes for which a mapping routine has not been defined are output as a string of characters (which are the morpheme name without any separator characters).

An example of a mapping routine is:

```
            !  Routine maps morpheme MORPHX
%MAP MORPHX
            !  A note-card is written with 4 parameters
            ! for starting time, instrument, length of
            ! note and frequency respectively
   %OUTPUTCHAR ("NOT    ")
   %OUTPUTCHAR (TIME)      ;!  TIME is a
               ! variable equal to total time elapsed
   SEP        ;! produces a TAB character
   %OUTPUTCHAR (INSTR)      ;! function instrument
               ! selects an instrument and returns its number
   SEP
   DUR        ;! routine DUR determines the
               ! duration for this note, outputs it
               ! itself and updates the variable TIME
   SEP
   %OUTPUTCHAR ("440")      ;! same as %OUTPUTCHAR (440)
   %OUTPUT NEWLINE

%END
```

which might produce,

```
NOT     150     2     20     440
```

In the original implementation of the non-standard digital synthesis instrument (Holtzman 1978) performance data was defined as pairs of bytes, where the first byte contained the number of the sound object to be performed, and the second byte contained the duration in 50ths of a second - both were represented as binary integers. A mapping routine which produces such performance data might take the form:

```
%MAP MORPHX
   %OUTPUT (1)       ;!  MORPHX is sound object # 1
   %OUTPUT (DUR)     ;! function DUR returns a duration
 %END
```

The PASCAL Implementation

Using the IMP implementation of the GGDL programming language it was possible to determine not only whether using generative grammars could usefully aid composers in the process of composition, but also whether the language used for describing the grammars of languages, ie. GGDL, was suitably designed and whether any improvements could be made. Results using the GGDL implementation demonstrate that grammars can usefully aid composers and that the GGDL language is a flexible programming language to use for describing the grammars of music languages. The division of the generative process into three distinct stages has several advantages and the use of rewrite rules and the separation of control of rewriting in the grammar definition is an elegant way to divide the definition process.

It was no suprise, however, that a number of inadequacies of the language became apparent with its use. Therefore, based on experience gained using the IMP version of the GGDL language, a number of alterations and extensions were made for a new implementation of the GGDL-compiler. For purposes of portability, it was decided that the new implementation should be made in a language available at many computing centres. The PASCAL programming language was chosen for the implementation of an extended GGDL compiler.

The choice of PASCAL itself required several changes to be made to the GGDL language specification. The most significant of these was the operators available for arithmetic expressions. In PASCAL it is not possible to easily and efficiently implement the logical operators that were available in the original specification and IMP implementation.

Some of the changes to the GGDL language were a result of inconveniences and difficulties imposed by the rigid program format of the original specification. The specification of the PASCAL implementation will permit declarations, routine and function definitions and rewrite rules to be defined in any order, though the order of the rewrite rules' definition will still be significant. The definition of a control-function may be above or below the rule with which it is associated, rather than at the top of the program. It will also permit nested routine and function definitions.

The data-types available will be also extended to include two dimensional arrays and constant integers and arrays. as well as the declaration and use of strings and string functions. A string function will be allowed to be called to generate a RHS in a rewrite rule; rather than returning an index to the selection the function will return a string which itself will be the selection.

In addition to several operators that may be used to relate strings, such as concatenation and pattern matching, several system routines have been provided for manipulating strings. Because the strings are strings of terminals and non-terminals, where character sub-strings are the names of these objects, it is possible to treat the strings like 'linked names'. System routines permit the the removal of the 'head' or 'tail' of the string, or the joining of a new 'head' or 'tail' to the string.

The effectiveness of these improvements can only be gauged after further experience with the language. The implementation of a PASCAL GGDL compiler and GGDL-Generator compiler-compiler has not yet been completed.

Appendix 2: Example GGDL Programs

An Example GGDL Grammar Program

This is an example GGDL generative grammar definition program. The generative grammar defined by the program describes a language which includes among many utterances, the note-duration structure of the Trio (in an 'abstract' representation) from Schoenberg's Piano Suite, Op. 25 (Schoenberg 1925) of Figure 4-2. This program was used to generate the 'abstract' (ie. morphological) representation of the structure of the composition shown in Figure 4-3. With minor changes, initialising "OBJNUMBER" to '9' rather than '12' in the 'initialisation block' and changing 'Meta-Rule 1' so that it only generated a 'SERIES' from nine objects, the same program was also used to generate the abstract structure of the composition given in Figure 4-4.

```
        ! *** A GGDL GRAMMAR DEFINITION PROGRAM ***
%GGDL

        ! program declarations
%functionspec dstringsel
%functionspec selectduration
%integer objcnt, objnumber, seriescnt, fsstring

%initialize
objnumber=12     ;! "objnumber" is total number of sound objects.
fsstring=3       ;! The third rhythmic group will be generated using
                  ! a finite-state metaproduction rule (cf. Meta-
                  ! Rule 3 "DURATIONS FOR THIRD SERIES").
                  ! "objcnt" and "seriescnt" by default are initialized
%end             ;! to zero


        ! function definitions

%function selectdurationstring
%integer selection
          ! Each time the function is invoked it selects a string
          ! of duration values for a new SERIES of objects.
          ! For the 'fsstring' of each STRUCTURE it selects the
          ! string of durations generated by Meta-Rule 3.
          ! Otherwise the string selected is that generated by
          ! Meta-Rule 2.
     ! a global variable counts the SERIES of a STRUCTURE
   seriescnt=seriescnt+1

   %if fsstring=seriescnt %then
      selection=2
   %else
      selection=1
   %finish

        ! reset "seriescnt" ie. modulo 4
```

```
   %if seriescnt=4 %then
       seriescnt=0
   %finish

         ! selects "RHYTHM" string type
   %result=selection

%end


%function selectduration
             ! function "selectduration" selects a morpheme which
             ! represents a duration value.
      ! a global variable counts the objects in the series
     objcnt=objcnt+1

     %if objcnt <= 6 %then
        %if objcnt=6 %then %result=2 %else %result=1 %finish
     %else
        %if  objcnt=objnumber %then
               ! resets "objcnt" when "SERIES" is complete
           objcnt=0
           %result=4
        %else
           %result=3
        %finish
     %finish

%end




! ********************** GRAMMAR RULES ******************************

       ! "SERIES" represents a string of all the objects
       ! - the total number being assigned to "objnumber" -
       ! in which each object occurs once only.
       ! "SERIES" acts as a variable and with different generations
       ! using this grammar one can alter the ordering of objects in
       ! the group without affecting the macro-structure.
! META-RULE 1
" SERIES -> # objnumber # ! obj1,  obj2,  obj3,  obj4,  obj5,
        obj6,  obj7,  obj8,  obj9,  obj10,  obj11,  obj12"
! META-RULE 2
" DURATIONS OF SERIES -> #objnumber# <selectduration>
      duration 1, . duration 1 * 3, . duration 2, . duration 2 * 3, "
! META-RULE 3
" DURATIONS FOR THIRD SERIES -> # objnumber #  * <4,1>
     ( duration 2, . 3 . 1 . 0 . 0 )
     ( duration 1, . 0 . 0 . 2 . 2 )
     ( duration 3, . 0 . 1 . 0 . 0 )
     ( duration 4, . 1 . 0 . 0 . 0 ) "
```

```
        ! THE REWRITE RULES

        ! The macro structure of the composition will always consist
        ! of two "STRUCTUREs" in canon.  the structures have the same
        ! 'SERIES-RHYTHM' relations such that the grammar defined may
        ! generate macro-equivalent canons with different SERIES.
! RULE 1
[ CANONSTRUCTURE -> voice1, STRUCTURE, voice2, STRUCTURE ]

        ! A structure consists of four different versions
        !  of the SERIES, each with an associated rhythmic structure.
! RULE 2
[ STRUCTURE -> # 4 # M ( Z (GROUP) ) (RHYTHM)  ]

        ! The non-terminal "VERSION-TYPE" generates a unique version of
        !                          the SERIES.
        ! There are eight versions - the original, the original inverted,
        ! the retrograde, and the retrograde inverted, and these transposed
        ! up by an interval of a six semitones.
! RULE 3
[ VERSION-TYPE (_) -> !  _ . I (_) . B (_) . B ( I(_) )  .
          T #6# (_) . T #6# ( I (_) ) . T #6# ( B (_) ) .
          T #6# ( B ( I(_) ) )  ]

        ! The above rules generate an 'object-structure'.
        ! Durations for the objects of a SERIES may be generated
        ! either by using the function 'selectduration' to select
        ! durations from four possible durations (cf. Meta-Rule 2)
        ! or using a finite-state rewrite rule (cf. Meta-Rule 3).
        ! In both cases 'objnumber' durations, ie. one duration
        ! for each object of the SERIES, are generated.

! RULE 4
[ RHYTHM -> <selectdurationstring> DURATIONS OF SERIES  . DURATIONS
                    FOR THIRD SERIES ]

%endofprogram
```

An Example GGDL Mapping Program

This is an example GGDL morphological definition program. The GGDL grammar program generates the 'abstract' structure of Figure 4-3b. The mapping program given below was used to map the abstract representation of this structure to an easily transcribable representation. An example of the generated output of the GGDL grammar program using the mapping program below is given in Figure 4-3a.

```
          ! *** %MDL MORPHOLOGICAL MAPPING PROGRAM ***
! generating output for transcription into tradtional music notation.

%MDL

%morphemeset    ;! Declarations of the morphemes.
   [ obj1,obj2,obj3,obj4,obj5,obj6,obj7,obj8,obj9,obj10,obj11,obj12 ]
   [ voice1,voice2,duration 1,duration 2,duration 3,duration 4,
            duration 1 * 3, duration 2 * 3 ]
%end

      ! Declarations of routines and variables.
%routinespec writeduration (%integer)
%routinespec writedvalsy (%integer)
%routinespec seriesnumbercheck
%integer objnumber,objcnt,seriescnt,fsstring,durval,voiceflg
%integer measure,entry

%initialize
objnumber=12       ;! "objjnumber" is total number of sound objects.
fsstring=3        ;! coordinates with %GGDL program
durval=1           ;! time-unit for rhythmic structure
                    ! ie. one 16th note.
%end

              ! mapping routines

%map obj1
   %outputchar (1)
%end

%map obj2
   %outputchar (2)
%end

%map obj3
   %outputchar (3)
%end

%map obj4
   %outputchar (4)
%end
```

```
%map obj5
   %outputchar (5)
%end

%map obj6
   %outputchar (6)
%end

%map obj7
   %outputchar (7)
%end

%map obj8
   %outputchar (8)
%end

%map obj9
   %outputchar (9)
%end

%map obj10
   %outputchar (10)
%end

%map obj11
   %outputchar (11)
%end

%map obj12
   %outputchar (12)
%end

                    ! duration morphemes

%map duration 1
   objcnt=objcnt+1
   %if seriescnt#fsstring %then
      writeduration (durval*2)      ;! ie. 1/8th note
   %else
      fsstring (durval*2)
   %finish
   seriesnumbercheck
%end

%map duration 2
   objcnt=objcnt+1
   %if seriescnt#fsstring %then
      writeduration (durval)    ;! ie. 1/16th note
   %else
      fsstring(durval)
   %finish
   seriesnumbercheck
%end

%map duration 3
   objcnt=objcnt+1
```

```
   fsstring (durval * 4)    ;! ie. quarter note
   seriesnumbercheck
%end

%map duration 4
   objcnt=objcnt+1
   fsstring (durval * 6)     ;! ie. dotted quarter note
   seriesnumbercheck
%end

%map duration 1 * 3
   objcnt=objcnt+1
   writeduration (durval * 6)     ;! ie. dotted quarter
   seriesnumbercheck
%end

%map duration 2 * 3
   objcnt=objcnt+1
   writeduration (durval * 3)    ;! ie. dotted eighth note
   seriesnumbercheck
%end

               ! mapping "voice" morphemes
%map voice1
   voiceflg=1             ;! indicates which voice is being generated.
   seriescnt=1
   %outputchar ("voice1")
   %output newline
   measure=1
   entry=3
%end

%map voice2
   voiceflg=2
   seriescnt=1
   %outputchar ("voice2")
   %output newline
   measure=2
   entry=3
%end

              ! routine definintions
%routine writeduration (%integer duration)
   %outputchar ("(")
   %outputchar (measure)
   %outputchar (",")
   %outputchar (entry)
   %outputchar (")")
   %output newline
   entry=entry+duration
      %if entry > 12 %then
         measure=measure+1
         entry=entry-12
      %finish
%end
```

```
%routine fsstring (%integer duration)

   %if objcnt <= 4 %then
     writeduration (duration)
   %else
     %if objcnt <= 8 %then
        %if objcnt=5 %then
                     ! ie. first of 2nd group, reset counters
          %if voiceflg=1 %then
             measure=5
          %else
             measure=6
          %finish
          entry=3
        %finish
        writeduration (duration)
     %else
                     ! last group
         %if objcnt=9 %then
                     ! ie. first of last group, reset counters
          %if voiceflg=1 %then
             measure=6    ;!  ((((6*2)+(2*2))+((6*1)+(1*2)))*2) / 12 = 4
                     ! ie. the first two "dvalsx" groups are
                     ! 2 measures each. "dvalsy" will also be two
                     ! measures:      6 - 4 = 2
          %else
             measure=7
          %finish
             entry=13
                     ! output silence as twice duration
                     ! of last - ie 9th object - object of group
             entry=entry-(duration*3)
             writeduration(duration)
             %outputchar ("s")
             writeduration(duration*2)
          ! reset "entry" and "measure" as overflow into next measure
          %if voiceflg=1 %then
             measure=6
          %else
             measure=7
          %finish
             entry=13-(duration*3)
         %else
                     ! other objects of the group
             entry=entry-duration
             writeduration (duration)
             entry=entry-duration
                     ! if last of group reset counters to 'normal'
             %if objcnt=objnumber %then
               %if voiceflg=1 %then
                  measure=7
               %else
                  measure=8
               %finish
                        ! add extra silence so next group
                                 ! begins on upbeat
```

```
                    %outputchar ("s")
                    entry=1
                    writeduration (2)
                %finish
            %finish
        %finish
    %finish
%end

%routine seriesnumbercheck
            ! checks if it is end of object-group and resets
            ! counters if it is
    %if objcnt=objnumber %then
      objcnt=0
      seriescnt=seriescnt+1
    %finish
%end

%endofprogram
```

Appendix 3: A Grammar for Generating Non-Standard Sound Synthesis 'Functions'

A GGDL grammar program which describes the grammar the 'automated non-standard synthesis instrument' (see Chapter 6) uses to generate 'Functions' is given below. The grammar generates a program of virtual instructions. In order to execute code generated on a 'real' machine, a mapping program needs to be defined to 'compile' the virtual program.

In the case of the 'automated non-standard synthesis instrument' described in this thesis, the grammar was directly implemented on the PDP-15/40 which supports it. That is, rather than use the GGDL grammar program and a mapping program, a more efficient implementation was written for the PDP-15/40 which runs as a 'system utility' under the operating system designed and implemented to support the synthesis instrument (see Chapters 6 and 7).

In this implementation, a user may influence the generation of non-standard descriptions of sounds by assigning values to a number of variables in the control mechanism of the grammar of the synthesis instrument. The variables may be used, for example, to control the number of Functions generated with a particular set of grammar variables (ie. 'numberoffunctions' in the grammar), the bounds for the number of 'statements' in a Function ('minstatements' and 'maxstatements'), the relative occurrence of the different types of statement expressions (ie. 'memoryweight', 'randomweight' and 'operstringweight'), the bounds of the number of operators in an expression (ie. 'minoperators' and 'maxoperators'), and the relative occurrence of assignment and conversion statements (ie. 'assignstatement' and 'outstatement'). Other variables control the number of constants and variables and the relative frequency of their use in a Function.

The values for the variables of the grammar are transmitted as 'messages' from the VAX-11/780 to the PDP-15/40 and similarly, instructions for writing and performing Functions may be transmitted. In this implementation, it is not possible for a user to alter the rewrite-rules or the available control functions.

In the grammar below, the control functions are written immediately below the rewrite rule with which they are associated. In the IMP-implementation (see Appendix 1) of GGDL, all functions are defined before any rewrite rules are given; this freer format is permitted in the PASCAL-implementation of the GGDL language.

```
!********************** GGDL Grammar for ****************************
! *****  Non-Standard Noise Synthesis Program Code Generation  *****

     ! This is a Generative Grammar Definition Language program
     ! which describes the grammar the Non-Standard Digital Synthesis
     ! machine uses to generate noise-synthesis programs (ie.
     ! compose noises).   From the 'Non-terminal' SOUNDOBJECT the
     ! program code for a 'Function' (using a virtual instruction set)
     ! is generated.
     ! The user may control the generation of the grammar by
     ! the assignment of the 'control' variables of the grammar.
     ! The Function written will consist of a string of virtual
     ! machine instructions which can be mapped onto the machine
     ! instruction set of a given machine with a 'morphological
     ! mapping' program (which may also be defined in GGDL).
          ! underlined integer names are user assignable
          !         control variables of the grammar.
          ! CAPITALIZED character strings are the 'non-terminals'
          !         of the generative grammar.
          ! The 'terminals' of the language - ie. the virtual
          ! machine instructions - are lower case character strings.

%GGDL
          ! the selection functions used for rewrite control
%functionspec numberofstatements, chooseoperation, numberofoperators,
                chooseoperands, choosestatementtype
          ! RANDOM is a system known function which returns a
          ! number randomly selected between two bounds - it
          ! needn't be declared
%functionspec random (%integer(2))

          ! the user-assignable control variables
%integer numberoffunctions, minstatements, maxstatements,
        memoryweight, randomweight, operstringweight, minoperators,
        maxoperators, outstatement, assignstatement, minvars, maxvars,
        mincons, maxcons, varweight, conweight


                  ! ***** The REWRITE RULES *****

[ SOUNDOBJECTS -> # numberoffunctions # FUNCTION ]

[ FUNCTION -> # numberofstatements # STATEMENT ]

     %function numberofstatements
         %result = random ( minstatements, maxstatements )
     %end

       ! STATEMENTs may be memory-fetches, strings of operators or the
       ! random operator - the result of the computation (left
       ! in the accumulator) is sent to DAC or deposited in memory.
[ STATEMENT -> STATEMENT-TYPE, OPERATION, ]

[ OPERATION -> < chooseoperation > random . memory-fetch, OPERAND .
```

```
                    OPERATORSTRING ]

     %function chooseoperation
         %integer temporary
         temporary = random ( 1, memoryweight + randomweight +
                              operstringweight )
         %if temporary <= memoryweight %then
             %result = 1
         %else
             %if temporary <= memoryweight + randomweight %then
                 %result = 2
             %else
                 %result = 3
             %finish
         %finish
     %end

[ OPERATORSTRING -> # numberofoperators # OPERAND, OPERATOR, OPERAND ]

     %function numberofoperators
         %result = random ( minoperators, maxoperators )
     %end

       ! This 'context-sensitive' rule ensures that operation syntax
       ! is correct where operators are in strings
[ OPERAND, OPERAND -> OPERAND, ]

       ! In this finite-state transition matrix all the transition
       ! values ( t ) are independently assignable.
[ OPERATOR -> * <10>
     (  add(+)   . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  conj(&)  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  antiv(!) . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  minus(-) . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  mult(*)  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  div(%)   . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  disj     . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  equiv    . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  implic   . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  )
     (  exclu    . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  . t  ) ]


        ! OPERANDs may be either VARiables or CONstants
[ OPERAND -> < chooseoperand > VAR . CON ]

     %function chooseoperand
         %if random ( 1, varweight + conweight ) <= varweight %then
             %result = 1
         %else
             %result = 2
         %finish
     %end

          ! The "!" directly after the rewrite arrow indicates
          ! to the generative mechanism that each of the right-hand
          ! side terminals may only occur once until all others
```

```
            ! have occurred at least once.  This helps ensure semantic
            ! consistency by prohibiting the use of the same operand
            ! twice in one phrase!
[ VAR -> ! var1 . var2 . ... varN ] ;! where N =
                                          !   random ( minvars, maxvars )

[ CON -> ! con1 . con2 . ... conN ] ;! where N =
                                          !   random ( mincons, maxcons )

[ STATEMENT-TYPE -> < choosetermin > outputtoDAC . assign, VAR ]

      %function choosetermin
           %if random ( 1, outstatement + assignstatement ) <=
                                 outstatement %then
              %result = 1
           %else
              %result = 2
           %finish
      %end

%endofprogram
```

Appendix 4: Using the GGDL CAC System

The Edinburgh GGDL composition/synthesis system consists of a two computer processors interconnected to permit the parallel performance of tasks. These include the generation of data required for performance, performance, ie. synthesis, of sounds, and control of 'message' transmissions. A user interactively sends commands to the control processor (ie. the VAX-11/780) from a video terminal. One synthesis instrument may be controlled, ie. the PDP-15/40 (see Chapter 7).

GGDL Compiler

The compiler accepts both GGDL grammar and GGDL morphological mapping definition programs prepared using a standard text editor. On VAX, the compiler is run with the 'source' GGDL definition file as input and the 'object' file as the first output stream. Optionally, a second output stream may be specified for an alpha-numeric listing of the generated object code (which may be useful for debugging). Compilation errors are reported to the user's terminal. A list of error messages is given in Figure A4-1.

A run of the compiler on VAX might look like:

```
$GGDL
Streams: SOURCE/OBJECT,LISTING
```

GGDL-Generator

The GGDL-generator program accepts three input streams and one, and optionally a second, output stream. The input streams correspond to 1) the 'initialising' string for generation, 2) the compiled GGDL grammar definition, and 3) the compiled GGDL morphological definition. Either of the latter two (but not both) may be omitted in which case either an 'abstract' structure will be output (ie. a morpheme string) or the input to stream one will be morphologically mapped (without grammar generation). This enables one to generate and map structures independently. The specification of output streams is optional. The output of the grammar and/or morphological processing is to stream 1 - which by default is the user's terminal. A trace of the generation and/or mapping processes can be output to stream 2. This may be useful for debugging purposes.

A running of the GGDL-Generator (invoked as "GGDLGEN") might look like:

```
$GGDLGEN
Streams: STRING,GRAMMAR,MAPPING/STRUCTURE,DIAGNOSTICS
```

Run-time error messages executing either the grammar or mapping program are reported to the terminal. A list of the error messages is given in Figure A4-2.

COMPILER ERROR MESSAGES

Any errors detected by the compiler will generate error messages indicating the type of error found, the line number (in the input stream) and the text of the line where the error was found. The following types of errors may be detected:

ARRAY BOUNDS FAULT	-	array bounds exceeded in an %INITIALIZE block statement
BRACKETS DO NOT BALANCE	-	incorrect bracketing in an expression
COMMAND NAME ERROR	-	a keyword command unknown to the compiler or illegal in its context
DUPLICATE	-	duplicate name of same type in declarations
? EXIT	-	queries whether exit from a cycle loop is possible - this is not a fault to the compiler
FORM	-	improper form in statement
IF-THEN-CLAUSE MISSING	-	generated by an extra %FINISH
IMPROPER IF-THEN-ELSE SYNTAX	-	eg. %ELSE out of place
INCORRECT NUMBER OF ARGUMENTS	-	in a function or routine
NAME	-	an undeclared name
CYCLE MISSING	-	generated by an unmatched %REPEAT statement
RESULT ILLEGAL IN ROUTINE	-	%RESULT found in a routine definition
RETURN ILLEGAL IN FUNCTION	-	%RETURN found in a function definition
THEN MISSING	-	%THEN not found after %IF statement
TYPE ERROR	-	eg. a routine or function parameter does not match its type specification

The following error messages are given with a line number only at the end of a routine or function definition:

FINISH MISSING	-	incomplete IF-THEN-FINISH clause
LABEL MISSING	-	a jump in text is made to a label not used
REPEAT MISSING	-	%CYCLE loop without matching %REPEAT statement
RESULT MISSING	-	%RESULT statement not found in a function

If certain faults are detected by the compiler, compilation will be abandoned.

ILLEGAL PROGRAM HEADER STATEMENT		
LEXEME SET DECLARATION BLOCK MISSING	-	in an %MDL program
TOO MANY ERRORS		

Figure A4-1: GGDL compiler error messages.

RUN-TIME ERROR MESSAGES

Errors may occur during the execution of either a compiled %GGDL or %MDL program by the GGDLGEN generator. An error message is transmitted and execution of the program halted. Some run-time diagnostics may also be given. The error messages that may be transmitted are:

INPUT STREAM 2 IS NOT A PROPERLY COMPILED %GGDL GRAMMAR DEFINITION FILE
- not a compiled %GGDL program as input

INPUT STREAM 3 IS NOT A PROPERLY COMPILED %MDL MORPHOLOGICAL DEFINITION FILE
- not a compiled %MDL program as input

With the following messages the contents of the program counter, the number of the rule or mapping function name (in a %GGDL or %MDL program respectively), and the dynamic listing of the program are also given.

ARRAY BOUNDS FAULT IN GLOBAL ARRAY - array bound index out of specified
ARRAY BOUNDS FAULT IN LOCAL ARRAY bounds

ILLEGAL INSTRUCTION - attempt to execute an illegal instruction

CONTROL FUNCTION HAS RETURNED FOR RHS SELECTION A VALUE OUTSIDE OF BOUNDS OF RHS ELEMENTS - control function has selected element out of bounds

The below error messages also print the generated string at the time of the error - ie. with remaining structural change indexes, etc. - indicating the transform being executed.

UNDECLARED MORPHEME - undeclared morpheme found either at transformational or morphological mapping stage of generation - the morpheme is given

RIGHT BRACKET IS MISSING FOR TRANSFORMATIONAL ARGUMENT
- incorrect bracketing

LEFT BRACKET IS MISSING FOR TRANSFORMATIONAL ARGUMENT
- incorrect bracketing

ARGUMENT IS NOT A VALID NUMBER - illegal transpose or merge number

UNKNOWN TRANSFORMATION - IE. STRUCTURAL CHANGE INDEX
- unknown transformation indexed

INSUFFICENT NUMBER OF OBJECTS FOR MERGE - not the same number of objects in all sets of merge argument

Figure A4-2: GGDL-Generator run-time error messages.

Graphics Editors

DRAW: Draw is a simple graphic editor with a small menu of commands which may be used to define/edit waveforms (consisting of 4096 12-bit samples) or envelopes (with an arbitrary number of turning points).(1) It is invoked as "DRAW". The program will prompt the user to set the "Mode". This may be either a 'Wave' or 'Envelope'. Having set the mode, a user may then use the menu of commands to define a set of points in a 4k by 4k window - which correspond to the real points of a waveform (the points are used for interpolation) and the virtual points of an envelope. The available set of commands is:

1) Waveform and envelope definition files, as well as performance files, and so on, are kept as alpha-numeric files. This permits any files to be edited and inspected with a standard text editor as well. The format of definition files generated using the graphic editor is compatable with the format of messages for transmission to the synthesis instrument. That is, files generated using the graphics program may be sent as messages to the PDP-15/40.

P(x,y) - This defines a point by stating the x and y coordinates in the 4k by 4k window by integer values between 0 and 4095. The point will be 'marked' and the wave or envelope shape drawn.

C - This invokes the cursor. The x and y coordinates may be set by use of a cursor. When the cursor is set any character on the keyboard may be hit and the point will be 'marked' and the wave or envelope shape drawn.

D - This may be used to delete a point. The program will prompt for the X-coordinate. If the user responds with a newline, the cursor may be used to define the point, otherwise an integer must be given.

Store (or Save) - The program will prompt for a filename and will store either in wave or envelope format the data from the shape drawn. The default extension of the filename is either ".WAV" or ".ENV". Waves are stored as properly formatted messages which may be transmitted to instruments.

Fetch - This command 'fetches' from memory a file containing data for either a wave or envelope. Only files generated by use of the store command may be used. The program will prompt for for the filename. By default the program takes ".ENV" or ".WAV" as the filename extension.

%C - All the above commands are applied to the currently 'set' wave or envelope (initially '1'). Up to four waves or envelopes may be edited at one time - all defined waves or envelopes present

appearing on the screen. This command (%C) allows one to exit from the definition/editing process of a wave or envelope using the above commands. After the command '%C' the program will prompt with "Set", a request to change to one of possibly four shapes being defined (user responds with an integer between 1 and 4). If there is no data present for the set editing file, the program will prompt with "Mode". The status of all files being edited is kept on the top part of the screen.

R - This resets the presently edited file, ie. erases the wave or envelope and clears the file. The program will prompt with "Mode".

$ - This will exit from the program and return the user to monitor level. All files not stored are lost.

PLOT: Plot is run with a specified input stream - either a wave or envelope - generated by the DRAW program, any other program or manually with the text editor. It plots the wave or envelope on the screen and permits the visual inspection of waves or envelopes not generated by the DRAW program. The file to be plotted must, in the case of a wave, consist of 4096 points (integer values between 0 and 4096), and, in the case of an envelope, an arbitrary number of x and y coordinates.

A run of the PLOT program on VAX might look like:

```
$PLOT
Streams:FILE
```

Utility Programs

Any programs available on the central processor system may be accessed and run from monitor level. These may include user defined programs, system editors, compilers, graphics, VLSI design, etc.

Communications

'Messages' may be transmitted from VAX to the PDP-15/40. Messages may be of five types:

1) Performance-data Messages
2) Execution Commands
3) Waveform Definitions
4) Function-writing control data (for Non-standard Synthesiser)
5) Abort Command

A user will prepare messages as text files - either with an editor or directly typing in a message whilst inside the message compilation/transmission program. The user's text must be checked for correct formatting and data types, compiled and assembled into a machine readable representation that can be transmitted (as bytes) between the two processors, and then packed with a correct identifying header and end-of-file character. This can be done by using the commands available for instructing the message

compilation and transmission program, invoked as TALK. The available set of commands is:

P - Transmit a 'Performance Data' message. The program will prompt for the input stream. This may be either a filename, or, if the stream prompt is answered with a newline, direct input from the terminal.

X - Transmit an 'execution-command' message. The program will prompt for the execution command. These are:

E - execute performance data

W - write Functions (on non-standard synthesiser) using received Function-Data messages.

G - write Functions and then execute performance data.

P - reset performance data (ie. erase all P-messages received).

D - clear waveform tables (ie. erase all W-message received).

R - reset (ie. overwrite any written) functions

F - clear Function writing control information (for PDP-15 non-standard synthesiser)

T - clear Operator Transition Matrix (for PDP-15 non-standard synthesiser)

Z - reinitialise instrument (ie. erase all messages received).

W - Transmit a waveform definition - ie. 4096 12-bit samples. The program prompts for the input stream to which the user responds with a filename, by default, its extension is ".WAV".

F - Transmit Function writing control information for the non-standard synthesiser. Data in 'F-messages' corresponds to the non-standard synthesisers grammar's control variables (see Appendix 3). The program prompts for the stream which may be either a filename or newline (ie. data input from terminal).

T - Transmit values for the Operator Transition Matrix for the non-standard synthesiser. The program prompts for the stream which may be either a filename or newline (ie. data input from terminal).

A - This aborts whatever the instrument is doing. If it is performing it deactivates the performance processes and waits for further messages containing command instructions. If the non-standard synthesiser is writing Functions, it abandons this process.

Messages may contain comments - which will be ignored by the compiler. These are preceded by '!' and all characters until the next newline line are ignored. Comments may not be included within statements in a message.

Performance Messages consist of three types of statements (or instructions) for the performance control process of a given instrument. These are 'synthesiser identifiers' which tell the performance control process what type of synthesis is to be performed (subsequent performance data must be in the appropriate format for that type of synthesis), 'wave identifiers' which tell the performance control process which waves are to be used for synthesis (if waves are used!), and performance data - ie. an event queue of sounds to be synthesised.

'Synthesiser identifiers' take the form:

```
I = n
```

where 'n' is an integer value identifying the synthesis technique to be used. Only one type of synthesis technique may be used at a time (on any given instrument). For values of 'n':

```
1 = Direct-waveform synthesis (one voice only)
3 = Non-standard synthesis (on PDP-15 instrument only)
4 = Direct-waveform synthesis (in two voices)
5 = Direct-waveform synthesis with envelope shaping
            (one voice only)

6 = Waveshaping Synthesis
```

By default, the synthesis method used it set to direct-wave synthesis.

The sampling rate on the PDP-15/40 for direct-wave synthesis is (for one voice) 30,927 hz. - with a maximum frequency of one quarter that. For two voices, approximately half of that (15472), one voice with envelope shaping just below that (13897).

'Wave identifiers' take the form:

```
W = n
```

where 'n' is an integer value between 1 and 4 selecting the nth wave transmitted to the given instrument (as a message). By default, the first wave transmitted is selected.

Performance Data varies for the type of synthesis to be performed. In the first case, for direct wave synthesis, an event consists of a 'P' (ie. perform) statement consisting of a frequency (a 'real-integer' value) and a duration (in 50ths of a second). Values are separated by a comma and enclosed in parentheses. For example,

```
P(1000,50)
```

which will perform a wave at a frequency of 1000 hz. for 1 second. Silences are written as 'S' statements with a duration given in 50ths of a second, eg.,

```
S(100)
```

which performs a two second silence.

For non-standard synthesis, performance data consists of voice identifiers - either 'V1' or V2' for the two voices possible - and event data. Events are specified by either an integer value identifying the noise to be performed or an 'S' (indicating a silence), and a duration given in 50ths of a second. The performance data for a voice is terminated with a '$'. Data for voice 1 to be performed simultaneous with voice 2 must precede the data of the second voice. If, after data for voice1 and/or voice2, more data is given, its performance will begin after the completion of the performance of any preceding data. All data for the non-standard synthesiser must be terminated with a '$'. Performance data for the non-standard synthesiser could be:

```
V1
1,100
2,100
$
V2
S,100
1,100
2,100
$
V1
1,100
$
$
```

Performance data for direct wave synthesis in two voices is specified as that for direct-wave synthesis (ie. " P(frequency,duration) " etc.) with voice identifiers (as in instrument 3) and an exit from the instrument indicated by a '$' (after the dollar closing the last voice's data).

Waveform Messages consist of a list of 4096 waveform samples of integer values between 0-4095 (ie. 12-bits). Values must be separated by newlines. The message must be terminated with '$'.

Function-Data Messages consist of "Index/Value" pairs (terminated with a '$').

Indices are numbers between 1 and 16 and refer to the user assignable control variables of the PDP-15 Non Standard Noise Synthesis Program Generator Grammar (see Appendix 3).

```
 1 = number of functions to be written with the data
 2 = minimum number of statements in a function
 3 = maximum number of statements in a function
 4 = operator-statement weight
 5 = memory-statement weight
 6 = random-statement weight
 7 = output statement weight
 8 = assignment statement weight
 9 = minimum number of operators in a operator statement
10 = maximum number of operators in an operator statement
11 = variable weight
12 = constant weight
13 = minimum number of variables (for a function)
```

```
14 = maximum number of variables
15 = minimum number of constants (for a function)
16 = maximum number of constants
```

Values must be of appropriate size (0-256). A maximum of 30 functions (in total) may be written, consisting of no more than 500 statements. Each function may have a maximum of 25 variables and 25 constants. Where values do not make 'sense' - eg. lower bound is greater than upper bound - the system will impose semantic consistency on the values.

Packets are terminated with '$'.

An example Function-Data Packet is:

```
 ! This Function-Data Packet generates a Ramp Wave
 ! if only the addition operator is permitted in
 ! statements (ie. Operator Transition Matrix (1,1)=1)
 ! Two statements are generated. Though both are
 ! indicated as assignment statements, the program
 ! generator forces the second statement to be an
 ! output statement to attempt to ensure semantic
 ! consistency: no sound will be generated if no values
 ! are output to the D-A convertor. The operator
 ! phrase will always add a constant and a variable
 ! and assign the result to the variable: ie. X=X+Y.
 ! The assignment phrase will output the same expression,
 ! ie. DAC <- X+Y. This generates a ramp-wave. Both
 ! values will be used in the expressions as values
 ! are not used twice, a system imposed constraint to
 ! ensure that 'senseless' statements are not generated
 ! (see Chapter 6), and there are only two values that
 ! may be used: one constant and one variable.
 1=1
 2=2
 3=2
 4=1
 5=0
 6=0
 7=0
 8=1
 9=1
10=1
11=1
12=1
13=1
14=2
15=1
16=1
$
```

Operator Transition Matrix Messages consist of "Two-dimensional array Index/Value" pairs (terminated with a '$').

The indexes to the two dimensional array refer to the Operator Transition Matrix of the PDP-15 Non-standard Noise Synthesis Program Generator Grammar (see Appendix 1) - a 10 by 10 matrix. The index is written as two integer values corresponding to the matrix row and column, separated by a comma (',') and enclosed by parentheses.

Values must be integers between 0-255.

Messages are terminated with '$'.

An example Operator Transition Matrix Message is:

```
        ! This is an Operator Transition Matrix Message
(1,1)=10    ;! This permits addition to follow addition
            ;! Multiplication may follow addition
(1,5)=20    ;! (twice as often as addition follows addition)
(5,1)=10    ;! Multiplication is followed by addition

$
```

The transmission of messages may be compiled and transmitted in an interactive manner from a terminal connected to the control processor. The control processor transmits the message to the synthesis instrument and conveys to the user any error messages received from the synthesis instrument. A conversation might look like:

```
$TALK

COMMAND:X
Execution Command:E

PDP-15 cannot execute command "E"
Performance Data-packet required

COMMAND:P
Stream:
P(1000,100)
$

COMMAND:X
Execution Command:E

PDP-15 cannot execute command "E"
Require more waveforms for performance
        0 waveforms defined
        1 required for performance
```

```
COMMAND:W
Stream:SINE.WAV

COMMAND:X
Execution Command:E

COMMAND:$

$
```

In this conversation, the 'message' transmission program is run - invoked as "TALK". A request to perform is sent to the PDP-15/40. The PDP-15/40 cannot perform without performance data. Performance data is typed at the console. It specifies the performance of a pitch of 1000 hertz for 2 seconds (ie. 100 * 50th second, the default clock setting) using the default, ie. the first transmitted, waveform. Again, a request to perform is sent and the PDP-15/40 responds that further data is required, the waveform. When performance messages are received they include a header specifying how many waveforms and non-standard noises are required to execute the event-queue. In this case, one waveform is required, but none have been transmitted to the PDP-15/40. A sine wave is transmitted and the performance data is then executed.