FORMAL LANGUAGES IN MUSIC THEORY

by

Glendon Diener
Faculty of Music
McGill University, Montreal

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment
of the requirements for the degree of
Master of Arts

© Glendon Diener 1985
ABSTRACT

In this paper, the mathematical theory of languages is used to investigate and develop computer systems for music analysis, composition, and performance. Four prominent research projects in the field are critically reviewed. An original grammar-type for the computer representation of music is introduced, and a computer system for music composition and performance based on that grammar is described. A user's manual for the system is provided as an appendix.
RÉSUMÉ

Dans ce travail, nous utilisons la théorie mathématique des langues afin d'examiner et de développer des systèmes informatisés visant l'analyse, la composition et la réalisation de la musique. Quatre études majeures de ce domaine sont soumises à une analyse critique. Nous introduisons un nouveau type de grammaire qui permet une représentation numérique de la musique. De plus, nous décrivons un système informatisé pour la composition et la réalisation musicale rendue possible par cette grammaire. En appendice, nous présentons un guide d'utilisation du système.
# TABLE OF CONTENTS

## Volume I

**Preface** .................................................................................. v

**Chapter I:**
An Introduction to Formal Language Theory......... 1  
- Basic Concepts................................................................. 1  
- Rewriting Systems......................................................... 2  
- Generative Grammars.................................................... 3  
- The Chomsky Hierarchy.................................................. 5  
- Derivation Trees.............................................................. 7  
- Transformational Grammar............................................ 11  

**Chapter II:**
Grammatical Models in Computer Music Research... 13  
- Terry Winograd............................................................... 13  
- SCHENKER........................................................................ 25  
- Generative Grammar Definition Language................. 32  
- Structured Sound Synthesis Project............................. 36  
- Conclusion......................................................................... 40  

**Chapter III:**
T Grammars......................................................................... 42  
- S-events............................................................................. 42  
- Strings................................................................................ 45  
- T-events and T-trees......................................................... 50  
- AMA................................................................................... 60  
- Evaluation.......................................................................... 64  

**Bibliography**......................................................................... 67  

## Volume II

**Appendix:** AMA User's Manual
The metaphor of music-as-language is a commonly used theme in music theory. In recent decades it has acquired new meaning within the sphere of computer music, where the mathematical science of formal languages provides a ready-made set of models and a vocabulary for computer music research. In the present paper, this involvement of formal language theory and the computer in music composition, analysis, and synthesis will be examined and fostered.

Music theory permeated by linguistic science but with little or no connection to digital computing is certainly feasible, as the work of Lerdahl and Jackendoff attests. Concerning the role of computing in their research, they state: "In particular, we will not be concerned whether or not our theory can readily be converted into a computer program--one of the more fashionable criteria of formalization." By contrast, the present paper focuses exclusively on computer-oriented research. There is a practical reason for this: given the expansion of computer technology into virtually all areas of musical exploration, the need of a rational foundation for computer music software is today extremely acute.

---

The present paper takes a formal-language approach to music theory in order to avoid the potentially misleading consequences of any music–natural language analogy. Chapter I introduces fundamental concepts of formal language theory in an informal way. This material serves as a basis for the two independent essays of chapters II and III. Chapter II presents critical reviews of four major research projects in the field. These projects, by their diverse methodologies and goals, illustrate the broad range of theoretical issues to which formal language theory is applicable. By concentrating on these four projects rather than attempting a thorough review of the field, the chapter is able to probe certain issues to some depth. Chapter III develops an original grammar-type for representing music in digital systems for composition and performance. A computer system based upon the grammar is described. The system, known as "A Music Automaton" (AMA), has been fully implemented in the electronic music studios of McGill University's


3 For further information on formal language theory the reader is directed to the bibliography for texts by Harrison, Hopcroft and Ullman, and Salomaa, as well as the papers by Chomsky.

Faculty of Music. A user's manual for AMA forms an appendix to the present paper.

I wish to thank my adviser, Prof. B. Alphonce, for his patience and encouragement in preparing this thesis, and Prof. P. Pedersen for providing the opportunity to create AMA.
CHAPTER I

AN INTRODUCTION TO FORMAL LANGUAGE THEORY

This chapter is directed at two different kinds of readership. For those readers familiar with formal language theory, it establishes the technical vocabulary of this thesis: a necessary preliminary in a field where terminology is far from standardized. For those readers unfamiliar with the field, it introduces just those facets of the theory necessary for an understanding of the ensuing chapters. It is assumed that all readers have at least an intuitive notion of the mathematical concept of a set. Any concepts requiring a more extensive mathematical background have been relegated to the footnotes.

Basic Concepts

Formal language theory is concerned with certain finite sets of indivisible symbols called alphabets. Symbols from an alphabet, written one after the other, are called strings. To illustrate, if the alphabet $Y$ is the set $Y = \{ \circ, \mathcal{D} \}$, then the following are examples of strings over $Y$:

\begin{align*}
  w &= \circ \\
  x &= \mathcal{D} \mathcal{D}
\end{align*}

Figure 1-1 Two strings over the alphabet $Y = \{ \circ, \mathcal{D} \}$.

In formal language theory, it is convenient to refer to individual strings by variable names. The two strings of figure 1-1, for example, are associated with the variable names
w and x respectively. Define the operation of concatenation, whereby two strings are joined together to form a third. Then the concatenation of the strings w and x, written wx, is the string '0 d d'.

Clearly, the set of all possible strings over the alphabet Y contains an infinite number of elements. One special string, called the null string or I, is that string consisting of no symbols at all. The set of all possible strings over any fixed alphabet Z, including I, is denoted Z*. By a (formal) language over Z, we mean some subset of the set Z*.

Rewriting Systems

The concept of a rewriting system is predicated on these fundamental notions of alphabets and strings, for a rewriting system is a duple (V, P), where V is an alphabet and P is a finite set of ordered pairs of strings over V*. The ordered pairs (w, x) which make up the set P are called productions, and are customarily written in the form w \rightarrow x.

A production may be thought of as an instruction to rewrite the string on the left side of the arrow by that on the right. For example, if we are given the string 'Sue likes Bach', and the production 'Bach \rightarrow Mozart', then we have:

Sue likes Bach \rightarrow Sue likes Mozart.

---

1 We may define the set of all strings over an alphabet Z more formally as follows:

\[ Z^* = \{ w : (w=I) \text{ or } (w=xy, \text{ for } x \text{ in } Z \text{ and } y \text{ in } Z^*) \} \]
Note the distinction between the symbol '->', used for the specification of a production, and the symbol '=>', denoting the actual application of a production to a string in a rewriting process.

The notion of a derivation plays a critical role in formal language theory. If a string $x$ can be produced from a string $w$ by the application of one or more rewriting processes, then we write $w \Rightarrow x$. In case $x$ can be produced from $w$ by the application of zero or more productions, then we write $w \Rightarrow x$. The actual sequence of rewrites, $w \Rightarrow a \Rightarrow b \Rightarrow ... \Rightarrow x$, is termed the derivation of $x$ from $w$.

**Generative Grammars**

A generative grammar may be understood as an elaboration of the rewriting-system concept. Unlike rewriting systems, however, generative grammars make use of two disjoint alphabets called the variable alphabet $V$ and the terminal alphabet $T$. One special symbol $S$ from the variable alphabet is called the start symbol. More formally, a generative grammar is a quadruple $(V, T, P, S)$, where $V$, $T$, and $S$ are as described above, and where $P$ is a finite set of productions with the provision that the left side of each production in $P$ must contain at least one symbol from the variable alphabet $V$.

---

2 To understand the distinction between the meaning of "$\Rightarrow$" and "$\Rightarrow$", note that for a given rewriting system, the set of all strings $x$ such that $w \Rightarrow x$ may or may not include the string $w$ itself, whereas $w$ is necessarily included in the set of all strings $x$ such that $w \Rightarrow x$. 
Any string in a given grammar for which there is a derivation from $S$ is called a sentential form of the grammar. From this, the language $L(G)$ generated by a generative grammar $G$ is defined as the set of all sentential forms of $G$ consisting entirely of symbols from the terminal alphabet.\(^3\)

As the following figures demonstrate, generative grammars are capable of representing certain music-theoretic ideas in a highly comprehensive way.

$$B = (\{FBASS, C, D, E, F, G\}, \{c, d, e, f, g\}, P, FBASS)$$

with productions $P$:

1. $FBASS \rightarrow C \ G \ C$
2. $C \ G \rightarrow C \ E \ G$
3. $C \ E \rightarrow C \ D \ E$
4. $E \ G \rightarrow E \ F \ G$
5. $D \ E \rightarrow I$
6. $E \ F \rightarrow I$
7. $C \rightarrow c$
8. $D \rightarrow d$
9. $E \rightarrow e$
10. $F \rightarrow f$
11. $G \rightarrow g$

Figure 1-2 The grammar $B$. The symbol $I$ on the right side of productions 5 and 6 represents the null string. The productions have been numbered for reference.

Next, consider figure 1-3, showing a typical derivation of a string from $L(B)$, the language of this grammar $B$. Since the string 'cfgc' consists entirely of symbols from the terminal alphabet, and since $FBASS \Rightarrow cfgc$, it follows that 'cfgc' is in $L(B)$. In fact, $L(B)$ generates the pitch-name sequences of

\(^3\) In set builder notation, $L(G) = \{ x \mid x$ is in $T^*$ and $S \Rightarrow x \}$
FBASS => C G C \( (1) \)
=> C E G C \( (2) \)
=> C D E G C \( (3) \)
=> C D E F G C \( (4) \)
=> C F G C \( (5) \)
=> c F G C \( (7) \)
=> c f G C \( (10) \)
=> c f g C \( (11) \)
=> c f g c \( (7) \)

Figure 1-3 Derivation of a string according to the grammar B of figure 1-2. The numbers in the column to the right indicate which production of B has been applied to produce the rewriting.

all space-filling motions of a fundamental bass in a Schenkerian background graph.\(^4\)

The Chomsky Hierarchy

Generative grammars have been classified into four main types, with each type characterized by restrictions on the form of its productions. The resulting classification scheme, introduced by Noam Chomsky,\(^5\) is known as the Chomsky hierarchy.

A type 0 grammar imposes no restrictions whatever on the form of its productions, other than the restriction inherent in all generative grammars: that there be at least one non-null variable on the left of each production. These type 0 grammars are often called unrestricted grammars, while the languages they generate are termed the recursively-enumerable sets.

\(^4\) In C major only. For a listing of these motions in staff notation, see Heinrich Schenker, Free Composition, translated and edited by Ernst Oster, Supplement: musical examples, figure 14, (New York: Longman 1979).

Every production of a type 1 grammar can be written in the form \( xA\ldots z \rightarrow xyz \), where 1) \( A \) is a single symbol from the variable alphabet, 2) \( x, y, \) and \( z \) are arbitrary strings of variables and terminals, and 3) \( y \) is not the null string. In other words, we are allowed to rewrite the variable symbol \( A \) as the (non-null) string \( y \) only when it is wedged between the (possibly null) strings \( x \) and \( z \). For this reason, type 1 grammars are commonly called context-sensitive grammars. The languages they generate are termed the context-sensitive languages.

In a type 2 grammar, the left side of every production must be a single symbol from the variable alphabet. Such productions can always be written in the form \( A \rightarrow x \), where \( A \) and \( x \) are as defined above. Productions of type 2 grammars, unlike those of context-sensitive grammars, place no restrictions on the context in which the left-hand-side variable may be rewritten. For this reason, grammars of type 2 are called context-free grammars, while the languages they generate are called the context-free languages.

If, in a given grammar, all productions are of the form \( A \rightarrow xB \) or \( A \rightarrow x \), where \( A \) is a symbol from the variable alphabet and \( x \) is a (possibly null) string of terminals, then the grammar is said to be right-linear. Similarly, if all productions are of the form \( A \rightarrow Bx \) or \( A \rightarrow x \), then the grammar is left-linear. Right or left-linear grammars are called type 3 grammars, and the languages they generate are called the regular sets.
Any grammar of type 3 is also of type 2, for any production in a type 3 grammar qualifies equally well as a context-free production. In fact, the language classes of the Chomsky hierarchy form a chain of inclusion relationships, for the regular sets are a subset of the context-free languages, the context-free languages are a subset of the context-sensitive languages, and the context-sensitive languages are a subset of the recursively-enumerable sets.

Researchers proposing grammars for music are necessarily concerned with these notions of grammar types, for some language classes are inherently more tractable than others. Type 3 grammars seem too weak for most musical purposes, while type 0 grammars are clearly too powerful.6 The main theoretical issue is to determine what aspects of music require context-sensitive treatment as opposed to the more manageable context-free model.

Derivation Trees

Derivations in a grammar can be expressed in the form of tree graphs, provided the production set of the grammar rewrites but a single symbol per production. Such graphs, called derivation trees, make explicit the underlying structural description of strings in the language, and lead directly to the important definition of ambiguous grammars.

6 Virtually any language which we can think of is context-sensitive. Those languages known to be recursively enumerable but not context-sensitive are, in general, extremely abstract mathematical entities.
A graph (strictly speaking, a directed graph) consists of a set of points called nodes and a set of ordered pairs of nodes called arcs. Any arc \((x, y)\) is said to leave \(x\) and enter \(y\). To illustrate, the graph of figure 1-4 contains five nodes labelled \(A, B, C, D,\) and \(E\), together with the arcs \((A, B), (A, C), (C, D),\) and \((C, E)\).

![Figure 1-4 A directed graph.](image)

In a graph, any sequence of nodes joined by a chain of arcs is called a path. Referring to figure 1-4, there are paths from \(A\) to \(D\) and from \(A\) to \(E\), but no path from \(B\) to \(E\), since there is no chain of arcs between these two nodes. If there is a path from any node \(x\) to another node \(y\), then we say that \(x\) is an ancestor of \(y\) and \(y\) is a descendant of \(x\). If this path spans but a single arc, then \(x\) is an immediate ancestor of \(y\) and \(y\) is an immediate descendant of \(x\).

A tree (strictly speaking, an ordered, directed tree) is a particular form of directed graph. In any tree, there will always be one node, called the root, which no arc enters, and there will always be one and only one path from the root to any other given node. Any node from which no arc leaves is called a leaf node; all other nodes are called interior nodes. In figure 1-4, \(A\) is a root node, \(B, D,\) and \(E\) are leaf nodes, and \(A\) and \(C\) are interior nodes. Clearly, this graph is a tree, for
there is one and only one path from the root node A to each of the nodes B, C, D, and E.

We are now in a position to see how derivations in a given grammar may be expressed by tree graphs. The variables and terminals in a derivation become the labels of nodes in the tree, with the root node labelled by the start symbol of the grammar. If any node labelled x has immediate descendants y and z, then the production x → yz must be in the production set of the grammar. The following example elucidates these points by presenting a grammar for the major structural units of a "textbook" rondo.

$$G = ( \{ \text{RONDO}, X, Y, Z, C \}, \{ \text{theme}, \text{episode}, \text{coda} \}, P, \text{RONDO} )$$

with productions $P$:

- \text{RONDO} → Y Z C
- Z → X Y
- Z → Z X Y
- Y → \text{theme}
- X → Z X
- X → \text{episode}
- C → \text{coda}
- C → I

Figure 1-5 A rondo grammar.

This grammar uses the symbol RONDO as its start symbol. The last production, C → I, rewrites C as the null string, allowing the generation of rondos without codas. Figure 1-6 presents one possible derivation in the language, together with its corresponding derivation tree.

The string "theme episode theme episode theme coda", consisting entirely of terminal symbols, labels the leaf nodes
RONDO => Y Z C
  => theme Z C
  => theme Z X Y C
  => theme X Y X Y C
  => theme episode Y X Y C
  => theme episode theme X Y C
  => theme episode theme episode Y C
  => theme episode theme episode theme C
  => theme episode theme episode theme coda

Figure 1-6 A derivation and its associated derivation tree in the grammar of figure 1-5.

RONDO

R

O

D

O


Y

Z

C

theme

Z

X

Y

coda

X

Y

episode

theme

episode

theme

theme

episode

theme

Coda

Figure 1-7 An alternate derivation tree for the string derived in figure 1-6.
of the derivation tree in figure 1-6. In figure 1-7, this same terminal string is generated again from the same grammar, but in this case the associated derivation tree has a different form from that of figure 1-6.

Figures 1-6 and 1-7 illustrate the fact that the same string may have more than one derivation tree for a given grammar. A context-free grammar is said to be ambiguous if and only if it generates at least one string with two different derivation trees.

Transformational Grammar

Researchers have proposed and developed many other language-generating formalisms in addition to the grammars of the Chomsky hierarchy. Although the languages generated by these formalisms can always be produced by at least one of the grammars of types 0 to 3, the structural descriptions they assign to particular strings can vary widely. Of all such formalisms, none has been as prevalent, particularly in the area of theoretical linguistics, as that of transformational grammar.

Though the details of transformational grammar need not concern us here, a brief overview of its mechanics will be needed in order to understand some of the material of the next chapter. Transformational grammar can be understood as

---

consisting of two components. First, a base component, most often a context-free grammar, generates structural descriptions, or deep structures, of intermediate representations of strings. Second, a transformational component consisting of a set of transformational rules acts on the deep structures of the base component to produce surface structures. Briefly stated, these transformational rules rearrange and transform trees to produce new trees.

Despite the pervasiveness of transformational grammar in theoretical linguistics, applications of the formalism, when applied to music, have been received with contention. To quote from Curtis Roads:

No clear musical analogy between these natural language constructs [active-passive verb relations, auxiliary verbs, etc.] and music exists. Thus there is little justification for importing an obligatory transformation level (which would modify the effect of the production rules) into a music grammar.

Nevertheless, as witnessed by chapter II of the present paper, transformational grammars have played a significant role in computer systems for music analysis and composition.


CHAPTER II
GRAMMATICAL MODELS IN COMPUTER MUSIC RESEARCH

This chapter will review four research projects in which grammatical models and the computer were used to explore important topics in music theory. These particular projects have been chosen for the diversity of goals they exhibit: the first two treat the analysis of tonal music from very different perspectives, the next is concerned with composition, while the last presents a hierarchical data structure for the computer representation of musical scores. The variety of theoretical issues emerging from these reviews will attest to the wealth and power of the grammatical model in computer music research.

Terry Winograd

By 1968, the American artificial intelligence researcher Terry Winograd had implemented a computer program capable of doing harmonic analysis of tonal music.¹ The music-theoretic model embodied therein was totally unoriginal. In fact, Winograd carefully avoided the creation of new music theory. He took instead an existing theory of harmony, that of Allen Forte's Tonal Harmony in Concept and Practice², and built a program which embodied a mapping of Forte's teachings into a linguistically derived model.

Winograd's modus operandi was in many ways typical of artificial intelligence work of the time. The researcher strives to manifest machine intelligence by mimicking a process unique to thinking men and women. The results of this process can be judged alongside those of the human specialist, so researchers need never be in doubt as to the success or failure of their experiments. They seek to reduce intelligent human activity to the form of computer algorithms without unduly modifying or expanding the existing theory supporting that activity. Within its own well-defined limits, it must be conceded that Winograd's experiment succeeded admirably.

The grammatical model employed therein freely adapted the systemic grammar schemes of the British researcher M. A. K. Halliday and his colleagues. Winograd's program accepted a suitably encoded musical work as its input and, with the help of a systemic grammar, produced a harmonic analysis of that work as its output. His program, then, used systemic grammar purely as an analytic tool. Nevertheless, it is instructive to turn the process around and examine how this grammar might function as a generative device.

---


4 Generation in a systemic grammar has not been defined with anything like the formal rigour of the generative grammars described in chapter I. The following procedure is specifically designed for Winograd's grammar of tonal harmony, and is not intended to serve as a model for systemic generation in general.
This process begins with the choosing of a set of symbols called 'features' in accordance with a graph called a 'system network'. Figure 2-1, adapted from Winograd's article,5 shows one such network, with 'features' (composition, major, minor, CN, ..., BN) appearing as individual symbols without underlining. The underlined symbols 'mode' and 'root' do not represent features; they are instead convenient labels identifying subsystems of the network as a whole. The presence of a curly bracket instructs us to choose features from all subsystems to the right of the bracket. Square brackets, on

\[
\begin{align*}
\text{Composition} & \left\{ \begin{array}{c}
\text{mode} \\
\text{minor} \\
\text{major}
\end{array} \right. \\
\text{root} & \left\{ \begin{array}{c}
\text{CN} \\
\text{CS} \\
\text{DF} \\
\text{BN}
\end{array} \right.
\end{align*}
\]

Realizations:
Composition: \( +T^*(T)\ldots^*(T) \)
(Features from the mode and root systems are realized through Winograd's \#K\# system which will not be discussed here.)

Constituents:
T:: Tonality(simple, I)

Figure 2-1. A system network. In the notation of systemic grammar, '+A' indicates that a constituent associated with the function A must be included in the structure, '(A)' signifies the optional inclusion of the constituent A, and '+' signifies concatenation.

5 Terry Winograd, "Linguistics and the Computer Analysis of Tonal Harmony", p. 14. In both the figures 2-1 and 2-2, Winograd's networks have been modified to render the following discussion more lucid. These changes involve the expansion of certain abbreviations, as well as the omission of details which will not be dealt with in the present paper. No changes to the meaning of the network are brought about by this process.
the other hand, instruct us to choose but one feature from among the alternatives to the right. Thus, in figure 2-1, the curly bracket to the right of the feature 'Composition' instructs us to choose features from both the mode and root systems. In the mode system, the square bracket indicates that one of the features 'major' or 'minor', but not both, must be chosen. Similarly, a single feature must be chosen from among the alternatives presented by the root system. The end result of this process will be a set of features which, in the case of figure 2-1, might be \{ Composition, CN, minor \}, or perhaps \{ Composition, EF, major \}, to illustrate but two of the many possibilities.

The second step in the generative process uses realization rules to map the set of features onto a string of symbols called functions. To put it another way, this (ordered) string of functions realizes the (unordered) set of features. In figure 2-1, for example, the realization rule for the feature 'Composition' specifies that the string of functions must include the function T, followed by an arbitrary number of

6 Normally, in generation according to a systemic grammar, such functions form an unordered set. At the next step of generation, a set of objects called 'constituents' are chosen which act to fulfill these functions, then the constituents themselves are ordered as specified by other realization rules. With this scheme, it is possible for the same constituent to fulfill more than one function. In Winograd's systemic grammar of tonal harmony, however, such a situation does not arise: the mapping of functional units onto constituents is always either one-to-one or one-to-many. For this reason, the generation algorithm presented here imposes ordering before the functions are mapped onto constituents.
other such functions T. Note that there are no realization rules for features from the mode and root systems. In Winograd's systemic grammar formulation, these features are realized by a separate process called the 'K' system which, for reasons of brevity, will not be discussed here.

The string of functions which realizes the set of features is not of itself the end product of the generation process, for each function in the function string must be mapped onto symbols called constituents to produce a constituent string. In the network of figure 2-1, every function T is mapped onto the constituent 'tonality(simple, I)'.

The generation process just described, then, is a three-step procedure. First, a set of features is chosen in accordance with a system network. Second, this set of features is mapped onto a string of functions by realization rules. Finally, further rules map the string of functions onto a string of constituents. Like the first step, the mappings of the second and third step are nondeterministic: they may involve the choosing of symbols from a set of alternatives.

Obviously, a constituent such as 'tonality(simple, I)' does not itself represent the encoding of a musical surface. Instead, it effects the repetition of the three-step generative procedure just described, this time using the system network of figure 2-2. Not the network in its entirety, however, for the constituent 'tonality(simple, I)' specifies that the set of features chosen from figure 2-2 must include the features
Realizations:

Tonality :: +dominant; +(sec)...(sec); (sec)^dominant
complete :: +tonic; dominant^tonic
simple :: tonic^#
modulating:: +sec2^(sec2)^...^(sec2); tonic^sec2
implied :: +domprep; domprep^dominant;
dominant^Chord Group Direct
relative root, mode root :: The realizations of the relative root system act through the connections with the mode and root systems to produce #K#.

Constituents:
dominant :: Tonality(simple,V) or Chord Group(V or VII)
sec :: Tonality or Chord Group
tonic :: Chord Group(I, direct)
sec2 :: Tonality(simple or implied) or Chord Group
domprep :: Chord Group(II or IV or VI)

Figure 2-2. A system network for the feature 'Tonality'. The symbol '^' is the boundary symbol: it indicates the beginning or end of a string.
'simple' and 'I'. Other than this constraint, the generation procedure is repeated as before.

A constituent string resulting from such a repetition might well include, for example, the constituents 'Chord group(I, direct)' and 'Tonality(simple, V)'. In the first case, another network for 'Chord group' would be employed, with the constraint that the features 'I' and 'direct' must be chosen. In the second case, the network of figure 2-2 would itself be employed again, this time with the constraint that the features 'simple' and 'dominant' be chosen. Eventually, the successive repetition of the process through these and other system networks results in the generation of an encoded representation of a musical work.

The following comparison of the generative capacity of this grammar with the generative formalisms described in chapter 1 will help to demonstrate its relationship to the more thoroughly understood categories of the mathematical theory of languages. Such a comparison immediately presents itself, for it is evident that the system networks of figures 2-1 and 2-2 are readily translatable into context-free grammars. The network of figure 2-1, for example, could be represented as shown in figure 2-3. The sole difference between the generative capacity of this representation and that of figure 2-1 is that the generative version produces an ordered string of symbols, whereas the system network generates these same symbols (features) as an unordered set.
G = \{ S, \text{comp}, \text{mode}, \text{root} \},
\{ \text{Composition, major, minor, CN, CS, ..., BN} \}, P, S

with productions:

S \rightarrow \text{Composition comp}
\text{comp} \rightarrow \text{root mode}
\text{root} \rightarrow \text{CN ; CS ; DF ; ... ; BN}
\text{mode} \rightarrow \text{major ; minor}

Figure 2-3 A context-free grammar representing the system network of figure 2-1. The vertical bar, '\|', is a shorthand convention representing 'or'. Thus the production 'mode \rightarrow major ; minor' is equivalent to the two productions 'mode \rightarrow major' and 'mode \rightarrow minor'.

Continuing the translation process, it is evident from figure 2-1 that the rules mapping features onto functions as well as those mapping functions onto constituents are readily translatable into generative grammars.

1. G(Composition) = \{ \{ \text{Composition} \}, \{ T \}, P, \text{Composition} \}
   with productions:
   \text{Composition} \rightarrow \text{Composition T ; T}

2. G(T) = \{ \{ T \}, \{ \text{Tonality(simple, I)} \}, P, T \}
   with productions:
   \text{T} \rightarrow \text{Tonality(simple, I)}

Figure 2-4. Context-free grammars representing the realization rules for functions and constituents in figure 2-1. The symbol 'Tonality(simple, I)' in 2. above is an indivisible symbol.

The grammar G of figure 2-3, as well as the grammars G(Composition) and G(T) of figure 2-4 can all be linked together to form a single generative device by a simple procedure. First, generate a string w in L(G), then rewrite every instance of the symbol 'Composition' in w with a string from G(Composition) to produce another string x. Finally,
replace every instance of the symbol 'T' in \( x \) with a string from \( L(T) \).

To put it another way, the three grammars may be joined together into a kind of 'grammar of grammars' through a process which maps the symbols \( Y \) in the strings of one language onto strings from another language \( L(Y) \). Such a mapping has been defined as 'substitution' in formal language theory. Furthermore, it is known that all families of languages in the Chomsky hierarchy are closed under substitution in the following sense. If \( L \) is a language of type \( i \), \( i = 0,1,2,3 \), and if for every symbol of \( L \) there is associated a language of type \( i \), then the resulting language is also of type \( i \). Since the grammars of figures 2-3 and 2-4 are all context-free, it follows that the language generated by a substitution mapping of \( L(G) \) onto \( L(Composition) \), then \( L(Composition) \) onto \( L(T) \), is also context-free. In other words, there exists at least one context-free grammar which generates precisely the same language as the system network of figure 2-1.

Why, then, utilize a systemic grammar scheme for the analysis of harmony? Is there some way in which it is more efficient, more revealing of structure than the equivalent generative grammar formulation? To answer this question, consider the structural description assigned a given terminal string by both the systemic representation and the equivalent generative grammar. For in any grammatically-based music analysis scheme, it is this structural description, and not the grammar's generative capacity, which is ultimately of interest.
Any useful analytic statement based on a grammatical model will take the form of a string captured between the grammar's start symbol and the generated terminal strings encoding the musical surface. Such intermediate representations, hereafter called intermediate sentential forms, will consist of at least one variable with perhaps some terminals. More formally, a string $X$ is an intermediate sentential of $G$ if and only if, for some grammar $G$,

$$S \Rightarrow X \Rightarrow w, w \text{ is in } L(G)$$

An intermediate sentential form, then, is any string derivable from $S$ which contains at least one variable symbol and which derives a string of terminals. Now, for any reasonably complex grammar used to analyse a given musical representation, the number of intermediate sentential forms will be enormous, and many of them will have little or no analytic significance whatsoever. Confronted with the derivation of a musical work produced by a generative grammar, the researcher must discard by hand the vast majority of these strings to uncover those forms which are truly revealing. As Winograd notes, most theories fail to ask "Which features of a syntactic structure are important to conveying meaning, and which are just a by-product of the symbol manipulations needed to produce the right word order."\(^7\) Winograd's systemic grammar, however, is designed so that analytically significant

intermediate sentential forms are automatically distinguished—they emerge as the terminal strings of each subgrammar. Thus the systemic grammar cuts the equivalent generative grammar into a number of subgrammars at precisely those points judged to be significant in Forte's harmonic theory.

The systemic grammar scheme has yet another important advantage. Consider the problem of completing the translation, begun earlier, of Winograd's systemic formulation into substitution-mapped generative grammars. Though it would not be difficult to translate the system network of figure 2-2 into a generative grammar \( G(\text{Tonality}) \), this is not exactly what is needed. Rather, we require a grammar \( G(\text{Tonality(simple, I)} \) in order to define a substitution for the symbol 'Tonality(simple, I)''. Furthermore, the grammars \( G(\text{Tonality(simple, V)} \) and \( G(\text{Tonality(simple)} \) would have to be defined. In fact, completing the translation would require defining separate grammars for almost all of the features and constituents of Winograd's grammar. The resulting generative construct would not only be many times larger than the equivalent systemic grammar, it would blur the regularities of the tonal system so elegantly summarized by the system networks.

There are many other fascinating aspects of Winograd's ground-breaking research which could be investigated. His scheme for musical representation, his treatment of non-harmonic tones as linear chords, and his semantic-directed parsing algorithm come immediately to mind. But perhaps the most interesting topic from the music-theoretic point of view
is his manner of dealing with the inherent ambiguity of the tonal language.

In formal language theory, ambiguity appears when for a given grammar, some word has more than one derivation tree. The equivalent situation arises in musical analysis when two or more readings present themselves as plausible explanations for the same passage. Further criteria may be invoked by the theoretician to select an allegedly best reading from among these alternatives. Such criteria may be diverse and complex, possibly extending into such ill-defined regions as the theoretician's experience and personal taste. Despite these difficulties, any computer analysis scheme based on an ambiguous grammar and claiming to produce more than merely plausible structural descriptions must provide some mechanism for choosing from among alternative readings.

Winograd's scheme functions by assigning small integers, called 'plausibility values', to harmonic progressions. To illustrate, the most plausible tonal progression, that of the authentic cadence, is given the value 0. Less plausible progressions receive successively greater integers. The decision mechanism for choosing from among several derivation trees for the same passage, then, is a simple matter of choosing the one whose plausibility values sum to the smallest integer.

Winograd states that these values could have been incorporated into the grammar itself. This indicates that,
given the inclusion of such rules in the grammar, the resulting unambiguous system could assign but one structural description to each work examined. Since different listeners hear music in different ways, however, a grammar capable of accounting for such diversity must either be ambiguous or provide for each listener type a unique analysis. For this reason, an ambiguous grammar may well provide more insight into tonal music than an unambiguous one.

SCHENKER

The analytical approach underlying Stephen Smoliar's program SCHENKER ⁸ appears, at first glance, to be very similar to that underlying Winograd's research. Both projects attempted the computer implementation of preexistent music theory—Schenker on the one hand, Forte on the other. Both chose linguistic theory as their fundamental operating model, and both chose LISP as the language of implementation. A more critical examination, however, reveals important differences between the two projects. A study of these differences will reveal some of the difficulties inherent in attempts to apply linguistic theory to the analysis of music.

In the first section of this chapter, the primary focus of

---

Winograd's research was identified as a demonstration of machine intelligence through the automation of a particular music-analytic task. Nothing could be farther from the goal of the SCHENKER project. As Smoliar points out in the conclusion to his article, "A Computer Aid for Schenkerian Analysis":

This has not been a discussion of the automation of musical analysis. Rather, the focus has been on the establishment of a rigorous framework for the discussion of a particularly [sic] analytic technique ... To paraphrase Salzer, the computer system discussed herein establishes a grammar for Schenkerian analysis...9

The following scenario showing Smoliar's program at work on a given analytical task will help to reveal how it functions. The analyst sits at a terminal, the work to be analysed at hand. By the successive entering of a series of commands, a computer representation of a Schenkerian graph for the work is gradually built up.10 Most of these commands are abbreviations denoting compositional prolongation techniques. They expand and mold the computer representation in ways which mimic not only those Schenkerian concepts on which they are based, but also the transformational rules of transformational grammar. Should the analyst judge at any point in the process that the sequence of commands so far executed is straying from the path towards a successful analysis of the work at hand, a


10 For an explanation of the Schenkerian concepts used in this article, see Allen Forte and Steven E. Gilbert, Introduction to Schenkerian Analysis (New York: W.W. Norton and Company, 1982).
backing up process may be evoked, undoing the effect of previous commands.

A major contrast with Winograd's work immediately suggests itself. With Winograd, preexisting computer-coded musical surfaces are automatically broken down in accordance with a fixed grammar to produce intermediate sentential forms representing analytical statements about that surface. As virtually the direct antithesis of this strategy, Smoliar's system invites the human analyst to use the computer as an aid in finding a sequence of commands which transform Schenker's 'chord of nature' into a given intermediate sentential form, though again in accordance with a fixed grammar.

What, then, was the motivation for Smoliar's investigation? Ultimately, nothing is created which did not already exist, for that which is analytically relevant in a derivation tree, the intermediate sentential form, must already exist, at least in the analyst's mind, in order for the process to function. If the goal of Smoliar's research was to establish a Schenkerian analytical grammar, then one possible motivation for SCHENKER's creation was to establish a rigorous testing ground for the transformational rules themselves. These rules, then, become the postulates of the system. That they are capable of generating strings which encode Schenkerian graphs ensures their validity—they may be incorporated into systems of analysis with full, machine-certified confidence of their validity.
The above argument, however, is illusory, and serves to illustrate one of the methodological problems inherent in applying grammars to music theory. For in order to show that SCHENKER's rules constitute a grammar $G$ for Schenkerian graphs, it must be shown that the language they produce, $L(G)$, is exactly the language, $L(S)$, of Schenkerian graphs. Schenkerian theory, however, has not been formulated with anything like the mathematical rigour necessary for such a demonstration.

Furthermore, it seems doubtful that the mapping between Smolar's encoding scheme and Schenkerian graphs is particularly close. The following bracketed string, for example, is given by Smolar as a representation of a Schenkerian third-line:

```
(SIM
  (SEQ (C -1) (G -1) (C -1))
  (SEQ (E 0) (SEQ (D 0)) (C 0))
)
```

Figure 2-5. A Schenkerian third-line in the bracketed-string notation of SCHENKER, together with the tree structure it represents. In the bracketed-string notation, all strings consist of an 'event' surrounded by brackets, where an event is one of:

1) A pitch name followed by an integer representing octave.
2) The word SIM followed by a list of simultaneously beginning events.
3) The word SEQ followed by a list of sequentially occurring events.
This representation, due to its inability to specify anything other than the immediate simultaneity of the beginnings of sequential events, could be interpreted in any of the following ways:

![Figure 2-6 Alternative interpretations of the bracketed string of figure 2-5.](image)

To claim that figure 2-5 corresponds to the first of these alternatives requires making an assumption about the simultaneity of the G and D, as of the C octaves. This information, without which the graph would make no sense, is totally absent from the bracketed-string representation of figure 2-5.

Among the many obstacles to be surmounted in any approach to analysis using grammars is the establishment of a representation scheme realizing the two-dimensional, horizontal-vertical character of multi-part music. Unfortunately, linguistic theory, as it has been applied to both formal and natural languages, concerns itself with the essentially unidimensional, horizontal concatenations of symbols into strings. This is not to say that coding functions mapping multidimensional constructs into linear symbol streams cannot be found. Such functions must, however, be powerful enough to ensure that their application does not result in a
significant loss of important information. In particular, any encoding system should allow the unambiguous reconstruction of that notation it represents. And, as demonstrated above, it is precisely in this respect that Smoliar's representation scheme fails.

One further facet of Smoliar's work merits attention—the nature of his grammatical rules themselves. In the Winograd project, grammatical transformation rules were rejected in favour of a systemic grammar approach. By contrast, the computer commands of Smoliar's system are entirely transformational: instead of modifying a musical score representation by rewriting rules, they modify tree structures themselves, pruning and grafting subtrees according to preset formulas.

Transformation rules, when applied to natural languages, reveal the similarity of meaning between such sentences as "Sue played Bach" and "Bach was played by Sue". Though both sentences could be generated by a simple context-free grammar, the associated derivation trees would be very dissimilar, even though the sentences mean the same thing. In transformational grammar, by contrast, both sentences would be generated by the same derivation tree in the base component; then, a transformational rule would transform the first sentence into its passive version.

It should be pointed out, however, that Smoliar's tree structures, as exemplified by figure 2-5, are very unusual
objects. They are not derivation trees like those generated by the base component of a transformational grammar according to some context-free grammar. Instead, such trees simply encode Schenkerian analytic notation. When transformation rules are applied to such objects, it is no longer clear what, if anything, is preserved by the transformation process.

Since Smoliar's grammar represents Schenkerian prolongational techniques entirely as transformations, it has no base component, context-free or otherwise. As a consequence, the grammar has no start symbol. In its place, a transformational rule called NATURE generates an initial tree. According to Smoliar, "It is not strictly a transformation; instead, it constructs a new tree 'from scratch'." By contrast, the start symbol of a generative grammar, with its connotation of fundamental axiom, is much closer to the Schenkerian view of the origin of the chord of nature.

To conclude, Smoliar chose to formulate a grammar for Schenkerian analysis using techniques at best poorly understood in both the mathematical theory of languages and linguistic science. Because of the extreme originality of his approach, it is very difficult to evaluate just what the project was attempting to accomplish, and whether or not it succeeded in that attempt. Before such questions can be answered, much more will have to be known about the grammatical processes which underlie SCHENKER.

Generative Grammar Definition Language

The ensemble of essays by S. R. Holtzman\textsuperscript{12,13,14} describe the mechanics and philosophical underpinnings of his unique, grammatically-based music automata. Like Smoliar, Holtzman adopts a purely generative scheme with no provision whatever for the parsing of musical representations. Also like Smoliar, Holtzman offers his program as a musicological tool for the development and verification of grammatical production hypotheses aimed at explicating repertoire. Here the similarity ends, however, for the primary focus of Holtzman’s Generative Grammar Definition Language (GGDL) is composition.

GGDL is a formal language expressly designed for the specification of musical grammars. Any utterance of GGDL will be a list of specially formatted production rules which, when passed to Holtzman’s GGDL compiler, will elicit the automatic creation of a string (composition) belonging to the language of that utterance.

The scope of GGDL includes, among other concepts, type 0 and transformational rules. Productions using such rules are processed by the GGDL compiler in a way which closely parallels


generation in transformational grammar. The only significant difference between the two is that in transformational grammar, that grammar which generates the base component is usually context-free. In GGDL, by contrast, the full generative capacity of type-0 rewriting is provided.

It should not be assumed, however, that this extra generative capacity is necessarily a good thing. A good deal of linguistic research has been directed at the opposite goal: to identify the most restricted grammar type which is capable of generating a given language. Nevertheless, nothing in GGDL obliges the user to employ the extra generative capacity. Holtzman's approach has been to provide the most general generative system possible, leaving the composer the freedom to decide what generative devices to use in the creation of a particular work.

Holtzman has described several of his own GGDL compositions in his writings, some of the most interesting of which may well be those which are the least original. In one experiment, after subjecting the trio of Schoenberg's 1925 Suite für Klavier, op. 25, to a manual analysis, he formulates, in GGDL, a description of its pitch and rhythmic structure.\textsuperscript{15} The resulting grammar generates an entire class of compositions, of which the Schoenberg is but one instance. Holtzman offers two further instances as examples.

\textsuperscript{15} S. K. Holtzman, "Using Generative Grammars for Music Composition", pp.53-56.
From the standpoint of analytic theory, this process of generating 'grammatical variations' merits attention. The ability to systematically produce variations of musical works according to a given grammar could serve useful analytic ends for, in many cases, that which the composer did not write may reveal something about that which he did. It is important to remember, however, that this notion of variation is by nature ill-defined. To illustrate, consider the following type-3 grammar:

\[ G(0) = ( \{ S \}, \{ \text{note} \}, \{ S \rightarrow S \text{ note } \text{ note } \}, S ) \]

Obviously, the language generated by \( G(0) \) includes the pitch-rhythm structure of the Schoenberg trio together with its variations. In addition, we may number all of Beethoven's symphonies, not to mention the entire Burmese saung kauk harp repertoire among its generations. Though all these works are generated by the same grammar, they are certainly not all variations of Schoenberg's trio. \( G(0) \) is simply too powerful a generative device to say anything significant, from the music-theoretic point of view, about similarities among the variations it generates.

Now consider the following grammar:

\[ G(1) = ( \{ S \}, Y, \{ S \rightarrow Z \}, S ) \]

where \( Z \) is in \( Y^* \), and \( Z \) is any coding of Schoenberg's trio. Since this grammar generates one and only one composition, it admits of no variations whatever. Now, to show that two
compositions were variations of each other, we might try to find a grammar which generated them both and was at once less restrictive than $G(1)$ and more restrictive than $G(0)$. Just exactly how restrictive this grammar should be, however, cannot be established in any formal way. The same methodological problem arises if all compositions belonging to some given musical style are considered as variations of each other. Seen in this light, it must be concluded that musical style cannot be defined on the basis of a generative grammar alone.

The man-machine interaction implicit in Holtzman's GGDL compositional scheme acts to produce what might best be called computer-aided composition. The role of the composer, in Holtzman's view, has been altered in the process:

The computer is seen as a tool and may be used as a composer wishes to achieve his or her compositional aims. In any case, any grammar that a composer may define is likely to generate more than one composition: a grammar defines a language in which there may be many utterances. Ultimately, a composer must choose which generated utterances to use, how to interpret the data generated by the machine, and so on. The composer may be seen as a selector.\footnote{S. R. Holtzman, "Using Generative Grammars for Music Composition", p.62.}

This characterization of the composer as selector may seem unnecessarily narrow in light of his or her role in defining the grammar which generates these selections. On the other hand, the set of compositions defined by this grammar represents of itself a selection from among the infinite set of all possible sonic concatenations. It is, however, a selection
of a unique kind, for its selection criteria, stated in terms of rewriting and transformational rules, are precise, predictable, and communicable.

Computer-aided composition within the Holtzman model, then, involves the composer in two types of selection processes. First, a set of grammatical descriptions is written which, with the aid of a computer, generates a set of compositions. Second, the composer makes a selection from among these compositional alternatives on the basis of musical intuition and experience. The situation parallels Winograd's systemic grammar-plausibility value scheme, and it serves to illustrate an important point about grammatically driven systems in general: mere grammaticality is not the only criterion for musical value. Despite the power and elegance of the linguistic model, significant intangibles remain unexplained. In the opinion of Holtzman, "Some music just sounds better!".17

Structured Sound Synthesis Project

The rest of this chapter focuses on a computer-music research project of a different nature: the work of the University-of-Toronto-based Structured Sound Synthesis Project (SSSP) and the context-free grammar embedded in the heart of their highly-developed digital music system.

In "The Use of Hierarchy and Instance in a Data Structure for Computer Music", the motivating factor behind the SSSP's grammatical score representation scheme is presented as follows:

In examining the literature it can be seen that most systems to date have gravitated towards one of two extremes: those which dealt with the score from a note-by-note approach (e.g., Vercoe, 1975), and those which dealt with the score as a single entity (e.g., Xenakis, 1971). It is obvious, however, that structures falling somewhere between the "note" and "score" level play an important musical role.

The SSSP needed a computer data structure which could represent not only the "note" and "score" levels of music, but a wide range of intermediate levels as well. For if a score is represented purely as a single entity, manipulating its subsections becomes difficult, if not impossible. These same manipulations are just as difficult when the score is represented purely as a sequence of notes. Tree-like structures, however, because of their implicit hierarchical organizations, greatly facilitate these manipulations.

In chapter I of the present paper, derivation trees were introduced as a vivid way of displaying derivations in a grammar. This natural correspondence between grammars and trees suggests the possibility of using grammars to describe the set of shapes tree-like data structures may take on. Hence

19 ibid., pp.11.
it is not surprising to find the SSSP using a grammar to describe their hierarchical score-representation scheme:

\[
\begin{align*}
\text{Composition} & ::= \text{Mevent} \\
\text{Mevent} & ::= \text{Mevent} \mid \text{Score} \mid \text{note} \\
\text{Score} & ::= \text{Mevent} \\
\text{note} & ::= \text{terminal}(\text{i.e., some musical note})
\end{align*}
\]

Figure 2-7. A BNF grammar for the SSSP's notion of a musical event. BNF is a notation system for context-free grammars often used in computer science. The symbol '::=' is equivalent to the arrow '->' in the notation system of the present paper.

This grammar emphasizes the notion of a composition as a musical event made out of sub-scores. As a didactic device, it serves well, but it should not be taken as a description of the data structure of the SSSP's computer music system. The variable 'Composition', for example, has no counterpart in that data structure, which is instead rooted in 'Score'. Perhaps the above grammar generates the same language as their data structure, but then so does the following:

\[
\begin{align*}
\text{Composition} & ::= \text{Composition} \ast \mid \text{terminal}
\end{align*}
\]

In both these grammars, however, those shapes which derivation trees may take on are not the same as those of the data structure described in the article. Since this data structure determines what hierarchies between the "note" and "score" levels may be represented, we will attempt to formulate a new grammar by carefully examining that data structure.

As previously pointed out, the highest point in the SSSP's hierarchical data structure is the object named 'Score'--this
will become the start symbol of our grammar. This 'Score'
object is connected to a list of arbitrary length whose objects
are each named 'Mevent':

\[
\text{Score} \rightarrow \text{Mevent}^* 
\]

It turns out, however, that Mevent objects come in two
forms. First, an Mevent of type Mnote is connected to such
(terminal) variables as volume, frequency, timbre definition,
channel number, and duration. Second, an Mevent of type Score,
called an Mscore, is connected, through an intermediate symbol
table, to further score objects. It is even possible to
dispense with the Mevent symbol itself, for there exist no
instances of Mevents which are neither Mscores nor Mnotes.
This leaves us with the following grammar:

\[
G(\text{SSSP}) = ( \{ \text{Score, Mscore, Mnote} \}, \\
\{ \text{volume, frequency, timbre, channel, duration} \}, \\
P, \text{Score} ) 
\]

with productions:

\[
\begin{align*}
\text{Score} & \rightarrow (\text{Mscore} \mid \text{Mnote})^* \\
\text{Mscore} & \rightarrow \text{Score} \\
\text{Mnote} & \rightarrow \text{volume frequency timbre channel duration}
\end{align*}
\]

Figure 2-8 A grammar for the SSSP's hierarchical data
structure for score representation.

From this, it can be seen how structures falling between the
"note" and "score" levels can be represented.

It should not be assumed that the paucity of variables in

---

20 Technically, this connection is accomplished in the
computer's memory by a pointer to a linked list structure.
G(SSSP) represents any lack of capability in the representation of complex hierarchies. In reality, each instance of the symbol 'Mscore' in a derivation may stand for a different object, for Mscore is of itself a compound structure uniting a group of variables. Its status in the grammar, then, is not as a single variable, but as an entire class of such symbols. It determines, on the basis of the values of these variables, the interpretation of the pitch, volume and rhythm of all Mnotes for which it is an ancestor in some derivation tree.

As a consequence of their score-representation scheme, the SSSP's scores are not simply one-dimensional concatenations of musical notes, but entire structural descriptions of musical works. The grammar G(SSSP) of figure 2-8, in describing the forms these structural descriptions may take, reveals the SSSP's achievement in a concise and elegant manner.

Conclusion

Throughout this chapter, a number of important themes have been found threaded through all of the projects under consideration. Diverse data-representation schemes have been examined, some more successful than others. The fallacy of conferring legitimacy on grammars on the basis of their ability to generate instances of a given language has been discussed. The notion of intermediate sentential form as analytic statement has been developed. Grammatical ambiguity has been found to mirror the diversity of musical analytic opinion. Concepts drawn from transformational grammar have been found in
applications far removed from natural-language description.

Clearly, grammatical models have played a highly significant role in computer music research. This chapter has demonstrated that formal language theory can be a powerful tool for the description and evaluation of that research.
CHAPTER III
T GRAMMARS

This chapter presents an original approach to the organization of musical material in the electro-acoustic medium. It introduces a type of grammar, T, dedicated to a single task of computer music: given a set of fundamental compositional units called S-events, T grammars can explicitly determine the temporal organization of these S-events within a musical work. The concept's practicality will be shown by introducing "A Music Automaton" (AMA): a computer system for music composition and performance based on a T grammar. A user's manual for AMA is provided as an appendix to this paper.

S-events

The theoretical concepts presented in this chapter will parallel the notion of a grammar as a quadruple \((V,T,P,S)\). In applying this formalism to music composition, the grammar's terminal alphabet will be associated with certain fundamental, atomic units of composition. Deciding just what sort of compositional event these fundamental units should be, however, is by no means a trivial undertaking, for any such decision will reflect the musical task for which the grammar is designed. The units could, for example, be integers representing air pressure oscillations. Now, although such a

---

1 See chapter I of the present paper for an explanation of terms drawn from formal language theory.
scheme might be a useful approach to timbral design, the
abundance of low-level acoustic detail it engenders would
unnecessarily complicate any grammar designed for composition.

In the words of Otto Laske:

We say that sounds are produced according to acoustic
models. We may equally say that compositions are produced
according to compositional models... While in sound
synthesis, we can speak of a 'lowest level' material--such as
samples--in composition this is not so easy, since defining
what is the lowest level in a composition is an integral part
of the compositional responsibility.2

In the traditional musical score, that musical event
which most often fills the role of fundamental unit is surely
the note:

... the smallest significant musical element, ... Western
musicians acknowledge that it is the note. But what is a
note?3

Now, the question 'what is a note' cannot be resolved
without reference to a particular acoustic model. Indeed, any
response can best be formed in terms of the sound-producing
apparatus of traditional acoustic instruments and the voice,
for it is from these sounding media that the idea of 'note'
originates in the first place. In the electronic medium,
however, methods of sound production are built upon very

2 Otto Laske, "Toward a Music Definition of Computer Music",
paper given at the International Computer Music Conference,
Denton Texas, November 1981, p. 11.

3 Pierre Schaeffer, Traité des Objets Musicaux, p. 281.
"... le plus petit élément musical significatif,... Les
musiciens occidentaux admettent que ce soit la note. Mais
qu'est-ce qu'une note?"
different technologies, even when they include means for imitating the 'notes' of non-electronic instruments. Identifying this set as the terminal vocabulary of a grammar for electro-acoustic music would unnecessarily limit the spectrum of sonic possibilities available to the composer.

In a T grammar, the terminal alphabet will be known as the set of S-events. These S-events are simply any low-level compositional events which form convenient fundamental units for some sound-synthesis scheme. This allows the form of an S-event to vary from application to application, providing a degree of generality to the T grammar concept. The only qualification that a T grammar imposes on the S-event is that it have a beginning, for, since T grammars are dedicated to the temporal organization of S-events, each S-event must have some specific moment which can be temporally located. This moment is taken to be the beginning of the event. By locating only the beginnings of events and not insisting they have definite durations or endpoints, the grammar is able to order events of indeterminate length.

To recapitulate, the purpose of T grammars is the temporal organization of sets of objects called S-events. Some other process may map S-events onto sonic realizations, but the nature of that mapping and the character of the resultant sounding objects are not determined by the grammar. As a result, T grammars are independent of any particular acoustic model. Their only purpose is the positioning, in time, of their terminal symbols: sets of S-events.
Strings

Any theoretic system developing the music-as-language metaphor will eventually encounter one fundamental dissimilarity between music and language: the multi-stream, polyphonic texture of most musical styles versus the single-stream, monophonic structure of language. This view of language as unilinear is as much a tenet of linguistic science as it is of the formal theory of languages, for both disciplines view language as a concatenation system:

In the realm of discrete systems, moreover, we limit ourselves to concatenation systems and to their further algebraic structure and their interrelations. In particular, we think of the flow of speech as a sequence of discrete atoms that are immediately juxtaposed, or concatenated, one after the other. Simple as this limitation may sound, it has some implications worth noting.4

In the following discussion, the implications of adopting a concatenation system for the computer representation of music will be examined. To illustrate, consider the following. Let X and Y be two symbols representing musical events. Without considering the possibility of repetition of these symbols, only two unique strings may be formed by their concatenation: XY and YX. If, as might well be the case in a musical score representation, left-to-right ordering represents a relationship of time, then at most two such temporal relationships between two musical events can be expressed by

concatenation. But, since temporal overlap is a possible relationship between musical events, many more than two temporal relationships can exist. As Milton Babbitt points out:

...I shall assume on purely empirical grounds that there are eleven qualitatively significant temporal relationships which can hold between two musical (say, pitch) events.5

Babbitt is able to distinguish eleven qualitatively significant temporal relationships by including both initiation time-point and termination time-point as attributes of the musical event. If the notion of musical event is taken to be the S-event, however, then there is no termination time-point to consider,6 and but three qualitatively significant temporal relationships can be held:

1. The beginning of X precedes the beginning of Y.
2. The beginnings of X and Y coincide.
3. The beginning of Y precedes the beginning of X.

Thus, although the number of possible qualitative relationships is now greatly reduced, there still exists one relationship too many to be represented by concatenation alone. The problem is even more acute with quantitative relationships—those which express, in addition to precedence, coincidence or succession, some measure of the time separating the events concerned.


6. Recall that the S-event has a specific beginning, but no specific end.
Does this mean that the wealth of temporal relationships possible between musical events cannot be represented by the concatenation of musical event-symbols? The answer is no, for the expression of both quantitative and qualitative temporal overlap is made possible through the introduction of a class of symbols dedicated exclusively to musical temporality.

Such temporal specification symbols may assume one of several forms. As a first example, consider the following:

\[ \text{C D E} \]

Here each pitch symbol \( \text{C,D,E} \) is followed by another symbol \( \text{D, D, D} \) indicating the duration of the pitch. Now, although strings of this type can specify quantitative temporal succession, they cannot specify temporal overlap. Augmenting the set of fundamental units with symbols of duration, then, does not by itself provide for the representation of overlapping events.

A more powerful approach associates with each event a symbol specifying onset time as a displacement from the beginning of the score. In the following example, S-events S1, S2 and S3 are each preceded by an integer representing onset time in seconds from the beginning of the composition:

\[ 0 \text{ S1 1 S2 1 S3} \]

Clearly, this scheme facilitates the representation of quantitative temporal overlap, for the durations of the S-events are independent of their onset times. The following
example, written in the language of Music V,\textsuperscript{7} combines both duration and onset-time techniques:

\begin{tabular}{|l|c|c|c|c|c|c|}
\hline
\textbf{field} & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\textbf{line} & 1 & \text{NOT} & 0 & 1 & 1 & 40 & 440 \\
& 2 & \text{NOT} & 1 & 1 & 1 & 40 & 660 \\
& 3 & \text{NOT} & 1 & 1 & 1 & 40 & 880 \\
& 4 & \text{NOT} & 0 & 1 & 1 & 40 & 220 \\
\hline
\end{tabular}

Figure 3-1 A Music V score. The six fields of each line have the following meaning:

1) Identifies the line as defining a musical event (i.e. \textit{NOTE}).
2) Onset time, in seconds, of the event.
3) An index specifying timbre definition--defined elsewhere in MUSIC V.
4) Length, in seconds, of the event.
5) Amplitude, in decibels, of the event.
6) Frequency, in hertz, of the event.

Several disadvantages of onset-time techniques can be pointed out. As figure 3-1 illustrates, left-to-right (or in this case, top-to-bottom) temporal coincidence or succession is not a feature of the technique, for the last event in this example temporally precedes the second and third events. As a result, a preliminary scan of the score is necessary before it may be properly interpreted. Those events scheduled to occur at the beginning of a composition, like those of lines 1 and 4 in figure 3-1, cannot be determined with certainty until after the entire score has been read. This problem can only be overcome at the cost of imposing ascending temporal order upon the onset times.

Another disadvantage of onset-time technique becomes

evident if one wishes to modify the score: subsections of such a representation cannot be temporally repositioned without adjusting their onset times. For the same reason, repeated sonic sequences, identical in every respect, cannot be coded as identical substrings if they are meant to begin at different times.

A final example of computer representations of music using temporal specification symbols employs the concept of entry delay. With this method, each event is followed by a number representing time delay until the onset of the next event in the string:

\[ S1 \ 1 \ S2 \ 0 \ S3 \ 3 \]

Here again, quantitative temporal overlap of events can be specified. Unlike onset-time techniques, however, such strings can be interpreted as they are read: no preliminary scanning is necessary. In addition, substrings representing identical sonic sequences may be coded identically, regardless of their position in the string. This technique is at once more powerful and flexible than both duration and onset-time techniques.

To summarize, any grammatically based computer-music system must include some method for the temporal organization

---

8 The time delay associated with the last event in such a string is clearly meaningless, as no event follows it. Associating with each event the time delay from the previous event in the string instead of the delay until the succeeding event does not help the matter: it simply associates a meaningless time delay with the first event of the series instead of with the last. Either delay could be 0.
of music. Concatenation of musical events alone lacks the power to represent all the possibilities of event succession and overlap. Augmenting the grammar's terminal alphabet with temporal specification symbols greatly increases the power of such catenary systems. Of the three such techniques just examined—duration, onset time and entry delay—the last is clearly the most successful. Accordingly, entry-delay symbols are the means by which T grammars will organize the temporal aspects of music. What makes T grammars unique, however, is that such symbols are taken out of the domain of the terminal alphabet and embedded in a higher-level hierarchical structure.

T-events and T-trees

In the quadruple \((V, T, P, S)\), elements of \(V\), in their normal usage as the variables of a grammar, have no intrinsic properties other than their own name. The concept of a variable having a value, however, is easily added to the formalism. In a T grammar, elements of the non-terminal set take the form \([A,X]\), where \(A\) is the label, or name of the variable, and \(X\) is its value. The value associated with a given label is unique and constant within any T grammar—every label is associated with one and only one value. To illustrate, the set \([A,1], [B,1]\) is possible, whereas the set \([A,1], [A,2]\) is not. It will sometimes prove more convenient to refer to any variable \([A,X]\) as simply \(A\), and, where necessary, to say that \(A\) has the value \(X\). In what
follows, the two notions will be used interchangeably.

The designed purpose of this scheme is that the values of the variables will represent entry delay expressed in multiples of some convenient temporal unit. Such variables will be referred to as T-events. The musical representation generated by a T grammar will be a derivation tree built of T-events and S-events by successive applications of rewrite operations from the set of productions P. Define a process E(t) which will scan the tree, visiting each event in a given order and at a given time. Events at identical depth are scanned from left to right with the constraint that if [A,Y] and [B,Z] are two T-events with a common immediate ancestor and [A,Y] is to the left of [B,Z], then [B,Z] will not be visited until Y temporal units have gone by since [A,Y] was visited. The following illustration serves to clarify this process:

```
[A,0]  
   /   
  [B,10] [C,0] [B,10]  
   /       
 [D,20]    [E,0]
```

Figure 3-2 A derivation tree composed of T-events.

Here, E(t) will visit [A,0] at precisely the same time as it visits the leftmost [B,10] and [D,20]. [C,0], however, will not be visited until 10 temporal units have elapsed. The rightmost [B,10] is visited 0 temporal units later. Finally, after a further 10 temporal units have gone by (20 units since [A,0]), [E,0] will be visited. Note that in this scheme,
events with a common direct ancestor must always exhibit left-
to-right temporal coincidence or succession, despite the fact
that the leaves of the tree, as exemplified by figure 3-2, may
not.

This method of score representation has all the benefits
that a hierarchical structure can claim over a purely linear
one. Some of these advantages are primarily implementational,
for data structures based on tree graphs have been studied
extensively by computer scientists. As a result, a large body
of well-understood algorithms exists for building and
manipulating such objects. Other benefits are chiefly music-
theoretic: in particular, the similarity of the model to
hierarchic branching structures in contemporary music theory.
If T-events are taken to be the labels of formal units in some
musical score, then it is easy to see how trees of such objects
reflect views of musical form in which the work is seen as a
group of sections, each of which is composed of still smaller
sections, and so on, to the depth of the fundamental unit.
Computer systems which implement schemes like the T-grammar
concept presented here have the opportunity to embed elements
of form into the very fabric of the score itself, rather than
relegating them to a position which is, in some sense, outside
of the score.

One of the obstacles that makes this grammatical model
conceptually difficult to grasp is the fact that the entry
delays somehow occur 'between' branches of the derivation tree.
In what follows, an alternative methodology for constructing
derivation trees is developed which makes the concept of entry delay more manageable.

In T grammars, derivation trees will be constrained to binary branching alone. A distinction will be made between left branching and right branching, and this distinction will be directly related to the form of the rewrite rules themselves. In particular, the first symbol on the right-hand side of a rewrite rule will be taken as the left branch of the left-hand side. The second symbol of the right-hand side represents the right branch of the first right-hand side symbol, the third, that of the second, and so on. To illustrate, consider the following productions:

\[ A \rightarrow B \ C \ D \quad B \rightarrow C \ D \]

Both figures 3-3 and 3-4 use these productions to create the same two-step derivation, but whereas figure 3-3 shows the corresponding multi-branching k-ary tree, figure 3-4 exhibits a binary derivation tree.

Step 1. \( A \rightarrow B \ C \ D \)

\[
\text{Figure 3-3 A two-step derivation and its corresponding k-ary tree.}
\]
Step 1. \( A \Rightarrow B\ C\ D \)

Step 2. \( B \Rightarrow C\ D \)

Figure 3-4 The two-step derivation of figure 3-3 and its corresponding binary tree.

Productions whose right-hand sides either mix variables and terminals or contain more than a single terminal cannot be interpreted according to this scheme. For example, if upper-case letters indicate variables and lower-case letters terminals, then the productions of figure 3-5 correspond to defective derivation trees. In both trees, terminal symbols appear as interior nodes of the tree, contradicting the convention that they may only be the labels of leaf nodes. Such defective trees may easily be eliminated, however, by imposing a particular form on the set of productions \( P \). If all
productions have one of the forms:

1. \( A \rightarrow X, A \in V, X \in V^+ \)
2. \( A \rightarrow a, A \in V, a \in T \)

then structures like as those of figure 3-5 are eliminated. Such productions will be termed semi-Chomsky normal, for they are often developed as a preliminary step in proving that any context-free grammar not containing the null string is equivalent to a grammar in Chomsky normal form.\(^9\) In terms of \( T \)-events and \( S \)-events, these productions may be expressed as follows:

---

\(^9\) A grammar in Chomsky normal form is one in which all productions are of the form \( A \rightarrow BC \) or \( A \rightarrow a \), for \( A, B, \) and \( C \) in \( V \) and \( a \) in \( T \). For further information, see Hopcroft and Ullman, *Introduction to Automata Theory, Languages, and Computation* (Reading, Mass: Addison-Wesley, 1979), pp.92-94.
1. A → X, A in TE, X in \(TE^+\)
2. A → a, A in TE, a in \(SE\)

-where TE is the set of T-events and SE the set of S-events.

We are now in a position to show how binary trees generated by productions in semi-Chomsky normal form can be useful models for the temporal organization of musical scores. First, referring to figure 3-4, the tree will be rotated counter-clockwise so that left branches become vertical and right branches horizontal. T-events C and D will be rewritten as S-events. Thus:

```
A
  \_B
    \_C
      \_D
        s3
  s1 s2

Figure 3-6 The binary tree of figure 3-4 rotated counterclockwise, with T-events C and D rewritten as the S-events s1 to s4.
```

Such a graph will be referred to as a T-tree.10 Suppose both T-events A and D in figure 3-6 above have the value 0, while C has value 5, and D 10. Expanding the T-events A, B, C, and D to show these values yields the following:

10 The rotation of the T-tree is intended as a user convenience and has nothing to do with the computer implementation of the scheme.
Define a function, \( T(X,Y) \), the temporal distance between any T-event variable \( X \) and one of its successors \( Y \), either a T-event or an S-event. \( T(X,Y) \) will be the sum of all the values of the T-events having horizontal branches on the path from \( X \) to \( Y \), not including the value of \( Y \).

More formally, if \( X \) and \( Y \) are two nodes in a T-tree such that \( X \) is an ancestor of \( Y \), and if the path from \( X \) to \( Y \) is described by the sequence \( X_0 \), \( X_1 \), \( X_2 \), \ldots \( X_n \), where \( X = X_0 \) and \( Y = X_n \), then

\[
T(X,Y) = \sum_{i=0}^{n-1} Z(X_i,X_{i+1})
\]

where

\[
Z(X,Y) = \begin{cases} 
\text{value of } X \text{ if } X_i \text{ branches horizontally} \\
\text{to } X_{i+1}. \\
0 \text{ otherwise.}
\end{cases}
\]

Note that if \( X \) is an ancestor of \( Y \), \( X \) must be a T-event, whereas \( Y \) may be either a T-event or an S-event.
Referring to Figure 3-7,

\[
\begin{align*}
T([A,0], s1) &= 0 \\
T([A,0], s2) &= 5 \\
T([A,0], s3) &= 10 \\
T([A,0], s4) &= 15
\end{align*}
\]

The advantages of a binary tree representation over a k-ary one, from the standpoint of computer implementation, are well-known: the binary solution leads to significant gains in storage efficiency while allowing simpler, more straightforward algorithms for traversal and maintenance. Though the critical function \(T(X,Y)\) may be defined on a k-ary tree, its definition is much simpler in the case of the binary T-tree. This simplicity results from the fact that the entry delays do not occur 'between' the branches of the tree, as they do in the k-ary case, but instead occur 'along' the branches, as shown in figure 3-8:

\[
\begin{array}{c}
[r,0] \\
[A,10] \quad [B,10] \quad [C,0] \\
\end{array}
\quad [r,0]
\begin{array}{c}
[A,10] -- [B,10] -- [C,0]
\end{array}
\]

Figure 3-8 A k-ary tree and its corresponding T-tree. In the T-tree, entry delays occur along the branches, rather than between them.

The T-tree's ability to represent ideas of musical structure based upon hierarchical categories merits some examination. Whereas events connected by horizontal branches represent an 'is followed by' relationship, those connected vertically may be thought of as representing an 'is made out
of' relationship. For example, a music-theoretic statement such as "this fugue is made out of exposition followed 20 seconds later by development followed 20 seconds later by stretto" has a direct T-tree correlate:

```
[this_fugue,0]
  |  
[exposition,20]---[development,20]---[stretto,20]
```

Figure 3-9 A T-tree representing "this fugue is made out of exposition followed 20 seconds later by development followed 20 seconds later by stretto".

The T-tree can easily represent relationships occurring between formal units which operate simultaneously. To illustrate, the statement "exposition is made out of subject followed 10 seconds later by countersubject followed 0 seconds later by answer" translates into the following T-tree:

```
[exposition]
  |  
[subject,10]---[countersubject,0]---[answer,10]
```

Figure 3-10 A T-tree representing "exposition is made out of subject followed 10 seconds later by countersubject followed 0 seconds later by answer".

To conclude, T grammars are dedicated to the temporal organization of sets of fundamental compositional units called S-events. Unlike many score-representation schemes, however, this ordering is not achieved by uniting a set of temporal specification symbols with the grammar's terminal alphabet. Instead, temporal information is embedded in the variables themselves. As a result, derivation trees of T grammars not
only represent structural descriptions of the strings of S-events they generate, but also specify the exact temporal location of the beginnings of these S-events within a composition. Such T-trees, by restricting themselves to binary branching alone, are more efficient than their κ-ary counterparts from the standpoint of computer implementation as well as being more attractive means for representing music in a computer in light of several music-theoretic considerations.

AMA

This section presents an overview of a computer system for music composition and performance entitled "A Music Automaton" (AMA). Based on a T grammar, AMA provides a concrete example of the theoretical ideas developed in this chapter. The system has been fully implemented on the Synclavier II digital musical instrument.

An important feature of the S-event concept is its adaptability to diverse methods of sound synthesis. The Synclavier II provides additive synthesis with frequency modulation through a bank of digital oscillators; accordingly, each S-event in AMA controls the operation of a

---

12 See the appendix to the present paper for a user's manual for AMA.

13 Synclavier II is a registered trademark of New England Digital Corporation, box 540, White River Junction, Vermont.

single oscillator for an arbitrary length of time.

As figure 3-11 illustrates, S-events have an internal tree-like structure consisting of seven primary branches, where each such branch corresponds directly to one physical input of an oscillator. Each branch in turn forks into two further branches: the leftmost points to a list of data values ready for delivery to the corresponding oscillator input, the rightmost points to a list of time values representing entry delay between corresponding data values.

Figure 3-11 The internal structure of an S-event in AMA. The various abbreviations have the following meaning:

lower case strings - identifiers representing lists of values
CW - carrier wave
CF - carrier frequency
CE - carrier envelope
MF - modulator frequency
ME - modulator envelope
VO - volume
PO - stereo position
DA - data
TI - time
The AMA user employs a special-purpose editor to create as many S-events as desired. Every S-event is given a unique name, and the set of such names constitutes the terminal alphabet of the T grammar. T-events, the name-value pairs which make up the grammar's variable alphabet, are similarly created by a special-purpose editor. One particular T-event \([r,0]\), the start symbol of the grammar, is created automatically by AMA.

Computer graphics techniques greatly facilitate the user's communication with AMA. The heart of the system is a graphically-represented T-tree which is created and modified in response to user commands resembling the rewriting rules of generative grammar. Each such command has the following form:

\[
\text{pathname} \Rightarrow T^+ \mid s, \text{ for } T \in \text{TE (the set of T-events)} \\
\text{and } s \in \text{SE (the set of S-events)}
\]

where 'pathname' is a sequence of T-events, each preceded by the slash character, '/'. The sequence names those T-events which lie along some path in the T-tree commencing with the start symbol. The last T-event in a pathname is that symbol which will be rewritten by the right-hand side of the production. Since there may be many instances of the same T-event in the T-tree, the use of a pathname rather than a single symbol as the production's left-hand side provides a means for specifying which T-event in the tree should be rewritten. Figure 3-12 shows a simple T-tree, together with the commands used to generate it.
Figure 3-11 A simple T-tree together with the 4 commands used to create it from its initial state: a single T-event r, the start symbol of the grammar. The rewriting arrow '=>' is omitted from such commands. The symbols r, t1, and t2 are T-events, while s1, s2, and s3 are S-events.

Occasionally, both the horizontal and vertical branches issuing from some T-event will lead directly to successor T-events having the same name. In such cases, a pathname may be ambiguous. By convention, AMA follows the vertical branch. This default interpretation may be overridden by preceding the name of the successor T-event with two slash characters instead of one--this instructs AMA to follow the horizontal branch. In the following T-tree, for example, the pathname '/r/t1/t2' refers to the T-event t2 at the bottom, while '/r/t1//t2' refers to the one on the right:

Figure 3-12 A T-tree in which the two T-events t2 have the same pathname. This ambiguity is resolved by referring to the one on the bottom as '/r/t1/t2' and the one to the right by '/r/t1//t2'.

T-tree performance is accomplished by means of this same pathname concept. Upon receiving a pathname, AMA performs the
entire subtree rooted in the last named T-event of the path. With this scheme, any subsection of the score may be singled out for performance. Upon receiving the pathname 'r', AMA will perform the T-tree in its entirety.

AMA is but one possible implementation of the T-grammar concept. The structure of the AMA S-event, for example, reflects the architecture of one particular synthesizer: it can easily be modified to accommodate other sound-synthesis technologies. A more ambitious implementation of the T-tree concept would associate with each T-event certain variables affecting the interpretation of that event's descendants in the T-tree, much like the Mscore object of the SSSP's data structure discussed in chapter II.

Evaluation

This chapter has described a type of grammar, T, dedicated to the temporal organization of certain fundamental compositional units called S-events into musical works. A computer system for composition and performance based on the T-grammar concept has been introduced. This system, entitled AMA for 'A Music Automaton', has been implemented at the electronic music studio of McGill University's Faculty of Music, and has seen considerable use by student composers. Both AMA and the theoretical concepts on which it is based may be evaluated in light of this experience.

Because of the simplicity of the procedures for traversing a T-tree, AMA is able to perform extremely complex scores in
real time. Very little processing of these scores is necessary before they can be played by the synthesizer. Consequently, the time which the AMA user must wait from when a score is modified until that modification may be heard is of the order of a few seconds: a fact which greatly facilitates interactive composition.

AMA is not a system easily mastered by the unsophisticated user. The T-grammar concept, with its basis in the formal theory of languages, is quite unlike anything in the educational background of most musicians. By contrast, users with a background in computing science master the system quickly and easily: many of its structures and concepts are already familiar to them.

Although AMA has greatly facilitated the realization of many compositional projects, it may not be the perfect vehicle for others. Works based primarily on some form of compositional algorithm may be more conveniently produced with the aid of a general-purpose programming language. Works which make extensive use of a traditional 'note' concept may be inconvenient to create, though this is more a result of AMA's S-event structure than of the T-grammar itself. In fact, AMA's rejection of the 'note' concept has been seen by most users as a distinct advantage: it encourages them to explore other dimensions of compositional thought.

AMA, then, is not a replacement for other computer music systems, but an alternative to them. Its strength lies in its
ability to represent compositions as hierarchical branching structures: a strength which results directly from the T-grammar concept on which it is based.
BABBIOGRAPHY


------. "Introduction to a Generative Theory of Music", Sonological Reports, No. 1b, Institute of Sonology, Utrecht, the Netherlands (January, 1973).


Roads, C. Composing Grammars. Published by the Computer Music Association, P.O. Box 1634, San Francisco, California 94101.


AMA USER'S MANUAL

AMA is a computer system for music composition and performance based on a T-grammar. The system is designed for New England Digital Corporation's Synclavier II® digital music synthesizer.

AMA supports the sound synthesis technology of the Synclavier's digital oscillators: Fourier synthesis and frequency modulation. No attempt was made to augment these technologies, for the T-grammar concept is not a theory of digital sound production. Rather, it organizes the computer's resources in a way which is at once coherent for the composer and efficient for the host machine.

AMA was developed entirely at the electronic music studios of McGill University's Faculty of Music.
# Table of Contents

1. Overview.................................1

**The main modules:**

2. Module Interchange......................6
3. Module Catalog..........................8
4. Module Load/Save........................10
5. Module Play.............................12
6. Module Objed............................14

**Submodules of objed:**

7. String Editors..........................22
8. Wave Table Editor.......................25
9. Envelope Graphic Editor.................28
10. S-event Editor..........................31
11. The T-event and TEtree Editors.........33
12. Module Form............................40

**Miscellaneous:**

13. Theory of operation....................45
14. Installing AMA.........................53
List of Plates

Plate I  Module Interchange
Plate II  Module Catalog
Plate III  Module Load/Save
Plate IV  A typical string editor
Plate V  The S-event editor
Plate VI  The wave table editor
Plate VII  The envelope graphic editor
Plate VIII  The TEtree editor
Plate I  Module Interchange.
**Plate II**  Module Catalog. A portion of the catalog :AMA is displayed.

---

| Catalog- > : ama |
|---|---|---|---|
| Extent: 08001 sectors Used: 06611 Free: 01389 |
| Cat : 00032 entries Used: 00022 Free: 00010 |

| Options |
|---|---|---|---|
| L | List current catalog |
| D | list a Different catalog |
| U | Unsave a file |
| M | Move a file |
| C | Copy a file |
| E | Exit: return to interchange |

| Commands |
|---|---|---|---|
| > | |

| AMA.X | -- subcat -- 12032 wds. -- 02001 secs. |
| AMA.MAX | -- subcat -- 12032 wds. -- 02001 secs. |
| SCRNS | -- subcat -- 12032 wds. -- 02001 secs. |
| ADJUST | -- exec -- 03087 wds. -- 000113 secs. |
| AMA | -- exec -- 05139 wds. -- 000213 secs. |
| CAT | -- exec -- 05913 wds. -- 000243 secs. |
| CHAIN | -- exec -- 00430 wds. -- 000023 secs. |
| COMP | -- exec -- 07897 wds. -- 000313 secs. |
| DISK.BANK | -- data -- 23296 wds. -- 00092 secs. |
| EMIT | -- exec -- 05515 wds. -- 000213 secs. |
| FORM | -- exec -- 06392 wds. -- 00039 secs. |
| FORMA | -- exec -- 06003 wds. -- 00022 secs. |
| LOS4 | -- exec -- 07372 wds. -- 00029 secs. |
| OBJED | -- exec -- 07479 wds. -- 00030 secs. |
### Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Load workspace</td>
</tr>
<tr>
<td>S</td>
<td>Save workspace</td>
</tr>
<tr>
<td>C</td>
<td>Create workspace</td>
</tr>
<tr>
<td>O</td>
<td>Open a workspace for copying</td>
</tr>
<tr>
<td>P</td>
<td>Put object into current WS</td>
</tr>
<tr>
<td>D</td>
<td>Display objects in current WS</td>
</tr>
<tr>
<td>I</td>
<td>Display objects in open WS</td>
</tr>
<tr>
<td>E</td>
<td>Exit: return to module interchange</td>
</tr>
</tbody>
</table>

### Commands

```
> 
```

Plate III  Module Load/Save.
Plate IV  A typical string editor. This plate shows a data string called 'pitch' together with a time string called 'duration'. The 'pitch' object has type FS and attribute PI; 'duration' has type TS.
Plate V  The S-event editor. This plate shows an SE called 'tree' with the ES object 'env100' attached to its CE DA slot. All other slots are empty.
Plate VI The wave table editor. In this plate, a WT object is being edited with the cross hair device.
Plate VII The envelope graphic editor. In this plate, an ES object called 'en' and a TS object called 'et' are being edited using the cross hair device.
Edit TETree

Enter production or control command:

/r/long a b a
/r/long/short/long a b

System response:
Okay.

Production forms:

pathname -> pathname : TE+
pathname -> SE
pathname redraw TETree from pathname.

Control commands:

<ctrl R> Redraw TETree from root
<ctrl D> Display TAGlist objects
<ctrl E> Exit to parent module
<ctrl P> Play an object

Plate VIII The TETree editor. The tree is composed of TE objects 'long', 'short', 'a', 'b', in addition to the root of the TETree, 'r.'
1. OVERVIEW

Invoking AMA

AMA is invoked by first running the Scientific XPL operating system, then entering the following sequence:

```
READY
enter <subcatalog name>
READY
old :ama:ama
READY
run
```

The screen will then be cleared, and the main module, the Module Interchange, will fill the screen.

Modular Organization

AMA consists of several programs, each of which is known as a module. Modules are invoked by typing commands in response to AMA's prompt character, '>'. The overall organization of these modules is shown in the following figure:

```
Interchange

Catalog  Objed  Play  Load/Save

--------
```

Figure 1.1 AMA's modular structure.

The Interchange module serves as a link to the other modules. You can invoke other modules from the interchange, but the only path out of these modules is back to the interchange.

Module Load/Save is dedicated to the management of files on disk. The commands of this module allow you to store and recall compositions, to copy sections of one composition into another, and to clear AMA's memory, called the 'Workspace', in order to start a new work.

Module Catalog is used to display file names and file information. In addition, the module provides facilities
for file copying, moving, and deleting, allowing access to all subcatalogs on the system.

Module **Objed**, for 'object editor', is the heart of AMA. It is through this module that the actual entering of information for a composition is done. It contains many sub-modules, each of which is tailored to the demands of entering and editing the data which AMA requires to control the digital oscillators.

Finally, module **Play** serves to perform the score representations created by module **Objed**. Special links between **Objed** and **Play** allow entering **Play** without passing through module **Interchange**. This is the only exception to the otherwise strictly tree-like structure of AMA, and is indicated by a dotted line in figure 1.1.

**The Workspace**

All information entered into AMA is stored in a block of memory known as a workspace. Workspaces may be saved on disk, then recalled later for performance or editing. You must give a name to each workspace you create.

Workspaces are of a fixed size as determined by the hardware configuration of the host computer. It is possible to fill a given workspace to the point where no more information may be entered. When this condition arises, AMA will deliver a short message about the condition, then return control to the operating system. At this point, a composition cannot be made any larger except by splitting it into smaller subcompositions and reassembling them on audio recording tape.

The main purpose of module **Load/Save** is the saving and recalling of the AMA workspace from disk. In addition, the module provides the facility to create new, empty workspaces.

AMA maintains a default workspace called 'COREBANK'. Whenever you move from one module to another, AMA saves a copy of the current workspace in COREBANK. When you run AMA, this workspace is automatically loaded. This means that if AMA should 'crash' due to a power failure or system problem, little, if any, of your data will be lost. You may exit from AMA, then reenter AMA, all without saving the current workspace. This workspace will remain intact. It also means that if you use AMA immediately after another user, you will find it loaded with the workspace of the previous user.
Objects

All of the essential elements of an AMA score are known as objects. Every object in AMA must have a user-supplied name and type. Among the possible types which an object may have, three types are compound; that is, they are constructed of other objects. The first object we will examine is a compound type known as the S-event (see plate V).

An object of type S-event may be understood as a recipe for one of the digital oscillators. In order for the oscillator to emit a sound, it must be provided with a set of values representing such parameters as waveform, frequency, envelope, volume, and stereo position. The S-event is simply an object uniting these parameters into a single unit.

There are seven parameters, or data values, which the oscillators require in order to produce sounds. For each of these parameters, the S-event has a separate field. These fields resemble slots, each of which has a name. Into each field the user may write the name of another object. S-event fields are named as follows:

- **cw** -- carrier wave -- Specifies the waveform of the oscillator.
- **cf** -- carrier frequency -- Specifies the frequency the oscillator will emit.
- **ce** -- carrier envelope -- Specifies the shape of the overall sound envelope.
- **mf** -- modulator frequency -- (optional) Specifies a frequency which will modulate the cf field.
- **me** -- modulator envelope -- (optional) Imposes an envelope on the index of modulation.
- **vo** -- volume -- Sets the overall volume of the oscillator.
- **po** -- position -- Specifies the stereo output position of an oscillator.

In AMA, you are not restricted to specifying single, individual data values for the fields of an S-event. Instead, each of the S-event slots can be filled with the name of an object containing a list of data values. Such an object is called a string, and most of the submodules of Object are concerned with the creation and modification of these strings.
For every string of data values, the S-event must also be provided with a string of durations. If you created a list of, for example, four frequency values, and wished them to be played one after the other, AMA must know how long each frequency value is to last in order to deliver the next value to the oscillator at the correct time. In AMA, a string of durations is an object of type TS, for Time String, and may be attached to any slot in an S-event labelled TI, for Time.

**T-events and the TEtree**

S-events, as the fundamental compositional units of an AMA score, can themselves be organized to form larger-scale forms. In AMA, a T-event is an object whose role is to specify the time-point separation between S-events. A tree-like hierarchical structure, called a TEtree, is built of T-events and S-events, and this TEtree is used by module Play to schedule the performance of S-events.

Mastery of the concept of the TEtree is essential for a working knowledge of AMA. Based on concepts drawn from the mathematical theory of languages, The TEtree is the most innovative aspect of the system. TEtrees are created and modified using a system of commands called productions or rewriting rules. As these commands are issued, the evolution of the TEtree is graphically displayed on the computer screen.

**Names**

As mentioned above, all objects in AMA (including the workspace) must be given a user-supplied name. Names may be from one to eight characters long, and may be comprised of any printable character other than the slash character, '/', or the space. If you attempt to include a '/', a space, or any non-printable character in an object's name, then AMA will automatically replace that character with a period, '.'.

Upper and lower case letters are significant in the names of AMA objects. This means that the names 'name', 'NAME', and 'NaMe' are, as far as AMA is concerned, three completely different objects. By contrast, upper and lower case letters are not significant in AMA commands.
Commands

All commands in AMA are issued by typing a single character followed by a carriage return, <RETURN>. In the module Objed, this character must be typed with the <ctrl> key held down in order to distinguish commands from object names. Commands may be entered in either upper or lower case. In all AMA modules, the command E (<ctrl E> in objed) is used to exit from the module. As a convenience, simply typing <RETURN> alone will have exactly the same effect as typing E.
2. Module Interchange

Introduction

The Module Interchange, plate I, is the first module encountered when AMA is invoked. Its sole function is to link together other modules which do the actual work of creating and playing AMA scores.

At the top of the screen you will see a line printed in inverse video (dark characters on a light background—shown in regular print in plate I). A small arrow pointing to this line identifies the name of the current module; in this case, module Interchange. The rectangle labelled 'OPTIONS' to the right of the screen briefly explains the commands available from this module.

All commands are entered by typing the first letter of the command, then <RETURN>. Thus, to enter module Catalog, type the character 'c' followed by <RETURN>. The character you type will be echoed in the rectangle labelled 'COMMANDS'. This command rectangle displays a command history: whatever you type is echoed and scrolled in this region, so the previous three lines are always visible.

Commands may be entered in either upper or lower case—AMA converts all characters to upper case before it examines them. You may, if you wish, leave the CAPS LOCK button on the keyboard depressed.

Commands

The following text describes all commands available from module Interchange.

C Entering this command will cause AMA to invoke the Catalog module. Enter this command if you wish to see what files you have saved on disk, or if you wish to delete, copy, move, or join disk files.

O This command invokes the Object Editor. The Object Editor, or Objed, is the most important module of AMA, for it is here that the actual creation of scores is done.
If, after issuing the O command, you receive the message 'No workspace yet selected', it is because an AMA system file called COREBANK, normally loaded automatically when you run AMA, is not present on the disk. In this case, enter the module Load/Save, create a new workspace with the C command, then return to the module Interchange. AMA will create the COREBANK file, and you will be able to enter Objed successfully.

P Issuing this command will invoke module Play. Play is the module from which scores are performed.

L This command invokes the module Load/Save. Load/Save is used for the management of AMA workspaces. Invoke this module if you wish to create a new workspace, to save the current workspace on disk, to load a previously created workspace from disk, or to copy objects from existing workspaces into the current workspace.

E This command will return you to the Scientific XPL operating system. Typing <RETURN> by itself has exactly the same effect as typing E<RETURN>.
3. Module Catalog

Introduction

Module Catalog, plate II, is used to display the names of files in catalogs and to manipulate them in various ways. The format of the screen and the manner in which commands are entered is identical to that of module Interchange. Unlike the module Interchange, however, module Catalog may require further information than simply the single character command which served in the Interchange. When this is the case, AMA will specifically prompt you for the needed information.

In order to use this module effectively, you must understand the Synclavier II's subcatalog system and, in particular, the notion of a file path or pathname. All of the necessary information can be found in the Synclavier II instruction manual. One point needs stressing: if AMA prompts you to enter the name of a complete pathname, you must enter the entire path. The following example may serve to make this distinction clear.

Suppose that AMA asks you to enter a 'complete' pathname. If you wish to reference a file called, say, FILE1, in a subcatalog called, say, TRIALS, which is in turn a subcatalog of your own subcatalog, say, MYCAT, then the complete pathname of the file is ':MYCAT:TRIALS:FILE1'. This is the case regardless of the subcatalog you were in when you entered AMA. If, on the other hand, AMA asks for simply the pathname of a file, and not its 'complete' pathname, and if the subcatalog you were in when you entered AMA was ':MYCAT:TRIALS', then you may, if you wish, simply enter FILE1. AMA will automatically supply the pathname of the current subcatalog (':MYCAT:TRIALS in the present example) to the beginning of any file name which does not begin with the colon character.

Commands

L This command lists the names of all files in the current catalog. If there are more names than can be conveniently shown on the screen, you will be prompted to enter either <RETURN> (to view the remaining names) or S<RETURN>, for Stop (to ignore remaining names).

D This command is very similar to L, except that it is used to display catalogs other than the current catalog. You will be prompted to enter the complete pathname of the catalog you wish
displayed.

U This command is used to delete (unsave) a file from disk. You will be prompted to enter the name of the file to delete. If this file is in the current catalog (the catalog at time of entry to AMA) then you need not enter its complete pathname. If you wish to delete a file in some other catalog, the complete pathname must be specified.

C Use this command to make a copy of a file. You will be prompted for the name of the file you wish to copy from as well as the name you wish to give to the copy.

M The move command is similar to C, the copy command. With M, however, the original file is deleted from disk after the copy has been made, effectively moving the file from one place in the catalog structure to another.

E Use this command to return to the module Interchange. Simply typing <RETURN> by itself will have the same effect.
4. Module Load/Save

Introduction

Module Load/Save, plate III, is dedicated to the management of the AMA workspace. It is the only module from which workspaces may be loaded from or saved to disk. Use this module to create a new workspace when you begin a new composition. Facilities are also provided for copying objects from one workspace to another.

The format of commands in this module is identical to that of the modules Interchange and Catalog. When you enter Load/Save, you will see the name of the current workspace (or the message 'No Workspace Created'), as well as the message 'No Open Workspace'. The purpose of the so-called 'Open' workspace is to facilitate copying objects from one workspace to another. You can copy objects from any workspace on disk into the current workspace, but before you do, you must explicitly 'open' the workspace you wish to copy from by entering the command O.

When you create a workspace by the command C, you will be prompted for a name. Every workspace must have a name, and this name can be up to 8 characters long. When you save a workspace on disk, you will be prompted for another name—the name of the disk file which your workspace will be saved in. This name may be completely different from the name of the workspace. By allowing the disk file name of a workspace to be different from the actual, 'internal' name of the workspace, you can have various versions of a workspace, say, COMP, saved on disk under names like COMF.BAK (for COM.BAcKup) or COMP.V3 (COMP, version three).

Commands

L The load command is used to load a previously created workspace into memory. This workspace will then become the 'current' workspace. You will be prompted for the filename of the workspace. If you attempt to load a file which is not a workspace, an error message will be issued.

S The save command will save the current workspace from memory onto disk. You will be prompted for the name of the file you wish to save the workspace in. This name need not be the same as the name of the workspace. If a file of the same name already exists on disk, you will be asked if you wish to replace that file.
C This command is used to create a new workspace. AMA will prompt you to enter the name you wish to give to the workspace. **Note:** Any previous workspace not saved on disk by the command S will be destroyed by this command.

O Use this command to open a workspace for copying. If you wish to use the command P, for put, you must have first opened some workspace so AMA will know into which disk file it must look to find the objects you wish to copy.

P The put command can be used to copy (put) objects from one workspace into the current workspace, provided the first workspace has been opened by the O command. You will be prompted for the name of the object you wish to copy. If the name of this object conflicts with the name of an object in the current workspace, then you will be prompted to supply a new, unique name. Otherwise, the object will be copied into the current workspace without any change of name. **Note:** unlike commands, the names of objects are case sensitive.

Only simple, not compound objects may be copied. This means that you cannot copy a wave list, an S-event or a TETree.

D The display command is used to display a list of objects in the current workspace. You will be prompted for the type of objects you wish displayed. If, for example, you wish to see a list of all the S-events in the current workspace, issue the command D and then, following AMA's prompts, type SE, for S-event. You will then see a list of all S-events so far created.

I Issuing this command has exactly the same effect as the D command except that objects in the open workspace will be displayed instead of those in the current workspace. It is an error to issue this command if you have not already opened a workspace by the command O.

E This command returns you to the module interchange. Typing <RETURN> alone has the same effect.
5. Module Play

Introduction

Module Play is dedicated to the performance of the scores created by module Objet. It will do any one of three things: return to the module Interchange, perform an S-event, or perform the TEtree or any subtree of the TEtree.

To return to the module Interchange, type <ctrl E>. Simply typing the character E will cause module Play to search the current workspace for an S-event with the name E. If it finds one, it will play it. If not, you will get an error message.

An S-event may be played by itself, and this can be accomplished by simply typing its name. To perform the TEtree, type the following command:

/r

This is the AMA-style pathname of the root of the TEtree. Any subtree of this tree may be performed by typing the pathname of that subtree (for a complete discussion of pathnames in the TEtree, refer to chapter 11, page 35).

Two shortcuts are provided to avoid unnecessary typing. First, simply entering the slash character '/' by itself is equivalent to entering the current TEtree root as displayed by the TEtree editor. Second, entering a double slash '//' will cause whatever object was last played to be played again. This alleviates the necessity of retyping long pathnames.

This manual assumes that the Synclavier II host is equipped with 32 oscillators. Since each S-event uses a single oscillator, it follows that no more than 32 S-events can be active at any one time. In reality, AMA reserves one of the oscillators for its own purposes, leaving 31 available for general use. If, during the course of performing a work, more than 31 oscillators are needed, AMA will issue an error message and terminate the performance.

When module Play receives the name of an object which it can perform, it first initiates a short compilation process. Upon completion of this, it prompts you to type <return>, after which the object will immediately be performed. If the object
is an S-event, then at the termination of that S-event you may enter another name for AMA to perform. If, however, the object is the TEdtree or any subtree of the TEdtree, then AMA requires you to enter an additional <ctrl E> before you can perform another object or enter the <ctrl E> command to exit module Play. You may terminate the performance of any object before it has finished playing by typing <ctrl E>. As a convenience, simply typing <RETURN> by itself is equivalent to typing <ctrl E>.

Commands

<ctrl E> This control character command will normally return you to module Interchange. As mentioned in the introduction of this manual, however, there is a link between module Objed and module Play. If Play has been entered from Objed, then <ctrl E> will return you to Objed (or one of its submodules), and not to Interchange. Typing <RETURN> by itself will have exactly the same effect as typing <ctrl E>.
6. Module Objed

Introduction

Module Objed, AMA's object editor, is by far the most important module of the entire AMA system. It is made up of several submodules, each of which is specifically designed for editing one particular type of object. Upon entry, Objed assumes you wish to edit an object. As a result, it expects you to type the name of that object you wish to edit. In Objed, as in all other modules, there is a group of single character commands, but these commands must always be entered with the control key held down in order to distinguish them from the names of objects. If you forget to hold down this <ctrl> key, your intended command will be interpreted by Objed as the name of an object, not as a command.

In order to use this module effectively, it is necessary to have a good grasp of how AMA organizes objects into other objects called S-events. An S-event may be thought of as a recipe for an oscillator. In order for an oscillator to emit a sound, it must be provided with the proper ingredients: lists of numeric values representing such parameters as frequency, volume, and waveform. An S-event is a compound structure composed of these objects—strings (or lists) of frequency information, of envelope information, and so on. Each of these objects must first be created using a specific editor designed for that function. They may then be combined into an S-event using the S-event editor. At that point, the S-event may be played using module Play, or linked into the TETree using the TETree editor.

There are no explicit commands for invoking the specific editors for each object type. Instead, AMA expects you to simply enter the name of the object you wish to edit. It will then search the workspace (actually, a list of names called the TAGlist) for that object. If the object is found, the appropriate editor will be automatically invoked. If, however, the object is not in the current workspace, or, to put it another way, on the current TAGlist, then AMA will create it. You will then be asked to specify a type (i.e. frequency, envelope, S-event, etc.) for this newly-created object, as well as any additional information AMA requires. The appropriate editor for that object will then be invoked.

The following example may serve to make this scheme clear.
Suppose you wish to create a frequency object with the name 'freqobj'. First, type the name of the object, 'freqobj'. AMA will search the TAGlist for the name 'freqobj' and, not finding it, assume you wish to create it. It will then prompt you to enter the type of that object. Since you wish freqobj to represent frequencies, enter FS, for Frequency String. A frequency string is a list of values with a certain format. You may create this list out of either note names, hertz, or Synclavier II key numbers. Just which of these methods you choose is up to you, but you must inform AMA as to your choice. AMA will prompt you for this frequency string 'attribute', and you must enter either PI for pitch names, HZ for hertz, or KE for key numbers.

Since a list of frequencies is of little musical interest unless it is associated with a list of durations, AMA requires that you edit FSs in association with a list of time values. This list is another object which has type TS, for Time String. After you have entered the attribute of object 'freqobj', AMA will prompt you for the name of a time string to associate with it. Suppose you enter the name 'tstring'. AMA will again search the TAGlist for the name 'tstring'. Not finding it, it will create it and automatically assign it the type TS. At this point, an editor which allows you to create and modify 'freqobj' and 'tstring' together will be automatically invoked.

The next time you wish to edit 'freqobj' and 'tstring', you need only type 'freqobj <return> tstring <return>' and the appropriate editor will be invoked. AMA remembers the types and attributes of all objects which have been previously created.

Clearly, the concept of 'type of object' is very important in AMA. The following is a list of the possible types which an object can have.

SE An object of type SE, for S-event, is a compound object containing slots, or fields, in which the names of objects can be written. There are seven such fields, each composed of two subfields. The first subfield is reserved for data strings: objects representing frequency, volume, etc., whereas the second subfield must always be a time string. Figure 6.1, page 16, serves to make this concept clear.

In figure 6.1, there are seven main fields—Carrier Wave, Carrier Frequency, Carrier Envelope, Modulator Frequency, Modulator Envelope, Volume, and Position. Each of these fields is divided into two subfields, one for a string of Data values, another for a string of Time values. Creating an S-event is
Figure 6.1 The structure of the S-event.

A matter of rewriting the x's in this figure with the names of other objects (the mechanics of this process are described in chapter 10, page 31). Such objects must 'fit' the slots in which they are written: it is, for example, an error to fill a slot labelled TI with any object which does not have the type TS. The various fields and the type of object which must fill them are defined as follows:

<table>
<thead>
<tr>
<th>S-event Data Subfield (DA)</th>
<th>filled by objects of type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF: Carrier Frequency:</td>
<td>FS: Frequency String: a list of frequencies having one of the attributes KE, HZ, or PI.</td>
</tr>
<tr>
<td>CE: Carrier Envelope:</td>
<td>ES: Envelope String: a list of numbers representing the amplitudes of turning points in the contour of a volume envelope.</td>
</tr>
<tr>
<td>MF: Modulator Frequency:</td>
<td>FS: Frequency String: as above, but AMA will use this string as a modulation frequency.</td>
</tr>
<tr>
<td>ME: Modulator Envelope:</td>
<td>ES: Envelope String: as above, but controls the index of FM modulation.</td>
</tr>
<tr>
<td>VO: Volume:</td>
<td>VS: Volume String: sets the overall volume for the oscillator.</td>
</tr>
<tr>
<td>PO: Position:</td>
<td>PS: Position String: a string of values indicating the stereo position of the S-event.</td>
</tr>
</tbody>
</table>
S-event Time Subfield (TI) filled by objects of type:
In all fields:

TS: Time String: a string of time values. Each of the data subfields must be associated with a TS in order for AMA to assign durations to their contents.

Naturally, you are not required to use FM synthesis. If you choose not to, then S-event fields MF and ME need not be filled with FSs, ESs and TSs.

It is possible to link the same event to several different S-event fields, provided the types match. You can, for example, use the same frequency string for both CF and MF. All seven S-event fields labelled TI could be filled with the same object, provided it has type TS. In addition, any single object may be linked into as many different S-events as desired.

FS As noted above, an object of type FS is a list of frequencies expressed in either Script-style note names (PI), in hertz (HZ—actually, hertz * 10), or in Synclavier II key numbers (KE). Any single frequency string can have only one of these attributes: you cannot, for example, have a frequency string which mixes HZ and PI.

Script-style note names have exactly the same format as pitch names in the Synclavier II Script language: a single character pitch (a,b,c,d,e,f or g) followed by an optional single- or double-character chromatic inflection (#,f,n for sharp, flat, or natural, with ## and ff representing double sharp and double flat respectively) followed by a single integer representing octave, with middle c in octave 3.

If you choose to enter frequency in hertz, then the values must be entered in units of hertz times 10. Concert A, for example, would be entered as 4400. Because of the limits of the sixteen-bit architecture of the ABLE computer, the largest value you may enter is 32,767 (i.e. 2^15 - 1), representing a frequency of 3,276.7 hertz. This does not mean that higher frequencies cannot be produced, only that the highest fundamental frequency you may specify by the hertz method is 3,276.7. You may create higher frequencies by many methods, the simplest being to create a waveform composed of an upper harmonic with no fundamental.

The keys on the Synclavier II keyboard are numbered internally with consecutive integers starting from 0 as the lowest to 60 as the highest (middle C = 24). You may, if you wish, enter frequency by referring to these numbers.
Envelope strings are strings of numbers representing the turning points of an envelope contour. In AMA, there are no restrictions of the 'attack, steady-state, initial decay, final decay, etc.' variety placed on an envelope. Instead, envelopes are simply arbitrarily long and arbitrarily complex straight-line graphs representing a volume contour to be imposed on the output of an oscillator.

The values making up an envelope string represent volume amplitudes in the range of 0 to 1023. When associated with a time string, they define an envelope contour. To illustrate, consider the following envelope contour graph:

![Envelope Contour Graph](image)

Figure 6.2 A typical envelope contour graph.

This graph would be represented by the following ES/TS pair:

<table>
<thead>
<tr>
<th>ES</th>
<th>TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td>100</td>
</tr>
<tr>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>0</td>
<td>300</td>
</tr>
</tbody>
</table>

A few points require explanation here. First, an envelope always begins at 0 amplitude. Consequently, it is not necessary to explicitly state the first values, 0 and 0, in the ES and TS. Second, each value on the time string represents the time for the oscillator to change from its current volume level (no matter what that is) to the level represented by the corresponding ES value (between 0 and 1023). Finally, if a given ES terminates by leaving the ES at a non-zero value, AMA will automatically 'close' the amplitude, inserting a zero value at the end.
given ES terminates by leaving the ES at a non-zero value, AMA will automatically 'close' the amplitude, inserting a zero value at the end.

Envelope strings may be used to control the index of frequency modulation when they are attached to the modulator envelope data field of an S-event. Their format and range is exactly the same as when they are used to envelope the carrier.

The response characteristics of the AMA envelope contour are 'piecewise exponential' with respect to a given envelope string value. The following graph details the relation between volume and ES value.

<table>
<thead>
<tr>
<th>Volume</th>
<th>0</th>
<th>128</th>
<th>216</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Minimum)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Maximum)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ES value

Figure 6.4 The relation between volume and ES value.

From figure 6.4, it can be seen that the envelope contour's response is linear between the '.'s, but the position of these '.'s themselves increase exponentially.

VS Each oscillator has a separate volume control. Its primary use is to adjust the output level of oscillators in relation to each other, but it may also be used in conjunction with ESs to create more dynamic control. Like ESs, VSs make little sense until they are associated with a TS. Unlike ESs, VSs have a maximum value of 255. If AMA finds a number greater than 255 in a VS, it will not report an error, but will interpret it
WL  Wave lists are compound objects—they are made up of the names of objects of types CA (for Coefficient Array) or WT (for Wave Table). CAs and WTs are objects which define the wave shape which will be loaded into the carrier side of the oscillator. A wave list consists of a list of CAs and WTs (you may mix the two types on the same list). The associated TS will determine how long each wave table will be used until the next one in the series is loaded. Thus a single S-event may use many different wave shapes.

Although the S-event contains a carrier wave field, there is no corresponding field for modulator waveform. This is because the Synclavier II's oscillators contain a preset sine wave table which is always used for frequency modulation. This modulator waveform cannot be altered.

PS  Objects of type PS determine an S-event's stereo position at a given time. A value of 0 indicates all the way left, a value of 63 indicates all the way right.

TS  Every object type we have examined so far (except the S-event) was described together with an associated object having type TS. Time strings are lists of numbers which AMA will interpret as spans of time in milliseconds. The longest value you may include in a time string is 32767, representing 32.767 seconds. This does not represent any limitation of duration on an object, for you may always enter a data value and its corresponding time value more than once. Entering a single value greater than 32767, however, will result in undefined operation.

CA  Objects of type coefficient array do not fit directly into any of the fields of an S-event. Instead, they form part of the WL objects explained above. A coefficient array is simply a list of values between 0 and 100 representing the amplitude of a given harmonic in a waveform composed of harmonically-related sinusoidal components. To illustrate, the string

```
100
0
100
0
10
```

represents a waveform whose components are a fundamental, 3rd harmonic, and 5th harmonic, where the 5th harmonic has one tenth the amplitude of the other two non-zero harmonics. You may specify as many harmonics as you wish, though those beyond the 128th will not form part of the resultant waveform.
WT Wave Tables, like coefficient arrays, are not directly linked to S-events. Instead, their names may appear as elements on WLs. A wave table is a list of 256 values between 0 and 255 which may be loaded into the digital oscillators to represent a waveform.

Commands

<ctrl P> This command provides a quick link to module Play, circumventing the necessity to pass through the module Interchange. After issuing this command, the prompt character will change from '>' to 'P>' to indicate that everything typed is being read by module Play, not the S-event editor. Other than this however, the computer screen will not change: Objed's screen will remain intact.

<ctrl D> Issuing this command allows you to examine the current state of the TAGlist. You will be prompted for an object type. The names of all objects of the given type currently in the workspace will be displayed.

<ctrl T> A special editor for editing TEtrees is invoked by entering this command. See the section on the TEtree editor (chapter 11, page 33) for details.

<ctrl F> This command invokes a submodule called Form. Form is used for doing convenient operations on objects: You may delete or modify existing objects, or create new objects using random numbers. Refer to chapter 12, page 41 for a complete description of this submodule.

<ctrl E> E stands for exit. Issuing this command will return you to the Interchange module. Simply typing <RETURN> will have the same effect.
7. String Editors

Introduction

For every type of object in the AMA workspace, AMA provides an editor especially designed for creating and modifying that object. Many of these editors share a similar format, one which is very close to the Synclavier II's line editor. In AMA, such editors are called 'string editors' (see plate IV) because the objects they act upon are strings of values.

The vast majority of AMA objects are edited using a string editor. The exceptions are the TETree, T-events, S-events, and wave tables. These objects are edited by other means, primarily graphic in nature. One object, the envelope string, has both a string editor and a graphic editor--choose one or the other depending on your purpose.

Since all the objects which come together to form S-events must be paired with time strings, string editors are designed to edit data string/time string pairs. Two pages of data will be displayed at once, with each page divided into two columns: one for 'data' and one for 'time'. Below each column there is a 'command rectangle': an area in which you may type in values or control character commands. Any values typed under a data column will change the corresponding data column, leaving the time column unchanged. Similarly, anything typed below a time column will change values displayed in the time column, without altering the data column.

The only type of object which a string editor can edit without an associated time string is a coefficient array. In this case, the column for 'time' information is not used. There is no way to edit an individual 'time' string directly: it must always be done in connection with some 'data' string.

AMA string editors are line number editors: all lines are numbered, and changes to any particular line are made by referring to that line's number. Only one value may be entered on any one line, hence every value you enter has a line number associated with it. In AMA, line numbers have a specific format: consecutive even numbers starting at 2.

To enter a value into a line, type the line number, the data, then <RETURN>. The data which you typed will be entered into the appropriate column. AMA does all the format checking it can at this time, and issues appropriate error messages
describing problems. It is possible to enter data with a line number which skips over existing line numbers, leaving a gap in the ascending sequence of 2. When this happens, however, the editor will automatically adjust the line number, removing the gap, before it stores the value.

All data typed into the editor must begin in column 4. This means that if you type a one-digit line number, then you will require two spaces between the number and the data. If, however, you type a two-digit number, then only one space is necessary.

Changes in already entered data are effected easily. To change a value, type the line number of that value followed by the new value--this new value will replace the old. Typing the line number alone with no value following it will delete the value at that number. To insert a value before another one, use the line number one less than the line number you wish it to precede. AMA will adjust all the line numbers to make room for this new value.

To get from the data column to the time column quickly, use the <TAB> key. One touch of this key will move the cursor into the next column to the right. It is possible to enter both a data value and a time value on the same line.

The string editors display two pages of values at a time, and these pages may be changed. Issuing the <ctrl P> instruction causes AMA to prompt you for a page number. The number you enter in response to this query will become the page on the left-hand side of the screen. The right-hand side will display the next page.

Unfortunately, there is not enough room on the screen to allow a display of the available commands. There are, however, very few commands to remember. In addition to the <ctrl P> command mentioned above, there are only two others: <ctrl D> will display objects on the current TAGlist by type, and <ctrl E> will return control to module Objed.

**Commands**

<ctrl P> The page command is used to select which page will be displayed as the left-hand page on the screen. The current page number is always displayed at the top of the screen, together with the highest numbered page of the object.
<ctrl D> Issuing this command allows you to examine the current state of the TAGlist. You will be prompted for an object type, after which the names of all objects of the given type currently in the workspace will be displayed.

<ctrl E> This command returns control to the main Objed module.
8. Wave Table Editor

Introduction

The wave table editor, Plate VI, is used for graphically creating a 256 point array of numbers which will be used by AMA to represent a wave shape. This wave shape, if it is attached to a wave list, may be linked into an S-event for eventual loading into one of the Synclavier II's digital oscillators.

The usual way of creating wave shapes in AMA is to specify a set of harmonic coefficients. This can be done by creating objects of type CA, for Coefficient Array, using a string editor. Objects of type WT, for Wave Table, offer an interesting alternative to this method. Instead of specifying the harmonic content of the desired waveform in terms of sinusoids, the wave table is actually drawn using interactive graphics facilities.

The wave table editor offers two methods of drawing wave forms: either the light pen or the computer terminal's built in crosshair device may be employed. When you invoke the wave table editor submodule, the crosshair method will be initially selected for you. To select the light pen, simply type T, for toggle device, and the light pen will be enabled, disabling the crosshair. If the light pen is enabled, then typing T and pressing the light pen to the terminal's screen will disable the light pen and enable the crosshair.

Although at first glance the light pen and the crosshair may seem to be very different devices, their usage is similar. Both of them are used as pointing devices. To point with the light pen, push the tip of the pen against the computer terminal's screen at the desired place until the sleeve in the pen tip retracts. The screen should flash briefly. This action is called 'entering a hit'. To enter a hit with the crosshair, move the crosshair to the desired location on the screen using the cursor arrow control keys, then type a space using the terminal's space bar. The effect will be exactly the same as entering a light pen hit at that point.

The wave table editor uses what might be called the 'two hit' algorithm for entering waveshape information. Initially, you will see within the editor's window display a wave shape representing a wave which is simply a straight line. To modify this shape, you must enter two 'hits' with either the light pen or the crosshair. AMA will then redraw the wave to stretch around the points you have entered.
The following illustrations serve to make this scheme clear. First, here is an illustration of the computer screen before any 'hits' have been entered.

![Figure 8.1 The wave table editor's screen display.](image1)

Now here is the same screen after two hits have been entered, but before AMA has had time to redraw the wave.

![Figure 8.2 The display of figure 8.1 with two 'hits'.](image2)

Finally, here is what the screen will look like after AMA has updated it:

![Figure 8.3 The display of figure 8.2 after updating.](image3)
The dotted lines in the preceding figures are guidelines: they serve to locate the middle of the window.

Commands issued to the wave table editor take a form slightly different from those of other modules. If the currently selected device is the crosshair, then the single character commands may simply be typed. Do not, however, type a carriage return after entering the command (although this is necessary in all other modules). The same holds true if the selected device is the light pen. In this case, however, it is necessary to enter a 'hit' after issuing the command. Make sure that your 'hit' is somewhere within the wave table window display.

**Commands**

**T** The T command will toggle the currently selected device from the light pen to the crosshair and vice versa. The name of the currently selected device (light pen, crosshair) will appear in the bottom rectangle of the screen.

**G** This command is used to toggle guidelines on or off. These guidelines are the dotted lines within the screen windows.

**E** This command will return you to the main Objec Module.
9. Envelope Graphic Editor

Introduction

Although envelope strings may be edited using a string editor, the envelope graphic editor, Plate VII, provides a more convenient way of modifying envelope contours. Many envelopes may best be constructed using the two editors: the string editor for basic entering of raw data, then the graphic editor for polishing and checking the data.

Like the wave table editor, the envelope graphic editor uses two interactive input devices: the light pen and the crosshair. When you enter this submodule, the crosshair will be enabled. To enable the light pen, type T, for Toggle device, and the crosshair will disappear from the screen. The input device will then be the light pen. To deactivate the light pen and reselect the crosshair, type T again, then press the pen to the screen, entering a 'hit'. If the hit is within either of the two windows displayed by the editor, the crosshair will reappear.

To enter a hit with the crosshair, move the crosshair to the desired position, then press the space bar. To enter a hit with the light pen, press the light pen to the screen at the desired point. Both the crosshair and the light pen each have their advantages and disadvantages. The crosshair can be positioned with great accuracy, but it is slow to move across the screen. The light pen lacks accuracy, but has none of the time delay problems associated with moving the crosshair.

The editor display consists of two windows, each of which displays the envelope string/time string pair being edited. The lower window, labelled 'panorama', provides a panorama of the entire envelope contour. The scale of this graph will change as the envelope is edited, growing or shrinking as the case may be. No matter what the length of the envelope, the entire envelope will always be visible in this window. The upper window is labelled 'close up', and always provides a 768 millisecond close-up view of some portion of the envelope. The portion of the envelope displayed in this close-up view is indicated in the panorama window--it is that area between the two arrows ->< and <-.<

To move the close-up window to display any desired portion of the envelope, the M command is used. If the input device is the crosshair, position the crosshair in the panorama window at the point where you wish the left-hand side of the close-up window to be displayed, then type an M. The editor will then
redraw the close-up window for you. If the input device is the light pen, first type the M command, then deliver a hit at the desired position in the panorama window. The two arrows will move to indicate exactly what portion of the envelope contour is currently displayed in the close-up window.

There are many commands available for modifying an envelope contour, but they are all issued in the same manner as M. If the input device is the crosshair, position the crosshair first (using the cursor movement keys), then enter the single character command. If the device is the light pen, type the character first, then press the light pen to the screen at the appropriate position.

Commands

C The change command is used to change a time/data pair forming part of an envelope contour graph. On the graph, any given straight line segment is defined by its rightmost endpoint. To change this point, issue a hit (with the C command) in the close-up screen somewhere along the length of the line segment. The close-up screen will then go blank, and you may now enter a new point. This point will replace the point which defined the old line segment, changing a value in both the data string and the time string being edited. If the input device is the light pen, then it is not necessary to enter a character before issuing this hit. If the device is the crosshair, then position the crosshair first, then press the space bar (or any other character which is NOT a single character command).

D The delete command will remove a line segment from the contour graph. This involves removing one value from the time string and one from the data string being edited. To use the command, issue a hit in the close-up screen somewhere along the length of the line segment defined by the point (line segments are defined by their right-hand end point).

I The insert command is virtually identical to the change command, C, except that the new point will not replace the old point, but will instead be inserted before this point.

M The move command is used to move the close-up window to display a different part of the envelope contour. Issue a command hit at the appropriate place in the panorama window. This place will locate the left-hand side of the close-up display.
R The R command stands for rubber band— one of the classic techniques of interactive graphics editing techniques. To understand its effects, imagine that the envelope contour graph is made of a rubber band stretched around thumb tacks. This command allows you to modify the contour by putting a new thumb tack into the graph, causing the rubber band to be stretched around this new point. The length of the envelope will remain unchanged by this command. To accomplish this, AMA will insert a new point into both time and data strings as well as shortening the time values of the points both before and after the new point.

T This command is used to toggle the current input device from the light pen to the crosshair or vice versa.

E The exit command returns you to the main Objed Module.
10. The S-event Editor

Introduction

The S-event editor, Plate V, has been designed to facilitate the creation and modification of objects of the S-event type. It implements a graphic display of the S-event in the form of a tree structure, with the leaves of the tree representing those objects which make up the S-event. With each leaf the user may associate the name of some other object.

The following diagram shows the state of a typical S-event called 's1':

![Diagram of S-event 's1'](image)

Figure 10.1 An S-event called 's1'.

In AMA, the x's in figure 10.1 will appear on the screen as the lower-case Greek letter omega. An omega character labelling an S-event field signifies that that particular S-event field has not as yet been labelled by the name of an object. To replace the character with the name of an object, type the 'pathname' from (but not including) the root of the tree through to the omega character, typing the name of an object in place of omega. A single blank is used to separate the names in such a pathname. To illustrate, referring to the S-event of Figure 10.1, an object called 'env100' can be attached to the carrier envelope's data field by typing the following:

```
ce da env100
```

If there is an object on the current TAGlist called env100, then AMA will respond by redrawing the S-event with 'env100' replacing the omega character (see Plate V).
Upon receiving a pathname, AMA performs certain error checks before it redraws an S-event. It first ensures that the pathname is free from spelling errors. Next, it searches the current TAGlist to be sure the named object really exists. Finally, it makes sure that the object is of the right type to be linked into the S-event field that the pathname specifies. It is, for example, illegal to link a time string into a slot for a data string, or to link an envelope data string into a slot for a frequency data string. For a list of all the S-event fields and the types of objects which must fill them, refer to chapter 6, page 16.

Commands

<ctrl D> Issuing this command allows you to examine the current state of the TAGlist. You will be prompted for an object type, then the names of all objects of the given type currently in the workspace will be displayed.

<ctrl E> Issuing this command will return you to the main Objed screen. Simply typing <RETURN> will have the same effect.

<ctrl P> This command provides a convenient path to module Play without the necessity of passing through the module Interchange. After issuing this command, the prompt character will change from '>' to 'P>' to indicate that everything typed is being read by module Play, not the S-event editor. Other than this however, the computer screen will not change: the S-event being edited will remain on the screen.
11. The T-event and TEtree Editors

Introduction

The TEtree concept is the most innovative facet of the AMA system. A TEtree is an object composed of T-events and S-events linked together to form a tree graph. This graph serves to locate S-events in time: to schedule their performance. It allows S-events to be played successively, simultaneously, or with any desired degree of overlap.

Since a TEtree is made out of T-events and S-events, you must have a set of such objects at hand before the tree may be constructed. S-events may be created as explained in chapter 10. T-events, unlike S-events, have an extremely simple structure. A T-event is simply a label, or name, together with an associated number, representing time in milliseconds. Creation and editing of T-events is done in the main Objed module.

To create a T-event, enter the main Objed module and type the T-event's name. An object called, say, 'long', can be created and given the type T-event by typing the following sequence:

```
>long
Enter Type>te
Value = 00000
Change(y\n)?
```

A 'y' response to the question 'Change(y\n)?', will cause AMA to prompt for a new time value. The next time the name 'long' is typed, AMA will remember its type, proceeding directly to the 'Change(y\n)' portion of the editing sequence.

T-event time values have a minimum and maximum practical size. The largest value a T-event may meaningfully take on is 32767 milliseconds. At the opposite end of the scale, the smallest practical value a T-event may have is 15 milliseconds (though one special T-event called the root may have a value as low as 0 milliseconds).

Having created a set of T-events and S-events for the purpose of making a TEtree, the user may invoke the TEtree editor (see Plate VIII). <ctrl T>, if issued while in the main Objed Module (not one of its submodules), will invoke this editor.
The TEtree has the structure of a tree graph, with the vertices (branching points) labelled by T-events, and the leaves (or ends of the branches) labelled by either T-events or S-events. A special vertex, the root, is labelled 'r', and is provided automatically whenever AMA creates a new workspace. This object is nothing special: it is just another T-event, and it can be seen on the current TAGlist by issuing the <ctrl D> command and requesting a display of TE-type objects. This 'r' object is fixed: r will always be the root of the TEtree. The rest of a TEtree structure must be built by the user.

A TEtree is constructed by issuing commands which take the form of the 'rewrite rules' or 'productions' of mathematical linguistics. To attach two T-events called 'long' and 'short' to the TEtree, type the following line:

/r long short

This is an example of a 'production'. AMA will interpret the first field of the production, 'r', as the point in the tree at which 'long' and 'short' will be rooted. Thus, typing the above command changes the screen display from:

- r ——— 00000

to:

/r——— 00000

[long ——— 20000 ——— short ——— 00010

Figure 10.1 Two TEtrees.

The numbers in the TEtree represent the time values associated with their respective T-events, as set in the main Objed module. These numbers are used by AMA to designate the time at which the objects in the tree will be played. If time is measured from the beginning of the composition, then the sum of all those numbers on the path from the root, /r, to any object in the tree represents the time, in milliseconds, at which that object will be performed by AMA. Of course, if that object is a T-event, there is nothing to hear. On the other hand, if the object is an S-event, then that S-event will be performed at the designated moment.
The procedure for modification of the TETree can be described symbolically as follows:

\[
\text{pathname} \rightarrow \text{S-event} \\
\text{pathname} \rightarrow \text{T-event}^+ \\
\text{pathname} \rightarrow \text{pathname}
\]

Where the names and symbols have the following meaning:

- **pathname** is the name of all the T-events in the currently displayed tree on the path from the top of that tree to another T-event, the 'target' object. If a pathname appears to the left of the arrow, '->', it means that the target object represents the 'left-hand side' of the production, and is the object to which all the other objects in a production will be linked. (This arrow appears here only to make the concept of left-hand side and right-hand side clear. In AMA, the arrow never actually appears in a production). If a pathname appears on the right-hand side of the arrow, it signifies that the target object designated by that pathname will be linked into the tree as a 'descendant' of the left-hand-side target object.

- **S-event** is the name of an S-event on the current TAGlist.

- **T-event** is the name of a T-event on the current TAGlist. This T-event will become a 'descendant' of the target object on the left-hand side of the production.

- **+** The plus sign superscript, '+', signifies that the symbol to which it is attached may be repeated as many times as desired, though it must be chosen at least once. To illustrate, the symbol 'X+' stands for any string of one or more 'X's, such as X, X X, X X X, X X X X X, and so on.

- **->** This arrow serves only to separate the left-hand side of the production from its right-hand side. In AMA, the arrow is not typed: it only appears here to make the diagram clearer.

Less formally, a production consists of a left-hand-side pathname followed by either the name of a single S-event, a pathname, or a list of T-events. All these objects will become the descendants of the object named by the left-hand-side pathname.
A pathname, in AMA, has a slightly different format from that of the Synclavier II operating system's catalog structure. First, the symbol '/' replaces XPL's ':' (it is easier to type). Second, a pathname must always begin with this symbol. Thus, the following is illegal:

r/section1/motiv1

and should appear as:

/r/section1/motiv1

Note that according to the above format, it is an error for an S-event to be on the left-hand side of a production. It is also an error for more than one S-event to appear on the right-hand side of a production. When an S-event is linked into the TETree, its name will show in inverse video (dark characters on a light background).

It is very difficult to display a tree structure like the TETree on a small computer graphics terminal. For this reason, AMA implements a unique way of representing the tree. First, only 3 'generations' of the tree may be seen at any one time. The first generation will be a single symbol. The second generation will be a list of symbols displayed horizontally (no more than 5 may be displayed) and attached to the first generation object. The third generation is displayed vertically, with elements joined by an angled line. The following illustration serves to make this scheme clear:

```
        r - 00000
       /\         /\         /\         /\
      long - 20000 - short - 00010 - long - 20000
     /\         /\         /\         /\
   a - 00010   a - 00010   a - 00010
  /\         /\         /\         /\
 b - 00100   b - 00100   b - 00100
 /\         /\         /\         /\
 a - 00010   a - 00010   a - 00010
```

Figure 10.2 A typical TETree.

Note that the TETree of figure 10.2 can be created by the following production sequence:
When a pathname appears on the right-hand side of a production, it does not represent just a single object. Instead, it represents the entire subtree rooted in that object. The purpose of this scheme is to allow the copying of entire subtrees in a single production. For example, the following production:

/r/long/short /r/long

has a pathname on the right-hand side. It will transform the TEtree of figure 10.2 into the following:

```
  r     00000
     /
    /   
  long 20000 short 00010 long 20000
    /
   /   
   a   00010   a   00010   a   00010
   /
  b   00100   b   00100   b   00100
   /
  a   00010   a   00010
```

Figure 10.3 The TEtree of figure 10.2 transformed by the production '/r/long/short /r/long'.

There is a possible ambiguity associated with AMA-style pathnames. Consider the following TEtree:

```
  r     00000
     /
    /   
  long 20000 short 00010
     /
    /   
    short 00010
```

Figure 10.4 A potentially ambiguous TEtree.
In figure 10.4, both T-events called 'short' have exactly the same pathname. AMA resolves this ambiguity, allowing the user to specify which of the two objects a path refers to in the following manner:

1) AMA searches downwards before it searches across. In the above example, the 'short' which is directly below the 'long' will be found first, thus it is to this event that the pathname '/r/long/short' refers.

2) If an object in a pathname is preceded by a double slash, '//' instead of a single slash, '/', then AMA will search across the tree before it searches downwards. Thus, to refer to the object to the right of 'long', use the pathname '/r/long//short'.

Although the graph of a TETree is only able to display 3 generations of S-events and T-events, this does not mean that TETrees are limited to this size. A TETree may grow to any depth, and any subtree of the TETree may be displayed. To display a subtree, simply type the pathname of the object you wish to become the new root of the display. The display will then change, with that object being displayed at the top. From now on, all pathnames must begin with this object, and not with the /r.

It follows from the above paragraph that it is only possible to move through a tree in a single direction — downwards. To move the display back towards the root, type the command <ctrl r>. This will move the display back to what might be called 'root position'.

There is an important point to be made about organizing a composition using the TETree. First, although there is no limit to how 'wide' a generation can be, there is a practical limit of five events across which is imposed by the size of the screen. If this limit is exceeded, the extra events can be heard, but not seen. There is nothing to gain by having more than five events across in the tree, for the same musical result can always be created by increasing the depth of the tree instead of its width.
Commands

<ctrl r> This command will cause the currently displayed TEtree to be redrawn with the root symbol, r, at the top. It is possible to change the name of this default root symbol by using the rename command of submodule Form. This will not, however, effect the format of this command--it will always be <ctrl r>.

<ctrl D> The display command is used to display a list of objects in the current workspace. You will be prompted for the type of object you wish displayed. If, for example, you wish to see a list of all the T-events in the current workspace, issue the command <ctrl D> and then, in answer to AMA's prompt, type TE. You will then see a list of all S-events so far created.

<ctrl E> The exit command will return you to the main Objed Module. Typing <RETURN> alone will have the same effect.

<ctrl P> This command provides a convenient path to module Play without the necessity of passing through the module Interchange. After issuing this command, the prompt character will change from '>' to 'P>' to indicate that everything typed is being read by module Play, not the TEtree editor. Other than this however, the computer screen will not change: the S-event being edited will remain on the screen.
12. Module Form

Introduction

Module Form might well be characterized as a submodule of the Objed module, for it is only accessible from Objed, and always returns to it. It is designed to help the composer form new objects, primarily by modifying existing ones in certain well-defined ways. A set of random number generation functions is provided. Mastery of the command functions of this module will go a long way towards eliminating the tedium of typing long lists of values to the string editors: in many cases, one of module Form's commands can be invoked to help build an object automatically.

The module includes many commands, all of which are summarized on the screen. Since the main purpose of Form is to execute these commands, it is not necessary to hold down the <ctrl> key before typing the single characters which invoke the command functions. When the commands of Form require information, such as object names, constants, etc., they will prompt you for the values in a conversational manner.

Many of Form's single-character commands are designed to modify objects composed of strings of values: envelope strings, volume strings, time strings, coefficient arrays, and wave lists. Such strings may be multiplied by a constant using the command T (Times), increased by a constant with P (Plus), multiplied by another string using M (Multiply), or added to another string with A (Add). Strings may also be created through random processes using the X (Xrand), Y (Yrand), and Z (Zrand) commands. The command S (Seed) puts AMA's random number generator into a known state. The commands J (Join) and F (Form) create new strings from old ones through catenary processes.

A second group of commands serve to manage the AMA workspace: U (Unsave) deletes objects from the workspace, C (Copy) duplicates objects, and R (Rename) changes their names.

Most of the commands of the Form module require the names of objects in order to function. The command M, for example, prompts the user for the names of the two strings it is to multiply together, then prompts for the name of a string in which to place the result. In most cases, the types of these strings is important. For example, it is an error to multiply a frequency string by an envelope string. For a complete list of the type admissibility imposed by the commands of Form, see figure 12.1, page 44.
Commands

A The Add command is used to add together two objects. The objects must have the same type, and only operations which 'make sense' are allowed: it is, for example, an error to add together two S-events. You will be prompted for the name of the two objects which you wish to add together (source1 object and source2 object), as well as the name of the object which will contain their sums (the destination object). Both the source objects must be on the current TAGlist, whereas the destination object must not. Form will add the two objects together pairwise, that is, the nth value of the first source object will be added to the nth value of the second source to produce the nth value of the destination object. The destination object will have the same length as either the source1 object or the source2 object, whichever is the shorter. If the result of the addition would result in a value which exceeds the allowable range for the given object type, then Form will truncate that value to the maximum.

C The Copy command is used to make an exact copy of an object.

D The Display command is used to display a list of objects in the current workspace. You will be prompted for the type of objects you wish displayed. If, for example, you wish to see a list of all the time strings in the current workspace, issue the command D and then, in answer to AMA's question, type TS. You will then see a list of all time events so far created.

E This command returns control to the main Objed module. Typing <RETURN> alone will have the same effect.

F The Form command of module Form is used to create a new object by concatenating an existing object with itself n times. If, for example, a frequency string is composed of the pitches C and D, then 'forming' the string with a value of n = 4 produces the string C D C D C D C D. The command is only defined on object types for which it 'makes sense': you cannot, for example, 'form' a wave table.

J The Join command will concatenate, or join together, two objects of the same type to form a third object.

M The Multiply command is identical to the A command, except that the source strings involved will be multiplied together instead of added together.
P The Plus command will produce a new string by adding a constant, \( n \), to every value of an existing string. The value \( n \) may be either a positive or a negative integer. If the result of the addition would produce a value which would be either too great or too small for the allowable range of the given object, then the value will be entered as either the greatest or the least allowable value.

R This command is used to rename an object on the existing tag list. All objects can be renamed, including the name of the root of the TETree, \( r \).

S Module Form contains a pseudo-random number generator used by the \( X \), \( Y \), and \( Z \) commands. Every time you enter module form, the generator will generate the same series of random numbers. If you wish to generate a different series, issue the \( S \) command, and enter some integer other than '12345' (this is the default seed value which the pseudo-random number generator will start at if you do not issue the \( S \) command).

T The Times command is identical to the plus command, except that the destination string will be formed by multiplying the source string times a constant. This constant may be either an integer or a floating point number (a number with a decimal point). Transposition by an equal-tempered semitone, for example, can be accomplished by multiplying a frequency string with the hertz attribute by the twelfth root of two, 1.05946.

U The Unsave command will delete an object from the current tag list. Any type of object can be deleted. There are a couple of points to note in connection with this command. First, if you delete an object which is currently linked into an S-event, then undefined operation might result. AMA will not check all current S-events to determine whether or not a given object is so linked. It will, however, notify you that the S-event contains an undefined field (one which points to an 'unsaved' object) the next time you enter the S-event editor for that S-event. At this time, the terminal will beep twice, and the S-event field will be filled with slash characters, '//////'. The second point to keep in mind is that if you delete the root of the TETree (originally \( r \), though this name may be changed by the \( R \) command), then the TETree will no longer be editable. Conclusion: don't do it!

X The Xrand command is used to create a new object out of pseudo-random variables uniformly distributed between a minimum and a maximum value. You will be prompted for the type of the object you wish to create, their minimum and maximum values,
and the number of variates to generate.

Y The Yrand command is identical to the Xrand command, except that the random variates generated will have the normal probability distribution. You will be prompted for a minimum and maximum value, the mean and standard deviation you wish the variates to exhibit, and the number of variates to generate.

Z The Zrand command is used to create a new object by randomly choosing elements from an object on the current TAGlist. The newly created object, then, will contain only the values of the source object, though in a completely different and randomly chosen order. This newly created destination object may contain as many values as you wish.
Type Admissibility

The following chart details the object types for which each of the command functions of module Form are defined. A '*' signifies that the given function is defined for the corresponding type, a blank signifies that it is undefined.

<table>
<thead>
<tr>
<th>type-→</th>
<th>TE</th>
<th>SE</th>
<th>TS</th>
<th>FS</th>
<th>HZ</th>
<th>PI</th>
<th>KE</th>
<th>VS</th>
<th>WL</th>
<th>WT</th>
<th>CA</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute-→</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>command</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 12.1 This figure shows which of module Form's functions are defined for which object types.
13. Theory of Operation

This chapter presents a brief technical description of the AMA system. This material will be of interest to those with a background in computer science who wish a deeper understanding of the ways in which AMA accomplishes its tasks.

AMA is written entirely in the XPL programming language as supplied by New England Digital Corporation (NED) for the Synclavier II computer music system. Procedures for driving the Synclavier's digital oscillators as well as a multi-tasking algorithm have been adapted from NED's MAX procedure library.

AMA coexists with the Synclavier II's operating system and catalog structure, though it does its own file management and input/output. The system is designed with expansion in mind: source code has been extensively commented and its presence on disk is considered an integral part of the system.

Catalog Structure

AMA files, when properly installed, will exhibit the following hierarchical catalog structure:

```
other system files
   
 AM  
   
 executable files
for all AMA modules
   SCRNS
   AMAMAX
   AMA.X
```

Figure 13.1 The catalog structure of the AMA system.

The catalog AMA contains the catalogs SCRNS, AMAMAX, and AMA.X, as well as the executable files for all AMA modules. It must be a subcatalog of the system catalog. SCRNS contains both source files and data files: the sources create the data files; the data files contain strings of bytes representing the various graphic screen layouts of the AMA modules. AMAMAX contains subroutine libraries for driving the digital oscillators, the terminal, and for disk input/output. Finally,
AMA.X contains the actual source code for all AMA modules. Compiling this code produces the executable files contained in the catalog AMA.

AMA files which contain source code all have the .x extension. All AMA procedures names as well as the names of many constants and variables are terminated by a period, '.'.

Chapter 14 describes a procedure whereby command files are invoked to create AMA's catalog structure automatically, then to load the AMA files themselves into the appropriate catalogs. A listing of these files, together with a brief description of their contents may be found at the end of this chapter.

System Organization

AMA's source code compiles into many separate programs: far too many to be resident in memory simultaneously. Instead, these programs, called modules, are linked together through a chaining algorithm. This scheme allows AMA to appear to the user as one large program when, in reality, individual modules are constantly swapped in and out of memory as they are needed.

A Module called CHAIN is responsible for doing the work of chaining together the various modules. CHAIN is a program which sits in high memory, loads modules into low memory, then transfers execution to them. It was created by first compiling the program 'CHAIN.X' to produce an executable file, saving that file under the name 'CHAIN', then running the program 'ADJUST'. Since NED's XPL compiler always produces code which executes at the same (absolute) address in low memory, the program ADJUST changes all necessary addresses in CHAIN so that CHAIN will run in high memory. The actual address that CHAIN runs at is defined in the file AMA.H. Any module can chain to any other module by first calling the procedure loadchain(filename) (defined in :AMA:AMA.X:SUBS.X), then executing the line 'GOCHAIN.' (defined in :AMA:AMA.X:AMA.H).

Just under the CHAIN program in high memory sits the AMA workspace--a data area extending down to the memory location at hexadecimal address 8000 (the computer is word addressable). All memory below 8000 hexadecimal is reserved for modules and for data locations local to those modules. In the lowest 256 words of the workspace sits an area which is dedicated to communication among the various AMA modules. These locations are all defined in :AMA:AMA.X:AMA.H using XPL's CORE function. To illustrate, the variable 'mempnt.' is defined as
core(COREBANK. + x), where COREBANK. is the starting address of the workspace and x is a displacement from this address. Any module which needs to refer to the variable mempt. is actually referring to this address.

Data Structures and Memory Management

AMA data structures are all defined in the file STRUCT.X. This file provides dynamic memory allocation by the procedures 'malloc.(words)' (memory allocator) and 'salloc.(words)' (string allocator). 'malloc.' allocates memory for data structures of the 'structured' type, and delivers memory from the bottom of the workspace upwards. 'salloc.' allocates memory for structures of the 'string' type—it delivers memory from the top of the workspace downwards. When the internal pointers of 'malloc.' and 'salloc.' meet, all memory has been used, an error message is printed on the terminal, and control is returned to the operating system.

All AMA objects have a correlate in an internal data structure. The following structures are of the 'linked list' type:

TAGs: For every AMA object, there is a TAG. These TAG structures contain the name of the object, the object's type, and the object's address in the workspace. A singly-linked LIFO list of TAGs is maintained, and the variable 'TAGlist.' points to the head of this list. A linked list of free TAGs is maintained.

S-events: S-events are not linked to each other: they are located in memory through their corresponding TAG. A linked list of free S-events is maintained. Each S-event contains fields for the addresses of the TAGs of the objects which it unites.

Wave Tables: Like S-events, Wave Tables are not linked to each other: AMA locates them by searching the TAGlist for their TAGs. Each wave table is 128 words long.

Tnodes: These structures are used by AMA to build the TEtree. A list of free Tnodes is maintained.

Stack memory: AMA maintains its own push-down stack for storing local variables during recursive subroutine calls. More specifically, a link list of stack pages is maintained.
All other AMA objects are of the 'string' type. There are two string formats: the single-word string and the double-word string. In both formats, the first word contains the number of single or double-word units in the string. The last word of a single-word string contains a -1 terminator (this word is not included in the first word's element count). AMA objects have the following string types:

- Time Strings........single word
- Volume Strings.......single word
- Coefficient Arrays...single word
- Wave Lists..........single word
- Position Strings.....single word
- Frequency Strings....double word
- Envelope Strings.....double word

STRUCT.X maintains strings very efficiently: when strings are deleted, the 'hole' that may be created in memory is removed by copying all strings below this hole upwards, then adjusting the addresses of their TAGs to reflect their new positions.

The AMA TEtree is built out of a binary tree structure of objects called Tnodes. Each Tnode contains the address of the TAG of either a T-event or an S-event, two link fields—one for the vertical successor and one for the horizontal successor, and one field identifying the vertical successor as being of either type SE or type TE.

Performance

When AMA performs the TEtree, it first goes through a short compilation process. This consists of recursively descending the TEtree, replacing all the TAG addresses of the S-events in the tree by the addresses of the S-events themselves. Since the fields of the S-events also point to object TAGs, not to the actual objects, these addresses are similarly changed.

The performance of a compiled score is accomplished by three main procedures. First, the tlpars, for 'top level parser', continually searches through the TEtree, looking for an S-event whose performance time has arrived. This tlpars is implemented as a task (tasks are procedures which appear to execute simultaneously). When an S-event whose 'time has come' is found, a new task, called 'llpars' for 'low level parser' is started. Up to 31 copies of llpars may be active at one time: one for each of the 31 oscillators which AMA uses. The llpars continuously examines the S-event, calling the 'subdispatcher' procedure whenever a particular time string
value indicates the moment has arrived to send a new data value to an oscillator. The subdispatcher is responsible for calling the appropriate procedure to actually send the value or values to the oscillator concerned.

AMA is designed with the idea of expansion in mind: musician/programmers are invited to improve upon the model. Of particular interest is the concept of the 'subroutine word' in an S-event. This word contains a bit for each of the S-event fields. If this bit is cleared, then the corresponding field is taken to be an AMA string. If the bit is set, however, then the corresponding field is taken to contain the address of a procedure. At present, AMA does nothing with such addresses. Their design purpose, however, is to allow procedure calls to replace strings of values in an S-event. These procedure calls could, for example, read I/O devices attached to the system, turning AMA into a real-time machine.

Disk Input/Output Procedures

The module :AMA:AMAMAX:DISPAK.X contains a group of procedures used by AMA for managing disk I/O. Files can be created, read, written, and deleted. DISPAK.X requires that any file which you wish to work with be first opened by the 'open.' procedure. This procedure returns a small integer, called a file descriptor, which is used by other DISPAK procedures when referring to that file. A block of information about each open file is maintained by DISPAK. Included in this block is a variable called the 'read/write pointer'--a character pointer into the file. When a file is read, it is read from the current position of this pointer, and the pointer advanced to the end of that read. When data is written to a file, it is written from the pointer's current position. When a file is first opened, this pointer is set to the end of the file. To move the pointer, call the 'seek.(file_descriptor, origin, offset)' procedure.

Graphics Procedures

The graphic displays of AMA are driven by a series of procedures found in the module :AMA:AMAMAX:GRAFIX.X. These procedures are designed for the Retro-Graphics enhanced Digital Equipment Corporation VT100 terminal with light pen enhancement. Of particular interest is the concept of the 'IMAGE'. GRAFIX.X allows strings of bytes representing graphic information to be assembled using IMAGE definitions. For example, to create a string of bytes which will clear the screen, then draw a horizontal line across the middle from left to right:
dcl demo fixed;
IMAGE demo = tovecmode + reset + lineto.(0,400) +
          lineto.(1023,400)
ENDIMAGE;

'demo' will now contain the address of a string containing the bytes necessary to do the job. When these bytes are sent to the terminal, they will accomplish the task:

    print string(location(demo));

The string 'demo' may also form part of another IMAGE definition through GRAFIX's 'setimage.' procedure. Thus:

dcl cross fixed;
IMAGE cross = setimage.(demo) + lineto.(500,0) +
           lineto.(100,789)
ENDIMAGE;

Since IMAGE definitions allow the precomputing of most graphics information, AMA makes extensive use of them. The programs found in the subcatalog :AMA:SCRNS are responsible for making groups of bytes which, when sent to the terminal, will create basic graphic formats. These programs all have the .m extension, for 'make', and the data files they manufacture have the .s extension, for 'screen'.

Files

The following is a list of all files comprising the AMA system, together with a brief description of their contents.

In catalog :AMA:AMAMAX (delivered on AMA disk 1)

| ASCII.H   | Definitions of the ASCII character set. |
| AMASYN.X  | Procedures for driving the digital oscillators. |
| AMAMAX.X  | More procedures for driving the oscillators. |
| AMATAS.X  | Multi-tasking procedures. |
| DISPAK.X  | Disk I/O procedures. |
| STRUCT.X  | Definitions of all data structures, procedures for memory allocation and management. |
| GRAFIX.X  | Graphics procedures. |
| DUMP.X    | A program for debugging providing facilities for examining and patching memory and disk. |
VERIFY A command file which verifies the integrity of all source files.

In catalog :AMA:AMA.X (delivered on AMA disk 2)

AMA.H Defines memory locations in the workspace.
AMA.X Module Interchange.
CAT.X Module Catalog.
LOSA.X Module Load/Save.
OBJED.X Main Objed Module, time event editor.
OBJSUB.X Procedure library for all Objed submodules.
OBJEDB.X S-event editor.
OBJEDC.X Wave list, position string editors.
OBJEDD.X Wave table editor.
OBJEDE.X Frequency string (PI attribute), coefficient array editors.
OBJEDF.X Frequency string (KE and HZ attributes).
OBJEDG.X Envelope string (non-graphic) editor.
OBJEDH.X Envelope string (graphic) editor.
OBJEDI.X Test tree editor.
FORMSUB.X Procedure library for Module Form.
FORM.X Module Form, first file.
FORMA.X Module Form, second file.
COMP.X Procedures for 'compiling' the workspace.
EMIT.X Procedures for performing the workspace.
CHAIN.X Chain procedure.
ADJUST.X Procedure for modifying addresses of CHAIN.

In catalog :AMA (delivered on AMA disk 3)

Executable versions of files in :AMA:AMAMAX

DUMP A program for debugging providing facilities for examining and patching memory and disk.

Executable versions of files in AMA:AMA.X

AMA Module Interchange.
CAT Module Catalog.
LOSA Module Load/Save.
OBJED Main Objed Module, time event editor.
OBJEDB S-event editor.
OBJEDC Wave list, position string editors.
OBJEDD Wave table editor.
OBJEDE Frequency string (PI attribute), coefficient array editors.
Frequenay string (KE and HZ attributes).
Envelope string (non-graphic) editor.
Envelope string (graphic) editor.
TEtree editor.
Module Form, first file.
Module Form, second file.
Procedures for 'compiling' the workspace.
Procedures for performing the workspace.
Chain procedure.
Procedure for modifying addresses of CHAIN.

Definitions used by all files with .M extension.
Module Catalog options.
Module Catalog pointer.
Module Form options.
Graphics grid for all string editors.
AMA logo in Module Interchange.
Module Interchange options.
Module Interchange pointer.
Module Load/Save options.
Module Load/Save pointer.
Module Objed.
Definitions common to modules interchange, catalog, and load/save.
14. Installing AMA

AMA is delivered on four eight-inch floppy disks. In order to run AMA, it must first be installed on the Synclavier II's Winchester disk: it cannot be run directly from the floppies. Installation involves creating an appropriate catalog structure on the Winchester disk system, then copying the files from the floppies into this structure. Command files found on the floppy disks accomplish this process automatically.

Execute the following steps to accomplish the installation process:

Step 1. Boot the Synclavier II operating system: insert the Winchester Bootload Disk in the leftmost (f0) drive, then press the reset button.

Step 2. Insert the floppy disk labelled AMA disk 1 in f0, then type the following line:

```
do install/f0
```

Note 1: There must be no file called AMA in the system catalog of the Winchester disk prior to performing this process. If AMA is already on the Winchester disk and you are replacing it from floppy disk, then type 'uns :AMA' before typing the line 'do install/f0'.

Note 2: AMA uses 2501 consecutive sectors of disk space. If there is insufficient space on the Winchester, an error message will appear, and the installation process will terminate.

Step 3. Follow the instructions which appear on the screen to install the other three AMA disks.

AMA may now be run by typing the following lines:

```
old :AMA:AMA
run
```