The Use of Hierarchy and Instance in a Data Structure for Computer Music

William Buxton, William Reeves, Ronald Baecker, and Leslie Mezei

The Structured Sound Synthesis Project
Computer Systems Research Group
University of Toronto
Toronto, Ontario
Canada
M5S 1A4

1. General

1.1 Introduction

One of the most important aspects in the design of any computer system is determining the basic data types and structures to be used. In making such decisions the main consideration is the manner in which the data must function in their intended application. In defining the data structures for the music system of the Structured Sound Synthesis Project, we have been guided by our projection of the interaction between the tool which we are developing, and the composer. In this regard, we view the composer's action as consisting of four basic tasks:

1. Definition of the palette of timbres to be available. This we call *object definition*, which is analogous to choosing the instruments which are to comprise the composer's orchestra. The main expansion on the analogy is that the composer also has the *option* to "invent" his own instruments.

2. Definition of the pitch-time structure of a composition, a process which we can call *score definition*. In conventional music, this task would be roughly analogous to composing a piano version of a score.

3. The *orchestration* of the "score". Generally stated, attaching attributes (such as *objects* defined in Step 1) to *scores* defined in Step 2.

4. The *performance* of the material developed thus far, whether an entire (orchestrated or unorchestrated) score, or simply a single note (to audition a particular object, for example)[1].

From the above taxonomy of tasks derives one of our first major decisions: to have two major data types, *objects* and *scores*, which relate to the sonic level and deeper structural level, respectively. Secondly–taking into consideration that composers work in different ways–an important consideration was to structure the system such that there be no order imposed on the sequence in which the user undertakes the above four tasks. Therefore, a composer is allowed to perform a score before it has been orchestrated, for example. The implication is that the system should be capable of coping with incompletely specified data. The obvious solution is to ensure that the low level structures can support an elegant system of defaults. Finally, it was seen as important to design the data structures so as to facilitate the definition of the *scope* of operators which the composer would be invoking to affect the data base. The composer must be provided with a "handle" onto his data which goes beyond the note-by-note approach prevalent in most systems today.

In the remainder of this paper, we shall present the design of a data structure which was developed in light of these considerations. We shall begin by giving a general background and motivation for the two main data types *(scores & objects)* and then proceed to present the details of the actual implementation.

1.2 Scores

1.2.1 The Hierarchical Representation of Scores

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. Therefore, systems which lean towards the "note" and/or "score" level are seen as largely inadequate in dealing with these middle level structures. Truax (1973) recognized this and his POD system was an attempt to deal with the problem. His approach, however, was based on the use of stochastic processes, and therefore assumes other problems of compositional programs. The problem of dealing with the different structural levels of a composition – from note to score – remain largely unresolved.

Two observations concerning the above provide the basis of our approach to the problem. First, what have hitherto been considered two extremes are seen as two instances of the same thing. Both deal with the composition "chunk-by-chunk". The only real difference is the size of the chunk: a note or an entire score. If we could provide a structure through which the composer could cause an operator (e.g., "play", "transpose", etc.) to affect any "chunk" of the composition – from note to score – we will have gone a long way in overcoming the problems of previous systems.

The key to allowing this "chunk-by-chunk" addressing lies in our second observation: that the discussion of structural "levels" immediately suggests a hierarchical internal representation for scores. Such a structuring of the data goes a long way towards enabling the specification of scope (definition of "chunks") of operators. A "play" command, for example, can affect a terminal node (single note) or some non-terminal node (thus causing the sub-tree or "sub-score" below that node to be played). The important point to note is that such a structuring of the data allows any "chunk" of a score to be treated *in exactly the same manner as a single note*; with the same ease and clarity, regardless of "chunk" size!

1.2.2. The Musical Event

Given the temporal nature of music, it is "natural" that we define a *score* as "an ordered sequence of musical events". What we mean by a "musical event", however, is central to an understanding of our hierarchic representation of scores.

By "musical event" we mean simply an event which occurs during the course of a composition which has a start-time and an end. Thus, the entire composition constitutes a musical event (the highest level), as does a single note (the lowest level). Similarly, chords, motives, movements, etc., are all musical events. In fact, any of the "chunks" – as described in the previous section – can constitute a musical event[2]. Thus, any musical event (e.g., a motif) can be made up of composite musical events (e.g., chords and notes); hence the basis for our hierarchy.

In considering the concept of a musical event, it is important to realize that the starting time of the next event is completely independent of the duration of the current one. Therefore, as we see in Figure 1, for example, the same two events (G4 and C5) can occur in sequence (Bar one) or parallel (Bar two), or in some combination of the two (Bar three).

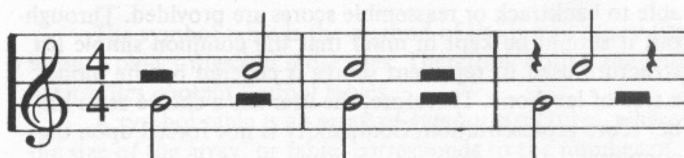

Figure 1. Temporal relationships between simple musical events.
Bar 1: sequence (melodic)
Bar 2: parallel (chord)
Bar 3: mixed

Similarly, we see in Figure 2, for example, that each of the four parts in a string quartet can be considered as a separate musical event (each made up of events of a lower level).

Allegro, ♩. = 110
Violino I
Violino II
Viola
Violoncello

Figure 2. High level musical events – an example. Each line of the quartet can be considered a single musical event. The events overlap in a relationship similar to Bars 2 & 3 in Figure 1.
(From Bartok, *String Quartet No. 4*, First Movement)

With the musical events, there are two autonomous notions of time: duration and entry-delay. The first is self-explanatory, and the second is the delay before the onset of the next event in the sequence. In melodic figures the two are equal. In a chord, the entry delay is equal to zero. The important thing to note is that in performance, for example, they can be modified independently or together. Changing both will vary tempo while adjusting the articulation proportionally. Adjusting duration independently of entry-delay will result in a change in the articulation of notes, for example. Thus, there is a great deal of potential for the "conducting" of a score built into the underlying structure.

We can express the notion of musical event as a simple grammar (where Mevent is an abbreviation for musical event):[3]

Composition	::= Mevent;
Mevent	::= Mevent* | Score | note;
Score	::= Mevent;
note	::= terminal (i.e., some musical note);

(The grammar is expressed in BNF, where "::=" means "is defined as", "|" means "or", "*" means "may be repeated".)

Besides the ability to isolate different components of the composition, this structure has the benefit that the tree structure actually represents a "recipe" of how the composition was put together. Thus the additional features of being able to backtrack or reassemble scores are provided. Throughout it should be kept in mind that the common simple list structure used to represent scores is covered by the model: a tree of level one. Therefore, the user has a choice as to his/her score representation. Complexity is not forced upon the composer.

1.2.3. Instantiation

Our choice of a hierarchic score representation makes possible additional features not yet discussed. Consider, for example, the common case where a composition is made up of certain base material which is then repeated, developed, transposed, etc. In this case, the score could contain several instances of a particular musical event, but each instance may be transformed in some way. One need only consider one of the examples in the literature of the "theme and variations" form to find a good illustration of this point. In terms of a tree structure, we see that this case could be described as there being more than one instance of a particular sub-tree. Where we can derive power from this observation is in stating that consequently, there should only be one *master-copy* of that sub-tree, and at each *instance* we store only the sub-tree identifier and the transformations to be effected for the particular instance[4].

There are a number of benefits to this approach. First, it is easy to isolate all instances of "motif A", for example. Second, the size of the score is reduced considerably, since only one copy of the motif is saved[5]. Third, it is clear that our file system and data structures must be able to treat any musical event as a free-standing self-contained structure; a sub-score. Therefore, any sub-score can be played, edited, etc., on its own. Most importantly, any change to the master copy of any sub-score in a composition will be reflected in *every* instance of that structure. Thus, if a re-occurring figure in our composition is an octave jump up, followed by a semitone fall, by simply changing the master copy of this figure to a major triad, all instances would be similarly affected *by this one action!*

1.2.4. Summary

In the preceding discussion, an argument has been made for the adoption of a hierarchically based internal representation of scores. Through this approach we can provide the basis for the composer's ability to address himself/herself (and his/her commands) to the "chunks" of the score with which he/she is concerned. Furthermore, through the use of instantiation we are able to exploit the redundancies inherent in musical structures and gain savings both in space and ease of operation.

1.3 Objects & Timbre Definition

1.3.1. Introduction

If we are going to synthesize sounds, we have an obvious interest in being able to control "timbre"; however, the nature of "timbre" for musical purposes is rather elusive. Traditional explanations (e.g. Helmholtz, 1954) have restricted their description to the physical (viz. acoustical) properties of sounds. Two things are clear, however: that ideally, timbre should be described in the *perceptual*, rather than acoustical domain; second, timbre is a multi-dimensional attribute of sound, such that the number of dimensions inhibits the understanding and control of the perceived phenomenon. Thus, our prime objective is to establish the underlying structures which will: (a) facilitate the implementation of different high-level external representations of our repertoire of timbres, and (b) support an effective editor for exploring the properties of the multi-dimensional attributes of this repertoire. Throughout, the intention is that initial work at the lower acoustical level will provide insights enabling us to develop a control mechanism functioning at the higher perceptual level. As our insights into the nature of timbre improve through experience and experimentation, we are able to refine our external representations accordingly.

In our approach the analogy to the timbre of a musical instrument is an *object* (after Schaeffer, 1966). By our definition, an object is: "a named set of attributes which will result in sounds having different pitches, durations, and amplitudes to be perceived as having the same timbre". In our definition, it is significant that we have stated nothing about the nature of those attributes constituting an object. The notion of an object simply provides a conceptual framework in which the composer can view his activities. All objects have a name and all instances of a particularly named object sound "the same"[6]. Conceptually, this is all that the composer need understand, plus the fact that there is an editor which will aid him/her in (a) controlling the palette of timbres—by defining and modifying his/her own set of objects, and (b) "orchestrating" the notes in a score from this set of objects.

In contrast to the SYN4B system at IRCAM (Rolnick, 1978), our approach to the problem is to take a few well-proven configurations of unit generators and "package" them so as to optimize on the ability of the composer to explore their full potential. Clearly this decision relates to the issue of strength *vs.* generality. Our choice is to take the more limited but strong approach. We are confident that research such as Moorer (1977) and Le Brun (1977) will help bring an ever-expanding repertoire of computer-based sounds to the repertoire of composers. Our prime concern is with the development of tools to aid the composer in controlling these sounds in a musical context.

Having adopted this approach, the problem is to select those instruments or *acoustic models* which we will support. In this decision, the prime considerations are: the range of the timbral palette, suitability to efficient implementation, ease of control protocol, and perhaps most of all, how well the model lends itself to the implementation of a user-congenial interface. Moorer (1977) gives a good survey of the alternatives. The models which we have chosen to support—each of which is implemented in hardware in the SSSP digital synthesizer (Buxton, Fogels, Fedorkow, Sasaki & Smith, 1978)—are: *fixed waveform, frequency modulation, additive synthesis, waveshaping,* and *VOSIM.*

2. Implementation

This section presents the implementation details of a data structure which conforms to the general description outlined in the previous section. A version of the structure has been implemented and utilized with success at the University of Toronto. An overview of this structure is presented in Figure 3. Here we see that there are four main types of structures, each of which constitutes a particular type of *file.* These are:

1. Scores
2. Objects
3. Functions
4. Waveforms

Each of these file types is made up of various composite structures. The purpose of this section, therefore, is to present the internal representation of each of these four types of files, and to define the methods of communication, or links, among these files.

Since it is a structure common to various file types, and since it is the prime medium of inter-file communication, we shall begin by presenting the form of the *symbol table* data structure[7]

3. Inter-file Communication

3.1. The Symbol Table

The symbol table is the method of linking auxiliary files to both object files and score files. Therefore, both score and object files contain symbol tables.

A symbol table is an array of *symbol* structures, where the size of the array, or table, corresponds to the number of symbols, or entries, in that table[8]. The symbol structures for a particular table are stored in contiguous memory. In the programming language "C", each entry in a symbol table has the following structure format:[9]

```
struct symbol
   {
   char name[FNAME_SIZE];  - File name
   int stype;              - Contains fields for type
                             of symbol.
   int svalue;             - Value of symbol
                             (may be pointer)
   };
```

The *name* field simply contains the name of the file associated with the symbol in question. The *stype* field then indicates the

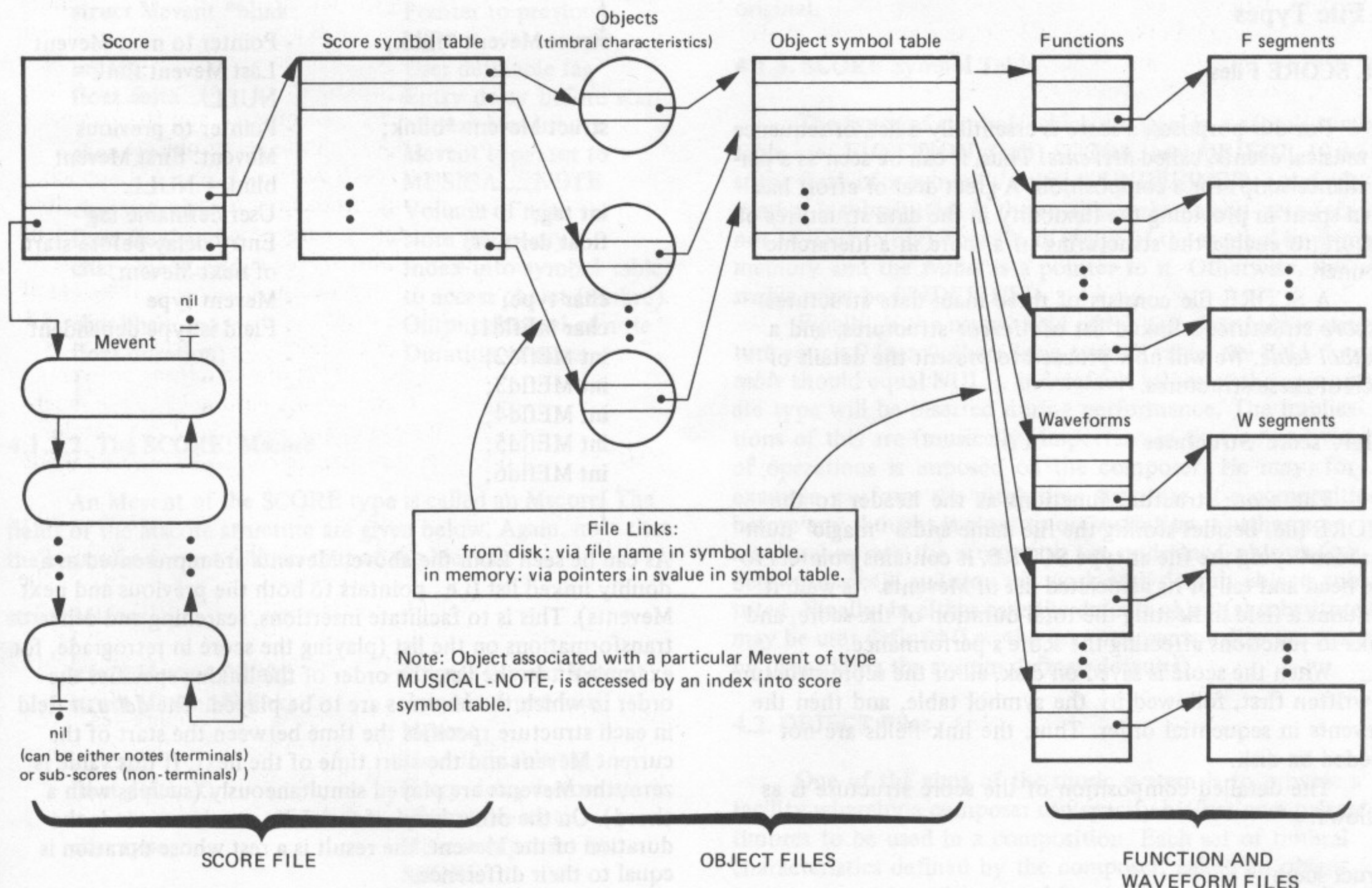

Figure 3: Overview of Data Structures

type of file this entry in the symbol table is. Valid symbol types include:

1. OBJECT
2. SCORE
3. FUNCTION
4. WAVEFORM

Each of these symbol types corresponds to one of the file types mentioned above, and will, therefore, be dealt with in more detail below. Finally, if the file in question is in primary memory, the third field of the symbol entry – the *svalue* – contains a pointer to the file's core image.

We see, therefore, that access to subordinate files is accomplished through a symbol table, via the *name* fields for files not in primary memory (i.e., those requiring system i/o), and via the *svalue* field for others (thereby avoiding the time-consuming i/o)[10]

NOTE: A particular symbol entry is accessed by providing an index into the table. An important convention to note in this regard is that the first entry in the table is accessed by an index of one (1) *not* zero (0). An index of zero into the symbol table has the special meaning that the symbol to be referenced is not yet defined; a default symbol of the appropriate type (context-dependent) is substituted. Thus, the mechanism for handling *default* situations is provided, the user never having to provide details beyond his current concern.

4. File Types

4.1. SCORE Files

For our purposes, a *score* is essentially a list, or sequence of musical events, called *Mevents*. Thus, it can be seen as a performance script for a composition. A great deal of effort has been spent in providing the flexibility in the data structures of a score to enable the structuring of a score in a hierarchic manner.

A SCORE file consists of three main data structures: a *score* structure, a linked list of *Mevent* structures, and a *symbol table*. We will now proceed to present the details of each of these structures.

4.1.1. 'score' Structures

The *score* structure functions as the header to the SCORE file. Besides storing the file name and a "magic" number identifying the file as type SCORE, it contains pointers to the head and tail of its associated list of Mevents. As well, it contains a field indicating the total duration of the score, and links to functions affecting the score's performance.

When the score is saved on disk, all of the score structure is written first, followed by the symbol table, and then the Mevents in sequential order. Thus, the link fields are not needed on disk.

The detailed composition of the score structure is as follows:

```
struct score
    {
    int magic;                    - Magic number
    char fname[FNAME_SIZE];       - File name
    int nsyms;                    - Number of entries in
                                    table
    struct symbol *sym_table;     - Pointer to symbol table
    float tot_dur;                - Total duration of score
    int nMevents;                 - Total number of Mevents
    struct Mevent *head;          - Pointer to start of
                                    Mevent list
    struct Mevent *tail;          - Pointer to end of Mevent
                                    list
    char dyn_ind;                 - dynamic (volume)
                                    variation
    char tempo_ind;               - tempo variation
    char deltat_ind;              - entry-delay (articula-
                                    tion) variation.
    };
```

The last three fields are indices into the SCORE symbol table accessing functions controlling global features of the score: dynamics, tempo, and articulation, respectively.

4.1.2. The Mevent

As stated above, an essential component of a score is a sorted list of musical events which we call Mevents. While there are various recognized types of Mevents allowable in this list, they all conform to the following structure template:[11]

```
struct Mevent
    {
    struct Mevent *flink;         - Pointer to next Mevent
                                    Last Mevent flink =
                                    NULL.
    struct Mevent *blink;         - Pointer to previous
                                    Mevent. First Mevent
                                    blink = NULL.
    int tag;                      - User definable tag
    float delta_t;                - Entry delay before start
                                    of next Mevent.
    char type;                    - Mevent type
    char MEfld1;                  - Field is type dependent
    int MEfld2;                   -         "
    int MEfld3;                   -         "
    int MEfld4;                   -         "
    int MEfld5;                   -         "
    int MEfld6;                   -         "
    };
```

As can be seen from the above, Mevents are represented as a doubly linked list (i.e., pointers to both the previous and next Mevents). This is to facilitate insertions, searching and other transformations on the list (playing the score in retrograde, for example)[12]. In the list, the order of the linking specifies the order in which the Mevents are to be played. The *delta_t* field in each structure specifies the time between the start of the current Mevent and the start time of the next. If this value is zero, the Mevents are played simultaneously (such as with a chord). On the other hand, if the *delta_t* value exceeds the duration of the Mevent, the result is a rest whose duration is equal to their difference.

The currently available types of Mevents (as specified in the *type* field) are: MUSICAL_NOTE, and SCORE. An Mevent of the MUSICAL_NOTE type is simply a single sound

event. A SCORE-type Mevent is just that, a (sub-)score which commences at a particular point in a composition. It is this implementation of the notion of *sub-score* which enables us to create scores which are hierarchically structured. More formally, we can view a score as a tree structure in which a SCORE Mevent (called Mscore) constitutes a non-terminal node, and each MUSICAL_NOTE Mevent (called Mnote) constitutes a terminal, or "leaf", in the tree.

In the above structure, one feature is of particular note. This is the choice of using "delta" rather than "absolute" values for time (i.e., the entry delay value *delta_t*). This choice is based on the ease with which several instances of the same sub-score can be merged into another "master" score.

Since the interpretation of the Mevent structure fields MEfld 1-6 are dependent on the Mevent type, we shall now consider the individual types in more detail.

4.1.2.1. The MUSICAL_NOTE: Mnote

An Mevent of the MUSICAL_NOTE type is a single sound event which has certain characteristics (or parameters) as defined by the following Mnote structure. (Note that this structure exactly follows the template of the Mevent structure.)

```
struct Mnote
    {
    struct Mevent *flink;      - Pointer to next Mevent
    struct Mevent *blink;      - Pointer to previous
                                 Mevent
    int tag;                   - User definable tag
    float delta_t;             - Entry delay before start
                                 of next Mevent.
    char type;                 - Mevent type: set to
                                 MUSICAL_NOTE
    char volume;               - Volume of note
    float frequency;           - Note frequency
    char object_ind;           - Index into symbol table
                                 to access object (timbre).
    char chan_no;              - Output channel of note
    float duration;            - Duration of note
    };
```

4.1.2.2. The SCORE: Mscore

An Mevent of the SCORE type is called an Mscore. The fields of the Mscore structure are given below. Again, note that the structure format follows that of the Mevent.

```
struct Mscore
    {
    struct Mevent *flink;      - Pointer to next Mevent
    struct Mevent *blink;      - Pointer to previous
                                 Mevent
    int tag;                   - User definable tag
    float delta_t;             - Entry delay before start
                                 of next Mevent.
    char type;                 - Mevent type: set to
                                 SCORE
    char vol_factor;           - Relative shift for sub-
                                 score. (vol. is log, so is
                                 addition)
    float pitch_trans;         - Relative shift for sub-
                                 score. (i.e., pitch trans-
                                 position)
    char score_ind;            - Index into symbol-table
                                 to access sub-score.
    char time_interp;          - Determines if time_fac-
                                 tor affects entry_delay,
                                 duration, or both
    float time_factor;         - Temporal transformation
                                 (augmentation/diminu-
                                 tion) of sub-score.
    };
```

There are a few very important points to note regarding the use of sub-scores. First, note that each appearance of a particular sub-score constitutes an *instance* rather than *master copy* of that sub-score. The difference is that there is only one master copy of the sub-score (accessed through the symbol table) and any changes to the original are reflected in each instance during a composition. Therefore, if we view a score as a tree structure, then the *pitch_trans, vol_factor,* and *time_factor* fields of the Mscore structure will effect transformations on the sub-score (or sub-tree) below them. Musically, therefore, these fields allow for the occurrence of the sub-score starting at any pitch (i.e., transposition), the dynamics to be scaled, and the augmentation and diminution of the time structure[13]. The result is that we can obtain several versions of a single "score", while maintaining only one copy of the original.

4.1.3. SCORE Symbol Table

The types of symbols which are legal in a score's symbol table are: FUNCTION, (sub) SCORE, and OBJECT. If the stype field of a symbol's entry is UNDEFINED, a default symbol is substituted. If the entry's *svalue* is non-zero (viz., not UNDEFINED), there is an image of the symbol in primary memory and the *svalue* is a pointer to it. Otherwise, the *svalue* must be UNDEFINED.

Finally, if the nsyms field of the associated score structure equals 0 (zero), there is no symbol table, the field **sym_table* should equal NULL, and default values of the appropriate type will be inserted during performance. The implications of this are (musically) important in that no ordering of operations is imposed on the composer. He may, for example, perform the pitch/time structure of a composition before any thought is given to orchestration. Furthermore, he may orchestrate the score with yet undefined objects (see below), and still audition the work with default objects substituted. Finally, in either case the default object(s) substituted may be user defined (i.e., the user may personalize the system by over-riding the system defined defaults).

4.2. OBJECT Files

One of the aims of the music system is to provide a facility whereby a composer can specify his/her own palette of timbres to be used in a composition. Each set of timbral characteristics defined by the composer, called an *object*, is then stored in a file named by the composer. Notes in a score may then be "orchestrated" by establishing an association with a particular object file. This is accomplished via the

object_ind field of the Mnote structure, in combination with the score symbol table (as outlined previously).

We saw above that there may be several instances of the same (sub-)score in a composition. Similarly, there may be numerous Mnotes of various durations, pitches, and amplitudes, all deriving their timbral characteristics from the same object. Furthermore, any change of the object file will cause that change to be reflected in all instances of that object in a score. We see, therefore, that the object functions as a type of timbral "template". Finally, due to this template nature of the object, the only restriction on how many instances of that object which may occur simultaneously is the number of oscillators in the synthesizer. This is in contrast with the notion of "instrument" as developed in MUSIC IV (Mathews, 1969), for example.

While all objects serve the same musical purpose of timbral control, there exist different internal representations for object data. These differences primarily reflect the different modes—or acoustic models—whereby sound can be generated by the SSSP digital synthesizer. We will see, therefore, that there are three main data structures in an object file. These are: the *object* structure and *symbol* (table) structure (both common to all objects, regardless of mode); and the *type_object* structure, which contains the data peculiar to the mode of that particular object.

4.2.1. 'object' Structures

The object structure contains information common to all objects, regardless of mode. Such information includes the objects' name, mode, and a "magic" number to distinguish OBJECT files (from, for example, SCORE files) during various operations such as reading and writing. The structure also specifies the number of critical resources (i.e. synthesizer oscillators) required by that object. This information is represented as follows:

```
struct object
    {
    int magic;                    - Magic number
    char fname[FNAME_SIZE];       - File name
    int nsyms;                    - Number symbols in table
    struct symbol *sym_table;     - Pointer to symbol table
                                    nsyms long.
    int mode;                     - Designates type of object
    int noscils;                  - Number of oscillators
                                    needed.
    union type_object
      struct fixedwf_object fwfobj;
      struct fm_object fmobj;
      struct bank_object bankobj;
      struct ws_object wsobj;
      struct vosim_object vosimobj;
          *data;                  - Defines "*data" as a
                                    pointer to an object.
    char rigidfunc_ind;           - Index into symbol table
                                    to access basis of func-
                                    tion time-scaling.
    };
```

One field of the object structure warrants special attention. This is *rigidfunc_ind*. As will be seen below, each mode of object specification includes the specification of functions which determine how parameters vary over time. The time base of such functions, however, must be able to be scaled over Mevents (e.g., Mnotes) of various durations. This is in keeping with the notion of an object being a general template for timbre. One problem is, however, that in compressing and expanding functions we do not always want the scaling to be linear. That is to say, if we consider the x (or time) axis of a stored function as a spring, we do not always want the spring to be of uniform stiffness. In imitating sounds which occur in nature, for example, we would want the attack and decay portion of the amplitude function to be more "stiff" than the steady-state. Similarly, in other objects we might want just the opposite. In view of this problem, each object has associated with it a user-definable "rigidity" function, which determines how the functions of that object are to be scaled—in time—in their various instances throughout a composition. The *rigidfunc_ind* field provides, therefore, an index into the symbol table which identifies the "rigidity" function for that object.

4.2.1.1. Object Types

As was stated above, there are different methods of representing objects which reflect the method of sound synthesis used. These modes are as follows:

1. Fixed Waveform (FIXEDWF)
2. Frequency Modulation (FM)
3. Pulse Modulation (VOSIM)
4. Additive Synthesis (BANK)
5. Waveshaping (WAVESHP)

The amount and type of data required is different for each of these modes. Therefore, there is a different type of structure used for each. The definition of the structure peculiar to each object type is given below. The appropriate structure for a particular object's type-specific data is accessed *via* the **data* field in the *object* structure, whose *mode* field indicates the structure's type.

4.2.1.2. FIXEDWF Objects

The fixed waveform synthesis mode utilizes a single oscillator as a function generator. The only parameters at the object level in this mode are: the waveform used, the amplitude contour (or "envelope"), and the frequency contour (or deviation over time). The amplitude and frequency contours are stored functions (see FUNCTION files, below) and are accessed through the object symbol table. The format for FIXEDWF data is as follows:

```
struct fixedwf_object
    {
    char fwf_ind;        - Index into object symbol
                           table to define wave-
                           form.
    char envel_ind;      - Index into object symbol
                           table for amp. function.
    char freq_ind;       - Index into object symbol
                           table for freq. function.
    };
```

4.2.1.3. FM Objects

The FM mode of object specification enables sound to be synthesized by having one oscillator ("m") modulate the frequency of another (the "c", or carrier oscillator). The resulting relevant parameters include: the ratio between the frequencies of the two oscillators (the "c:m" ratio), the maximum degree of modulation and how modulation varies in time, and the amplitude and frequency contours (as seen with FIXEDWF objects). The format of FM mode data is as follows:

```
struct fm_object
    {
    struct fixedwf_object car;   - Carrier waveform (as in
                                   FIXED_WAVEFORM)
    char mfwf_ind;               - Index into object symbol
                                   table defining mod.
                                   waveform.
    char mdev_ind;               - Index into object symbol
                                   table for mod. function.
    int maxindex;                - Max. Modulation Index
    int cval;                    - C term in C:M freq.
                                   ratio.
    int mval;                    - M term in C:M freq.
                                   ratio.
    };
```

4.2.1.4. VOSIM Objects

The VOSIM mode enables voice-type synthesis via a form of pulsewidth modulation. There are different degrees of complexity possible; generally, the more complex, the more oscillators or "VOSIM functions" must be used. Besides the pulse-shape (waveform) select and the amplitude and frequency functions, each VOSIM function also has the following parameters: the pulse-width, how the pulse-width varies in time, and the degree of randomization (to produce consonants, or noisy spectra). The format for the VOSIM data is as follows:

```
struct vosim_object
    {
    struct fixedwf_object vosfn; - As in FIXEDWF.
    char maxdev;                 - Maximum deviation (i.e.,
                                   noise) factor.
    char dev_ind;                - Index into object symbol
                                   table for dev./time func-
                                   tion.
    char pw_ind;                 - Index into object symbol
                                   table for pulse-width
                                   (i.e., formant) change
                                   function.
    int pwf;                     - Pulse-width expressed as
                                   frequency.
    };
```

Note: Complex VOSIM objects utilize more than one VOSIM function or oscillator. When this is the case a table of vosim_object structures is kept in a contiguous portion of memory. The number of entries in this table is given by the *noscils* field in the parent *object* structure.

4.2.1.5. BANK Objects

This mode enables the use of several generators together, such that each oscillator functions as one component, or partial, in a complex tone. The frequency and amplitude of each component may vary over time. The actual frequency of any component is its partial number times the fundamental frequency (where the fundamental frequency is considered partial number 1). The data for the various partials in a particular object are stored in a table of *bank_object* structures. The format of these table entries is given below. The number of entries – which are stored in contiguous memory – is given by the *noscils* field of the object structure.

```
struct bank_object
    {
    struct fixedwf_obj bnkmd;    - As in FIXEDWF.
    float partial;               - Partial number (fund.
                                   = 1).
    };
```

Note that the partial number is specified as a "float" value so as to enable arbitrary partial structures.

4.2.1.6. WAVESHP Objects

Waveshaping is a technique which enables the synthesis of complex sounds having time-varying spectra. The technique makes use of a form of controlled non-linear distortion. Essentially, the output of one oscillator is scaled by an index (which may be a time-varying function), and then used as an address into a look-up table. The sample taken from the table (which contains the "distortion" function) is then used as a waveform sample and therefore scaled in amplitude and sent to a digital-to-analogue converter. The technique utilizes two oscillator modules, and has its parameters stored in the following structure format:

```
struct ws_object
    {
    char fwf_ind;                - Index into object symbol
                                   table to define waveform
                                   previous to distortion.
    char envel_ind;              - Index into object symbol
                                   table defining envelope
                                   of waveform after distor-
                                   tion.
    char freq_ind;               - Index into object symbol
                                   table for frequency func-
                                   tion.
    int dindex;                  - Index of distortion to
                                   which dist_ind function
                                   is scaled.
    char dindfn_ind;             - Index into object symbol
                                   table specifying time-
                                   varying function for
                                   dindex.
    char distfn_ind;             - Index into object symbol
                                   table indicating wave-
                                   form buffer containing
                                   distortion function.
    };
```

4.2.2. OBJECT Symbol Table

The only valid symbol types for object symbol tables are: FUNCTION and WAVEFORM. Functions at the object level provide the means of specifying how parameters of the micro-structure vary in time. This is essential for sounds to be of musical interest.

Just as with the score symbol table, if object.nsyms equals 0, or if any function named in the table is UNDEFINED, default functions will be substituted. Again, the user is able to override the system defined defaults.

4.3. FUNCTION Files

Stored functions are used throughout the various hierarchies of the music system to control the variation of parameters over time. Just as there may be many instances of the same score or object in a composition, there may be several instances of the same function. In addition, each instance of the same function could quite conceivably be affecting a different parameter. Functions are stored as a set of straight line segments which approximate a continuous curve. Each file has a *unique, user defined name.* In performance, each function is scaled in both the x and y domains according to application. Since there is no set number of segments in a function, we resort to using two different types of structures for their representation. These are outlined below.

4.3.1. 'function' Structure

This is the "header" structure of a function. It contains the function's name, a "magic" number to identify the file as type FUNCTION, the total number of segments, and a link to the segment data. The actual format of the structure is as follows:

```
struct function
    {
    int magic;                     - Magic number defining
                                     function
    char fname[FNAME_SIZE];        - File name
    int nsegs;                     - Number of segments
    char starty;                   - Initial y value
    struct segment *breakpoint;    - Pointer to breakpoints.
                                     Functions as:
                                     breakpoint[nsegs]
    };
```

4.3.2. 'segment' Structure

The data for the actual segments is stored in a table of *segment* structures. There is one structure for each segment and all structures are in contiguous memory. Rather than represent segments by integer breakpoints, we have chosen a slightly different approach which is computationally more efficient (when scaling functions during real-time performance). Simply, the y value is stored as would be expected, but the x value is stored as a fractional value representing that segment's relative duration with respect to the complete function. The format of the structure is as follows:

```
struct segment
    {
    float reldur           - Relative duration of
                             segment
                             (0.<=reldur<=1.)
    char yval;             - End Y value of segment
    };
```

4.4. WAVEFORM Files

Waveforms are a particular form of function which we choose to treat differently than FUNCTION files. In the case of a waveform, we store the function as a series of point samples, where the number of points equals the size of a synthesizer waveform buffer (i.e., 2048). Consequently, only the y value of the function need be stored, the x value being the index into the table. Besides storing the actual function data, the *waveform* structure also contains a "magic" number to distinguish it as type WAVEFORM, and the actual waveform name. The data format for waveforms is:

```
struct waveform
    {
    int magic;                    - Magic number.
    char fname[FNAME_SIZE];       - File name.
    int wfsamp[WFB_SIZE];         - Waveform samples.
    };
```

5. Conclusions

It should be emphasized that this research is not to be taken as a bit-by-bit prescription for others to emulate. The more important aspect of this work is to be found in the overall approach to musical data structures. The details of these systems are constantly evolving and changing. If readers are interested in hearing about subsequent changes as they occur, they are invited to contact the authors.

6. Acknowledgments

The work described in this paper has been undertaken as part of the research of the Structured Sound Synthesis Project (SSSP) of the University of Toronto. The SSSP is an interdisciplinary project whose aim is to conduct research into problems and benefits arising from the use of computers in music composition. This research can be considered in terms of two main areas: the investigation of new representations of musical data and processes, and the study of human-machine interaction as it relates to music.

The research of the SSSP is funded by the Humanities and Social Sciences Research Council of Canada, while logistic support comes from the Computer Systems Research Group (CSRG) of the University of Toronto. This support is gratefully acknowledged.

7. Notes

1. We include performance as part of the compositional process based on the opinion that a piece of music is not completed until it is heard. While some theorists would dispute its need, we would argue that composers of

conventional music have always had such aural feedback—in the mind's ear—as enabled by a familiarity with the long tradition of western music; a tradition which does not exist for the composer of contemporary music.

2. This notion of an event being either a simple sound or a more complex structure is somewhat similar to the use of *sound pattern* (simple) and *gemisches* (complex) in the system of the Institute of Musicology, Arhus, Denmark (Hansen, 1977; Manthey, 1978).
3. In the grammar, non-terminals begin with an upper-case character.
4. This notion of instance was developed and used extensively by Sutherland (1963) in his SKETCHPAD system.
5. This is admittedly at the expense of speed. However, consider that if we do have to do an expansion before the score can be performed, we are still no worse off than the linked list representation of MUSIC V, for example. Furthermore, we still have the hierarchic representation intact, as a master "recipe" enabling backup, transformation, etc.
6. Note that we use the notion of instance here in exactly the same manner as during our discussion of scores. That is, there is only one *master-copy* of any particular object. Any change to that master-copy is therefore reflected in every instance of the object. This provides an efficient mechanism for refining the definition of a trumpet timbre, for example, or changing all "trumpets" to "flutes."
7. Note that in the discussion which follows, any name or value specified entirely in upper-case characters (such as OBJECT, UNDEFINED, etc.) is a defined constant for the music system.
8. In the implementation described, the symbol table size is limited to 256 entries, which has not proved to cause any constraints on the user. We can, therefore, take advantage of a space saving in that indices into the table can be represented by a single byte of information. ($2^8 = 256$).
9. Note: all examples are given in the programming language "C" (Kernighan and Ritchie, 1978). In the examples, a structure is an aggregate of data. The name following the *label* "struct" is the name of the aggregate. The names within the curly brackets define a template for the data in the aggregate. The first value in each row indicates data-type (char: 1 byte; int: 2 bytes; float: 4 bytes), while the second value is the variable name. Values preceded by a "*" are pointers to data of the indicated type (such as a structure). Pointers occupy one word of memory. Memory for such structures may be dynamically allocated or freed, and several structures of the same type may be allocated space in contiguous memory to form a table, or vector, or structures (as with a symbol table). Finally, variables terminating with a value in square brackets ("[" and "]") are arrays whose dimensions are contained within the brackets.
10. Note the special case for WAVEFORM files, where we interpret primary memory as the eight 2k word waveform buffers in the synthesizer. Thus, a non-NULL *svalue* for a WAVEFORM entry indicates which of the buffers (1-8) contains the waveform.
11. It is important to note that the Mevent structure is simply a template. It functions as a generalization for the different types of events which may occur in a score, and is included for purposes of convenience.
12. Many music systems, for example Tucker et al. (1977), avoid linked lists in the score. Instead, "notes" are stored in contiguous memory, the pointers then being implicit. While such a representation provides a more compact representation and a more efficient perform program, editing—which is the prime function of our system—is considerably less efficient. Furthermore, if in using the linked list approach the performance is too complex for the system to keep up with in real-time, we have found that enabling a score to be "compiled" into a more efficient representation is adequate for handling these special cases.
13. Note that space/time trade-offs dictate that the *time_factor* field affects either Mevent *durations* or *delta_ts*, or both (as determined by the *time_interp* field).

8. References and Bibliography

Baecker, R. M. (1969). *Interactive Computer-Mediated Animation.* MIT Project MAC, Report No. TR-61, Cambridge, Mass.

Benade, A. (1976). *Fundamentals of Musical Acoustics.* New York: Oxford University Press.

Buxton, W., & G. Fedorkow, (1978). *The Structured Sound Synthesis Project (SSSP): an Introduction.* Technical Report CSRG-92. Toronto: University of Toronto.

Buxton, W., A. Fogels, G. Fedorkow, L. Sasaki, & K.C. Smith. (1978). *An Introduction to the SSSP Digital Synthesizer*. Printed elsewhere in this issue of *Computer Music Journal*.

Chowning, J. (1973). "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation." *Journal of the Audio Engineering Society*, Vol. 21, No. 7, pp. 526-34. Reprinted in *Computer Music Journal,* Vol. 1, No. 2, pp. 46-54.

Grey, J. (1975). *An Exploration of Musical Timbre.* Ph.D. Thesis, Stanford University. Distributed as Dept. of Music Report No. Stan-M-2.

Hansen, F. E. (1977). "Sonic Demonstration of the EGG-synthesizer." *Electronic Music & Musical Acoustics,* No. 3. Arhus, Denmark: Institute of Musical Acoustics, University of Arhus.

Helmholtz, H. (1954). *On the Sensations of Tone.* New York: Dover Publications.

Kaegi, W. (1973). "A Minimum Description of the Linguistic Sign Repertoire: Part 1." *Interface,* No. 2, pp. 141-53.

------- (1974). "A Minimum Description of the Linguistic Sign Repertoire: Part 2." *Interface,* No. 3, pp. 137-157.

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

Kernighan, B., & D. Ritchie. (1978). *The C Programming Language.* Englewood Cliffs, N. J.: Prentice-Hall Inc.

Le Brun, M. (1977). "Waveshaping Synthesis." Unpublished Manuscript, CCRMA, Stanford University.

Lycklama, H. (1978). "UNIX on a Microprocessor." *Bell Systems Technical Journal,* Vol. 57, No. 6, Part 2, pp. 2087-2102.

Lycklama, H., & C. Christensen. (1978). "A Minicomputer Satellite Processor System." *Bell Systems Technical Journal,* Vol. 57, No. 6, Part 2, pp. 2103-2114.

Manthey, M. (1978). "The EGG: A Purely Digital Real Time Polyphonic Sound Synthesizer." *Computer Music Journal,* Vol. 2, No. 2, pp. 32-36.

Mathews, M. (1969). *The Technology of Computer Music.* Cambridge: MIT Press.

Mathews, M., & F. R. Moore. (1970). "GROOVE – A Program to Compose, Store, and Edit Functions of Time." *Communications of the ACM,* Vol. 13, No. 12, pp. 715-721.

Moorer, J. A. (1977). "Signal Processing Aspects of Computer Music – A Survey." *Proceedings of the IEEE,* Vol. 65, No. 8, pp. 1108-1137. Reprinted in *Computer Music Journal,* Vol. 1, No. 1, pp. 4-37, 1977.

Risset, J. (1969). *An Introductory Catalog of Computer Synthesized Sound.* Murray Hill, N. J.: Bell Telephone Laboratories.

Risset, J., & M. Mathews. (1969). "Analysis of Musical-Instrument Tones." *Physics Today,* Vol. 22, No. 2, pp. 23-30.

Rolnick, N. B. (1978). "A Composer's Notes on the Development and Implementation of Software for a Digital Synthesizer." *Computer Music Journal,* Vol 2., No. 2, pp. 13-22.

Schaefer, R. A. (1970). "Electronic Musical Tone Production by Nonlinear Waveshaping." *Journal of the Audio Engineering Society,* Vol. 8, No. 4, pp. 413-16.

Schaeffer, P. (1966). *Traité des Objets Musicaux.* Paris: Editions du Seuil.

Schottstaedt, B. (1977). "The Simulation of Natural Instrument Tones using Frequency Modulation with a Complex Modulating Wave." *Computer Music Journal,* Vol. 1, No. 4, pp. 46-50.

Sutherland, I. E. (1963). "SKETCHPAD: A Man-Machine Graphical Communication System." MIT Lincoln Laboratory, Report No. TR 296. Lexington, Mass.

Tempelaars, S. (1977). "The VOSIM Signal Spectrum." *Interface,* No. 6, pp. 81-96.

Thompson, K. (1978). "UNIX Implementation." *Bell Systems Technical Journal,* Vol. 57, No. 6, pp. 1931-1946.

Thompson, K., & D. M. Ritchie. (1974). "The UNIX Time-Sharing System." *Communications of the ACM,* Vol. 17, No. 7.

Truax, B. (1973). "The Computer Composition – Sound Synthesis Programs POD4, POD5, & POD6." *Sonological Reports*, No. 2. Utrecht: Institute of Sonology.

Tucker, W. H., R. H. T. Bates, S. D. Frykberg, R. J. Howarth, W. K. Kennedy, M. R. Lamb & R. G. Vaughan. (1977). "An Interactive Aid for Musicians." *Int. J. Man-Machine Studies,* Vol. 9, pp. 635-651.

Vercoe, B. (1975). "Man-Computer Interaction in Creative Applications." Cambridge: MIT Studio for Experimental Music, unpublished manuscript.

Xenakis, I. (1971). *Formalized Music.* Bloomington: Indiana University Press.