E-SCAPE

An Extendible Sonic Composition and Performance Environment

Tim M. Anderson

Thesis submission for the
Doctor of Philosophy degree

University of York

Department of Electronics

September 1993

Abstract

This thesis focuses on the use of computer-based systems which facilitate the creation of music which is designed for performance using synthesised and sampled sounds, and examines the problems of composing music and controlling sound using such systems.

A survey and critique of the structure and problems of using existing systems is presented. Two problems focused on are: (i) coping with the complexity and density of the control task for the complex synthesis algorithms which are needed in order to generate sounds that are of adequate musical interest. (ii) creating and controlling synthesis algorithms which are distributed amongst different devices of different types, and use different protocols.

This leads to a solution with three aspects: proposals for the organisation and functionality of such systems, the design of standards for communication between their components, and the design of the 'E-Scape' composition software system which attempts to solve the perceived problems. These chiefly consist in the presentation of complex instruments for use by composers, and the flexible control of sound generating structures on a variety of synthesis devices.

The solutions proposed will result in a composer being able to use the synthesis or compositional control aspects of existing systems as interchangeable components within a larger-scale, loosely-coupled system. Such a system can then be constructed and customised by composers to suit the kinds of activities they want to engage in, their experience of compositional methods and languages, and their synthesis requirements for performance.

The 'E-Scape' software has three functions within the thesis:

• to illustrate the difficulties involved in attempting to create such a loosely-coupled scenario using current synthesis systems.

• to illustrate the kind of functionality needed by any composition system to enable it to integrate into the proposed system organisation.

• to exemplify the features determined as desirable for electroacoustic composition which are absent from existing systems.

Contents

II - Proposals and design goals

III - Implementation of the 'E-Scape' demonstrator software

IV - Assessment and Conclusions

List of tables and illustrations

Acknowledgements

The basis of this work was funded by SERC, with additional valuable and appreciated support during development from the Drake Research Project.

I would especially like to thank Andy Hunt, Ross Kirk and Danny Oppenheim for their involvement and assistance in my work.

I must also thank Robert Sherlaw-Johnson for providing me the opportunity to start on this exciting path in music technology, and to Ross Kirk and Richard Orton, whose vision and hard work set up these paths at York during the nineteen eighties.

I would also like to express my appreciation to the following people for their specific academic or material assistance in various ways:

Martin Atkins, Steve Brewster, Simon Buckingham-Shum, Adele Drake, David Howard, Nigel Morgan, Richard Orton, Dave Rossiter, Tim Tozer, and Ron Patton.

... and thanks for inspiration in its many aspects, and for moral or actual support during the last four and a half years to:

K. and P. Anderson (my parents), James Angus, Andrew Cleaton, Harvey and Matt Dowdy, Adele Drake, Shirley Davies and Mike Edgley, Simon Emmerson, Tom and Liz Endrich, Declan Flynn and Debbie Hearn, Ian Gibson, Tony Hood, David and Clare Howard, Andy and Caroline Hunt, Ross Kirk and family, Dave Levett, Dave Malham, Gary Morgan, Nigel and Susan Morgan, Tony Myatt, Francis Newton, Danny and Ruth Oppenheim, Richard Orton, Mark Pearson, Judith Robinson, Dave and Nancy Rossiter, Mark Rowland, Al and Leona Schaff, Sue Taylor, Tony Tew, Jan Thomas, Dave Troughton, John Tuffen and Bron Smith, Edward and Judy Williams.

Finally, to Ruth McInnis, for everything.

Author's declaration

Aspects of the work in this thesis have been published previously.

These papers are reproduced in appendix 3.

I - A survey and analysis of the issues addressed in this thesis

This section, (chapters 1-3) describes the subject background in which this project is set, and surveys the current situation and makes an analysis of some of its difficulties.

Chapter 1 introduces and describes the overall nature of the subject matter of the research - computer music systems used for composition and synthesis - and places in it context within the world of music technology.

Chapter 2 then describes in detail the various aspects of computer music systems which pertain to the issues within the domain of the thesis.

Finally in this section, chapter 3 surveys a cross-section of existing systems, and presents a critique and analysis of the problems and difficulties - both general and specific - which arise in their construction and usage.

This sets the scene for the section II which proposes approaches to tackling the problems identified.

1. Introduction

This DPhil research project examines the usage of computers for composition, and some of the difficulties associated with such use. Its objective is then to address these problems and suggest ways of solving them, in the domain of the *organisation* of computer systems, and specific features of their *components*.

Various questions may legitimately be asked by a composer or musician, when confronted by the issue of the use of *computers* in the production of music. The initial questions may typically be general:

Why should a musician be interested in using a computer?

What *use* is a computer from a musician's perspective?

These questions may be tackled by first inquiring about composers - finding the answers to questions such as:

What sort of things do composers *do* or *want to do*?;

What compositional *activities* do they undertake, or would they *like* to undertake?

One can then inquire how computers can be used to support these compositional activities, by answering such questions as:

What do composers actively *do* at present with computers?

What support for such activities can be provided by a computer?

What compositional activities does the use of a computer make possible?

What kind of tools do computers provide for these activities?

These questions are examined in the following two sections.

1.1. Compositional activities

Many composers are drawn into using computers to create music, because they promise the ability, in theory, to generate any possible sound:

> "It is perhaps through the use of computers in sound generation..... that the great dream of electronic music - the realisation of musical conceptions without any limits other than those of the imagination - will at last become a possibility" (Griffiths 1979).

The term 'electro-acoustic' is used to describe music which places such emphasis on the timbre and temporal development of sounds.

Composers of electro-acoustic music thus want to *control* how this sound is structured in order to carve coherent musical and sonic structures out of the infinite possibilities - the creation of any sound or combination of sounds - which the computer presents.

To facilitate this kind of control, composers need to be able to perform various compositional activities when working with sound created by the computer, and thus require a computer-based composition system to provide them with certain abilities:

- the ability to specify and view the time-based *structure* of a piece, ie to specify the relationships between different events or groups of events, and group events into structures. This implies the ability to edit event structures by moving, copying, deleting events (or groups of events) within them.
- the ability to *generate* events via user-created algorithmic processes ('algorithmic composition').
- the ability to select groups of events on which to perform various operations: either directly, or via conditional criteria.
- the ability to specify, view and edit sound-generating algorithms.
- the ability to associate each event with a sound creating algorithm ('instrument'), and specify parameter values for the algorithm, either for individual events, or for structures containing them.
- the ability to *hear* the piece, or audition segments of it.

Composers thus use computers in a far wider range of activities than when composing music for conventional acoustic instruments. A composer can also take on the role of sound designer or inventor, orchestrator, performer, conductor, or sound projectionist[1].

Thus computer-based composition is a holistic activity, embracing the creation of sounds, the design of algorithmic processes to create those sounds, and the organisation of events which use and control these processes. Many composers view a computer as an integrated environment - a virtual studio where all the activities associated with composition can be carried out.

1.2. The use of computers for composition

Abbott distinguishes five major research issues associated with the use of computers in music:- algorithm design, graphical representations for music notations, computer architectures, programming languages, and real-time interactive systems (Abbott 1985).

This thesis focuses on the musical use of computers by *composers*, as opposed to performers, computer scientists, music analysts, and teachers (although there is not a clear-cut boundary between these and composers' activities).

From the 1950's onwards, composers have had an interest in creating new sounds and musical structures (relationships of events) by generating streams of digital sound sample data. These sample values can be created by some combination of two strands of compositional activity: 'sound synthesis', and 'sound processing'.

---

1 Sound projection is the controlling of the ultimate distribution of sound to an audience, typically via multiple loudspeakers.

1.2.1. Sound synthesis

The first strand of compositional activity involves a composer in controlling the direct computation of sound sample data, by (i) defining various low-level algorithmic processes which generate (synthesise) sound data, and then (ii) specifying higher-level parameters which control and structure the operation of these synthesis algorithms.

1.2.1.1 Defining synthesis processes

Examples of sound-synthesis processes include linear predictive coding (LPC), formant wave functions ('FOF'), frequency modulation (FM), amplitude modulation (AM), subtractive synthesis[1], the 'Karplus-Strong' algorithm, additive synthesis, and granular synthesis.

The composer is able to define these processes using various levels of computer interface which drive a lower-level software system. Such interfaces vary greatly in depth, complexity and presentation, for example:

- micro-coding of instructions for low-level digital hardware;
- a high-level text-based programming language interpreter;
- an iconic graphic user interface (GUI).

A composer can specify algorithms as mathematical functions within a programming language structure, or by employing pre-defined processes such as oscillators, table-readers, multipliers, filters etc, which can be used as building blocks to construct a specification of more complex algorithmic processes.

1.2.1.2 Defining sound events

The composer can then define sound events, by specifying time-ordered *parameters* for a selected process which control its timbral characteristics and their temporal development. The available parameters for such processes typically include those which initiate a new copy of the process, so as to create a new discrete sound event with a defined start and stop, and hence a duration.

To then create a piece of music using these algorithms, a composer is able to specify a structure for these sound events, often nesting events with time offsets inside larger structures. The composer can perform this kind of specification via a direct manipulation GUI, or via a text-based language - either as a separate list of parameters and times (eg CSound), or as a set of functions in a proprietary computer music language (eg POD, CHANT, Formula, Fugue, Pla, Symbolic Composer). These systems are described later in section 3.1.2.

These parameters then control the operation of the lower-level sound generating processes, to produce the stream of digital sound samples. These samples are usually stored in a 'soundfile' - a set of ordered sample values stored (usually contiguously) in a computer

---

[1] This is often colloquially termed 'analogue synthesis' - see 2.4.4.1.

data file stored on a fast access storage medium. This medium has historically been magnetic tape or disk, although optical media have recently become available. The soundfile can then be played by converting the sample values to sound via Digital to Analog Converters (DACs).

More recently, devices which can support real-time synthesis processes (ie which can generate sound samples at a rate fast enough to convert immediately to sound) have become available (see 2.4.2.2). The use of the intermediate soundfile has thus been augmented by the option to immediately convert the sample stream into sound.

1.2.1.3 Sound synthesis system structure

Figure 1 below illustrates a generalised computer system which supports these activities, and shows the flow of data through it. The algorithmic sound generating processes (shown in the centre of the figure) can be specified by the composer. The digital audio sample data can then be stored in a soundfile (bottom left), in so called 'off-line' operation. These sample values can then subsequently be read out and converted to sound by the DACs (bottom right). Alternatively, the samples may be converted immediately (in so called 'real-time operation), without the intermediate stage of soundfile storage.

As well as a composer wishing to specify the particular operation of a sound-generating algorithm, he/she is likely to want to exert additional control at a higher-level, so as to be able to describe musical events. Thus, the same sound-generating algorithm can be employed with different high-level parameters. These can include:

- the choice of sound synthesis algorithm,
- control parameters which affect the operation of the algorithm,
- the starting time and time interval (duration) during which the algorithm will run.

These last parameters together specify an single algorithm to run for a particular period of time - this can be termed an 'event'. An event may be likened to the conventional concept of a 'note' - an entity which traditionally would have a duration of no more than a few seconds. However, an event may be much longer - an entity whose timbral characteristics may alter during its course, or which has a discernible micro-structure[1]. Such an 'event' could, at the limit, constitute a whole piece of music.

In addition, a composer wants to be able to specify *relationships* between the timings or parameters of events, in order to create larger-scale musical structures. This specification of high-level event structures and event control parameters may require a different set of computer-based tools, in a different subsystem, as shown at the top of figure 1 above.

A composer can thus interact with a high-level event specification software subsystem to create and organise musical data and structures. This data is then processed into lower-level

---

[1] An event with such a micro-structure can be perceived as containing other shorter discrete but associated sound events.

data which can communicate via some kind of interface - whether physical or a software connection - to control the sound generating subsystem (the 'synthesis device').

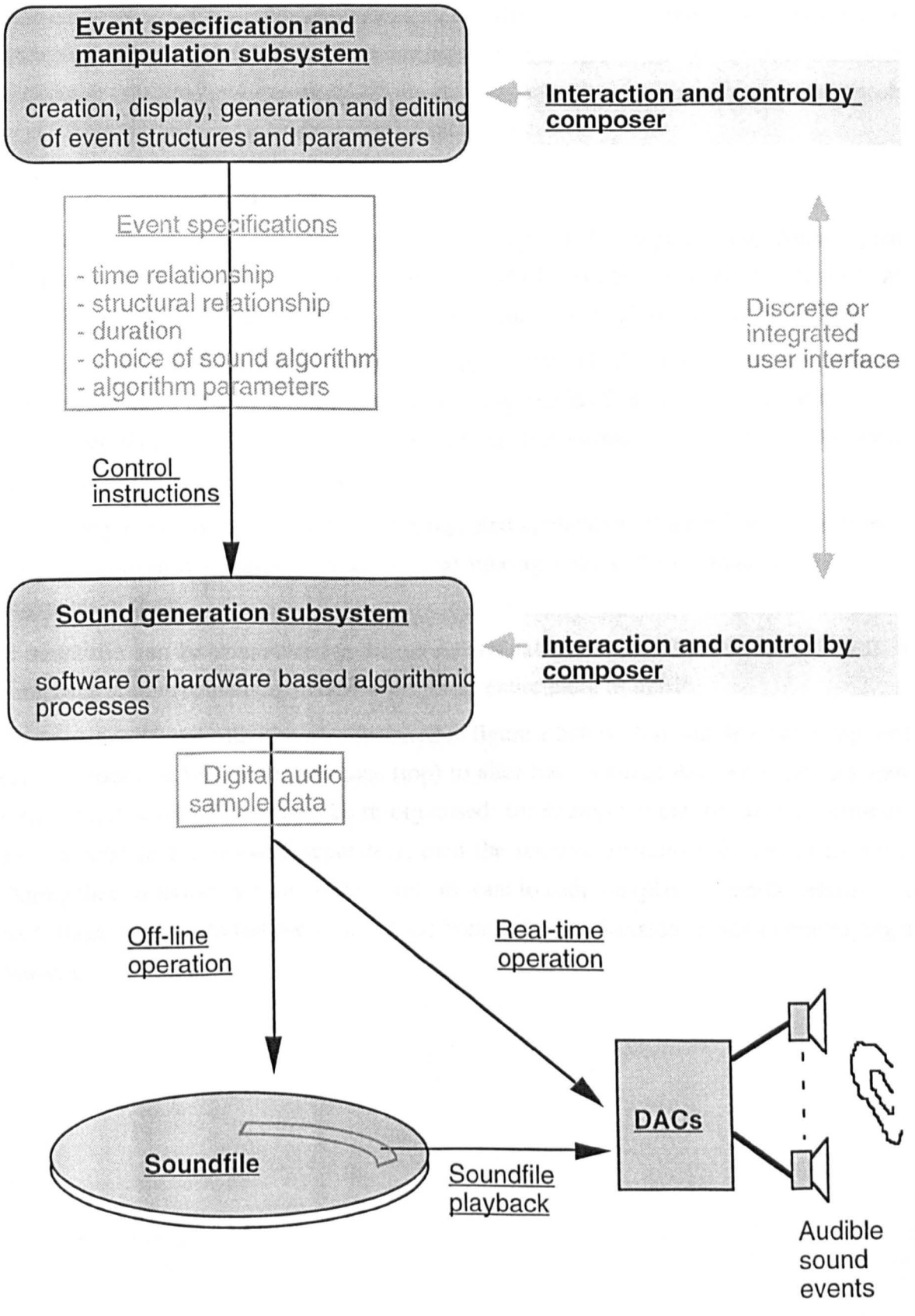


Fig. 1 Composition by specifying events and synthesis algorithms

The low-level processes running on this sub-system can also be specified by the composer, and indeed in some systems *require* being specified. However, a composer may wish to

work *only* at the higher 'event' level, and not become involved in having to specify the characteristics of the lower-level sound-generating algorithms.

Some computer-based composition systems overtly separate these two levels of compositional activity, with some actually consisting of two discrete subsystems. Other systems, however, attempt to facilitate a composer working at either level within the same working environment, and even provide the ability to use the *same* kind of functions at each level. Chapter 3 presents a cross section of such systems.

1.2.2. Sound processing

The second strand of the usage of computers by composers lies in performing digital signal processing operations on one or more soundfiles which have been created by synthesis (as in 1.2.2.1 above), or by sampling real sounds via Analog to Digital Converters (ADCs).

Such processes can include reverberation, time-reversal, the application of complex amplitude or pitch envelopes, manipulation and resynthesis of analysis data files derived by phase-vocoding (for such effects as pitch shifting, time-stretching, or merging the sonic characteristics of different soundfiles).

Soundfile processing can also include the repeated application of more basic operations - such as 'cutting and pasting', splicing and mixing - on sets of output soundfiles or fragments of them.

A soundfile can be considered to be an individual event (usually relatively short), a composite section containing many events, or an entire piece of music.

These compositional activities are illustrated in figure 2 below. A soundfile (shown top left) can be processed by various methods (top) to alter the soundfile data, or produce a new soundfile. A soundfile can also be re-organised: for example it can be cut into sections, each section then processed separately, then the sections recombined in various ways. During these activities, a composer is likely to want to audition (play) soundfile segments at each stage, in order to test the result of the compositional decisions made in employing a process.

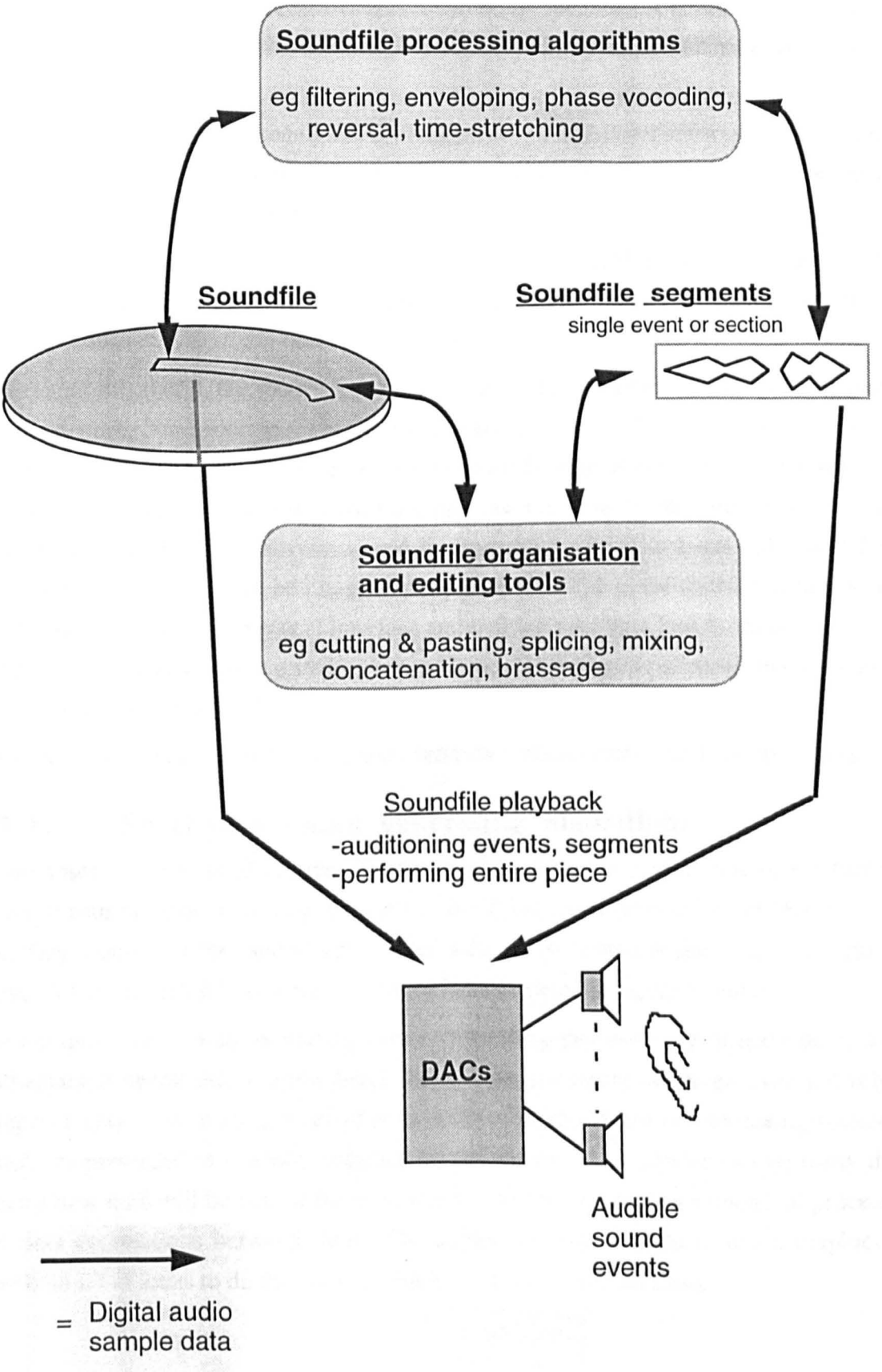


Fig. 2 Sound processing operations

1.3. Composers' usage of computer music composition systems

The structure of computer composition and synthesis systems (as illustrated in figure 1 above) thus consists of two (actual or notional) component subsystems in some combination:-

• an *'event specification and manipulation subsystem'* - a high-level software system with a user-interface to allow a composer to specify and organise event-level data, or control its generation via high-level algorithmic processes.

• a *'sound generation sub-system'* - a set of processes enabling musical events to be realised ('performed') via synthesis algorithms which produce sound data, and to which the event specification system can send control data.

Throughout this thesis, the word 'device'[1] will be used to signify an entity (a subsystem) which performs the processes which produce sound. Such 'devices' may or may not incorporate their own user interface systems for specification of event or sound generation processes. A device may consist of software processes running on the same host computer as the event specification subsystem, and be integrated with it (see examples in 3.2.1). Alternatively, a device may be physically separate from the event-specification system, which can control it via a physical interface and defined messages (see examples in 3.1.2). Whichever physical structure a system has, a 'device' is always a *notionally* distinct entity, and can be dealt with as such.

Composers may use and interact with such computer music systems in a variety of ways:

1.3.1. Specifying sound generating algorithms

A composer - especially of electro-acoustic music, where the emphasis is on the timbral nature of sounds - may often want to start by specifying sound-generating processes. To do this, they would use the 'sound generation' subsystem shown at the centre of figure 1 above. A typical such subsystem is illustrated in more detail in figure 3 below.

The composer may wish to specify sound-generating processes by directly describing mathematical operations at a low level, using a programming language. Alternatively a composer may wish to simply *select* from a set of primitive signal processing routines[2] which are presented as *modules* supplied by the system. A composer can typically then specify how data will be passed *between* these modules, to create a network of processes and data connections between them. The composer may be able to use a graphically presented set of icons to do this, or use a high-level text-based language.

---

1 A device will always be in lower case, to distinguish it from the Device class of software object, which (as for all such classes) are capitalised (see chapters 7 and 8).

2 These primitive processing modules are referred to as 'units' throughout this thesis; see chapter 2.

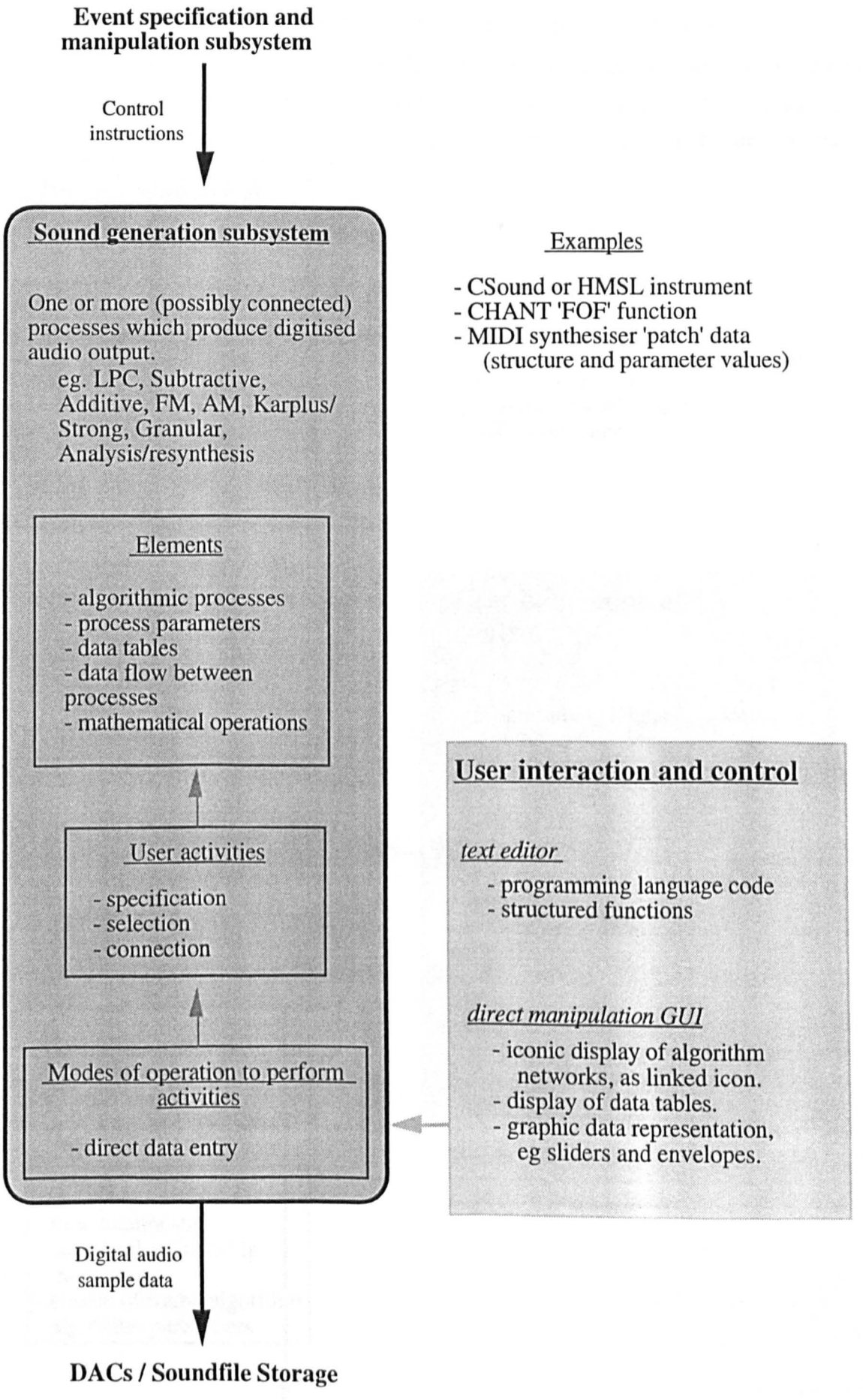


Fig. 3 The *sound generation* sub-system of a computer-based composition system

1.3.2. Specifying higher-level musical structures

Having described a sound-generating algorithm, a composer can then concentrate on specifying higher-level structures using an 'event specification and manipulation' subsystem, whose typical facilities and modes of use are summarised in figure 4 below.

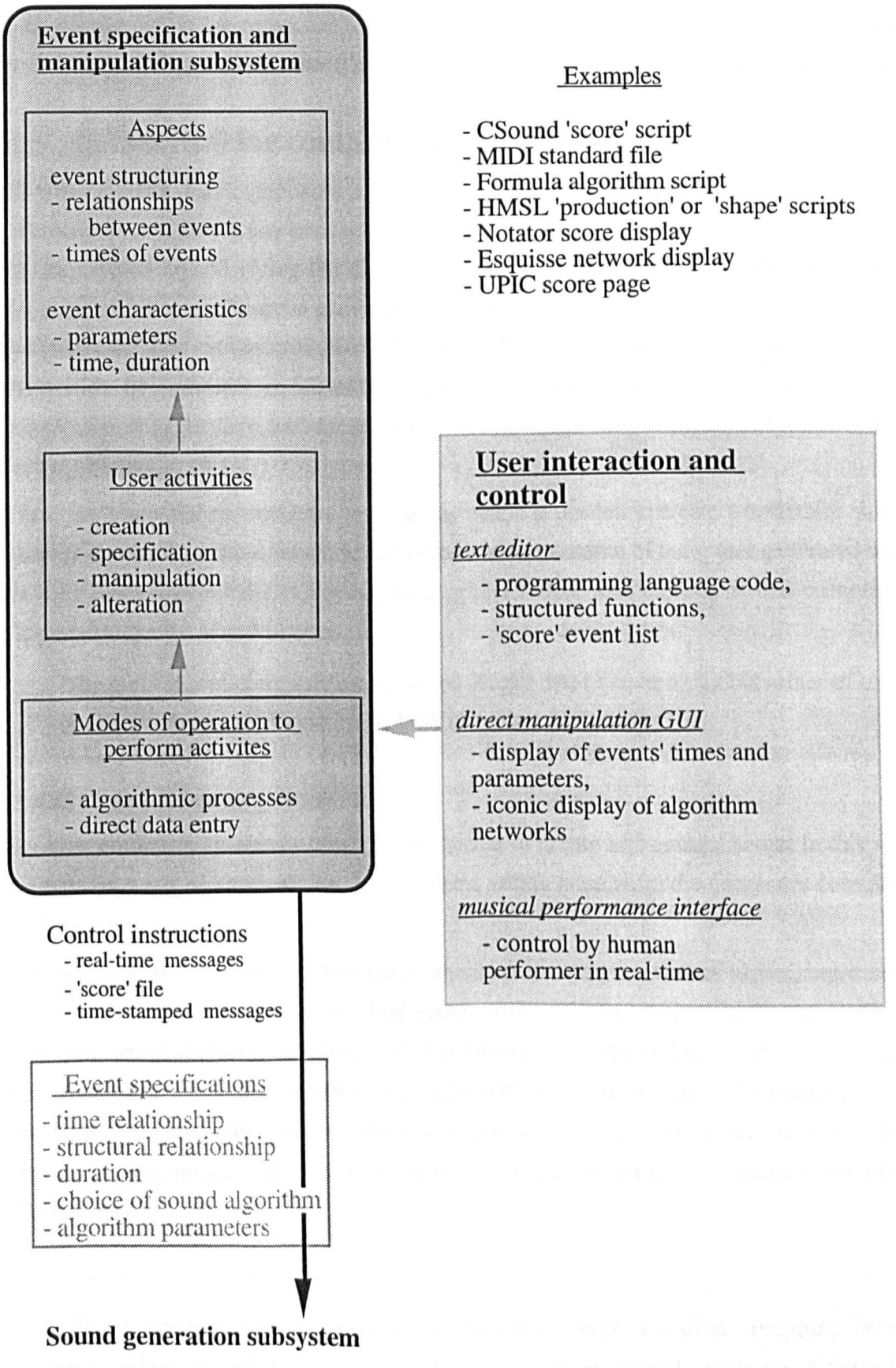


Fig. 4 The 'event specification' subsystem of a computer-based composition system

A composer may use a text-based language to specify score data or algorithms to generate or process data. Alternatively, a composer may be able to use a GUI which allows him/her to directly enter event data, and also provides a visual display of score data (usually in a time-based framework). The same GUI can also allow a composer to define the *structure* of a piece and specify the relationships of events using a direct manipulation[1] interface. Some systems also allow the composer to describe high-level event generating algorithms using an iconic GUI similar to that used in describing sound processing networks (see 1.2.1.1).

1.4. Difficulties for computer composition systems

As described in 1.1, composers are attracted to using the computer by its promise of enabling them to create any conceivable sound or structure. Although in theory, any sound can be created by specifying the digital data which describes it, in practice some way of specifying and controlling the generation of data via high-level processes - and parameters to those processes - is required. It is just not feasible to envisage *manually* typing in tens of thousands of numbers to describe each second of sound - both in terms of the laboriousness of the task, and the difficulty of knowing *which* numbers to type in order to create a particular sound.

Thus, the *complexity of structure* at all levels which is needed to create worthwhile musical conceptions which utilise the sonic and organisation potential of computer generated sound introduces new problems of management, presentation and control of this complexity. Oppenheim quotes Max Mathews:

> "the problems of computer music are no longer that of technology but rather of our ability to control it" (Oppenheim 1989).

1.4.1. Existing systems

Existing computer systems - used by composers to create and control sound in this way - have many ways of approaching the problems which arise from the necessary complexity of the sound synthesis task.

The multiplicity of ways of conceptualising and presenting the task of organising, creating and controlling synthesis and musical event structures has resulted in the existence of a large number of different systems. Each has its own strengths and limitations, and a corpus of human expertise and experience in its effective compositional use. All systems provide some kind of *framework* for compositional activity. This is very necessary, not only for electroacoustic music, but for any kind of compositional style of significant complexity (Toop 1985).

---

[1] A direct manipulation interface is one in which there is a direct mapping between *displayed icons*, and the *objects* which they represent. Visual *attributes* of these icons (eg their colour, length, shape, vertical or horizontal position) can also have a direct relationship with corresponding *properties* of the objects.

This diversity of compositional systems is both inevitable and essential; each focuses on some aspects of the control 'problems' quoted above, and the working practices needed to enable the creation of musical or sonic structures as conceptualised in a particular way.

1.4.1.1 General problems

Existing software systems have some problematic aspects which have hindered their uptake and proliferation in the 'serious' or 'academic' composition world.

There are several problems areas:-

- Systems may allow the specification of synthesis processes, but are tied to a particular platform, device or synthesis method - there is no facility to cope with new types of synthesis device.
- Systems may be able to communicate to several devices using different protocols, but have no facility to cope with new protocols which may be introduced, or augmentations and developments of existing ones. Such an eventuality would require the system software to be rewritten or updated, and continued usage of the system is thus reliant on the software writers still being interested, and in business.
- Systems cannot usually distribute a synthesis process over multiple types of device, or often, even over more than one device of the same type, thus placing limits on the complexity of sound which is achievable.
- The specification of event parameters by a composer requires him/her to have a detailed knowledge of the internal workings of the synthesis processes (and their control inputs) which will realise that event.
- To undertake algorithmic composition with any degree of understanding will always require a composer to learn some kind of programming language (Roads 1985c), but he/she is then locked into the limitations of that system.

1.4.1.2 'MIDI system' problems

Existing *MIDI-based* systems (see 3.1 below) usually have a well developed (often graphically-based) user interface, but have several additional problems:

- The 'event specification' subsystems are inflexible - the user may not expand the range of possibilities for structuring or presenting data.

- The complexity and variety of synthesis algorithms on most MIDI-controlled devices is limited.

- MIDI's communication bandwidth is too low to facilitate real-time control of complex multi-parametric processes, and devices have no capability for control via time-stamped communication from the composition subsystem.

- MIDI-controlled devices all operate their synthesis processes in real-time. Thus they cannot 'degrade gracefully' (Pope 1992b) into non real-time operation to allow more complex synthesis operations to be undertaken.

• a score must either be very 'sparse' - containing little specification of sonic detail - or else become highly device specific, by using particular MIDI system exclusive or controller messages (see 2.5.1.2) which are likely not be understood, or interpreted in the same way by other devices.

1.4.1.3 'Custom system' problems

Existing 'custom' systems (see 3.2 below), while far more flexible than MIDI-based ones, have a different set of drawbacks:

• The 'event specification' subsystems often have very under-developed score specification and presentation facilities, with little provision of graphic time-based displays of events' structure and parameters.

• Their instruments are also presented - even if graphically - in a way which is unintelligible unless its user has knowledge of signal processing algorithms. Even an instrument designer can find it hard to understand the functioning or mode of usage of his/her own instrument (Desain 1993).

• Such systems are often very insular: most, unless they use MIDI (in which case they incur all the difficulties above) are highly specific in their interface to synthesis functionality, and scores or sound specifications are very device specific. For example, such systems as CSound, UPIC and HMSL (see 3.2.1 below) cannot interchange their components - a CSound score cannot be performed on the UPIC or HMSL synthesis hardware; an HMSL score cannot be performed using a CSound or UPIC synthesis engine.

1.4.2. Obsolescence and adaptability of systems

Some systems utilise existing communication standards to enable specialised software and synthesis components within them to communicate. The MIDI standard, for example, has made possible some major advances in portability and specialisation, and has shown that standardisation can work. However, MIDI has too many limitations (see 3.3) to be effectively used for open-ended complex computer synthesis and composition systems.

Most systems, however, have event-specification subsystems which are either tied rigidly to a single sound generation subsystem, or are integrated with one in a single system, with the resulting lack of flexibility to expand their capabilities as new synthesis devices become available. These systems exist in isolation, dependent on the maintenance and adaptation by institutions of their software and hardware components - many fine system components of the past have either been lost, or have absorbed years of human effort in conversion.

For example, the CAMP[1] system (Free 1988) is based on the earlier SSSP[2] system (Buxton 1978) which has become obsolete, but:

---

1 Computer Assisted Music Project.

2 Structured Sound Synthesis Project.

> "... the original SSSP software contained many hardware/operating system dependencies, and the code had to be completely rewritten" (Free 1986).

This larger scale problem looms over the development and introduction of any new specific computer-based music composition system: namely how to ensure that existing knowledge and systems are maximally utilised, and that the effort invested in gaining expertise and experience in a new system will not be wasted in years to come, as it too is forced to be scrapped. For example, Lippe comments on the IRCAM institution's problems since the cessation of hardware production by NeXT corporation, as their systems are based on it.

> "This hardware crisis has focused IRCAM's attention on some issues that are of general concern to the pro-audio world... These are primarily software issues and include portability, commitment to future development, compatibility, standardisation, and most important: hardware dependence/independence"
>
> (Lippe 1993).

1.5. Thesis domain and objectives

The areas of compositional activity upon which this thesis concentrates lie within the *first* strand of activities described in 1.2.1, ie on the use of computer-based systems which facilitate structuring and control of sound events using synthesis processes.

It focuses on the aspects of such systems which facilitate algorithm specification (for the generation of both event and sound data), and graphical score representation and editing.

This thesis aims to examine and attempt to solve some of the problems described when using such computer-based composition systems to create and control music and sound.

The solutions presented involve two strands:

• proposing a way of *organising* such systems, and presenting a set of system organisation and intra-system communication standards.

• proposing specific features of the component sub-systems, and presenting a prototype 'event specification' software sub-system which exemplifies these recommendations.

1.6. Outline of the thesis structure

• Chapter 1 presents the background and problem domain to be tackled in this thesis, and sets the scene for the following survey and analysis in chapters 2 and 3.

• Chapter 2 presents a detailed analysis of the concepts, philosophy and structure which pervade computer systems which facilitate control over composition and synthesis. This provides the background knowledge needed for an understanding of chapter 3.

• Chapter 3 then examines a cross-section of existing computer-based music systems of various types. This is not an exhaustive survey of all systems, languages and devices, but focuses on representative examples of different approaches. This is then followed by a consolidated critique of current systems, with an analytical overview of the areas of concern.

• Chapter 4 then outlines and justifies a set of proposals for the organisation and structure of composition and synthesis systems, as a solution to the perceived problems resulting from the analysis in the previous chapter. It also summarises the intra-system communication facilities necessitated by this organisational paradigm, and the required functionality of devices and 'event specification' (composition) software, in order to act as component sub-systems within the proposed system organisation.

Recommendations are presented for:

- The organisation and structure of composition and performance systems.
- Synthesis device functioning and facilities.
- Inter-communication standards between event specification software and synthesis devices.
- The facilities provided by event specification software.

• Chapter 5 then presents the detailed design for a proposed communication protocol which will facilitate intra- and inter-system communication within the proposed system organisation.

• Chapter 6 outlines the design goals of the prototype 'E-Scape' 'event specification' software .

The facilities which are required to enable an event specification subsystem to be able to integrate into the proposed system intercommunication environment are illustrated by the design (and prototype implementation) of this 'E-Scape' composition system. E-Scape addresses some of the perceived deficiencies of existing composition systems outlined in chapter 3, as well as illustrating the functionality needed by a composition subsystem to enable it to integrate into the kind of system organisation which is recommended and outlined in chapter 4. It thus acts as a demonstrator for aspects of these system recommendations.

The E-Scape design aims to facilitate the specification of synthesis algorithms, graphical scores, and algorithmic programming, all within a single software environment. The innovative and significant design features of E-Scape are:

- The ability for a user to define new communication protocols, to enable usage and control of new types of device.
- The ability to allocate dynamically the synthesis resources needed to realise a score in the available devices, and determine the score's ability to be played correctly.
- The provision of an intermediate interface between the high-level compositional parameters presented to a composer, and the low-level control instructions for processes on synthesis devices. The *same* score may be performed using *different* devices.
- The ability to use synthesis processes *distributed* on different devices (which may utilise different protocols), using a single unified instrument structure which can be controlled by an event in a score.

• Chapter 7 explains the nature and value of object-oriented programming systems, and their usefulness and appropriateness for building music applications. The concepts behind object-oriented software design are also presented.

The choice of the Smalltalk-80 system is explained and selected language features illustrated, to enable comprehension of the subsequent presentation of the design and implementation of E-Scape.

• Chapters 8 to 10 then present the structure, implementation, functioning and usage of the E-Scape software within the Smalltalk object-oriented programming system.

• Chapter 11 presents an assessment of the E-Scape software system, and its success it meeting its design goals.

The communication and device functionality recommendations are evaluated to assess their feasibility via a detailed design study of the implementation of the proposed communication and device functionality standards for the MIDAS system. The extended design features of synthesis devices which will allow them to be integrated into the recommended system organisation are also suggested.

• Finally, chapter 12 summarises the aims of the research, its results, and the contribution thus made to various aspects of music technology:

If the recommendations for the structure of computer music systems are followed, then any synthesis device will be able to be integrated with, and controlled from, any event specification subsystem. Thus systems can become less dependent on specific hardware, and thus less prone to obsolescence. The adoption of standards for the behaviour of devices will also make it economic to construct more capable or specialised devices than at present, with a far larger potential user-base than presently exists for large computer music systems.

The 'E-Scape' event specification subsystem demonstrates various significant features:

- A synthesis algorithm can be distributed between multiple devices as well as in software, which will enable composers to employ more flexible and complex synthesis structures and processes using the devices available to them.

- The user can adapt the system to enable new types of device and communication protocol to be employed, reducing the possibility of obsolescence.

- A composer can use instruments which employ complex synthesis algorithms, yet present an informative compositional interface via their parameter input specifications. Hence, a far wider range of composers will be able to create scores using such instruments, and work transparently with complex sound generating processes without the requirement to comprehend their internal functioning or structure.

- Event parameters can be feasibly specified and understood within a graphic score, which will encourage the usage of such scores by composers of electro-acoustic music. This can increase the communication and shared understanding of the musical structures and thought processes involved in electro-acoustic composition, and may help it to approach the stature and communicability of the ideas conveyed by a conventionally orchestrated score.

2. Introduction to computer-based composition and synthesis systems

2.1. Overview

This chapter presents an analysis of the concepts and philosophy which pervade computer-based composition and synthesis systems. It then describes the structures within such systems, and discusses the issues surrounding their operation.

2.2. The musical applications of computers

There are two main areas of use of computers in music applications: (i) dealing with high-level musical data (*specifying* what sound or instrument to play, at what time and in what manner), and (ii) creating, processing and directly controlling the sound itself.

2.2.1. Dealing with high-level musical data

Data can be grouped into musical events which specify what sound (instrument) to play, when and in what manner. Their structuring and organisation involves high-level concepts such as rhythm, harmony, tonality and intervals. There are three types of activity within this area:

• The use of utilities such as the cataloguing and analysing of existing music, or aiding the teaching of music performance or composition.

• Using compositional tools or environments which enable a composer to structure, edit, generate and perform musical ideas. This may involve graphical notation input and editing, or some element of automated composing, where the composer may generate data via algorithmic processes which may be specified.

• The use of systems to provide performance assistance at a high level, and to facilitate interactive event selection, control of algorithmic event generation, or score following where the computer holds a list of events which are mapped to pre-defined input states, allowing a complex stored piece of music to be altered by a human performer during its playback.

2.2.2. Creating and controlling sound

The computer can be used in implementing a synthesis process itself and/or controlling that process to affect parameters of the sound (eg. its waveform, harmonic spectrum, formants, amplitude or frequency) and their temporal evolution. There are three types of use within this area:

• The direct creation of sound by a variety of synthesis processes (including the use of sampled real sounds). This could involve the direct *implementation* in software of synthesis processes, and/or the control of synthesis processes running elsewhere.

• The 'post' processing of existing sound, ie further processes on existing digital data describing sound events. This could be seen as an integral part of a synthesis process (in its wider sense) or maintained as a separate entity.

• Real-time performance control of direct synthesis and/or post processing. Here the control data is not present or necessarily known by the system at the start. Performance control implies that the sound creation processes are effected in real-time. Control may also may be effected via high-level musical data, as described above.

2.3. The structure of composition and synthesis systems

> "Controlling synthesis parameters gives the user complete flexibility in structuring, sculpting, and composing inner aspects of the sound - much more than processing natural sounds" (Risset 1985).

As stated above, this thesis focuses on the use of computer-based systems which facilitate the integration of composition and performance (sound realisation) using synthesised sound. Such composition and synthesis systems can be considered to consist of a combination of subsystems. A composer interacts with a high-level compositional software subsystem to facilitate the creation and organisation of event-level musical data and structures. This is then processed into low-level data which can control a synthesis 'device' (one or more lower-level synthesis processes) which realise audible sound events (Pennycook 1987). To realise a performance of the musical data, the compositional software needs to send this low-level control data to the synthesis device.

This division has historically been notional - eg in Music-N systems (Mathews 1969), where the textual interface for specifying musical and synthesis-process specification data is part of the same application which compiles and processes the data to audio output data. Many other systems also present no overt division to the user. Increasingly, however, systems are being divided physically, whether in software (several independent processes and applications running on the same processor), or in hardware (a physical link between applications running on different processors). Examples of both organisational structures are given below in 3.1.

These two subsystems must thus be linked by some kind of interface, whether this exists solely as a software connection or as a physical link.

In either case, the high-level composition software needs to send instructions to the device to effect its control. Such instructions are defined in a communication protocol: a system of low-level function or message definitions which sender and receiver both understand.

These functions or messages consist of data or commands which are received in the synthesis device, and initiate more primitive routines within it. For example, messages might load data for tables, or instantiate, connect, set the status of, or send data to various processes.

A system of such components can be implemented as a loosely or tightly coupled organisation.

2.3.1. Tightly coupled systems

A tightly coupled system implies and requires a homogeneity of components. These may be software modules running and inter-communicating on the same host, or hardware devices which will share a low-level bus interface and have the same input and output data transfer schemes. Such systems are one legitimate way of building a synthesis system, although potential timesharing vs. real-time contention problems require a careful implementation.

2.3.2. Loosely coupled systems

> "In a loosely coupled system, the processors are largely autonomous and require a protocol for inter-processor communication (eg Ethernet, RS232, MIDI)"
>
> (Loy 1986).

Loy is here discussing an envisaged structure *within* a synthesis system. Its advantages are that the processor nodes have a high degree of autonomy, and the system can thus more readily support a heterogeneity of components. This is useful as no one type of component can be expected to perform equally well, or even at all, on each type of processing task. Different nodes may usefully specialise in doing a subset of tasks well (Tanenbaum 1984). Furthermore, new nodes or new *types* of node can be readily added to the system, allowing it to expand in speed, versatility and scope. Such concepts have inspired the 'MIDAS' system among others (see 3.1).

The disadvantages of such loose coupling are its dependence on inter-process communication. Not only must a sufficiently high communication bandwidth be available, but valuable processing time is used up in the act of communicating via protocols. However, these speed problems are becoming less important as faster CPUs and interfaces are introduced. Some systems allow instructions to be transferred to, and stored by a subsystem along with a specified *time offset* at which each instruction should then be executed. This use of such 'time-stamped messages' (see 2.7) also reduces the importance of very fast communication.

Other negative factors in implementing a loosely coupled system are the additional complexity and expense incurred in providing enough intelligence (ie a CPU) at each node to support the communication protocol. Furthermore, if a node malfunctions, it can stop responding, making debugging it and the system difficult. However, if the concept of loose-coupling is scaled up, with each 'node' in a system itself being a large, more complex system, then these objections are removed, as each node will then inherently support such 'intelligence', and can be debugged if necessary in isolation from the rest of the network. The advantages of flexibility, specialism and expandability remain, subject to processing and communication speed limitations. Each 'node' of such a system would be able to communicate - via a common, sufficiently general protocol - whilst also being capable of stand-alone operation.

Some music systems take this concept further, with the aim of *decoupling* the high-level host specification language or system as much as possible from the engine which runs the

synthesis algorithms. This has been achieved even in otherwise tightly coupled systems via an interface layer consisting of defined functions, a good example being the NeXT 'MusicKit' (Jaffe 1989a; Jaffe 1991). New synthesis devices will only require programming to implement these function calls to be usable with all high-level software which uses the interface layer.

This concept of deliberately decoupling subsystems as a way of ensuring flexibility and expandability has gained in popularity, as general processing power increases have negated the speed overheads involved. Such systems can more easily accommodate developments in synthesis hardware.

Such adaptability to link with new subsystems is a way of avoiding wasting resources due to obsolescence. The history of computer-generated music, and computer-controlled music systems is littered with examples of man-years of high-level software development having to be abandoned because systems are tied to controlling synthesis devices which have become obsolete or unmaintainable, a prime example being the SSSP system, as described in 1.5 above.

To summarise, a typical composition and synthesis system can be considered to consist of:

• A high-level 'event specification' software subsystem, with which composers can interact.

• Sound generating synthesis processes running on a 'device' - another discrete subsystem, or a notional software subsystem.

• An interface and communication protocol to communicate from one to the other, whether this consists of function calls within the notional components of a composite system, or messages between discrete physical components.

2.4. Synthesis devices

Various kinds of synthesis processes can be employed to create the final audible sound output of the system. Such synthesis processes can be conceived of as sound output 'devices', and this term will be employed henceforth.

As stated in chapter 1, the word 'device' is used to signify a subsystem which produces sound. Such a subsystem may or may not incorporate its own user interface to enable direct user specification of its sound generation processes. However, to be described as a 'device' by the definition in this thesis, the subsystem must include a facility for *external* control by *another* discrete subsystem (even if both actually consist of software running on the same host). A 'device' must therefore be able to be treated as a *separate* entity.

2.4.1. Synthesis methods

There are two methods employed by synthesis devices to generate audio output.

2.4.1.1 Analogue synthesis

Analogue devices employ a network of electrically[1] controlled electronic hardware modules which interact with each other. These methods were used in the first synthesis devices (Moog 1965). To facilitate control by an external computer-based 'event' specification' subsystem, its (necessarily digital) control messages must be converted into analogue voltage signals via Digital to Analog Converters (DACs). This arrangement is known as a Hybrid System (Chamberlin 1980). Examples include the GROOVE and MUSYS III systems (Loy 1985b), the latter also employing directly digitally controlled modules.

While almost extinct in academic computer music circles, such hybrid structures still persist in various commercial devices (eg the Oberheim 'Matrix 12' synthesiser). Such conversion from digital to analogue control voltage signals can also be performed by commercially available processors, facilitating computer control of various older analogue devices[2].

Such devices *must* be controlled in 'real-time' (see 2.4.2.1), ie control values sent to the device from a computer-based event specification system are acted on immediately by it. Additional user control can also be added during performance in the manner of a conductor (Risset 1985).

2.4.1.2 Digital synthesis

Digital devices employ a software or firmware program which calculates a series of digital samples according to a mathematical process or algorithm. Most such systems (now all pervasive) generate 44100 16bit words per second (so called 'CD quality'), typically on 2 or 4 discrete audio channels, which are finally converted to sound by one or more DACs, or output in a standard digital format, eg AES/EBU (AES 1985). This synthesis software may be *loosely* or *tightly* coupled (Loy 1986) to the processor or system (the 'host') on which the high-level event specification software is running:

- In a loosely coupled system, the synthesis software may be run on a discrete device which is physically separate from the host, and communicates with it via some physical connection. Such software can consist of specialised DSP algorithms, or more general linked processes (Chamberlin 1980). These processes can run on custom VLSI chips, various types of dedicated DSP chips, or more general purpose microprocessors,

---

1 Usually via *voltage* control.

2 Note that the term 'analogue' is often now misapplied to commercial synthesis device which employ digitally controlled circuit elements. Its use in this context usually implies the use of a simple oscillator -> resonant filter -> envelope architecture, with *simple* waveforms used for the oscillators.

which could be the CPU of another computer system. Commercial MIDI[1]-controlled devices all fall into this category, mostly using custom VLSI chips running custom processes (Mauchly 1987), but recently there has been an increase in models which use general purpose DSP chips, for example the Peavey 'DPM' range, and the Evolution 'EVS1' both of which use Motorola 56000 series DSP chips.

- In a tightly coupled system, the synthesis software may be run on the host itself; the event specification subsystem can then communicate with the synthesis processes via software connections such as UNIX sockets, or simple function calls within an integrated environment.

However, in both of these situations, the synthesis device can be considered to be a *separate* entity. In each case the control software communicates to it via low-level function calls, whether these result in communication via software or hardware connections.

2.4.2. Modes of device operation

2.4.2.1 Analogue real-time synthesis

All purely *analogue* synthesis devices inherently generate sound output continuously. Output is produced in immediate response to data states on their inputs, ie there is no sense in which 'processing time' is required to generate sound. Obviously, a 'delay' module used in such a system will allow a time delay to be introduced into the sound generation process, but this is an overt effect which can be deliberately used as desired, rather than a sound generation artefact or by-product.

2.4.2.2 Digital real-time synthesis

On the other hand, *digital* synthesis devices, as described above, need to calculate a series of digital 'sample' values which are then converted to the final sound output by one or more DACs which require 'feeding' with samples at a specified rate.

If they can carry out the specified operations (eg. as defined by a network of connected algorithms) fast enough, such devices can generate sound samples at a high enough rate so as to be instantly converted to sound output via the DACs.

For example, a device may be required to produce two channels of sound output at a sample rate of 44.1 kHz - ie each second's duration of sound requires 44100 sample values to describe it. This sound output can be produced immediately by a device, *if* its synthesis processes can calculate 88200 final sample data values per second. This kind of operation is known as real-time.

MIDI-controlled commercial synthesisers all (at present) work in this mode, with fixed numbers and types of algorithm which guarantee that processing can be achieved. Some

---

[1] MIDI (Musical Instrument Digital Interface) is a widespread communications standard for electronic music devices, especially in the commercial world. It is described in detail in section 2.5.

limited scope for varying the complexity of synthesis processes is often provided (typically by a simple duplication of processes, eg using four instead of two oscillator-filter-amplifier chains). The increased processing required for a sound is then offset by a concomitant *reduction* in the number of simultaneous sounds which can be realised at once (known as 'polyphony' in the MIDI-device context), so as to maintain real-time operation.

Real-time operation has advantages and disadvantages:

Advantages of real-time operation

• Real-time operation provides immediate feedback to composers of how the piece of music *sounds*, especially as they may often be using synthetic instruments, which can be highly complex as well as novel. They are thus often unfamiliar to a composer in the sound they produce, and even expert composers cannot then rely on their normal expertise of 'hearing' via their musical experience how an instrument will sound in context, let alone a combination or transformation of such instruments.

This feedback enables the building up of a perceptual map in the user's mind between the synthesis and music structures and parameters employed, and the psychoacoustic perceptual parameters which are parsed by the brain (Bregman 1990) from the resulting sound output.

• Real-time operation also makes it is far easier to synchronise processes occurring on several synthesis devices at the same time to achieve a composite sound output. This might be desired for a number of reasons: to implement a large number of different instruments, to realise a score with a large polyphony (number of simultaneous events), or even to implement a large complex instrument using synthesis processes distributed over a several devices.

In such a composite network of devices, control messages can be sent to each synthesis device from a central event-specification system, and all devices will respond immediately, if operating via real-time control. Each device will actually have a small but finite delay in responding to control messages, usually of the order of microseconds. These small response delays will typically be different for each device, but such delays can be allowed for if necessary by the event-specification subsystem. In practice, unless sound components from different devices are designed to interact in a fashion which makes small time differences significant (eg to produce phasing effects), then such response delay differences will not be perceived in practice.

Such a co-ordinated response is far easier to control than if one or all devices operate via creating an intermediate 'off line' soundfile which must then be replayed (see next section below). For such a set of devices to be usable as an integrated whole, the playback of each of these soundfiles would then have to be synchronised, both with each other, and with any devices in the system which are being operated via real-time control.

Disadvantages of real-time operation

There are a number of problems which arise when real-time synthesis processes are employed:

- The bandwidth of the communication link between control software and synthesis device must be large enough to cope with maximum anticipated data density for the control of synthesis processing (ie sending data to inputs of the synthesis processes).
- The message must be acted on immediately by a device.
- The exact time a message is sent is crucial.
- The control software's interpretation of score data structures, and subsequent conversion to device-level messages is time-critical.
- Heavy demands are made on hardware and control software.
- The system is unable to *guarantee* accurate performance in advance, and it is difficult to predict *when* accuracy or data will be lost by timing errors, insufficient device processing resources, or host-to-device communication bandwidth limitations.

2.4.2.3 Digital off-line synthesis

A synthesis device may not have the processing speed or power to carry out the specified operations fast enough to generate sound samples at a rapid enough rate to be convertible to sound in real-time. For example, 88,200 16-bit samples may be required per second - as described in the example in 2.4.2.2 above - but the device may only actually be able to calculate 60,000 such samples per second.

To cope with this situation, the output sample rate may be lowered (eg to 22.05 kHz) which reduces the rate of demand for output sample values. This enables the device to produce these samples fast enough, but with a consequent reduction in output sound quality, in this case from a reduced frequency response.

Another strategy would be to reduce the *size* of the output data samples ('words'), eg from 16 to 12 or 8 bits per sample. In the above example, the device might be able to calculate 100,000 8-bit samples per second. Again this reduces the quality of the resulting sound, in this case by a reduction in dynamic range.

If such degradation of sound quality is felt to be unacceptable, then the full requirement of 88,200 16-bit samples will have to be calculated for each second of final sound output, but at the *lower* rate of 60,000 per second. In a converse situation, processes may actually be able to produce data samples at a rate *faster* than is necessary for real-time conversion (ie *more* than 88,200 per second in this example), perhaps as an intermediate stage in a larger scale operation. In either case, these samples cannot be instantly converted to sound output, but will have to be stored in the form of a 'soundfile'. This is a (usually contiguous) file of sample data - typically stored on an optical or magnetic disk medium - which represents the final sound output.

When the synthesis processes have been completed and the soundfile created, it can then be read back from disk at the necessary rate to be converted to sound via DACs as normal.

This process is referred to as 'off-line' synthesis, and also has advantages and disadvantages:

Advantages of off line synthesis

• There are fewer demands on the device, and the communication link control software, in terms of speed of response, and processing power.

• The fact that a significant delay is incurred between the specification of sound events, and hearing the result can encourage a composer to think more carefully about what he/she is trying to do. For example, a composer may wish to set up orchestration-like constraints in a pre-compositional phase of activity, to test and establish a palette of tonal colours and configurations.

Thus, off-line synthesis may encourage more reflective and structured compositional conceptualisation and design, as the results of improvisational 'trial and error' techniques are *not* so easily obtained (Morgan 1993). In fact off-line synthesis more resembles the situation of a composer scoring for conventional instruments, than does real-time synthesis.

Disadvantages of off line synthesis

• More user operations are required in order to hear the resulting sound at each stage of the compositional process. Some composers may be unused or unwilling to accept the time delay between specification and audition of the sonic results.

• As discussed in 2.4.2.2, it is more problematic to synchronise the playback of a distributed (multi-device) system which is running in an off-line mode.

• The formation in the composer's mind of experiential links between synthesis parameters and the resulting sound characteristics is inhibited.

2.4.3. Controlling synthesis devices

Each device requires some kind of user instructions to control its sound creating processes. Certain *types* of control instructions for devices can be generally distinguished. In addition, such instructions may be sent to a device for immediate execution, or carry a time-stamp for when they are to be executed. The message protocols used may also be custom designed for a particular device, or be taken from a standardised protocol.

2.4.3.1 Types of control instructions

Control instructions can perform various general functions within a device:

• Specifying synthesis structures

Structures consisting of networks of synthesis units need to be instantiated in a device. Instantiation in this case refers to the coming into being of one or more processes (units), which are described by a named specification or template known to the device. Each unit then has an identity and location within the device, and can be referred as an entity by subsequent messages.

The structure of synthesis units in a device has to be specified to it by a user. This may be done via a user interface which exists either on the device itself (if the device is a discrete

entity), or in the system within which the device is embedded (if the device is integrated within a larger system). Alternatively, synthesis structures can be specified by messages sent from an *external* control system (often included as an aspect of an event specification subsystem).

This specification of structures within a device can be performed either immediately before sending a request to instantiate the structure, or in a prior operation. In the latter case, the specification will need to be *stored* by the device, so as to be usable in future.

Such specification may include:

- specifying which units are to be used.

- specifying what each unit is to be connected to.

- specifying any initial values their inputs should have.

- specifying any data tables or other structures to be downloaded, or to be created via built-in functions in the device.

• Instantiating synthesis structures

As just described, structures may first be specified to the device as a template which is stored in the device. The structure specification can then later be *recalled* from storage in the device, and sets of processes instantiated according to it. Some devices can store many such structure definitions, and may have pre-set definitions installed by manufacturers (in ROM or battery-backed RAM).

Alternatively, a structure may be specified and stored in the control system, then instantiated when needed, unit by unit. For example in the CSound system (see 3.2.1.5) an 'instrument' (structure specification) is defined by text in an 'orchestra' file; in the CHANT system (see 3.2.1.6) LISP functions are used to specify units (in this case FOF processes).

• Sending control data to units

Control data needs to be sent to a structure in the device. Such data may be addressed directly to an input of a specified unit within the structure. Alternatively, it may be addressed to an 'input' entity of the structure, which has a mapping to one or more connected units within it.

• Starting and stopping units

Instantiated units in a structure need to be able to be started and stopped running. In some devices, *each* unit may require a message sending to start or stop it running. More typically, a start or stop message can be sent just to specific units, or to one or more 'inputs' of the structure, or simply sent to the structure instance itself (identified by an id number). These start and stop messages are really just special cases of the control data messages above.

2.4.3.2 Real-time and time-stamped control of devices

As described in 2.4.2, synthesis devices can operate in two possible modes: 'real-time' and 'off-line'. The former produces sound output *immediately*; the latter produces output in the form of a data file, from which sound output can *later* be generated.

A device requires information to specify and *control* its synthesis processes. This control information may *also* be communicated to the device in two modes: 'real-time' or 'time-stamped'. In 'real-time mode, control messages or commands are received by a device from the controlling (event specification) subsystem, and acted on immediately. In time-stamped mode, control messages or commands are presented[1] to a device with a 'time-stamp' incorporated. This indicates a time offset in the future when the device is to *act* on the message. Note that these modes of device *control* do not necessarily correlate with the similarly named modes of device *operation*. For example, a device can still operate its *synthesis processes* in real-time, even if *control messages* are presented to it in a time-stamped manner, in advance of operation. The CSound system acting as a device with MIDI control (see 3.2.1.5 and 11.1.2.1) exemplifies this situation.

2.4.3.3 Custom and standardised control protocols

Before 1983, all systems controlled synthesis devices via what can be termed 'custom' communication protocols, as do many non-commercial systems of today. In such systems, the high-level composition software is linked to a custom-designed synthesis device, via control protocols which are defined only for that device by its designers.

Examples of such custom protocols are:

- The 'MIDAS' protocol consists of sets of messages to communicate from a high-level control system to the MIDAS system (see 3.1.2.9). This protocol has been developed by the author in consultation with the MIDAS development team, and is presented in chapter 11.
- The 'Kyma - Capybara' protocol consists of low-level messages, used to communicate between the high-level Kyma software - an event specification subsystem - and the connected DSP-based Capybara synthesis device (see section 3.2.1.3 for details).

  Unfortunately, despite requests, the commerciality of this system has prevented any details of this protocol being made public.

Various efforts to formulate a standard control protocol for synthesis controllers and devices culminated in the creation in 1982 of the 'Musical Instrument Digital Interface' (MIDI) standard (Loy 1985a). This is a hardware and software protocol which over the last 10 years has become ubiquitous in the commercial electronic music world, enabling synthesiser devices, performance controllers and computers (as well as a host of specialist products) to communicate music performance and synthesis control data.

---

[1] A device could read messages or commands from a file, or receive them via a physical communication link.

Thus different devices from different manufacturers can operate with each other at least to a certain level of functionality, ie there is a *de facto* subset of available messages which all devices can be safely assumed to implement. This level of standardisation is almost unprecedented in the world of communication and computers, especially given the entirely voluntary nature of the level of compliance with the standard specification. The strengths and weaknesses of the MIDI protocol has been the subject of intense debate ever since in the electronic music community, and will be discussed in section 3.2. A discussion of certain aspects of the MIDI protocol necessary for this ensuing analysis are presented in the next section.

2.5. The MIDI Protocol - a discussion

The MIDI (Musical Instrument Digital Interface) protocol is a widely used standard (IMA 1988) which has extended both in functionality and usage since its initial publication in 1982 as a common communication system for commercial musical keyboards and synthesiser devices. It is now administered by the International MIDI Association (IMA).

There are many aspects to MIDI, only a subset of which are relevant to this project. There are many good explanatory texts and articles which give fuller details (Loy 1985a; Anderton 1987), but an overview of MIDI is presented in appendix 5.

One of the aims of the 'E-Scape' software system (described in chapter 6) is to be able to use a *variety* of protocols to communicate with synthesis devices. Thus, MIDI is only one of *several* possible protocols used by E-Scape, and only enough detail of its specification and messages will be given here to enable the reader to follow the subsequent discussion. The nature of the MIDI protocol is also very much bound up with synthesiser devices' implementation and response to its messages, hence a fuller picture will emerge from the analysis of MIDI-based devices to be given in 3.1.1.

MIDI uses a unidirectional serial interface, and defines a set of message types which can be used in various combinations. The specification allows for extensions, and many have in fact been added since MIDI's inception. Thus any systems using MIDI need to be expandable by a user, if the user is not to have to rely on the support of the system manufacturer to continue to hard-code the latest messages into the system. Messages are categorised as either 'Channel' or 'System' messages.

2.5.1. 'Channel' message types

Channel messages can address synthesis structures in devices via a 'channel' number, of which there can be sixteen on any one MIDI link. They can transmit commands whose names derive from the keyboard origins of MIDI, although they can and have been used for other purposes (eg 'note on' messages used to trigger snapshots of mixer, or lighting controller settings).

2.5.1.2 Note.on

This message starts a synthesis structure in a device playing, with data which specifies integer 'pitch' and 'velocity' values. The 'pitch' datum conveys a semitone 'key' number (showing MIDI's keyboard origins) with 69 as A 440Hz. Many synthesiser devices can, however, be set up to interpret each key number as an *arbitrary* pitch via a table stored in the device. Recent extensions to MIDI also provide system messages to specify each note pitch in small fractions of a cent[1] (Scholz 1991).

The 'velocity' datum is usually interpreted as influencing some aspect of the synthesis process which is initiated by the 'note on' message. Its name is derived from the speed of

---

[1] A cent is a unit of pitch, equal to 1/100 of a semitone.

key depression during keyboard performance - the original intended use of MIDI. However, the *sonic* meaning of the 'velocity' field is *arbitrary*, ie the response of the synthesis process to it is not defined, and some (older) devices will even simply ignore it. This problem is dealt with at length in 3.3.3 below.

2.5.1.3. Note off

This message is designed to match a corresponding 'note on' message, to command the playing of a note to cease[1]. It *also* has 'pitch' and 'velocity' data, the latter name derived from the speed at which a key is released during keyboard performance.

The 'pitch' datum conveys no new information, being used as the 'match' field by the device in order to identify which note to stop. Thus only a single note of any one pitch can be playing, on any one channel. Again, this reflects the keyboard origins of MIDI, where such a restriction is perfectly natural. Strictly speaking, some MIDI devices *do* allow more than one note to be initiated, by sending several note on messages with the same 'pitch' field value. However, this still *effectively* sounds a *single* note, as there are no messages currently available in MIDI which can be addressed to a *particular* one of these notes - notes are only identifiable by their *pitch*, and all these notes have the *same* pitch. Use of such 'multiple' notes is not a deliberate design feature of such devices, and only results in using up polyphony on the device.

This message does not necessarily *stop* a synthesis structure in a device operating (playing), but instructs it to move to its 'release' phase of operation which culminates in its termination. The 'release time' - the time elapsed between receipt of this message and note cessation is again not specified by the message, and depends on settings within the synthesis process. The 'velocity' datum may be used to affect parameters of the 'release' phase, most typically this release time.

2.5.1.4. Controller

This message specifies a 'controller number' which allows a number of independent control messages to be identified and used on the same channel, although such messages will affect all notes on this channel. Some controller numbers have been defined with a name which conveys their intended use (eg number 6 is named 'data entry'). The majority of controller numbers are presently left undefined, and most MIDI-based software does therefore allow controller numbers to be named by the user.

However, almost all control messages do not have a standard *response* defined for the device. For example controller number 1 ('modulation wheel'), or controller number 4 ('foot controller') may have any number of effects on the sound output, depending on settings within the synthesis structure in use, or globally in the device.

---

[1] The 'note on' message can also be used to the same effect by using a 'velocity' value of zero.

The message then specifies a 7 bit value, but a second 'matching' message (with a 'controller number' offset by 32) may optionally also be sent to convey 7 more bits as an LSB. Again, whether or not a device responds to this additional LSB message, or indeed to any particular controller number is *not* standardised.

The number of controller numbers is finite (128), and thus there is a facility to use a large number of additional ones via a scheme known as 'Registered Parameter Controllers' (see Appendix 5).

2.5.1.5. Channel Aftertouch

This message is similar in effect to a controller message[1], able to convey time-varying data values which can affect synthesis processes for *all* the notes playing with this channel .

Its name derives from the originally expected source of the message, pressure applied to a keyboard key *after* it has been fully depressed. However, as with the 'controller' message, the sonic *meaning* and *effect* of the message is undefined, depending on a device's internal settings.

2.5.1.6. Pitchbend

This message can alter the pitch of all notes playing on a particular channel. It has two data values, giving a potential 14 bit resolution, but most present devices only respond to the MSB value (ie with 7 bit resolution), although some more recent devices have started to increase this resolution.

Again, however, the *response* of a device is not specified: a few devices do not respond at all; most will alter the pitch by an amount which depends on another setting within the device (usually termed 'bender range' or 'pitch bend sensitivity'). Values can typically be set to integers between 1 to 12, 24 or 36, and indicate the positive pitch variation in semitones when the maximum 'pitch bend' value is received. The value for which *no* pitch variation is effected is actually the half way point in the value range, with a zero value producing maximum negative pitch variation.

The 'pitch bend sensitivity' or 'bender range' setting can often be transmitted to the device via a MIDI message, and this capability is a de facto standard on recent devices.

2.5.1.7. Program change

This message can specify a 'program number' to select one of the 'programs' or 'patches' - synthesis units or networks - whose specifications are stored in a device. The selected 'program' is then installed in an active memory area in the device (with a designated MIDI channel). The synthesis structure can be played (ie set running) and sent parameter data by messages on this channel, or addressed directly within the device's memory map, typically via system exclusive messages (see below).

---

[1] Except there is *no* facility to extend the data range to 14 bit resolution.

A major shortcoming is the small limit (128) of different 'programs' which can be selected; many - even cheap - MIDI devices now have more than this. The new recent definition of controller number 0 as a 'bank select' message gets round the problem, but is inelegant.

However, it does emphasise the fact that even using a standardised protocol does not release a system which uses that protocol from the necessity to be adaptable to new developments.

2.5.1.8. Polyphonic aftertouch

This message is the only one which allows a continuously varying parameter value to be sent which is assigned to an individual note, again using the note's 'pitch' value for matching. Unfortunately, very few devices at present support this message, and extensive use would risk exceeding the MIDI bandwidth, as discussed below.

2.5.2. 'System' messages

System messages can be used to perform various utility or synchronisation tasks (see appendix 5). In this thesis, the most important category of system message is the 'System Exclusive' message which can contain an arbitrary data format. The message is identified by a 'manufacturers id' (a number assigned to a particular commercial manufacturer registered with the IMA) or a 'non-commercial id' such as is used by the University of York's MIDAS device. Some other reserved ids are used for extensions to the MIDI standard (see appendix 5).

The format of the data following the id is not specified, and is left up to each manufacturer. Many of the more subtle and complex timbral modifications to a sound produced by a MIDI device can be controlled by these messages, and any composition system which aims to use such devices must provide a way of defining the fields and kind of data needed for such messages.

2.5.3. General MIDI

This recent extension to MIDI provides for some degree of standardisation for synthesis devices which conform to the 'General MIDI' standard, both in terms of the facilities they must provide, and their response to certain messages. For example, devices must respond on all 16 MIDI channels, and be able to assign a different program (sound) to each. Its other main features are an association of each 'program number' value with a particular identifiable sound type. For example, a value of 1 should result in an 'acoustic piano' type of sound being installed in the device. These standardised sound associations are mostly emulations of real instruments; the descriptive vocabulary for the kind of timbrally innovative sounds used in electro-acoustic composition is not developed enough to allow such associations; in any case the emphasis of such composition is on the creation of new, unheard of sound textures.

However, the concept of devices having a standardised *response* to messages is a useful one.

2.6. Future standard protocols

Other standards than MIDI have been suggested for music communication purposes, but none has yet achieved MIDI's ubiquity.

2.6.1. 'MIDI 2'

This chimeral entity has been mooted almost as soon as the MIDI standard achieved widespread usage.

In fact there have since been so many extensions to the initial MIDI 1.0 standard of 1983 that some commentators have justifiably claimed that MIDI 2.0 is now here. The main extensions have introduced new sets of messages. Some of these facilitate MIDI control for completely new areas such as theatre lighting (MIDI 'Show Control' messages) or tape recorders (MIDI 'Machine Control' messages). Other new messages address perceived deficiencies of the original standard, such as MIDI 'tuning commands', which can set up an arbitrary tuning map in a device for each MIDI note number, or specify a tuning as each note is sent.

The most commonly criticised feature of MIDI is its relatively low bandwidth which is just adequate to convey musical performance nuances from a single monophonic instrument (Moore 1987), and the inherent timing errors which arise when attempting to communicate many contemporaneous values over a serial interface (as MIDI messages *have* to be one after the other).

Solutions which have been used include the use of multiple physical MIDI ports on computer-based devices, although most individual synthesiser devices still only have a single MIDI input port. Some recent devices, such as the Roland 'Sound Canvas' series of synthesisers, have two MIDI input ports, but these are merged internally. However, other recent MIDI devices, such as the Kawai 'K2' synthesiser, have two *independent* MIDI input ports, thus allowing *32* MIDI channels to be used on a single device.

Another attempted solution is to use a faster serial link designed to interface directly to a computer, and use a serial to multiple MIDI port converter device. Some recent MIDI devices (such as the aforementioned 'K2') include such a serial interface on the device itself.

Another idea is that of encoding the MIDI protocol within some faster standard such as SCSI, making it a lot faster and bi-directional on the same connection. But this would still not solve the other problems, and would require the upgrading of all existing equipment.

2.6.2. MIDI-LANs

A set of device and event specification sub-systems needs to be inter-connectable into a network, with every device interfacing to every other. This is problematic in a complex MIDI-connected system, as each link is unidirectional, and route switching or merging devices would be required.

The concept of a LAN system with local MIDI servers has been proposed (Mauchly 1987) in order to address these issues, amongst others. Each MIDI device connects to the LAN via a server which converts the LAN protocol messages to MIDI ones for the device.

Other servers could of course convert LAN messages into *other* custom formats for different non-MIDI standard devices, and a single control software subsystem could communicate to all devices in a uniform manner.

Such a system has been developed by Lone Wolf, USA, as the high speed 'Medialink' protocol which can communicate embedded MIDI, SMPTE, and digital audio data at rate between 30 and 3000 times faster than MIDI (Rona 1989; Westfall 1989), with 'MidiTap' and 'AudioTap' server devices converting data to/from MIDI and digital audio respectively. This system could form a practical basis for the system organisation proposal in chapter 4.

2.7. Time-stamped communication protocols

2.7.1. Concept

The system organisation concept described above in 2.3 entails a high-level 'event specification' subsystem which controls a distinct synthesis subsystem ('device'), communicating with it via a set of instructions or messages.

Most existing 'custom' systems integrate these two subsystems. The control subsystem usually incorporates a text-based language editor which allows a user to enter or algorithmically create time-stamped textual instructions. These are sent directly as function calls to the integrated synthesis engine when the instructions are compiled or interpreted. Examples of this communication might be a line of a CSound score file, a CHANT or Formula statement, or a MIDAS command line (see 3.2.1.9). Details of these systems will be given in 3.2 below. In the loosely-coupled system structure envisaged in this thesis, such communication would entail the user entering or creating such time-stamped instructions in an 'event specification' subsystem (a disparate high-level control system), which are then transferred to the device.

The only way of achieving this communication of time-stamped instructions with systems as *presently* existing would be for the event specification subsystem to write out the instructions or messages as a file in the appropriate format, and load this into the device. It should be remarked that present custom systems can be considered to be synthesis devices, even though they may have their *own* immediate user-entry facility for event specification.

This entry and transfer process is akin to a higher-level score processing or data entry subsystem such as 'S11Input' (Strasburger 1990), interfacing with the 'CSound' synthesis device. 'S11Input' produces a file of score instructions which has then to be transferred to 'CSound' (by saving to, and loading from disk).

The concept of the non real-time transfer of time-stamped event data from an event specification subsystem to a synthesis device subsystem may well raise questions: what is

the use of such communication of time-stamped control data, and how is it to be performed?

To answer the former question, consider the analogy with the current state of systems which communicate in real-time via MIDI. Because data can be interchanged easily between systems, both synthesis and control systems are free to each specialise in some aspect. An algorithmic software subsystem can simply output its MIDI events, without needing to provide graphic score printing or synthesis facilities. However, many systems have to deal in *time-stamped* data - primarily because of the complexity of their synthesis processes compels them to operate and be controlled in an 'off-line' mode (see 2.4.3.2). It would be desirable to be able to build such systems with similar specialised components to these real-time MIDI systems.

In answer to the latter question, protocols have been designed which are presented in chapter 5. The following section briefly outlines the time-stamped transfer capability existing at present within the MIDI standard.

2.7.2. Time-stamped communication within the MIDI protocol

MIDI facilitates the transfer of information in a time-stamped form, in a variety of ways.

2.7.2.1 MIDI standard files

These files provide a standard interchange format for communicating event control data, via files stored on disk in a standardised format. Files contain MIDI messages and their send times, either stored in a linear structure with all messages at the same level, or in a 'track' based format with messages grouped by MIDI channel number.

MIDI standard files thus enable a form of time-stamped communication between MIDI subsystems according to the outline in 2.7.1 above. For example, a MIDI *'note on' event* plus a *time* may be entered by a user in a textually-based event list editor of a MIDI-based "sequencer" application (an event specification subsystem). This time-stamped event must now be transferred to a device which can perform it. The only existing implementation of this concept would be *another* discrete MIDI sequencer software system. The transfer would be performed by saving the time-stamped events to a MIDI standard file on disk, then transferring and loading into the second system. This process would have to be done manually, and is rather laborious, and pointless in this case, as the system is designed for real-time performance sending out the messages over MIDI.

2.7.2.2 MIDI sample dump standard

This facilitates the communication of low-level digitally sampled sound data, ie soundfiles. However, the speed of MIDI data transfer (31.25 kbaud) implies a rate of only 1000 16-bit

samples per second[1]. For example, at 44.1 kHz sampling rate, it would take about 44 seconds for each second of soundfile. Thus, for example, a one minute soundfile would take 3/4 hr to transfer, which is too slow to be usable in practice.

2.7.2.3 MIDI time code 'cue lists'

The MIDI Time Code (MTC) specification includes the ability to transmit 'cue lists' (containing time offsets, each with an associated MIDI message). These cue lists can be downloaded via MIDI to devices which support this feature. The device can then be triggered to start reading its stored list, and each messages will be output from the device when its associated time offset is reached.

2.8. Event specification subsystems

As described in chapter 1, an 'event specification' sub-system includes the software user interface with which a composer interacts. The sub-system may also provide facilities for musical data input, editing and display, and the specification of compositional algorithms or programs (Loy 1989), which may be provided via textual and/or graphical interfaces.

In addition, systems may allow a composer-user (as opposed to a programmer-user) to construct or specify sound production algorithms as 'instrument' specifications for performance on an attached synthesis device or process (as defined in 2.4 above). The system is thus not simply outputting numbers or printed scores, but is exercising *control* over a device to enable performance of the score using the specified synthesised instruments.

Various general aspects of such systems can be distinguished, as described below.

2.8.1. User interface and presentation

The display and user-interface of computer-based event specification subsystems have historically been text-based, for several reasons.

Firstly, they have evolved from their origins as computer programs or conventional computer programs languages. For example, the PILE language of the late seventies grew out of the 'MACRO-15' assembler language for the PDP-15 computer (Berg 1985).

Secondly, the bit-mapped display hardware necessary for WIMP[2]-based GUIs has only become affordable in the last 15 years, and software development has historically always lagged behind hardware by several years (Thomas 1989a).

---

1 At 10 bits per MIDI byte, the transfer rate is ~ 3000 MIDI bytes per second. There are 7bits of sample data per MIDI byte, thus a 16 bit sample actually requires 3 MIDI bytes. Thus, the rate of transfer of sample data will be 1000 samples per second.

2 See glossary

Thirdly, there have been few portable graphics-based environments or standards until relatively recently, and developers have in some cases wished consciously to maintain the relative portability of text-based user interfaces (eg CSound).

2.8.1.1. Presentation of synthesis algorithms

Some systems facilitate the editing and display of synthesis structures and their input parameter values. They may present the user with a textual parameter list, or structure specification language but, increasingly, provide for interaction via a GUI display. This may display simple parameter values as before, but employ graphic on-screen scroll bars to display and allow user editing of values. Parameters which pertain to particular entities such as envelopes, may be presented via an analogous time-based graphic representation, whose parameters can be altered by manipulating elements (eg line segments) of the graphic analogy. Synthesis *structures* may be presented as pictures which may be selected by the composer.

For devices which support the flexible construction of synthesis structures, these may be presented using icons for each element of the structure. These elements may then be connected by the user in a variety of ways, by drawing corresponding links on screen using graphic 'wires' to connect the icons.

The presentation to the user for devices which do and do not support such flexible construction of synthesis structures is very different.

2.8.1.2 Presentation of event data

Recent event specification software, especially that which is MIDI-based has employed GUIs extensively. When used to display a musical score in the accepted sense, the conventional notion of time drawn horizontally is used with events' start and stop times being positions from left to right, and their parameter values (eg pitch) shown by their vertical position.

Events can also be shown as symbols on a staves according to the Common Practice Notation (CPN) convention (Loy 1985b). However, systems which are more concerned with the *sonic* nature of events (rather than as simple notes to be played on an arbitrary instrument) often use more abstract graphic displays which can provide more detailed information about the characteristics of each event, albeit with less immediate interpretability compared with CPN.

Such displays typically have an icon for each event, whose horizontal length correlates to its duration. The icon's horizontal position then indicates the event's time offset within the higher level structure represented by the display screen. The icon's vertical position represents some parameter value, most typically pitch. Other data pertaining to the events may also be displayed on the screen as one or more time-varying traces. This data may pertains to all the events in the display. Many MIDI-based commercial event specification subsystems provide this kind of display. Figure 5 below shows an example of this kind of graphic display from Opcode's 'EZ Vision' software.

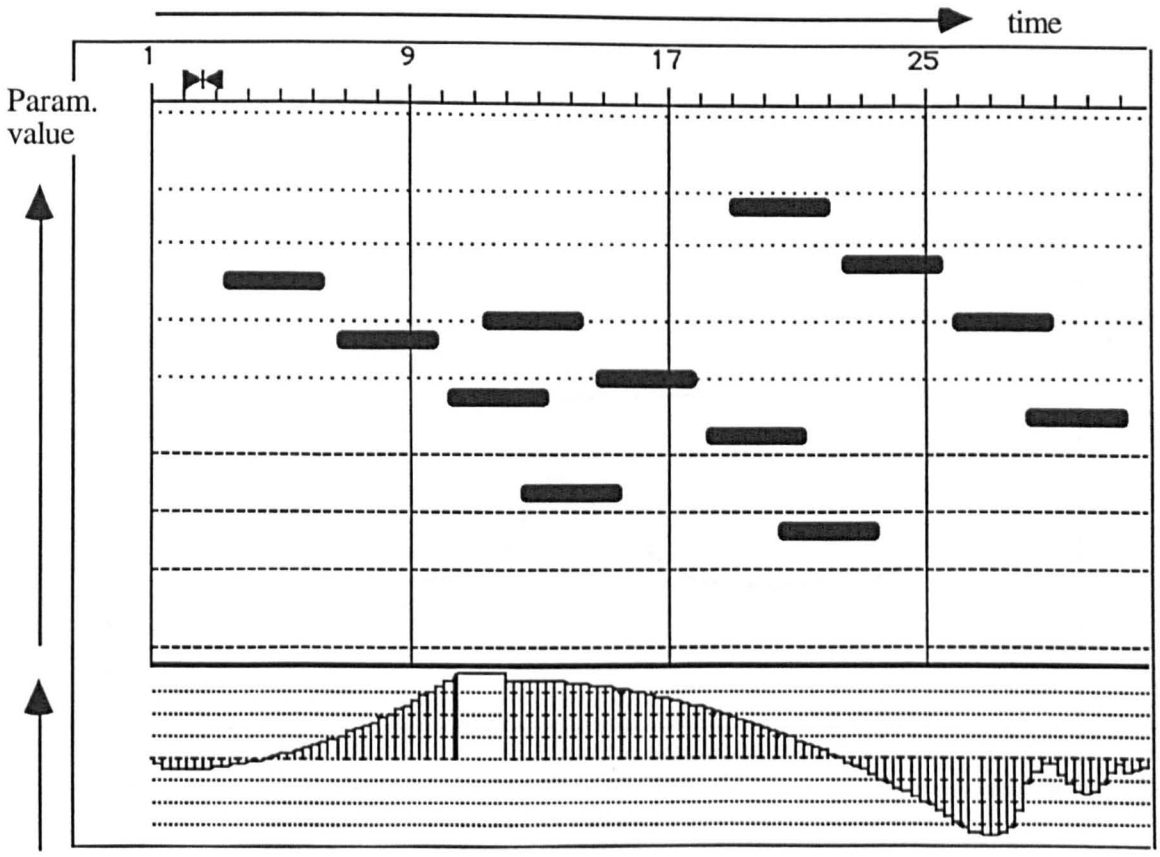


Fig. 5 Event display - events are shown as icons with a single individual parameter, plus globally applicable variable data

Other kinds of display on other kinds of system, may allow time-varying data to be specified for individual events. Again time is conventionally mapped to the horizontal dimension, with vertical position again usually correlating the pitch parameter values. Figure 6 below illustrates a generalised display of this type. Systems which employ this display mode include UPIC (see 3.2.1.4), and Freehand (see 3.2.1.12), although these only show a *single* event parameter in a display window.

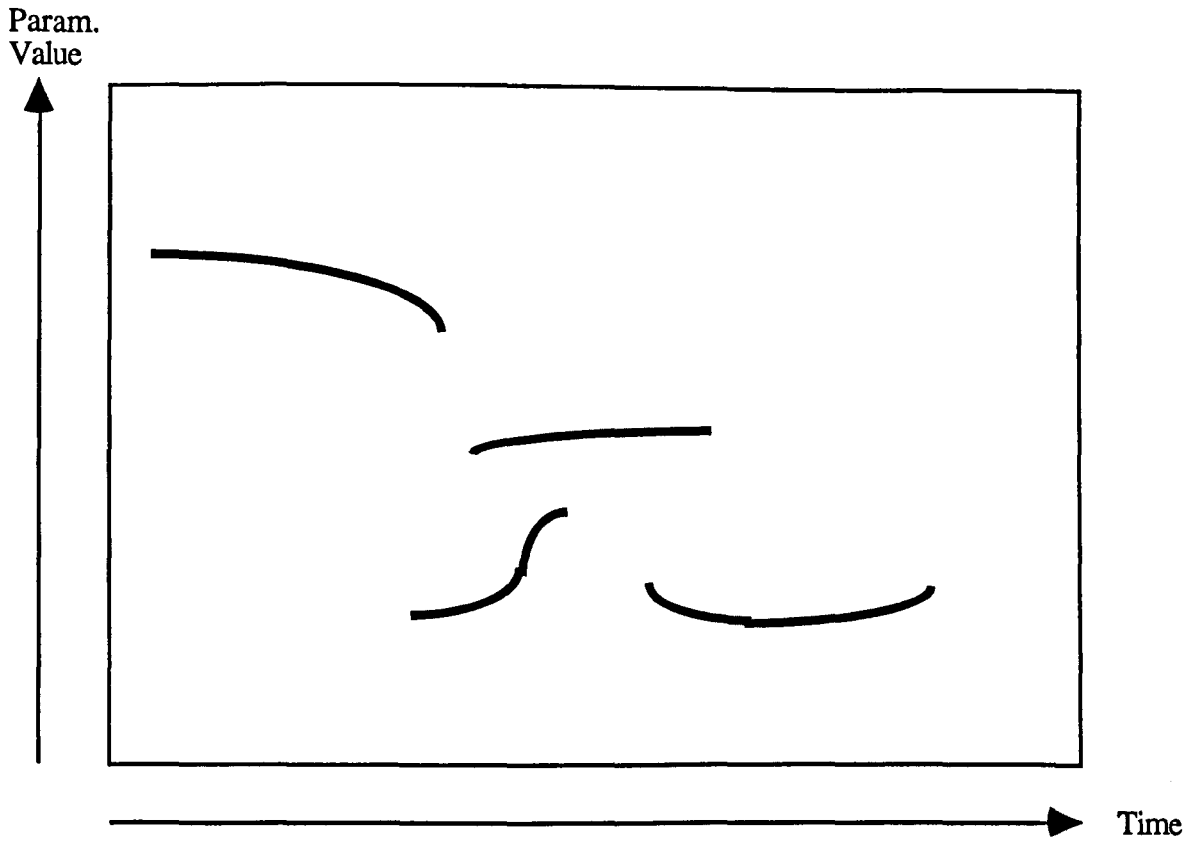


Fig. 6 Event display - events shown as traces
with time-varying parameter values for each event

Many other systems allow data to be stored in globally available in tables as a set of breakpoints (each having a time and value). This table data can be used by synthesis algorithms or be assigned to a particular event, but often is *not* usually displayable in the context of a score display.

Most user interfaces allow a user to manually specify events and their parameter values. These include such things as start time, duration, the synthesis structure or process ('instrument') the event uses, and various input parameters for this process.

The composer may specify such information via a direct manipulation GUI, using a mouse cursor to draw events on the display, with parameter values generated by horizontal and vertical position. A facility for textual specification of numerical parameter values may be provided, instead of or as well as this GUI.

Events and their parameter values may also be generated using algorithmic methods. The composer may specify such algorithms textually via a programming language, or by connecting icons (via a GUI) which represent data flow through primitive operations, in a similar fashion as for the construction of synthesis structures described in 2.8.1.2 above.

2.8.2. Event structuring and manipulation

Events can be organised into structures by nesting them within a higher-level entity. Some systems facilitate this kind of organisation in a flexible fashion, allowing any level of nesting. Others (mainly MIDI-based) impose a framework of organisation on events, typically into single-level structures which may then be manipulated relative to each other, and played together.

Events and their parameters can then be altered by the user in a variety of ways:

• Events, or groups of events at various levels of a structure can be deleted, or copied to different times or locations within the structure.

• Textually-presented event parameters may be easily altered in the manner of a word processor.

• Graphically-presented icons can be moved horizontally or vertically on the score display (usually using the mouse). Moving an icon horizontally will typically have the effect of changing the start time of the event.

• Altering the horizontal length of an event's icon will typically change its duration; the mapping between icon length and this duration will depend on the horizontal time-scale of the graphic event display.

• Moving an event's icon vertically on the screen usually alters the value of a parameter of the event, usually pitch.

• Algorithmic methods may also be used to alter event data, or to select groups of events via logical criteria. These algorithms can be specified via a programming language or connection of GUI icons in the same manner as described above in 2.8.1.1.

2.8.3. Performance

For real-time control, the synthesis process control data which corresponds to event parameter data is sent at the appropriate time as messages or functions calls to associated synthesis devices which create sound.

For time-stamped control, a set of messages which incorporate time offsets instructions can be sent to a device, or textual data or functions can be written to a file for subsequent loading and acting on by a device.

3. A survey and critique of existing composition and performance systems

This chapter examines a representative cross-section of existing computer-based music systems of various types, which facilitate composition using synthetic sound. Such systems include an event specification software subsystem, and synthesis devices which are employed to generate sound. This is not an exhaustive survey of all languages and devices, but instead focuses on representational examples of different approaches, and an emphasis on systems which facilitate user *interaction* via a graphical or textual interface, rather than simple programming languages. This survey sets the scene for a consolidated critique of current systems, with an analytical overview of areas of concern.

The survey is divided into MIDI-based (mostly commercial) systems, and other 'custom' systems which mostly exist as unique systems within research or academic institutions. Both categories of system are important and prevalent, yet very different in characteristics and capabilities. It may be re-iterated here that one of the outcomes of this research is the proposal of new standards which should help to break down this barrier between the flexibility of custom systems, and the availability of MIDI-based ones.

3.1. MIDI-oriented systems

MIDI-oriented computer systems typically contain a software subsystem which facilitates event specification, and control commercial synthesis devices using the MIDI protocol. Such systems have many other facilities connected with recording human performances to capture event data. These other aspects are beyond the scope of this project. Some 'custom' systems include MIDI facilities, but are not included in this section, as they also incorporate facilities for control of 'custom' synthesis devices, and hence have a diversity of features which makes it more appropriate to consider them individually (see 3.2).

Such systems are based around the presumption that all communication from an event specification subsystem to devices uses the MIDI protocol. These devices have various features which reflect their use of MIDI control. There are many event specification software control systems available commercially, which assume these device features, and are not typically built around particular MIDI synthesis devices.

It thus makes sense to divide the following analysis of MIDI-based systems into synthesis devices and control software.

3.1.1 MIDI-controlled synthesis devices

MIDI is a communication standard; thus MIDI devices have to interact with MIDI in certain set manners, and have evolved particular types of structure and behaviour which reflect their involvement with the MIDI protocol. The following features of MIDI-controlled devices are relevant to their use by composition systems:-

3.1.1.1 Specification of synthesis processes

A network of primitive synthesis processes ('units') can be specified within a device, along with default input values. This specification can usually be stored in memory within the device, and is usually referred to as a 'patch' or 'program'. Any connections between units are usually pre-ordained; either completely fixed, or selectable by the user from a small number of possibilities. The patch is described by a set of numbers which define the values of its input parameters. Some of these parameters can control the instantiation of these units, with different units being invoked by setting a parameter to a particular value.

MIDI-based devices have a fixed memory map, which has fixed locations in which patches (sets of units) can notionally 'exist'. These locations are termed device 'slots' for the purposes of discussion in this thesis. The device's memory map has offsets for each slot, within which further offsets then exist for each input parameter of any patch instantiated in that slot.

A patch specification can be instantiated in a device, ie called into being according to a definition described by a template or specification. The instantiated structure is then installed in the device, either into newly-allocated device memory, or into a set location - some form of 'current' memory area. Once installed in the device, the structure becomes active, and will have a set address for each of its input parameters. Units within the structure[1] can then be addressed via messages which use the id and/or address of the slot they are in (known as a MIDI 'channel') rather than by an individual id number for that unit.

As stated above, different units can be invoked within a patch by setting one of its input parameters to a particular value, and the patch itself can be notionally instantiated within a device slot by sending one of more messages to the device to recall the patch specification from storage. However patches (or their component units) are not really *independent* entities existing within the device is the conventional sense.

A patch's control inputs are accessed via messages which pertain to (and use data from) the characteristics of the *slot* the unit is in, rather than the patch itself. For example, a patch existing in a device slot may have an input parameter which is accessed by sending a particular MIDI message to a specific address in the device. This patch might then be replaced by a *different* patch which does not have this input. However, the address and the message are still defined, as they pertain to the *slot* (even though the new unit cannot understand such a message). Thus a data value is sent to a device with a message which incorporates an address, or the id or 'channel' of a slot within the device. There will then be a synthesis process notionally present in that slot which may or may not respond to this data.

Thus, there is no sense in which objects are really instantiated (within a blank structure); the most that can be said is that in some devices, parameters can be sent to select *which* object

---

[1] A 'structure' may just consist of a single unit.

to install in a certain location. MIDI based commercial devices are *not* object-oriented in their structure or usage (see chapter 7 for a discussion of the meaning of 'object-oriented').

3.1.1.2 Recall of patch specifications

As described above, a stored patch definition - a specification of a network of synthesis units - can be loaded into a device's 'current' memory buffer, from where it can be run, to perform sound events. This loading can be initiated by a user from the device's front panel, or can be controlled remotely, typically by sending a MIDI 'program change' message (see 2.5.2.1). This message, however, is limited to selecting from only *128* different stored patches. Recent synthesiser devices with more memory and more (eg 384, or 1000) patches thus either need to use the new[1] MIDI bank select message, or an 'unofficial' combination of MIDI messages not envisaged in the MIDI standard. An example of such an 'unofficial' method is found in the Oberheim 'Matrix 1000' device (Oberheim 1988) which has 1000 patches stored in ROM and RAM, in 10 banks of 100. It can be switched into a 'bank select' mode (ie a received program change then changes the *bank* number) if a value greater than 64 is received for the MIDI controller #1 ('modulation wheel') message.

It should be remembered that patch selection messages in fact recall and set up *structures* as well as default input values to those structures. In addition to sending messages to recall a 'patch' specification from storage in a device, the patch can usually also be downloaded directly to the device's current memory buffer by transmitting a set of data values via MIDI system exclusive messages for the device. This is commonly known as a 'bulk dump'.

3.1.1.3 Device structure

One or more device slots - memory areas in which patch structures can be active - are often available in a device. The same or different patches can then be loaded each slots in the device, which can thus be treated as a set of independent devices. A device with this feature is known as 'multi-timbral' in the commercial MIDI environment.

Alternatively, such a device can be considered to be a *single* device which can instantiate structures more complex than those normally envisaged. This requires that patch structures in *several* device slots should be able to be treated by a event specification system as a *single* complex entity. This is one of the design goals of the E-Scape system (see chapter 6), in order to enable composers to flexibly be able to create more complex timbres from the available synthesis devices.

3.1.1.4 Polyphonic messages

Most MIDI devices support the playing of several 'note' events on a *single* synthesis structure, ie a single 'patch' (or 'instrument') installed in a single device 'slot' addressed via a single MIDI channel. They do this using 'polyphonic' messages which can specify

---

[1] As was stated in 2.6.1, many extensions to the original MIDI specification have been made. This new 'bank select' message uses the MIDI 'controller' message, with a defined controller number of 0.

different values for events on the *same* structure. This contrasts with systems such as CSound, which overtly create a new instrument allocation, each time one is needed (ie when a new event needs to be played, but all existing instances are currently busy playing existing events. Obviously, the low-level processes being carried out within the MIDI device are allocating events to synthesis resources, but the presentation to the 'user' (composition software or performer) is of a single entity which can support multiple events. However, using MIDI, the parameters which can be *different* for individual events on the same instrument are restricted to note number, note on and off velocity, and polyphonic aftertouch, only the latter of which is continuously variable.

Thus, overlapping events with *different* varying parameters which use the *same* instrument entity within a MIDI device are *possible* in a limited way (ie only with *one* variable parameter) controlled by the 'polyphonic aftertouch' message. However, this message is not usually implemented by devices. This is probably due to a perceived lack of need for such messages, as the concept of using MIDI to 'perform' data into a composition system is still very prevalent. Thus, the fact that a human performer would have great difficulty in creating precise MIDI 'aftertouch' data (by pressure on individual notes on a MIDI keyboard) has led to a general lack of provision for this message by devices.

3.1.1.5 Non-polyphonic messages

Most MIDI messages are 'non-polyphonic', ie they affect *all* events (notes) which use an instrument on a single channel. Examples include MIDI 'pitch bend', aftertouch', and 'controller' messages. Thus, overlapping events (which require *different* values for these messages) would require the creation of multiple instruments (in multiple device slots) which use *different* MIDI channels.

3.1.1.6 Hardware communication facilities

There is usually only *one* physical MIDI input port on synthesis devices, with the resultant limitations from the inherent MIDI bandwidth restrictions. However, several recent devices (such as the Roland 'Sound Canvas' and Kawai 'K2' synthesisers) have two MIDI inputs; others (such as the Yamaha 'TG100') have fast 1MHz serial ports which can interface directly to a computer, while still using the MIDI message format. The computer software has to know about such interfaces, but if so, can be set to a 'fast' MIDI mode.

In most cases, however, present devices[1] are still limited to receiving only 16 channels of independent information for messages which are 'non-polyphonic' (see 3.1.1.5 above).

---

[1] The sole exception at the time of writing is the Kawai 'K2' whose two MIDI input ports are *independent*, allowing 32 MIDI channels to be addressed.

3.1.2 MIDI event specification systems

Such systems exist mainly in the commercial domain - many are extensively used in the pop/rock music business for their event recording and structure ('arranging') facilities. Their focus is mainly on relatively simple 'note' events, ie with little concern for timbral control over the event. Some systems provide facilities enabling display and specification of any MIDI message, including system exclusive, although graphical displays are mainly built around viewing notes or block of notes with their 'pitch' and 'velocity' data field values. Other types of message are less supported, often with arbitrary restrictions, such as only having one 'non-polyphonic' message displayable at a time, or only messages using a single MIDI channel.

Many commercial systems have the graphical time-based event displays described in 2.8.1.2, and may also include CPN score displays, numerical event lists, or other idiosyncratic display formats. They allow event data to be entered by a user, or recorded from a live musical performance. They can include some algorithmic operations which can be used to mutate or alter data, but do not usually focus on *generating* events from scratch.

Other systems may have little or no emphasis on visual display and recording aspects, and are more text-based, allowing a composer to create and structure events and other MIDI data using functional programming via a specialised language.

Several representational examples will be briefly described.

3.1.2.1 Symbolic Composer

Symbolic Composer (Tollinen 1993) is a language-based composition environment built on Common LISP. It follows in the traditions of other languages and systems which use the LISP language as their basis, such as MIDI-LISP (Wessel 1987), Le_Lisp and Le_Loup (Duthen 1987), and preFORM (Boynton 1986a). Symbolic Composer provides extensive tools for music-oriented structuring and data processing, via numerous data generator and converter functions which can operate at a symbolic or numeric level. It also offers some graphical visualisation facilities for viewing data patterns and defining score events. The system can output data in the form of MIDI messages or standard MIDI files, and can also create wavetable files which may be used to construct CSound score files (see 3.2.1.5).

3.1.2.2 Formula

Formula (Anderson 1991) is a language based on Forth programming language. It supports event-based algorithmic processes which create output MIDI messages. Processes include 'shapes' (similar to those in HMSL - see 3.2.1.8), time deformations, note and sequence playing, and input handling (including MIDI). Its strengths are in creating interactive instruments, score interpreting, and algorithmic composition.

Formula has no pretensions to real-time event recording or graphic editing, and provides no score display or storage mechanisms.

3.1.2.3. Notator

'Notator' (Lengeling 1992) has extensive facilities for recording, and event entry and editing via a CPN score display and/or time-based grid. It also provides a graphic display of many MIDI message values at once.

Events can be selected into a group for editing purposes, and have editing operations performed on their parameters via various simple rules selected via a GUI.

3.1.2.4. KCS Omega

KCS Omega (Tobenfeld 1984; Tobenfeld 1993) also has some of the event recording and graphical display facilities of Notator, but in addition allows the use of the computer keyboard (and other data-entry devices) for real-time control of musical structure. It has a modular design, allowing a wide range of external application modules to be used within its shell. These include CPN score displays, multi-track graphic editing, and an algorithmic composition package. It also provides extensive facilities for the generation and processing of MIDI data via built in functions, and a programming module (with a variant of the BASIC language) which allows users to program their own algorithms.

3.1.2.5. Dmix

'Dmix' (Oppenheim 1989; Oppenheim 1992) is a large compositional system written within the Smalltalk-80 object-oriented language environment (see chapter 7). It enables the structuring of MIDI events into arbitrary groupings, provides extensive graphic display and manipulation facilities for events and time-varying data patterns. It can abstract such data from parameters of MIDI events, or apply data to the parameters or timings of events, via user-definable operations. Real-time human performance recording and processing, GUI-based event entry, text-based editing and algorithmic event generation (Oppenheim 1990) are all supported, and free and intuitive inter-conversion of data between these various modes of working is possible. In addition the power of the 'raw' Smalltalk-80 programming language is always available and can be combined with the existing Dmix functionality. At present Dmix deals exclusively with MIDI, and it very much 'event oriented', with limited support for control of MIDI system exclusive messages and continuously variable parameters.

3.2. 'Custom' systems

Custom systems mainly exist in the academic music world. As described in 2.3, they typically consist of an event specification subsystem which interfaces with custom-designed synthesis devices using custom protocols of functions. Some systems do also include the provision of MIDI control, by providing the ability to output MIDI messages.

Such systems are more integrated within themselves than MIDI-based systems, with relatively little interchangeability between components of different systems. Hence, it makes sense to deal with systems as a whole, rather than treat devices and control subsystems separately.

3.2.1. Example systems

A number of different systems have been examined as representational examples.

3.2.1.1 MAX

Concept

MAX is a generalised, customisable, hierarchical, real-time data processing system. Its operation and presentation is in terms of primitive modules or objects which represent signal processing algorithms. Many basic such algorithms are provided, and new ones can be built from these by the user. However, MAX cannot be said to be an object-oriented system (see below).

It comes in two variants (running on different platforms) which are not completely compatible, having proceeded along separate development paths for some years with Opcode Systems Inc (Dobrian 1990), and IRCAM (Puckette 1991c).

The *IRCAM* version of MAX runs on the IRCAM Signal Processing Workstation (ISPW) which consists of a NeXT host computer, communicating with up to three plug-in boards each with two Intel i860 processors, and a Motorola DSP56001 processor (Lindemann 1990; Lindemann 1991a). It has a real-time operating system - 'CPOS' - which provides a set of system calls to access processes and memory on the attached processors (Viara 1990; Viara 1991). In addition, the 'FTS' toolbox (Puckette 1991a) is an intermediate software layer which enables processes running on the processor board to be treated as objects. FTS provides functions that support the creating and deleting of such process objects, and the sending and receiving of messages between objects.

MAX can control and specify low-level DSP operations on these boards and specify signal flows which are at audio rate. It can thus build instrument-like structures which map to processes in the DSP boards.

The *Opcode* version of MAX runs on the Apple Macintosh computer only, and controls external synthesis hardware using the MIDI protocol as standard. External C code can be used to interface to other ports and create new processing objects, and the internal data representation is not bound to the MIDI format, hence it could be adapted by a C programmer to use some other protocol. Extensions have recently been created to enable MAX to control DSP algorithms in cards attached to the Macintosh.

Networks describe control-rate data flows between data processing objects within the MAX software. Hardware synthesis devices are then controlled by creating suitable MIDI messages which are output from the host.

The networks themselves do not map to a specification of external objects instantiated in hardware. This is not to say that MAX cannot set up synthesis structures - objects can store blocks of MIDI system exclusive data which do just this, but there is no way of generating such data by the *act* of connecting lower-level modules. Connections only describe data flow within the MAX software.

Comments

MAX at present has little support for score display or entry, and only limited graphic time-based functions (at present a single drawing screen). Its GUI is very much geared towards real-time event and data processing and recording. More a generalised graphical *language*, it supports various structures but does not provide templates for constructing particular structures or allowing specific ways of working. For example there is no way of creating score event structures with parameters, other than the supplied MIDI or DSP formats.

Processing structures can be nested in MAX, but there is little indication of the *purpose* of an input to a module, ie what *range* or type of data can be sent to it, and what its effect will be. This lessens the benefit of such modular construction - if the user is required to know what is inside a module, then the effectiveness of using modules to simplify the task of creating complex structures is reduced.

As described above, MAX allows a user to build structures from objects, but is not designed for full object-oriented interaction. For example, one can *send* data to an input of an object, but cannot send it a message asking it for information, or ask it to build and load data structures while running. Pope and others (Desain 1993) have asserted that MAX is not an object-oriented graphical programming language at all (as advertised by Opcode). It does not fit with any of the defining features of an object-oriented programming system (see chapter 7), and is more of a data-flow configuration tool. Section 11.3.4 contains further discussion on the nature and usage of MAX.

3.2.1.2 MusicKit

The Music Kit (Jaffe 1989a; Jaffe 1989b) is a low-level DSP process manager running on the NeXT computer platform. It facilitates the definition of instruments - synthesis unit networks - as object classes (see chapter 7).

The Music Kit provides portable, high-level access for applications (written using the NeXT's interface-building tools and Objective-C) to synthesis structures, on the internal 56001 DSP chip (Smith 1989) or external synthesisers via MIDI. An example is the 'SynthEdit' GUI-based application (Minnick 1990) which facilitates user construction of unit generator algorithms, which run on the DSP via the MusicKit. SynthEdit also manages the allocation of appropriate objects within the MusicKit system to enable the performance of score files.

Many other applications are in existence on the NeXT - too many to list (Jaffe 1991), providing a powerful integrated suite of tools, all of which however are tied to running on the NeXT computer. External control of the NeXT's DSP by other systems is not facilitated.

3.2.1.3 Kyma / Capybara

The Symbolic Sound Corporation's 'Kyma/Capybara' system (Scaletti 1987; Scaletti 1988; Scaletti 1989; Scaletti 1992) is a commercial loosely-coupled system. 'Kyma' is the high-level control software component which is totally object-oriented, as it is implemented within the Smalltalk-80 programming language[1]. The emphasis is very much on user construction of synthesis, sound and event-level processing algorithms (via an iconic MAX-like GUI). These can be built into hierarchical structures which can control the generation or processing of sound in real-time or off-line.

> "A Kyma score is not a data file, but a Smalltalk-80 program. Events can be generated algorithmically, directly in the score" (Symbolic-Sound-Corporation 1992).

All entities are 'Sound' objects, which can be of two types. An 'atomic Sound' is a source of sample data (read from a file, table or real-time ADC input). A 'composite Sound' performs processing functions on other Sound objects (termed 'sub Sounds'). A 'composite Sound' can also contain time offsets, and sequence information about its child 'sub Sounds'.

> "Sound objects are used to define all levels of a composition, from the micro-structure of timbre, to the macro-structure of an entire composition" (Symbolic-Sound-Corporation 1992).

A user is also able to create new classes of 'composite Sound' object by "lifting" an existing Sound object - ie copying its structure and generalising it to form a description of a *class*[2] of Sound objects (Scaletti 1991).

Kyma also supports external MIDI control of any of its processes, and can output MIDI messages, but is otherwise dedicated to controlling low-level processes within one to nine 'Capybara' 56001 DSP cards which are attached to the host on which the Kyma software runs. It provides no graphic scoring capability, being

> "...primarily a language for sound synthesis and processing; it also provides basic ... tools for graphic editing of digital recordings" (Symbolic-Sound-Corporation 1992).

The Capybara synthesiser might have been a good candidate for control by the E-Scape prototype event specification system as a synthesis device, as it already has an interface to the Smalltalk environment. Unfortunately, the commercial nature of the enterprise has resulted in a refusal to divulge any of the low-level communication protocols used.

---

1 See chapter 7 for a description of object-oriented systems and Smalltalk-80.

2 Again, see chapter 7 for a detailed description of such object-oriented terms.

3.2.1.4 UPIC

UPIC is a system developed at the CEMAMu institution, consisting of a software GUI-based composition subsystem interfaced to a hardware-based synthesis subsystem. This provides for real-time control of specific synthesis processes (64 wavetable oscillators with FM) in DSP-based hardware modules (Raczinski 1988).

A user can specify or access waveform libraries, and draw and display frequency or waveform parameters graphically on a score display, enabling a composer to specify structure at the macro and micro-compositional level in a uniform manner (Lohner 1986).

A user can create mass structures graphically, and replicate and transform them graphically as a single entity (Pape 1992). However, the event traces are always presented and organised on a single level, ie there is no facility to nest events within hierarchies, which can then be treated as a single object.

Melodic shapes can be transformed graphically, eg by stretching the graphic representation. A frequency table can be used to map vertical position on the graphic score display to specific pitch values. A composer can thus work within a continuous, or a fragmented pitch space as desired, and use various scales which may be micro-tonal (Marino 1990). Thus the same score can be mapped to different pitch spaces instantly, but this results in a discrepancy (an inconsistent mapping) between the score as displayed, and what is heard. The prototype E-Scape software implements this concept differently (see 8.4.4.1), in order to present scores in a "wysiwyh" manner - what you see is what you *hear*!

The UPIC software also implements the concept of a 'sequence' which specifies how the horizontal time dimension of the score is to be interpreted. Rather than the normal linear traversal from left to right, the user can draw a function which specifies how the score is to be traversed in time - eg to loop, scroll back and forth.

3.2.1.5 CSound

CSound is a software synthesis system which uses a textual user interface, allowing a user to specify *instrument* and *score* definition files in a proprietary format (Vercoe 1986). An instrument definition consists of a textually specified network of synthesis units (known as 'unit generators') to which external values can be sent from a score. A score consists of textually specified events: for each is specified a start time, an instrument, a duration and values for each instrument parameter (known as a 'p-field').

This score is then parsed, and audio rate data generated in software according to the network of processes - unit generators - specified in the instrument definition. This processing may be performed in real-time (if the computer platform has enough processing speed) to produce immediate sound output, or in off-line mode to create a soundfile (see 2.4.2). In addition, unit generators can be used which interpret MIDI messages; these can initiate score events and receive control data in the same way it would be from a p-field of a score file. Thus both real-time and time-stamped control (see 2.4.3.2), as well as real-time and off-line synthesis operation (see 2.4.2) are possible.

Many instrument input parameters are often necessary to produce sounds of musical subtlety and interest, but this makes it laborious to write a score, as many values have to be specified for each event, and there is no provision for default values.

It is difficult to 'try out' an instrument, since the user needs to know the meaning of, and sensible values for, each instrument input. Thus, instruments are often difficult to use by others who have not been involved in their design without a large amount of detective work. As *all* parameters need specifying, a composer cannot pick up a new instrument and gradually experiment with altered values. This could only be achieved if a carefully prepared example score were to be provided which could then be gradually altered.

A variety of GUI-driven instrument specification systems have been developed to create CSound instrument specification files, such as 'Patchwork' (see 3.2.1.7 below).

Various other applications, such as 'S11Input' (Strasburger 1990) can generate or process score files, often derived from MIDI-based performance or file data.

3.2.1.6 CHANT

CHANT is a software synthesis system (Barrière 1991) containing functions written in C which implement FOF[1] (formant wave function) synthesis algorithms. It is embedded within the Common Lisp environment, and can use Lisp or Scheme (a dialect of Lisp) code for high-level specification (eg to specify fof banks, filters or other sound sources and start and stop play). Now quite portable, it can run on several computer platforms. It shares the advantages and disadvantages of systems based on a computer language, as detailed in 3.4.

3.2.1.7 Patchwork / Esquisse

Patchwork (Laurson 1989; Pinkston 1991) is a MAX-like system for creating and linking functional modules using a GUI, now implemented in the Common Lisp language. It includes such things as editors to create functions (abstract 2D data patterns) which can be used at any level.

'Esquisse' (Duthen 1990) is a library of compositional rule modules for generating and manipulating musical material at the note level. It has now been integrated with the Patchwork system, and modules can be mixed at will, with data used at the event or sound creation level.

The software system can be used to specify CHANT synthesis structures, generate score and orchestra files to control CSound synthesis, or control DSP-based synthesisers (Barrière 1991).

---

1 FOF is an abbreviation in French for 'formant wave function'.

3.2.1.8 HMSL

HMSL (Hierarchical Music Specification Language) is a programming language written within the object-oriented 'ODE' Object Development Environment (Polansky 1987). This in turn is written in the FORTH computer language, and a user can write program code at all these three levels.

It allows flexible hierarchical organisation of object-oriented data structures in a very general and unconstrained way. Objects of various types can be constructed:-

- 'shape' objects are abstract n-dimensional data patterns;
- 'collection' objects schedule other sub-objects;
- 'structure' and 'Tstructure' objects use intelligent 'behaviour' to alter the execution of child sub-objects;
- 'production' objects process and transform 'shape' objects.
- 'job' objects are specialised 'productions' which understand time and can output other objects as time-based streams.
- 'instrument' objects can translate the general HMSL data structures into the form needed by specific output devices such as MIDI, graphics, or local computer sound generation.
- 'player' objects are 'jobs' which have an associated instrument and shapes.

Complex processing of shapes is possible, such as non-linear transformations (to provide such things as interval augmentation or ornamentation), or stochastic evolution (used, for example, to mutate an origin shape into a target shape, using an algorithm with changing scale factors and random variables).

The latest version of HMSL (Burk 1991) has routines built into it to allow user specification (via a macro language) and downloading of 56000 code. It can connect to a variety of DSP cards (sited in the host computer) in a relatively independent way, by using device drivers. But code has to be *built in* to HMSL to interface to these drivers, and any new drivers needed for different DSP hardware will need additional code adding to HMSL. This is an inevitable by-product of the DSP boards being interfaced directly to the computer, rather than via a defined protocol with drivers at each end.

3.2.1.9 MIDAS

MIDAS (Musical Instrument Digital Array Signal Processor) is an expandable system supporting a network of processors connected in a fast LAN-like 'ring' system (Kirk 1990). Processors communicate via defined protocols, which enable higher-level applications to be insulated from the low-level processor architecture. Thus new processors can be added to the system with minimal high-level impact.

MIDAS can implement synthesis processes as a set of distributed units called 'unit generator processes' (UGPs) - analogous to CSound's unit generators - which can be linked to form arbitrarily complex networks. UGPs communicate with each other via messages on the MIDAS ring. Scheduling UGPs provide the means for the device to store

time-stamped messages destined for UGPs. This can include starting other scheduler UGPs running, thus time-based data structures can be downloaded if desired.

Functions are provided for applications compiled within the MIDAS environment (McGilly 1992) which allow a high-level compositional system to be integrated into the system directly via low-level function calls (which create the ring messages) if desired.

However, a further level of insulation from the precise system architecture (including address and processor locations and resource allocation) is provided by an interactive application called MII (Midas Intermediate Interface). This provides an intelligent interface to the low-level messages and includes a database of the addresses and states of UGPs on the ring. A high-level system can thus address all entities in MIDAS via an id (Anderson et. al. 1992a), and can request functionality such as storage and scheduling of event or control data.

Messages can be sent to MII in time-stamped or real-time form. The latter are acted upon immediately by MIDAS to generate sound; the former may be sent and processed at a lower rate than is necessary for sound output, with synthesis then occurring off-line.

In addition, it is planned for MII to implement the communication standards proposed and described in chapters 4 and 5. A design for how MII and MIDAS will implement the message types within this standard has been developed as part of the work described in this thesis (see chapter 11).

3.2.1.10 Accelerando

Accelerando (Lent 1989) uses the 'Patchwork' high-level symbolic compiler (see 3.2.1.7) to convert GUI-based iconic specifications of synthesis algorithms into textual descriptions. These form an intermediate 'Music56000' software level with unit generators described as macros in 56000 DSP assembler language. Instruments can thus be specified at the graphic or text level.

The synthesis hardware consists of a stand-alone 56001 DSP-based box with standard interfaces for control and audio data: MIDI, Yamaha digital I/O, plus a parallel I/O port for optional host computer connection.

3.2.1.11 Cmix / Patchmix

Cmix is a software synthesis and programming system akin to CSound, except that it requires a user to write 'C' language code. Patchmix (Helmuth 1990) is a GUI-based instrument designer for Cmix - akin to SynthEdit (see 3.2.1.2), and Patchwork (see 3.2.1.7). A distinguishing feature of Patchmix is its provision of default values for unit generator inputs, allowing default instruments to be quickly built. However, Patchmix units cannot be built up into modules for re-use, and thus these default values cannot be inherited by the inputs of the higher-level module[1].

---

1 E-Scape *does* implement this feature - see 8.6.4.2.

3.2.1.12 Freehand

Freehand (Orton, Kirk 1990) has an event specification subsystem which uses a GUI to allow a composer to draw events as graphic traces, with a single time-varying parameter (usually pitch) shown in the vertical dimension. Freehand then creates data tables from these event parameters, which can be used as input files for an additive synthesis application or in the creation of CSound score files. Freehand is thus one of the few graphical scoring systems (along with UPIC and E-Scape) which displays time-varying parameters which pertain to individual events, although only a single parameter can be used.

3.2.1.13 Gnot music project

The Gnot music project (Kahrs 1992) consists of a set of circuit boards supporting generic DSP chips (various synthesis algorithms), custom FM chips, a digital mixer, and MIDI and SMPTE i/o. All are linked by an 8 bit bus which can receive (via a decoding card) control signals from workstation control software with real-time scheduling software which is transparent to the high-level user software. However, the latter is relatively crude, with only basic note pitch generation, but could be adapted to enable the system as a whole be used as a synthesis device as described in 4.3.

3.2.1.14 SSSP

The SSSP (Structured Sound Synthesis Project) system was in many ways ahead of its time in operation and user-interface conception and presentation. It employed several (fixed) synthesis methods implemented in hardware (Buxton 1978)with a variety of innovative graphic and textual editors, and C language routines, all operating on the same event and voice data structures (Buxton 1979). The user interface provided a menu driven system couched in musical language, and facilitated easy adaptation to user preferences. Icons for each event could be placed on a pitch- and time-based scoring grid in the 'SCRIVA' graphic editor.

There was generous use of defaults (eg for rhythms or note parameters), enabling users to concentrate on selected aspects of a composition. In addition, real-time user inputs on a variety of transducers could be assigned, each to a *set* of event parameters (Pennycook 1985). This excellent and well-loved system (Pope 1992b) was nevertheless dependent on its hardware (both the synthesis hardware device, and the user interface system host computer) which was limited, and which became unmaintainable after a few years (Free 1986).

3.2.1.15 SORT MACHINE

The SORT MACHINE (Nottoli 1986) is a real time fully programmable computer music system. It is completely modular, with a host computer and synthesis modules connected via a PC host communication bus, and a DSP bus.

System-level software can be written using low-level function calls which support sound synthesis, managed by a modular control program (MSYS), which provides defined

procedures and access. The system also has several high-level programs for entry of score level data, and the definition of software sound objects which define synthesis methods.

3.2.1.16 MODE

The MODE (Musical Object Development Environment) is a software environment which provides tools to facilitate access to soundfiles, DSP boards, and MIDI, and enables event structures to be constructed (Pope 1992b). It can be extended by writing Smalltalk-80 programming language code. Further details will be given in the discussion of the object-oriented paradigm for musical purposes in chapter 7.

3.2.1.17 Others

Other systems can be mentioned to give a flavour of the variety available:-

• POD (an acronym for Poisson Distribution) was a synthesis system using a host-based user interface, and connected synthesis hardware (Walraff 1979). It allowed a user to specify time-varying functions which control probability distributions (Dodge 1985, p266) for the specific event parameters of pitch, amplitude, event density, synthesis routine and spatial distribution (Pennycook 1985). These 'tendency masks' (Loy 1985b) allow a 'top-down' approach to composition, first specifying large scale parameters, then refining the inner details.

• MP1 is a FORTRAN-based non-interactive computer program (Tipei 1987) which supports stochastic and deterministic procedures (Dodge 1985, chapter 8).

• Fugue (Dannenberg 1991) is a functional language which views music as a *process* rather than as a simple series of notes. Thus a score is described by a series of language expressions, and the *same* language can be used to describe both scores and synthesis structures (instruments).

3.3. A critique of MIDI-based systems

The limitations incurred when communicating information using MIDI protocol have been well documented, eg (Moore 1987), and only additional relevant comments will be made here.

MIDI's origins as a protocol to facilitate communication of a keyboard performance by a single performer led to an initially limited vision of its uses, and have forced it to be continually adapted to cope with such uses as multi-track 'sequencing', non-standard tuning, and continuous control of non-musical systems such as mixers, lighting controllers or tape machines. However, the fact that MIDI has proved so adaptable is a tribute to its original design structure, however short-sighted that might have been.

MIDI's data value resolution has been criticised as inadequate to convey nuances of sonic parameter variation. However, the data resolution of the MIDI 'controller' and 'pitch bend' messages is perfectly adequate using their 14 bit mode (see 2.5.1 above), although few MIDI-based devices or performance controllers implement this mode. However, problems arise if more than a handful of such message streams are in use on a single MIDI connection, as the bandwidth of MIDI is soon exceeded. This difficulty is insurmountable with MIDI as it stands.

Various aspects of MIDI which pertain to control of devices in the context of complex score performance are now examined.

3.3.1. Inflexibility of devices

Devices all operate their synthesis processes in real-time. While this is an advantage for most users, it does result in a limitation to the complexity and variety of synthesis algorithms which can be implemented on MIDI-controlled devices.

In addition, the communication bandwidth is too low to facilitate real-time control of complex multi-parametric processes (Moore 1987), and devices have no capability for control via time-stamped communication from the composition subsystem.

There is no mechanism to allow a device's performance to "degrade gracefully" (Pope 1992b) into non real-time operation to allow more complex synthesis operations to be undertaken.

3.3.2. Standardisation of control messages

As MIDI-based 'event specification' software applications can be used in conjunction with a large variety of hardware devices (to which they can communicate via the MIDI protocol) they can appeal to a far larger range of users than if dedicated to a single device. Thus, economies of scale have allowed commercial companies to develop quite complex and robust, yet relatively inexpensive software systems. Such systems can be guaranteed to be able to exert at least minimal control over sounds generated by MIDI-controlled devices.

However, there are many limitations in this guarantee of control, with little standardisation of the response of devices to messages. In fact, there is no compulsion in the MIDI

standard for devices to follow any or all of it. Thus non-standard message interpretation, or omission by devices is widespread: each MIDI device may implement a different subset of messages, or implement strange combinations, such as sending an 'all notes off' MIDI controller message (controller #123) whenever all keys on a keyboard performance controller are released.

This interchangeability of data has enabled specialist software - which supports just one or two types of musical activity well - to be developed. Speciality activities include such things as algorithmic composition, interactive performance, transcription and printing of scores in CPN , editing and storage of instrument algorithms for synthesiser devices.

This independence of devices and control software has enabled the development of flexible systems (within the limitations of the MIDI standard outlined in this section) which can employ a customised mix of specialised MIDI devices and MIDI software. Such complex systems can of course then create new problems if the user is trying to control every operation, and specify every structure from a single control and storage centre.

Most MIDI-based event specification sub-systems assume MIDI as the only possible communication and data storage format, although some have additional output formats. However, all these systems have a *structural* weakness: they are not expandable to be able to interact with new or updated protocols or device types.

Thus, in many cases, a user of such a system has to depend on a software upgrade being provided in order to be able to make use of modifications to the MIDI standard.

3.3.3. Correlation of score event parameters with sonic parameters

Events defined and displayed in MIDI-based 'event specification' software are completely tied to the MIDI protocol - a set of standardised messages. However, MIDI message parameters are *not* bound to sonic parameters, thus the precise sonic effect of sending a message to a device is *uncertain.*

MIDI messages pertain to notional *performance* actions. For example: the MIDI 'controller' message with a 'controller number' value of 1 is designated as being sent by moving a particular performance controller (usually termed a 'modulation' wheel or joystick) on a synthesiser keyboard. Other messages are assumed to result from, for example, increasing the keyboard *pressure* (pressing harder on a key after it has been depressed), or changing the *strength* (MIDI 'velocity') with which a key is struck.

However, the sonic *effect* of a MIDI message is *not* standardised. For example, the 'pitch' field of a MIDI 'note on' message actually has a standard meaning within the MIDI standard as a 'note number' which conveys semitone pitches - eg a note number of 60 correlates to middle C (C3). However, many devices have 'tuning tables' which can assign each MIDI note number to an arbitrary frequency. Thus the sonic effect of sending a 'pitch' field value of 60 in the 'note on' message is not guaranteed.

Only the MIDI note message parameters are stored and displayed in the event specification software. Thus, for example, the score display could show a pitch value of 60 or C3, but

the software has no knowledge of the device's actual *response*. Even using the messages defined by the new MIDI tuning extensions (Scholz 1991) cannot guarantee that a device will implement the extension, or to what accuracy or resolution[1].

The *effect* of changing the input parameter values of a patch ('instrument') depends on the structure and other parameters of the patch, and even on other settings on the receiving device.

> "Rarely is it possible to know precisely what the patch is actually doing, or what effect the controllers will have on the sound, if any" (Smith 1991).

For example, the MIDI 'note-on' message has three fields: 'channel', 'velocity' and 'pitch'. The MIDI 'velocity' parameter is thus conveyed as a field within the 'note on' message, ie only a *single* value can be specified at the start of an event ('note'). Data from this 'velocity' field is typically routed to various inputs of the notional internal modules from which the synthesis structure is built, but the type or degree of effect may depend on the data sent to *other* inputs.

For example, this 'velocity' parameter can alter *some*, *none* or all of the following in different synthesisers (each by a *different* amount and polarity):

- frequency and/or amplitude (of one or more sample readers/oscillators).
- centre frequency of a filter.
- The basic amplitude, pitch offset, filter frequency and/or resonance.
- The degree to which an envelope affects any or all of the above.
- The duration of any of the stages of an envelope.

A particular instrument may use the 'velocity' value to control the amount by which the frequency of a resonant filter is changed by an envelope, and the attack time of this envelope. The limits of the frequency change and the range of attack times will map to particular 'velocity' values, but this mapping is *not* conveyed anywhere in the displayed score - the only parameters displayed in the score will be 'velocity' values between 1 and 127.

This lack of correlation between displayed data and sonic effect has several important ramifications for the creation and realisation of electroacoustic scores:

---

1 These messages allow the pitch of each note to be directly specified with a maximum resolution greater then 1/100 cent, but the actual pitch resolution of the receiving device is *not* specified in the standard.

3.3.3.1. Specifying sonic parameters to instruments

A particular MIDI message value will only have a specific sonic effect on an instrument which has been overtly designed to respond to it.

In the example above, a composer who wants to be able to specify particular sonic parameters has to build or find an instrument which has been set up to respond to the MIDI 'velocity' value in the desired way.

There are many companies at present who design and supply instrument specifications ('patches') for commonly available MIDI-controlled synthesisers. However, these patches are *not* characterised or catalogued according to the sonic effect of sending them a particular MIDI message. In order for a composer to be able to control a specific sonic parameter, s/he will have to investigate the structure and settings of a supplied instrument, or construct it him/herself.

3.3.3.2. Display of score parameters

Score event parameters - as displayed in the event specification subsystem - are likely to have little or no sonic meaning to a reader. It is impossible to 'hear' an electroacoustic score (ie one in which more than very basic parameters - eg 'pitch' and 'volume' - are specified) just by reading it, unless the reader knows the device and the instrument in use intimately.

Compare this with the ability to read and 'hear' a score using conventional instruments. This task is made easier by the familiarity of the reader with the instruments in use, and the sonic effects that score parameters (eg 'pp') or performance instructions (eg 'stretto', 'arco', 'snap pizz') will have (Cole 1974).

However, if electroacoustic instruments could have score parameters which more directly reflected their sonic effect, this may allow the development of conventions so that an electroacoustic score could also be 'read' by an experienced musician. This could assist in the development of a shared understanding and means of communication of electroacoustic ideas and forms, just as musicians can communicate certain musical ideas through shared understanding of CPN scores.

3.3.4. Event parameters

MIDI's most significant deficiency, when used in the production of any kind of timbrally complex musical events, is the difficulty of communicating parameters which pertain to a single event.

As MIDI was defined from the premise of communicating information derived from keyboard performance, there is limited scope provided (and hence also implemented in MIDI-based synthesisers) for parameters which are *different* for each note (event).

> "The MIDI specification simplifies the performance-instrument interface down to that of a piano-roll plus some continuous controllers" (Smith 1991).

This is broadly true, although allowing for the use of 'system exclusive' and 'polyphonic aftertouch' messages does extend this interface somewhat beyond this. There are in fact

only three parameters which can be specified for a *single* MIDI 'note' event (addressed via its pitch number):

- the aforementioned 'velocity' of the 'note on' message (an onset parameter).
- the 'velocity' of a 'note off' message (which pertains to the characteristics of the cessation of an event).
- the 'polyphonic aftertouch' message value (see below).

Otherwise parameters can be specified only for *all* events playing on an instrument installed in a particular location ('slot') in the device. These events can be referred to using the MIDI 'channel' number currently assigned to the slot the instrument is in, or by a system exclusive message using some kind of address within the device.

Most MIDI-oriented event specification software (colloquially termed 'sequencers' within the MIDI world, for historical reasons) assume a track structure (Yavelow 1985; Yavelow 1986b) so that each track contains events which use a single instrument. Thus, the splitting of a series of events amongst different channels so that each event could have *different* variable parameter values is not supported easily. A user would have to select a different track in which to place each overlapping note. There would then be *several* tracks containing events on the same instrument (but on different channels), but it would be difficult to display and to manipulate such events. For example, to move an event would possibly require also moving it to a different track, so as to maintain the non-overlapping of events on the same MIDI channel.

3.3.4.1 Specification and display of *time-varying* event parameters

In addition to the above failing, it is difficult to communicate *time-varying* parameters for a single event.

MIDI continuous controllers can be varied during events, *if* the device responds to these messages. However, as discussed in the previous section, these are tied to a particular MIDI channel (of which there are 16 on each physical MIDI link).

On some MIDI sequencer control software (eg Dr. T 'Beyond', Emagic 'Notator'), each *track* can typically display several continuous parameters for that channel, *but* these are assumed to pertain to that channel as a continuous data stream, affecting all events which are occurring. There is no concept of tying these varying parameters to each event separately, because in conventional MIDI system usage there will be overlapping events on the same channel.

This means that the continuous data displays in MIDI-based event specification systems - which simply mirror and display the message format of MIDI - are limited to parameters which pertain to *all* events on a 'track'. For example, the Emagic 'Notator', composition software has an edit mode (called 'Hyper Edit') which can display selected multiple traces, but each trace still pertains to *all* events on a particular track or channel.

The only MIDI message which can control a single parameter for an individual event and vary during that event is 'polyphonic aftertouch'. Significantly, it is implemented in only a

tiny minority of MIDI performance controllers and synthesisers, and the author knows of no MIDI sequencer software which facilitates its display for each event. The UPIC (see 3.2.1.4) and 'Freehand' (see 3.2.1.12) systems are the only non-MIDI systems which provide this kind of individual event display facility.

3.3.4.2 Division of events from instruments

There is a clear dividing line in the conceptual structure of existing software between score parameters and instrument parameters.

Score parameters are seen as performance parameters of the instrument, ie alterable (possibly continuously) control input values to the synthesis processes running in the hardware machine. These parameters are those which (in the philosophy of MIDI based communication) would be generated by a performer.

On the other hand, instrument parameters are seen as static set-up parameters of an instrument (synthesis structure), and are then fixed as part of that structure. Some of these indeed pertain to the organisation and instantiation of the synthesis structure, and thus *are* appropriately viewed as instrument set-up parameters. However the majority of such 'instrument' parameters are simple control values which are sent to the inputs of various synthesis processes of which the instrument can be considered to consist. This division is therefore in the wrong place - these instrument parameters (often sent using MIDI 'system exclusive' messages - see 2.5.2.2) are actually available to be specified and varied at any time in the score.

Many of these notional instrument parameters are conceptually no different from the conventional score parameters - both allow timbral variation specified remotely from the software. The only difference is that the former typically use MIDI system exclusive or, increasingly, MIDI 'unregistered parameter controller' messages, whereas the latter use only MIDI channel messages such as 'controller', 'pitchbend', and 'aftertouch' (see 2.5.2.1).

Thus, instrument parameters may be alterable during a piece, but most if not all GUI-based MIDI event specification subsystems do not facilitate the display or editing of them as *score* parameters.

3.3.4.3 Multiple event parameters

Most MIDI event specification software limits the number of event parameters which can be viewed *simultaneously* . This is because it assumes a primarily real-time input mode (building up structures by overdubbing (layering) successive performance inputs (Yavelow 1985). Hence, it assumes that there will be relatively *few* variable parameters for each event, as a human performer using conventional performance equipment would be unlikely to generate such multiple parametric data.

3.3.5. Unequivocal score performance

The details of a device's synthesis resources are not known to the controlling event specification system. Hence when the event specification system sends a request to the device - eg to start some process - the device may be unable to comply. More importantly, the event specification software does not know this, and the user cannot take action. A device's response to requests it is unable to fulfil (ie how it copes with demand 'overload') is not certain, nor consistent between different devices.

As described in 3.1.1, synthesis units are addressed via messages which use the id, address or MIDI 'channel' of the device *slot* they are in rather than by an individual id for the unit.

Hence, any control software that is to request actions from such a device can only send messages addressed to a slot. It would thus need to keep track of *which* unit is currently installed in each slot, in order to have any degree of certainty that the request will be carried out. This is not undertaken by existing MIDI-based event specification software systems which send messages 'blind', with no concept of what devices, if any, are receiving them, and how they will respond. Some recent systems do keep track of unit *names* on devices, but certainly have no idea of what a device's response to particular MIDI messages will be.

3.3.6. Protocol flexibility

MIDI-based event specification systems can often *only* use MIDI controlled devices, and their internal data structures are presented as, and consist of MIDI messages. Extensions to the MIDI standard, or additional communication methods or protocols are impossible for the system to cope with, whatever the sophistication of the higher-level functionality of the system.

3.3.7. Device specific scores

A score must either be very 'sparse' (ie containing little specification of sonic detail), or be highly device specific (using MIDI system exclusive messages, or particular MIDI controller messages etc).

It is thus very difficult to realise (play) an electro-acoustic score - where there is significant specification of timbral variation - on an instrument or synthesis device different to that on which it was originally created.

The only way that the *same* sonic effect can be produced from a score which specifies MIDI parameter values is to construct one specially with this in mind. As stated above, commonly available instruments are not categorised in this way and a large amount of work is necessary to build other instruments on other devices which respond in the same way to the same scored MIDI parameter values.

The instrument characteristics have to be replicated, and *different* messages may need to be sent to effect the same timbral change.

3.4. A critique of 'custom' systems

3.4.1. Instrument design and control

In general, custom systems' raisons d'être and emphasis is on the design (both graphical and textual) of algorithm networks for signal processing or synthesis.

There is a prevalence of object-oriented description and presentation for visualising and creating synthesis structures, but such systems do not really employ the object-oriented implementation and programming metaphor (see chapter 7 for a detailed exposition of meanings of 'object oriented').

A user may be able to built a complex structure out of nested modules (for example as in MAX), but these modules are still lacking in ease of reusability - a key object-oriented concept. The presentation of the module interface can be lacking in information. There are no limits specified on the type and value range of data which may be sent to a module input.

Thus, it is difficult to know which of its inputs to connect to, and what to data to send to achieve the desired result. Such information could be documented by the builder of a structure, but this would have to be performed 'manually'. The data pertaining to an input of a lower-level module nested within the structure does not automatically present itself as the default data for higher-level structure inputs connected to it[1].

3.4.2. Score displays

Present systems seem to have an emphasis on instrument design, rather than on score displays, and there is little provision of *graphic* scoring facilities.

A score will often just consist of an event list, or is created as a result of running some set of algorithmic processes.

Those systems based around an existing computer language often do not support the creation by a composer of a score which is independent of the instrument algorithm specification. A user is effectively writing program code, in which specialised music and synthesis functions provided by the synthesis system can be used. Basing an event specification system on a language make high-level structuring or complex algorithmic event generation quite feasible, but it is not easy for a composer to grasp the event structure *resulting* from such operations. The 'score' is encoded and presented as a 'procedure', rather than as data. A 'score' thus often exists only in the sense of a functional algorithm specification, usually expressed as code text in a language. There are some exceptions such as the Esquisse / Patchwork system (Duthen 1990) which also provides a graphic score display, as well as the ability to specify algorithm networks which can create events.

---

1 For an example of this within the E-Scape software, see 8.6.4.2.

3.4.3. Devices

Many synthesis devices use DSP or VLSI hardware with local micro-coded synthesis algorithms, usually in conjunction with higher-level code downloaded from a connected computer. Increasingly however, synthesis algorithms are being coded in a high-level language (eg 'C') which runs on a fast general purpose CPU. This results in increased portability and upgradability of the system, although the system is still often too slow to perform *real-time* synthesis of significant complexity.

3.4.4. Event structuring and description

Musical events are susceptible to being organised in a large variety of ways(Yavelow 1986a). Many researchers have proposed or developed hierarchical music structuring and description formats or systems (Dannenberg 1986a; Greenberg 1986), which provide a variety of sophisticated ways of conceptualising and designing musical structures. The compositional aims in designing and using such structures are wide-ranging:

- facilitating the creation of musical data via compositional algorithms (Punch 1991);
- allowing the interchange of data between systems by providing a common format to share high-level musical data.(Balaban 1988);
- uniting independent music representation schemes in the same score (Diener 1988; Diener 1989);
- providing different interpretations ('versions') of a set of events (Dannenberg 1986a);
- displaying and editing musical processing functions and multi-parametered events (Oppenheim 1986).
- building hierarchies of pitch classes as a basis for AI-based editing (Bôcker 1988);
- triggering complex event and timbre objects (Free 1986).

A major strength of many 'custom' systems is the flexibility they allow in structuring and manipulating event structures. There are however a multitude of *different* ways of structuring; each system has its beneficial features. A general observation, however, is the prevalence and usefulness of the *hierarchical* structuring of events, often in an object-oriented way, with the same requests being able to be made to a variety of music structures. Many systems - eg MIDAS / MII (Anderson 1992a) - provide facilities to build, manipulate and process hierarchical musical structures in an object-oriented manner, providing evidence of the power and appositeness of this concept for musical purposes. Overtly object-oriented systems include MIDAS (see 3.2.1.9), HMSL (see 3.2.1.8), MODE (see 3.2.1.16), and Dmix (see 3.2.1.5). An exposition of the meaning of 'object-oriented' and its advantages in musical systems will be presented in chapter 7.

3.4.5. Communication facilities

Composition subsystems are either integrated with the synthesis engine, accessing it via function calls, or control external devices via proprietary protocols.

For integrated systems, new functions to cope with new facilities in the engine may be definable within the system, but are usually limited to the hardware or synthesis engine 'supplied', and are thus reliant upon a system upgrade to access any different low-level algorithms.

Some systems incorporate the use of MIDI for control of external devices or to facilitate performance control in real-time. When MIDI is incorporated, it is also built in as a fixed standard - inaccessible to the user - and any extensions would require a system upgrade. The *user* of a system is unable to extend the communication facilities to new devices.

Those systems which do implement separation of high-level control software and low-level synthesis devices achieve communication via custom low-level protocols. Users also cannot specify these protocols themselves, and are hence tied to performance on those devices only.

Thus, in general, a user cannot update to use new synthesis device unless the system/software designers provide a new version. Existing devices may not then be usable with the new software version.

A similar problem can arise if an old device becomes obsolete (hence unsupported), or the software supplier or creator ceases dealing with it for a variety of reasons. The event specification software (into which a considerable time and effort may have been invested building expertise and experience) may become unusable, or else necessitate a large degree of effort by an institution to modify or update it.

> "IRCAM cannot afford to continue porting its constantly growing number of compositions and research projects to new platforms. That is why, among our stated goals for the future, the development of a technology-independent system is important" (Lippe 1993).

All this has forced most systems to be institutionalised; a team of programmer-users is then typically available to alter the system for composer-users who are also based in the same place. This has resulted in few systems being usable (even if affordable) outside such institutions, a notable exception being the CDP system (Orton 1989).

3.4.6 Dependency of scores on devices

Composition systems are built around specific synthesis devices and methods, and have specific commands to build instruments (sets of unit processes), and schedule and specify score data for them.

For example, a score (set of events) designed to be realised on the CHANT software synthesiser (Barrière 1991) will contain instructions *specific* to this engine. Thus, to specify the central frequency of a single formant to glide from 200 to 500 Hz within an event of duration three seconds requires the function (in the Common Lisp score code):

```
(set-fofbank-par fofbank 0 4 ((0.0 200.0) (3.0 500.0)))
```

Even if a 'replica' instrument is designed within another system (say CSound or Kyma/Capybara) to replicate the functioning of another instrument, the control interface to the replica is almost invariably *different*, either in terms of its mode of usage (eg Kyma sends a message to separate DSP hardware), or its control input function syntax or both.

Thus the *score* would also need completely reworking - each event needing some or all of its parameters changing, or a different function call - in order to realise it on a synthesis device different to that on which it was originally created.

3.4.7 Specification of instrument parameters

Various difficulties arise in the creation of scores which use synthesis instruments:

3.4.7.1. Instrument complexity

Musically interesting instruments typically need to employ complex algorithms. Such complexity may be overtly visible within the algorithm (eg if it uses large numbers of linked processes), or lie in the nature of the process itself (eg cellular automata (CA), or non-linear systems such as FM). A composer then needs to *control* such an instrument in order to specify expressive variation in timbre and its development. Such control will usually require the specification of *many* parameter inputs, or in some cases fewer inputs but with a more complex (less-intuitive) relationship between parameter values and intended sonic effect (eg for FM, or CA).

3.4.7.2 Specification of parameters

Score events then specify values for instrument (algorithm) input parameters. This allows a composer to gain access to the inner detail of synthesis processes, allowing precise control over sound generation.

The composer must specify varying input parameters for an instrument in order to provide the nuances and shades of timbre which are normally supplied by the musical experience of a human performer, as well as the physics and acoustics of the acoustic instrument. Music without such detail and musical variation sounds highly unmusical, whatever definition one takes of 'musicality'. This breadth of activity necessary to produce computer-based music is an important issue.

> "In traditional music, the instrument and the performer take care of sounding musical; the composer can rely on it. In computer music, the composer must be, as it were, an 'instrument designer' and 'performer' as well" (Loy 1985b).

Most systems require this large number of parameters (above) to be specified by the composer for each event in the score.

A composer may often not wish to specify varying values for some of these inputs, or may wish to use some default value, but often is forced to laboriously copy values for a multiplicity of parameters for each score event, only a few of which may be of interest.

Compare this with Common Practice Notation where only the pitch and time are specified directly by a composer, with dynamics, timbre and articulation indicated by semantically

dense symbols and icons (or exhortations in Italian!) whose meaning is interpreted by human performers using "hundreds of years of performance practice" using "often unarticulated knowledge of musical style" (Loy 1985b).

3.4.7.3 Musical meaning of input parameters

Most parameters (inputs to an instrument's process units) will often have little connection with any obvious compositional meaning. For example the ratio between the outputs of two table readers deep within an instrument structure may be specifiable, but the effect this has on the resulting sound may not be obvious. This connection between input value and sonic effect may be unfathomable to a user of the structure, without either a deep knowledge of the processes and structure of an instrument, or extensive testing and experience of varying just this parameter (which could easily be one of thirty or more) so as to build up a correlation with the resulting sound characteristics in the mind of the composer.

3.4.7.4 Abstract representations for events and parameters

Most systems use very *abstract* representations for events, ie they do not consider the instrument or process which might be used to realise the events, when scoring event parameters. It is thus difficult to compose with these parameters in context. The composer can have difficulty in knowing what the sonic or musical effect will be, or may specify inappropriate values, having little idea what 'good' values are.

3.5. Conclusions

Emmerson describes the kind of working practices which composers of more complex electroacoustic music need to undertake (Emmerson 1989). This requires that composers can *hear the aural result* of compositional decisions they have made, so as to follow a more heuristic kind of compositional process, where a compositional decision/action can subsequently be tested by listening to its result. This is likely to be fed back into a subsequent modification of the action. For pitch-based music using CPN or similar, composers learn "the relationship of a paper symbols to the resulting sound" (Emmerson 1989).

Thus systems to facilitate the creation of complex music using synthesised or processed sounds should include a performance (score realisation) component or subsystem, with the option of real-time response if possible. The disadvantages of having quick aural feedback (described in 2.4.2.3) are outweighed by the need to hear the acoustic results - composers can no longer create music with abstract parameters which they can 'hear' in their head without instruments. There are simply too many sonic variables to hold them all in mind.

There are already a large number of high-level composition software subsystems (some of which have been described in 3.1 and 3.3 above), which enable musical data to be created and organised in a large variety of ways. They are linked to a performance subsystem, which consists either of one or more commercial synthesis devices via a standard communication protocol (typically MIDI) or of a particular custom synthesis subsystem.

Oppenheim (Oppenheim 1991) lists several opposing characteristics of such subsystems:-

off-line	<->	real-time
low-level	<->	high-level
compositional algorithms	<->	performance/improvisation
note-lists	<->	graphics
programmable	<->	user-friendly

Different features may be needed by different users, or the same user at different times, yet many current systems only allow *one* of each pair.

Each system has its own mixture of strengths and weaknesses. For example, commercial software systems which control devices solely via MIDI (disregarding the addition in some of digital audio recording, which does not constitute sonic *control*) are tied to its restricted view of musical parameters, and the restrictions of MIDI devices. Yet they also have many interactive GUI-based manipulation features which promote ease of use.

Thus, no single system allows *all* types of interaction for all types of device, and we should start to think in terms of a world of multifarious systems which can be used *with* each other as necessary.

Many systems are highly complex and highly capable, yet have many of the problems outlined above - both philosophical and practical - which significantly affect the way a user can compose with synthesised sound. This has resulted in the formulation of the new recommended standards which are summarised in chapter 4, and detailed in chapters 5 and 6.

3.5.1. Instrument construction

Within custom synthesis systems, the 'instruments' (synthesis algorithms) which can be specified are often highly complex. Several systems (eg MAX on the ISPW, or Patcher controlling Chant) allow an instrument builder to build networks hierarchically from modules, but these modules are only useful to a user who has intimate knowledge of what is inside it - usually the designer of the module. This is because their inputs are inadequately specified to a user in terms of their function, and the range and type of data which can be sent. Their resulting data output is also not described to an external user. This therefore does *not* conform fully to the object-oriented metaphor, where an object's internal state is hidden, its functionality can be accessed externally by a defined set of understood messages, and it is thus fully usable without any knowledge of its internal workings.

Hence such systems require a composer to have an intimate knowledge of the workings and structure of instruments. This knowledge can only be gained by long acquaintance with the instrument, either by studying it or having built it. Composers are thus forced to become synthesis experts and instrument designers in order to be able to compose with such instruments.

3.5.2. Device specific scores

As discussed in 3.3.7 and 3.4.6, scores written using both MIDI-based and 'custom' systems not only convey little information about the resulting sonic events, but depend heavily on the particular synthesis device used.

Some systems do aim to provide device independence within a certain limitations, by allocating resources within devices in a flexible fashion. For example, an experimental system built within the FORMULA language (see 3.1.2.2) allocates notes to MIDI channels and synthesisers. Device 'configuration routines' contain the necessary information about polyphony, 'part' and 'patch' set-ups on different channels for each synthesiser. Various 'note priority' algorithms can then take the musical importance of each note into account and the 'cost' of switching off a sounding note to allow the playing of a new one; the user need not specify MIDI channels or other parameters on the hardware once the configuration in use is set up (Lohner 1986). However, this system assumes that instruments consist of a single synthesiser 'patch' and only provides the minimum 'pitch' and 'volume' parameters. If a composer wants to specify more complex timbral parameters, there are many difficulties, and no existing system allows this in a device independent way. This is one of the goals approached by the E-Scape software design (see chapter 6).

The difficulty is not so much the replication in a new device of the algorithms used in an existing instrument, but rather the inability to interface these newly implemented algorithms to an *existing* score specification which will contain specific commands, messages or language elements which pertain to the device or system the score was *originally* created for. Composers talk of the need to 'renovate' or 'reconstruct' old electroacoustic scores in order to perform or modify them after the original system or device has disappeared (Koblyakov 1992).

This difficulty has important ramifications in the lack of any kind of notational *standards* for electroacoustic music. The difficulty of performing a piece on any other hardware makes it difficult to foresee the growth of any kind of common repertoire of 'classic' pieces for interpreting in performance[1] and hence any integration of electroacoustic music into the mainstream or instrumental music.

3.5.3. Scoring parameters

Some systems, such as UPIC, concentrate on the score structuring interface, and provide a finite set of instrument configurations. Thus the input parameters to those instruments are known, and a composer can specify these in a score with a meaningful presentation: for example, specifying an amplitude or pitch envelope shape.

---

1 Performance here can have its normal meaning of human players reading a score and controlling performance instruments, as well as the realisation of the score by a machine (eg a computer) directly reading the score, and controlling synthesisers by sending them the specified messages.

Those systems which provide MIDI control assign MIDI messages *directly* to events, with the resulting problems discussed in 3.3.3.

Many systems allow the composer to build a variety of different instrument structures for a score event. The input parameters to an instrument are then presented in an *abstract* manner (eg as input numbers) for specification in a score. A composer then needs intimate knowledge of instrument internals to know the *meaning* of each of the presented inputs.

Some systems, such as MODE, Dmix and HMSL (see 3.1 & 3.2) *deliberately* set out to keep score data parameters abstract - with no device or instrument parameter context. When a score event is then played (realised on a device using an instrument structure) each event parameter value is assigned or processed into a device specific form. This results in some problems:

• Each data item assigned to a score event parameter is treated in isolation from other parameters (thus precluding the use of parameters which interact, for example the value of a 'filter frequency' event parameter may also need to take account of the 'pitch' parameter in being translated to a device-level value.

• A parameter value may be nonsensical, either because the device driver used cannot understand that parameter, or because the value itself is outside the device's range. The driver can report these problems to the user, but only when the event is to be played. Thus values can be specified and displayed which are inappropriate, and this will only be 'caught' by attempting to play the event. The device driver is also going to have difficulty knowing what to suggest, beyond telling the composer that 'I don't understand this parameter' - the composer has to decide which device input to assign each score event parameter to.

This has ramifications for the device independence of a score, as discussed in the previous section. If a different instrument or device is assigned to an event, then the composer must laboriously reassign and/or re-scale the values of each *score event* to fit the new instrument's available parameters and ranges (Koblyakov 1992).

Rolnick laments the difficulties of playing an electronic score on a different synthesis device other than the one it was composed for. Although commenting more about the live performance of electronic music, the problems he describes are equally pertinent to music 'performed' by data from a composition system.

> "We can see that, if a piece of music is to be performed at different times... then the problem of moving the patch information between different instruments is going to be unavoidable - particularly if the piece aspires to a longer lifetime than the duration of a specific synthesiser's popularity" (Rolnick 1986).

3.5.4. Communication between composition software and devices

The world of custom composition (event specification) and synthesis systems is highly fragmented. There exist a large number of composition ('event specification and

manipulation') sub-systems which are each tied to a particular custom synthesis sub-system.

In some *recent* systems there has been a welcome trend towards separating the synthesis engine from the high-level composition specification subsystem. This may either be a more notional separation within a single integrated environment such as in the CHANT, CSound, or Formula systems, or a physical separation, such as in the Kyma / Capybara, or UPIC systems (see section 3.2).

Many existing systems provide the facility to specify musical and synthesis structures as time-stamped instructions, and indeed for many this is the only such mode. If such score or instrument data is produced by a different compositional subsystem - perhaps because of its different structuring facilities or conceptual framework - it is often laborious to then transfer this data to the target system for further processing or performance.

For example, several systems such as Patchwork allow a user to specify CSound instrument structures via a MAX-like iconic interface (see 3.2.1.1). Other systems such as S11Input convert performance data in MIDI form into CSound score data. In both cases, the resulting data has to be output as a CSound 'orchestra' and 'score' text file respectively, which must then be loaded into CSound and run. A user thus has to manage this set of systems, preferably on the same system or network, and cannot simply treat CSound as a device, which can be requested remotely to 'play this'.

User who want to use a graphic instrument design and score system cannot then easily perform their data *from that system* using CSound. As it happens, there are no existing systems which can provide integrated control and display of both instruments and scores of the complexity of which CSound is capable, but the reason for this may be that control of CSound as a slave synthesis engine is not possible without extensive human intervention.

Another example might further clarify this issue. A simple FM table-reading instrument could be designed in CSound which replicates the functioning of the DSP engine in the UPIC system, or which possesses additional subtlety and complexity to go beyond the scope of the UPIC hardware. A user might then wish to draw complex scores on the UPIC GUI screen, and have them played on CSound. Of course at present, UPIC cannot create CSound score files from its graphically derived music data, but this feature could be feasibly added.

An event specification subsystem could be designed to attempt to cope with writing different time-stamped formats for the different existing systems - acting as synthesis devices - which can or must read text files from disk (eg the MIDAS system or CSound). A user interface could be envisaged which attempts to enable a user to create new formats and conversion methods to do this *without* recourse to a programming language.

However, such a user interface is likely to have to be as flexible as a programming language to be able to cope with the different format possibilities. Even then the inherent problems of data transfer and remote control of the target system remain. Thus, this aspect has not been implemented in the E-Scape prototype design (see chapter 6), and instead, a

new system communication and operation design has been proposed (as presented in chapter 4). This proposal will not only make such control of remote 'devices' (ie systems acting as devices) feasible, but also make realisable the goal of facilitating user creation of protocols for time-stamped communication without recourse to a programming language.

II - Proposals and design goals

This section, (chapters 4-6) describes the proposals made to address the difficulties elucidated in section I.

Chapter 4 presents a proposed set of standards for computer music system organisation, behaviour, and inter-communication.

Chapter 5 then presents details of a proposed communication standard which will facilitate the operation of the system organisation described in chapter 4.

Finally in this section, chapter 6 describes the detailed design goals of the 'E-Scape' composition software system which will demonstrate the relevant aspects of the proposed system functionality.

4. Recommendations and proposals

This chapter presents a set of recommendations and standards which will address the difficulties and weaknesses current computer-based composition systems discussed in the previous chapter.

The attempt to solve these problems has resulted in the formulation of recommendations and standards for:

- The *organisation* and *structure* of computer-based composition systems as a network of 'event specification' and synthesis device sub-systems.
- The structure and behaviour of synthesis device sub-systems.
- Inter-communication standards between sub-systems.
- The facilities and functionality of 'event specification' sub-systems.

This chapter summarises these recommendations, which will then be backed up with a prototype implementation. This includes a prototype 'event specification' software system which incorporates the recommended features, and a detailed protocol design which follows the recommendations. The extended design features of synthesis devices - both MIDI-based and custom - which will allow them to integrate into the recommended system organisation are also suggested. Testing of the software within the limitations of the existing devices available has also been carried out.

4.1. System structure standards

For the reasons discussed in section 3.5, systems which will be used for composition using sound synthesis typically include a performance or 'score realisation' component (ie one or more synthesis devices), as well as an 'event specification' software component with which the user interacts. The structure and organisation of such systems - ie how their components are organised and connected - is an issue of prime importance.

In some systems, the synthesis devices (implemented on DSP, custom VLSI or general purpose CPU) are physically separate from the event specification (compositional control) subsystem. This can then communicate with the devices via an interface and message protocol. The hardware interface is more usually a standard type, such as SCSI, MIDI, VME, or IBM 'multibus'. On the other hand - except for MIDI-based systems - the message protocol is usually custom-designed, with a specification of the types and formats of commands and data to be transmitted.

In other systems, a synthesis device can be implemented as a set of routines within a *single* system. However, these may still be *structured* as a separate entity, with defined communication links - through a standardised software protocol - to the 'event specification' subsystem. Several more recent systems take this concept further, with the aim of *decoupling* the high-level specification language or system (the 'event specification' subsystem) as much as possible from the processes which run the synthesis algorithms.

Such system typically have three components: a host computer with *some* kind of high-level control and specification software, a processor-based synthesis engine (eg DSP, RISC, CISC) which creates sound output (usually with real-time output capability), and an *interposing* operating system layer ('toolbox' etc) which provides access to the processor's functionality to higher-level systems via defined functions.

New synthesis devices will then only need to respond to these function calls in order to be usable with all high-level software which uses the interface layer. Examples of this approach are the NeXT 'MusicKit' functions (see 3.2.1.2), the Frox Digital Audio System (Loy 1992), the MIDAS system (see 3.2.1.9), the MARS Project (Andrenacci 1992; Palmieri 1992), Unison (Bate 1992), and the IRCAM 'Unified DSP Interface' Environment (Depalle 1990). All these systems have graphic editing facilities for synthesis/processing algorithm creation, but little or no graphic *scoring* facilities.

4.1.1. Systems as components

The decoupling and separation of the components within systems has, in the author's view, correctly been seen as the way to facilitate future portability of systems to new synthesis or computer hardware.

The restrictions on system communication (outlined in 3.5.4) also affect the usability of existing systems with different kinds of synthesis devices. This in turn restricts the complexity, expandability and adaptability of the system to accommodate developments in synthesis hardware. As described in chapter 1, the history of computer-generated music, and computer-controlled music systems is littered with examples of man-years of high-level software development having to be abandoned because the software subsystem is tied to using or controlling synthesis hardware which has become obsolete or unmaintainable.

The composer Gary Kendall comments:

> "... a kind of problem...which is that by the time someone writes all this software, especially by the time someone debugs the software, it's probably going to be obsolete, and new developments and new ways of thinking are going to come up" (Rodet 1991).

In addition, much human effort and expertise is spent in having learning to use new systems, as old ones are abandoned. Max Mathews comments:

> "I wonder if we're not overwhelmed not by the variety of machines that exist, but rather by the rate at which the world is changing, by the fact that new machines come along before we can learn to live with, and create on, the old machines" (Goebel 1991).

Risset among others has emphasised the importance of attempting to keep systems open ended - with components uncoupled from intimate dependence on each other. He stresses:

> "...the importance of keeping systems open-ended, maintaining flexibility to adapt their possibilities to musical needs when they arise. This is vastly preferable to

> fixing the design and limits of digital music systems on the basis of technical decisions" (Risset 1985).

This decoupling is thus the direction in which future systems must go, if duplication of development effort, and dissipation of composers' energies is not to be avoided, and the availability and choice of event specification (composition) software, and synthesis devices available to composers is to increase. Such a structure, either in concept or in fact, is of a *loosely coupled* system, as described in 2.3.2.

> "In a loosely coupled system, the processors are largely autonomous and require a protocol for inter-processor communication (eg Ethernet, RS232, MIDI)" (Loy 1986).

Loy is here discussing the structure *within* a composition and synthesis system, which consists of a network of communicating processors, each acting as a node within the network. The advantages of a loosely coupled system are that each processor node can have a high degree of autonomy, and the system can be more flexible and expandable as it more readily supports a heterogeneity of components.

These advantages of loose coupling remain if a loosely coupled structure is implemented at a *larger* scale, with each 'node' being a larger more complex system itself, communicating via a common, sufficiently general protocol.

Systems as they exist today could be conceived of as nodes in a *larger* system, if they implemented the required 'node-like' behaviour to enable them to participate, in an 'event specification' subsystem and/or synthesis device role, as described in 2.3.

The *disadvantages* (see 2.3.2) of such loose system coupling (such as the need for each node to possess CPU-level intelligence to support the protocol), become less critical if the components (nodes) of the system have a CPU level of processing power. This also means that if a node malfunctions, it can more easily be debugged: each node can act as a stand-alone system itself.

Limitations due to inter-component communication bandwidth and the processing time used up in communicating between nodes are also becoming less important, as faster CPUs and interfaces are introduced. If, as is proposed, the facility is provided for all communication to be out of real-time (by sending time-stamped messages), then such problems also become less important. One option would be for each node to store time-stamped control messages and execute them later on receipt of a 'run' command. This might require a significant amount of memory on each node. A more elegant and flexible solution might be for each processor to also be able to operate its synthesis processes in off-line mode. Each node could thus be running, as it executes a small *temporary* store of time-stamped control messages, which would be deleted after execution. During running, further control messages could be sent to the node. Thus, almost any size of memory buffer and any processing speed on a node can be catered for, and the system could 'degrade gracefully' (Pope 1992b): simply taking longer to compile the resulting soundfile output.

4.1.2. Proposals for system structure

It is therefore proposed that *all* computer music systems should include functionality which enables them to also act as components within a larger-scale system. These components should be able to interact via a flexible communication standard.

All existing and future systems should thus be able to participate in a larger computer music scene, where integrated and flexible systems can be built out of the most appropriate components. Each system should provide the facility be able to be accessed by other systems as if it were loosely coupled - ie as if it consisted of an 'event specification' subsystem which is communicating with one or more synthesis device subsystems via defined protocols - even if the system's *actual* structure is homogenous.

For example, existing *integrated* systems such as CHANT, UPIC, or CSound could be decoupled into 'event specification' (compositional control) and 'synthesis device' components. This will then facilitate external systems being able to access the functionality of each of these components. Systems' existing functionality as a complete and usable integrated system can still be maintained (most probably meaning slightly faster operation), much as a MIDI keyboard synthesiser can still be used as a stand-alone instrument with no use of MIDI whatever.

This decoupling and opening up of system components to external access might eventually lead to the development of some form of interface standards. This would allow discrete components to be usable in a single system, communicating via one or more hardware interface standards (eg a fast SCSI, or MADI[1] type connection), with a defined standardised protocol, along the lines of that proposed in 4.2 below. This degree of standardisation may, however, be too restrictive: thus it is proposed that composition systems should be able to convert their data structures into the recommended standard form, but use a *variety* of software and hardware protocols. Thus, what *information* is in a message is standardised, but the *format* of that information will depend on the protocol used.

A composition system may need to control several devices (or other systems which are *behaving* as devices); each via a different hardware connection, and/or message format. However, in proposing that systems should be controllable from *externally* input messages so as to act as synthesis devices, it is important to state clearly what is *not* being proposed:

- *all* features of a device need not necessarily be controllable from outside it.
- a device is not prohibited from having its *own* control mechanisms, incorporating high-level event specification software, or being operable as a stand-alone system.

---

1 Multi-channel Audio Digital Interface - a communications protocol for transmission of multiple digital audio streams between devices.

• any such 'custom' control software in a device is not necessarily required to use *standardised* protocols or links when communicating internally with its own dedicated synthesis processes.

This last concept has parallels with MIDI-based equipment, and a useful analogy can be drawn with MIDI synthesiser keyboards. These pass messages (derived from their keys, other performance controls or front panel buttons) in a custom low-level format to their synthesis hardware ('device') section, through internal communication links. Some or all of these messages are *also* implemented as MIDI format messages, and are available to external control. This is the kind of external access which is being proposed for larger-scale software-based systems.

The central proposal is thus that all systems, whether loosely or tightly coupled within themselves, should be able to be partitioned so that they can *appear* to the outside world to be loosely coupled. This will enable one or more of their *component* subsystems (synthesis devices, and/or event specification interfaces) to be available to be used in building larger-scale flexible systems.

This flexibility will encourage the maintenance of the diversity to be found within event specification (composition) software as well as synthesis devices; this is both necessary, inevitable and welcome. Composers will be able to select from an even larger range of possible system component combinations if the proposed system structure proposals are taken up, hence actually *lessening* the need for any *one* system to attempt to provide every type of interface, conceptual model and presentation.

4.2. Intra-system communication standards

The concept of interchangeability, allowing specialised systems or modules to communicate with each other in a guaranteed fashion has been a long-held wish amongst users of such systems (Graham 1980).

The music technology world has shown that it can formulate and adopt communication standards if there has been a perceived need, and if the standard is sufficiently flexible and not too costly (financially and conceptually) to implement.

From the late 1970's onwards, there were several laudable attempts by individual commercial synthesiser manufacturers to formulate communication standards for their products. This culminated in the formulation in 1982 of the MIDI standard (see 2.5), which finally achieved the aim of allowing some degree of guaranteed communication from one device or system to another.

Another prominent example is the AES/EBU digital audio communication standard (AES 1985), but less obvious items such as EPROM memory cards have also successfully had a flexible standard formulated and agreed upon by the JEIDA and PCMCIA industry associations (Russ 1992).

One of the proposals made by this thesis is that this concept of communication standards should be promoted *at a higher level* between computer music composition and synthesis

systems, which should allow external access to one or both of their composition ('event specification' and synthesis ('device') aspects or components.

What is being recommended here is that the interface between the two notional components of a system can be opened up to other external systems, by means of standardised kinds of messages which will have a defined effect on a system's 'device' component.

4.2.1 Proposals for communication standards

There needs to be a standard set of *types* of message to communicate within components (subsystems) of future systems with the structure proposed above. Each type of message should have a defined *purpose* (as exemplified in 4.3.3 below). In addition there should be a standardised *meaning* for each message, ie a set *response* from a synthesis device to a particular message type.

Note that conformance of messages to a particular message format or hardware interface is *not* necessarily being proposed. Such a standard would be too restrictive and unwieldy, and restrict the development of new devices and synthesis methods. Messages may well be conveyed on one or more of a variety of hardware interface standards, and the fields may be in various orders, and contain data of different sizes. Example implementations might be:

- using the SCSI-2 interface standard to communicate between computer and synthesis device, with 32-bit fields for each message. Ids could all be numerical.
- using a fast serial link, with an extending[1] system of bytes encoded as 7-bit ASCII characters, with fields in reverse order.
- using UNIX sockets to a communicate from an 'event specification' application to a synthesis 'device' which consists of software routines running on the same UNIX network. A 'device' could, for example, consist of the CHANT software system coupled with a 'front end' software layer which receives and decodes time-stamped messages, interpreting them to control the writing and compiling of CHANT files.

What the standards *do* specify are the types of message, which data is to be conveyed, and the structural scheme behind this data which bears on control software and device functionality. In other words, the general features and organisational model of a device *as presented to an external user system* should be standardised.

For example, devices should be able to respond to a message whose purpose is to request a particular type of synthesis unit to be instantiated. This could be likened to a MIDI program change command: recalling a structure or object from memory and installing it in current memory buffer ready for running. In addition this message should contain a 'user id' which can be used for future references to the unit, and an 'instrument id', allowing units to be grouped. Messages to connect units should be provided, if such flexible structuring is implemented on a device.

---

1 The eighth bit can signify that a further byte follows for the same field.

Other messages can send data, either to all units within a particular instrument id, or to particular units. Note that the effect if this data may be to start or stop a unit or group of units running. This facilitates a flexibility not provided at present from MIDI-controlled synthesisers, and as previously discussed, provides a standard way of interfacing custom computer music systems to any 'event specification' software system which can implement the appropriate message format.

The proposed communication standards support both real-time and time-stamped communication, at various levels of sophistication. Systems should, at the least, support messages (sent from a controlling 'event specification' subsystem) to request a synthesis device to create and connect synthesis units, to start and stop them, and send score parameter data to them.

Additional facilities should allow a controlling subsystem to define and store *instrument-templates* within a device. Each template consists of a *specification* of a network of synthesis units, with defined inputs which connect to these units. The controlling subsystem can then request a device to create *instances* of an *instrument-template* within itself - ie install active processes which are described by the instrument-template specification. The controlling subsystem can then send data to the inputs of the instrument, rather than needing to address individual units within in.

Further levels of sophistication within the proposed standard allow a controlling subsystem to download hierarchical event data to a device which, if it supports such sophistication, can store and schedule events and their parameters in advance of performance.

A set of message types which exemplify the proposed communication standards has been developed, which are presented in chapter 5. In chapter 11 these message types are then examined in the context of possible development of the present MIDI standard, and currently available computer music systems. The message types have been further evaluated by designing a detailed protocol for the MIDAS system which implements the communication standards proposals. This protocol also incorporates a design for the required functionality within MIDAS and its MII 'front end' component (see 3.2.1.9) which will enable it to implement this protocol.

4.3. Synthesis devices

There are many problems in attempting to control and use *all* device types from a single 'event specification' subsystem. Hence as a result of this research, some new standards are presented and proposed, both for the functionality and structure of synthesis devices, and their control by a communication standard.

This standardisation of control facilities for all synthesis devices has some parallels with the lower-level standardisation which has increasingly been promulgated to allow uniform machine-independent high-level access to DSP algorithms which may be implemented on a variety of hardware. The proposals in this thesis are for device functionality standards at a higher level.

4.3.1. The concept of the 'device'

As discussed in 4.2 above, many systems which are at present seen as stand-alone synthesis and composition environments or systems (with a user-interface, high-level software[1] and their own dedicated synthesis processes or hardware) can be considered to be *'devices'*, if their synthesis component can *also* be *externally* controlled according to defined standards.

Thus, a device as considered here consists of the low-level synthesis routines *plus* a 'wrapper' subsystem or software component which manages them. Often, at present, this wrapper actually forms part of a self-contained system. For example, the MIDAS system has standard functions to create and connect processes, but also a higher-level control software subsystem (MII[2]) which manages the lower-level entities and keeps track of their addresses etc. A higher-level controlling subsystem would interface to this intermediate MII layer, and it is at this level of functionality that standards are being proposed.

A synthesis 'device' should thus be able to be communicated to via standardised kinds of message (whether via a physical interface, or within a software environment). A device should support various primitive synthesis processes (which are likely to vary from device to device), a set of general operations which all devices should be able to perform, and a standardised way of *controlling* and using whatever facilities it provides. The 'device' is thus an essential component of the system proposals presented in this chapter.

4.3.2. Examples of 'devices'

Several examples of how present systems might behave as devices under the proposed standards definition will further clarify this concept of 'devices'.

4.3.2.1 MIDI-based systems

Existing MIDI-based synthesiser devices could be partially integrated into this system concept using some kind of 'MIDI-2' or 'MIDI-LAN' standard (see 2.6); any such standard must go far beyond the present concepts embodied by MIDI in terms of flexibility, speed, and scope. However, most of the device operation standards proposed this section are not supported by existing MIDI synthesiser devices.

Discrete conversion devices could be developed and marketed which translate certain "new protocol" messages into appropriate MIDI messages, to enable MIDI-based synthesisers to be incorporated seamlessly (albeit with restricted modes of operation) into a system. Another useful function of such devices could be to convert MIDI messages from MIDI performance controllers into corresponding messages (a small subset) of the 'new protocol' to enable them to be used as performance controllers within the new system scenario.

---

1 Indeed, as will be seen by the proposals for device functionality below, a medium-level software component incorporated within a device is likely to be almost essential in order to provide the proposed external control facilities.

2 MIDAS Intermediate Interface - see 3.2.1.9

More sophisticated software systems can be envisaged which provide data storage and intelligent resource allocation functionality, so as to enable the full control of a number of connected MIDI-controlled synthesisers, according to the proposed communication standards. Some of these concepts can be illustrated using the several MIDI-controlled synthesisers connected to the University of York 'MidiGrid' software (Hunt 1988; Hunt 1990; Hunt 1992). This provides an innovative GUI to enable the user to trigger events, which can consist of hierarchical MIDI 'note' structures of arbitrary depth, each note of which can use a different synthesis structure (up to the limit of the available MIDI channels). Thus a single user action can initiate and control a large number of nested events on a variety of MIDI devices. MidiGrid also provides the ability to initiate these events remotely via defined messages, in this case further MIDI 'note on' messages (Hunt 1991). Thus the MidiGrid software, along with a set of connected MIDI-controlled synthesisers can not only be considered as a system (usable in its own right), but *also* as a single (more complex) device which can respond to external messages. For example, a single 'note on' message can command the MidiGrid 'device' to perform large clusters of complex sounds. However, there is no mechanism for an *external* system to control the building up of such structures in MidiGrid.

4.3.2.2 UPIC

The 'UPIC' system (see 3.2.1.4 above) at present uses a custom synthesis device - the 'real-time unit'. This consists of 3 processing boards, which are linked to the event specification software (running on a PC) by an IBM 'Multibus I' interface (Raczinski 1988). The 'real-time unit' can store downloaded graphic scores, and process and realise them using wavetable-reading oscillators with various enveloping and FM facilities.

The UPIC control and user-interface software is innovative and interesting, but is locked into controlling this device, via proprietary messages.

If UPIC were to conform to the proposed system structure, then the UPIC score information would be translated into the proposed standardised message types (see chapter 5). These messages, for example, facilitate the communication of data tables, which naturally supports the UPIC's score trace data structures. These messages would be realisable on a variety of synthesis systems, so that the same score data could be performed on other synthesis devices. In addition, other composition control software subsystems could utilise the UPIC synthesis unit.

The UPIC composition software would then be upgradeable in the same way (albeit with the other problems described in section 3.4) that a MIDI-based software composition subsystem can effectively be upgraded in sonic capability - ie by adding to or replacing the MIDI synthesis devices which are used.

4.3.2.3 CSound

Similarly, the recent CSound implementation incorporating real-time control (Vercoe et al. 1990) allows it to be considered as a synthesis device, even though it is of course a powerful synthesis system in its own right (albeit with a relatively crude user-interface).

MIDI messages or ASCII score events (from a console or other process) now allow the CSound system to be treated as a device which can respond to unscheduled control inputs.

This system is likely to be relatively easily adapted (either within itself, or via additional software processing layers) to support the device capability standards proposed here.

4.3.2.4 Others

Other examples of 'devices' as defined in this section are:

• 'SynthEdit' is a GUI-based application which facilitates user construction of unit generator algorithms; these run on the NeXT computer's integrated DSP via its MusicKit software (see 3.2.1.2). SynthEdit also *manages* the allocation of appropriate objects within the MusicKit system to enable the performance of score files. Note that the MusicKit software is too low-level to be a viable 'device' in the above definition.

• Patchwork / Esquisse (Laurson 1989; Duthen 1990) is a MAX-like GUI-based system for creating and linking functional modules which can generate and manipulate musical material at the event or sound creation level. It can specify CHANT synthesis structures, generate score and orchestra files to control CSound synthesis, or control DSP-based synthesisers.

• In the MIDAS system, the MII 'front end' software component (see 3.2.1.9) has been conceived, and a design specification formulated, in collaboration with the author during the course of this project. One of MII's major roles is to provide 'device' functionality for MIDAS, ie the ability for MIDAS to be externally controllable from other systems, in addition to its own user-interface components.

4.3.3. Proposals for device functionality

It is proposed that all devices should implement a set of standard message types. This is *not* to imply that all devices must support the same kind of processes or facilities, but that those which are provided are accessed in a *uniform* manner. Each device will have a published specification, detailing what synthesis processes are supported. These processes should be presented in the form of primitive units which the user may not (and should not need to) break down. All synthesis algorithms should be able to be built up as larger structures from these units via defined messages. As explained earlier, the exact format (field sizes, orders, hardware connections etc) need *not* necessarily be standardised, but only the *types* of messages to be provided, and their information content. For example, there should be a message to request the instantiation of a unit process of specified type, with a supplied user id, which can be used to refer to the process subsequently within the device. Exactly *which* processes ('units') can be requested, or the codes which identify each unit type in the device can be left unspecified in a standard. This allows for the necessarily great disparity in synthesis device functionality.

This proposal is thus *not* based on some form of 'General MIDI' scheme (see 2.5.3) where the facilities provided are tightly specified - eg a MIDI 'program change' message with a value of 1 must call up a synthesis structure which results in an 'acoustic piano' kind of sound. If it were proposed that all synthesis devices should implement the same unit

processes, and have the same inputs for them, this would be rightly seen as unworkable, given the range of types of synthesis technique, and modes of operation in use.

What then is being proposed is that devices should allow their synthesis processes to be controlled and specified in terms of *primitive* processes (henceforth referred to as 'units'). A device should have a published specification of the *format* of the messages it understands. As stated above, it is unrealistic within the computer music development community to *prescribe* a particular format which messages should conform to. What are prescribed, however, are the *purposes* of the messages which should be understood by a device (whatever their format). For example, messages must be provided to perform functions such as 'instantiate', 'connect' (if appropriate), 'delete', 'start' and 'stop' a unit, 'send data' to its inputs, or 'store data' within the device if appropriate (eg for a 'table' unit).

Enhanced levels of functionality should be provided by more advanced devices. Such features include:-

• the ability to send messages to a device with a simple *time-stamp*, which indicates a time offset when that message is intended to be acted on. The device will then store and schedule the message, and execute it at the appropriate time offset when signalled to start 'running'.

• the ability to allow *networks* of units to be specified, stored and referred to as single objects.

• the ability to allow the downloading and scheduling within the device of hierarchical score data structures. Such messages allow more *complex* event structures and data to be specified and loaded to the device, which would be laborious using single time-stamped messages.

When it is considered that many current devices (as conceived of in this proposal) already contain, or are integrated, into a high- or medium-level software subsystem, then the provision of this kind of functionality can be considered to be feasible in the near future. A synthesis device could exist as a separate subsystem, incorporating software which is designed around a set of low-level (eg DSP-based) processes. This software component would provide the device with higher-level functionality to incorporate storage of algorithm definitions, unit process connections and data, as well as supporting the downloading and scheduling of events and input data from the composition specification subsystem. This will facilitate the specification of high-density audio-rate or control-rate information by the composition subsystem which would not be able to be processed or transferred by it in real-time.

This device software could also provide its own user interface to allow the creation, connection and testing of algorithms, networks etc, without needing the high-level system to be in use. This might be considered to be the state of most synthesis systems at present - a relatively crude textual user interface, facilitating user specification of algorithm and event data.

A device should allow synthesis units to be instantiated (see 3.1.1.1) with an assigned high-level 'user id'. All subsequent communication from the controlling software to the unit in the device is then done via this 'user id', rather than via some fixed 'channel' or address parameter which depends on *where* the object is held in the device. Thus the device will be perceived to have a dynamic memory map, with objects addressed via indirection. The device will map received id numbers for a unit to its memory location within the device (which may or may not *actually* be fixed).

The controlling software can then create, connect and use objects using only ids, which obviates the problems found in knowing and calculating address offsets, or assigning device 'slots' and channels' in fixed memory. A device should be able to allocate units - without a fixed limit on their type or number - until it reaches some overall limit of processing or memory capacity. The device can then either report an error to the controlling subsystem (which can then report an error to the user, or take some other actin, such as requesting the device to delete a selected object), or delete another unit itself. The standard could be extended to facilitate more sophisticated bi-directional communication between control system and device.

4.4. Event specification software standards

Many researchers have investigated the ideas behind different score data structures, and have proposed various standards which would allow uniform description of musical structures, so as to facilitate interchange of data between composition systems (Balaban 1988; Pope 1992a; Pope 1992b). This topic is a thesis in itself - the major feature is that there are innumerable ways to generate and structure musical data, and a large variety of 'event specification' sub-systems[1] in existence to facilitate this.

An ideal system should support a number of ways of viewing or creating music, in the way the Dmix 'event specification' software (see 3.1.2.5), for example, attempts to do in the area of high-level event and data manipulation. However, it cannot be claimed that any one system can facilitate *every* way a composer may wish to work with, or conceive of, musical or sonic structures. However, with the system organisation proposed in this chapter, no single 'event specification' (composition) subsystem would *need* to attempt to support every conceivable methodological structure, which is arguably an inherently unrealistic task in any case.

Thus, the recommendations presented below are for the *additional* functionality which such 'event specification' subsystems would need to provide to enable them to function and

---

1 The phrase 'sub-system' will be used (here and elsewhere) when describing event specification (composition) software, even if this in fact exists as a complete system in its own right with incorporated synthesis facilities. This word 'subsystem' is used in order to emphasise that it is solely the system's event specification (user interface) *component* which is being focused on in this context.

communicate effectively within the proposed system scenario. The prototype E-Scape 'event specification' software presented in chapter 6 implements these features.

Other features of an event specification subsystem can be argued as *desirable* to solve some of the problems illustrated in chapter 3. These features are also among the design goals of the prototype E-Scape software presented in chapter 6.

4.4.1. Proposals for event specification software

4.4.1.1 User extendibility of communication

Synthesis device subsystems which conform to the recommended standards are nevertheless very likely to employ a degree of flexibility in their communication protocols. This is both likely and desirable, as described in 4.3 above.

Thus, software composition and control systems need to be designed with equally flexible communication protocols to be able to utilise and control a variety of devices. Since a system designer cannot anticipate all possible developments in the control protocols of future devices, a system should ideally allow (knowledgeable) users themselves to *define new communication protocols.* A protocol should be definable in terms of the types of messages, their fields and their content, the choice(s) of hardware interface used, and any special formatting each interface requires. Such considerations form an important design goal of the 'E-Scape' prototype software presented in chapter 6.

4.4.1.2. Control of distributed processes on multiple synthesis devices

Within the proposed system organisation, a single event specification subsystem should be able to use any number of synthesis devices. These devices, it should be remembered, may actually be complete systems in their own right (eg CSound), but which may be used *as* devices by an external controlling subsystem. This control will be effected using types of message which conform to the proposed communication standards (presented in detail in chapter 5).

Thus, an event specification subsystem first needs to be able to control different types of device, via *different* communication protocols if necessary (as described in 4.4.1.1).

To fully take advantage of the compositional possibilities made available by using a network of multiple synthesis devices, however, the event specification subsystem should be able to communicate with a *distributed* synthesis process. It should be able to specify and control a synthesis process which has components on *several* devices (possibly of different types), and be able to present it to a composer for use as a *single* entity. This feature is one of the major design goals of the prototype E-Scape software (see 6.2).

4.4.1.3. Device resource allocation

As synthesis units or networks (corresponding to software 'instruments') are identified by an id within devices in a flexible dynamic manner, the control software should be able to allocate an instrument to each event (which could still be presented via the current 'track' or 'pattern' paradigms). The control system should then be able to allocate and request the

resources in the device required to perform the score. This is preferable to requiring a device to allocate resources itself, as it will be more difficult for a device to possess full knowledge of the context of each event in the score, and the musical importance attached to it by a composer. See 6.6.3 for more details on this topic.

5. Communication standards proposals

As stated above in section 4.2, there is a need for a standard set of message *types* to facilitate communication between 'event specification' software and synthesis devices, acting as subsystems within a computer-based composition and performance environment. It is also important to specify a standardised *response* from a synthesis device to each message type. The proposed communication standards make no restriction on the data types, the order or size of message fields, the hardware communication protocol or connection type, or on the coding used for messages. What the standards *do* specify is the logical structure of message protocols: the types of message, the fields of those messages, and the structural scheme behind them which bears on 'event specification' software and device functionality.

The standards proposals do not include handshaking messages sent back from a synthesis device to the event specification software, beyond a simple 'error' message. Many additional such messages can be conceived which report details of the status of the device. For example, a message could report that a device reference number ('id') is already allocated, with the event specification software then perhaps required to send back some kind of confirmation message in order for the device to act upon the original message. Other future *extensions* to this proposed standard might provide for more extensive handshaking, for example that *every* message sent to a device returns a confirmation, or a variety of error messages. This proposal is obviously only a first step in the formulation of any agreed communication standard, and has been restricted to one-way communication in order to simplify the initial specification. Reply messages will depend on the agreed messages which are *sent* to a device, so it makes sense to formulate these first.

The initial communication standards proposal focuses on the compositional use of sound *synthesis* (described in 1.2.1), and does not specify messages for the control of soundfile *processing* functions within a device (described in 1.2.2). An example of such a message might be "copy from 1.0 to 3.45s of a specified soundfile, then merge (mix) it with a second specified soundfile starting at 5.677s, with an amplitude ratio of 3:5". Such functions as these are required within any comprehensive composition environment, and provide scope for further work.

The facilitation of both real-time and time-stamped control of devices is required (see 2.4.3.2), with the ability to *directly* transmit time-stamped data to a device, rather than have to save data, then transfer and load it to each device (typically in different formats for different devices).

To achieve these goals, three levels of communication facility have been designed, each of the higher levels including the functionality of the lower ones:

• **Level I** messages allow composition 'event specification' software to directly create and connect synthesis units in a device, to start and stop them, and to send score parameter data to them. Messages can be sent to be executed immediately, or with a time stamp.

Existing MIDI devices implement most of the *non* time-stamped aspects of the proposed level I communication standard

• **Level II** messages allow 'event specification' software to define *instrument-templates* within the device consisting of networks of connected units. An instrument-template has defined *instrument-inputs* which connect to one or more of its component units.

The 'event specification' software can then request the creation of *instances* of an *instrument-template*, and send score data to their *instrument-inputs*, rather than to individual units.

• **Level III** messages allow 'event specification' software to download and schedule hierarchical score data to a device in advance of performance. This is of crucial importance to allow complex scores to be performed in real-time with a distributed system. The processing effort for score performance is then greatly or totally reduced for the high-level composition software subsystem.

Each communication level includes the functionality of the lower levels, thus event specification and device subsystems may implement communication to level I, level II or level III, each of which may be usable and appropriate for different systems. This scenario is summarised in figure 7 below.

The MIDAS system - with its 'MII' front end processor - is envisaged to facilitate communication at all three levels (see section 11.3).

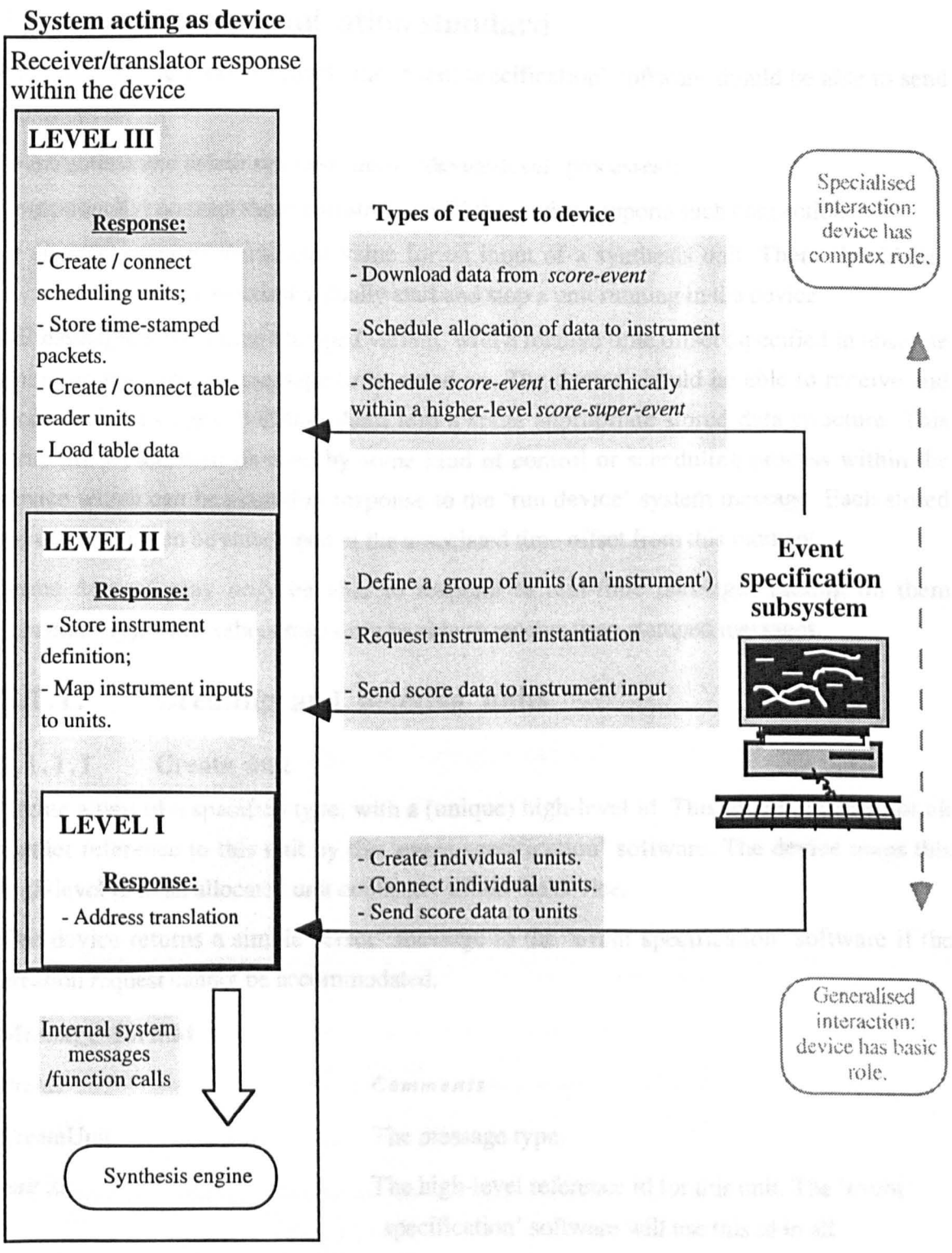


Fig. 7 Three levels of interaction between two systems

The following message types and data fields are proposed for each level. It should be reiterated that no particular hardware interface is specified, and the data format used to communicate the information is flexible - the field order and data size is *not* prescribed by the proposed standard.

Note that a message field value *may* appear as an ASCII-coded *name* (as well as a number) in the event specification software, or even in the front-end interface to the device. Some of the example messages below show such names, for clarity.

5.1. Level I communication standard

Before or during score playback, the 'event specification' software should be able to send messages to:

- instantiate and delete synthesis units ('device-level' processes);
- (optionally) connect them into structures if the device supports such connections;
- specify a control parameter value for an input of a synthesis unit. There should be control parameters to individually start and stop a unit running in the device.

All messages have a time-stamped variant, with a relative time offset (specified in absolute units (eg ms)) for the message to be acted on. The device should be able to receive and store such messages, building them into a some appropriate stored data structure. This structure would then be used by some kind of control or scheduling process within the device which can be started in response to the 'run device' system message. Each stored message will then be acted upon at the associated time offset from this moment,

Some devices may *only* be able to respond to real-time messages (acting on them immediately), while others may only be able to receive time-stamped messages.

5.1.1. Creating and deleting units

5.1.1.1 Create unit

Create a unit of a specified type, with a (unique) high-level id. This id will be used for all further reference to this unit by the 'event specification' software. The device maps this high-level id to an allocated unit contained within the device.

The device returns a simple "error" message to the 'event specification' software if the creation request cannot be accommodated.

Message format

Fields	*Comments*
CreateUnit	The message type.
unit id	The high-level reference id for this unit. The 'event specification' software will use this id in all subsequent references to this *unit.*
unit-type id	The type of the *unit* to be created.

Message Format		
CreateUnit	*unit* id	*unit-type* id
Examples		
CreateUnit	1	'OSCIL'
CreateUnit	90	'PART'

5.1.1.2 Create unit with time stamp

This message is as 5.1.1.1 but with an additional supplied time offset. The message will then be scheduled by the device, and acted on at the specified time, after it has received a 'run' message (see 5.1.4.1).

Message Format			
CreateUnitAtTime	Time	*unit* id	*unit-type* id
Examples			
CreateUnitAtTime	500	1	45
CreateUnitAtTime	0	90	9

The above two example messages would request the creation of two synthesis units: a unit of type 45 (a type code understood by the device) at a time of 500ms, and a unit of type 9 at a time of zero. These times are the offsets from when the device is started running, ie when it receives a 'run' message (see 5.1.4.1).

5.1.1.3 Delete unit

This message requests the device to delete a specified unit, using its high-level id.

Message format

Fields	***Comments***
DeleteUnit	The message type.
unit id	The id of the *unit* to be deleted.

Message Format	
DeleteUnit	*unit* id
Example	
DeleteUnit	1

5.1.1.4 Delete unit with time stamp

This message is as 5.1.1.3 with a supplied time offset for when the message is to be acted on by the device. If the unit is still running at this time, the device should first stop the unit running before deleting it. A less sophisticated implementation might require the controlling system to send a prior message to the device to stop the unit, before requesting its deletion.

Message Format		
DeleteUnitAtTime	time	*unit* id
Example		
DeleteUnitAtTime	500	90

5.1.2. Connecting and disconnecting units[1]

5.1.2.1 Connect unit to another unit

If the device supports the connection of units to each other (rather than simply grouping them to run in parallel), then this message will request the device to connect a specified output of a specified unit (using its high-level id) to a specified input of another[2] unit (also using its high-level id). A synthesis device will usually have at least one unit which acts as, or represents, a sound output port (eg a DAC, or disk file).

Message format

Fields	*Comments*
ConnectUnit	The message type.
Source *unit* id	The id of the *unit* the connection is from.
Source *unit* output id	The id of the *unit* output the connection is from.
Destination *unit* id	The id of the *unit* the connection is to.
Destination *unit* input id	The id of the *unit* input the connection is to.

Message Format				
ConnectUnit	Source *unit* id	Source *unit* output id	Destination *unit* id	Destination *unit* input id
Examples				
ConnectUnit	1	'A-OUT'	90	'AMP'
ConnectUnit	5	1	6	2

5.1.2.2 Connect unit to another unit with time stamp

This message is as 5.1.2.1 with a supplied time offset value for scheduling.

Message Format					
ConnectUnitAtTime	Time	Source *unit* id	Source *unit* output id	Destination *unit* id	Destination *unit* input id
Examples					
ConnectUnitAtTime	0	1	3	89	'AMP'
ConnectUnitAtTime	500	1	3	90	'AMP'

---

1 NB. These messages are not statutory.

2 Or even the *same* unit to set up feedback networks.

5.1.2.3 Disconnect unit from unit

This message requests a device to disconnect a specified output of a specified unit (using its high-level id) from a specified input of another unit.

Message format

Fields	*Comments*
DisconnectUnit	The message type
Source *unit* id	The id of the *unit* the connection is from.
Source *unit* output id	The id of the *unit* output the connection is from.
Destination *unit* id	The id of the *unit* the connection is to.
Destination *unit* input id	The id of the *unit* input the connection is to.

Message Format				
DisconnectUnit	Source *unit* id	Source *unit* output id	Destination *unit* id	Destination *unit* input id
Examples				
DisconnectUnit	1	'A-OUT'	89	'AMP'
DisconnectUnit	1	'A-OUT'	90	'AMP'

5.1.2.4 Disconnect unit from unit with time stamp

This message is as 5.1.2.3 with a supplied time value for scheduling.

Message Format					
DisconnectUnitAtTime	time	Source *unit* id	Source *unit-output* id	Destination *unit* id	Destination *unit-input* id
Examples					
DisconnectUnitAtTime	0	1	'A-OUT'	89	'AMP'
DisconnectUnitAtTime	500	1	'A-OUT'	90	'AMP'

5.1.3. Sending data to units

5.1.3.1 Send data value to unit

This message sends a data value to a specified input of a specified unit in a device (using its high-level id). This data may start or stop the unit if addressed to the appropriate input.

Message format

Fields	*Comments*
SendValue	The message type.
unit id	The id of the *unit* to be sent to.
unit-input id	The id of the *unit* input to be sent to.
data value	The data value to be sent.

Message Format			
SendInputValue	*unit* id	*unit-input* id	data value
Examples			
SendInputValue	1	2	10000
SendInputValue	70	'PITCHBEND'	68
SendInputValue	7	'PITCH'	446.8

5.1.3.2 Send data value to unit with time stamp

This message is as 5.1.3.1 with a supplied time value for scheduling.

Message Format				
SendInputValueWithTime	time	*unit* id	*unit-input* id	data value
Examples				
SendInputValueWithTime	0	1	1	10000
SendInputValueWithTime	0	1	2	470
SendInputValueWithTime	10000	70	'PITCHBEND'	68
SendInputValueWithTime	10010	70	'PITCH'	446.8

5.1.4. System messages used in 'time stamped' mode

If time-stamped messages are sent to, and stored by, a device, then messages will need to be sent to it, either to set up the device, or start or stop its scheduling processing running. These messages are designed to request an immediate response from a device, and

consequently have no time-stamped variants. When a device starts running, it should then start processing its stored time-stamped messages - acting on them at the appropriate time.

5.1.4.1 Run device

Start the appropriate device scheduling process. This will then perform whatever device-specific action is appropriate to act on the list of time-stamped messages previously received. Such action could consist of parsing a score or instruction list text file which has been built up from the time-stamped messages, or starting up a process to read stored time-stamped messages stored in it. The effective system clock time should be reset to zero, thus the device should start from the beginning of its stored schedule.

Message Format
RunSystem

5.1.4.2 Stop device

Stop ('pause') the device's system clock, thus suspending processing of the schedule. The device should take appropriate action to suspend or kill the running of all active units.

Message Format
StopSystem

5.1.4.3 Continue device

Restart the device scheduler, but without resetting the system clock time to zero, thus the device will start from the current system time.

Message Format
ContinueSystem

5.1.4.4 Set device system time

The device's system clock time is set to the specified value. A 'Continue device' message will then start the scheduling from here.[1]

Message Format	
SetSystemTime	*time*

---

1 As messages with a time-stamp which is before this 'start time' will not then be executed, the correct system 'data state' may not be present when the device's stored message list is run from an arbitrary time. Thus, whether it is *useful* to start from a place other than the beginning is up to the high-level user.

5.1.4.5 Reset device

This will erase from the device any temporary data structures which have been built up by previous messages, and empty the message schedule (eg clear the file, or empty the memory buffer used to store such messages). Any subsequent time-stamped messages received will therefore be assembled as part of a new performance.

Message Format
ResetSystem

5.1.4.6 Set device system sample rate

This sets the sample rate of the system, ie the number of samples required to be calculated per second for each sound output channel in use. These channels may be audio output ports (DACs) for real-time sound production, or channels present on a soundfile which is produced.

Message Format	
SetSystemSampleRate	*rate*

The level I communication scenario is summarised in figure 8 below, with messages from the composition system (on the right) being processed within the device by some kind of higher-level data handler, which may then communicate with lower-level synthesis processes within the device, as shown on the right.

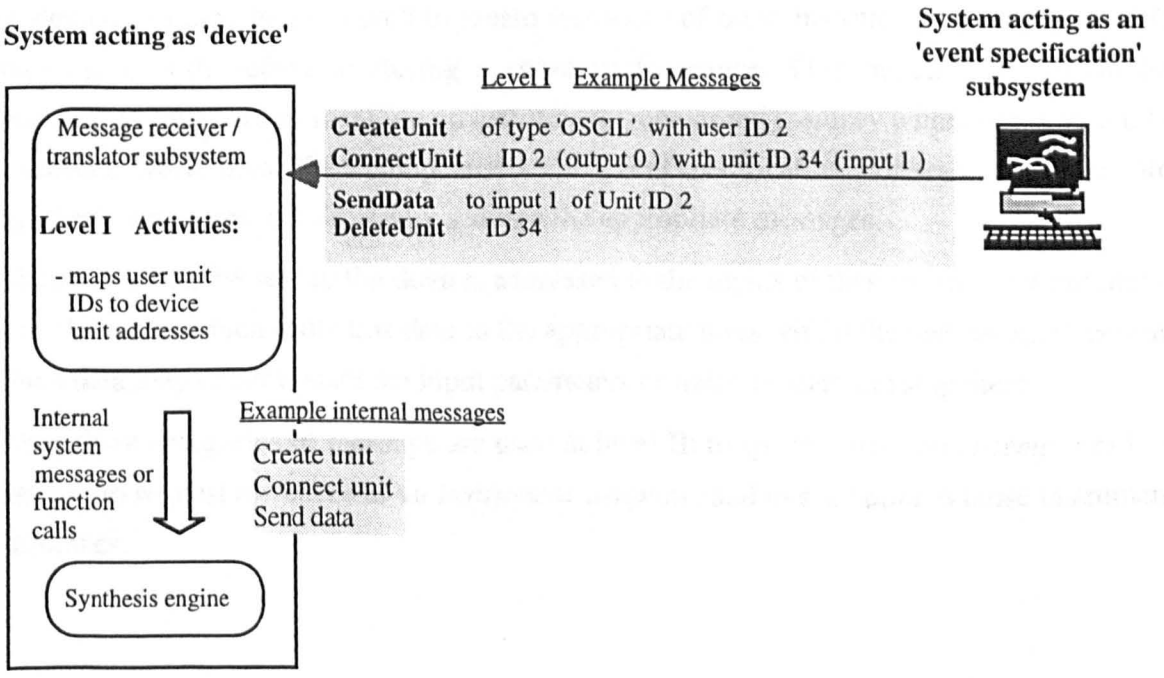


Fig. 8 Level I communication between two systems: one acting as an event specification subsystem and the other acting as a device

5.2. Level II communication standard

Level II incorporates all the functionality and messages of Level I, as well as additional facilities to enable an external 'event specification' system to specify and store an instrument-template (ie a synthesis structure specification and optional default values for its inputs) in device memory.

This level of communication requires a device to possess the kind of data structure and processing complexity which is more appropriately handled by a high-level software process which then communicates with the synthesis engine. This software could be hard-coded into the operating system of commercial devices. For existing academic custom systems, it could be a separate program which processes external control commands and data, writing out and executing any text files, or sending software commands as required by its synthesis engine.

Many devices will also have their own specific facilities to allow instrument specifications, to be defined via a user-interface system built in to the device itself, or via lower-level device specific data input methods, such as reading a file, or inputting low-level device data. For example, most MIDI devices can have instrument-templates ('patches' or 'programs') specified from their front panel controls, or via any number of editing software packages; the MIDAS device (as described above in 3.2.1.9) can have structures specified via textual input into its MII interface layer, or via the graphic 'Canute' editor program.

Thus, a device should be able to store instrument-templates which describe instrument specifications. These may be defined by a possible variety of methods, as discussed above, but should include the use of level II messages (described below) to enable a remote composition system to specify *instrument-templates* for storage in a device in a standardised way.

A device can then be requested to create *instances* of these instruments by other level II messages, both before or during a score performance. This instantiation might be understood by a user as recalling an instrument from stored memory within the device into a current 'work' area. Thus composition and 'event specification' software can instantiate synthesis structures in a device by sending the appropriate messages.

Finally data can be sent to the device, addressed to the inputs of these instrument instances. The device will then route this data to the appropriate units within the instrument structure. Such data may either update the input parameters of units, or start and stop them.

Thus three categories of message are used at level II: to *specify instrument-templates* to a device, to request *instances* of an *instrument-template*, and to send *data* to those instrument instances.

5.2.1. Defining an *instrument-template* within a device

An *instrument-template* is a description of a network of units[1] in the device, plus (optional) default 'initialisation' input values for them. It consists of a number of connected *unit-templates* plus any default initialisation values. A *unit-template* is an entity within an *instrument-template* which defines a unit which is to be incorporated within an instrument when it is instantiated. Thus, an *instrument-template* can be used to create one or more instrument instances, each of which contains one or more units; each derived from a corresponding *unit-template.* This is illustrated in figure 9 below.

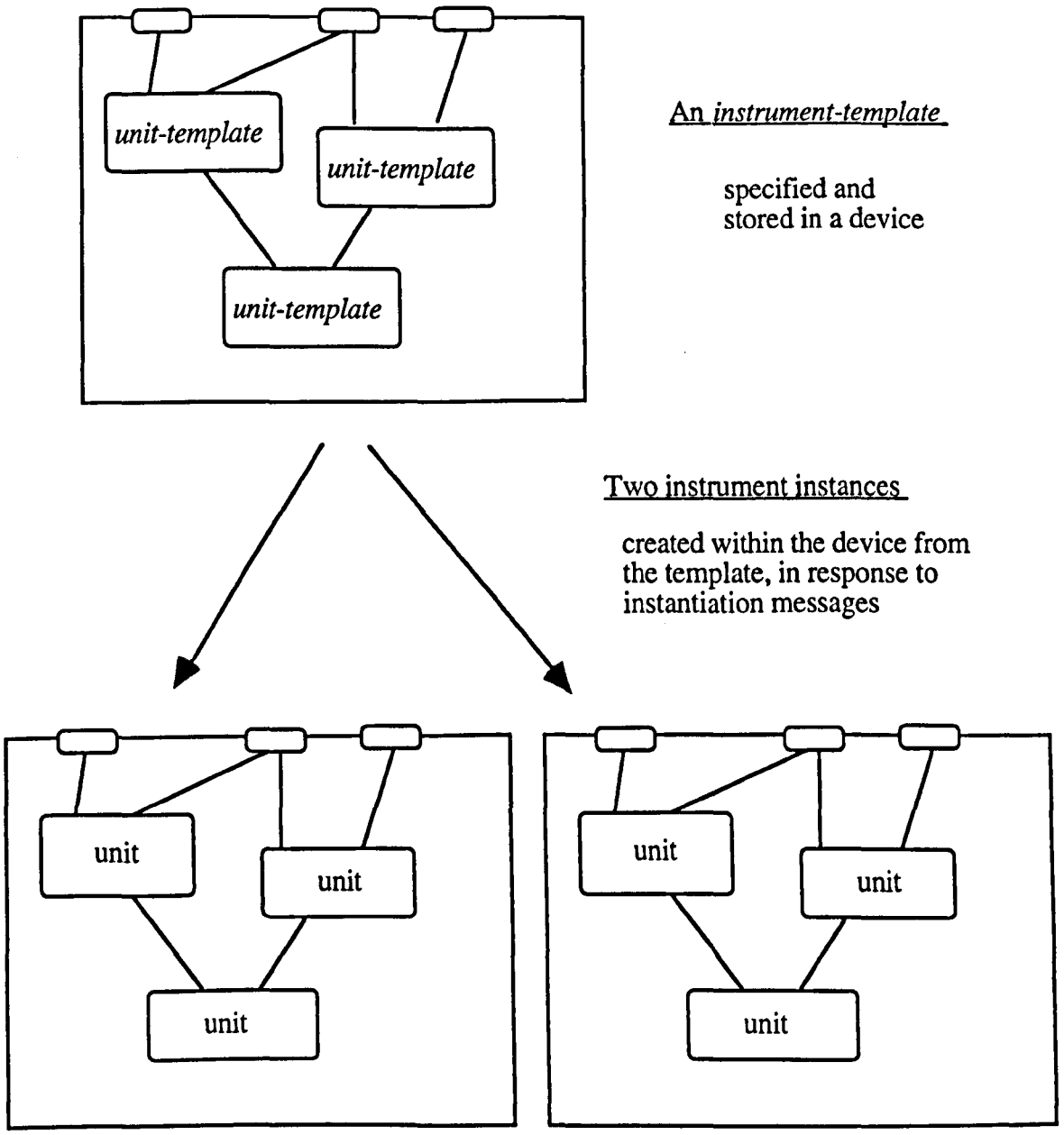


Fig. 9 Creating instrument instances from a stored *instrument-template.*

---

1 Any device-level synthesis process or module which is *not* composed of independent lower-level entities.

Table data to be used by all instances of this *instrument-template* may also be specified and stored in the device.

There is no necessity to have time-stamped variants of these messages, as they are simply setting up structures in a device to define an *instrument-template*. Thus, they could all be sent to a device at the start of a session.

6 message types are used to specify *instrument-templates*:

5.2.1.1 Start *instrument-template* definition

This signals to a device that a new *instrument-template* definition is to be specified. The user id which is supplied by this message is then used in all subsequent references to this *instrument-template*. If an *instrument-template* with this id has *already* been defined in the device, it should return an error message to the event specification system.

Message Format

Fields	*Comments*
StartInstrumentTemplate	The message type.
instrument-template id	The id of the *instrument-template* being built up.

Message Format	
StartInstrumentTemplate	*instrument-template* id
Example	
StartInstrumentTemplate	1
StartInstrumentTemplate	'Complex Am'

5.2.1.2 Specify a *unit-template* within an *instrument-template*

This message specifies to a device the existence of a *unit-template* of defined type within the designated *instrument-template*. This template must have previously been declared to the device via the previous message (5.2.1.1). The *unit-template* has an associated type code, which will be known to the device as one of the unit types it supports. It should return an error message if the unit type is not known to it.

One or more parameters pertaining to the precise nature of the unit in the device may also be specified, eg an 'N-MIXER' *unit-type* may need the number of inputs specifying when it is instantiated.

Message Format

Fields	*Comments*
SpecifyUnitTemplate	The message type.
instrument-template id	The id of the *instrument-template* being built up.
unit-template id	A unique number/symbol identifying each *potential-unit* within the *instrument-template*. The 'event specification' system will subsequently refer to this *unit* by this number.
unit-type id	The id of the *unit-type*.
[*unit-type* instantiation parameters..]	One or more (optional) parameters specifying any user-definable characteristics of the *unit*, eg the number of inputs for an ADDER.

Message Format				
SpecifyUnitTemplate	*instrument-template* id	*unit-template* id	*unit-type* id	[*unit-type* instantiation parameters]
Examples				
SpecifyUnitTemplate	'Complex Am'	3	'OSCIL'	
SpecifyUnitTemplate	'Complex Am'	2	'N-MIXER'	2

5.2.1.3 Specify a connection between *unit-templates*

This message specifies a connection from a ('source') output of a *unit-template* to a ('destination') input of another *unit-template* within an *instrument-template.* Note that a single output may be connected to the inputs of *several* units by sending a series of these messages.

Message Format

Fields	*Comments*
SpecifyUnitConnection	The message type
instrument-template id	The id (name or number) identifying the *instrument-template* being built up.
Source *unit-template* id	A unique id identifying the source *unit-template.*
Source *unit-template* output id	An id identifying the output of the source *unit-template* to be used for the connection.
Destination *unit-template* id	A unique id identifying the destination *unit-template.*
Destination *unit-template* input id	An id identifying the input of the destination *unit-template* to be used for the connection.

Message Format					
SpecifyUnitConnection	*instrument-template* id	Source *unit-template* id	Source *unit-template* output id	Destination *unit-template* id	Destination *unit-template* input id
Examples					
SpecifyUnitConnection	'Complex Am'	4	'A-OUT'	6	'IN-1'
SpecifyUnitConnection	'Complex Am'	4	'A-OUT'	7	'IN-1'
SpecifyUnitConnection	3	50	1	49	2

Thus the first two example messages above would connect from the 'A-OUT' output of a *unit-template* with id = 4, to the 'IN-1' inputs of two *unit-templates* (of ids = 6 & 7).

The third example would connect from the output (id=1) of the *unit-template* with id = 50, to the input (id =2) of the *unit-template* with id = 49.

5.2.1.4 Specify an *instrument-template* input.

An *instrument-template* has inputs which act as entry points for data. This message declares to a device that an input is present on an instrument-template. Each input can then be specified (via the next message 5.2.1.5) as being connected to one or more potential units within the template.

When an *instrument-template* is subsequently instantiated by a device to create an instrument (as in figure 8 above), each of the *unit-templates* in the template is instantiated in the device as a unit. Each *input* of the template should then be mapped by the device to the appropriate[1] unit input(s) in the instrument.

A data value can subsequently be sent to the device (by message 5.2.3) addressed to a specified *instrument-template* input. The device will then route the data (via this mapping) to these unit input(s).

Message Format.

Fields	*Comments*
SpecifyInstrumentTemplateInput	The message type.
instrument-template id	An id identifying the *instrument-template* being built.
instrument-template input id	An id identifying an input of the *instrument-template.*

Message Format		
SpecifyInstrumentTemplateInput	*instrument-template* id	*instrument-template* input id
Examples		
SpecifyInstrumentTemplateInput	'Complex Am'	'am waveform'
SpecifyInstrumentTemplateInput	2	3

1 Corresponding to the inputs of *unit-templates* connected to the input of the *instrument-template.*

5.2.1.5 Specify a connection from an *instrument-template* input to a *unit-template*

This will be a connection from a 'source' *instrument-template* input to a 'destination' input of a *unit-template.*

NB. A single *instrument-template* input may be connected to the inputs of *several* units by sending a series of these messages.

Message Format

Fields	*Comments*
SpecifyInputConnection	The message type.
instrument-template id	The id of the *instrument-template* being built up.
instrument-template input id	The id of the source *instrument-template*-input.
Destination *unit-template* id	A unique number identifying the destination *unit-template.*
Destination *unit-template* input id	A unique number identifying the input of the destination *unit-template.*

Message Format				
SpecifyInputConnection	*instrument-template* id	*instrument-template* input id	Destination *unit-template* id	Destination *unit-template* input id
Examples				
SpecifyInputConnection	'Complex Am'	'am waveform'	5	'TABLE-ID'
SpecifyInputConnection	1	0	5	0

5.2.1.6 Specify an initialisation input value for a *unit-template*

This message specifies a value for the designated *unit-template* input within an *instrument-template*. The device stores this value, and will send it to the input of the corresponding unit when it instantiates an instrument from this template. This will be appropriate for those unit inputs which do *not* change value from one *score-event* to the next.

Note that the controlling 'event specification' subsystem should *not* subsequently be able to access this unit input, *unless* a connection from an *instrument-template* input has been set up (by message 5.2.1.5 above).

Message Format

Fields	*Comments*
SpecifyUnitInitialisationValue	The message type.
instrument-template id	The id of the *instrument-template* being built up.
unit-template id	A unique number identifying a *unit-template* within this *instrument-template*.
unit-template input id	The id of the input of this source *unit-template*.
Initialisation value	The value assigned to be sent to this input when this *instrument-template* is first instantiated in the device.

Message Format				
SpecifyUnitInitialisationValue	*instrument-template* id	*unit-template* id	*unit-template* input id	Initialisation value
Examples				
SpecifyUnitInitialisationValue	'Complex Am'	3	'TABLE-ID'	'loop1'
SpecifyUnitInitialisationValue	2	4	1	100

5.2.2. Requesting the creation and deletion of instrument instances in a device

Four message types are used in this category:-

5.2.2.1 Create instrument instances within a device

Request the creation (instantiation) of a specified number of instrument instances using a specified *instrument-template.*

The number of instrument instances requested is equivalent to the 'polyphony' (N), ie the number of instances ('voices') of this *instrument-template* which need to be created and run simultaneously in order to cope with the maximum number of simultaneous *score-events* assigned to that *instrument-template.* If this number is higher than the device can manage in the time allocated, an error message should be sent back to the 'event specification' system.

The device should allocate ids for these instrument instances in the range 1 to N. Alternatively a variant message could be defined which has a user-specified instrument id. In this case, the device would report an error if the specified instrument id is *already* in use by an instrument instance with this template id.

Message Format

Fields	*Comments*
CreateInstrumentInstances	The message type.
instrument-template id	The id of the *instrument-template* being built up.
Number of instrument instances	The number (N) of instrument instances requested to be created with this *instrument-template.* These can then be referred to by the high-level 'event specification' system using consecutive instrument instance id numbers (from 1 to N) in conjunction with this *instrument-template* id.

Message Format		
CreateInstrumentInstances	*instrument-template* id	Number of instrument instances
Example		
CreateInstrumentInstances	'Complex Am'	3

5.2.2.2 Create instrument instances with time stamp

This message is as 5.2.2.1, with a time-stamp when the instrument is to be created. This message would typically be used if an instrument instance is required during a performance, but device resources are insufficient to allow it to be instantiated at the *start* of the performance (see 5.2.2.4 for further details).

Message Format			
CreateInstrumentInstanceAtTime	Time	*instrument-template* id	instrument instance id
Examples			
CreateInstrumentInstanceAtTime	2370	'Complex Am'	1
CreateInstrumentInstanceAtTime	25500	'Complex Am'	2
CreateInstrumentInstanceAtTime	50050	3	1

5.2.2.3 Delete an instrument instance within a device

This message requests the deletion of an instrument instance within a device. An instrument is referred to by its *instrument-template* id, plus an instance id of this template. These *two* id numbers (for the template and instance *of* that template) are used so as to simplify data handling at each end - for example, a 'event specification' system can then more easily store and find the ids of the instances of a particular *instrument-template.*

Message Format

Fields	*Comments*
DeleteInstrumentInstance	The message type.
instrument-template id	The id of the *instrument-template.*
instrument instance id	The instrument instance id.

Message Format		
DeleteInstrumentInstance	*instrument-template* id	instrument instance id
Examples		
DeleteInstrumentInstance	'Complex Am'	1
DeleteInstrumentInstance	'Complex Am'	2
DeleteInstrumentInstance	3	45

5.2.2.4 Delete an instrument instance with time stamp

This message is as 5.2.2.3, with a time-stamp when the instrument is to be deleted. This message would typically be used if device resources were known to be inadequate to support the total number of instruments required *throughout* a performance. Thus some reallocation would be required, with instruments no longer needed being deleted so that this can then be instantiated.

Message Format			
DeleteInstrumentInstanceAtTime	Time	*instrument-template* id	instrument instance id
Examples			
DeleteInstrumentInstanceAtTime	20000	'Complex Am'	1
DeleteInstrumentInstanceAtTime	95100	'Complex Am'	2
DeleteInstrumentInstanceAtTime	200000	3	1

5.2.3 Sending data values to instruments in a device

Two message types are used in this category:

5.2.3.1 Send an input value to an instrument instance in a device

This message sends a single data value to a specified input of the designated instrument instance in the device. As in the previous message, instrument instances are referred to using two id numbers for the template and instance *of* that template. Note that this data could, if directed to a 'start' or 'stop' input of the instrument, start or stop it running.

Message Format

Fields	*Comments*
SendInstrumentInputValue	The message type.
instrument-template id	The id of an *instrument-template* in the device.
instrument instance id	The id of an *instance* of this template within the device
instrument input id	The id of the *instrument-template* input the value is to be sent to. The device should match this id with the corresponding input of the instrument instance.
value	The input value to be sent

Message Format				
SendInstrumentInputValue	*instrument-template* id	instrument instance id	instrument input id	value
Example				
SendInstrumentInputValue	'Complex Am'	9	'nuance'	3

5.2.3.2 Send an input value to an instrument instance with time-stamp

This message is as 5.2.3.1, with an additional time-stamp for when the input value is to become active.

Message Format					
SendInstrumentInputValueAtTime	Time	*instrument-template* id	instrument instance id	instrument input id	value
Example					
SendInstrumentInputValueAtTime	500	'Complex Am'	9	'nuance'	3

The level II communication scenario is summarised in figure 10 below. Note that all level I communication facilities will also be available at level II.

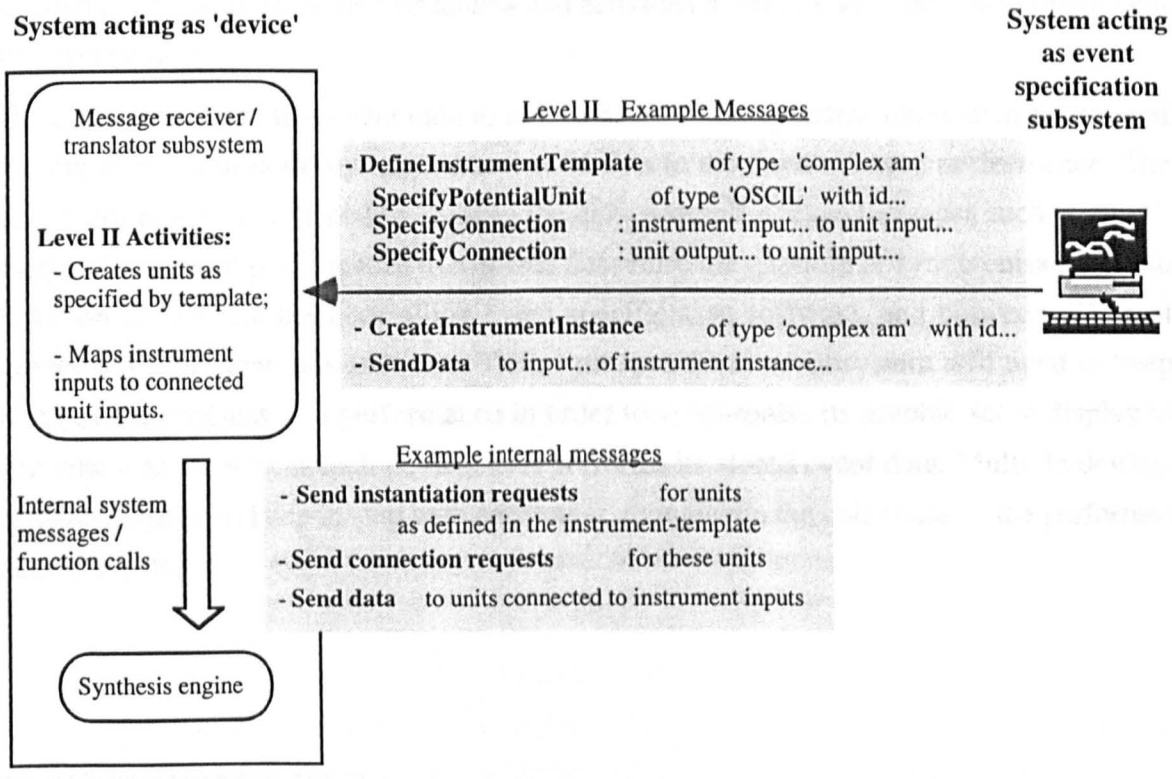


Fig. 10 Level II Communication between two systems: one acting as a high-level event specification subsystem, the other acting as a device

5.3. Level III communication standard

Level III communication allows score events and their parameter values to be *downloaded* from the controlling 'event specification' subsystem into a device.

Any 'event specification' subsystem which facilitates object-oriented score structuring (as described in 3.4.4) will possess hierarchical score structures which are here termed *score-super-events*[1]. A *score-super-event* will consist of a series of lower-level child events each with a time offset within the parent. These child events may *also* be *score-super-events* or be low-level 'note-like' events which do *not* contain further child events. These low-level events are here termed *score-events* [2].

Thus, to implement level III operation, a device will need to create data structures within itself (and allocate appropriate processes and memory) to provide scheduling facilities. This will enable parameter data from score events to be loaded into the device, and associated with particular instrument inputs. In addition events can be scheduled in a hierarchical structure within a device. Section 11.3 below illustrates (using the MIDAS system) an example of the kind of device structures and activities necessary for a device to implement this functionality.

Having downloaded this event data to a device, the event specification is then relieved of having to transmit possibly large amounts of data to the device during performance. The music can now be performed by sending the device simple control messages such as 'start'. Such delegation of performance to a device does raise the question of synchronisation, both between device and the controlling event specification software, and between different devices, if more than one is in use. The event specification subsystem will need to keep track of the progress of a performance in order to synchronise its graphic score display to the time elapsed within each device, as it performs its stored event data. Multiple devices obviously needs to keep in step with each other, to maintain the coherence of the performed sound events.

---

1 Note that *score-super-event* is hyphenated and in lower case. It denotes a *generalised* score structure, and should be distinguished from similar names for such structures within *specific* composition systems. For example, the E-Scape system describes such structures using a SuperScoreEvent object (see 8.3 below), while the Dmix system uses a HierarchyHolder object. Note the capitalisation in both cases, which indicate that these are *software objects.*

2 Again, note that *score-event.* is hyphenated and in lower case, and should be distinguished from the specific names used by systems. For example, E-Scape uses a ScoreEvent object to implement this structure, while Dmix uses a NoteEvent (again note the capitalisation).

Thus, some kind of master clock is required, which could be incorporated into the event specification system, or be a specific entity from which *all* the subsystems then derive a time clock.

Level III communication involves the use of Level II messages (5.2 above) to define *instrument-templates* and request the creation of instrument instances within a device in the same way as for level II operation. Additional messages are then used for the following purposes within level III:-

- 5.3.1. Defining *score-event* structures within a device.
- 5.3.2. Deleting *score-event* structures from a device.
- 5.3.3. Creating and deleting *score-super-event* structures within a device.
- 5.3.4. Starting and stopping event[1] structures within a device.
- 5.3.5. Sending real-time data values.
- 5.3.6. Creating and deleting data tables within a device.

5.3.1. Defining *score-events* within a device

A device should be able to create a data structure which corresponds to a *score-event* in the composition system, and refers to an allocated instrument instance in the device. The inputs of this instrument can be assigned to data tables within the device.

Four message types are used:-

5.3.1.1 Start *score-event* definition

This message requests the definition of a new data structure in the device to correspond with the specified *score-event*, using a designated instrument instance. This instrument instance will have been previously created within the device (by message 5.2.2.1 or 5.2.2.2), and is referred to by the id of the *instrument-template*, and the instance id 'within' this template.

To determine *which* instrument instance to request with this message, the 'event specification' software needs to select a "free" or "potentially free" instrument instance id from those available in the device. A "free" instrument instance is one which has not yet been assigned to any *score-event*. This will be the case for the first N *score-events* using this *instrument-template*, where N is the 'polyphony' (number of instrument instances). A "potentially free" instrument instance is one which is already assigned to one or more *score-events*, which do *not* overlap in time with the new *score-event* (ie its *score-events* will all stop before the new one is to start, or start after the new one is to stop).

To select an appropriate instance within the device, the event specification subsystem needs to use its knowledge of the start and stop times of the *score-event*, plus the start and stop

---

1 An event can be either a *score-super-event* or *score-event*.

times of other *score-events* which are *already* allocated to instances of this same *instrument-template.*

If not enough instrument instances can be created in the device to cope with the demands of a long score, the event specification subsystem can request the *deletion* (by message 5.2.2.3 or 5.2.2.4) of instrument instances which have been used by earlier *score-events* but are no longer needed. The event specification subsystem can then request the creation of new instrument instances which can then be used to define new *score-events.*

This functionality could in theory be provided by a *device,* as most MIDI-based devices do at present. It is preferable, however, to provide the decision-making in the event specification subsystem. Not only can it provide user-control of the criteria for pre-empting existing events if device resources are fully utilised, but the event specification subsystem's knowledge of the score allows it to allocate device resources intelligently in advance of performance, so as to reduce communication traffic, and device allocation load. The device should return an 'error' message if the specified *score-event* id is already in use - ie the *score-event* has previously been defined in the device and not deleted (see message 5.3.2.2).

Message Format

Fields	*Comments*
StartScoreEventDefinition	The message type.
score-event id	A unique number identifying a high-level *score-event.*
instrument-template id	The id of the *instrument-template* to be used by this *score-event.*
instrument instance id	The instrument instance id.

Message Format			
StartScoreEventDefinition	*score-event* id	*instrument-template* id	instrument instance id
Examples			
StartScoreEventDefinition	23	'Complex Am'	1
StartScoreEventDefinition	1209	34	1

5.3.1.2 Assign a table to an instrument input

This message specifies a data table in the device which is to be linked to a designated instrument input, for a particular *score-event.* Thus the message specifies:

- A table id.
- An *instrument-template* input id.
- The *score-event* id.

A table with this id will be required to be known within the device (specified via messages in 5.3.6), although it need not necessarily be present when the device receives this message. The *score-event* id is mapped by the device to its corresponding data structure as described in 5.3.3.1 above .

Message Format

Fields	*Comments*
AssignTableToInput	The message type.
score-event id	A unique number identifying an E-Scape *score-event.*
instrument-template input id	The id of an input of the *instrument-template* used by this *score-event.*
Table id	The id (name or number) of the table to be assigned.

Message Format			
AssignTableToInput	*score-event* id	*instrument-template* input id	Table id
Examples			
AssignTableToInput	1209	'am waveform'	'triangle'
AssignTableToInput	100	5	1

5.3.1.3 Assign initial instrument input value

This message specifies a single initial value for a specified *instrument-template* input, for a specified *score-event.* The *score-event* id is mapped by the device to the instrument instance which has been assigned to this *score-event.*

The value specified in the message is associated with the designated input of this instrument instance[1]. It will remain active for the entire *score-event,* unless other values are sent during it (by message 5.3.5.1).

Message Format

---

1 More rigorously, with the instrument input which corresponds to the specified input of the *instrument-template* from which this instrument instance has been derived.

Fields	*Comments*
AssignInitialValueToInput	The message type.
score-event id	A unique number identifying a high-level *score-event*.
instrument-template input id	The id of an input of the *instrument-template* used by this *score-event.*
value	The data value to be assigned to this input

Message Format			
AssignInitialValueToInput	*score-event* id	*instrument-template* input id	value
Examples			
AssignInitialValueToInput	99	3	14000
AssignInitialValueToInput	100	3	15000
AssignInitialValueToInput	100	'body waveform'	'sine'

5.3.1.4 Finish *score-event* definition

This message informs the device that all *instrument-Template* inputs have had values or tables specified for this *score-event*, and specifies its duration. This duration may later be used by the device when compiling this *score-event* within a higher-level *score-super-event* structure.

Message Format

Fields	*Comments*
FinishScoreEvent	The message type.
score-event id	A unique number identifying a high-level *score-event*
duration	The duration in an agreed measure (typically milliseconds)

Message Format		
Finish *score-event*	*score-event* id	duration
Example		
Finish *score-event*	90	1200

5.3.2. Deleting *score-events* within a device

Two message types are used:-

5.3.2.1 Un-assign a data table from an instrument input

This message requests the unlinking of the table assigned to a particular *instrument-template* input for a specified *score-event.*

This is the inverse of 5.3.1.2 above.

Message Format

Fields	*Comments*
UnassignTableFromInput	The message type.
score-event id	A unique number identifying a high-level *score-event* .
instrument-template input id	The id of the *instrument-template* input to be unassigned for this *score-event.*

Message Format		
UnassignTableFromInput	*score-event* id	*instrument-template* input id
Example		
UnassignTableFromInput	90	5

5.3.2.2 Delete a specified *score-event.*

Requests the removal from the device of the data and structures associated with the specified *score-event.* This message is the inverse of 5.3.1.1 above.

Message Format

Fields	*Comments*
DeleteScoreEvent	The message type.
score-event id	The id of a *score-event* which has been previously defined in the device.

Message Format	
DeleteScoreEvent	*score-event* id
Example	
DeleteScoreEvent	90

5.3.3. Creating and deleting *score-super-events*

For full level III communication, a device must be able to build a data structure to facilitate the hierarchical scheduling of lower-level events. The event specification subsystem can thus download such an event structure to the device in *advance* of performance, then start or stop it with a single message. This has obvious benefits in negating communication bandwidth restrictions, but incurs costs: a time lag while downloading occurs, and there is a lack of score playback flexibility - the score is effectively 'compiled' into a single parent *score-super-event* data structure in the device. This can then only be played as a composite entity - ie with *all* its event components - albeit with the ability to start and stop at arbitrary times within it.

However, if the device supports the appropriate synthesis units, an *instrument-template* may be able to be defined which facilitates the creation of sound which is split into sonic events. A single such *score-event* may drive an algorithmic 'instrument' which generates many perceived 'sonic' events. The phrase 'sonic event' is used here, because it may *not* necessarily correspond to a *score-event* in the event specification system. The *score-event* would then not be seen as corresponding to a single sonic event itself (as it normally would), but rather, as defining one or more *control* parameters which affect the algorithmic *generation* of these sonic events.

Thus, a piece or section of music could then be created which consists of such sonic events, but which would be treated by the controlling event specification system as an algorithmic *process* (ie an instrument) which generates the sonic events. Such an instrument (process) would then be controllable *during* a performance (eg to alter which events are created) via data input messages (see 5.3.5) sent to the device from the event specification subsystem.

This is an example of a movable boundary between score and instrument data - one of the features of the E-Scape prototype event specification system (see 6.5.3).

A device may optionally omit implementation of this message category, which would then require the composition system to send messages to individually start and stop each *score-event* [1] using messages in 5.3.4 below.

Four message types are used in this category:

---

[1] These *score-events* will have been defined in the device by messages in 5.3.1.

5.3.3.1 Define a new *score-super-event*

This message requests the definition in the device of a new data structure to correspond with the specified parent *score-super-event* id, with the option to specify a maximum number of child events which will be subsequently loaded to it. Some devices may not support dynamic memory allocation (ie the allowing of the memory allocated to a structure to expand as new data is added). In this case the maximum number of child events which are to be loaded within this new *score-super-event* structure in the device must be specified when it is first defined.

Message Format

Fields	*Comments*
DefineScoreSuperEvent	The message type.
score-super-event id	The id of the new *score-super-event.*
[Max. no. of events]	[optional] The maximum number of child events which will subsequently be able to be loaded into this parent *score-super-event.*

Message Format		
DefineScoreSuperEvent	*score-super-event* id	[Max. no. of events]
Example		
DefineScoreSuperEvent	2	50

5.3.3.2 *Schedule* a child event

This message requests a device to schedule ('compile') a specified child event within the data structure of a specified parent *score-super-event.* in the device. The child event has a relative start time specified, within this *score-super-event.*

This child event may itself either be a *score-event* or another *score-super-event* (if the device supports such structures). This facilitates the downloading of hierarchical score structures from the event specification system to a device in advance of performance. These structures can then be performed in response to a single message from the composition system.

NB. These schedule request messages should *not* need to be sent in the time order of events within the parent event, but they do need to be sent to the device sufficiently in advance of performance.

Message Format

Fields	*Comments*
ScheduleEvent	The message type.
Event id	The id of the event to be scheduled within the parent *score-super-event.*
score-super-event id	The id of the parent *score-super-event.*
Start time	The relative time at which the event is to start within the parent *score-super-event.*

Message Format			
ScheduleEvent	Event id	*score-super-event* id	Start time
Example			
ScheduleEvent	90	2	0

5.3.3.3 *Un-schedule* a child event

This message requests a device to remove a child event from a specified parent *score-super-event* structure in the device.

This is the inverse of message 5.3.3.2.

Message Format

Fields	*Comments*
UnScheduleEvent	The message type.
Event id	A unique number identifying the child event to be *un-scheduled.*
score-super-event id	The id of the parent *score-super-event* the child is at present within.

Message Format		
UnScheduleEvent	Event id	*score-super-event* id
Example		
UnScheduleEvent	90	2

5.3.3.4 Delete a specified *score-super-event.*

This message requests the removal from a device of the data structures which correspond with the specified *score-super-event.* The device should then automatically recover any memory and processing resources allocated to this structure.

This is the inverse of message 5.3.3.1.

Message Format

Fields	*Comments*
DeleteScoreSuperEvent	The message type.
score-super-event id	The id of the parent *score-super-event.*

Message Format	
DeleteScoreSuperEvent	*score-super-event* id
Example	
DeleteScoreSuperEvent	2

5.3.4. Starting and stopping events within a device

A compiled *score-event* or *score-super-event* structure within a device can be started or stopped running (playing).

Two message types are used:-

5.3.4.1 Start an event immediately

This message requests the device to start the performance (playing) of a specified event. This event must have been previously defined in the device by messages in 5.3.1 or 5.3.3. The event can be played from its beginning to its end, or (optionally) *from* a start time and *to* a stop time within it.

The event could be a *score-super-event* containing deeply nested other events which define an entire piece, or could be a single *score-event* representing a single note in a piece.

The device should then perform the appropriate low-level synthesis processes, as specified by these events and their parameters. If the device is operating in off-line mode, it should be able to write a soundfile, then play it back *without* further instructions from the controlling event specification system.

Message Format

Fields	*Comments*
StartEvent	The message type.
Event id	A unique number identifying the event to be started.
[Start time	Optional start time within the event to start playing from.
[Stop time]]	Optional (if a start time is specified) stop time within the event for it to stop playing.

Message Format			
StartEvent	Event id	[Start time	[Stop time]]
Examples			
StartEvent	1	500	
StartEvent	2		
StartEvent	3	1000	4400

5.3.4.2 Stop an event immediately

This message requests a device to immediately stop an event which is playing.

Message Format

Fields	*Comments*
StopEvent	The message type.
Event id	A unique number identifying the event to be stopped.

Message Format	
StopEvent	Event id
Example	
StopEvent	2

5.3.5. Sending data values to *score-events* within a device

One message type is used:-

5.3.5.1 Send an input value to a *score-event*

This message sends a single data value *at any time* for a specified *instrument-template* input for a specified *score-event* (previously defined in the device).

This data value would typically come from a *score-event* parameter value within the event specification subsystem. Such values *could* be loaded into a data table in the device (as described in 5.3.6), then this table assigned to a *score-event* (see 5.3.1.2). This message provides the option of avoiding such use of tables in a device, which may be appropriate if only one or two data values are specified for a *score-event*, or if the device does not support the storage of tables or has insufficient memory for them.

This message also provides the facility to control a device in real-time (if the device can *operate* in real-time). This can allow a device to be used by a live performer.

Message Format

Fields	*Comments*
SendScoreEventInputValue	The message type.
score-event id	A unique number identifying the *score-event* defined within the device.
instrument-template input id	The id of the *instrument-template* input the value is to be sent to. The device should match this id with the corresponding input of the instrument instance used by this *score-event*.
value	The input value to be sent.

Message Format			
SendScoreEventInputValue	*score-event* id	*instrument-template* input id	value
Examples			
SendScoreEventInputValue	90	'nuance'	3

5.3.6. Creating and deleting tables within a device

Tables of data can be loaded to a device (if it supports such structures), with a specified high-level id. The event specification system can subsequently refer to the table by this id, which will be referenced to the appropriate table stored in the device. Other 'standard' tables may be available within the device already, and these should be accessible via similar table ids.

Other messages (for example to specify the algorithmic *generation* of tables in the device - as in CSound's 'GEN' functions) are likely to be required, and can be added in the future. A further degree of standardisation agreement on the selection and names of table generating functions will need to be agreed first, hence this aspect has been left for future work. In any case, algorithmic table generation could be carried out by the event specification subsystem and manually downloaded to the device for storage (in which case these messages are sufficient). At present, three message types are used:

5.3.6.1 Load table data

This message sends a number of values to be stored in a table within the device. This table is identified by a reference id.

Values can be grouped singly (eg. for a wavetable), or into twos (eg. for breakpoint functions), or threes or more. The format will only be important when the data is *read* and interpreted by a unit in the device. The data consists of the number of values to follow, and the reference id of the table, followed by a set of values. The device then references the table by this id. An error should be returned if a table with this id has already been allocated.

Message Format

Fields	*Comments*
LoadTable	The message type.
Table id	A unique id (number or name) identifying this table.
No. of values	The number ('N') of data values to follow.
value 1	Data for the first table value.
[value 2 ... value N]	Optional data for additional values.

Message Format							
LoadTable	Table id	No. of values	value 1	[value 2	...	...	... value N]
Examples							
LoadTable	'looped-att'	512	0	-3	-12	-25	... 0
LoadTable	2	128	0	0.2	-0.3	0.8	... -0.3

5.3.6.2 Load breakpoint table data

Load a specified number of data value pairs to a table with a specified id.

This is a special case of message 5.3.8.1. with values grouped explicitly into pairs called 'breakpoints'. Each breakpoint consists of a time and a value which is associated (ie pertains at) that time. This message is used to make overt the loading of such breakpoint tables - a breakpoint table *could* also be loaded using the more general message 5.3.6.1.

The message specifies the number of breakpoints to follow, followed by that number of breakpoint (time + value) data pairs.

Message Format

Fields	*Comments*
LoadBreakpointTable	The message type.
Table id	A unique number or name[1] identifying this table.
Number of breakpoints	The number ('N') of data *pairs* [time+value] to follow.
Breakpoint-1 data	Data for the first breakpoint. It consists of two data items usually signifying time and value.
[Breakpoint-2 data ... Breakpoint-N data]	Data for optional additional breakpoints.

Message Format								
LoadBreakpointTable	Table id	Number of breakpoints	Bp-1 Time	Bp-1 Value	Bp-2 Time	Bp-2 Value	... Bp-N Time	Bp-N Value
Example 1								
LoadBreakpointTable	Table id	Number of breakpoints	Bp-1 Time	Bp-1 Val	Bp-2 Time	Bp-2 Val	Bp-3 Time	Bp-3 Val
LoadBreakpointTable	'am waveform'	3	0	3	50	7	75	7
Example 2								
LoadBreakpointTable	Table id	Number of breakpoints	Bp-1 Time	Bp-1 Val	...Bp-3 Time	Bp-3 Val	...Bp-7 Time	Bp-7 Val
LoadBreakpointTable	45	7	0	1	...560	2	...1200	0

1 Same naming conventions as for breakpoint tables.

5.3.6.3 Delete table data

This message requests a device to delete a data table with the specified id. It is the inverse of 5.3.6.1 and 5.3.6.2.

Message Format

Fields	*Comments*
DeleteTable	The message type.
Table id	A unique id (number or name) identifying a table

Message Format	
DeleteTable	Table id
Examples	
DeleteTable	45
DeleteTable	'am waveform'

The level III communication scenario is summarised in figure 11 below. Note that all level I and II communication facilities should also be available at level III (see figure 7, above).

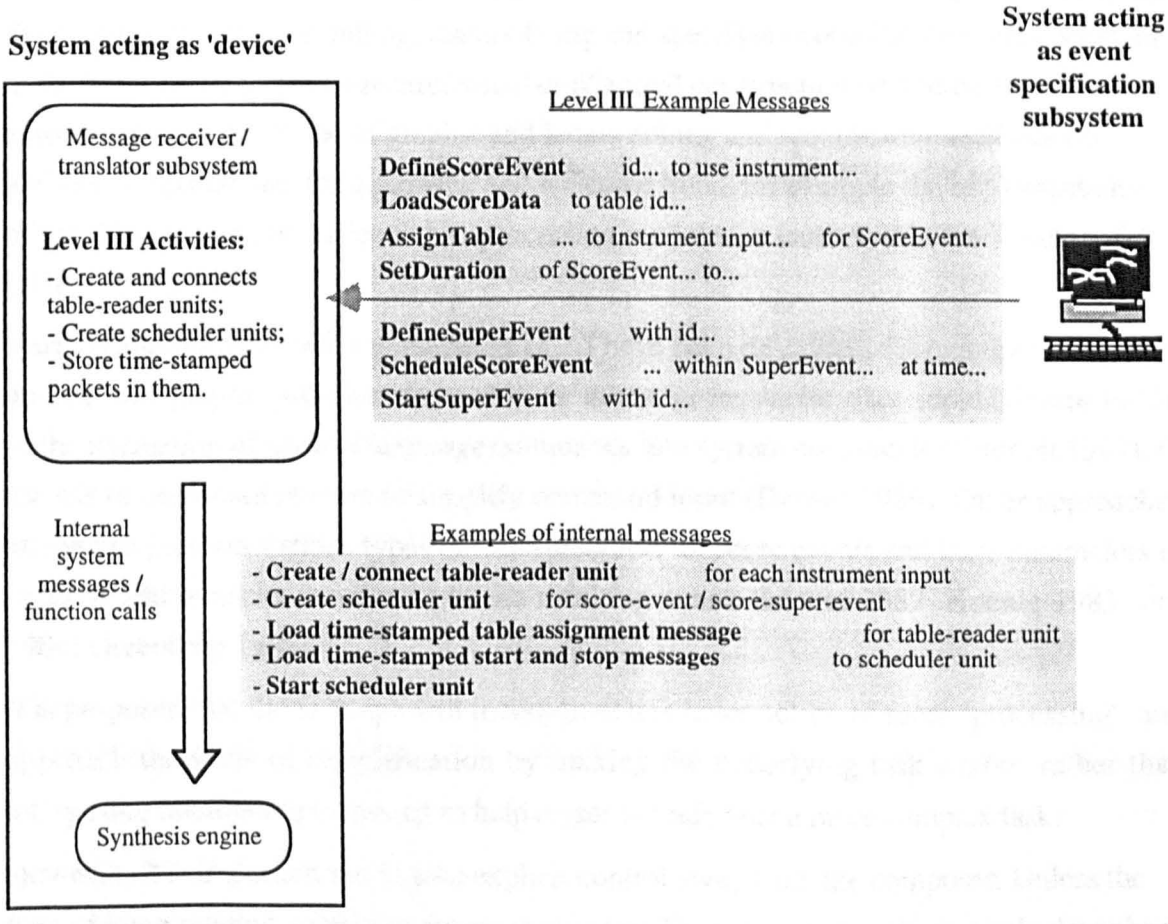


Fig. 11 Level III communication between two systems: one acting as a high-level event specification subsystem, the other as a device

6. Event specification software - the E-Scape system design goals

An 'event specification' software system (for composition and synthesis-control) called 'E-Scape' has been designed, and a prototype implemented, with two main objectives:

• to attempt to solve some of the difficulties and drawbacks of existing computer-based composition systems (as discussed in 3.3 &3.4);

• to illustrate the recommended composition software standards (outlined in 4.4) which will enable such systems to be usable within the system communication scenario proposed in chapter 4.

This chapter presents the design goals of the prototype E-Scape software. An evaluation of the degree of success with which each goal has been achieved in this implementation will be presented in chapter 12.

6.1 Design goal 1: Facilitation of score writing using *complex* instruments

6.1.1. Simplification of complex tasks

It has long been a recognisable general goal for composition software designers to simplify the composer's task in handling, manipulating and specifying complex structures, whether at the level of events (ie score structures) or of sound construction (ie synthesis algorithms). The provision of graphic and iconic editing and specification facilities (in addition to textual ones) is a growing and welcome trend, for example the implementation of the 'Esquisse' compositional data processing modules as icons within the 'Patchwork' GUI (see 3.2.1.7).

Various approaches involving elements of AI have been described, for example, the use of textual and graphic user interfaces able to allow for *imprecise* user input (Desain 1986), or the translation of *natural language* commands into system commands (Schmidt 1987), or the use of *command macros* to simplify command input (Decker 1986). Other approaches attempt to perform various types of 'interpretation' of score events and their parameters in order to add nuances and details to the resulting sound (Moore 1982; Koenig 1983; Fry 1984; Greenberg 1988).

It is proposed that the E-Scape will incorporate this latter notion of score 'processing', and approach the issue of simplification by making the underlying task *easier*, rather than interposing intelligent processing to help a user to undertake a more complex task.

However, it is important not to take explicit control away from the composer. Unless the type of interpretation applied to events is very familiar to him/her, or is precisely described, then there may be a danger that what is heard when playing a score does not correspond with exactly what is specified or displayed in it.

6.1.2. Presentation of instrument parameters for scoring

As described above in section 3.5.3, scores which use synthetic instruments often convey little information about their sonic or perceptual results. They also depend heavily on the particular synthesis devices used, and need to contain detailed data inputs for an algorithm implemented on a specific device or system.

Synthetic instruments thus require detailed score specifications, which must thus also be device specific.

The CHANT device, for example, facilitates complex transitions in sonority from a bell to a voice...

> "...by simultaneously manipulating the relative amplitude and width of the formants, and the rate of occurrence of impulsions[1]" (Wishart 1986).

CHANT uses a significant number of pulse generators and filters to create formants. This results in the need to specify a large number of parameters to create a single sound event (even neglecting the complex function calls needed for each of these parameters - see the CHANT text example in 3.4.6). In this CHANT example, it would be useful for an instrument designer to be able to *encapsulate* the relationships between the rate of generation of impulses and the many formant amplitudes and widths, and the degree of 'voice' or 'bell' character. Such a relationship could involve such things as data value limits at various points, and logical operators which can make decisions about data routing or values.

Such an instrument would be far easier to use, and could, if desired, insulate a user from the many unpredictable non-linearities inherent in many complex synthesis algorithms. For example, such an instrument could present a composer-user with parameters for initial and final degree of 'bell-like' character, and the composer could then simply specify a value for these parameters, and a duration. Composers should thus be able to make musically meaningful use of an instrument without being *required* to have prior knowledge of its inner workings.

This extreme example of simplification probably goes too far in shielding the user from the inner workings of a complex process, but another instrument could be designed to provide, say, four or five scoring parameters. The instrument would contain the same specification of the processes going on at 'device level' (eg within the CHANT synthesis functions), but with *different* overlaid software processing layers which present their inputs differently.

These instruments will be more complex to build, but as the composer no longer has to know about, design or build them, this task may be done by specialist instrument designers (who might not have many compositional ideas!). Most present systems require a composer to also be an instrument designer.

---

1 The word 'impulsions', as used here, denotes pulses of energy as produced by an impulse generator.

Although the topic of live performance is not dealt with in this thesis, it should be mentioned at this point that instruments which have such a simplified interface could feasibly be used by a performer. The relatively small number of score parameters needed means that it would be feasible to print out a copy of an E-Scape graphic score, assigning each event parameter to a performance controller[1] parameter to enable a live performer to play the score using the same instrument.

Thus, it would be desirable for a composer to be able to use a synthetic instrument in a similar way to a conventional acoustic instrument. For example, if a (rather detailed) score specifies a pizzicato pluck on a violin, then any violin can be used, with the assumption that it is built to the same standard (ie the same inputs will produce a comparable acoustic output). This works because of the high-level interface which the violin naturally provides for its operation. For example, the specification 'pluck it' (with parameters saying how hard, and where) actually affects a multitude of parameters of the physical processes which then occur in the violin body and strings.

The designer of a *synthetic* instrument, however, needs to explicitly specify its internal processes. What is then needed is for such an instrument to provide (as part of its design) a high-level entry for control information which affects its internal processes in a similar manner to the violin. This defined entry point could then be used (if desired) by composers to specify parameters which have a sonic or perceptual *meaning* (Anderson 1993b), as well as facilitate the *use* of an instrument without requiring a deep understanding of its internal functionality.

To continue the violin analogy, the present situation is as if a composer had first to design a violin as a mathematical model of its acoustic functioning (derived from first principles of physics), then specify the values of all the parameters which this mathematical model requires. To additionally then compose musically interesting material for this instrument would clearly require a prodigious and rare breadth of expertise.

6.1.3. Parameter processing by software Instruments

Ideally then, instruments should be controllable via high-level parameter inputs (as described above), which are processed within the *device*. However, this is unlikely to be easily achievable, as such processing will probably require quite complex algorithmic operations, and often require information about *other* parameters of score events, or even about the score as a whole (eg 'what is the overall *range* of a parameter as used throughout the score). Thus, E-Scape software should provide the facility to present, and then process, high-level compositional scoring parameters. This requires E-Scape to define an instrument definition in two parts:

---

1 Performance controllers include such things as keyboards, wind controllers, and other physical transducers which can translate human performance movements into data which controls or initiates musical events; even a computer mouse can be used as a performance tool (Hunt 1988).

• A set of *processes* which E-Scape will carry out (in software) in order to convert high-level scoring parameters[1] into the lower-level input parameters to send to the device.

• A *specification* (description) of the structure to be instantiated in the device.

This new broader definition of an instrument will be indicated by capitalisation - an Instrument (capitalised) is a software object which has two parts: (i) a *specification* of a synthesis structure on a device (an 'instrument' - lower case), (ii) additional processing functionality which is carried out in software.

The dividing line between these two aspects of an Instrument should be movable. The processing of the high-level scoring parameters could theoretically all be done by algorithms implemented as processes on the *device* (in addition to the lower-level synthesis processes); in this case the *specification* part of the Instrument would just specify these processes to be used on the device. Alternatively, most of the processing work might be done by E-Scape in software; the device would then only implement a simpler set of synthesis processes.

This means that an existing Instrument can be re-implemented using a new device, but still maintain the same 'compositional interface' within the score. Hence existing scores will *not* have to be changed. Alternatively, a new instrument could behave slightly *differently* (perhaps incorporating some more subtle parameter processing to give a perceived perceptual improvement). Again, the *score* would stay the same. This would be equivalent to a better player, with a better violin reinterpreting an existing score. This latter concept of an Instrument being able to *add* information to a score (ie to effectively 'interpret' it in the manner of a human performer) has been described in 3.4.7.5.

Thus electroacoustic scores could be seen as more like traditional CPN scores - ie as artistic creations which endure through time, which can be studied by students, and which can even (if a reader is familiar with the synthetic instrument concerned) be 'heard' in the mind's ear, as experienced musicians can at present 'hear' a conventional score when reading it.

This contrasts with the present divisive situation of electroacoustic music being perceived as 'different': as a specific set of instructions for a specific algorithm, rather than a musically meaningful specification.

---

1 Other terms for such high-level synthesis parameters might be 'compositional parameters', 'event parameters', 'sonic parameters' or 'perceptual parameters'.

6.1.4. E-Scape Instruments

Using this structure, a composer would be able to specify relatively *few* parameters for a score event, which yet uses and controls a *complex* synthesis structure - possibly distributed on several devices (see 6.2). E-Scape will tackle this goal in two ways:

• Firstly, an Instrument[1] entity (data structure) in E-Scape will define synthesis structures on one or more devices. However, an Instrument will *also* contain (within the E-Scape software) a wrapper - a 'front end' - of data processing entities. These will present a relatively small number of musically meaningful scoring parameters to a user who is creating an event which uses this Instrument.

Each of these parameters will then be translated into typically *many* control input values to synthesis structures (in one or more devices). This translation will be performed via arbitrarily complex algorithms, which an Instrument designer can specify within the software structure of the Instrument. Thus a user will need to specify far fewer input values, which can be defined to have a perceptual meaning (Anderson 1993b). The complexity will exist within the translation algorithm - hidden from a composer inside the Instrument. The Instrument can thus be effectively used in a score without a composer requiring knowledge of its inner workings.

• Secondly, all scoring parameters should have default values which can be specified by the Instrument designer. Thus a composer will need only to specify values for parameters he/she is interested in, and have no necessity to even look at the others.

In addition all Instrument input parameters should have maximum and minimum values which make sense (ie which produce musical output within the sonic scope of the instrument). Such a value for an Instrument input may *not* necessarily correlate with the maximum or minimum values of the *unit* input (within the synthesis structure defined by the Instrument) to which the Instrument input is connected, but may be constrained within a narrower range, so as to prevent certain values from being sent to the unit input. This would typically be done if certain input values - although 'legal' - would produce an unacceptable output from the unit in the context of the synthesis structure within which it exists, and other input values sent to it.

Thus composers can be presented with a 'safe' Instrument, ie one which will always produce sound within a designated sonic region. As stated above, this will obviate the need for composers to have intimate knowledge of an Instrument's workings before they can use it. The composer will be prevented from specifying parameter values which would produce unacceptable results.

---

1 The word 'Instrument' (with capitalisation) is used to describe a software entity within E-Scape, as distinct from an 'instrument' (lower case) which is a structure or network of units existing within a device.

This will be especially useful for non-linear synthesis algorithms such as FM. In some such cases, the maintenance of 'safe' output from a non-linear algorithm may require the input values to be constrained within several non-contiguous bands. An Instrument designer may also wish to deliberately constrain an Instrument user (composer) to particular parameter values for other reasons, such as forming a particular pitch scale for the Instrument.

To cope with such situations where non-contiguous *ranges* of acceptable values are to be specified for an input (rather than a simple maximum and minimum), then each input needs to be described by a more complex data structure. This should allow any number of individual values, or *ranges* of values to be specified as acceptable. Each range will consist of a maximum, minimum and resolution (value spacing).

If a composer *wants* to go outside the designed boundaries for an Instrument, he/she can edit the score parameter processing entities, to change the maximum or minimum values allowed. This will only *not* be possible if a specified score value would (after processing) produce a value outside the range of the *instrument* input - ie the input of synthesis unit(s) on a device. The E-Scape Instrument will contain a software definition of the synthesis structure in the device, which incorporates unit descriptions which specify the ranges of acceptable input values. An E-Scape Instrument will thus be able to detect (and prevent) a score parameter outside these unit input limits from being specified.

Such input value constraints on unit or synthesis structure inputs should be propagated up automatically to the higher-level 'perceptual' input of the Instrument, when these units or structures are loaded into an Instrument when being constructed. To do this will require some form of reverse processing through the translation algorithm, in order to determine the acceptable Instrument parameter values. Obviously, in some Instruments, some low-level (unit input) parameters will interact, making this process problematic. These aspects of Instrument design will be further clarified in chapter 8.

6.1.5. Presentation of Instruments

To recap, an E-Scape Instrument will consist of a specification / description of processing structures on one or more kinds of synthesis device, together with front-end layer of input parameter processing. This 'front end' can be designed to present a user of the Instrument with relatively few input parameters which are processed within software to produce (often many more) input parameters to the synthesis structure defined by the Instrument, and effect complex changes within the sound output.

Each parameter will have a name (designated by the Instrument designer) which may not necessarily be familiar or immediately meaningful to a composer. For example, an Instrument might have a parameter called 'flutter' which could alter in various ways the depth and rate of vibrato for each partial of a sound (via control inputs to the synthesis structure in the device). However, because an Instrument parameter can be designed to control many aspects of a synthesis algorithm which together can produce coherent spectro-

morphological[1] percepts (Smalley 1986), a composer is able to heuristically built up an experiential understanding of the connection between a parameter value and its sonic result.

A complex Instrument can thus be presented to a user in a compositionally useful way, with each of its parameters having a perceivable connection to the sonic output of the Instrument. A composer can thereby engage in scoring activity with such an Instrument in a straightforward manner, just as with an acoustic instrument.

A composer will first select an Instrument to use for a score event, and can then choose to specify values in this event for any of the Instrument's parameters. as well as a default value and allowable ranges of values. Thus a composer will not *need* to specify values for *every* parameter, and is prevented from specifying inappropriate values.

A composer should thus be shielded from the requirement to know or understand the details of an Instrument's construction and functioning in order to be able to compose scores using that Instrument. He/she will thus be able to use complex instruments - which can be designed by others (eg specialists) - in a musically natural way.

---

1 The aspect of sound structure which concentrates on the shaping in time of the entire range of frequencies (partials) which constitute a sound.

6.2. Design goal 2: Specification and control of *distributed* Instruments

6.2.1 Distribution of Instruments across devices

The E-Scape system should allow the construction and control of 'distributed Instruments' which use synthesis resources from several devices. Thus a *single* Instrument (assignable to a single event) can be created out of units or modules on *several* devices, which furthermore may be of different *types*. This Instrument would appear to a user as a single coherent entity, thus justifying the term 'distributed' (Tanenbaum 1984). Each device in the network acts as a dedicated server for particular units within the Instrument structure. This is one of the unique features of E-Scape, to the best of the author's knowledge.

Thus, a user of such a distributed system can specify very complex synthesis structures even if individual devices do not support such complexity. An example of such use would be for E-Scape to control four Yamaha SY99 MIDI synthesiser devices, so as to provide the basic synthesis facilities of the UPIC system (see 3.1.2.14). Each SY99 device provides 16 sampled sound wavetable oscillators, with frequency modulation using a further 16 oscillators, all with pitch and amplitude envelopes. An E-Scape Instrument could be designed which uses the UPIC architecture of a sampled wavetable oscillator with FM. The resources of four connected SY99s would then be allocated by E-Scape as necessary to perform a score. Graphic scores could thus be created by this system in a similar way to UPIC.

Some devices may not support inter-linking audio connections, but some present synthesis devices do allow external audio input. It is thus possible to conceive of a network of such devices linked by an externally-controlled[1] patch-bay (connection matrix) to create a controllable complex inter-linked synthesis structure, with data (either control signals, or analogue or digitised sound) passing between devices.

6.2.2 Blurring of division between Instrument and score

A event-oriented model *abstracts* a lot of the information which is produced by a functional model such as performance (Loy 1986).

In other words, events in such a model will contain relatively *little* information (eg in CPN[2], an event will specify pitch, duration and start time, with perhaps a further volume or articulation instruction). The remainder of the extensive and detailed data needed to produce a sound is supplied by a performer, in conjunction with the functioning - eg the physics and acoustics - of the instrument.

Hence, a single *event* can imply (ie contain or require) a large amount of continuous data. This large amount of data will then be *generated* from relatively sparse event data - ie from a

---

1 Many commercial examples utilise MIDI for external specification of connections.

2 Common Practice Notation - see 2.8.1.2.

few event parameters which then effect control over a complex synthesis process or algorithm.

Alternatively, a composer may wish to directly *specify* some or all the data *within* the event itself. Thus the event contains complex data, with a possibly simpler synthesis process. He/she should be able

A system should allow a composer the ability to choose which of these models to use, with either or both approaches available within the same system. E-Scape should thus enable the information pertaining to an event either to be overtly specified within it, or to be produced (functionally) by its instrument's algorithms, or both.

6.2.3 Distribution of Instruments between software system and devices

An E-Scape Instrument will thus contain both a software 'front-end' which converts score parameters into control inputs for low-level synthesis processes, and a *specification* of such processes to be carried out within a device. Thus, any control rate data processing can be specified by the Instrument to be performed either within the E-Scape software, *or* in the synthesis device[1]. This software/device boundary will be *movable*, with processing able to be shared out between the two elements as appropriate. It would normally be desirable to carry out as much processing as possible in the device for efficiency reasons, but many devices may not support all the logical and data flow control objects (as device units) which may be required. Thus some processing is always likely to be needed within E-Scape itself. E-Scape will carry out this processing *before* a score is played (see 9.2.2), storing the resulting device-level input values; thus score performance will not be compromised by a large complex process being carried out in software. However some significant time delay is likely when a score event is added or edited for example, as E-Scape may have to process a large number of data values in a time-varying parameter trace for a score event.

This concept of a movable boundary between software- and device-based processes can be illustrated with the design of two example E-Scape Instruments, both of which implement vibrato with 'second order morphology' (Wishart 1985). This allows a composer to specify the nature of a *change* in vibrato. These example Instruments both allow a composer to specify a starting and ending vibrato rate (the speed at which the pitch of the event varies cyclically), and a shape which governs the *transition* between these rates during the course of the event. The amplitude and shape of the vibrato itself are also specified but (for the sake of clarity within these examples) simply remain constant during the event.

The first example Instrument is shown in figure 12 below, and specifies all the necessary processing within the synthesis structure specification (in the bottom half of the figure), ie as a set of units in the device.

---

1 The synthesis device may *also* consists of software routines, but at a lower-level. In this chapter 'software' implies the higher-level E-Scape event specification system.

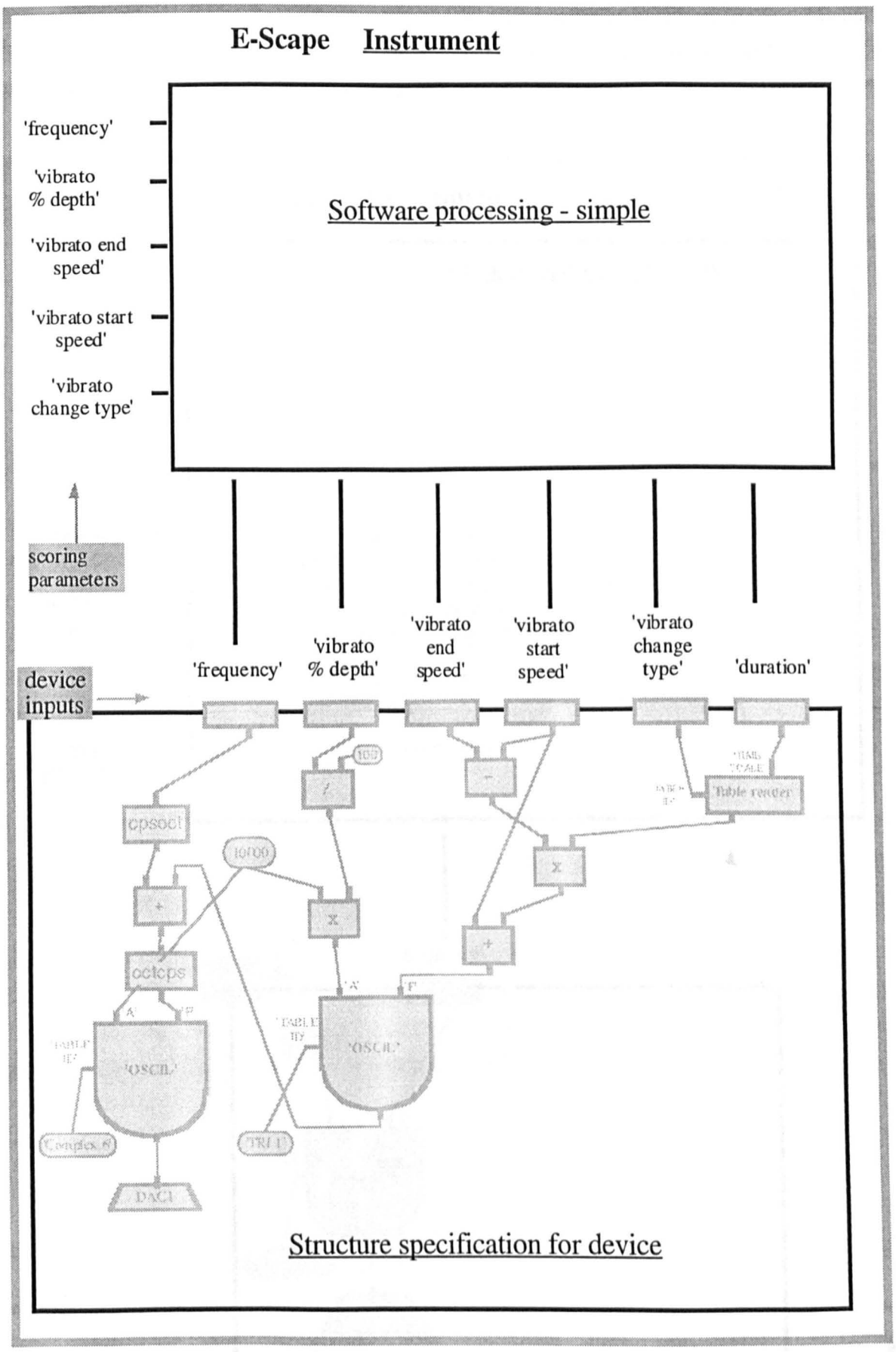


Fig. 12 Vibrato Instrument example 1

The details of the functioning of this device structure will be given later in section 9.5.2. but in order to construct it, the device must have certain types of unit process available, which can be interconnected freely. In this case, the units required include such things as logical operators, adders, dividers, as well as converters between different pitch units.

A second example Instrument which performs the same function is shown in fig. 13 below. This Instrument would be created so as to enable the use of a type of device which does *not* support the requisite units of the first example.

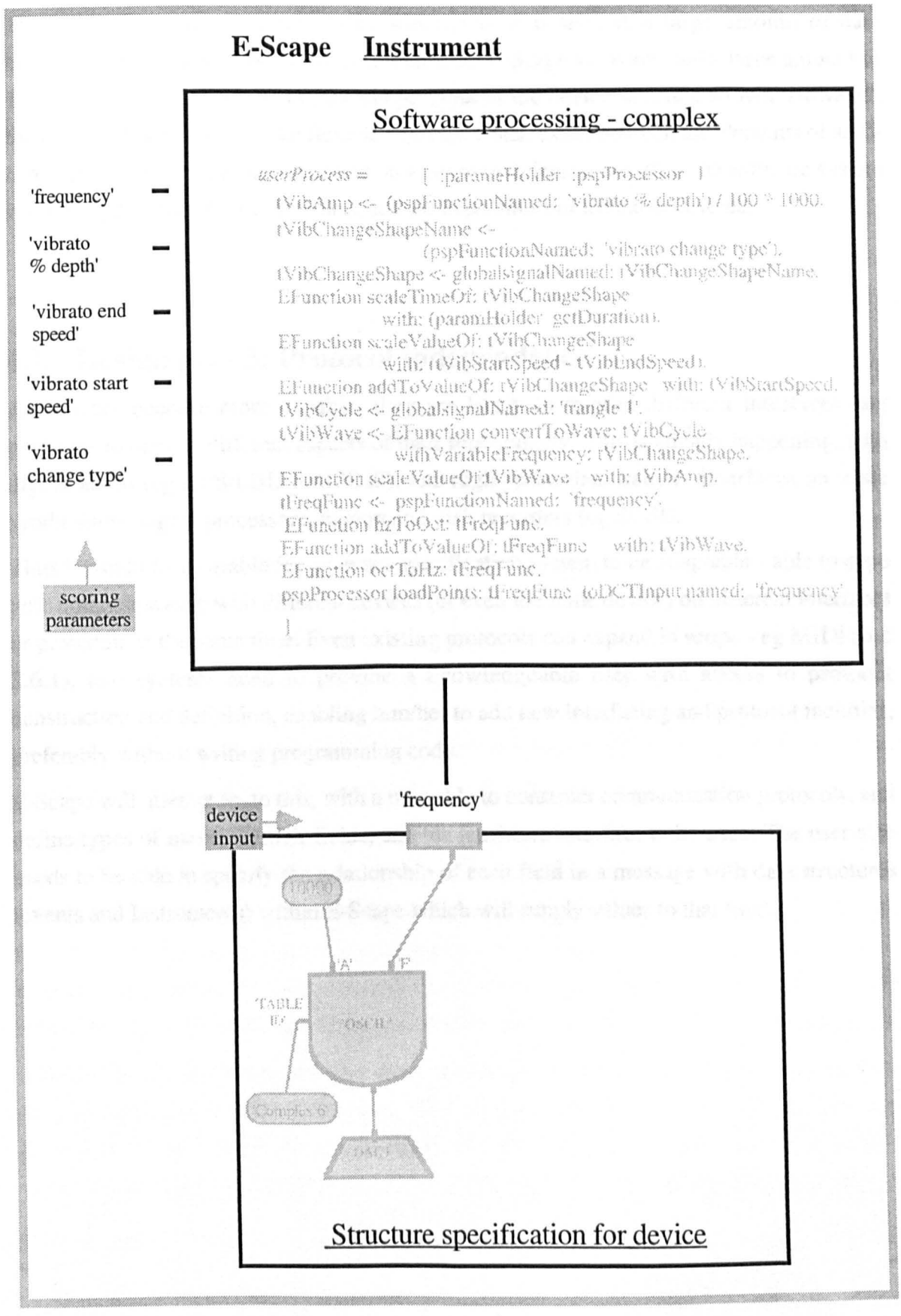


Fig. 13 Vibrato Instrument example 2

Note that the second Instrument has exactly the same *presentation* to the composer, with the five scoring parameters shown at the top left. However, in this case, the structure in the device is very simple, and all the processing occurs via functions within the software level of the Instrument (shown at the top of the figure).

This second example is likely to be less practical in use, as a large amount of data processing is required to be performed within the E-Scape software, and a large amount of data will then need to be sent to the single input of the device structure shown. However, these examples do illustrate the flexible interface which exists between the elements of an E-Scape Instrument; either implemented on a connected device, or within the software system itself, as appropriate for the situation, and the capabilities of available devices.

6.3. Design goal 3: Protocol independence

As devices become more capable, they are likely to require different interfaces and protocols to operate different aspects of their functionality. This is already happening, with digital audio (eg AES/EBU or SPDIF), and high-speed data transfer interfaces on some synthesisers, signal processors, mixers and disk recorders (eg SCSI).

Thus it would be desirable for an event specification system to be adaptable - able to cope with communicating with different devices (or even the same device) on different interfaces or protocols at the same time. Even existing protocols can expand in scope - eg MIDI (see 2.6.1), and systems need to provide a knowledgeable user with access to protocol construction and definition, enabling him/her to add new interfacing and protocol facilities, preferably without writing programming code.

E-Scape will attempt to do this, with a user able to construct communication protocols, and define types of message, their fields, and the hardware interface to be used. The user also needs to be able to specify the relationship of each field in a message with data structures (events and Instruments) within E-Scape which will supply values to that field.

6.4. Design goal 4: Presentation of device structures in an object-oriented and uniform manner

The paradigm of connected modules or units which contain signal processing algorithms is taken as the model of a synthetic instrument. All computer-based synthesis systems examined fit this model, and even dynamic systems involving bi-directional rules to model physical situations can be implemented as a network of unit processes (Pearson 1993).

A description of the synthesis units which constitute such a network in a device will form part of an Instrument definition. This network of synthesis units should be able to be structured into modules, each containing one or more units. Each module should be a self contained entity whose role is made explicit, with the type and range of data which pertains to its inputs and outputs clearly defined. Thus, if a composer wants to alter or examine the structure of an Instrument, he/she will be able to do so in a modular fashion. He/she will then only need to focus on the module(s) of interest, rather than being required to understand a complex entity in its entirety. Changes will then be easy to make, with a module within an Instrument being able to be altered or substituted for another module with no effect on any of the others. This concept will be clarified when object-oriented concepts are presented in chapter 7.

E-Scape will attempt to allow a user to specify and design synthesis structures on *all* types of synthesis device in a unified, consistent way. A user should be able to construct such structures from simpler modules, which are presented in a uniform manner even if implemented on *different* types of device. This allows faster, and easier instrument design by allowing building from pre-constructed lower-level modules, which may have been constructed by other users. Specifications of complex algorithm networks will be able to be *distributed* among different device types (see 6.2) if desired.

To cope with devices encountered which do not possess a modular object-oriented[1] structure, the E-Scape design should allow module specifications to be stored within *categories* for each type of device. For category message formats will be specifiable by a user for creating modules of that category (when sent to the device). A user will also be able to define restrictions on how modules of that category can be constructed - such as *which* categories of child module can be used inside its structure (see 8.6.4 for examples).

E-Scape should also provide the facility for a user to define a variety of information for a module input, which allow the module to be meaningfully treated as an entity in later use. An input can, for example, have a specified data range, rate, user name and default value.

---

1 See chapter 7 for details of this concept.

6.5. Design goal 5: Object-oriented structuring of scores and events

6.5.1 Facilitation of hierarchical event structuring

As described in 3.4.4, the ability to create and manipulate hierarchical event structures is of prime usefulness to all kinds of compositional activity (Punch 1991). Events should be nestable to arbitrary depth, (ie each able to contain others if desired), and these event structures should be able to be displayed as well as manipulated at any level. Gestural (time-varying) data should be applicable to parameters of events grouped at various levels, and have a specifiable effect on them (see 6.5.2 below).

6.5.2 Extraction and application of abstract gestural data

The ability to consider events as consisting of *abstract* data is desirable, and is prevalent in existing systems (see 3.4.7). However, as discussed in 3.5.3, there are also problems incurred if dealing with event data which abstracted from its context, ie if an event is isolated from the Instrument which will realise its data as sound output.

E-Scape should thus provide both of these structural models. To facilitate this, abstract data should be able to be extracted from, or applied to, events or higher-level event structures. For example, a set of breakpoint data (consisting of time and value pairs) could be extracted from the pitches of a set of events. This data could then, for example, be stretched (ie each time value proportionally increased), then applied as a modifier to some other parameter of events.

Thus, data should be able to be derived from any parameters of any group of score events at any level in a hierarchy.

6.6. Design goal 6: Provision of multiple, graphically presented, time-varying event parameters

6.6.1 Multiple event parameters

In 1967 Winckel wrote, regarding the notation of sounds:

> "The classical representation by means of a note head seems insufficient for the needs of contemporary composers, who are already beginning to make sense of the complexity of the individual sound" (Winckel 1967).

Composers who have used computers for both conventionally scored, but especially *electroacoustic* music composition have long wanted to be able to specify timbral details as part of a score, ie to be able to work in multi-parametric 'multi-dimensional vector space' (Xenakis 1971). In such a space, each event can possess many parameters beyond the pitch, loudness and duration of notes in conventional Common Practice Notation.

Such parameters can also represent more abstract compositional concepts, whose specification by the composer "becomes part of the subjective compositional process..." (Tipei 1987). Such 'compositional' parameters could for example globally alter, specify or affect parameters of lower-level events', or control an aspect of the algorithmic *generation* of such events such as 'texture' (Smalley 1986).

Thus E-Scape should allow the specification and display of multiple parameters for each event. These parameters can then control the synthesis processes which are assigned to that event, or algorithmic event-generating processes.

6.6.2 Time-varying event parameters

In addition to the above, composers have wanted to be able to use gestural time-varying data both in soundfile manipulation - eg (Banger 1983), and for sonic control during the course of a score event.

This use of *gesture* pertains especially to composers of electroacoustic music where the focus is on the timbral nature and development of sonic events.

> "The word 'gesture' has meaning in composition particularly in music that deals with timbre and gradual change rather than in a note-oriented approach"
>
> (Helmuth 1992).

Wishart introduces the concept of an "aural landscape" containing 'aural or sonic space'. This space could be considered as a multi-dimensional continuum out of which electroacoustic events can be 'carved' by specifying their time-varying sonic parameters (Wishart 1985). Wessel also describes sound events as consisting of:

> "2-D timbre slices located in n-D perceptual space" (Wessel 1987).

This gestural data must thus be specifiable and displayable for *multiple* parameters.

> "A gesture is something more than a change in one parameter. It is a combination of changes producing a unique unit of sound" (Helmuth 1992).

Emmerson recommends that the 'events' and 'signal change' procedures should be conflated into a single score, rather than the present divide into events which simply trigger a pre-designed sound-generating algorithm.

> "Recent software developments suggest that we are now in a position to incorporate signal processing instructions into the score" (Emmerson 1991)

Thus a *score* should be able to exercise continuous control over sound-creating processes, rather than just initiating them.

Such control can enable "an escape from predictable spectral morphology" inherent in some synthesis techniques (Roads 1985a). Thus the well-known (sometimes to the point of cliché) timbral evolution inherent in some synthesis techniques such as FM, can be altered by the composer by specifying parameters which *vary* during each event and allow flexible spectro-morphological changes within an event to be specified by a composer.

Thus E-Scape should provide for the specification and graphical presentation of time-varying parameters, either within each event or at a higher-level in the event hierarchy.

6.6.3 Event parameter display

There is still much research work to be done on the most effective way of visually presenting multi-parametric data to the user (Haus 1983). The use of colour, shape, texture or 3-D perspective could all be options, for example. This topic is in the area of HCI[1], and is beyond the scope of this project, hence the most straightforward and *familiar* presentation scheme has been chosen (as used in almost all the systems examined in chapter 3) which will thus involve the least need for *interpretation* by a user.

E-Scape will therefore display time-varying data for each event parameter as a time-domain trace in a 'piano-roll' style graphic editor (see fig. 5, section 2.8.1). Where multiple parameters are present, all the displays are placed vertically below each other with the same time base for all shown at the bottom of the consolidated display window. Parameter traces will thus appear like 'shadows' of the event in different parametric dimensions. This approach is obviously rather space inefficient, but users can select which parameters they wish to view. A composer would not typically be working on more than a handful of parameters at the same time. In addition, large or virtual scrolling screens can allow a workable number of simultaneously available parameters to be edited. The parameter processing features of E-Scape Instruments (described in 6.1) will also encourage a reduction in the number of event parameters which are needed.

---

1 Human Computer Interaction

Figure 14 below illustrates the design of the lowest-level[1] E-Scape score display. Data traces are presented vertically below each other, in a time-aligned fashion. Each event thus typically appears as a trace in *several* display windows, all these traces having the (same) duration of the event. In the figure, five event parameters are being displayed ('body waveform', 'fm waveform', 'warble', 'pitch' and 'nuance'). These parameters are amongst those available as inputs to the Instrument(s) in use by these events.

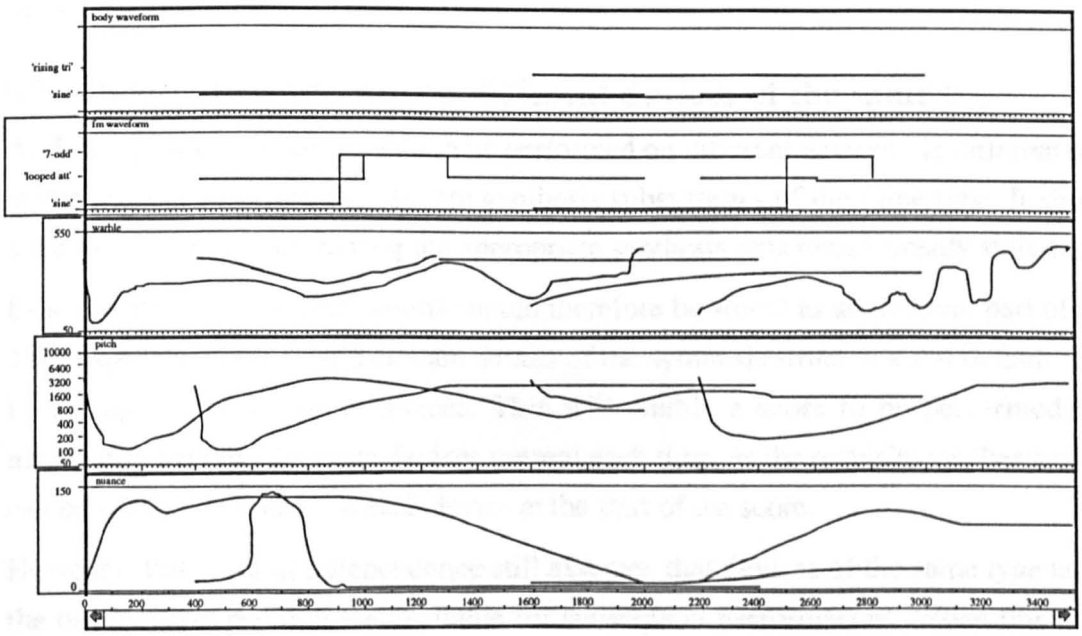


Fig. 14 Initial design for E-Scape score display

The display shows four score events, whose traces in each sub-window may overlap, as all are shown on the same scale. For example, the first event starts at time = 0 and ends at 2000ms, thereby overlapping with the second event which starts at 400ms. Another style of display can be envisaged where each event's traces are separated from the others for visual clarity, but no longer then are displayed on the same scale.

1 There is also an edit screen for a *single* score event which could be considered to be at a lower level than this.

6.7. Design goal 7: Support for device-independent scores

For the reasons discussed in 3.5.2, it is highly desirable for an event specification system to allow score to be *device-independent*. This will help to encourage the acquisition by compositions[1] of a historical 'solidity', by enabling their scores to be performed, edited and discussed by others in future years, rather than having to rely on a recorded realisation (performance), or the original devices still being available. There are two levels to such device independence:

6.7.1. Score performance on different devices of the same type

An E-Scape score should be able to be performed on different devices - ie different physical instances of a hardware or software synthesis subsystem - of the same type. It should not have to rely on a device having the appropriate synthesis structures[2] already stored in it.

E-Scape Instrument specifications should therefore be stored as an inherent part of a score. These specifications should contain details of the synthesis structures and default values to be set up in one or more devices. This will enable a score to be performed without necessarily needing the same devices present each time, as the requisite synthesis structures can be specified (loaded) to each device at the start of the score.

However, this level of independence still assumes that devices of the same *type* as used in the original composition are available for subsequent performances. When this is *not* the case a second level of independence is required.

6.7.2. Score performance on different *types* of device

E-Scape should attempt to facilitate identical performance of a score on *different types* of device, where possible. Such devices might be commercially available MIDI-controlled devices, or unit process-based hardware (eg MIDAS) or software (eg CSound).

An E-Scape user should be able to design Instruments which have the *same* scoring parameters as each other, yet which can create (via software processing) *different* low-level input values for synthesis modules in *different* types of device. Thus the same value for a parameter in a score should (within the limitations of the processes supported by a device - see below) give an equivalent sonic result.

A score written for performance on particular synthesis devices could then be performed on *different* devices. This would be done by designing new E-Scape Instruments which will have the *same* score parameter inputs, but which translate these into *different* low-level messages for synthesis structures on the new devices.

These synthesis structures and the translation would aim to replicate the sonic effect of the originals, but this does presume the device supports similar processes. For example, an

---

1 This supposes that the composition has been produced entirely by synthesis, rather than by any soundfile processing as described in 1.2.2.

2 For example, a MIDI-based synthesiser 'patch' or CSound 'instrument'.

original Instrument might use a 'resonant filter' unit on a device, and send messages to alter its cut-off frequency. Any newly-employed device must then also support processes which have the effect of a resonant filter (however it is actually implemented) and allow access to its cut-off frequency.

The newly-designed Instruments would then be substituted for corresponding original ones in the score - each event having its Instrument replaced with a new equivalent which has been designed to have the same parameters with the same mapping of parameter values to sonic effect. Thus there will then be no requirement to edit any of the score events - the design work is all embodied within the Instrument.

This facility should be provided automatically as a by-product of E-Scape's Instrument design.

6.8. Design goal 8: Transparent allocation of synthesis resources

A composer should not need to worry about the *allocation* on synthesis devices of the resources needed to perform a score. Such a facility will be especially useful when using devices (eg present MIDI-based synthesisers) which have fixed memory maps, and which require access to musical structures to be via fixed 'channel' allocations.

6.8.1. Knowledge-base of available device resources

An E-Scape user should be able to specify details of the structure and capabilities of the available connected devices. E-Scape should then to be able to access this information when deciding which synthesis resources (on these connected devices) to use for a score.

6.8.2. Analysis of resources required by score

E-Scape should automatically allocate these device resources (ie the creation or selection of synthesis structures, with their the id or location within the device) by analysing score requirements. E-Scape should decide what synthesis structures are needed in which devices at any particular time within the score. Such decisions are likely to involve knowledge of events' beginning and ending times, their parameter values, and other information about the synthesis structures employed by an event's Instrument (for example whether an input is 'polyphonic' - see 3.1.1.4).

6.8.3. Unequivocal score performance

E-Escape should able to perform (realise) a score in such a way that any shortfall in the required device resources can be made known to the composer. To do this, E-Scape will allocate device resources in *advance* of score performance. E-Scape will thus normally forego *interactive* performance[1] facilities - it will be primarily a *score* specification and playback system. However, this gains the advantage that any device inadequacy will be known to E-Scape in advance, allowing it to request appropriate action from the user.

However, a user should also be able to use E-Scape in a degraded mode of operation, where resources are *not* allocated in advance. This will enable 'interactive playing' - where pre-defined score events are triggered to play by real-time human performance control - as in MidiGrid for example (Hunt 1988).

A user will thus know overtly that a score will (or will not be) performed as specified. If not, the user will be able to decide the strategy for allocating the available resources. This will be far preferable to having to rely on the divers - often arbitrary or musically

---

[1] The term 'performance' is somewhat ambiguous. It is used in this thesis to mean the *realisation* of synthesised sound according to an score specification. The more conventionally understood meaning of the word is the real-time human manipulation of sounds via a musical instrument controller. This concept is denoted here by such terms as 'live performance', 'interactive performance' or 'real-time human performance'.

inappropriate - strategies used by devices, which have no knowledge of the score and the musical context of pre-empted events (Free 1986).

6.9. Design goal 9: Facilitation of algorithmic composition

As discussed in 6.2.2, any composition system, whatever the sophistication of its graphically-presented facilities, should *also* facilitate the generation of complex event or sonic structures via algorithms (programming style constructs) specified by the composer. (Pope 1986; Oppenheim 1991).

6.9.1 Algorithm operation

A composer will first create or select an algorithm, then specify operational parameters for it. Such algorithms can employ processes running at audio rate (ie creating sounds themselves), or a lower rate (ie creating perceivably distinct musical events), or both.

As a result, many events may be generated and/or sounds created. The parameters given to the algorithm may be considered to be contained within a separate set of 'score' data (eg CSound), or generated within the same system as the sound-creating algorithm (eg Fugue or CHANT). Some systems such as MIDAS or Kyma allow either approach to be used.

Such algorithms may:

- generate (and store) score events in software, which are then 'played' in real-time eg via MIDI. In this case the events are relatively crude control events and enjoy the same advantages and disadvantages of 'normal' MIDI-based systems.
- specify processes (eg copy a sound sample repeatedly and reverse it) which are to be carried out in the device - whether in hardware (eg Capybara - see 3.2.1.3), or by software signal processing primitives (eg CSound, Fugue). Either may necessitate off-line operation (writing, then playing back an intermediate soundfile) if processing resources are inadequate for real-time output.

However, in existing systems, the high-level algorithm specification system is tied to a particular device which the system is designed for - eg the CSound compiler (CSound), the sound Primitives (Fugue), or the ISPW (MAX).

6.9.2 Algorithm definition

Many existing systems provide some kind of user interface, which facilitates user specification or definition of algorithms in a variety of ways:

- by constructing a complex instrument definition, eg in CSound.
- by defining complex functions and a program which calls it call them in, eg in Fugue (3.2.1.7), Midiforth (Degazio 1988), CHANT (3.2.1.6), Formes (Rodet 1984; Boynton 1986a) or Pla (Schottstaedt 1983).
- by writing a high-level code script within a graphical system, such as the Quill editor (Oppenheim 1990) within the Dmix system (3.1.2.5).
- by graphically connecting a network of MAX-like graphic data processing icons, eg in MAX (3.2.1.1), Patchwork/ Esquisse (3.2.1.7), or Kyma (3.2.1.3).

The newer systems attempt to provide an easier to use graphical interface to the construction of algorithms and processes, but the power of textual specification via a language should not be underestimated, and the facility to use it should be provided, in addition to whatever other methods are used to control and specify algorithmic data generation. Even if a composer uses a graphical interface, he/she still needs to *understand* programming principles, in order to be able to construct algorithms.

"The musician who does not invest the energy to learn to program will never fully appreciate the possibilities of computers" (Roads 1985c).

Systems, such as Fugue, which generate algorithmic data at event *or* sound level do so to allow a composer to:

> "... express signal processing algorithms for sound synthesis, musical scores, and higher-level musical procedures all in one language" (Anderson 1991).

This is because composition systems preceding Fugue which have been used to control complex synthesis processes, have also been text based, hence Fugue aims to allow a composer only to have to learn one language, to enable event and synthesis data to be specified by the same functions.

However if score events or algorithms may be defined graphically by a user, then this uniformity may be less of an advantage - different user-interface mechanisms could be used in one system, as the learning process for a user is typically easier than for a text-based language. However, using a uniform interface for all types of algorithmic specification is still an elegant and satisfying feature.

E-Scape should support the specification and use of this kind of algorithmic event or data generation. This should be done in a *uniform* manner consistent with its existing structure, and retain E-Scape's ability to work with many different types of device. The algorithmic generation of *events* should thus be approached within E-Scape's proposed Instrument structure (see 6.1.4).

Thus, the definition and use of 'algorithmic Instruments' - a variant on the 'normal' Instrument - should be facilitated. An 'algorithmic Instrument' will create new score events using parameters from a single 'source' event to which it is assigned. Such a source event might, for example, supply the algorithmic Instrument with time, duration or other parameters to control aspects of the event generation process within it.

6.10 Conclusion

This chapter has introduced the design goals set for the E-Scape software system, which can be considered to be in two categories:

Some design goals have been formulated to illustrate the proposed standard features of computer music systems which are deemed necessary in order for them to be able to operate as sub-systems or modules within the system organisation proposed in chapter 4.

Other goals address the other problems faced by composer in using computer-based composition systems which were presented in chapter 3.

III - Implementation of the 'E-Scape' demonstrator software

This section (chapters 7-10) describes the implementation of the E-Scape software system whose design goals were described in chapter 6.

Chapter 7 first introduces the object-oriented concept, then describes the nature of object-oriented programming environments and their rationale for use in E-Scape implementation.

This prepares the way for chapter 8 which describes the structure of E-Scape, and chapter 9 which describes its functioning.

Finally in this section, chapter 10 presents the features of E-Scape which are available to a user.

7. The Object-Oriented Paradigm

This chapter has two main purposes:

Firstly (in sections 7.1 - 7.3), it provides the reader with a basic grounding in the nature of *object-oriented concepts*, and the features of object-oriented programming systems (OOPS). The suitability of object-oriented concepts for musical purposes is then discussed, along with some of the perceived benefits of using such systems for building musical applications. These sections will be of interest to the general reader, who is interested in the musical uses of object-oriented techniques and systems.

Secondly, (in sections 7.4 - 7.7) this chapter describes various aspects of the *use* of an OOPS for development of the E-Scape prototype application. This includes a discussion of the *selection* of the Smalltalk-80 system for software development, and its specific *features* and practical implementation on computer platforms. Those Smalltalk *language* elements are presented which are necessary for a full understanding of the explication of E-Scape's operation (in chapters 8 and 9). Finally, a brief overview of the nature of software design within an object-oriented programming system is described, with a simple musical example. These sections will be of interest primarily to the reader who wishes to understand the object-oriented software structure of E-Scape, and for software writers who would like to use or develop the E-Scape system.

7.1. Key concepts

What does object-oriented (OO) mean? It can be used as an adjective in many areas: for example, OO environments, OO applications, OO databases, OO specification, OO design (OOD) and OO analysis (Deutsch 1991). There are also OO programming (OOP) languages and OO programming systems (OOPS). What do they have in common? There are many conceptions of 'object-oriented', for example:

> (As a general *concept* , object-oriented means) "... abandoning the process-centric view of the software universe ... in favour of a product-centred paradigm" (Deutsch 1991).
>
> "An object has a set of operations and a local shared state that remembers the effect of operations" (Wegner 1989).
>
> "We perceive the world around us as made up of objects, and our brains arrange information into chunks. By using objects in a programming language, we can tap into an existing pattern of thought" (Kaehler 1986).

Object-oriented (OO) means the description or modelling of processes in terms of separable entities, which each have state and behaviour, and which can supply information or perform actions in response to messages sent within a defined interface. Such OO schemes usually follow the system or situation they are modelling very closely, thus making a

complex model more familiar (as it is more rooted in reality), and thus easier to understand and modify.

7.2. Object-oriented programming systems (OOPS)

OO programming techniques can be used within most computer languages and systems, but the level of *support* for OO concepts which is provided by such systems varies enormously.

The term 'object-oriented programming system' (OOPS) can be applied to a programming language or environment which facilitates the use of object-oriented techniques in a *natural* way when programming a software application (Thomas 1989b).

> "Object oriented techniques can and should be considered separate from object-oriented languages" (Duff 1990).

Thus OOPS simply support - to a greater or lesser extent - the concept of objects as a *language feature* (Wegner 1989). They facilitate the creation of software systems which model the world as a set of interacting entities, each possessing information and abilities, rather than as a set of procedures operating on data. Objects within the system are sent messages which invoke a response, which usually involves sending messages to other objects.

7.2.1. Features of OOPS

OOPS have the following features which support this kind of modelling:

7.2.1.1. Encapsulation

Encapsulation is the protecting of an object's private data from outside access, except via defined messages (Duff 1990). It also implies that each object has data *and* code (functionality) bundled with it. The object's internal data is hidden behind a set of procedures through which access to the data must take place (Tesler 1986). This makes for safe reuse of an object in a new situation, as its interface to the rest of the system is clearly defined via messages which may be sent to it..

> "From the outside, you can only ask an object to do something (send it a message)" (Robson 1981).

Thus, you cannot access the data within an object except via the approved messages (see 7.5.2.3). Certain behaviour (program code) is invoked within an object by a message being sent to it. If this code is altered, then this will *not* affect any other object which *sends* such a message.

7.2.1.2. Dynamic typing

The data type of a variable is usually *not* specified in an OOPS. Thus a variable can be declared, and subsequently have *any* kind of data (object) assigned to it. This incurs some run-time overhead, but gives valuable flexibility and reusability to a language (Pope 1992b).

7.2.1.3. Polymorphism

Polymorphism refers to the ability to have different kinds of objects respond to the same message in different ways (Duff 1990), using *dynamic binding*. Many different routines (ie 'function definitions') can be associated with a single *message*: the language itself then determines which routine to invoke, based on the type (class) of the object. Thus, each kind of object can invoke different procedures to respond to the same message.

Thus, a single message can be sent to an object, safe in the knowledge that it will respond in an appropriate manner, whatever kind of object it is (at worst, returning an 'I don't understand' error). For example a message (such as 'play') could be sent to a collection of objects without worrying what *kind* of object each one is.

7.2.1.4. Inheritance

Inheritance is the ability of an object to derive its data and functionality from another 'parent' object. A new class of objects can be created as the descendent (subclass) of an existing (parent) class (Duff 1990). A class inherits the methods and data structures (instance variables) of its parent, but can add additional variables or methods, and/or *override* old methods (Robson 1981). Thus code can be shared amongst similar objects.

The activity of creating new classes in this manner is termed 'subclassing'. Each new child class is known as a 'subclass', and the parent class from which it inherits behaviour and structure is called its 'superclass'.

7.2.2. Comparison of OOPS with Procedural Programming languages

The 'C' language is a good example of a flexible and capable procedural language which is designed to be medium-level, general purpose and efficient. 'C' can be used to solve almost any problem, even to implement the basics of OOPS (ie encapsulation, polymorphism and inheritance). However, to use a procedural language in this way would then be highly inconvenient, difficult to maintain, and require much 'syntactic baggage' and coding effort.

An OOPS simply maps the *form* of an OO implementation more accurately to its *functioning*, ie it operates naturally as an OO system, with direct support for OO features. It does this by moving most of the syntax and control logic code that implements OO features into the system kernel.

Dynamic typing incurs some run-time overhead, as compared with the static typing more usual in procedural languages, but has many benefits for languages which aim to support exploratory programming activities. However, OO features can often be supported *without* incurring too great an operation speed penalty. For example, it has been claimed that (using cache technology) Smalltalk systems can execute a message send in half the time of an equivalent 'C' procedure (Krasner 1983).

> "Object-oriented programming does not necessarily imply larger and slower programs" (Thomas 1989a).

An OOPS application, however, will typically contain many such message sends between objects, hence is unlikely to compete on raw speed with some other procedural languages. However, the relentless growth in computer processing speed makes such penalties increasingly less significant.

7.2.3. Benefits and costs of using OOPS

"OOP does have benefits but they are not free" (Duff 1990).

7.2.3.1 Benefits of OOPS

Programmers find that once learned, object oriented conceptualisation of a problem is a natural way to proceed, with the code then following suite. OOPS can potentially provide significant scope for achieving economy of effort with a new project. Code can be reused, with a programmer often able to get the required functionality by altering existing object classes, or more typically subclassing them. However it is still difficult to build easily *reusable* software (see 7.7).

> "Smalltalk [an example of an OOPS] is an enormously productive tool - once you learn it" (Nielsen 1989).

It is safe - a programmer can safely create subclasses, overriding or adding new behaviour, and can create (or reuse) complex data structure objects, but maintain a simple interface for using them within the OOPS environment.

> (OOPS) "... let programmers try out ideas and get immediate feedback, with little fear of causing damage" (Nielsen 1989).

OOPS add features that allow the efficient implementation of well-factored, minimally coupled systems. This is thus in step with many principles of good software design - for example Myer's Composite Structured Design promotes 'high module cohesion' (achieved by OOPS via inheritance) and 'low module coupling' (achieved by OOPS via encapsulation) as broad benchmarks for system quality (Duff 1990). High module cohesion is the organising of modules into families with shared characteristics. This is aided in OOPS by inheritance (see 7.2.1.4): subclasses of object can be created which inherit (and thus share) state and behaviour from a parent class. Low module coupling is the protection of a module's mode of operation and state from the effects of changes to other modules. This is exactly what is effected by encapsulation in OOPS (see 7.2.1.1.).

These characteristics improve the life and maintainability of a system for the same reasons they would improve a structured program system:

> "systems that localise information and logic are simply less complex" (Duff 1990).

To sum up, Thomas states:

> ".. to appreciate the benefits of OOPS, you will need to use a real object-oriented programming environment, such as Smalltalk, on an interesting application" (Thomas 1989b).

The author, having done just this with Smalltalk, can whole-heartedly agree.

7.2.3.2 Costs of OOPS

People need to employ different 'non procedural' thinking to apply the OOP paradigm in software programming. There is a different underlying programming model of modular and distributed design (using inheritance and polymorphism to build re-usable components), and new skills and techniques which must be learnt. This rethinking of conventional programming concepts in order to adapt to a more naturalistic model (see 7.2.3.1) explains why newcomers to programming often learn to use OOPS more easily than experienced functional programmers.

There are also syntactic differences compared with procedural languages, mainly involved in the sending of requests (messages) to objects, but these are less of a problem to an experienced programmer, who is used to several languages already.

It is easy to be initially overwhelmed by the large amount of material already present within an OOPS (or available in add-on packages of object classes).

> "Some programmers will see Smalltalk's rich selection of data structures as powerful; others will be overwhelmed" (Nielsen 1989).

The general problems of coping with new thinking are compounded in the Smalltalk-80 OOPS, where the object paradigm is used everywhere, and a very large amount of material is already available to the user in a rich programming environment. 1-2 MBytes of standard 'system' source code is provided, in addition to whatever other programmers have added to the system. There are a large number of object classes already available, each with a large number of messages.

> "The most time-consuming part of the OOPS learning curve involves learning about the class libraries" (Duff 1990).

A lot of time is needed to investigate which existing objects are present, and how they work and interrelate.

> "Hands-on instruction and intelligent code-browsing tools are the most effective way of ensuring that programmers become familiar with the existing class libraries" (Duff 1990).

The sheer variety of tools and the volume of code available can be daunting, and make initial learning slow:

> "Students were comparatively more apprehensive at the beginning because of the variety of tools built in, and the sheer volume of code provided by the Smalltalk library" (LaLonde 1990c).

A complex part of the system can easily involve many dozens of message calls backwards and forwards to different objects, some of which may be related as subclasses. Commenting on Smalltalk, Nielsen says:

> "Smalltalk is a large system, and like many large systems, it is fairly difficult for new users to penetrate. One of the most vexing problems is the distributed nature of the code" (Nielsen 1989).

Within the Smalltalk environment, code organisation is well supported by management and presentation tools. However, once some experience and confidence has been gained, a programmer realises that it is often *unnecessary* to know *how* an object performs some task; it genuinely can be safely treated as a 'black box', with a defined interface and function.

However, the advantages of using OOPS still have to be paid for at the outset by the negotiation of a considerable learning curve.

> "... in Smalltalk, as in other [OO] programming environments, code reuse is far from free" (Nielsen 1989).

7.3. Musical applications and benefits of the OOP paradigm

Many researchers in the field of computer music have examined the power of OO concepts and systems for musical usage.

> "It is significant that object-oriented programming matches most of the requirements of Music Composition and Synthesis" (Rodet 1984).
>
> "the power of OOPS for Music Composition and Synthesis systems is well known" (Lieberman 1982).

Rodet makes a thorough analysis of the requirements of music composition and synthesis systems for manipulating musical structures, and the advantages of OOPS for implementing them (Rodet 1984). He cites the inadequacy of 'ordinary' (ie non OO) programming languages for the modelling of musical processes in computer-based composition systems. Such modelling offers a particularly clear example of a "complexity barrier" - the musical processes involve so many possible 'actions', and data structures of arbitrary complexity and interrelation, that a composer is easily overwhelmed when trying to overview and manage such a plethora of structures and processes.

Rodet states that the goals of such models of musical processes are that they:-

- should follow common human presuppositions - ie behave and interrelate naturally as entities in the real world do.
- should try to be independent of a particular synthesis technique.
- should allow modularity and hierarchical construction, "as the complexity of models necessitates that they be built from sub models".

Lieberman suggests that the following properties of OOPS support the goals of a composition system's modelling of musical activity:-

- extensibility - a user can develop new control structures within an OOPS.
- interactivity - OOPS support a *dialogue* between user and machine which promotes ease of use and encourages investigative user activity.
- modularity - OOPS enable programmers to describe a thing in terms of elementary 'sub-things', and thus help a user to describe *complex* things.
- support of multiple representations of a concept - OOPS allow a structure or process to be used in a variety of contexts. (Lieberman 1982).

Each of the above goals is directly supported by the features of object-oriented environments and OOPS, which has made them a popular choice amongst computer music researchers (Scaletti 1989). Each OOPS feature provides directly relevant benefits to constructors of musical processes and structures in composition systems:

7.3.1. Musical benefits of *encapsulation*

A variety of musical objects (eg representing an ostinato, a chord, a note, a whole piece, an algorithmic process) can have their internal state and functionality protected from outside access, except via defined messages, each of which invokes behaviour (a 'method'). A single message such as 'transpose' will invoke a method which is different in its *execution* for different types of object (eg a chord, note or arpeggio), ie has a different effect on the internal state of the object.

However, a user of these objects cannot alter this state except by these defined methods. Similarly a user may decide to alter *how* a particular type of object implements a method, but a *user* of this object will not know any difference in terms of the message sent, which results in a safe programming or composing environment. For example, a composer might use a score processing object at a certain place within a higher-level compositional algorithm. This score processing object can later be altered in its behaviour, but will still perform acceptably in the places where it has been used.

7.3.2. Musical benefits of *inheritance*

Different types of object can be defined in terms of inheriting characteristics from a parent (or superclass). For example, an 'arpeggio' object shares many features with a 'chord' object; both can be loaded with note pitches, given a duration, asked to play etc. The arpeggio also has additional or different features and state (eg it can be asked to reverse itself, and be given a 'direction'). However, by defining the arpeggio as a subclass of the chord, only these *additional* or *different* features need to be defined (Krasner 1980). If the arpeggio object receives a message it does not understand (ie does not have a method of that name defined), it looks automatically to its parent superclass to see if *it* understands the message. This in turn would then look to its superclass, and so on. Only if the message is not understood by the highest, most general class of object (simply called 'Object' in Smalltalk) will an error ("don't understand") method be invoked.

7.3.3. Musical benefits of *polymorphism*

A message such as 'play' will invoke different behaviour in different types of object. Thus, the user can simply ask whatever objects are present to 'play' themselves, and each object knows how to do this (assuming a 'play' method has been defined); the user need not worry what types of object are present (Krasner 1980).

Data structures such as Collections or Sets can contain any kind of object, again providing immense freedom and flexibility to a programmer at any level. Thus, *sets* of musical objects can be manipulated with the same message, enabling them to be treated as a single entity (Orlarey 1986).

Many different kinds of musical object (with different internal representation and structure) can be created, which can then be used interchangeably, for example all can respond to a *play* message (Pope 1992b).

7.4. Selection of Smalltalk-80 for E-Scape implementation

This section describes the background and reasoning behind the eventual choice of Smalltalk-80 as the language and environment for implementation of the E-Scape software.

An OOPS was a pre-requisite, given the features presented above, and a range of OOPS were considered for use in the implementation of E-Scape. Several criteria were regarded as important:

• The choice of necessity had to be restricted to those OOPS which were actually *available* for use, which could run on the computer platforms available, and which had suitable interfacing capability to the outside world. MIDI as a standard musical interface was deemed to be vital as a minimum capability.

• The OOPS had to support fully functional GUI capability, and preferably to already include functionality suitable for musical purposes.

• The speed of operation was vital to support time-critical real-time music data output. Although time-stamped control of synthesis devices was planned for E-Scape, it was felt essential to also provide for real-time control of devices.

Languages considered were C++ (Stroustrup 1987), CLOS (SIGPLAN 1988), Objective-C (Cox 1986), Smalltalk (Goldberg 1983), Actor, Eiffel, Object Pascal (Schmucker 1986), plus a variety of music-oriented languages or environments such as Pla (Schottstaedt 1983) FORMES (Rodet 1984; Boynton 1986a), MIDI-LISP (Wessel 1987), and MAX (see 3.2.1.1).

All were found lacking in providing the necessary composite level of support for music structures, graphics and/or programming flexibility, or were practically unavailable for development purposes, with the exception of Smalltalk. The Kyma composition system (see 3.2.1.3) would have been a promising candidate (as it is based on Smalltalk-80), but in 1991 it only existed in a run-time version, with no access to the Smalltalk compiler.

Smalltalk is available in two major dialects; Smalltalk-80 from Xerox Parc Place (Xerox Learning Research Group 1981), and Smalltalk V from Digitalk (Tello 1987). They are compatible in some aspects (exceptions include the GUI and aspects of the compiler operation). Smalltalk-80 is the more fully developed and fully-featured system, and one of its features - the support of multi-tasking synchronised processes - is vital for music scheduling applications. Smalltalk is available across many platforms, ranging from IBM PCs and compatibles, to high-performance UNIX workstations. It is widely used in industry (Cook 1993) and is also well established in academic computer music research (Hebel 1987; Diener 1989; Mellinger 1989; Pope 1989; Pope 1992) and commercial music systems (Scaletti 1989).

The choice of Smalltalk-80 was confirmed when 'DMIX' (see 3.1.2.5) - a complex MIDI-based graphic compositional application - was demonstrated to the author, running in Smalltalk-80. This proved Smalltalk's ability to cope with real-time musical data processing

and output, and provided the possibility of re-using some of DMIX's object classes for E-Scape development (see 7.7).

The following section describes in more detail the features of Smalltalk-80 which have made it such an ideal tool for the E-Scape software implementation.

7.5. Smalltalk-80

Smalltalk is not only a language, but an entire standardised WIMP[1] environment with program development tools such as 'browsers' and 'debuggers' (Goldberg 1984), plus a large body of object classes which are supplied as standard. These, for example, implement various complex data structures (Althoff 1981), or provide GUI functionality (LaLonde 1990b).

All system functionality - including the compiler and window scheduling - is provided within the environment via objects which can be altered by a user. The corollary to this provision of access to all parts of the system operation is that a naive user can easily stop the whole environment working by an injudicious change to the method code of such low-level 'system' objects. Most basic functionality is available somewhere in the system already; the programmer often operates more as a reader, investigating and modifying existing code (see 7.2.3 above), rather than programming each new application from scratch.

Smalltalk-80 is a *pure* object-oriented system in that *everything* is an object. For example, the code:

'3 + 4.5' denotes sending the message '+' to the Number object '3' with the Float[2] object '4.5' as an argument.

Applications created in the Smalltalk-80 environment are *inherently* portable (Deutsch 1991). The same code runs identically on different platforms, including all graphic and user-interaction facilities. "The object-oriented nature of Smalltalk contributes a great deal to GUI portability, because GUI features are encapsulated as very high-level abstractions" (Andersen 1991), for example methods to interrogate the mouse state or draw lines on the screen.

The author has used Smalltalk-80 on NeXT, Apple Macintosh, Atari, and Sun 3 computers, all with the same GUI, with no code conversion required to transfer an application (including its GUI) from one platform to another.

> "Binaries are fully portable with the final stage of compilation to native machine code automatically occurring when the program is first executed" (Andersen 1991).

---

1 A graphic presentation of Windows, Icons, Menus and Pointer - see glossary.

2 Float is a class of objects which describe floating point numbers - see 7.5.2.

E-Scape development has been on version 2.3 on the Apple Macintosh, but converting to the completely new version 'Objectworks \ Smalltalk release 4' is envisaged in the next year. This has a more consistent graphics programming implementation, and extends cross-platform graphic portability to include colour of any depth. It also enables the top-level windows and menus to follow the look and feel of the platform it is on (Deutsch 1991). However, Smalltalk 'release 4' is considerably different to v2.3 in the structure and operation of its GUI supporting classes, and this, along with the expectation of conversion has resulted in a deliberate minimisation of GUI development in the E-Scape prototype implementation.

7.5.1. Implementation of the Smalltalk-80 OOPS

In order to operate almost identically across many platforms, the Smalltalk-80 environment consists of two components: (i) The 'virtual image' contains the compiler, graphical display system, run-time processing etc, as compiled Smalltalk code which is identical for each machine. (ii) The 'virtual machine' is a small application which is machine dependent, linking the operating system and hardware of a particular computer platform to the virtual image code.

The overheads of performing the message passing between objects are surprisingly low, and the speed reduction which might be imagined from the fact that dynamic typing is performed at run time is actually quite small, as

> "Parc Place Smalltalk-80 performs dynamic compilation to native machine code" (Thomas 1989a)

and in practice there is sufficient speed for the real-time response appropriate to a music performance application.

7.5.1.1 Implementation of Smalltalk-80 on the Atari ST

Smalltalk-80 v2.3 is available and usable on the Atari ST computer, but is limited to using 4MB of RAM, which is only just large enough to contain the Smalltalk system and DMIX object classes. There is also almost no memory capacity available for storage of music data, thus making the Atari almost impossible to use effectively as a platform for E-Scape development. Another difficulty on this platform is the highly convoluted method of providing MIDI access via Smalltalk 'primitive methods' which link to hardware-level functions. Finally, with an 8MHz 68000 processor, the speed of operation was inadequate to enable sufficiently fast drawing of graphic score displays.

7.5.1.2 Implementation of Smalltalk-80 on the Apple Macintosh

Smalltalk v2.3 was used on the Apple Macintosh for E-Scape development, as the Dmix system, some of whose classes were reused, is implemented for this version. This implementation comes with a set of low-level event-scheduling routines (written in 'C') accessed via primitive Smalltalk functions.

Unfortunately, this version is relatively old, and ParcPlace were unable to provide any updating facility, and the low-level facilities providing access to machine ports were unable to be extended beyond what was originally supplied. There is only limited access to the physical Macintosh ports, the main restriction being that the MIDI output from the Macintosh can only use its 'Modem' serial port. Not only does this neglect the opportunity of using the other 'printer' serial port (to allow an additional 16 MIDI channels to be used), but means that the conventional use of this port as an RS232 terminal (used for example to connect to the MIDAS system) is impossible at the same time. Even worse, once the physical 'modem' port has been used for RS232 output, it is rendered inoperative for MIDI output - without 'reinitialising' it. Unfortunately, there are no facilities (methods) within the Smalltalk system for doing this - the code to initialise the 'modem' port for MIDI operation is built in to the Virtual Machine, and only operates when this is first launched.

In order to demonstrate E-Scape's capability for integrated control of different devices using different communication links, E-Scape Instruments have been designed which use and address structures in several different synthesis devices using both MIDI *and* RS232 protocols. Unfortunately, due to the above restrictions, these Instruments cannot be demonstrated using both protocols simultaneously. For demonstration purposes this failing has been partially obviated by using a graphic window to display the RS232 output codes within E-Scape, which are not then actually sent to the physical 'Modem' port. Conversely, the port can be used for the RS232 protocol, and the *MIDI* messages then monitored but not sent, in a similar manner. E-Scape can thus be demonstrated to facilitate the use of multiple ports, and will be able to physically do so when converted to the new Smalltalk 'release 4' version which has full support for all ports.

7.5.2. Smalltalk-80 language features

The following is not designed in any way to be a Smalltalk tutorial, as this is beyond the scope of this thesis. The aim is to introduce enough of the concepts and terms used in Smalltalk-80 to enable a reader to follow the succeeding material. For a tutorial introduction to Smalltalk, the reader is referred to (Kaehler 1986; Kaehler 1986) and (LaLonde 1990b; LaLonde 1990a). For a more formal presentation of Smalltalk-80 syntax and definitions refer to (Xerox Learning Research Group 1981).

Smalltalk is a *pure* OOPS in that *everything* is an object, for example the graphical display surface, the semaphores and processes, the arithmetic operators, the class definitions, and numbers are all objects and can be treated consistently.

7.5.2.1. Smalltalk objects and classes

In Smalltalk, objects are created according to a class definition. Class names are capitalised by convention, while instances of that class are generally denoted by an article preceding the class name. So for example, the *class* Point can have *instances* which will be referred to as 'a Point' or 'the Point'. Thus 'a Point' is shorthand for 'an instantiated object of class Point'.

A Smalltalk *class* acts as a *template* which describes the structure that objects of that class will have when instantiated. The class definition includes *methods* (equivalent to functions in procedural programming) which all objects (instances) of the class will then understand.

Objects of a particular Smalltalk *class* (eg DeviceType) are referred to in this thesis by using the class name. Such phrases as 'DeviceTypes', 'the Devices', or 'a Device' all denote *object(s)* of the class DeviceType rather than the class itself. Overt mention of object will often be made within the text to emphasise this (eg 'the DeviceType object') where this will not cause unnecessary obfuscation.

7.5.2.2. Smalltalk *instance variables*

An object's structure also consists of named *instance variables*. In subsequent text within this thesis, instance variables will be printed in italics.

Objects once instantiated can have their instance variables assigned to be *any* other object. (ie the data type of variables is only determined at run time).

Several instance variables can be assigned to (ie 'have' or own) the same object. They effectively only have a pointer to the object, but this fact is hidden from the programmer.

7.5.2.3 Smalltalk messages and methods

The only outside access to an object's instance variables is via named messages which invoke a method[1]. Messages can have arguments which can also be any other object, for example a Collection (containing other objects), a Number, or a Block object (which contains code with temporary variables).

Messages are sent to an object which is called the *receiver* of the message. The receiver object then carries out the method invoked by the message, using any arguments which are passed to it.

7.5.2.4. Subclassing

Classes are organised in hierarchies. A new class can be defined to be a *subclass* of another (which will then be the *superclass* of the new class). The subclass immediately has the instance variable of its superclass, and understands all the same messages, and is said to have *inherited* structure and behaviour from it. For example, in the object oriented design example in the next section, the Square and Triangle classes are defined as subclasses of the abstract class Polygon. An abstract class is one which is simply there to provide structure common to several subclasses. It is not designed to be instantiated itself, only to have subclasses defined from it .

The programmer is then able to modify selected methods for the new class, so that it behaves in a different way to its superclass when sent the same message. New methods can

---

1 An object *can* access its own instance variables directly, but the approved style is for an object to have a separate *method* to access each of its variables (either setting it or retrieving an object from it).

also be added, as can new instance variables. Such adaptation of an existing class to create a new subclass with similar but different structure and behaviour is called 'subclassing' and is a powerful feature of OOPS.

7.5.2.5. Smalltalk system object classes

There are a *very* large number of Smalltalk object classes which are supplied as standard in the system. The following are referred to in chapters 8 and 9, and are here briefly described:

A Dictionary is a set of object pairs[1] which are linked and cross referenced. The objects in each pair are designated as a 'key' and a 'value'. 'Key' objects are typically Strings or symbols, which effectively name the 'value' object. The Association (key and value pair) is written: '(key object => value object)', and the Dictionary is written as a list of these pairs

Collections and Sets are objects which can grow to accommodate the storage of an arbitrary number of objects. The subclass SortedCollection[2] can *sort* its objects by a user specifiable comparison (using for example a specified parameter of the objects). SortedCollections are used to store child events in E-Scape and Dmix, with the objects actually stored being time and event pairings (stored in an Association object), which are sorted on ascending time order. New time=>event Associations can be added to the SortedCollection object subsequently, and will be automatically sorted by it with no further intervention from the programmer.

---

1 The paired group is itself actually yet another kind of object - an Association.

2 A subclass of the Collection class.

7.6. Object-oriented design

A full discussion of the design techniques used in creating object-oriented systems would form a computer science thesis in itself, and this section merely attempts to give an overview of the kinds of activity which are undertaken when translating a design specification into an application within an OOPS. An initial design for a collection of interacting objects is first formulated, which can then be expanded if necessary during testing of the system. As described in 7.2, in a well-structured design, features and structure can be easily and safely added to objects without compromising the functioning of the system.

7.6.1. Initial design formulation

The process of determining the initial design employs the following steps:-

7.6.1.1. Determine which objects are needed

Object-oriented design typically starts by deriving the objects which are needed in order to model the desired system functionality (Tesler 1986), ie:

> "...by *finding the objects*; ie attempting to determine the object classes and their representation" (LaLonde 1990c).

Thus, object classes and the messages they should send each other are arrived at by examining the required system operation. After determining the features desired in a system, the designer must conceive which objects are needed to model these features (Beck 1992). The system which is being created should be structured as a set of objects which could interrelate in a natural way in the real world.

For example, a music system might have a Score object which contains Note objects. Each Note could have an Instrument object, a Pitch object, and a start time (Number). To play a piece, the Score object could ask each of its Note objects to play. Each Note would then ask its Instrument to play at the appropriate start time. The Instrument would respond to this request by asking the Note for its Pitch object. The Instrument could then ask this Pitch object for its value, then use this value to create the sound in order to play this Note.

Thus, in this simplified example, a system's behaviour has been modelled in terms of independent objects which communicate with each other. A successful such model will then translate easily and naturally into software objects within an OOPS.

> "You design and implement an object-oriented system as a simulation that assigns state and behaviour to each of the natural objects in the application" (Thomas 1989b).

> "Software objects *are* like real-world objects" (Wirfs-Brock 1992).

For example, a (highly simplified!) music package might initially be determined to require objects of classes 'Chord', 'Arpeggio', and 'Ostinato'.

7.6.1.2 Determine the objects' structure

The designer then needs to decide for each kind of object what their variables are, and what other objects will be assigned to them, ie which objects will be owned by which.

In this example, all the objects have *notes*, and each note has a *pitch* and *duration*. In addition the Arpeggio object has a direction (up or down) in which to play its notes in order. The Ostinato has no direction but instead has a number of times it is to repeat its notes. These features become the *instance variables* of the classes, and are shown in figure 15 below. When an *instance* of the class is created - ie when an object is created (instantiated) according to the class definition - each instance variable will need to be assigned to an appropriate object.

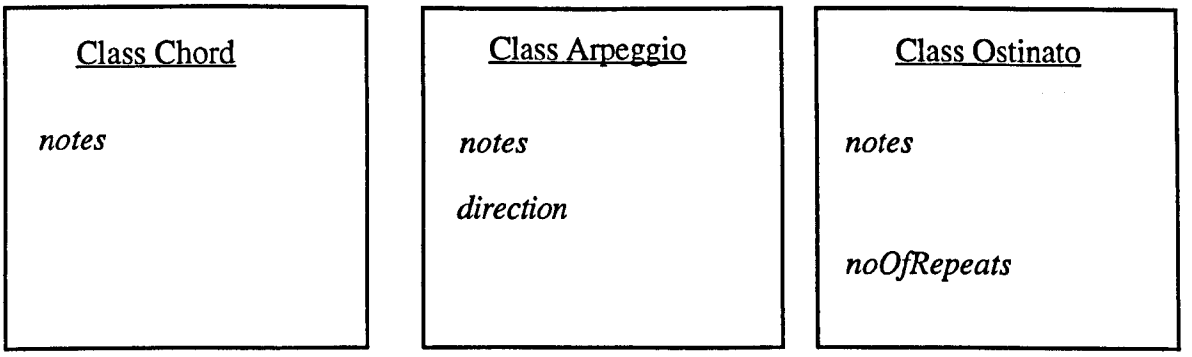


Fig. 15 Class definitions for three kinds of musical object

Note that in Smalltalk, dynamic typing means that *any* object can be assigned to an instance variable. The programmer must make it clear to a user of the object (often him/herself) what *kind* of object is expected, by using an appropriate name for the variable or argument.

An example using Smalltalk syntax will clarify this. In figure 16 below, a new object of class Arpeggio has been instantiated (by an appropriate method), and its instance variables assigned.

Its *direction* instance variable is assigned to the String object 'up'.

Its *notes* variable is assigned to be a Collection object: a class of object (supplied by Smalltalk as standard) which can contain and organise other objects. The Collection object is then filled with four Note objects. Each Note in turn has two instance variables: *pitch* and *duration*, both of which are assigned to Integer[1] objects.

---

[1] An Integer is a simple Smalltalk object which has an integral numerical value, and knows how to perform various arithmetical operations.

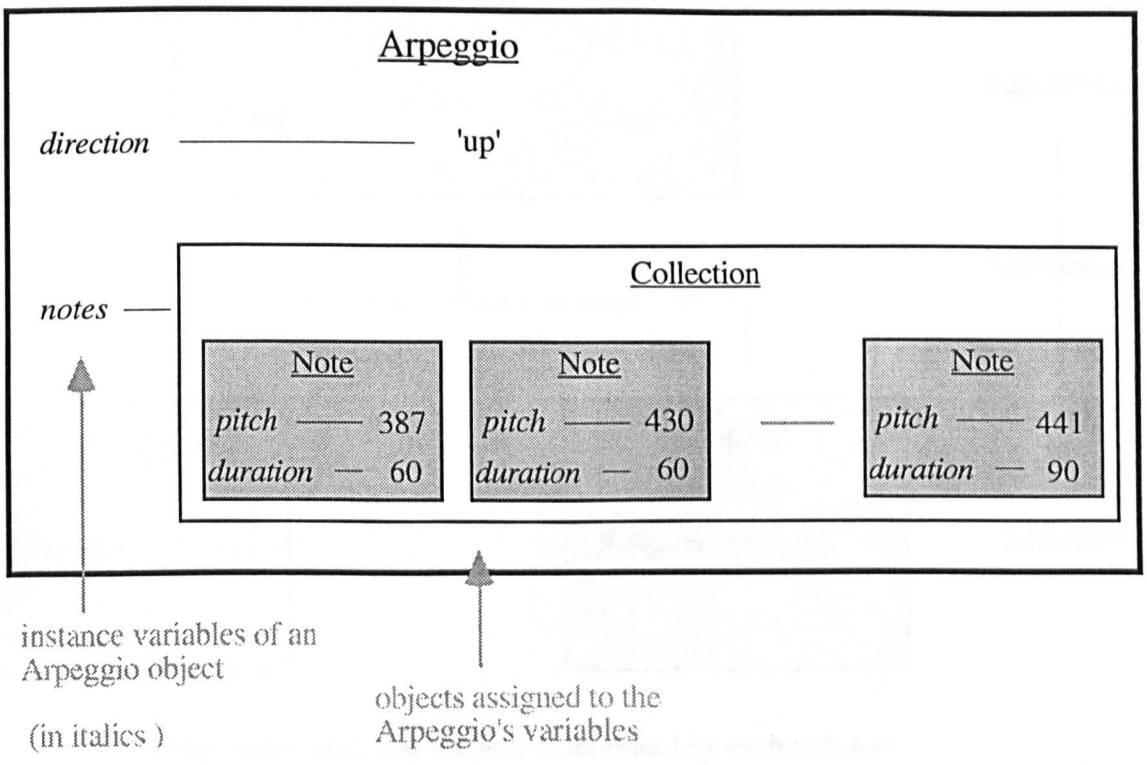


Fig. 16 An instantiated object of class Arpeggio

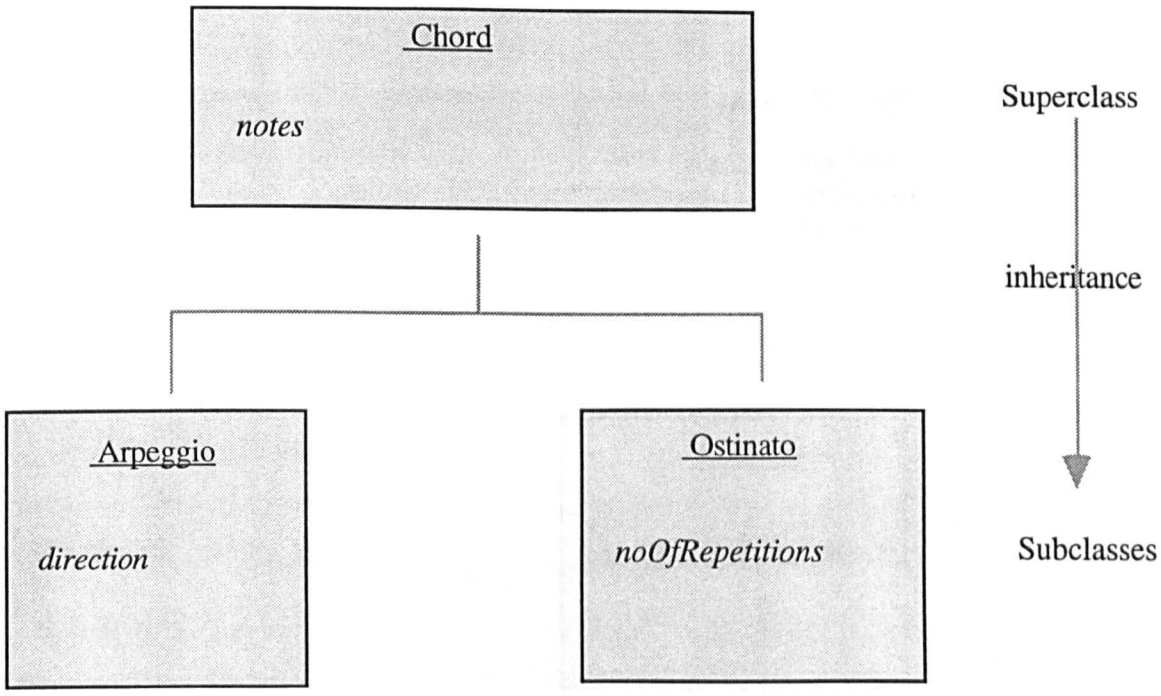
7.6.1.3. Specify the messages for each object

The designer must then determine what each kind of object can be asked to do, and the message which will sent to each object in order to request this.

In the above example, all the objects can be asked, for example, to return their collection of Notes, and to report their duration. The messages to do this might be 'getNotes' and 'getDuration'. Other messages can send data to the object (eg 'add a new Note'), ask it to change itself in some way (eg 'invert yourself'), or perform some action (eg 'play').

7.6.1.4. Organise objects into class hierarchy

The next step is to structure the object classes derived so far into *hierarchies*, by subclassing each class from another - possibly a system class (see 7.5.2.4). To do this, the designer must determine what *commonality* exists between classes. In the example above, all the shape objects have a *notes* variable, for example, thus it makes sense to "factor out the commonality" (LaLonde 1990c) using the Chord class as the superclass of the other two. This is illustrated in figure 17 below. The Chord superclass will now be given the *notes* variable, and the methods to access it; only the *different* state and behaviour then needs to be implemented in each subclass.



The *notes* instance variable is inherited by each subclass

Fig. 17 Inheritance of common state by subclassing

7.6.1.5 Implement a method for each message

The final step is to implement a method (ie code which defines actions) which each object will perform in response to a message. This will usually involve yet further interaction with other objects. In the example, each object will respond to the 'getNotes' message by implementing the method defined in the Chord superclass. This simply returns the value of the *notes* variable - ie the collection of Note objects, as illustrated in figure 18 below. The ^ symbol indicates to 'return' a value.

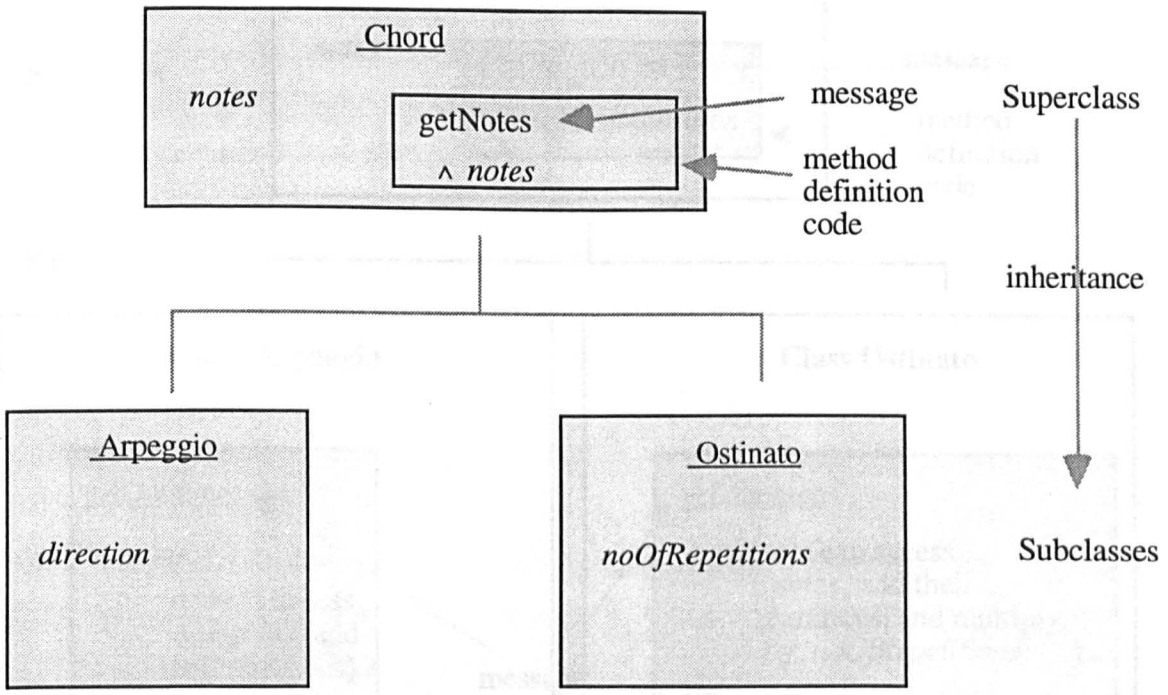


Fig. 18 Inheritance of common functionality by subclassing

However, in this example, a message asking for an object's *duration* is implemented *differently* for each class, ie the methods used by each object to respond to the 'getDuration' message may *differ*, as shown in figure 19 below. An object of class Chord will respond by asking one[1] of its Note objects for *its duration* (instance variable). The first Note is returned by sending the message 'first' to the *notes* Collection. This Note is then sent the message 'duration'. Smalltalk messages can be cascaded, hence the code for this method is

```
^ notes first duration.
```

On the other hand, an Arpeggio object will need to add the *duration* of *each* of its Notes to determine its total duration. Thus, the Arpeggio class will have a *different* 'getDuration' method, which will override that of the Chord superclass. The Ostinato class will also have a different method, needing to take into account the *noOfRepetitions* of its notes.

---

[1] All the Notes in the Chord should have the *same* duration, so *which* Note is asked is immaterial.

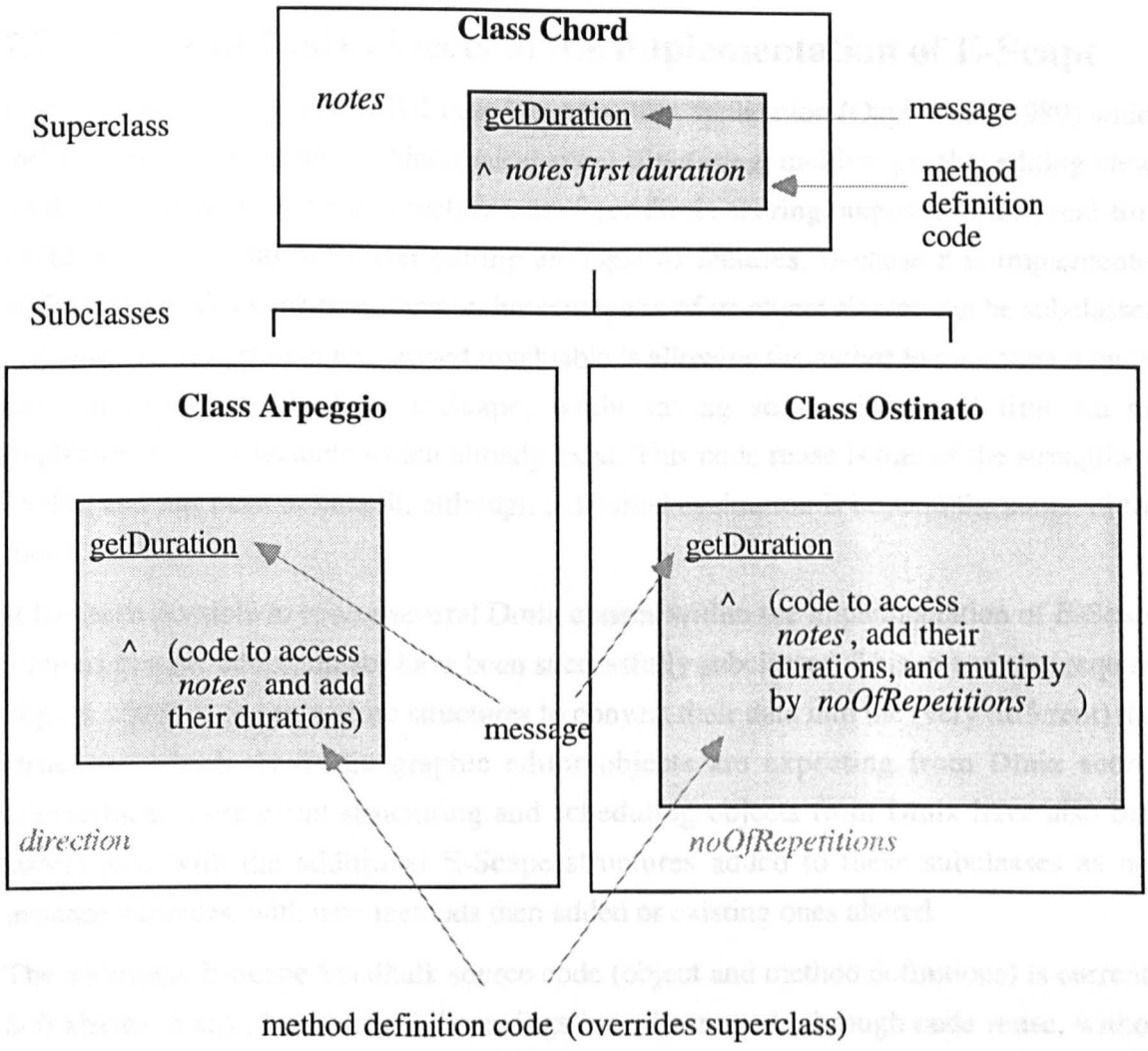


Fig. 19 A new method in a subclass *overrides* the method of the superclass

This ability to reuse the methods and behaviour of existing object classes is very powerful and useful when building complex systems.

7.6.2. Incremental refinement and re-implementation

Once the initial design has been created, the above processes are carried out incrementally throughout development. The need for new object classes may only become apparent during implementation, and these, along with new methods can be safely added. Existing classes can be safely modified, renamed, split or reorganised.

> "The best use of inheritance for code sharing is often not apparent until far into the design" (Beck 1992).

7.7 Reuse of Dmix objects in the implementation of E-Scape

Dmix is a fully functioning MIDI-based composition application (Oppenheim 1989) which includes 'sequencer' features, hierarchical event structuring, multiple graphic editing views on different aspects of event structures, an algorithmic scoring language editor, real-time event processing and parameter editing amongst its features. Because it is implemented within the Smalltalk-80 environment, however, any of its object classes can be subclassed, and adapted. This system has proved invaluable is allowing the author to concentrate on the new structures required by E-Scape, while saving some effort and time on the implementation of features which already exist. This code reuse is one of the strengths of OOPS, and has been of benefit, although a detailed evaluation is beyond the scope of this thesis.

It has been possible to re-use several Dmix classes within the implementation of E-Scape. Various graphic editor classes have been successfully subclassed. This re-use also required objects within E-Scape's score structures to convert their data into the (very different) data structures which the Dmix graphic editor objects are expecting from Dmix scores. Hierarchical score event structuring and scheduling objects from Dmix have also been subclassed, with the additional E-Scape structures added to these subclasses as new instance variables, with new methods then added or existing ones altered.

The additional E-Scape Smalltalk source code (object and method definitions) is currently 840 kbytes in size, but worthwhile savings have been made through code reuse, without which the new code size would be significantly larger than this. The programming effort saved in the long term has to be partially offset against the time needed to initially investigate and understand the existing Dmix objects, as well as the system classes.

The expected costs of OO code re-use detailed above in 7.2.3 have been encountered by the author, with a long learning curve due to the large number of object classes - both in Dmix and in the standard Smalltalk 'system' classes - only a fraction of which are used for subclassing and re-use.

7.8 Conclusion

This chapter has introduced the concepts behind object-oriented systems. It has then discussed the use of object-oriented programming for software development, and its benefits for musical applications.

The terminology used in object-oriented programming has been described, with particular reference to the Smalltalk-80 language. This background will be assumed in the following three chapters which describe the structure and operation of the object-oriented E-Scape software.

8. The Object Oriented structure of E-Scape

This chapter describes the features and structure of the 'E-Scape' prototype software application. E-Scape acts in the role of an 'event specification' subsystem within the envisaged scenario presented in chapter 4, and aims to implement the design goals for such systems outlined in chapter 6.

The introduction (8.1) gives a general overview of the functionality of E-Scape. The following section (8.2) then presents a summary of the Smalltalk objects which are required in order to implement this functionality.

The bulk of this chapter (8.3-8.10) presents the details of the objects and their inter-relations which comprise the E-Scape application. These sections, along with those in chapter 9, may be omitted by a reader who does not need to understand the detailed programming structure and implementation of E-Scape.

The conclusion (8.11) then gives a summary of E-Scape's structure. An overview of E-Scape's structure can thus gained from reading sections 8.1, 8.2 and 8.11.

8.1 Introduction to E-Scape

E-Scape is built from various kinds of interrelating object.

A score is created out of score events which each have a designated Instrument object. An Instrument has two main functions:

(i) An Instrument defines descriptions of synthesis structures in one or more types of device[1]. Descriptions of such synthesis structures can be defined by the user, by specifying and connecting lower-level objects which describe synthesis units. These structure descriptions are stored within an object which describes a particular type of device, along with other characteristics of that device type. Instruments can then be constructed which incorporate such structures for one (or *more*) types of device.

(ii) The Instrument also specifies the scoring parameters which will be available to the user. The user can then specify values for these parameters for each score event.

When a score event is inserted into a score, the resources necessary to perform it are allocated by E-Scape in synthesis devices connected to it. The Instrument assigned to the score event defines synthesis structures which can exist on particular types of device. Each of these structure definitions is then instantiated in a device of the appropriate type. This is achieved using E-Scape's knowledge of the configuration and capabilities of each connected device, and the specification of the synthesis units which can be supported by it.

E-Scape can then formulate messages - using this knowledge and the specification of each unit. Such message can be sent to each device in order to *effect* the instantiation of these synthesis units, if necessary[2] in particular locations within the device. Other messages can be sent to start and stop units at the correct time so as to perform the score event. E-Scape can then send input parameter values - which are derived from the scoring parameters - to these units.

The Instrument thus also contains an interface layer which converts each scoring parameter into one or more input values to be sent to units in a device.

8.2 Summary of E-Scape's structure - a chapter overview

This section overviews the main structure of E-Scape presented in this chapter, summarising the content of the following sections 8.3-8.10, and briefly describing each object and how it relates to others.

---

1 The word 'device' (lower case) will be used throughout to denote a hardware or software engine running one or more synthesis processes. Such a 'device' must be distinguished from a 'Device' (capitalised) which is a software object (an instance of the Device class) within E-Scape which *describes* the structure and functionality of a device.

2 If the device has a fixed memory map, with locations where units can exist.

Figure 20, below, summarises these E-Scape objects and their interrelations, and may also usefully be referred to throughout the rest of this chapter. The arrows indicate a variety of relationships, typically 'is derived from', 'is assigned to'. Dots (. . .) next to an object indicate that more than one such object will typically be present.

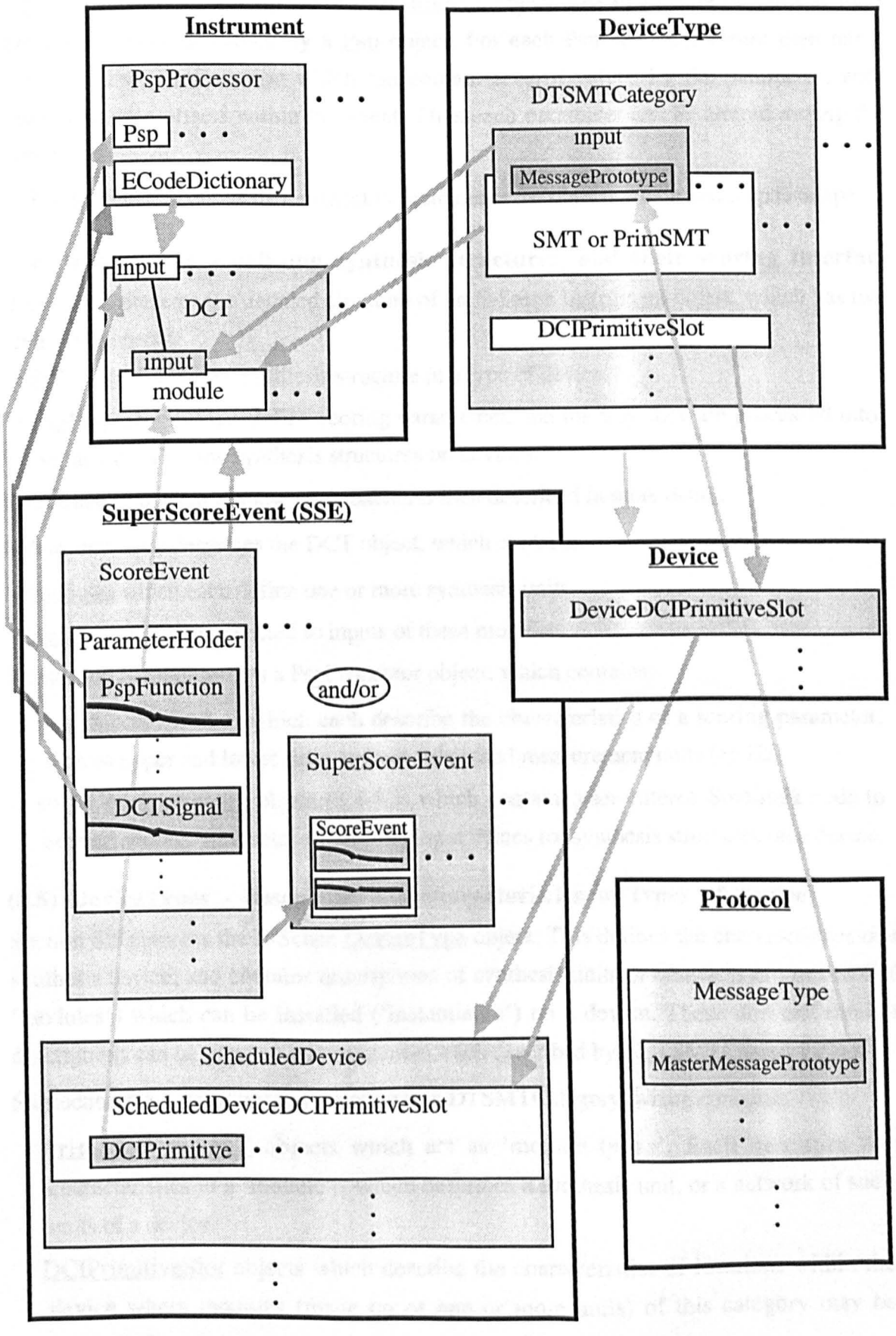


Fig. 20 Overview of the main E-Scape objects

Other co-ordinating overview diagrams are presented at the end of chapter 9.

(8.3) ScoreEvents and Instruments - a composer's view

Section 8.3 presents the top-level functionality of E-Scape as seen by a composer, and the objects which perform this:

A ScoreEvent object is assigned to an Instrument object. The Instrument presents scoring parameters, each described by a Psp object. For each Psp, the ScoreEvent then has a corresponding PspFunction which can contain *several* values for the parameter, with associated time offsets within the event. Thus each parameter can be altered *during* the ScoreEvent.

The following sections then present in more detail the objects which make up E-Scape.

(8.4) Instruments - defining synthesis structures, and their scoring interface

Section 8.4 presents the detailed structure of an E-Scape Instrument object, which has two main components:

- DCT objects define a synthesis structure in a type of device.
- PspProcessor objects define scoring parameters, and the way they are processed into input values for the synthesis structures on devices.

The structure of each of these components is then described in more detail.

Subsection 8.4.3 describes the DCT object, which contains:

- modules which each define one or more synthesis units.
- inputs which are connected to inputs of these modules.

Subsection 8.4.4 describes a PspProcessor object, which contains:

- Psp objects (8.4.4.1) which each describe the characteristics of a scoring parameter, such as upper and lower limit, default value, and measurement units (eg Hz).
- an ECodeDictionary object (8.4.4.2) which contains user-entered Smalltalk code to convert scoring parameter values into input values for synthesis structures on a device.

(8.5) DeviceTypes - describing the characteristics of types of device

Section 8.5 presents the E-Scape DeviceType object. This defines the characteristics of a synthesis device, and contains descriptions of synthesis units or *networks* of units (called 'modules') which can be installed ('instantiated') on a device. These unit and module descriptions can be grouped into categories, each described by a DTSMTCategory object.

Subsection 8.5.3 describes the structure of a DTSMTCategory, which contains:

- PrimSMT or SMT objects which act as 'module *types*'. Each describes the characteristics of a 'module' - which describes a synthesis unit, or a network of such units of a device.
- DCIPrimitiveSlot objects which describe the characteristics of locations within the device where modules (made up of one or more units) of this category may be installed. This is only needed if the device has a fixed memory map.

(8.6) Module types (PrimSMTs and SMTs) - describing synthesis modules

Section 8.6 described how module types are *used* as templates to create the modules (one or more units) which make up a DCT (see 8.4.3).

Subsection 8.6.1 describes how module types are constructed and defined by the user. A module type can be:

- defined directly, with the characteristics of its inputs specified, along with other parameters.
- built by assembling existing modules into a higher-level structure

This data will be subsequently be used by E-Scape to construct messages which can be sent to the device. Such messages may, for instance, instantiate a module (described by the module type), or send data to its inputs.

(8.7) Devices - describing the characteristics of particular devices

Section 8.7 describes Device objects, which are created from a 'template' DeviceType object. Each Device defines the characteristics of a particular synthesis device, as well as taking general information about the structure of the type of device from the 'parent' DeviceType.

A Device contains DeviceDCIPrimitiveSlots, which are derived from corresponding DCIPrimitiveSlots in the DeviceType (see 8.5.3). Each describes the particular characteristics (such as a 'channel' setting) of a location within a device within which synthesis units can be installed.

(8.8) Protocols and MessageTypes - describing the format of messages

Section 8.8 describes objects which are used to construct the actual message which will be sent from E-Scape to a device in order to communicate commands (eg to instantiate a unit), or data (typically to the *inputs* of a synthesis unit or structure) to it.

MessageTypes describe the structure of a message in terms of its fields, and how the data for those fields is derived or supplied.

Each MessageType can have one or more MessagePrototypes defined which each describe a usage of the MessageType with E-Scape. A MessagePrototype describes how, and from where, each field is supplied with data. It is assigned to one or more actual objects within the E-Scape score and Instrument structure. The characteristics of each MessagePrototype are defined in a corresponding abstract MasterMessagePrototype, which is stored in the MessageType.

MessageTypes which are associated by usage are contained within a Protocol, which also defines the computer ports from which a message - once formulated from a MessagePrototype - can be sent.

Each port may have additional manipulations of message data specified (such as inserting extra characters between fields) so as to conform to its particular requirements.

(8.9) ScoreEvents - describing event structures within a score

Section 8.9 describes the structure of a ScoreEvent. A ScoreEvent has an associated Instrument which contains PspProcessors, which in turn possess Psps (see 8.4.4.1).

The ScoreEvent's parameter data is contained within PspFunctions, which are stored within ParameterHolders. Each ParameterHolder matches a PspProcessor in the Instrument, and each PspFunction matches a Psp.

Parameter data will then be sent from each PspFunction (via its associated Psp) to a PspProcessor in the Instrument. This has an ECodeDictionary (see 8.4.4.2) which performs various operations on the data to create lower-level data which is assigned to inputs of a DCT within the Instrument (see 8.4.3). The low-level data itself is stored within a DCTSignal which is associate with each DCT input. The DCTSignals are also stored within the ParameterHolder in the ScoreEvent (see figure 55).

(8.10) SuperScoreEvents (SSEs) - describing a score structure, and the allocation of device resources to perform it

Section 8.10 describes the structure of an SSE object, which contains ScoreEvents, each with a time offset within the SSE. The SSE also owns ScheduledDevice objects which represent a *usage* of a device by the synthesis structures which are defined (by a DCT) within the Instrument of each ScoreEvent.

Each ScheduledDevice corresponds with an available Device within E-Scape, which represents connected synthesis devices.

A ScheduledDevice has ScheduledDCIPrimitiveSlots which represent locations within which resources can be allocated at various times by the SSE; each derives information from the corresponding DeviceDCIPrimitiveSlot of this Device (see 8.7). Within each ScheduledDCIPrimitiveSlot, the SSE can install DCIPrimitives, each of which represents an allocation of a unit in a device for a particular time for a particular ScoreEvent. Each DCIPrimitive corresponds with a 'device-level' module within a DCT of the Instrument used by this ScoreEvent.

8.3. A composer's view of E-Scape

The following section describes the structure of E-Scape as seen by a high-level *composer* user - someone who is interested primarily in constructing scores. However, the ability to design or edit instrument structures, device descriptions, or even define new communication protocols is still open to any user.

At the bottom level, scores are constructed out of atomic score events. A score event has a duration and is not broken down further into discrete events in the score. This is not to say that a score event might not be heard as many ('granular') sonic events, but this would be a product of processes defined within its instrument.

Each score event is described by a ScoreEvent[1] object, which is created by specifying an Instrument object and duration for it. An Instrument object has three main functions, which are reflected in its structure:

(i) to provide a definition of the scoring parameters which a composer may specify for the Instrument;

(ii) to provide a specification of the synthesis structures to be used on one or more devices.

(iii) to provide processing algorithms which translate these score parameters into 'device-level' input parameters for synthesis processes.

The ScoreEvent can then be placed within a higher-level score structure defined by a SuperScoreEvent object. A SuperScoreEvent can also be treated as an event and itself be nested within a *higher-level* SuperScoreEvent. As has been described above in section 3.4.4, this capability is widely seen as highly desirable, and has been found to be prevalent in many other systems. As this aspect of E-Scape is not then unique, development effort has therefore been focused first on its other aspects which are more innovative. Hence, this hierarchical score structuring has not yet been implemented in the present state of E-Scape prototype development. The software structures and design are present in E-Scape, however, to enable a SuperScoreEvent to be loaded into a *higher*-level SuperScoreEvent, or to be created by grouping existing ScoreEvents.

An Instrument's only interface to a composer who wishes to create a score is via its input parameters. These may have such immediately obvious names as 'volume', 'pitch', 'detune', 'fm index', 'position' etc, or more obscurely named parameters which reflect the Instrument designer's perception of their audible effect. A composer would need to become familiar with the effect of changing these parameters, much as he/she might need to 'play with' and investigate a newly-discovered acoustic instrument to find out what it might be capable of. Examples of such parameters might be 'chaos', density', 'weight', 'flutter', or 'rotation'.

---

[1] In the following, all capitalised words indicate a class of Smalltalk object, and to further clarify object names, they are underlined on first appearance.

All input parameters are described by a named Psp[1] object which specifies the measurement *unit* (such as dB, Hz) in use, the maximum and minimum values allowed, and a default value which will be used if no user values are specified in the score. An example Instrument is shown in figure 21 below, with three Psp parameters: 'pitch' 'detune' and 'volume'. The 'pitch' Psp has several units, but the current one in use (as shown) is called 'st 440'. This is the number of semitones above or below concert A 440Hz pitch. The unit has a value increment (not shown in the figure) of 0.01, ie pitch can be specified to 0.01 of a semitone, ie 1 'cent'.

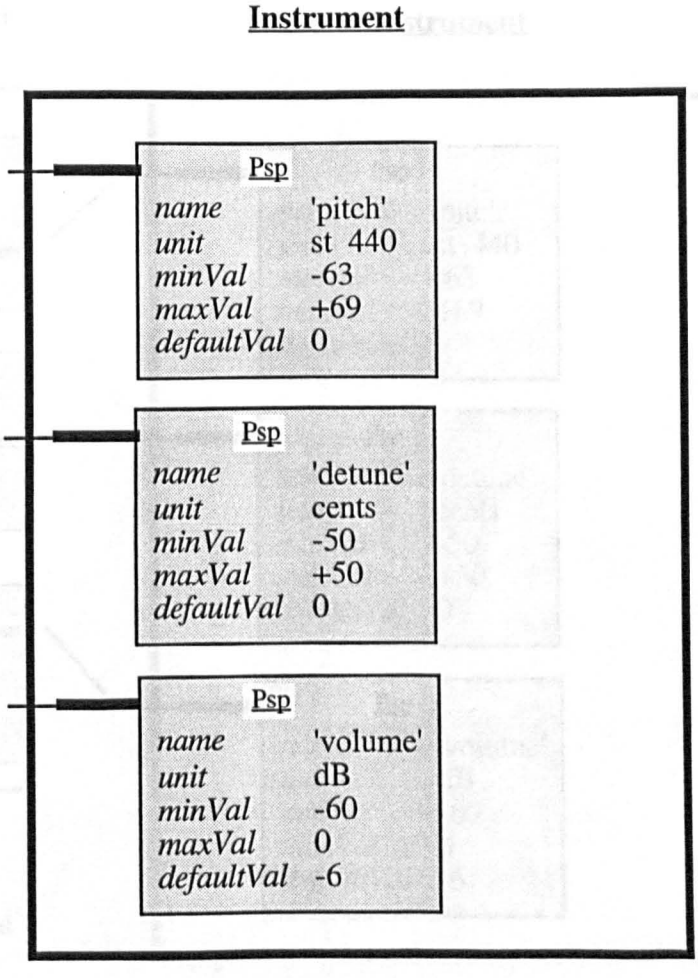


Fig. 21 An Instrument's input parameters, described by Psp objects

To create a new score event, the composer first selects an Instrument. A new 'default' ScoreEvent will then be created by E-Scape, which will contain a PspFunction object corresponding to each Psp input of the Instrument. This is shown in figure 22 below, where a new ScoreEvent has been created using the Instrument in figure 21. Each PspFunction contains a set of breakpoints (time and value pairs), and each PspFunction will initially be loaded with a single point: a data *value* equal to the appropriate Psp's default

---

1 In the early E-Scape design, 'Psp' was originally an abbreviation for 'Primitive sonic parameters', but these have now been termed 'Perceptual parameters' - see (Anderson 1993).

value, and a *time* of zero. This default value will thus pertain throughout the score event's duration. This is shown in the time-based display of the ScoreEvent as a straight horizontal line for each parameter, as illustrated in figure 22. Thus each new ScoreEvent has all its parameter values set on creation, and need not *necessarily* have any other values specified by the composer, although this would result in a highly boring piece of music! However, the composer is released from the necessity to laboriously specify a value for each parameter of each event, when he/she may only wish to deal with one or two event parameters.

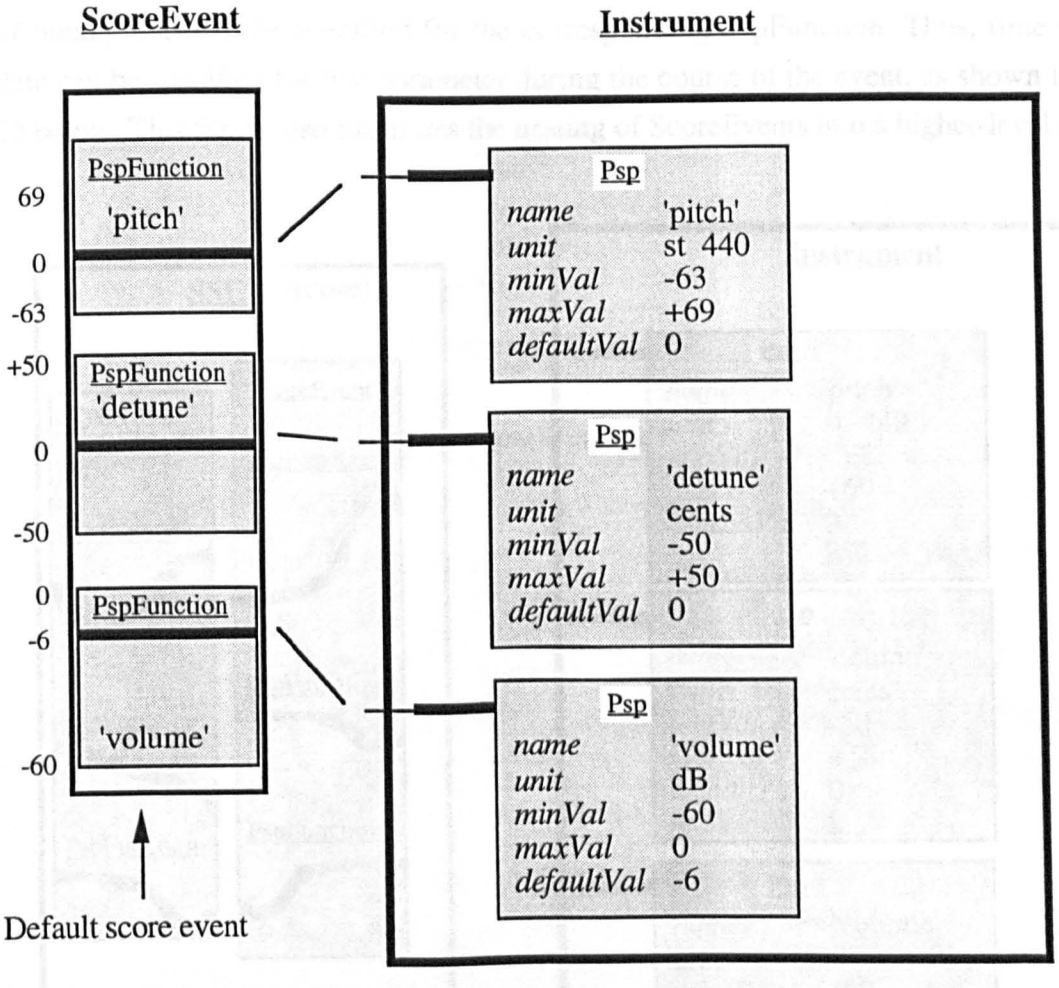


Fig. 22 A newly-created ScoreEvent, with its associated Instrument

A new ScoreEvent will usually be created within a SuperScoreEvent (SSE) object which defines a score structure (see 8.10). An SSE contains ScoreEvents, but can also itself be stored within a *higher*-level SSE. Thus any SSE can contain child events, which may either be ScoreEvents or SSEs, allowing nested event hierarchies to be created of any desired depth.

A composer may then specify (by a variety of input methods) a different value for any of the event parameters, which will be loaded into the appropriate PspFunction of the event.

Each Psp has a *rate* instance variable. This defines *when* the parameter can be altered during the performance of an event. Following CSound's nomenclature[1]:

- a rate of 'k' signifies that a parameter may be *varied* (ie its value updated) during an event.
- a rate of 'i' indicates that a value can only be specified once at the *start* of an event, and may not be updated subsequently.
- a rate of 'e' (not present in CSound) signifies that a value can only be specified at the *end* of an event - controlling some characteristic of *how* that event ends.

Thus, if a score parameter is described by a Psp which has a rate of 'k', then *any number* of breakpoints can be specified for the corresponding PspFunction. Thus, time-*varying* data can be specified for that parameter during the course of the event, as shown in figure 23 below. This figure also illustrates the nesting of ScoreEvents into a higher-level SSE.

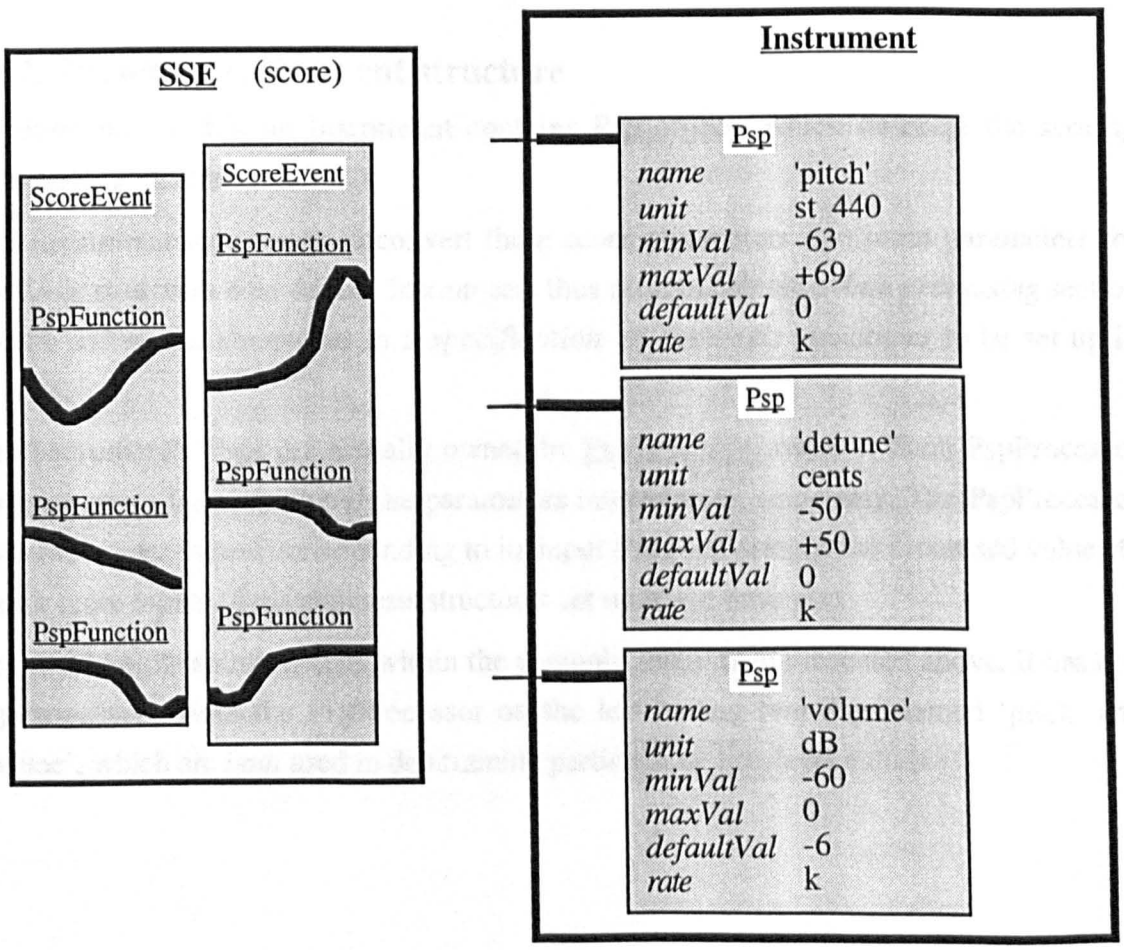


Fig. 23 Two ScoreEvents with user-specified parameter traces, and their Instrument

Instruments and their components are described more fully in the following sections; ScoreEvents and SSEs are presented in more detail in 8.9 and 8.10.

---

1 Note that CSound's 'a' (audio) rate is not supported in E-Scape, as audio rate data is not present within an E-Scape ScoreEvent.

8.4. E-Scape Instruments

8.4.1. Presentation of Instruments to user for scoring

As described in 8.3, a named Instrument object is selected by the user when creating a new ScoreEvent. The user is then presented with meaningful input parameters (such as 'pitch' or 'detuning') by this Instrument, each of which is described by a Psp object. A new ScoreEvent is created with a specified duration, using the specified Instrument. The ScoreEvent will then automatically be loaded with a default value for each Psp of this Instrument. A user may then optionally specify a data value or trace in a ScoreEvent, corresponding to each Psp in its Instrument: if no value in specified by a user, then the Psp's default value will be used.

Instruments should also specify synthesis structures to be instantiated on devices. These structures may be *distributed* amongst one or more types of device. Thus an Instrument needs to contain objects, each of which describes a structure on a single type of device.

8.4.2. Resulting Instrument structure

As described in 8.3, an Instrument contains Psp objects which describe the scoring parameters available.

The Instrument then needs to convert these score parameters into input parameters for synthesis structures on a device. Instruments thus need to contain a *data processing* section (in the software), connected to a *specification of synthesis structures* to be set up in devices.

The Instrument's Psps are actually owned by PspProcessor objects. Each PspProcessor can own more than one Psp if the parameters interrelate in some way. The PspProcessor processes score values corresponding to its input Psps and assigns the processed values to one or more inputs of the synthesis structures set up in the device(s).

Figure 24 below shows details within the example Instrument presented above. It has two PspProcessors, with the PspProcessor on the left having *two* Psps named 'pitch' and 'detune', which are *both* used in determining particular device-level values.

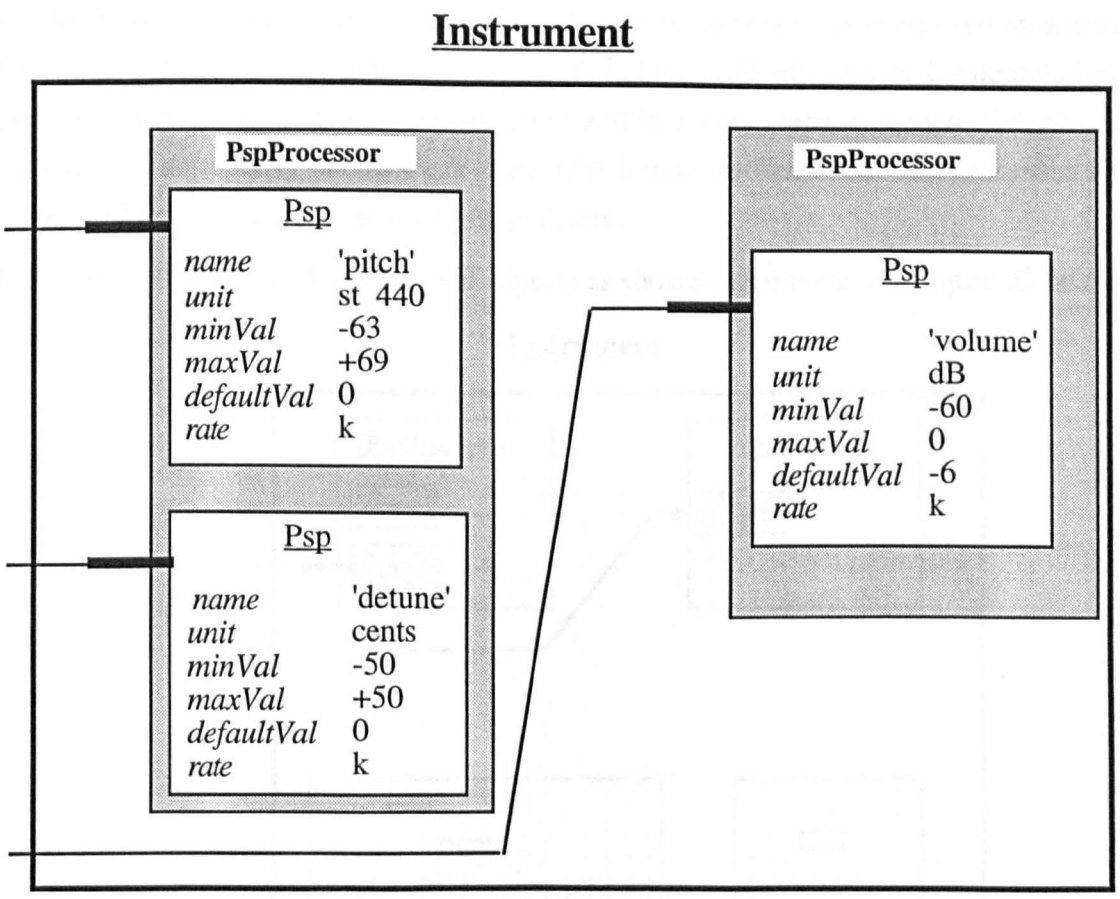


Fig. 24 'Front-end' structure of an Instrument - Psps are owned by PspProcessors

Thus, Psps act as the visible "front end" to synthesis structures on a device. Each Psp describes a musically meaningful parameter which can be specified in a score event which uses this Instrument.

Each Psp has:-

- a name
- one or more measurement units[1];
- allowed values - one or more value ranges (each with a minimum, maximum and increment) which need not be contiguous;
- a default value;
- a rate ('i', 'k' or 'e').

As stated in 8.3, each Psp is designated by the Instrument designer as being specifiable by a composer at the *start* of an event only (an 'i' rate Psp), at the *end* of the event only ('e' rate), or as a *continuous* time-varying value during an event ('k' rate). In figure 24 above, all the Psps in the Instrument are 'k' rate. Hence a composer may specify time-varying traces in a score-event which uses this Instrument.

---

1 A unit of measurement in a Psp (eg Hz) should not be confused with a *synthesis* unit - ie a unit process in a device.

The Instrument must contain a definition of the synthesis structures to be used on a device. These structures are defined by one or more DCT (Device Configuration Template) objects, each of which describes a synthesis structure within a single type of device. This structure consists of a network of one or more units, which may be connected. The processed score values will then be sent to inputs of this structure.

The example Instrument has two DCT objects as shown at the bottom of figure 25 below.

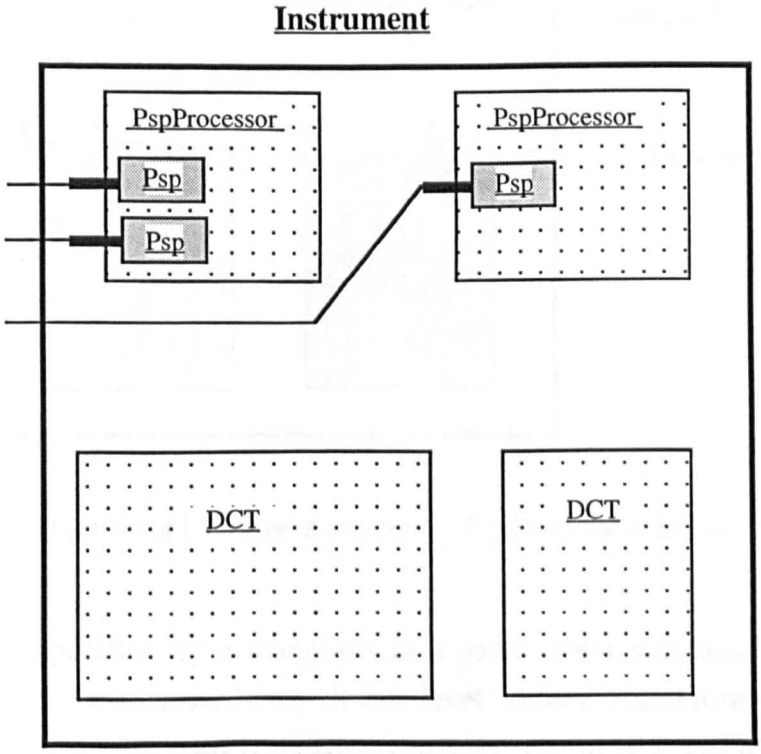


Fig. 25 Instrument structure, showing DCTs (links not shown)

An Instrument can contain more than one DCT, and can thus specify structures which can be *distributed* over one or more devices of one or more types. Thus, for example, a single E-Scape Instrument can both control MIDI-based *and* custom devices. Each DCT has inputs (DCTInput objects) which connect, and provide access to further structures within the DCT (described below). A PspProcessor has one or more of these DCT inputs assigned to its instance variables, and thus has a connection to them, as shown in figure 26 below.

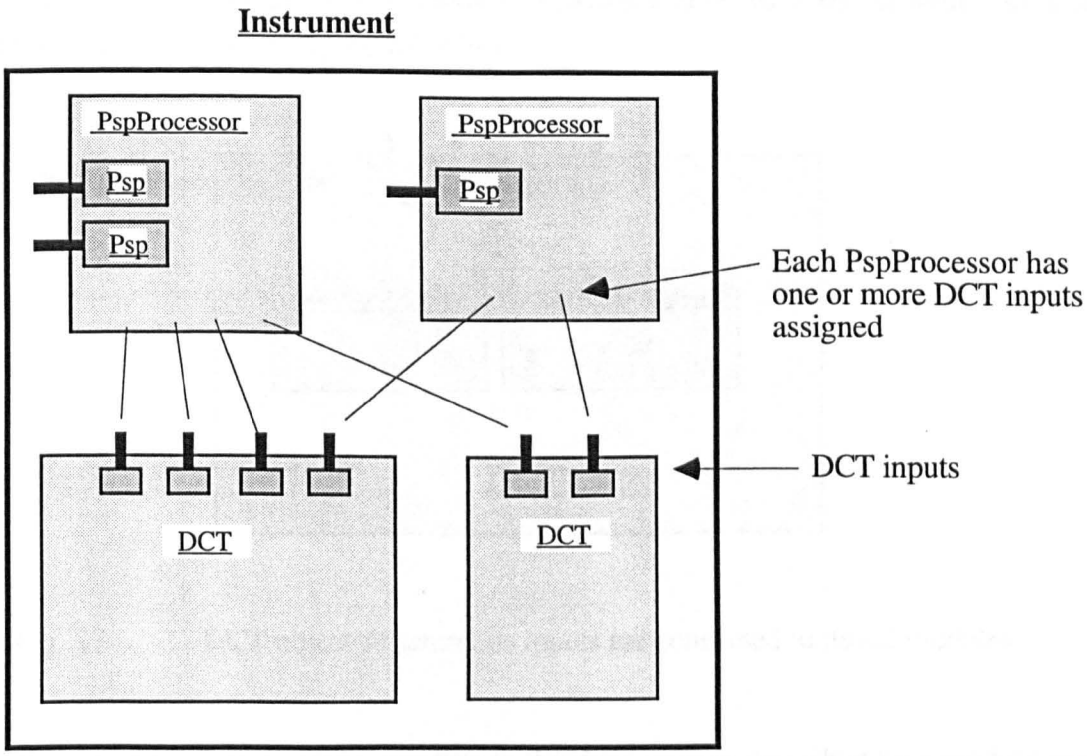


Fig. 26 High-level Instrument structure - PspProcessors linked to DCTs

Parameter value(s) specified by a composer in a score event will then be sent to a PspProcessor object to be processed into 'device-level' values. These lower-level values are assigned to the DCT inputs and will subsequently be sent to inputs of the structure in the device which is *described* by the DCT. The methods by which this processing is performed and specified will be described below, but it may involve a PspProcessor *interrogating* a DCT input to find out some information about the corresponding input of the device structure, such as its maximum permitted value.

If a device supports structures and units which can perform data manipulation (eg by expressions or conditional branches), then this kind of processing can be performed in the *device*. Some or all of the processing functionality which would normally within the PspProcessors could then be specified as part of the *DCT* (structure description) and instantiated in the device, thus requiring little or no work to be done by the PspProcessors. Thus the software / hardware boundary is moveable to fit the capabilities of the device.

The main components of an Instrument - DCTs and PspProcessors - are now described in more detail.

8.4.3. DCTs

Each DCT object defines a synthesis structure in a single type of device, consisting of a network of 'device-level' synthesis processes ('units'). A DCT is constructed from, and is seen by the user as containing, sub-modules (DCTSubModule objects). Each sub-module may describe a synthesis unit, or be more complex - describing a network of units.

A DCT has inputs (DCTInput objects), each of which is connected to one or more inputs of its submodules, as shown in figure 27 below.

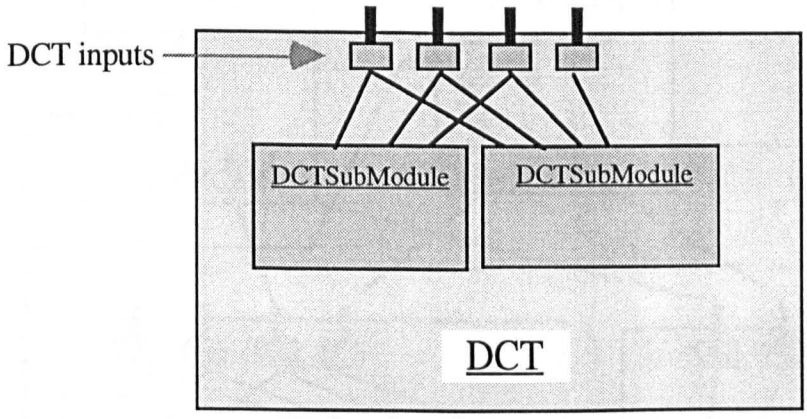


Fig. 27 DCT object structure: its inputs are connected to its submodules

Each DCTSubModule is created according to the specification described by a module *type* object[1] which defines the characteristics of the module and its inputs - for example, its address offset within the device's memory map, a type code identifying the module type to the device, or the range of input values which may be sent to a module input.

Thus, the full top-level structure of an Instrument appears as in Fig 28 below, with each PspProcessor owning one or more DCT inputs. A notional data flow path can thus be understood, with data 'arriving' from the score via the Psp 'inputs' (shown as three wires entering the left of the Instrument), being processed by the PspProcessors, then sent to the DCT inputs, and thence to the DCTSubmodules within the DCT.

These data values will eventually (after device resource allocation) be sent to the units in the device which these DCTSubModules describe. This stage will be described later in section 9.3.2.

[1] These module types objects are PrimSMT or SMT objects - described in 8.6 below.

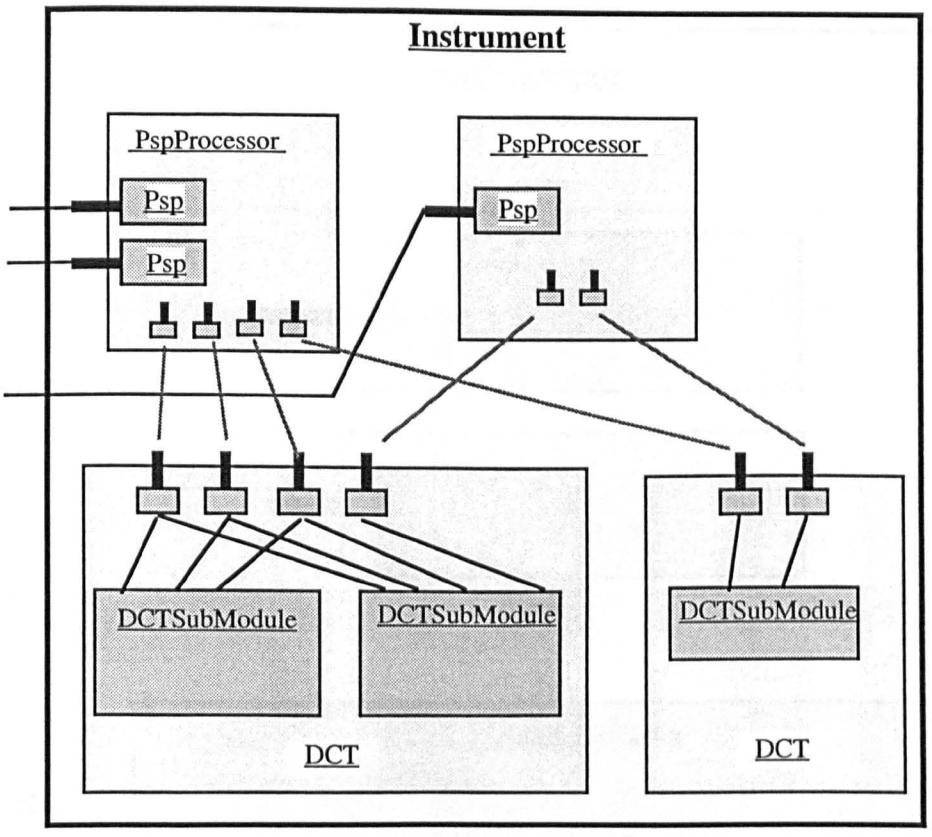


Fig. 28 The detailed structure of an Instrument

8.4.4 PspProcessors

A PspProcessor acts as the top-level layer of an E-Scape Instrument, translating score parameters into device parameters. It performs user-specified manipulations on score parameter data, to produce input data for a device synthesis structure.

A PspProcessor has three chief *instance variables* (see 7.5.2.2), as shown in figure 29 below:-

• *'inputPsps'* is assigned to a set of Psp objects. These define the input parameters which a user of the Instrument can specify from the score, ie which are available to a ScoreEvent which is assigned to this Instrument.

• *'outputDCTInputs'* is assigned to a set of DCT inputs (DCTInput objects) within the Instrument. It is to these inputs that the PspProcessor assigns its processed 'output' data. Note that these DCTInputs are also 'owned' by the DCT.

• *'codeDic'* is assigned to an ECodeDictionary object which actually performs the processing.

These Psp and ECodeDictionary objects will be described below.

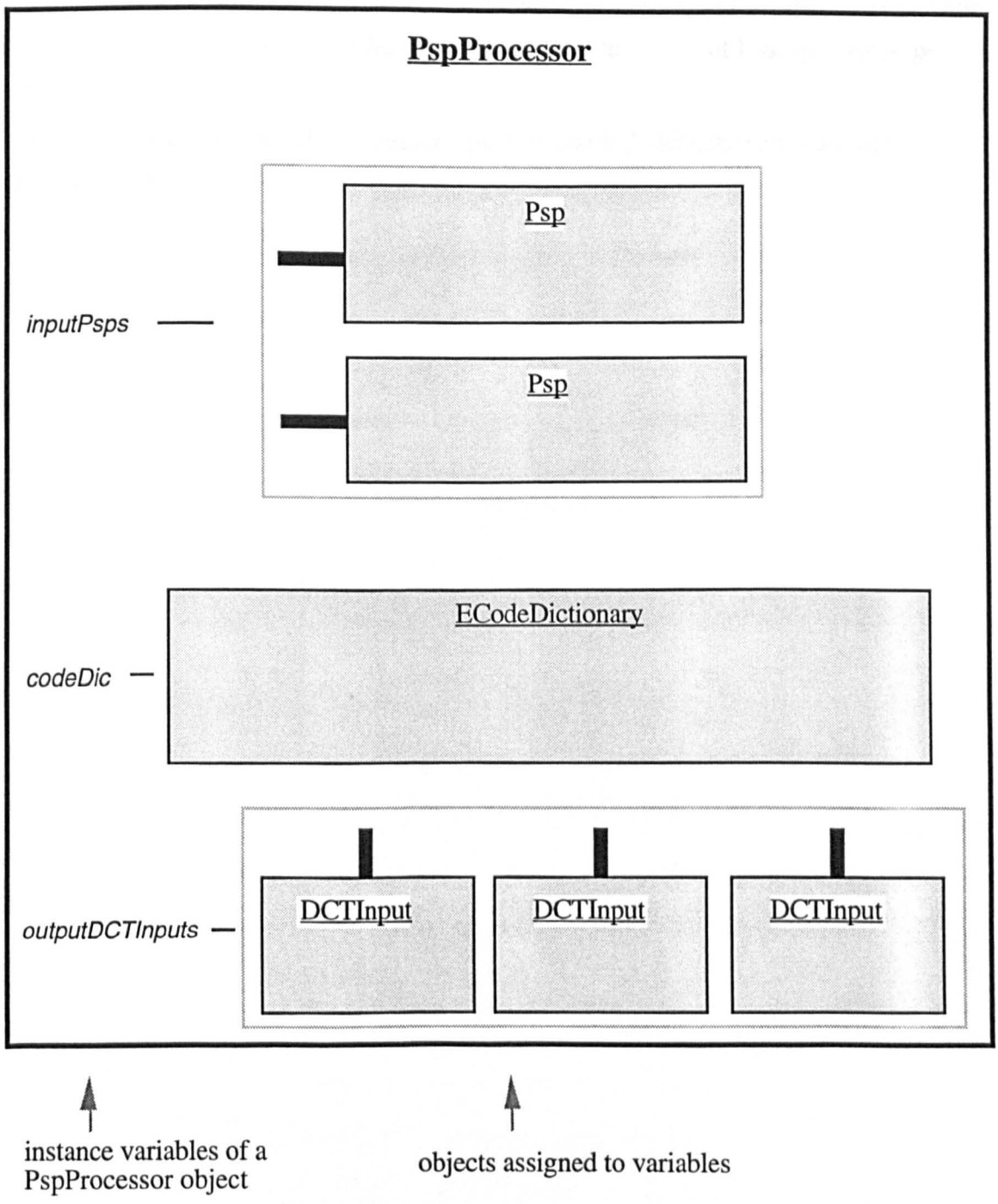


Fig. 29 PspProcessor object structure

A PspProcessor also has instance variables *inputGlobals* and *outputGlobals* (not shown) which can be assigned to one or more globally available data traces. For example a vibrato shape might be defined as a global signal and assigned amongst *inputGlobals*, enabling it to be used as another input of the PspProcessor which can be used at some stage of its score parameter processing. Similarly, a PspProcessor may create a data trace which can be assigned to its *outputGlobals* , thus making it available to other objects in the system. The concept of global data shapes is also present in HMSL amongst others. It has therefore not been given a high priority in the development of the E-Scape prototype. Time resources have been concentrated on those features of E-Scape which are innovative, at the expense

of the inclusion of facilities (however useful) which are to be found in other system. Thus, this aspect of E-Scape has not yet been implemented at this stage of E-Scape prototype development.

The main components of a PspProcessor - its Psps and ECodeDictionary - are now described in detail.

8.4.4.1. Psps

A PspProcessor has one or more inputs. These are defined by Psp objects which describe details of the score parameters which will be made available to a ScoreEvent using this Instrument. Each Psp of an Instrument effectively describes a perceptual or sonic parameter which a composer using that Instrument may specify. Figure 30 below illustrates the structure of a Psp object.

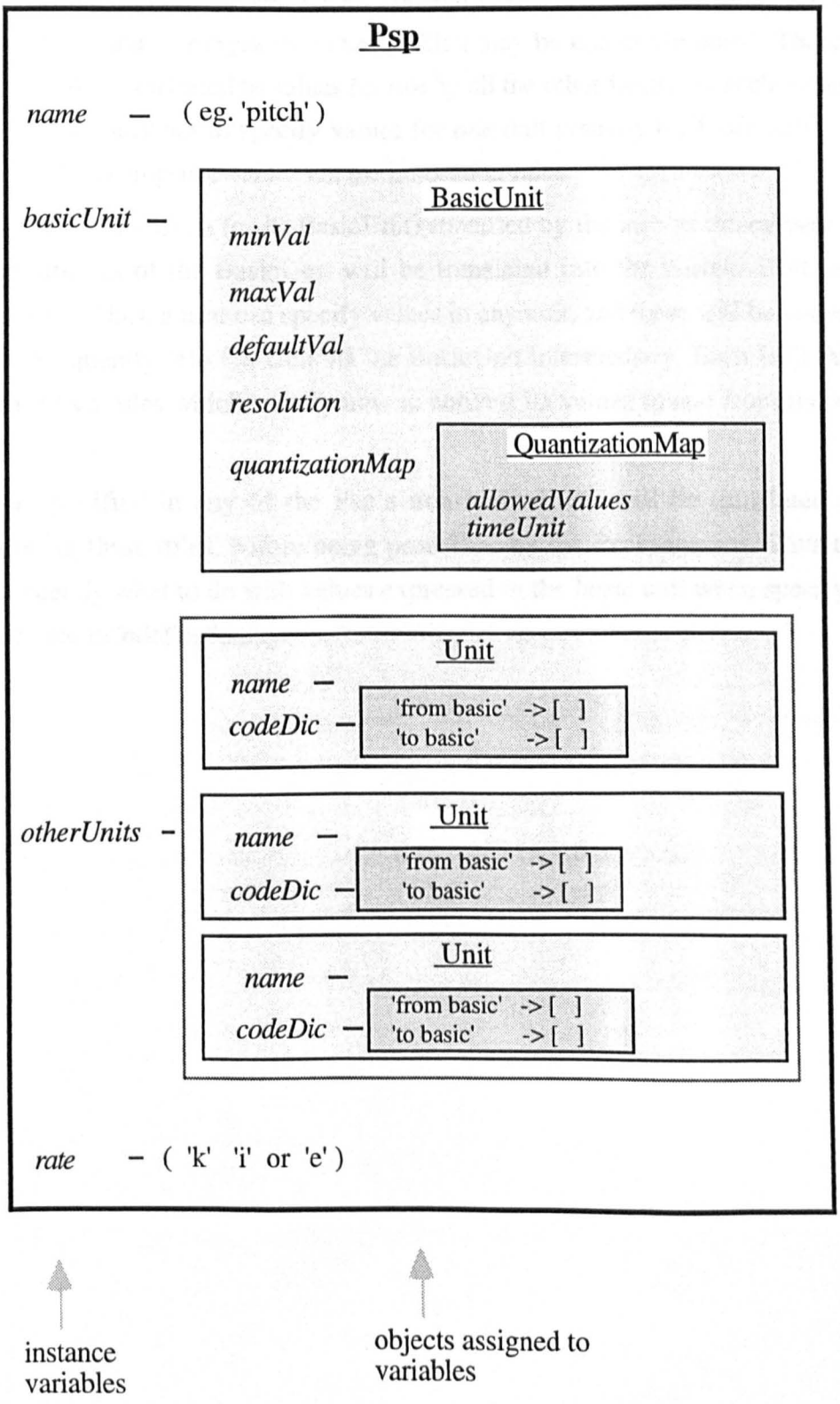


Fig. 30 Psp object structure

A Psp defines the units, acceptable input value ranges, and default values for score parameters. Each Psp has a name (eg. 'pitch'), and details of one or more measurement units which can be used for that parameter. Each unit is described by a user-specified Unit object, which has a name (eg 'Hz') and translation rules to and from another unit designated as "basic".

This "basic" unit is described by a BasicUnit object which actually stores the minimum, maximum, and default values, and has a QuantisationMap object which specifies the allowed time and parameter values. These 'allowed values' can be specified as one or more individual values, and/or *ranges* of values (which may be non-contiguous). These 'basic unit' values are then converted to values for use by all the other Units, via each's translation rules. Thus a user only has to specify values for *one* unit (usually the basic unit), and the others receive the appropriate values via the translation rules.

The Psp has one of its Units (or its BasicUnit) specified by the user as the *current* Unit in use. All parameters of the BasicUnit will be translated into the current Unit using its conversion rules. Thus, a user can specify values in any unit, and these will be converted to any other subsequently selected Unit via the BasicUnit intermediary. Each Unit thus only needs to have two rules which specify how to convert its values to and from those of the BasicUnit.

Any values specified in any of the Psp's non-basic Units will be translated into its BasicUnit using these rules, before being processed by the PspProcessor. Thus the user only has to specify what to do with values expressed in the *basic* unit when specifying the algorithms in the ECodeDictionary.

8.4.4.2 ECodeDictionary

This object is the heart of a PspProcessor object. It performs user specified manipulations on the parameter values of a ScoreEvent (contained in its PspFunctions) to create 'device-level' data which is assigned to inputs of a device synthesis structure (described by a DCT).

An ECodeDictionary object, shown below in figure 31 contains one or more named DBlockContext objects which act as user-defined functions. Note that the names on the left hand side are names (keys) within the Dictionary (see 7.5.2.5), rather than *instance variables.*

DBlockContext is a DMIX object class which has been subclassed (see 7.5.2.4) from the standard Smalltalk BlockContext class (Goldberg 1983). Objects of this class are commonly referred to simply as 'blocks', and contain Smalltalk code which can then be treated as an object, and executed with parameters passed if appropriate. The DBlockContext subclass adds the capability for code to be defined and edited as a *string* by a user which is then compiled to create the block.

The ECodeDictionary can contain any number of user defined blocks, but must contain at least one block named 'userProcess'. This contains code which can access data from a score event which is assigned to this Instrument, perform operations on it, and then assign the processed data to specified inputs of a DCT within the Instrument.

Figure 31 below shows (in faint print) a user-entered Smalltalk code string for the 'userProcess' block.

This code will be explained later in section 9.2.2, and is printed here to give an impression of the nature of a DBlockContext as seen by a user, who types this code in. As described in 6.1, the initial design for E-Scape envisaged a graphical iconic MAX-like[1] presentation of this processing code. This has not yet been implemented, however, as many other systems have been found to employ such presentation, and development effort has therefore been diverted to concentrate on other, more innovative, features of E-Scape.

---

[1] MAX allows graphic icons to be manipulated via a GUI in order to specify data processing networks - see 3.2.1.1.

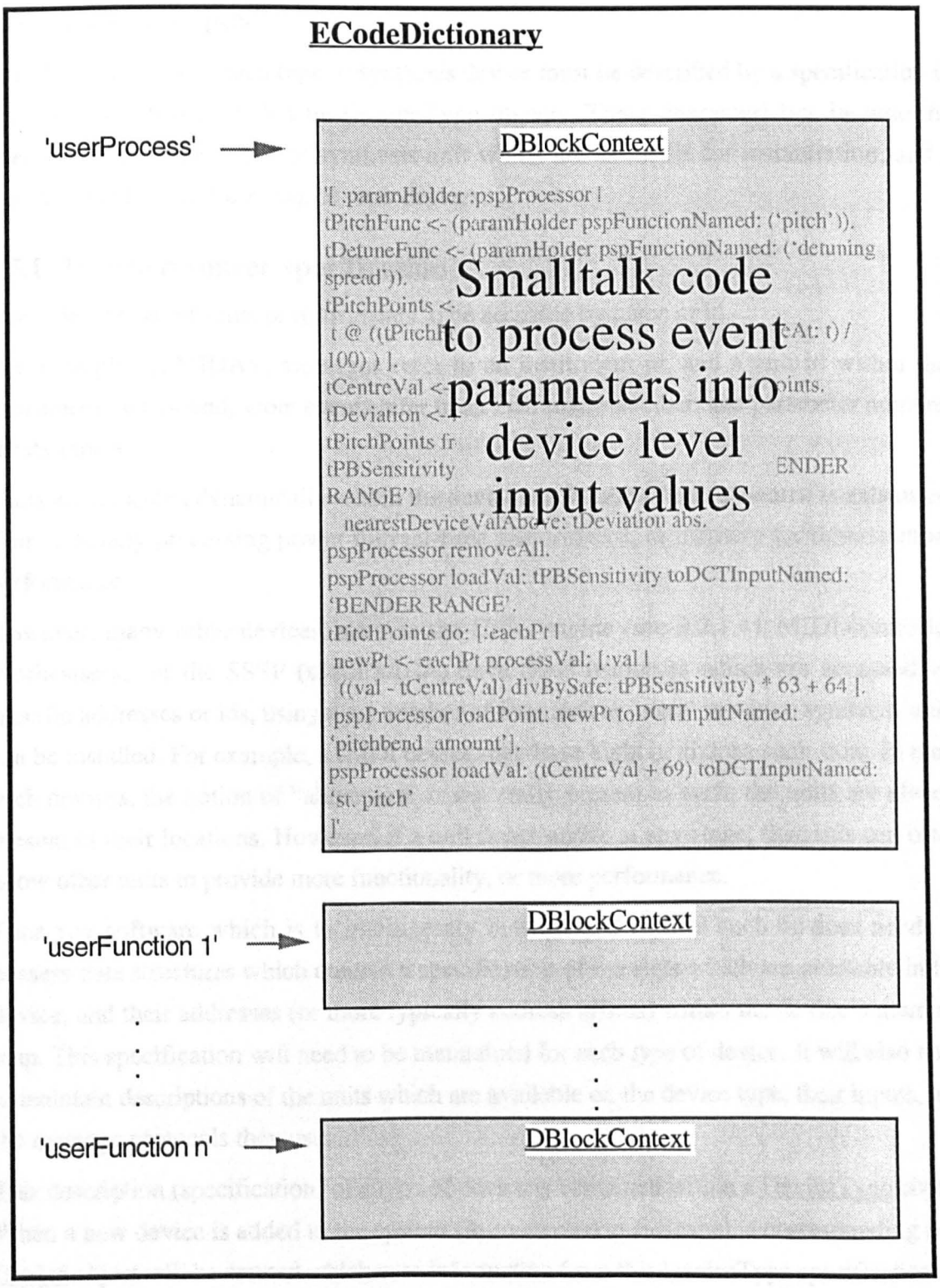


Fig. 31 CodeDictionary object structure

8.4.4.3 Output to DCT inputs

After processing score parameter values the PspProcessor then needs to assign the processed data to one or more inputs of DCT(s) in the Instrument (see figure 28 above).

The user can assign (within the 'userProcess' block code) one or more DCT inputs to receive data from the user-defined code block of the PspProcessor. These DCT inputs are then made *unavailable* for connecting to other PspProcessors. This avoids possible conflicting data being sent to a DCT input by different PspProcessors.

8.5. DeviceTypes

The characteristics of each type of synthesis device must be described by a specification in E-Scape, which is provided by DeviceType objects. These characteristics include the *specifications* of the types of synthesis *unit* which are available for instantiation, and a description of the address map of such devices.

8.5.1. Device resource specification

Many devices allow units or instruments to be accessed by name or id.

For example, in MIDAS, messages refer to an instrument id, and a unit id within that instrument; in CSound, score events refer to an instrument number, and parameter numbers for its inputs.

Units are allocated dynamically within the device until some system resource is exhausted. This is usually processing power for real-time performance, or memory for time-stamped performance.

However, many other devices, such as the UPIC engine (see 3.2.1.4), MIDI-controlled synthesisers, or the SSSP (see 3.2.1.14) have fixed resources which are accessed via specific addresses or ids, using a set number of locations or 'slots' in which synthesis units can be installed. For example, a MIDI device may have eight or sixteen such slots. In most such devices, the notion of "allocation" is not really present as such: the units are always present in their locations. However, if a unit is not *active* at any stage, then this can often allow other units to provide more functionality, or more performance.

Thus any software which is to intelligently manage and control such devices needs to possess data structures which contain a specification of the slots which are available in the device, and their addresses (or more typically address *offsets*) within the device's memory map. This specification will need to be maintained for each *type* of device. It will also need to maintain descriptions of the units which are available on the device type, their inputs, and the message protocols they use.

This description (specification) of a type of device is contained within a DeviceType object. When a new device is added to the system (ie connected to E-Scape), a corresponding new Device object will be created which uses information from this DeviceType specification.

8.5.2 Structure of a DeviceType

As stated above, a DeviceType object describes the features of a kind of synthesis device. Its chief components are shown in figure 32 below.

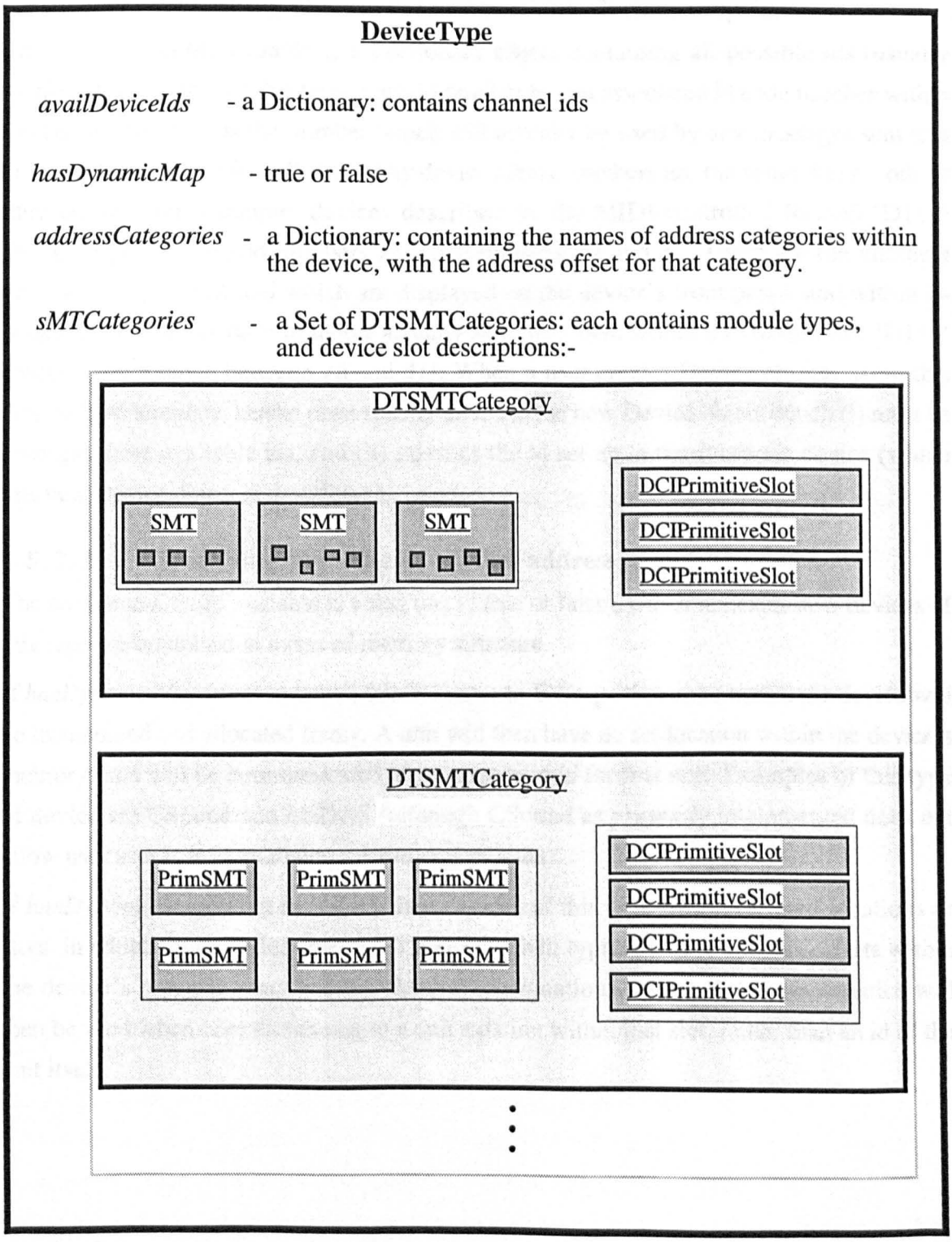


Fig. 32 DeviceType object structure, with example module categories

As usual, the *instance variables* of a DeviceType object are shown on the left-hand side, and the objects which are assigned to them are shown or described on the right. The

instance variable names are not seen by the user, hence are verbose but meaningful to the programmer. The same can be said of object class names. For example, a DTSMTCategory object is presented to a *user* as a simple 'category'. These instance variables are now described in detail.

8.5.2.1 Describing the available ids which devices may have

The *availDeviceIds* variable is a Dictionary object containing all possible ids (usually numbers) for devices of this type. Each id number has an associated id code number within the Dictionary. This is the number which will actually be used by any messages sent to a device which require its 'id'. On many devices these numbers are the same, but on others they are not. For example, devices described by the MIDI-controlled Roland 'D110' DeviceType can have id numbers assigned between 17 and 32. These are the numbers understood by a user, and which are displayed on the device's front panel, and within E-Scape. However, the code numbers actually sent (as a field within a message) to a 'D110' device have a range between *16* and *31*[1]. When a user creates Device objects using this DeviceType template, he/she must specify an id for the new Device object which (i) must be amongst these available ids, and (ii) matches the id set up in the synthesis device (which this new Device object is describing).

8.5.2.2 Specifying the type of device address map

The *hasDynamicMap* variable is a flag (set to true or false), which indicates how devices of this type are organised in terms of memory structure.

If *hasDynamicMap* is set to 'true', this indicates to E-Scape that units within the device will be instantiated and allocated freely. A unit will then have no set location within the device's memory, and will be communicated to by a specific *id* for that unit. Examples of this type of device are CSound and MIDAS (although CSound as *presently* implemented does not allow user access to instantiated instruments or units).

If *hasDynamicMap* is set to 'false', then devices of this type will have fixed locations or slots in which units are instantiated. These slots will typically have address offsets within the device's memory map, and/or 'channel' designations. These slot characteristics will then be used when communicating to a unit existing within that slot, rather than an id of the unit itself.

---

[1] This is as per usual with the MIDI protocol. Numbers (eg MIDI 'channels') start at zero *as transmitted*, but are often presented by devices to the user as starting at 1.

8.5.2.3. Describing the device address map

For types of device which have a *fixed* memory map (ie for which *hasDynamicMap* is false), the *addressCategories* variable is assigned to a Dictionary object which describes this map.

Some types of device (especially MIDI-based ones) may have a *complex* address map, in which each slot has *several* different address offsets within different 'categories' of address. Each address 'category' will also have a different base address within the device. The *addressCategories* Dictionary will thus need to contain a *base* address for each *category* of address within the device.

As described in 6.4, E-Scape attempts to treat all synthesis entities which can be independently started and stopped as *unit* objects within the device, in order to describe and present synthesis structures within different devices in a consistent manner. Each unit may however be built within a device out of lower-level component entities which may be *presented* and/or *stored* independently in the device, but which are not independent units in themselves. Each component of a unit may have inputs defined, which have offsets within the device address map. In typical such devices, each component of a unit may have an offset within a *different* section of the address map - described in E-Scape by an 'address category'. The components of a unit may thus have *different* address offsets for a particular device slot even though they are combined together to create a single unit within that slot. Thus, each input of a unit may have an address offset within the device in one of several address categories.

An example using the 'D110' DeviceType will illustrate these concepts:

A unit within a D110 device is described (in the manufacturer's manual) as being built using notional 'TONE' and a 'TIMBRE' entities. A 'TONE' entity has parameters which define most of the synthesis structure and its input parameters. A 'TIMBRE' entity is associated with a 'TONE', and adds additional parameters such as 'stereo panning' and 'overall volume' to the unit definition. These are *notional* entities within the device which can be *stored* independently, but which alone do not fully specify an independent synthesis unit, as defined above. Thus a unit (as defined by E-Scape as an object) subsumes both the 'TONE' and 'TIMBRE' notional entities.

A 'D110' device has eight slots for such units, and each slot has *different* address offsets within the device for the 'TONE' and 'TIMBRE' aspects of a unit within that slot. In addition, other aspects of a slot within the D110 device (eg the messages used to assign its 'channel' number) require address offsets in further address categories within the address map (eg the category named 'SYSTEM').

To summarise: in a device with a fixed address map, each slot may need to have its address offsets specified within *several* address *categories*. Its actual address will then be calculated using the *base* address within the device in each address category. Such a device *base* address is stored in the DeviceType's *addressCategories* for each address category.

8.5.2.4 Categories of modules within devices

The *sMTCategories* instance variable is a Set of DTSMTCategory[1] objects. Each DTSMTCategory stores a particular category of module type (which describes a set of synthesis units). A DTSMTCategory may also describe the *locations* in a device where this type of module may be installed (instantiated). It is described in detail in the next section.

8.5.3. DTSMTCategories

Many device types provide for the creation (instantiation) of networks of synthesis units which can be constructed with no restrictions on what units may connect to what - all units are indistinguishable so far as connecting and instantiating them goes. However, most, if not all MIDI-based devices encountered do not have this freedom, and provide for instantiation of *different* numbers of units in different *categories*, often with additional restrictions on any connections made or even which units may coexist at the same time.

Thus, a DeviceType object must allow units to be classified into different categories, when describing a device's characteristics. As just stated above, it does this by providing DTSMTCategory objects, each of which stores the specifications of synthesis units (of that category) which are available for instantiation.

These specifications are contained in PrimSMT or SMT objects which act as module 'type' or 'template' definitions. Each defines a unit or network of units on a device. A PrimSMT object describes basic (primitive) entities within a device which cannot be broken down any further. An SMT object is built up out of child submodules, which may be based either on PrimSMTs or other SMTs. It describes a higher-level entity within a device. Note that either object can describe a *unit* in a device.

A DeviceType will typically has more than one module category (DTSMTCategory), and a device will usually have more than one module type defined within each of these categories. For example, the (imaginary) DeviceType illustrated below in figure 33 has two of its DTSMTCategories shown. The category called 'variable filtering' contains three module type definitions (SMT objects), ie devices of this type have three different types of module (synthesis structure) in this category. The category called 'looped voices' contains six module type definitions (PrimSMT objects). These types of module are likely to have

---

1 An etymological note: The DTSMTCategory object class is a subclass of the SMTCategory class. The latter contains SMTs (SubModuleTypes), hence SMTCategory stands for SubModuleType Category. Those SMTs which are 'device-level' are a special case of these, requiring additional information to do with device resource allocation, and the characteristics of DeviceTypes. Thus the category for them is a DTSMTCategory (or DeviceType SubModuleType Category). It goes almost without saying that these object class names are verbose, and are never used unabbreviated. Any object class names chosen will be alien to some extent, and it is better to err on the side of semantic clarity, with rigorous and consistent conceptual structure, rather than simplistic linguistic readability.

specific locations in the device where they can be installed, as well as other characteristics (described in 8.5.3.1) which make them non-interchangeable, and hence requiring to be stored in separate categories.

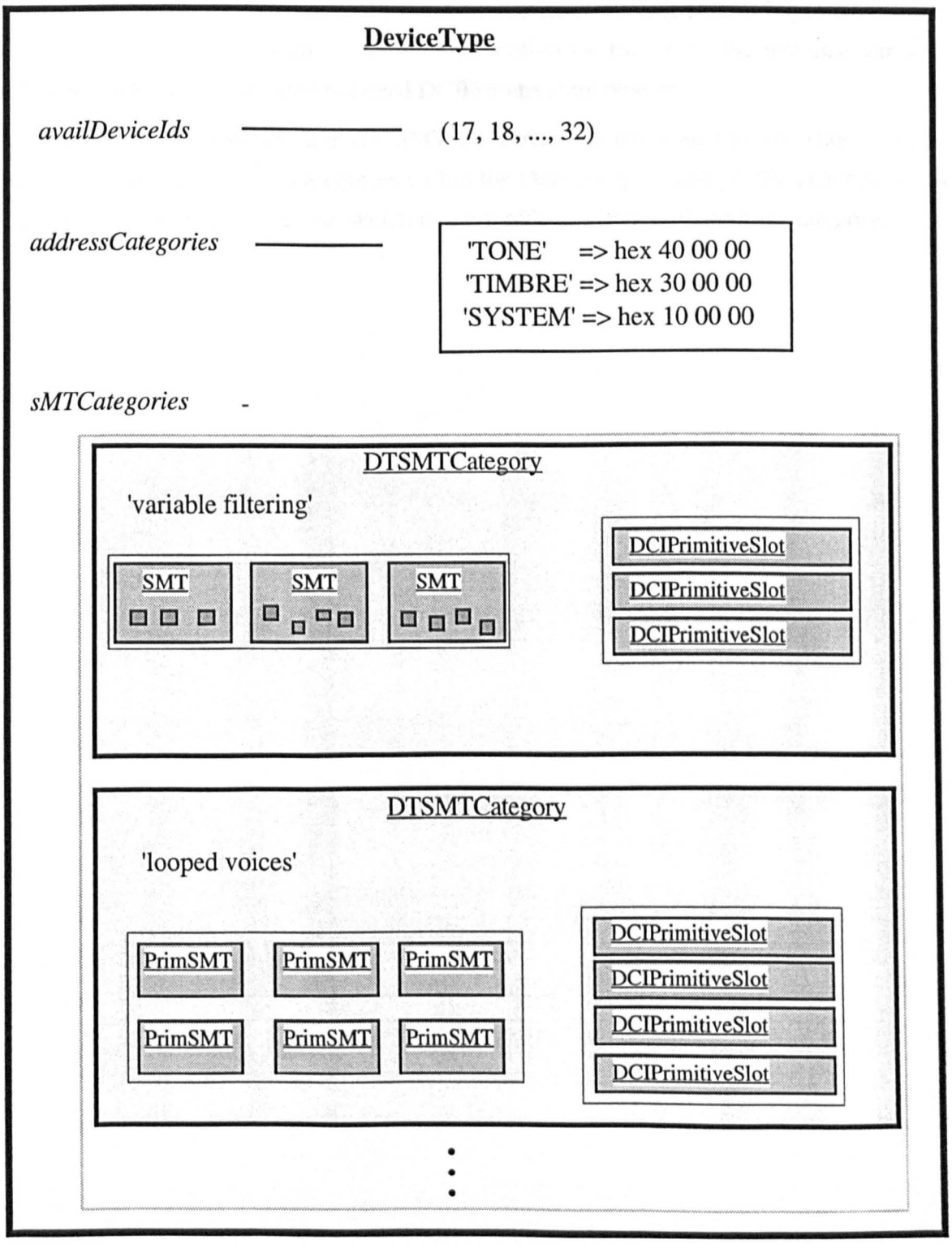


Fig. 33 An example DeviceType

The device resource slots (described above in 8.5.1) will also be different for each category, thus each category also contains the specifications of any 'slots' - ie locations or addresses - into which units may be instantiated on this type of device. The characteristics

of these slots are described by DCIPrimitiveSlot objects within each DTSMTCategory object. The example DeviceType illustrated above has three DCIPrimitiveSlots in its 'variable filtering' category, and four DCIPrimitiveSlots in its 'looped voices' category. It is thus able to instantiate four modules of the 'looped voices' category at once, and three of the 'variable filtering' category. It should be remembered that DeviceTypes describing devices which employ *dynamic* resource allocation (ie for which the instance variable *hasDynamicMap* is *'true'*) will not need DCIPrimitiveSlot objects.

Note that these categories of unit (DTSMTCategories) are not related to, and should not be confused with, the *address* categories within the DeviceType. Each DTSMTCategory has DCIPrimitiveSlots, as described, which possess offsets within each *address* category.

8.5.3.1 Structure of a DTSMTCategory

To summarise the description above, a DTSMTCategory object defines the characteristics of a particular kind (category) of unit within a device. Each DTSMTCategory contains specifications of units of this kind (as SMT or PrimSMT objects). If a device has a fixed resource allocation and address map, then the DTSMTCategory will also contain definitions of slots in the device (as DCIPrimitiveSlot objects) in which these units may be instantiated.

The structure of a DTSMTCategory is now shown in more detail in figure 34, below .

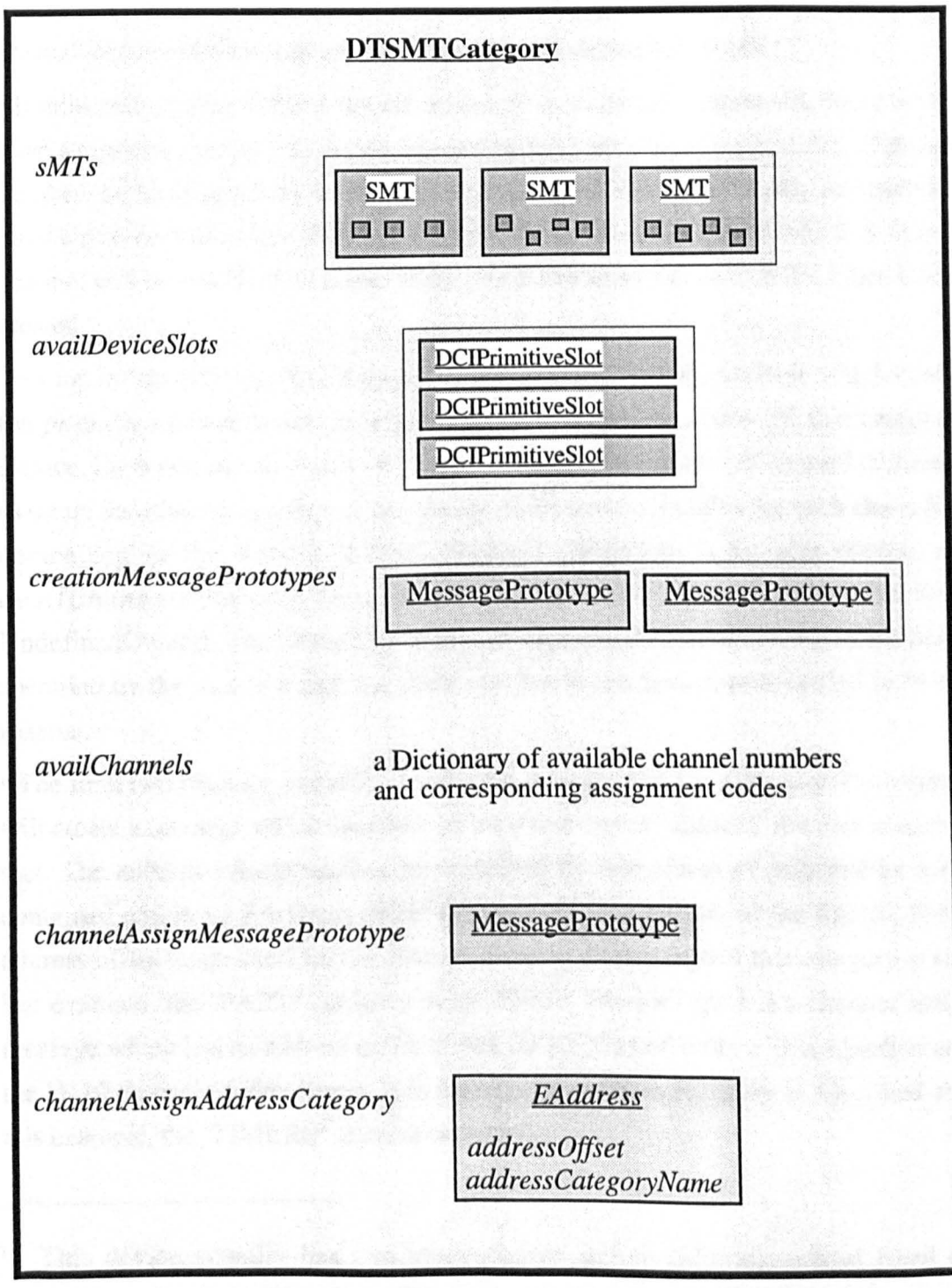


Fig. 34 DTSMTCategory object detailed structure

The chief instance variables of a DTSMTCategory are shown on the left:

• *sMTs* is a set of module type (SMT or PrimSMT) objects which specify units as described above. Their detailed functioning is presented in detail in 8.6 below.

• *availDeviceSlots* is a set of DCIPrimitiveSlots, again as described above. The structure of DCIPrimitiveSlots is presented in detail in 8.5.3.2 below.

• *creationMessagePrototypes* is a set of one or more MessagePrototype objects. When units of this category are to be instantiated on a device, messages of a certain format will need to be sent to the device. Thus each category contains one or more MessagePrototype objects which will create the appropriate messages to be sent when units of that category are created. MessagePrototypes are explained fully in section 8.8.2 below.

• If units instantiated within a slot are accessed via a channel assignment, then each slot will have a channel number which will be used by any unit occupying that slot. If this channel number can be assigned by a *user* for a device (rather than being fixed), then each slot may have a *different* user-specified channel in each device of this type which is in use. This channel will be specified by a user when a new Device object (which describes a device) is created.

To support this activity, the DeviceType has an *availChannels* variable which contains all the *possible* channel numbers which may be assigned to a slot (of this category) in a Device. Each number may also have an associated code which will be used in the message to set up this channel number. Alternatively, if the channel number for each slot is *fixed* in a device (eg in the Roland 'MT32' device[1]), then there is *no* user choice, and the *availChannels* instance variable will be set to 'nil' (an instance of a Smalltalk UndefinedObject). The *channel* number for each DCIPrimitiveSlot will then need to be specified by the user to match the fixed number in the device, as specified in its owner's manual.

• The final two instance variables contain the *address offset* and *MessagePrototype* which will create a message which will then be used to assign a 'channel' number to each device slot. The address offsets need to be described by two pieces of information which are contained within an EAddress object (illustrated at the bottom of the figure). Firstly, the address offset value itself for the channel assignment message of this category is required. For example, the 'PART' category of the 'D110' DeviceType has a channel assignment message which has an address offset of hex 00 10. This offset is within a particular part of the D110 device's address map, ie in a particular *address category* as described above; in this example, the 'TIMBRE' address category.

---

1 This device actually has two modes which define differing sets of fixed channel numbers for each of its slots. Which mode it is operating in will be determined by the Device's set-up messages. This illustrates the kind of arbitrary complexity which is so difficult to describe without introducing immense conceptual and practical complexity into the user-interface.

When an actual D110 device is to be connected to the system, a 'D110' Device object is created by a user[1] to describe it. Each slot in the new device (within the 'PART' category) must be assigned a channel by the user. A message will then be formulated, which will be sent to the device in order to set up this channel assignment for the slot. To determine the address to use (as one of the fields of the assignment message) the DeviceType's *base* address for the 'TIMBRE' address category will be added to this address offset value for the category. The slot itself will supply other data fields of the message, and the channel number itself (see next section).

Again, if the devices of this type do not have slots, then these objects will not be present (ie the instance variables will be set to 'nil').

8.5.3.2 DCIPrimitiveSlots

DCIPrimitiveSlots describe the features of a device's resource slots, if any - ie if the device has a fixed address map - as discussed above. Its chief instance variables are shown below in figure 35.

DCIPrimitiveSlot

name - derived from the *parent* category, and the *id* of this slot

parent - the DTSMTCategory which owns this slot.

id - the identifying number of this slot within this category in the device, eg = 1

channel - set to a number, if this slot has a fixed channel designation
- set to 'nil' if channels can be specified by user

channelAssignOffsetAddress
- the address offset value (and address category) to use in constructing the channel assignment message for this slot

addressCategoryOffsets
- a Dictionary of address categories, and offset values for this slot.

Fig. 35 DCIPrimitiveSlot object structure

---

1 Such a user may or may not be the same person as the composer. A composer need not necessarily have to operate at this level, and can simply use the Devices already defined within E-Scape.

• Each slot typically has an id number to identify it within the device, and this number is assigned to the *id* instance variable of the corresponding DCIPrimitiveSlot.

• Each slot also typically has a channel number, which may be fixed (eg in the Roland 'MT32' device), or be user-selectable in each device (eg in the Roland 'D110' device).

In the former case - ie if the channel number is *fixed* for each slot in a device - then the *channel* instance variable is used. This DCIPrimitiveSlot describes the features of a device slot in *all* devices described by this DeviceType. Thus when a user is creating a new DeviceType object (to describe a new kind of device) he/she must enter the *channel* number for each DCIPrimitiveSlot to match the slot channel numbers which exist in devices of this type. These numbers will be specified in the device's owners' manual. For example, the MT32 device has a 'PART' category which has eight DCIPrimitiveSlots (with fixed channel numbers from 2 to 9), and a 'RHYTHM' category which has a single DCIPrimitiveSlot (with a fixed channel number of 10).

A Device object may then created by a user (using this DeviceType as a template) in order to describe a device of this type which is being newly connected to E-Scape. The user will then *not* be required (or be able) to specify the channel of each slot in the new device - they are fixed for this DeviceType within the DCIPrimitiveSlot specification

In the latter case - ie if a channel number can be assigned by a *user* for each slot in a device - then the DCIPrimitiveSlot (which describes slots on a *type* of device) will *not* have a fixed *channel* value specified. Only when a Device object is subsequently created by a user (to describe a newly attached device), will each slot in the device have a channel number specified by the user. This channel assignment will then be set up in the device by sending it the appropriate message as described in the previous section.

• The *addressCategoryOffsets* variable is a Dictionary. It contains user specified address offsets (taken from the manufacturer's specifications for devices of this type) for this slot, for each address category. For example, in Roland 'D110' devices, there are eight slots which can hold instantiated units of the 'PART' category (see 8.5.2.3). Hence in the corresponding 'D110' DeviceType object, there will be a DTSMTCategory object named 'PART'. This will contain eight DCIPrimitiveSlots with ids of '1', '2', ... '8', and hence constructed names of 'PART-1', 'PART-2' etc. Each slot has an *addressCategoryOffsets* (Dictionary object) which contains address offset values for that slot for the 'TONE' and 'TIMBRE' address categories in the device.

8.6. Module types

To recap section 8.4.3, a DCT (see figure 27) is constructed out of DCTSubModule objects. Each DCTSubModule represents a synthesis structure in a device, consisting of one or more synthesis units, which may be linked (if the device allows such linking).

Each DCTSubModule object is created using a module *type* object as a template. These module types are described by PrimSMT or SMT objects (which are presented in detail in this section).

Note that this process of 'instantiation' is *not* the object-oriented instantiation of an object from a class definition. Here, both module type *and* module are objects. The user will create a new 'module' object using a 'module type' object as a template. The 'module' object will refer to information in the 'module type' object, but the latter is not a Smalltalk class description; thus a module can be created from it (via a user-interface[1]) *without* programming in Smalltalk.

Each DeviceType object will then contain a library of such 'module type' objects, each of which defines a kind of structure within a certain type of device in a consistent manner. Structures within *different* types of device are presented as being constructed in the same way out of primitive - possibly connected - units which have data inputs.

As stated above, these module types are used to create modules (DCTSubModules) within a DCT object, which then describes a synthesis structure as part of an Instrument definition. These modules can *also* be used to construct more complex module types, which are also stored in the DeviceType's library for later use in constructing DCTs or further higher-level modules. This latter use is described in detail in 8.6.4.

8.6.1 PrimSMTs and SMTs

The specification of each type of synthesis unit is described in E-Scape by a PrimSMT or SMT object[2]. A PrimSMT object describes basic (primitive) entities within a device which cannot be broken down any further. An SMT object is built up out of child submodules, which may be based either on PrimSMTs or other SMTs. It describes a higher-level entity within a device. Note that either object can describe a *unit* in a device.

The lowest-level module types in a DeviceType are described by PrimSMT objects. They describe the lowest-level primitive modules in a device which can exist as addressable entities. However, even though a PrimSMT describes an entity which has an existence within a device, this entity may *not* necessarily be able to be instantiated or run ('started') as

---

1 This user-interface has not yet been implemented, but as has been stated elsewhere, the functionality which allows a user to request a new module by specifying arguments to (existing) Smalltalk functions is present.

2 Etymological note: 'SMT' is an abbreviation for SubModuleType. A 'PrimSMT' is a 'Primitive SMT'.

an *independent* entity (a unit) within the device. If this is the case, then the PrimSMT (and the entity in the device which it describes) is termed 'below device-level', and only serves to enable higher-level entities to be constructed.

For example, in the 'D110' DeviceType, a PrimSMT in the category 'PARTIAL' has address offsets and describes an identifiable entity within the device. However this PrimSMT is *below* device-level, ie the entity it describes in the device cannot exist alone, but only as part of a higher-level unit in the device. Thus, this PrimSMT serves just as a *component* of a higher-level SMT in the 'PART' category. This 'PART' SMT *does* describe an independently startable unit in the D110 device.

A PrimSMT object thus defines *all or part of* a primitive synthesis unit which can be instantiated on a particular type of device. It includes such things as creation codes, address offsets, and specifications of its data inputs. These are described by further PrimSMTInput objects which also have address offsets, allowable value ranges, and (optionally) a default input value. A PrimSMT must, by definition, be specified by a user directly, using published information from the device manufacturer's or designer's manual (as described in the next section 8.5.5.2). Such user-specification of these PrimSMT objects would normally be done by a user as part of the process of creating a new DeviceType object, in which they are then stored.

8.6.2 Direct user specification of module types

There are two main elements to the direct user-specification of a module type: defining the properties of its inputs, and specifying parameters such as a 'creation code' and address offsets. These two aspects are now described.

8.6.2.1 Defining module type inputs

Various properties are defined by a user for a module type's inputs. Each input is described by an SMTInput or PrimSMTInput object, which defines the characteristics of the unit input which they are describing. Their structure is illustrated in figure 36 below, with instance variables again shown on the left.

SMTInput / PrimSMTInput	
rate	- 'k', 'i', or 'e'
defaultValue	- a Number
allowedValues	- a QuantisationMap object
parameterMessagePrototypes	- a Set of MessagePrototype objects
parameterOffsetAddress	- an EAddress object

Figure 36. SMTInput and PrimSMT object structure

These instance variables are now described:

• The *rate* is a symbol indicating *when* during an event (ie after the unit has started running) the unit is able to respond to changes in values sent to this input. The rate can be 'i', 'e' or 'k':-

A rate of 'i' indicates that only a single value can be specified at the start of an event, ie when a device unit which corresponds to this module type starts running.

A rate of 'e' indicates that a value can be specified which affects the manner in which a unit *stops* running, ie a parameters which affects the manner of cessation of an event.

A rate of 'k' indicates that values can be sent to this input at any time during the running of a unit, ie during the course of an event.

• The *defaultValue* must be specified, and will then propagate upwards through any module or DCT structure in which this module type is embedded (if not overridden by a designer of the structure). This default value will ultimately appear as a default Psp value, which can then result in a default parameter value for ScoreEvents. Thus, at each level of structure a designer *may*, but *need* not specify a default value. Only the user who is *directly* specifying a lowest-level module type (as here) is *required* to specify a default value.

• The *allowedValues* is a QuantisationMap object, which specifies the maximum and minimum values which may be sent to this input, along with a resolution of values between these limits. It also provides the facility for non-contiguous or non-linear *ranges* of allowed values to be specified. Again, as for the default value, these allowed ranges can propagate upwards through module structures, and must only be *compulsorily* specified at this lowest level.

• The *parameterMessagePrototypes* is a set of one or more MessagePrototypes. These define the format of messages which will be used to convey values to this input. MessagePrototypes are explained in detail in 8.8.2.

• The *parameterOffsetAddress* is an EAddress object (as described in 8.5.3.1). It is used if the device provides access to its components via addresses (rather than via dynamically allocated id numbers). It provides the address offset of an input within (ie relative to) its parent module type. If this module type is then used to create a new submodule within a *higher-level* module type (see next section for example), the new submodule will have an address offset within the high-level module type, and this will be added to the offset of the input.

The EAddress object also specifies an address *category* within the device. This address category should also be defined for any device slots (DCIPrimitiveSlot objects) within which units (created according to this module type specification) are to be instantiated.

For example, as described above, the 'D110' DeviceType has 'TIMBRE' and 'TONE' address categories, amongst others. Each of its DCIPrimitiveSlots (describing a slot in devices of this type) have an address offset for each of these address categories. Some

inputs[1] of module types in this DeviceType also have an address offset (in one of these address categories). When a unit is instantiated in a device corresponding to such a module type specification, the unit is placed in a particular device slot. The address which is to be used by messages communicating with the inputs of this unit in the device must then be determined. This requires the address offset of the module type input (which describes this unit input) to be added to the offset of this device slot *in the same address category.*

Thus for example, a 'PART' SMT has an input named 'PARTIAL-1<TVF RESONANCE'. This input has an address offset of hex 00 26 in address category 'TONE'. When a unit of this type is installed in, say, the 'PART-2' slot of the 'D110' device, its address offset in the same 'TONE' category (hex 01 76) will be added. Finally the DeviceType's base address for the 'TONE' address category (hex 04 00 00) will be added to provide the full address for this input of *this* unit installed in *this* slot.

It can be seen that complexity is arising in the design, in order to cope with the fixed address scheme and often arbitrary organisation and restrictions which appear within MIDI-based commercial synthesiser devices, and their non object-oriented structure.

8.6.2.2 Defining module type creation codes and address offsets

A module type is used to build a DCT structure, which is part of an Instrument definition. When an Instrument is used by a ScoreEvent, the unit(s) described by the module type must be *installed* in an appropriate device. In order to install (instantiate) a unit or set of units in a device, a message will need to be sent to the device. This message will almost certainly need to contain some kind of id number which identifies to the device the *type* of unit or structure to be installed in it. Such a code number may therefore be specified for a module type, which can later be used to construct such a message (see 9.3.2).

If objects in a device are referenced using addresses within its memory map (rather than via object ids), then the module type will require an address offset to be defined, along with an address category, as above. When a unit is to be instantiated within a particular slot in the device, a message will be formulated in E-Scape to be sent to the device. The absolute address in the device to which to send this message can then be calculated in the same way as for the inputs of the module type as described above.

Note that a device which references units by ids will not need such address offsets, nor will it if units are instantiated using messages which only use the 'channel' parameter of a device slot.

---

1 Those inputs of a synthesis unit which are defined as using messages which require device address information for one or more of their fields.

8.6.2.3 Specifying a didactic internal structure description

As stated above, a synthesis unit which is described by a *PrimSMT* module type cannot be broken down further into addressable entities within the device. However, when defining such a module type, a user may choose to describe and illustrate the conceptual structure of the processes occurring internally inside the unit (when it is running within the device).

This user may *not* be the same person as a composer who may later want to use the module type to construct an Instrument. Such a description would thus typically be created for didactic purposes, enabling a composer or other user who is not familiar with a unit on a device to understand what the unit does and what processes its inputs are notionally connected to inside it.

This internal structure is described using a network of notional lower-level 'virtual' sub-modules nested within it. These 'virtual' sub-modules are described by VPrimSubModule objects which have *no* corresponding actual entity within the device, ie the unit in the device has not been *constructed* from such lower-level entities, and cannot be actually broken up into them. These 'virtual' sub-modules then serve only to *elucidate* the sub-processes occurring in the device within the structure described by the PrimSMT object, although access to these processes is not available to the user.

Each VPrimSubModule is a notional instantiation of a 'virtual' module *type* (VPrimSMT), which may itself be broken down into lower-level VPrimSubModules. VPrimSMTs can also be stored in a DeviceType, thus a VPrimSMT can be specified once, then used to create modules within many different PrimSMTs.

Although a VPrimSubModule object does not correspond with an actual entity within a device, it may actually have 'real' *inputs* (PrimSMTInput objects) which *are* addressable within the device. Its function is then that of a *carrier* of PrimSMTInputs, which have all the usual features of such inputs (eg MessagePrototypes) and can have data values sent to them in a device. This enables device structures and data inputs to be described in terms of objects, when there are in fact no such objects in the conceptual scheme of the device, simply parameters which are sent to the device. A VPrimSubModule used in this way can thus be present at a higher level within a module type, alongside sibling (*non*-virtual) modules. This is illustrated in both of the two following examples in 8.6.4.

To summarise: there is *no unit* in the device which corresponds to a *virtual* sub-module, but the *inputs* of such a module may describe real addressable inputs within a device.

8.6.2.4 Examples of the direct user specification of module types

Example 1 - the 'PCM' PrimSMT

Examining the manufacturer's specification of the Roland 'D110' device, entities called 'PARTIALS' are described. These can be used to build up a higher-level 'PART' entity (see 8.6.4) on which notes can be played, ie which acts as a synthesis *unit* (as defined in this thesis). There are two different kinds of PARTIAL entity: one derives its audio data from PCM samples; the other by filtering a rectangle wave with variable pulse width.

Casting this situation in terms of E-Scape's object-oriented module structure, two PrimSMT objects can be defined in E-Scape which describe these entities. Each PrimSMT is given a name which fits with the device's terminology as closely as possible (say 'PCM' and 'filt. pulse'), and are stored within an E-Scape module category (called say 'PARTIAL'[1]).

Thus, two PrimSMT objects (named 'PCM' and 'filt. pulse') are to defined in a category named 'PARTIAL' in a DeviceType named 'D110'. This example looks at the specification by the user of the 'PCM' PrimSMT. As described in 8.6.2, this consists of two main elements:

(a) Defining module type inputs

This 'PCM' PrimSMT has several dozen inputs. Each input is defined by an PrimSMTInput object, as described above, which are created by the user (eventually via a graphic user interface). The user first creates a new blank input, then specifies its parameters using information from the device manufacturer's handbook.

• For example, the input named: 'TVF CUTOFF FREQUENCY' has the following information:

rate = 'k'.

defaultValue = 50.

allowedValues = 0 - 100 in steps of 1.

parameterOffsetAddress = hex 00 17 in address category 'TONE'.

parameterMessagePrototypes = a single MessagePrototype object of MessageType 'MIDI: Roland sys exc - D series univ' - this will be detailed in section 8.7

The name of this input is taken from the manufacturer' terminology, which can be unhelpfully long or technical; hence an input can also be given a more concise *userLabel*, in this case 'filter freq.'

(b) Defining module type creation codes and address offsets

The creation code and offset address is set to nil. This is because this PrimSMT is *below* "device-level" (as described above), and hence cannot describe a unit entity within a device,

---

1 Capitals are used by convention in E-Scape to denote a standard name in the manufacturer's device specification.

or be used to instantiate it. A higher-level module which contains this one will have creation codes which may depend on which submodules it is made up of, but these codes are not present at *this* level.

Example 2 - the '[pitch var. range scaler]' VPrimSMT

A 'PART' unit in a 'D110' can be considered to be built up from lower-level 'sub-module' entities, as described in example 1. However, it also has some inputs which do not correlate with any such submodules, but which can be described as being owned by a 'virtual' sub-module within the 'PART' unit. Thus entities within devices which are *not* wholly built of objects can nevertheless be described within E-Scape as if they are.

As stated in 8.6.2.3, a 'virtual' module type (a VPrimSMT object) does not correspond to a nameable entity within a device and cannot be instantiated in it. However, it is functionally present if describing the structure within a module in an object-oriented manner. This example looks at the user definition of the '[pitch var. range scaler]' [1] VPrimSMT. This VPrimSMT will later be used to built an SMT (in 8.5.5.3 example 2), and its function will then become clearer.

As before, a *userLabel* will be used in subsequent reference to this VPrimSMT object, rather than the less friendly '[pitch var. range scaler]' system name. This label is 'pitch scaler'.

Being virtual, it has no creation codes or addresses, functioning only as a carrier for inputs. Thus only definition stage (a) is relevant for this module. Its inputs (PrimSMTInput objects) are 'real' however, with all the usual features of such inputs, and are addressable within the device.

(a) Defining module type inputs

This 'pitch scaler' VPrimSMT has two inputs. Each input is defined by an PrimSMTInput object (as in example 1), which defines a parameter input in the device. The two inputs are named 'BENDER RANGE' and 'pitch bend amount'.

• The input named 'pitchbend amount' has the following information specified for its instance variables:

rate = 'k'.

defaultValue = 64.

allowedValues = 0 - 127 in steps of 1.

parameterOffsetAddress = nil, as this input has a message which does not utilise a device address. Instead it uses the *'channel'* parameter of the device slot in which the parent 'PART' SMT is instantiated.

parameterMessagePrototypes = a single MessagePrototype object which has a MessageType named 'MIDI: pitch bend' - this will be detailed in 8.7.

---

1 Square brackets [] surrounding a name denote it to be a *virtual* VPrimSubModule.

• The input named 'BENDER RANGE' has the following information

rate = 'i', thus can only be sent as the start of an event.

defaultValue = 12.

allowedValues = 0 - 36 in steps of 1

parameterOffsetAddress = nil, similarly to the 'pitchbend amount' input.

parameterMessagePrototypes = a Set of two MessagePrototype objects which have the MessageType named 'MIDI: controller' - this will also be detailed in 8.7.

8.6.3 Naming of objects in E-Scape

Some mention of system names and user labels has been made up to this point, but this section aims to consolidate this topic.

'System names' for many objects in E-Scape are derived automatically according to their relationship with other E-Scape objects. For example:

A module *input* uses the name(s) of its destination(s) and its parent module to build up its system name.

A module *type* uses the names of its constituent child modules to build up its system name.

Whenever available, names and ids should follow those used by a device's manufacturer, and the convention is employed of using upper case for names which appear in the device's manual, with lower case used for additional names which pertain to structures in E-Scape which have no named direct corollary in the device.

In the examples in the following section, note the way that system names are built up:

• The name of a *module type* includes the names of its child submodules, with an indication their relationship:

The '->' character indicates a connection between two modules.

The '/' character between two module names indicates that both modules have an equivalent relationship with the following symbol.

For example, the name 'PCM-A / PCM-B => mix' for a module type would indicate that, within the module type, both the 'PCM-A' and 'PCM-B' child modules are connected to the 'mix' module.

• The name of each *input* is built up using the name of the ('destination') child submodule the input is connected to within the module type, the '<' character, and the name of the input on this destination submodule.

As can be seen, these 'system names' enable a knowledgeable user to see the construction, derivation or connections of an object by interpreting the name, but this often results in a rather long name for normal display purposes. Two strategies are employed to reduce this complexity, if desired:

• Macros can be employed, eg the name 'Z' can be substituted for the above module type name, to produce a shortened system name.

• An additional *userLabel* instance variable can be specified, and can optionally be used and displayed instead of the full or shortened E-Scape derived system name.

A *userLabel* has been already used in the previous example - the '[pitch var. range scaler]' module is referred to as 'pitch scaler'. In the subsequent examples, userLabels are employed for clarity, although in the diagrams the original system names are shown faintly, to give an impression of their structure and verbosity. A user who is involved with designing module structures would probable want to display these names, as they do provide information about the origin and content of objects. Most users, however, will prefer to use the shorter user labels for referring to objects.

8.6.4 Building module types from lower-level modules

An module type (SMT) may also be built by a user out of existing, lower-level modules. The new module type can then 'inherit' various characteristics of these lower-level modules, rather than having them defined directly by the user. Two examples are now described to illustrate this process.

8.6.4.1 Building module types from lower-level modules - example 1

The example below shows how an SMT of category 'PART' can be built by a user out of lower-level modules. There are 380 possible SMTs in the 'PART' category of the 'D110' device, which would all have to be specified by a user individually, if the facility to allow them to be constructed from lower-level modules were not provided.

Each module is created according to a lower-level module *type*, (another SMT or PrimSMT object) which is stored in a category of the DeviceType object which describes the 'D110' device. These lower-level module types are actually *below* device-level because, although they exist as addressable entities with the 'D110' device, they cannot be started and stopped as independent entities, and only serve to enable higher-level structures to be created.

In this example, an SMT is built from three submodules, two of type 'PCM', and one of type '[pmix]'. The latter is a *virtual* module type, and does not correspond to a real entity in the device. It can have real inputs however, as in this case (see 8.6.2.3).

Figure 37 below shows, at the top, the two module types which are used to create the submodules within the new SMT. These module types (in this case both PrimSMT objects) are already present within a category of the DeviceType, having been specified earlier.

Two modules are then created using the 'PCM' module type; with each being given an *id* to distinguish them. These ids can be any symbol, but in this case are chosen to be '(1/3)' and '(2/4)' respectively to mirror the naming structure used by the device itself.

A third module is created from the '[pmix]' module type. This need not be given an id, as there is only *one* module present of this type. These submodules are then nested within a new higher-level module type (an SMT object). This new SMT is in the 'STRUCTURE' category, can then be used subsequently as a template to create new modules of this category.

In the figure, some of the more lengthy E-Scape system names are shown only in faint text, and have been substituted by shorter user labels. This new SMT (the lower half of the figure) is labelled 'structure 7', and will be used in the next example.

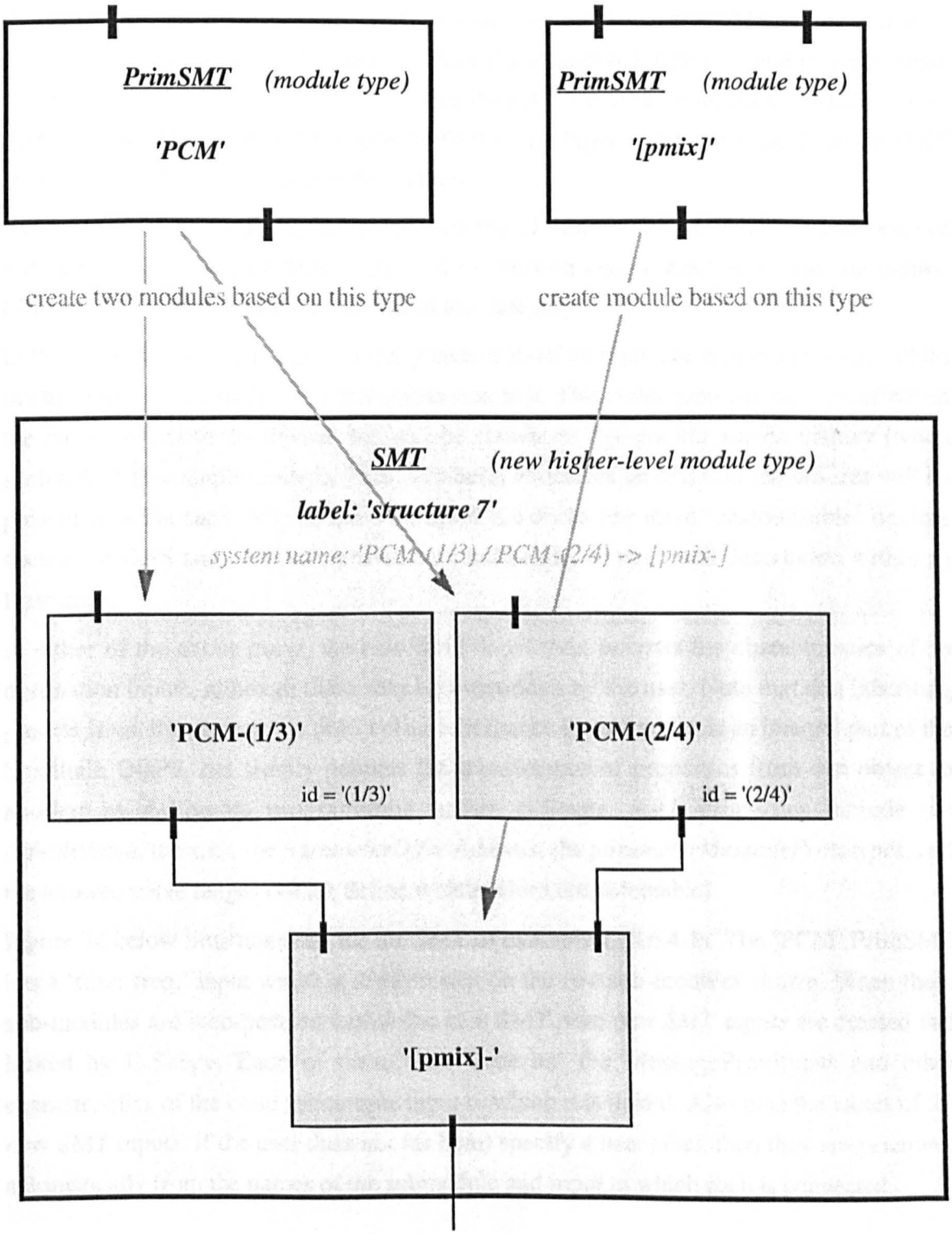


Fig. 37 Constructing a new SMT from lower-level modules - example 1

8.6.4.2 Inheritance by module types of submodule characteristics

A new higher-level SMT object will have inputs. If the device supports the specification of networks via declared nodes (eg CSound) or module inputs (eg MIDAS), this fact will be noted in the DeviceType description object for the device.

If this is the case, the user may overtly create each of these new SMT inputs and then specify a connection from it to one or more ('destination') inputs of one or more child submodules within the new SMT. If the user does *not* specify a connection to a child input, then the new SMT automatically creates itself a new input which corresponds to the child input, and installs a connection between them.

Alternatively the device may allow the building of modules out of child submodules, but *not* support flexibility of structuring, and of creation and connection of module inputs. Most MIDI-based synthesis devices are in this category.

In this case, the new SMT automatically creates itself an input corresponding to each child module input, and installs a notional connection to it. This connection will have no effect on the message sent to the device, but will be shown on any graphic screen display (when such a display is implemented). Thus, synthesis structures on MIDI-based devices will be presented in the same way as more complex networks (on more 'customisable' devices such as MIDAS and CSound), providing a uniformity of structural description within an Instrument.

In either of the above cases, the new SMT input then *inherits* the characteristics of its destination inputs, although these may be overridden by the user. Note that this inheriting process is *not* the same as the object class inheritance feature which is an integral part of the Smalltalk OOPS, but simply denotes the transference of properties from one object to another by deliberate programming within E-Scape. Such properties include the *defaultValue*, the *rate*, the *parameterOffsetAddress*, the *parameterMessagePrototypes*, and the allowed value ranges (which define which values are acceptable).

Figure 38 below illustrates this for the SMT of example 1 (8.6.4.1). The 'PCM' PrimSMT has a 'filter freq.' input which is then present on the two sub-modules shown. When these sub-modules are incorporated within the new SMT, two new *SMT inputs* are created and linked by E-Scape. Each of these then 'inherits' the MessagePrototypes and other characteristics of the child submodule input to which it is linked. Also note the labels of the new SMT inputs: if the user does not (as here) specify a user label, then they are generated automatically from the names of the submodule and input to which each is connected.

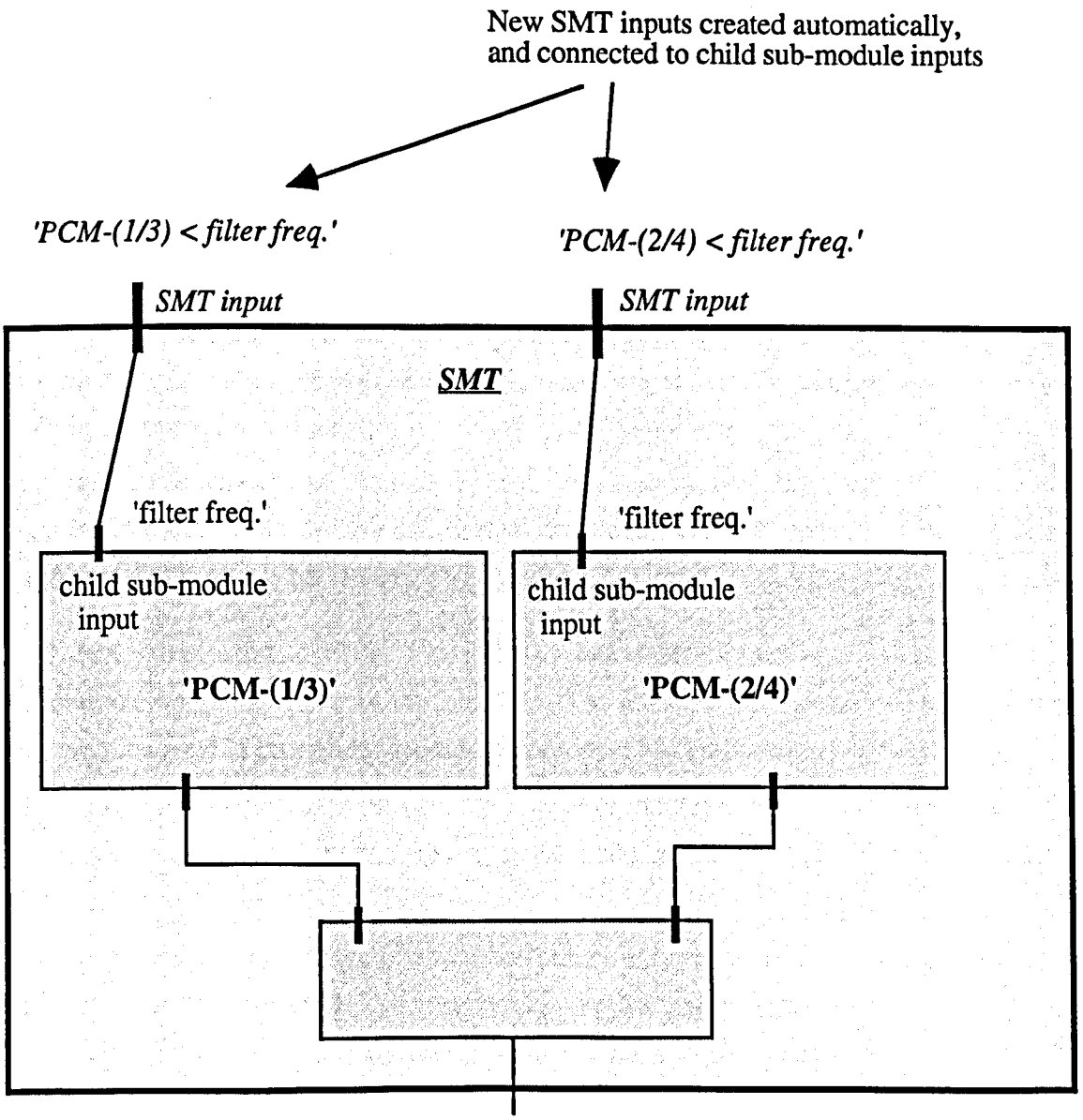


Fig. 38 The automatic creation of new SMT inputs, with connection to sub-modules

The transference of information from destination input to the new SMT input is not necessarily direct; in various cases, more involved operations are required:

• The destination input's *parameterOffsetAddress* is added to the offset address of the child submodule which owns it.

• If an SMT input is connected to more than one destination input, then the allowed value ranges are collated and the minimum range selected. If two or more destination inputs have *parameterOffsetAddresses* which are different this will cause an error. However, in practice devices *either* follow a flexible approach with inputs overtly created and freely connected using object ids to reference them (ie a having a dynamic indirect memory map), *or* have a fixed memory map, using address offsets (in addition to channel-based indirection), which does *not* allow creation of inputs and connection of units into networks

• A user (designer of the new SMT) may choose to restrict the range of the higher-level input, compared with the destination's range. Note that the destination's range may not be

exceeded, thus a designer is constrained to the allowable bounds of the values which may be sent to a unit input. A designer may often wish to so restrict the higher-level input range, in the light of its interaction with other input values. Thus, for example a CSound 'OSCIL' unit may have a range of 0-35535 on its 'A' input, but when installed inside a higher-level SMT in an FM configuration, values of more than 23400 may cause an unwanted distortion in output; thus the designer can restrict data sent to the SMT input (connected to the OSCIL inside it) to this range.

Thus all SMTs can safely be used at a higher level without a user needing necessarily to know what is inside. More complex restrictions on data values can be imposed than with a simple limit, as explained in 6.1.4.

In this manner, a user can create new SMTs in a particular category from child modules, although there is nothing to stop a user defining an SMT at any level 'manually' - ie by specifying its characteristics 'from scratch', as in 8.6.2.

8.6.4.3 Building module types from lower-level modules - example 2

A second example uses the (built in example 1) as a template to create two submodules. These are then used in constructing a new higher-level SMT of category 'PART'.

This example SMT is actually built from *four* submodules: two of the afore-mentioned 'structure 7' module type, and two *virtual* submodules, of type 'pitch scaler' and type 'volume scaler'.

Figure 39 below shows (at the top) the three module types which are used to create the submodules within the new SMT. These module types are already present within a category of the DeviceType, having been specified earlier, and are shown as different shaped for clarity.

Two modules are then created using the 'structure 7' module type (created in example 1), with each being given an *id* to distinguish them. As in example 1, the ids match those used by the device: in this case '(1&2)' and '(3&4)'.

The third and fourth (virtual) modules are created from the '[pitch var. range scaler]' and '[output scaler]' module types respectively. These need not be given an id, as there is only *one* module present of each type.

These four new submodules are then nested within a new structure which defines a new higher-level SMT object in the 'PART' category.

This new SMT can then be used subsequently as a template to create new submodules of this 'PART' category. In this case the 'PART' category *is* "device-level", ie module types of this category can describe a *unit* within a device. Hence as well as being usable to define still higher-level SMT structures, this SMT can also be used to build a DCT object (see above, section 8.4.3.1).

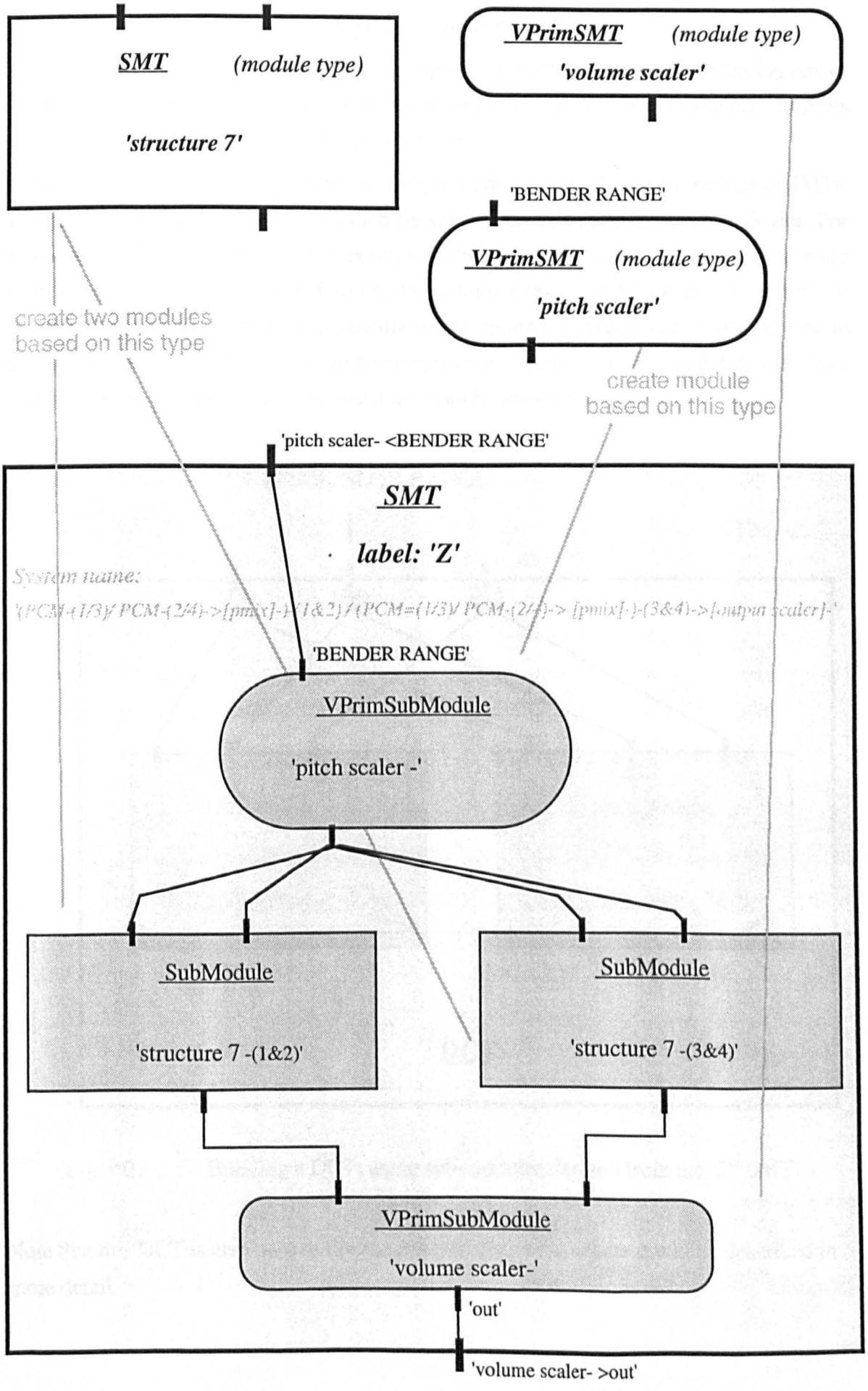


Fig. 39 Constructing a new SMT from lower-level modules - example 2

8.6.5 Using module types to build DCTs

To recap: a DeviceType can have several categories of module type, each containing one or more SMT or PrimSMT objects which can describe synthesis units or networks of units which may be installed on a certain type of device.

To build an Instrument, one or more DCT objects are installed. These are similar to SMTs, except that they describe structures which are specifically to be instantiated on a device. For example, the 'Z' SMT described in example 2 above (8.6.4.3), could then be used to build a DCT (as well as being used to build further - higher-level - SMT structures). Figure 40 below illustrates this, with two sub-modules being created according to the specification in the 'Z' module type. This figure can be compared to figure 27 in section 8.4.3, and it can thus be seen how SMTs fit into the structure of an E-Scape Instrument .

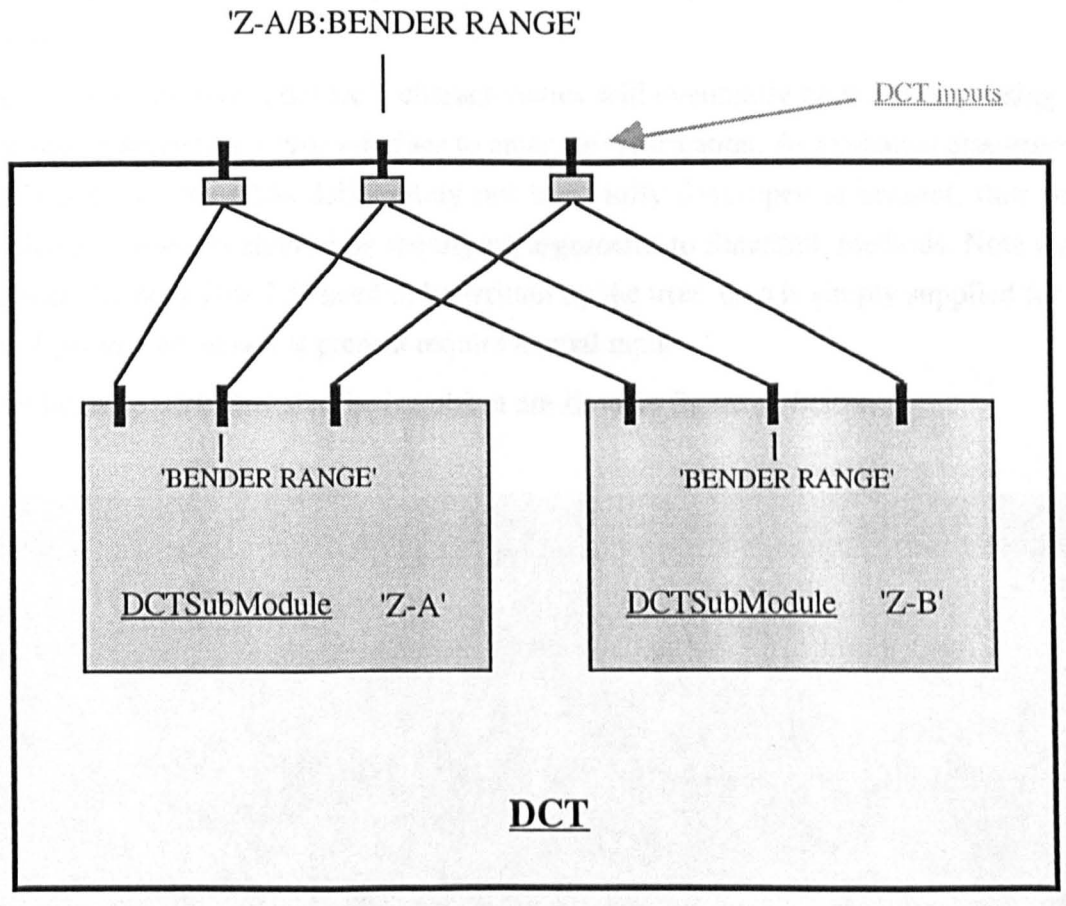


Fig. 40 Building a DCT, using sub-modules derived from the 'Z' SMT

Note that this DCT is also used in the examples in chapter 9, where it will be described in more detail.

8.7. Devices

When a network of synthesis devices is connected to E-Scape, this network must be described. When a device of a certain type is specified by a user as being connected and *in use* in E-Scape, then a corresponding Device object will need to be created. A *template* DeviceType object must have already been created (as described in 8.5) for each *type* of device which might be used. This contains details of the slots (if any) in devices of this type, and the types of synthesis unit which can be installed in them.

Many of the new Device's specifications will then be taken from this DeviceType object, but many other set-up parameters will be particular to each device, and these are specified by a user who is adding the device to the system. Such parameters will typically include such things as the device's id, any id or address settings for its slots, or any 'mode' settings for the device (ie how its operation, structure or responses are configured, if there is a choice).

This user specification of a device's characteristics will eventually be performed using a dialogue box or spreadsheet type interface to enter the information. As explained elsewhere, E-Scape's user interface has deliberately not been fully developed at present; thus the information at *present* is entered by specifying arguments to Smalltalk methods. Note that no Smalltalk methods (see 7.5) need to be written by the user: data is simply supplied for a number of parameters which at present require textual input.

The chief instance variables of a Device object are show in figure 41 below.

Device	
name	- automatically derived from the name of its DeviceType and its *id*.
id	- a number identifying this device (only needed if more than one of the same type are present).
modelType	- the DeviceType of which this Device is an example.
deviceSlots	- a Dictionary containing DeviceDCIPrimitiveSlot objects.

Fig. 41 Device object class definition

A Device object contains DeviceDCIPrimitiveSlots, which describe the available slots (locations) within the device within which synthesis units may be instantiated. Each

DeviceDCIPrimitiveSlot derives information from the corresponding DCIPrimitiveSlot in each category of the DeviceType (see 8.5.3.1). Each DeviceDCIPrimitiveSlot has a *channel* and *channelAssignCode*, and *channelAssignAddress*.

Figure 42 below shows two Device objects being created from a template DeviceType object. The channel number of each slot in this device may be specified by a user, hence the DeviceType's slots (DCIPrimitiveSlot objects) do *not* have a channel specified. They do, however, have a *channelAssignOffsetAddress*, and this is shown in the figure for the first slot (at the top of the DeviceType illustrated). As described in 8.5.3.1, the address offset has a value (in this case hex 0D), and an address *category* within the address map of this type of device. The DeviceType has a base address for each of these address categories; in this example the 'SYSTEM' category has a base address of hex 10 00 00 (not shown in the figure). When a slot (DeviceDCIPrimitive object) in the Device (shown on the right hand side) is created, it accesses the address offset of its parent slot in the DeviceType (shown on the left). This offset is then added to the base address for this category to give the Device slot its absolute address. This address value will then be used when constructing a message which is sent to the device to set up the assigned channel number to the slot. In the figure below, this is shown for the first slot of the Device with id 18 (bottom right):

- base address for category 'SYSTEM' in DeviceType	hex	10 00 00	+
- *channelAssignOffsetAddress* for DCIPrimitiveSlot in DeviceType	hex	0D	
	= hex	10 00 0D	

If necessary, each DeviceDCIPrimitiveSlot also creates a message (using the corresponding prototype in the DeviceType) which will be sent to the device to set it to the correct channel.

Note that these slots will *not* be present in a Device if the device employs dynamic unit allocation, addressed by ids - ie the DeviceType's *hasDynamicMap* instance variable is 'true', as described in section 8.5.2.2.

As described in 8.5.3.1, the user can specify the channel to be used for each slot in each device in a network. The specific channel must be within the allowed range for that category on the DeviceType.

If the device allows, this channel assignment can be set up in each device remotely. If not, the user must manually set up the channel on each device to match that specified in the Device description object.

This functionality could have been achieved using object-oriented subclassing, ie having a different class for each type of device. However, this would have involved the user creating a new Smalltalk subclass for each kind of device and assigning class variable to it. This would require a user to program in Smalltalk, the avoidance of which is one of the design goals of the E-Scape software. It would also have involved possible confusion, with the object class describing the device *type*'s characteristics, and the instances of that class describing the device's characteristics.

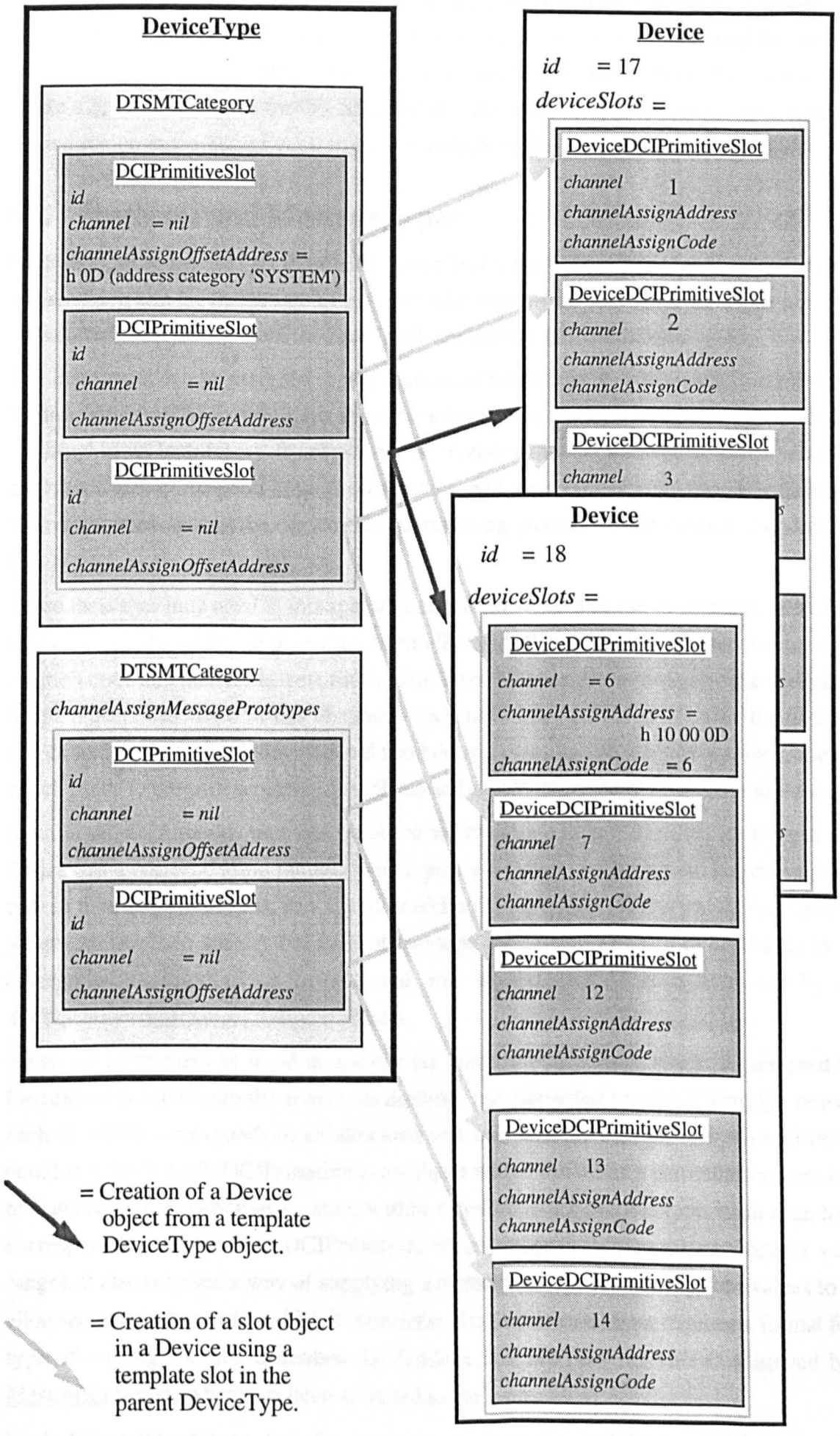


Fig. 42 Creating Device objects using a DeviceType, plus user-specified information

Device objects are stored globally in the E-Scape system, and are used when a ScoreEvent is installed within an SSE (SuperScoreEvent) structure, and is allocated the device resources (as specified in its Instrument) necessary to perform it. Each Device will then create a ScheduledDevice for this SSE to hold *allocations* of the requisite units (with the same duration as the ScoreEvent) in its slot. Details of this are given below in 9.3.2.

8.8. Protocols and MessageTypes

Each module type (SMT or PrimSMT) describes a possible entity (a unit or network of units) which can be instantiated in a particular type of device. An Instrument object is constructed from one or more modules, each having one of these module *types*.

The Instrument can be assigned to any number of ScoreEvents. When each ScoreEvent is loaded within an SSE, resources must be allocated in devices to realise the structures specified in an Instrument. If necessary, the modules may be unravelled to the primitive modules which correspond to units on a device, and the requisite units can be instantiated by sending messages to the device. This unravelling process will be described in detail in 9.3.1.

These messages may need to incorporate fields for data such as the id, channel, or address of the device slot in which a unit has been allocated, the id and type of the device, and a creation code and address offsets for the unit type. This data is provided by the various E-Scape objects described in this chapter, which have been set up to describe the unit, the device and its slots. Each device-level module type thus has MessagePrototypes, each of which needs to describe a message, its fields and where their data is to be derived from.

In addition, each device unit has inputs to which data can be sent. Each unit input in a device has a corresponding module type input in E-Scape, which specifies the range and type of data that can be sent, and any id or address information for that input. Each module input also needs to specify the type of message which will be used to send data to the corresponding input of an instantiated unit in a device. This is achieved by the aforementioned MessagePrototype objects.

When an Instrument is *used* in a score (ie when a ScoreEvent which is assigned the Instrument is added into the score), its modules are converted into DCIPrimitive objects, each of which corresponds to an allocated unit in a device. This process is described in detail in 9.3.4.3. Each DCIPrimitive is an object which describes a particular *instantiation* of a module, in a particular location within a device. Each module type input then has a corresponding input in the DCIPrimitive, which inherits its parameters such as value ranges. It also requires a way of supplying a message which can convey data values to the allocated unit in the device which it represents. To do this each input requires a format for a type of message, which describes the fields which are present. This is achieved by a MessagePrototype which has been assigned to the input.

Each MessagePrototype has (ie owns as an instance variable) a particular parent MessageType object (in which it is stored).

A MessagePrototype can reference its parent MessageType object which tells it what fields it must assemble, if (and how) any of those fields are derived from other data or other fields, and how data might need to be encoded for transmission (for example, putting additional formatting characters between field packets). A MessagePrototype has additional information specified about what *type* of object within E-Scape should supply data for each piece of message data, and what Smalltalk message (selected from a standard list) will be used.

These MessageTypes in turn are held in a Protocol object which is the top-level storage structure for types of message of a particular kind. Thus, looking at E-Scape's message organising structure from the top down, the highest level object is a Protocol, whose structure is illustrated below in figure 43. A Protocol contains MessageType objects and a set of the available Smalltalk classes which may be used by its messages to facilitate low-level data output from the Smalltalk environment. This may be via physical output ports, or internal connections, eg UNIX sockets.

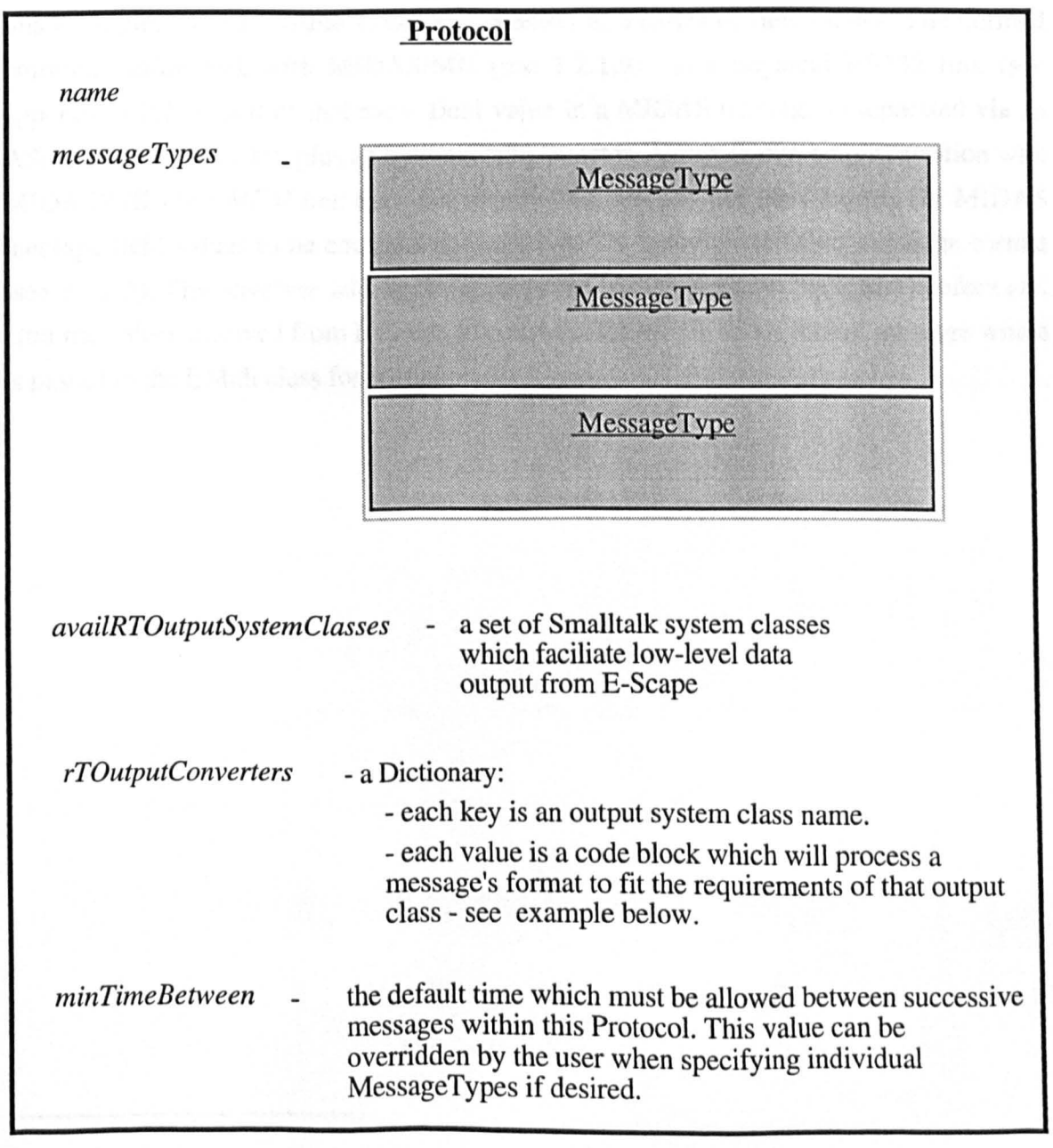


Fig. 43 Protocol object structure

The structure of a Protocol is best explained with an example. Figure 44 below shows a Protocol object named 'MIDAS'. As shown, it contains four[1] MessageTypes which serve various different functions to control UGPs (Unit Generator Processes) on the MIDAS system.

The Protocols *availRTOutputSystemClasses* instance variable has six different Smalltalk object classes assigned to it. These classes may be used to send messages out of the computer on which the E-Scape system is running: On the Macintosh computer, the ERS232 class sends data as ASCII symbols out of the computer's serial port. The EMidi class sends data as MIDI bytes, also out of this serial port, but assumes that an external MIDI hardware interface converter is attached to this port. The EMidi class has several subclasses shown below it (named EMidiMTP etc.), which each assume that a particular more specialised MIDI interface (which can address multiple MIDI connections) is attached.

The Protocol also has a instance variable called *rTOutputConverters* which is an ECodeDictionary (as in 8.4.4.2) which can contain specialised data processing operations for a message being sent out of a particular output system class (eg EMidi or ERS232). In this example, the data within a message consists of a series of field values. The defined communication link with MIDAS/MII (see 3.2.1.9) via a physical RS232 link (see appendix 1.3.2) specifies that each field value in a MIDAS message is separated via an ASCII comma character, plus an optional tab character. Alternatively, communication with MIDAS/MII via a *MIDI* link does not require commas etc, but does require the MIDAS message field values to be encapsulated into a MIDI 'system exclusive' message format (see 2.5.2.2). This involves adding the appropriate bytes (shown in the figure) before and after the values received from E-Scape to construct the whole MIDI format message which is passed to the EMidi class for output.

---

1 Note that the 'MIDAS' Protocol has many more MessageTypes than this in reality.

Protocol

name = 'MIDAS'

messageTypes =

MessageType
name = 'MIDAS: send data to UGP'

MessageType
name = 'MIDAS: create UGP'

MessageType
name = 'MIDAS: connect UGP'

MessageType
name = 'MIDAS: delete UGP'

availRTOutputSystemClasses

ERS232
EMidi
EMidiMTP
EMidiMTPII
EMidiStudio4
EMidiStudio5

rTOutputConverters

ERS232 => [code block which intersperses 'comma' and 'tab' ASCII values between message field values]

EMidi => [code block which:
i) prepends hF0, h7D, York Univ ID, and MIDAS type id bytes;
ii) appends an hF7 byte to the message field values] -
see Appendix 1.3, and section 2.5.2

minTimeBetween = 0

Fig. 44 An example Protocol named 'MIDAS'

8.8.1. MessageTypes

As described above, a MessageType object describes a message which can be sent from E-Scape to a device. Such a message consists of a separated set of named fields which can contain numeric or textual data. A MessageType object defines what fields are present in a message. Values for these fields may be supplied directly by various objects in E-Scape (such as a DCIPrimitiveSlot or Device), or may be *derived* from supplied values. Again, the use of the proposed communication standard message formats would render such complexity unnecessary. The structure of a MessageType object is shown in figure 45 below.

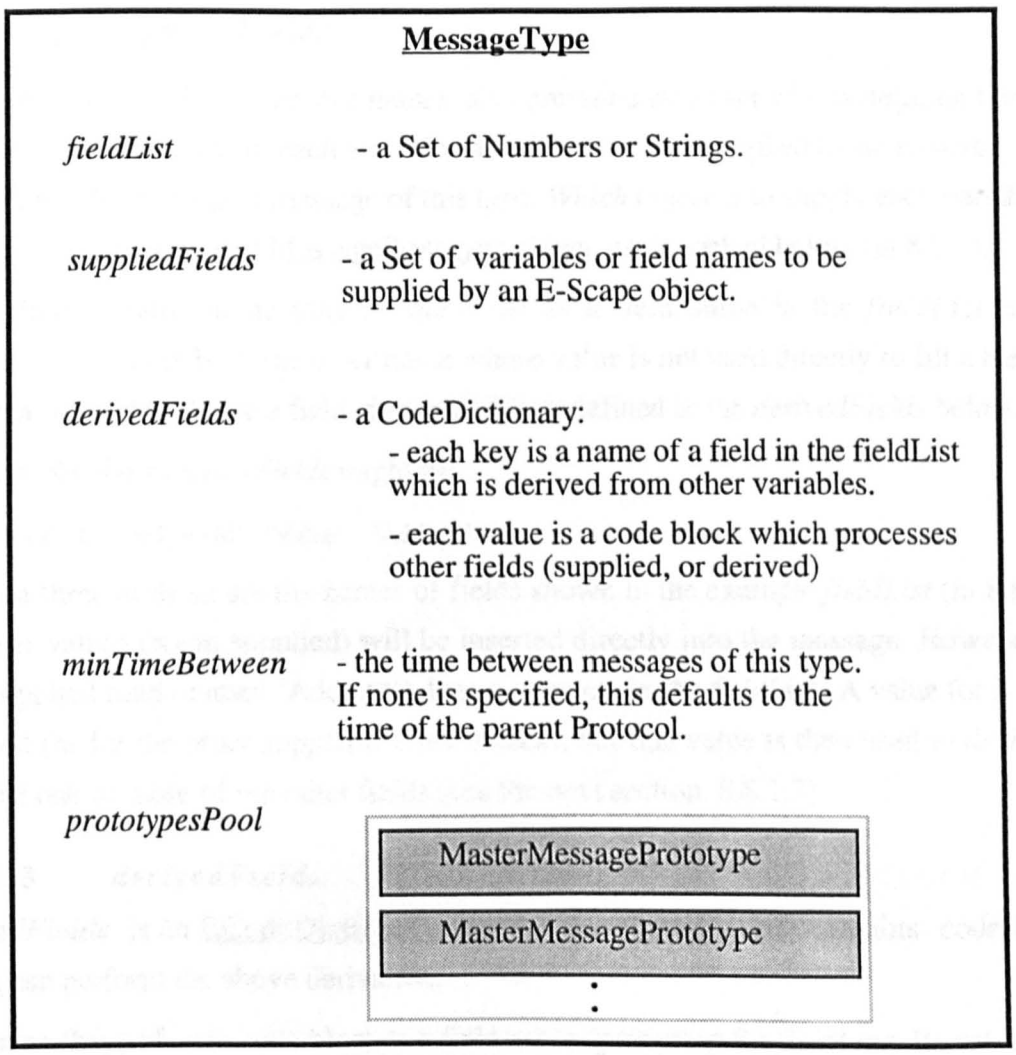


Fig. 45 MessageType object structure

As illustrated, a MessageType has the following instance variables:

8.8.1.1 *fieldList*

fieldList is a Collection, containing Numbers or Strings which are entered by the user, when defining a MessageType. The *fieldList* defines the order of the fields in a message of this type. Any strings will later be replaced by data values, while any numbers will be used directly (as is) in formulating a message. An example *fieldList* might be:

(hF0 [1] h41 'Device id' 'DeviceType id' h12 'Add1' 'Add2' 'Add3' 'Value' 'Checksum' hF7)

This list is provided by a user who is defining this MessageType within a Protocol. Each named field value will then either be *supplied* by an E-Scape object, or be *derived* from such a supplied value. Thus, each string entered here by a user must also be entered either in the *suppliedFields* or the *derivedFields* below.

8.8.1.2. *suppliedFields*

suppliedFields is a Set of variable names, also provided by a user who is defining this MessageType. The value of each variable named here will be supplied by an E-Scape object, when formulating a message of this type. *Which* object is to supply each variable (and how) is specified in a MessagePrototype object, as described below (in 8.8.2).

A supplied variable name may be the same as a field name in the *fieldList* above. Alternatively, it may be some other name whose value is not used directly to fill a message field, but is used to *derive* a field via a code block defined in the *derivedFields* below.

For example, the *suppliedFields* might be:

('Device Id' 'DeviceType Id' 'Value' 'Address')

The first three of these are the names of fields shown in the example *fieldList* (in 8.8.1.1), and their values (when supplied) will be inserted directly into the message. However, the final supplied field (named 'Address') does *not* appear in the *fieldList*. A value for it is still supplied (as for the other *suppliedFields* names), but this value is then used to *derive* the value of one or more of the other fields (see the next section, 8.8.1.3).

8.8.1.3 *derivedFields*

derivedFields is an ECodeDictionary (see 8.4.4.2 above) which contains code blocks which can perform the above derivation.

The name (key) of each code block is a field name (present in the *fieldList*). Its value is not supplied directly (hence is *not* present in the above *suppliedFields* list), but must be derived indirectly from another value which *is* supplied.

- Each dictionary key is a String which is the name of a field in the *fieldList.*
- Each value is a code block (DBlockContext object) containing Smalltalk code which can derive a value for this field using a value from another (supplied) field.

Thus a CodeDictionary could, for example, contain:

---

1 The 'h' prefix indicates a hexadecimal number

'Add1' => [:message | ((message valueOf: 'Address') bitShift: 16) & h7F]

'Add2' => [:message | ((message valueOf: 'Address') bitShift: 8) & h7F]

'Add3' => [:message | (message valueOf: 'Address') & h7F]

When a message is to be assembled, each of the named fields in the MessageType's *fieldList* is supplied with a value. If it is a *supplied* field (eg 'Device Id'), then this value will come directly from the appropriate object value. If, however, the field is a *derived* field (eg 'Add1'), then the block of code associated with it is run, and any values necessary to derive the field value are then requested (in this case 'Address').

8.8.1.4. *minTimeBetween*

minTimeBetween specifies the time to be allowed for a device to respond to successive messages of this type being sent to it. It is optionally specified by the user for each different MessageType; if not then the default time specified in the Protocol is used. This information will be used by E-Scape when scheduling messages within a score structure (see 9.3.4 for details).

8.8.1.5. *prototypesPool*

prototypesPool is a set of user-defined MasterMessagePrototype objects. These form a template to create MessagePrototypes of this type. Such a template is necessary because a MessagePrototype is *assigned* to a particular object which is then its *user* (see below). The MasterMessagePrototype describes the features of such a MessagePrototype but is abstract - ie is not assigned to any particular object, as MessagePrototypes are.

For example when a module type or its inputs are defined by a user (see 8.5.5.2 above) one or more MessagePrototypes are allocated to them. Many inputs, for example will use the same MessagePrototype, and a definition of this MessagePrototype will normally already be defined and stored in a Protocol in the form of a MasterMessagePrototype.

When a MessagePrototype is allocated to an E-Scape object by the user (assigned using a selected Protocol, MessageType and MessagePrototype name), the MessageType is then asked to provide a MessagePrototype of this name. The MessageType in turn then asks the MasterMessagePrototype with this same name (which it has stored in its *prototypesPool*) to supply a MessagePrototype. This new MessagePrototype has all the same features as its 'master' template specification, except it is *allocated* to (and thus owned by) the E-Scape object, which is assigned to its *user* instance variable.

Apart from this *user* variable, the MessagePrototype and MasterMessagePrototype objects are very similar in behaviour and structure. Thus, unsurprisingly, one is a subclass of the other, illustrating the inheritance principle described in section 7.2.1.4

This Protocol structure has been tested by defining, as a user, the MIDI specification and MIDAS protocols as Protocol and MessageType objects (see section 11.1.1.1) and successfully formulating messages for devices.

8.8.2. MessagePrototypes

A user will define one or more MessagePrototype objects as part of the process of defining a new MessageType. Each MessagePrototype represents a particular *application* of a MessageType. The MessagePrototype assigns a *specific* kind of E-Scape object which will supply the value of each field (as defined in the MessageType's *suppliedFields* (see 8.8.1.2). These values are supplied in response to particular (standard) Smalltalk messages (selected by a user from a manual, or via menus). As stated in 8.8.1.5, a MessagePrototype is defined by the user in the form of a MasterMessagePrototype object which is stored in the MessageType object (as shown in figure 45).

The structure of a MasterMessagePrototype is illustrated in figure 46 below.

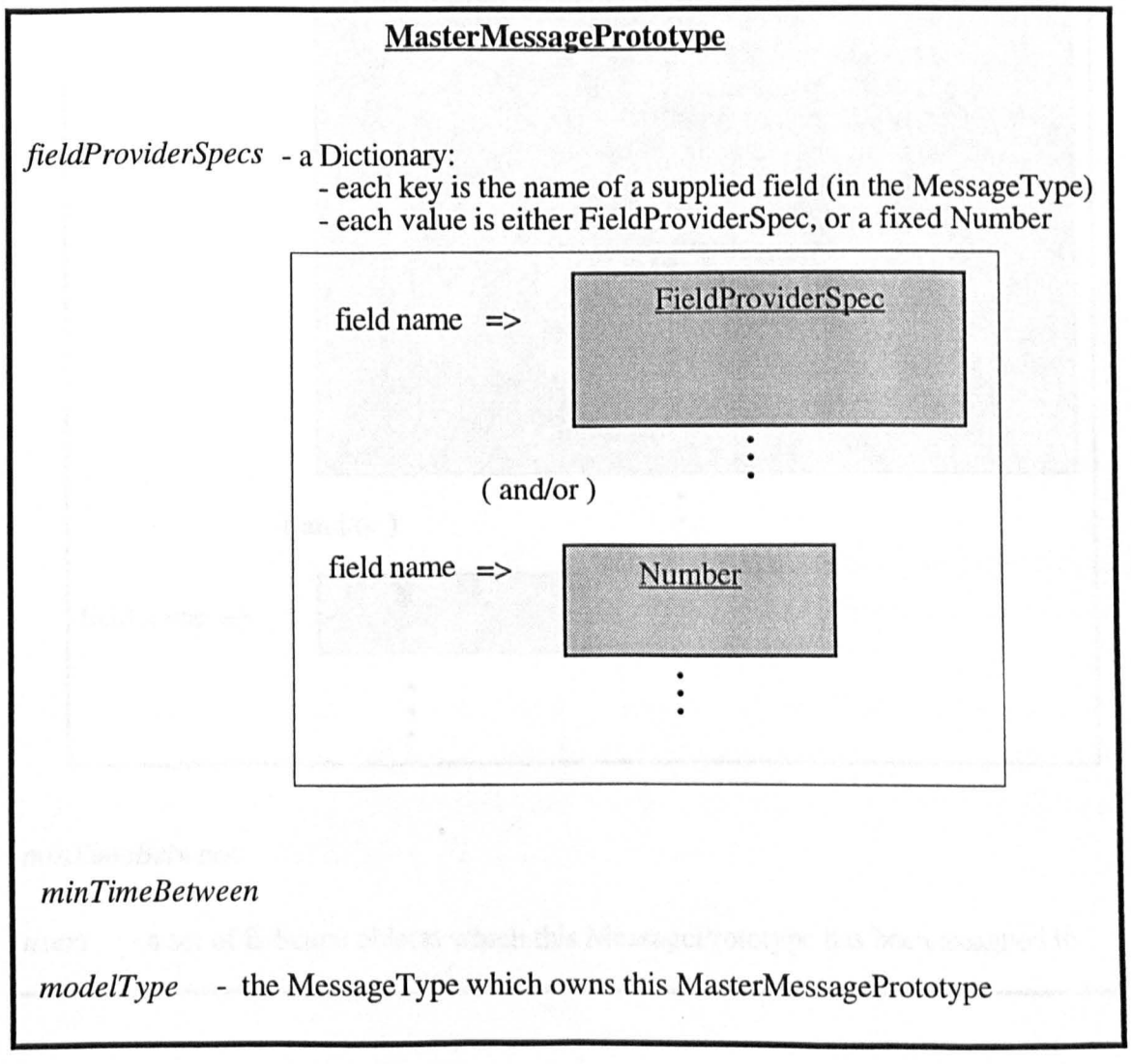


Fig. 46 MasterMessagePrototype object structure

A MasterMessagePrototype acts as an abstract template, and is used by the MessageType to create MessagePrototype objects, which are supplied to, and owned by particular objects. A MessagePrototype is thus the *same* as its template MasterMessagePrototype, apart from being *owned* by one or more objects within the E-Scape structure, which are assigned to its

users instance variable. The structure of a MessageType object is illustrated below in figure 47, with details of the FieldProviderSpec object shown.

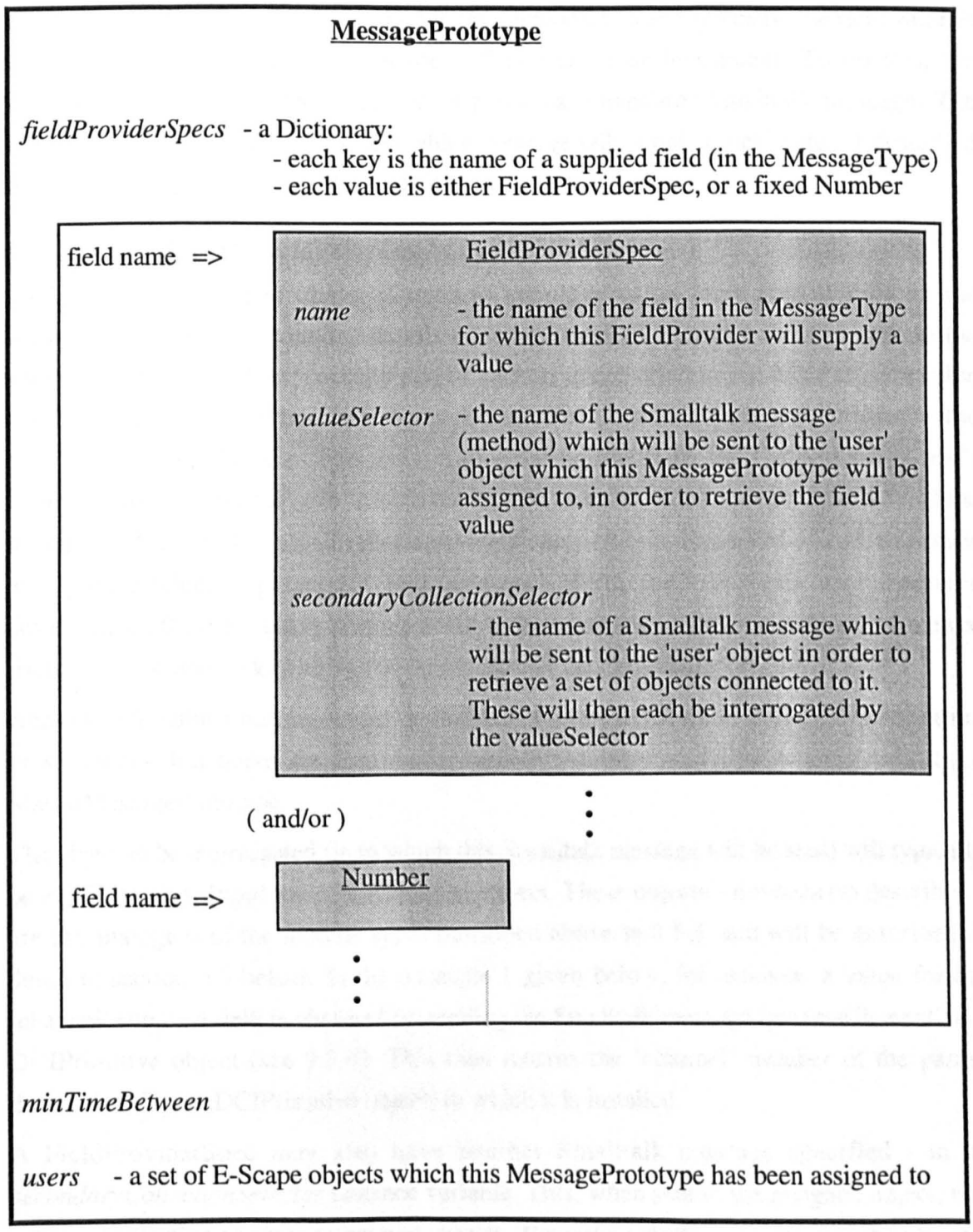


Fig. 47 MessagePrototype object structure

The chief functionality of a MessagePrototype is defined in its *fieldProviderSpecs* Dictionary (as shown in the above figure), which specifies how each message field value is to be determined. Note that this information is actually entered by a user when creating a MasterMessagePrototype, as part of the process of defining a Protocol.

This MasterMessagePrototype then creates MessagePrototype 'copies' of itself whenever a user assigns a named MessagePrototype to an E-Scape object.

A field in a MessagePrototype may have a *fixed* value[1]. If so, this field name will then have an associated (fixed) Number specified in the Dictionary. More typically, the field value is to be supplied by some object in the E-Scape score or Instrument. To do this, the designated object will need to be sent a particular (standard) Smalltalk message. The information about which object, and which message will supply a field value is described by a FieldProviderSpec object for that field name.

8.8.2.1. FieldProviderSpecs

Each FieldProviderSpec object is used to supply a value for a named field in the MessageType. It thus contains details of what Smalltalk message is to be sent to the appropriate E-Scape object (usually part of an Instrument structure) in order to get a value for the named field. When specifying a new FieldProviderSpec, a user installs the appropriate Smalltalk message name into it (assigning it to the FieldProviderSpec's *valueSelector* instance variable, shown above). To do this, the user selects from a small list of standard E-Scape (Smalltalk) messages which may be sent. Examples of such messages are 'paramValue:', 'parentsId' and 'address'. With the envisaged user interface development, these Smalltalk messages will be presented to the user via a menu, with more friendly labels, and no knowledge of Smalltalk will then be necessary.

Additional Smalltalk messages may be defined in future to take account of new parameters or structures, but these are then automatically available to a user who is defining a MasterMessagePrototype.

The object to be interrogated (ie to which this Smalltalk message will be sent) will typically be a DCIPrimitiveInput, or a DCIPrimitive object. These objects - not hitherto described - are the analogues of the module types described above in 8.5.3, and will be described in detail in section 9.3 below. In the example 1 given below, for instance, a value for the 'channel' supplied field is obtained by sending the Smalltalk message 'parentsChannel' to a DCIPrimitive object (see 9.3.4). This then returns the 'channel' number of the parent device slot (DeviceDCIPrimitive object) in which it is installed.

A FieldProviderSpec may also have another Smalltalk message specified - in its *secondaryCollectionSelector* instance variable. This, when sent to the assigned object, will return a Collection of other associated objects. The *valueSelector* message will then be sent

---

[1] This is slightly different from the fixed values which pertain to the MessageType. Such fixed values will be contained in the *fieldList* of the MessageType. Each MessagePrototype describes a kind of 'usage' of the MessageType, and different usages (ie different MessagePrototypes) may have different fixed values for some fields. The MessageType named 'controller' in the 'MIDI' Protocol is an example of this: there are many different MessagePrototypes of this type, each with a different purpose, and a different fixed value (Number) for the 'controller number' field.

to *each* of these associated objects in order to supply values for fields. A device message will be constructed for *each* of these associated objects. One common use for this *secondaryCollectionSelector* is when a module is connected to one or more others within an SMT module type (see 8.6.4.3). The user, when constructing an SMT, will create several child modules within it, which may be connected[1]. If so, the output of a 'source' module will be connected to the inputs of one or more 'destination' modules. A message will then typically need to be sent to the device to specify *each* of these connections. To facilitate this, the user will assign a MessagePrototype to each 'source' module output. Such a MessagePrototype will have 'source module id' and 'destination module id' fields (amongst others) defined in its parent MessageType.

The 'source module id' field will be supplied by sending the Smalltalk message 'id' (for example) to the module output. Thus the FieldProviderSpec for the 'source module id' field contains the *valueSelector* 'id'.

Values for the other 'destination module id' field are obtained by sending the 'parentsId' Smalltalk message. Thus, the FieldProviderSpec named 'destination module id' has its *valueSelector* (instance variable) set to 'parentsId'. This 'parentsId' Smalltalk message must then be sent, not to the module output which actually owns the MessagePrototype, but to each module *input* to which the output is connected. This output (as the owner of the MessagePrototype) is the object to which *valueSelector* messages would *normally* be sent to provide a field value.

However, in this case the FieldProviderSpec for this 'destination module id' field has a Smalltalk message ('destinations') assigned to its *secondaryCollectionSelector* instance variable. Hence E-Scape overrides its normal course, and it is *this* message which is sent to the module output. The output object responds by returning a Set of all the 'destination' module inputs to which is connected.

A device message is to be constructed for each of these source-destination connections, as stated above. A value for the 'destination module id' field is then supplied for each of these connections, by sending the *valueSelector* Smalltalk message ('parentsId') to each of the module input objects which have been returned by the output object (in response to the *secondaryCollectionSelector* message).

Such a complex object structure has been found to be necessary to allow users themselves to define message protocols within E-Scape. Protocols (eg MIDAS) which follow an object-oriented structure are straightforward to define, but even protocols (eg MIDI) which do *not* follow the recommended communication standards of chapter 5 can be described. This E-Scape protocol description structure is also capable of extension, if new kinds of protocol data structures are encountered in the future which go beyond those presently in use. It is not inconceivable that new Smalltalk objects may need to be introduced in future.

---

1 If the device type allows this.

However, the existing E-Scape Protocol structure can cope with such new objects or structures with no adaptation (new code) required.

The above description will now be illustrated by several examples.

8.8.3. MessageType example 1 - 'MIDI:controller'

This MessageType is called 'MIDI:controller' and is defined within the E-Scape Protocol named 'MIDI'. It has a specification entered into E-Scape by a user as shown in figure 48 below.

This MessageType describes the format of the MIDI 'controller' message (see 2.5.2.1). This consists of three bytes: hex Bn (where n is the MIDI channel number 0-15), a 'controller number' value (0-127), and a data value (0-127).

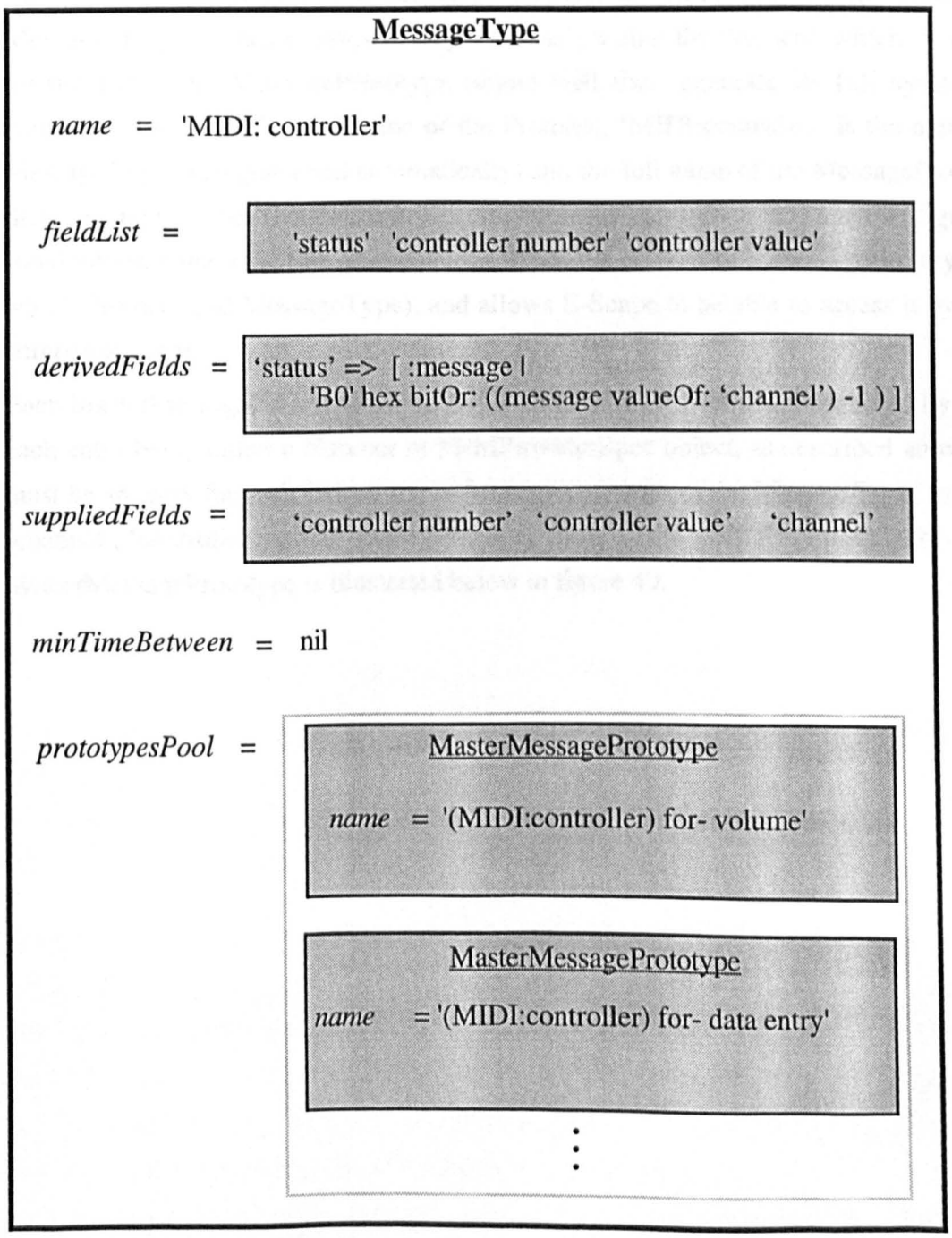


Fig. 48 MessageType example 1: 'MIDI:controller'

This MessageType has three fields which are named by three Strings in its *fieldList.*

Two of these fields ('controller number' and 'controller value') are to have their values supplied by E-Scape objects, and these names thus also appear in the *suppliedFields* list. Each MessagePrototype of this type will then be required to have a FieldProviderSpec object defined for each of these 'supplied' field names - each FieldProviderSpec specifies *what* object is going to supply a field value, and how is to be asked.

The third field ('status') is not supplied directly, but is *derived* from supplied values. It thus appears as a key in the *derivedFields* Dictionary, as illustrated above. Its code block performs a bitwise logical OR operation with hex B0 on the supplied 'channel' value, to derive the value for the 'status' field of the MIDI 'controller' message (see 2.5.2.1).

This 'controller' MessageType has several MessagePrototypes specified (in the form of MasterMessagePrototypes, as described above), of which two are illustrated. Each MessagePrototype has a 'purpose' (eg 'volume') within the Protocol, which is specified by the user. The MessagePrototype object will then generate its full system name automatically. 'MIDI' is the name of the Protocol, 'MIDI:controller' is the name of the MessageType (also generated automatically) and the full name of the MessagePrototype is then generated to be '(MIDI:controller) for- volume', as shown in the above figure. This standardised name structure allows a user to see the context of a MessagePrototype (ie in which Protocol and MessageType), and allows E-Scape to be able to access it by a single composite name.

Each MasterMessagePrototype has its *fieldProviderSpecs* Dictionary specified by the user, each entry being either a Number or FieldProviderSpec object, as described above. There must be an entry for each field name in the *suppliedFields* of the MessageType, in this case 'channel', 'controller number', and 'controller value'. This '(MIDI:controller) for- volume' MasterMessagePrototype is illustrated below in figure 49.

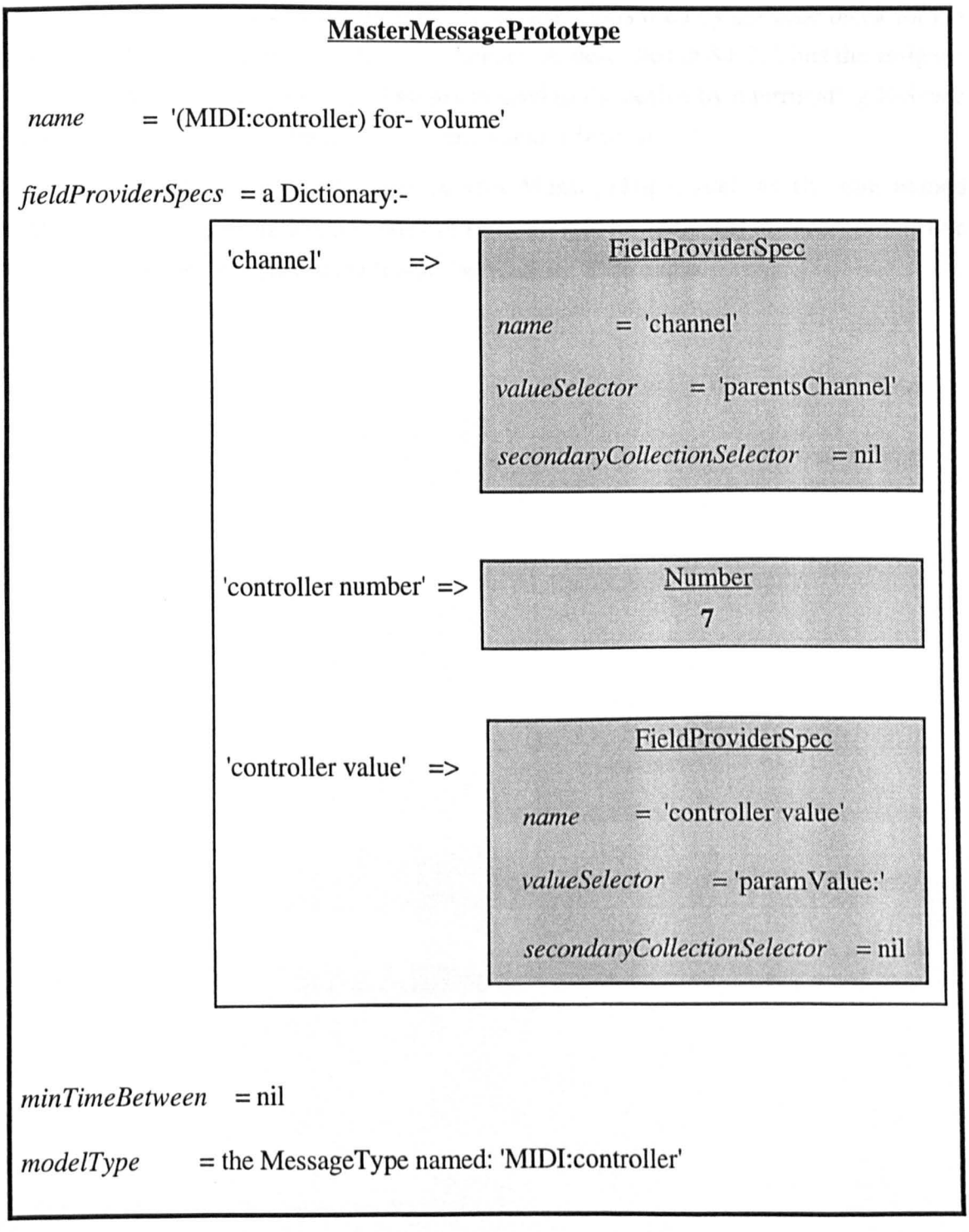


Fig. 49 MasterMessagePrototype example 1: '(MIDI:controller) for- volume'

A MessagePrototype created by this MasterMessagePrototype will usually be assigned to the input of a submodule within an E-Scape Instrument definition. This module input object will later be converted to an analogous object which describes an input of a unit allocated within a device. It is from this object that a value will be procured for each supplied field of the MessageType. This will be achieved by sending the *valueSelector* Smalltalk message - held by the corresponding FieldProviderSpec - to the object.

Thus the value for 'channel' variable will be returned by sending the message 'parentsChannel' to the submodule input which owns the MessagePrototype. This value is

not present in the MessageType's *suppliedFields* list, but is used by the code block for the 'status' field in the *derivedFields* CodeDictionary, as described in 8.8.3. Thus the assigned module input can build up the actual values to send to the device by interrogating E-Scape objects, with (optional) processing of the data values obtained.

The other MasterMessagePrototypes in this MessageType, such as the one named '(MIDI:controller) for- data entry' (shown bottom right in figure 48) differ from this one only in the Number assigned to the 'controller number' field name.

8.8.4. MessageType example 2 - 'MIDAS: send data to UGP'

The second example MessageType is called 'MIDAS: send data to UGP'. It is defined within the 'MIDAS' Protocol, and is illustrated in figure 50 below. This MessageType describes the MIDAS message (presented in 113.1.1) which sends a data value to an input of a specified MIDAS UGP. It which has the format:

SendValueImmediate, *UGP* id, *UGP-input* name, Data Value.

The 'SendValueImmediate' field has been set as the number 4 by the MIDAS development team.

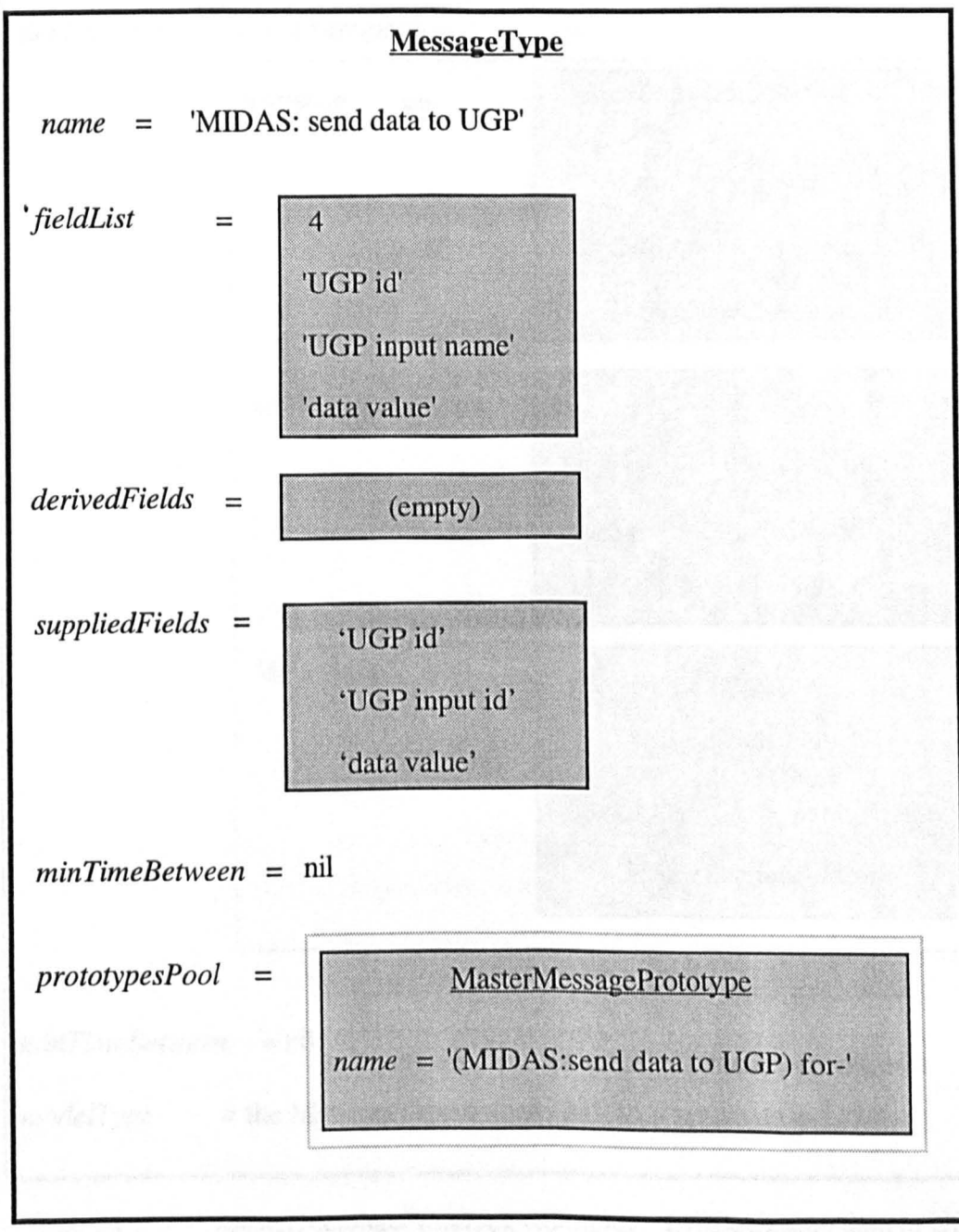


Fig. 50 MessageType example 2: 'MIDAS: send data to UGP'

In this MessageType, the first of the four fields is a fixed value of 4. No value processing is required for any of the fields, hence the *derivedFields* CodeDictionary is empty. The

MessageType has a *single* MasterMessagePrototype named 'MIDAS: send data to UGP', which therefore does not need its purpose specifying in its name. Compare this with the previous example 'MIDI:controller' MessageType, which had many MessagePrototypes, each being used for a different purpose within the Protocol.

The structure of this MasterMessagePrototype is shown in figure 51 below.

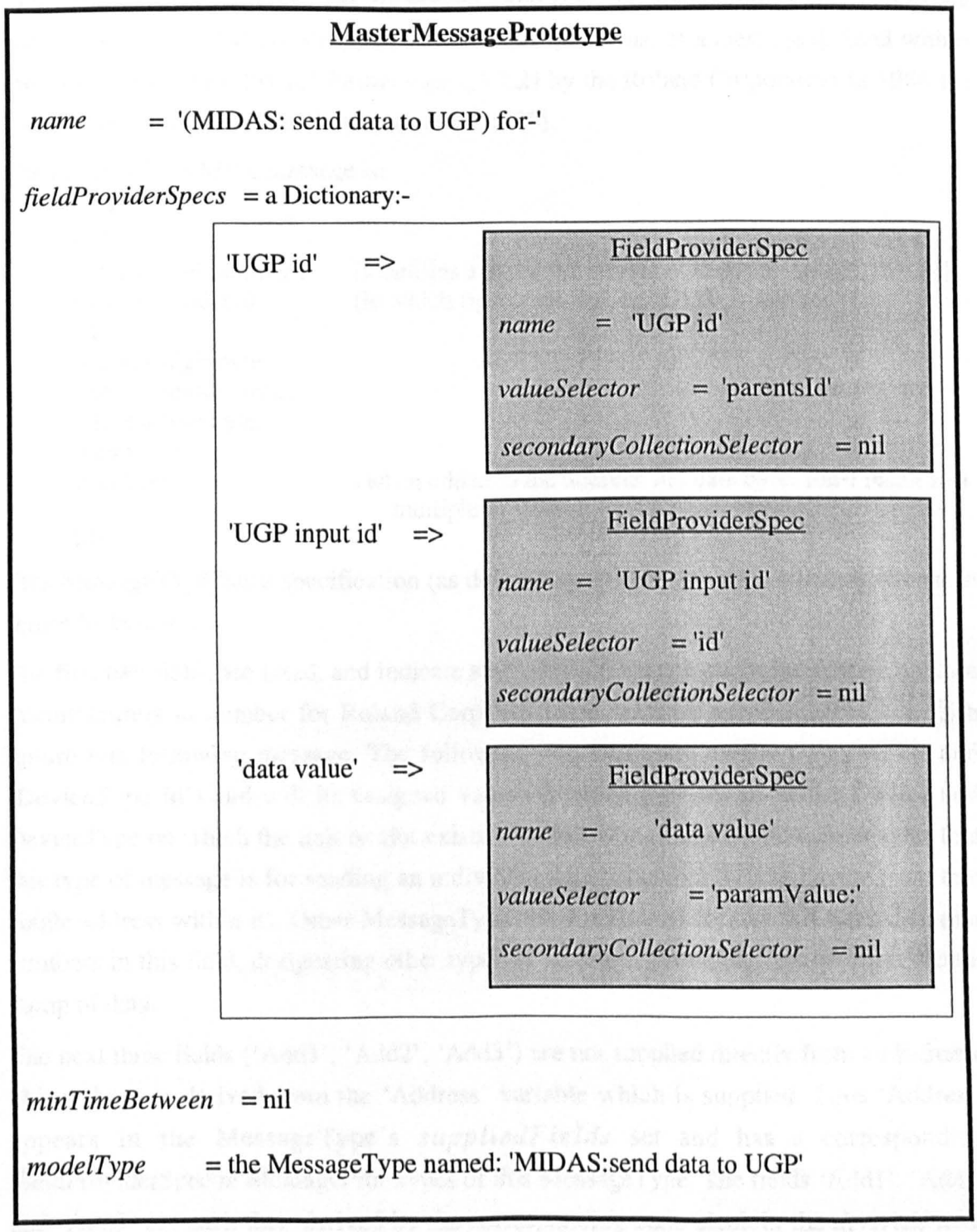


Fig. 51 MasterMessagePrototype example2: '(MIDAS:send data to UGP) for-'

The MasterMessagePrototype has three FieldProviderSpec objects, which send the Smalltalk messages 'paramValue:', 'id', and 'parentsId' to an object (a module input) to which the corresponding MessagePrototype is assigned. The relative simplicity of this

MessageType can be seen, reflecting the conformance of the MIDAS Protocols to those proposed in chapter 5. Many of the structures found necessary in E-Scape to cope with various devices would be unnecessary if all devices followed the clean object-oriented structures which underlay the proposed communication standards.

8.8.5. MessageType example 3 - 'MIDI: Roland 1986 sys. exc. standard'

This MessageType is called 'MIDI: Roland 1986 sys. exc. standard-data set', and exists within the E-Scape 'MIDI' Protocol. It describes the format of a message defined within the MIDI 'system exclusive' format (see 2.5.2.2) by the Roland Corporation in 1986 for their subsequent synthesiser devices (Roland 1988).

The format of this MIDI message is:

hF0, [1]	
h41,	
Device 'unit number',	(identifies a particular device of this type, range: 16 - 31)
Device 'model id',	(ie which *type* of device, eg 'D110' = h16)
h12,	
address high byte,	
address middle byte,	
address low byte,	
data value,	
checksum,	(when added to the address and data bytes must result in a multiple of 128)
hF7.	

This MessageType has a specification (as defined by an E-Scape user) which is shown in figure 52 below.

The first two fields are fixed, and indicate start of MIDI system exclusive format, and the manufacturers id number for Roland Corp. All other manufacturers' devices will then ignore this following message. The following two fields are named ('Device Id' and 'DeviceType Id') and will be assigned values depending on the id of the Device and DeviceType on which the unit or slot exists. The following fixed number designates that this type of message is for sending an individual data value to a Roland device (sent to a single address within it). Other MessageTypes in this Roland format will have different numbers in this field, designating other types of message, for example to perform a bulk dump of data.

The next three fields ('Add1', 'Add2', 'Add3') are not supplied directly from an E-Scape object, but are derived from the 'Address' variable which is supplied. Thus 'Address' appears in the MessageType's *suppliedFields* set and has a corresponding FieldProviderSpec in MessagePrototypes of this MessageType. The fields 'Add1', 'Add2' and 'Add3' are each then derived by the corresponding code block in the *derivedFields* CodeDictionary, as shown above. A further field called 'checksum' is derived using these three derived field values, as well as the supplied field value named 'Value'.

---

1 The 'h' prefix indicates a hexadecimal value.

MessageType

name = 'MIDI: Roland 1986 sys exc standard-data set'

fieldList =

```
( hexF0   hex41   'Device Id'   'Device Type Id'   hex12
'Add1'  'Add2'  'Add3'  'Value'  'Checksum'   hexF7 )
```

derivedFields =

```
'Add1' => [ ((message valueOf: 'Address') bitShift: 16)  & hex 7F]
'Add2' => [ ((message valueOf: 'Address') bitShift: 8)  & hex 7F]
'Add3' => [ (message valueOf: 'Address')  & hex 7F]
'Checksum' => [ (128 - ((message valueOf: 'Add1') +
                        (message valueOf: 'Add2') +
                        (message valueOf: 'Add3') +
                        (message valueOf: 'Value')))
          % 128 ]
```

suppliedFields =

```
'Device Id'   'Device Type Id'   'Address'   'Value'
```

minTimeBetween = 3 (ms)

prototypesPool =

MasterMessagePrototype

name =
'(MIDI: Roland 1986 sys exc standard-data set) for- PART input'

MasterMessagePrototype

name =
'(MIDI: Roland 1986 sys exc standard-data set)
for- PART+RHYTHM channel asssign'

•
•
•

Fig. 52 MessageType example 3: 'MIDI:Roland 1986 sys exc. standard-data set'

Note that the *minTimeBetween* instance variable is set to 3ms. This is the time a device takes to respond to a message sent using this MessageType format, and must be taken into account when E-Scape is scheduling messages to a device in accordance with a score (see 9.3.4). The time in a score (SSE) when a unit is required to be ready, or when a data state must be present, needs to be offset (brought forward in time) when E-Scape time-stamps

and stores the low-level messages which are to be sent to the device to achieve this. However, E-Scape at present makes no allowance for *different* devices to have *different* response times to messages of the same MessageType. This issue will require some additional work - each DeviceType really needs its own table of response times for each MessageType it uses.

This MessageType has a several MasterMessagePrototypes, two of which appear in figure 52. The first one shown is named '(MIDI:Roland 1986 sys exc. standard-data set) for- PART input', and is illustrated in figure 53 below.

MasterMessagePrototype

name = '(MIDI: Roland 1986 sys exc standard-data set) for- PART input'

fieldProviderSpecs = a Dictionary:-

'Device Id' =>	FieldProviderSpec *name* = 'Device Id' *valueSelector* = 'deviceId' *secondaryCollectionSelector* = nil
'Device Type Id' =>	FieldProviderSpec *name* = 'Device Type Id' *valueSelector* = 'deviceTypeId' *secondaryCollectionSelector* = nil
'Address' =>	FieldProviderSpec *name* = 'Address' *valueSelector* = 'address' *secondaryCollectionSelector* = nil
'Value' =>	FieldProviderSpec *name* = 'Value' *valueSelector* = 'paramValue:' *secondaryCollectionSelector* = nil

minTimeBetween = nil

modelType = the MessageType named: 'MIDI: Roland 1986 sys exc standard-data set'

Fig. 53 MasterMessagePrototype example 3a:
'(MIDI:Roland 1986 sys exc. standard-data set) for- PART input'

This MasterMessagePrototype has four FieldProviderSpec objects, as shown, which send the Smalltalk messages 'deviceId', and 'deviceTypeId' 'address', and 'paramValue:' to an object (in this case a module input) to which the corresponding MessagePrototype is assigned.

The second MasterMessagePrototype in this MessageType (shown at the bottom of figure 52) is named '(MIDI:Roland 1986 sys exc. standard-data set) for- PART+RHYTHM channel assign'. It is shown below in figure 54 below.

MasterMessagePrototype

name = '(MIDI: Roland 1986 sys exc standard-data set) for- PART+RHYTHM channel assign'

fieldProviderSpecs = a Dictionary:-

'Device Id' =>

FieldProviderSpec
name = 'Device Id'
valueSelector = 'deviceId'
secondaryCollectionSelector = nil

'Device Type Id' =>

FieldProviderSpec
name = 'Device Type Id'
valueSelector = 'deviceTypeId'
secondaryCollectionSelector = nil

'Address' =>

FieldProviderSpec
name = 'Address'
valueSelector = 'channelAssignAddress'
secondaryCollectionSelector = nil

'Value' =>

FieldProviderSpec
name = 'Value'
valueSelector = 'channelNumber'
secondaryCollectionSelector = nil

minTimeBetween = nil

modelType = the MessageType named: 'MIDI: Roland 1986 sys exc standard-data set'

Fig 54. MasterMessagePrototype example 3b:

'(MIDI:Roland 1986 sys exc. standard-data set) for- PART+RHYTHM channel assign'

As it is of the same MessageType, this second MasterMessagePrototype (3b) also has four FieldProviderSpec objects (with the same four 'supplied field' names). It is superficially similar to example 3a, except that such MessagePrototypes (created from this 'master') will be owned by a DeviceDCIPrimitiveSlot object (see 8.7), rather than a module input. This object has *channel* and *channelAssignAddress* instance variables which are accessed by the E-Scape messages 'channelAssignAddress' and 'channelNumber'. Thus, the fields of the MessageType are now supplied by a different object, with the Smalltalk message also being different for the 'Address' and 'Value' fields.

Note that all this structure can be specified by a user *without* recourse to writing Smalltalk code. The messages to be sent are selected from a limited choice, eventually via a menu, by a user who is defining a new MessagePrototype. The only Smalltalk code written at present is to specify the code block used to determine the value for the derived fields of the MessageType, and this uses a small number of mathematical functions, plus a handful of standard constructions which again could in future be loaded to the code block in response to user menu selections.

8.9. ScoreEvents

As described in 8.3, a ScoreEvent has an assigned Instrument, and contains event parameter data for the top-level (Psp) inputs of that Instrument.

This event parameter data is then processed within the Instrument into other lower-level values which are assigned to inputs of the device synthesis structure which is defined (and used) by the Instrument. These lower-level values are also be stored within the ScoreEvent (see 9.3.1 for details is this). An Instrument should *not* hold any score-related data itself, as a single Instrument will usually be used by many different ScoreEvents.

Some event parameters may interrelate, ie a value from each may be needed to derive a processed lower-level input value. These factor have resulted in the ScoreEvent structure outlined below. More details of this structure, and how it relates to the Instrument are given in 9.2.1.

8.8.1. ScoreEvent structure

As described in 8.3, a ScoreEvent's parameter data is contained in PspFunction objects, each of which corresponds to a Psp in its Instrument. This data must be *grouped* into the parameters associated with a *single* PspProcessor in the Instrument, in order for the PspProcessor to be able to access all the input data it needs. Thus, a ScoreEvent has a ParameterHolder object (which contains the appropriate PspFunctions) corresponding to each PspProcessor of its Instrument.

This is illustrated in figure 55 below. The Instrument has two PspProcessors, and the ScoreEvent thus has two corresponding ParameterHolder objects (shown outlined on the left of the figure - top and bottom) within the ScoreEvent.

Each PspFunction corresponds to a Psp (input) of the PspProcessor, and contains data breakpoints (each with a time and value) associated with that Psp. For example, the 'pitch' and 'detuning spread' Psps of the Instrument each have a corresponding PspFunction (shown top left) in the ScoreEvent.

These PspFunctions are then processed by the PspProcessor to create lower-level input data for the inputs to the device synthesis structures defined by the Instrument (described in detail in section 9.2.2). Each of these structures is a network of synthesis units on a particular device, which is defined by a DCT object (as described in 8.4.3). Each Set of processed values will be assigned to one of the inputs of a DCT. These values will share the same time-base as the event parameters (PspFunctions) from which they have been derived, and are thus *also* stored within the ScoreEvent. The processed values assigned to each DCT input are stored in a DCTSignal object, which exists within the *same* ParameterHolder as the 'source' PspFunctions from which the values have been derived.

Each ParameterHolder thus also contains a number of these DCTSignals (shown on the left of the above figure at the bottom of each ParameterHolder). Each DCTSignal contains time-stamped 'device-level' input values which are assigned to an input of a synthesis structure.

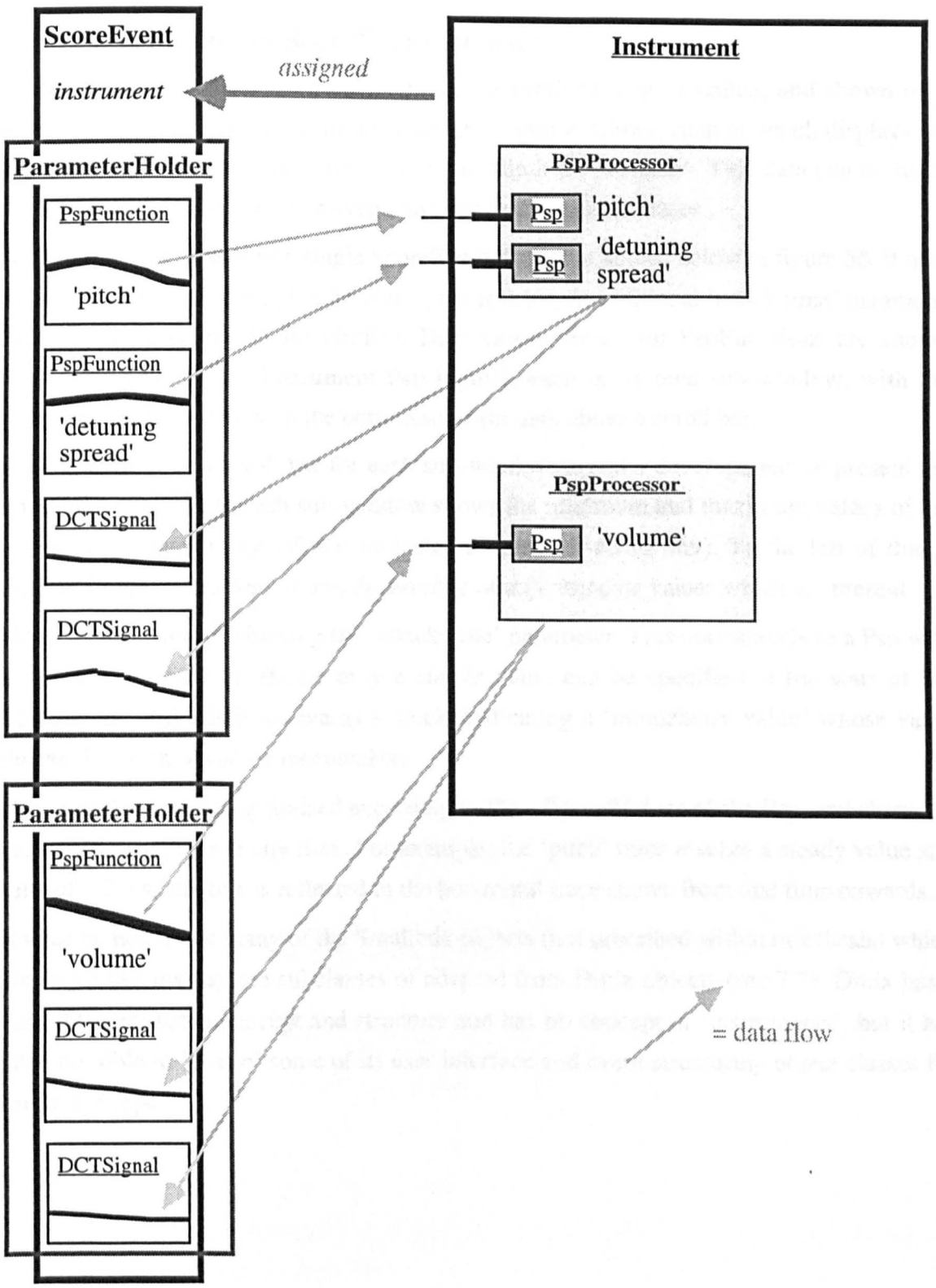


Fig. 55 Correlation of ScoreEvent and Instrument structures

These structures will be presented again in 9.2, when the operation of this data processing is described.

This DCTSignal data will later be *further* processed to create messages (to be sent to devices) when device resources have been allocated for the ScoreEvent. This is described in detail in section 9.3.4.

8.9.2. Presentation of ScoreEvents to user

Values for any or all of the parameters of a ScoreEvent can be edited, and shown on a screen display. This display is divided vertically into windows, each of which displays the data for a different parameter name, such as 'pitch' or 'volume'. This data can be time-varying during the course of an event, and can thus appear as traces.

An E-Scape screen shot of a single ScoreEvent display is shown below in figure 56. It uses the Instrument shown in figure 55 above, but is using an additional 'attack time' parameter (not shown in figure 55 for clarity). Data values from four PspFunctions are shown (corresponding to four Instrument Psp inputs), each in its own sub-window, with the common timebase shown on the bottom scale (in ms), above a scroll bar.

A vertical scale and scroll bar for each sub-window is under development; at present the box in the top right of each sub-window shows the minimum and maximum values of the sub-window display (eg -69:63 indicates a range of -63 to +69). To the left of this, a number in square brackets shows the number of *different* data values which are present.

Note the sub-window showing the 'attack time' parameter. This corresponds to a Psp with a *rate* of 'i' (see 8.3). Hence only a *single* value can be specified at the start of the ScoreEvent, and this is shown as a block, indicating a 'momentary value' whose value *during* the event would be meaningless.

The traces' values are quantised according to the *allowedValues* of the Psp, and show the value which pertains at any time. For example, the 'pitch' trace reaches a steady value at a time of ~ 2.3 s, and this is reflected in the horizontal trace shown from that time onwards.

It must be noted that many of the Smalltalk objects (not described within this thesis) which comprise this display are subclasses of adapted from Dmix objects (see 7.7). Dmix has a very different score concept and structure and has no concept of 'instruments', but it has been possible to 're-use' some of its user interface and event structuring object classes for use in E-Scape.

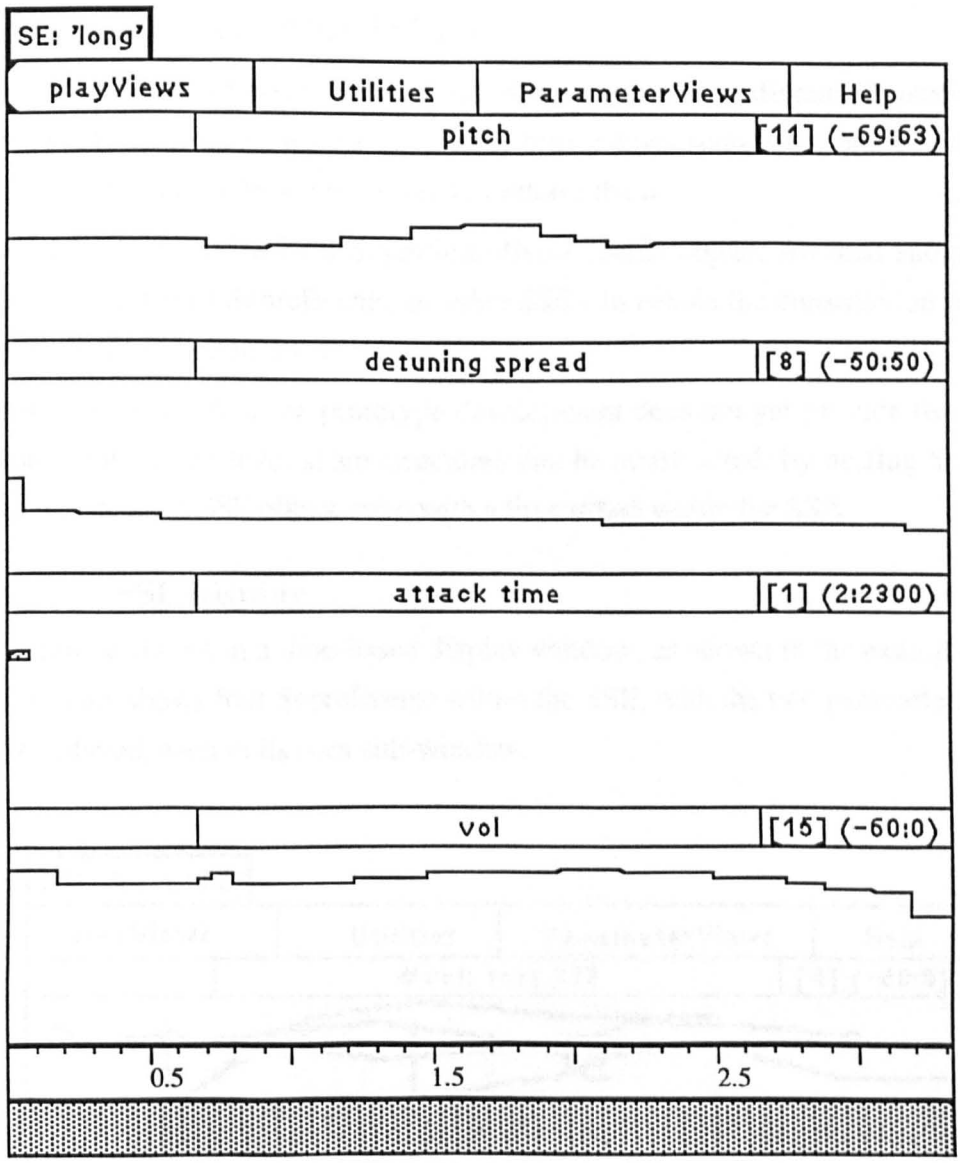


Fig. 56 A ScoreEvent display showing four parameter traces, with time shown horizontally from left to right.

If a sound event is considered to be a multi-dimensional object having parameters in many dimensions of sonic space (see 6.6.2), then each of these traces acts as a 'shadow' of the event in a single parameter dimension.

8.10. SuperScoreEvents (SSEs)

As stated in 8.3, a score can be created out of lower-level ScoreEvents. ScoreEvents thus need be able to be nested within a time-based higher-level score structure, which can then control the allocation of device resources to perform them.

This structure is provided by a SuperScoreEvent (SSE) object. An SSE should be able contain any number of ScoreEvents, or *other SSEs,* to enable the construction of an event hierarchy, as described in 3.4.4.

The present stage of E-Scape prototype development does not yet provide for nesting of SSEs, but simpler one-level score structures can be constructed, by nesting ScoreEvents within a higher-level SSE object, each with a time offset within the SSE.

8.10.1. SSE display

An SSE can be shown in a time-based display window, as shown in the example in figure 57 below. This shows four ScoreEvents within the SSE, with the two parameters 'vol' and 'pitch' displayed, each in its own sub-window.

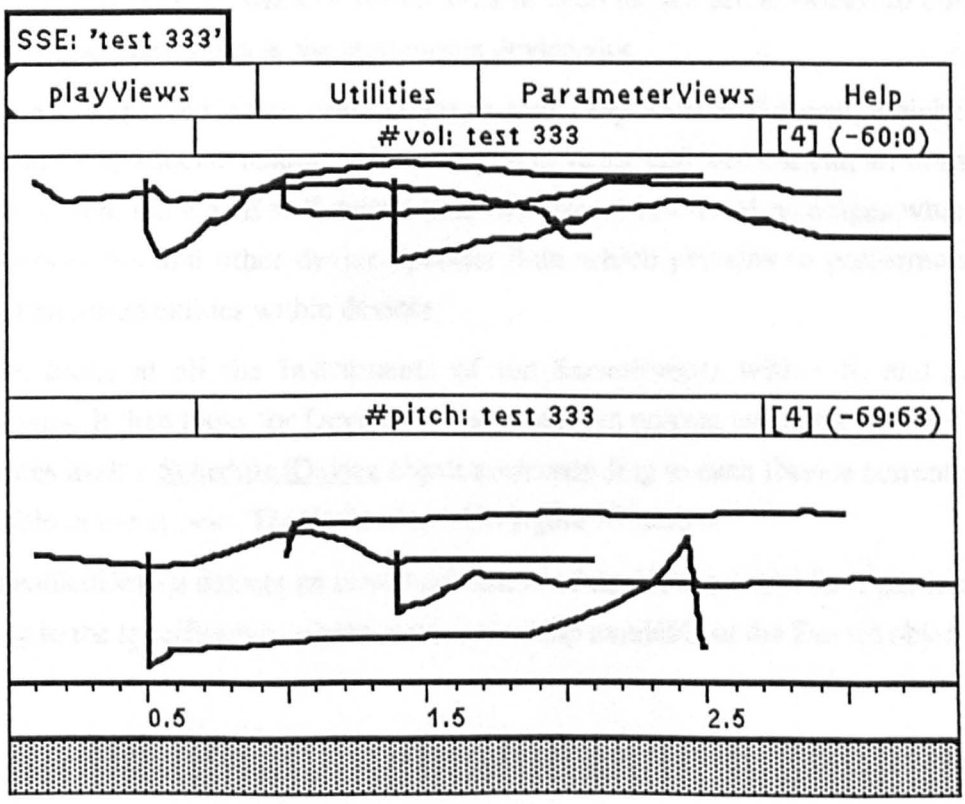


Fig. 57 An SSE display, showing two event parameters

Conventional score editing facilities (eg the copying, deleting, pasting, moving of ScoreEvents) are planned to be provided, but again these aspects are not innovative, hence have been given less development priority. Such facilities have therefore not yet been implemented at the current stage of development.

Each ScoreEvent within the SSE has an assigned Instrument, which may or may not be the same. If several different Instruments are in use, they may not necessarily have the same parameters. Hence, each ScoreEvent may or may not have a trace appearing in any given parameter sub-window of the SSE display.

Even if different Instruments have parameters with the *same* name, they will not typically have the same ranges of allowable values, resolution, or rate, and will be described by different Psp objects in each Instrument. A parameter window may thus be displaying ScoreEvent data pertaining to *several* Psps, and will have a maximum limit equal to the *largest* maximum limit of these Psps (and a minimum limit equal to the *smallest* minimum limit of the Psps). When a *particular* ScoreEvent is highlighted ('selected'), all the Psp views of *its* Instrument are activated, with the appropriate resolution, minimum and maximum limits for *that* Psp then shown in each parameter window.

8.10.2 Allocation of device resources by an SSE

In order to perform its child ScoreEvents, an SSE looks at the Instruments of all the ScoreEvents, and allocates the necessary synthesis resources to perform them, using the available connected devices. For those devices which do not perform dynamic allocation of units, the SSE must keep track of which slots in each device are allocated to units at any time, and request new units in the appropriate device slot.

Each score to be played is thus described by a *single* top-level SSE object, which is always the *only* entity to allocate resources in synthesis devices, and will contain all events which are to be played. Each SSE will derive (and then store) low-level messages which embed the addresses, ids and other device specific data which pertains to performance using particular allocated entities within devices.

The SSE looks at all the Instruments of the ScoreEvents within it, and gets their DeviceTypes. It then looks for Devices of these types in current use in the system. The SSE then creates itself a ScheduledDevice object corresponding to each Device currently flagged as available in the system. This is illustrated in figure 57 below.

The ScheduledDevice defines an actual *allocation* of device resources for a particular score, according to the specification of resources *potentially* available in the Device object.

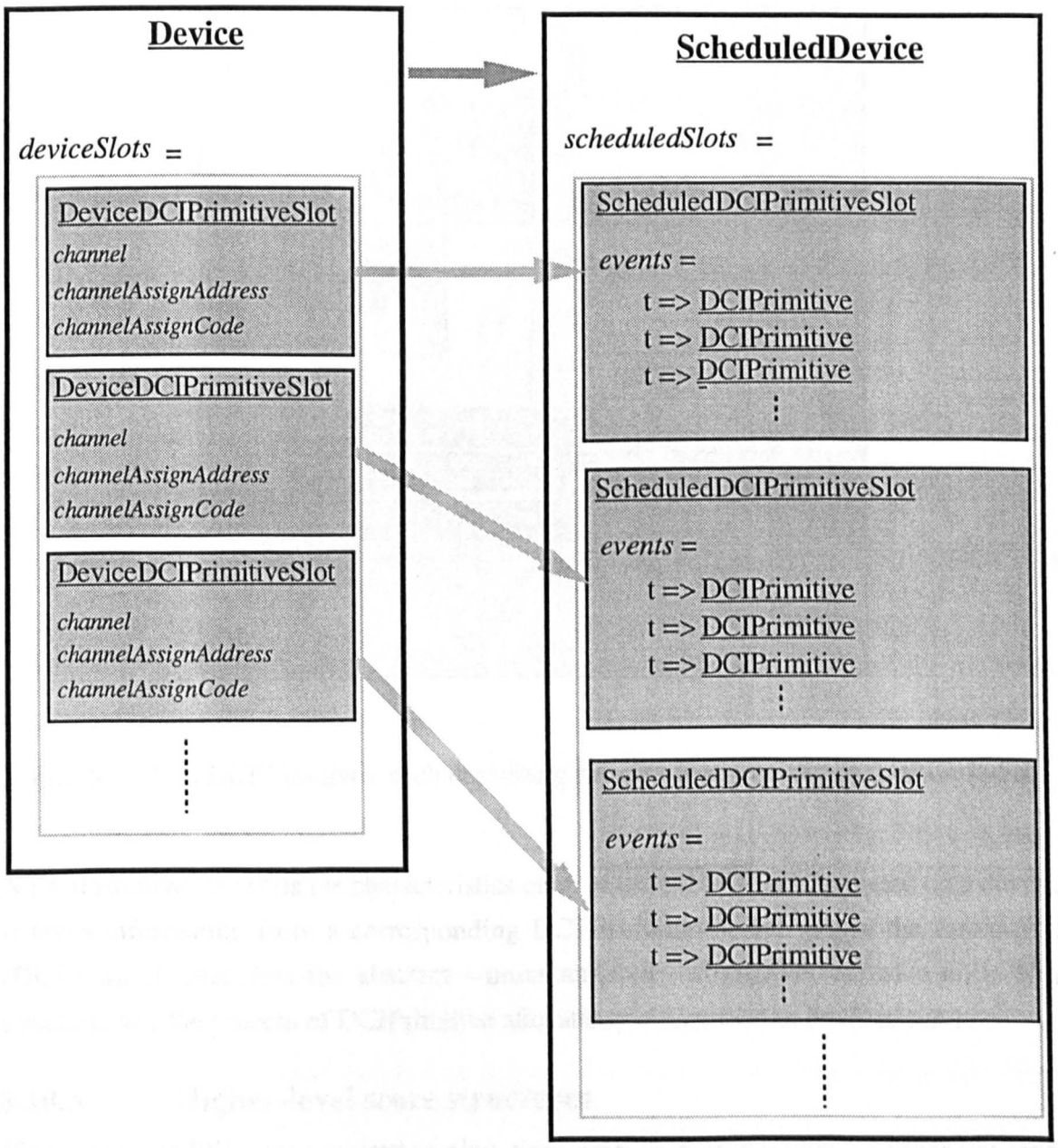


Fig. 58 Creating a ScheduledDevice from a template Device object

A ScheduledDevice has to describe the *allocation* of units in a device - ie when they are in use. It achieves this by having a number of ScheduledDCIPrimitiveSlots, each corresponding to a 'template' DCIPrimitiveSlot in the Device. These 'scheduled' slots differ from their template slots in that they have an idea of *time*. They hold DCIPrimitive objects which are quasi-events, ie they define an allocation of a particular unit in a device at a certain time for a certain *duration*, to match a score event which is using an Instrument which contains this unit within its definition (DCT).

Figure 58 below shows two DCIPrimitives created within 'scheduled slots' in an SSE, so as to match the requirements of two ScoreEvents. The durations of each DCIPrimitive describe the allocation of a unit in a device to perform the corresponding ScoreEvent.

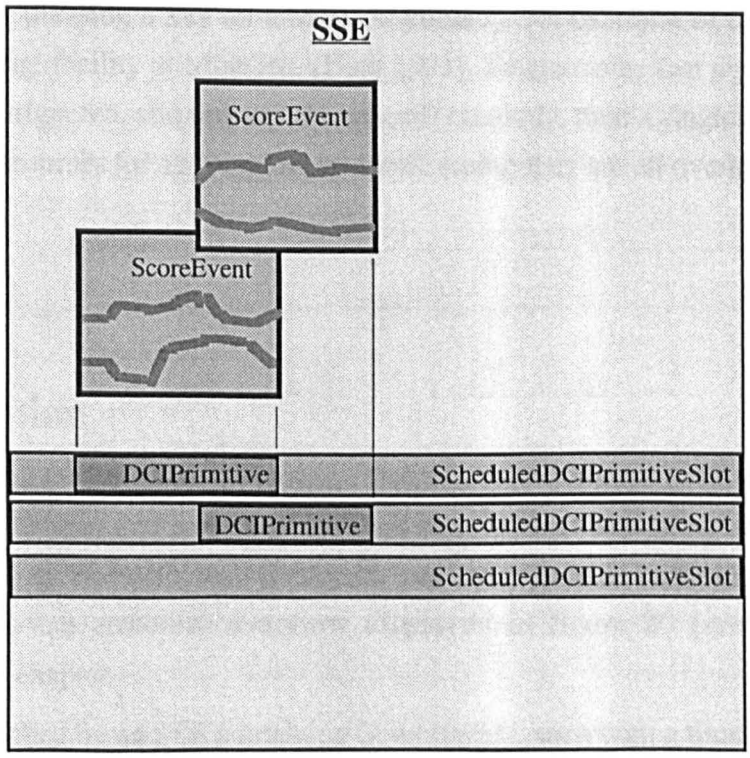


Fig. 59 Two DCIPrimitives, each describing the allocation of a unit for a ScoreEvent

A DCIPrimitive describes the characteristics of an *allocated* unit as instantiated on a device. It takes information from a corresponding DCTPrimitive object within the Instrument (DCT) which describes the abstract - uninstantiated - characteristics of a unit. This structure, and the process of DCIPrimitive allocation, is described in detail in 9.3.1.

8.10.3 Higher-level score structures

If two or more SSEs are required to play, they must be nested within a further top-level SSE, which will then allocate resources (DCIPrimitives) for the composite events of both these child SSEs. Each individual SSE may also *retain* its own resource allocation, so it can still be played independently. In each case, however, only a single SSE should be asked play at any one time, if the allocation of device resources is to correctly map to the events to be played.

Thus, two overlapping sections (SSEs) can be played independently, but if required to be played together, they must be loaded into a higher-level SSE structure. This SSE will then allocate resources - probably in a different way - to ensure successful performance. The top-level SSE will have to prevent possible conflicts between ScoreEvents, over device resources which may have been allocated to different ScoreEvents by each *individual* SSE, but which are now contemporaneous within the new structure. If the two SSEs are to be merged as described, a higher-level structure will need to look at such contentions, and reallocate resources for the otherwise conflicting events.

A user may want to employ *interactive* performance of a score, ie to be able to trigger events (components of the score) by performance actions (eg clicking on a GUI icon with a

mouse button, or pressing a key on a MIDI keyboard). An example of this would be the 'interactive scoring' facility of MidiGrid (Hunt 1991). To guarantee that any combination of events can be so triggered, and be played (realised) correctly, then a single SSE will have to allocate device resources for all these events by assuming they are all overlapping *within* the SSE (see 6.8.3).

8.11 Conclusion

This chapter has introduced the structure, rationale and relationship of the main objects present within E-Scape, and something of their functioning. This chapter now concludes with a summary of these object's structure and operation. It can usefully be read in conjunction with the structure overview displayed in figure 20 (section 8.2) at the beginning of this chapter.

- A score is described by an SSE containing ScoreEvents, each with a time offset within it.
- Each ScoreEvent has an assigned Instrument.
- Each Instrument contains Psps, which each define an available scoring parameter.
- An Instrument also contains DCTs, each of which defines a synthesis structure (a set of units) on a particular type of device, and has inputs which connect to various units within the structure.
- Each ScoreEvent can then have high-level parameter data specified as breakpoints (time and value) within a PspFunction (each corresponding to a Psp of the Instrument).
- Psps are contained within PspProcessors, each of which has a CodeDictionary. This contains simple Smalltalk code which is used by the Instrument to convert this parameter data into lower-level values. These are assigned to an input of a DCT, ie to the inputs of a device synthesis structure.
- This processed data is then stored back within the ScoreEvent within DCTSignal objects, each of which corresponds to an input of the DCT in the Instrument.
- Each DCT in an Instrument is constructed from lower-level modules which describe synthesis structures (units or networks of units) in a device. These modules in turn are built from 'module types' - SMT or PrimSMT objects. These act as templates and describe features of the synthesis structures, such as their address offsets within the device's memory map, or codes which identify a type of unit to the device.
- SMT and PrimSMTs have inputs (SMTInput and PrimSMTInput objects), which similarly define various characteristics of inputs to synthesis structures or units. These input characteristics may be directly specified by an E-Scape user, or be 'inherited' from lower-level modules within the module type.
- These module type inputs also possess one or more MessagePrototypes which define how messages are to be formulated which will convey data to the corresponding unit input in the

device. Each MessagePrototype defines how (and from what object) data for each field of a message is to be supplied.

• Related MessagePrototype are stored in groups within a MessageType, which also defines which fields are present in a message and how their data is derived from that supplied.

• MessageTypes are grouped within Protocol objects, which also supply details of the computer output ports from which messages of this kind may be sent.

• Module types are stored within module categories (DTSMTCategory objects) which group them according to their location within a device when installed in it. A DTSMTCategory also has DCIPrimitiveSlot objects, which each define the characteristics of a device location where synthesis structures of this category may be installed.

• These module categories are stored within a DeviceType object, which also defines the general features of a particular *type* of device, such as its address map, if any.

• Each synthesis device of a particular type which is connected to E-Scape will have a corresponding Device object. A Device derives information from the appropriate DeviceType, and additionally has user specified information about the set-up and configuration of the particular device.

• When a ScoreEvent is installed within an SSE, the SSE will allocate resources in the necessary devices, as specified by the DCTs in the ScoreEvent's Instrument. To do this, it uses information from the DeviceType of each DCT, the available Devices of this type, the DTSMTCategory of each module in the DCT, and the DCIPrimitiveSlots in this category. This process will be described in detail in 9.3.

9. Functioning of E-Scape

Chapter 8 introduced the structure of the E-Scape software: its objects and their inter-relationships. This chapter presents a description of the functioning and operation of these objects within E-Scape. It necessarily reiterates or refers to some of the object descriptions of the previous chapter, but focuses on the functionality of objects. In addition, some degree of repetition is deliberately employed within this chapter, in order to aid comprehension.

The reader of this chapter may find it useful to review, or refer to, sections 8.1 and 8.2, and to figure 20 therein, for an overview of E-Scape's structure.

- **Section 9.1 describes how Instruments are created**

9.1.2 introduces a new DCTHolder object which contains one or more DCTs (defining synthesis structures), plus various sets of PspProcessors (defining scoring parameters) which mate with these DCTs. An Instrument can be built by selecting a DCTHolder (which has the desired DCTs), then selecting one of its sets of PspProcessors to use with these DCTs.

9.1.3 then describes how a DCT is constructed. Sub-modules are created within a new DCT, using 'module type' objects as a template. These sub-modules can then be linked to each other and/or to inputs of the DCT.

- **Section 9.2 describes how ScoreEvents are constructed**

9.2.1 describes how an Instrument is first selected to create a new ScoreEvent. Parameter data can then be entered into this new ScoreEvent, stored in PspFunction objects which match the Instrument's Psps (scoring parameters). This parameter data is then processed in two stages:

9.2.2 describes stage 1, which involves feeding the parameter (PspFunction) data through the Instrument's PspProcessors to produce input data for the synthesis structure. The processed data is then stored in DCTSignal objects within the ScoreEvent. An example of stage 1 processing in then given in detail.

- **Section 9.3 describes how ScoreEvents are prepared for performance on synthesis devices**

9.3.1 describes how ScoreEvents are contained within a higher-level SSE (SuperScoreEvent), which allocates synthesis resources (units) in devices in order to perform them. To do this, the SSE uses a DCIPrimitive object (which has a time and duration) to describe each allocation of a synthesis unit in a device. Each DCIPrimitive is also assigned a location within a device, which is selected on the basis of which locations are free of allocations at any given time.

9.3.2 describes stage 2 of the processing of ScoreEvent data. This involves creating the actual messages which will be sent out to a device in order to (i) request the instantiation of the units described by the DCIPrimitives, and (ii) send parameter data to their inputs.

This processing is performed by MessagePrototype objects. These are owned by each 'module type' (and its inputs) from which the DCIPrimitive was constructed. Each MessagePrototype gets data values from various designated objects, to formulate low-level DeviceEvent objects. These contain sets of numerical data which will be sent to a device. DeviceEvents are stored back into the ScoreEvent from which they have been derived. The stage 1 example is then carried forward to illustrate these processes.

- **Section 9.4 describes how SSEs play**

9.4.1 describes how SSE can unravel their hierarchical structure into single level of ScoreEvents.

9.4.2 describes how the DeviceEvents contained in the ScoreEvents are sent out of the appropriate computer port.

- **Section 9.5 describes E-Scape's support for algorithmic composition**

It described the design of E-Scape Instrument structures which can support various types of algorithmic compositional activity, by creating or processing data for event generation or parameter control. The movable boundary between software (E-Scape) and external (device) processing structures is discussed, and illustrated with examples.

- **Section 9.6 describes E-Scape's top-level system organisation**

It presents a summary of E-Scape's objects, how they relate to each other, and where they are stored. The storage of ScoreEvents and SSEs within Composition objects is described, as is the grouping of sets of active Devices within DeviceSetup objects.

9.1. Creation of Instruments

9.1.1. Overview

The description of the structure of an Instrument (presented in 8.4) has been given in terms of the key objects - one or more DCTs and PspProcessors - which are visible within it. Each DCT object describes a set of synthesis units in a particular type of device. Each PspProcessor object presents one or more parameters which can be specified from a score which uses the Instrument. Figure 60 illustrates this basic structure within an Instrument.

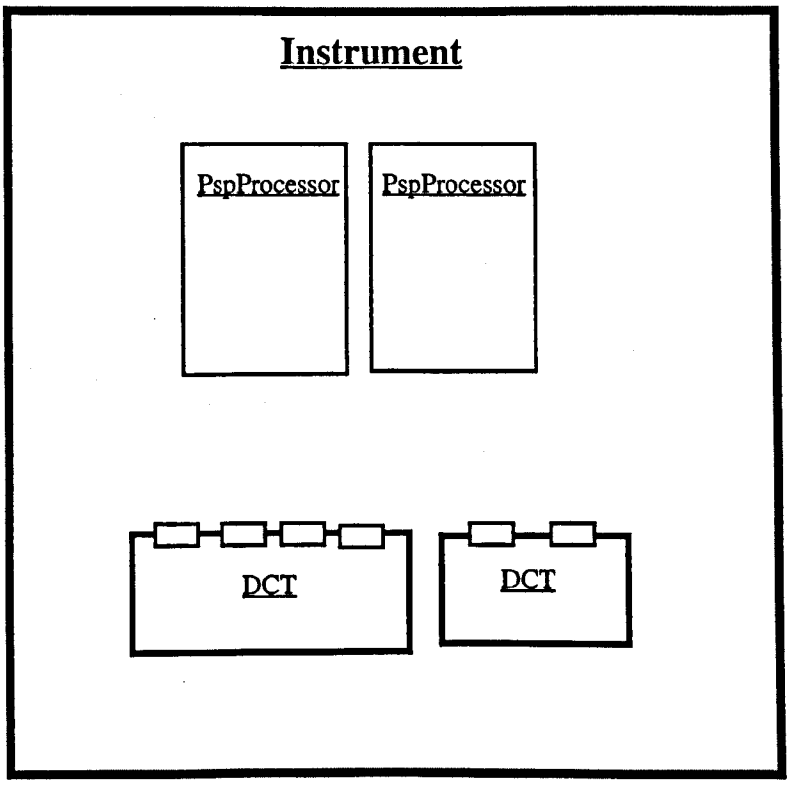


Fig. 60 Instrument structure

However, for a user to be required always to construct a new Instrument by creating *new* DCT and PspProcessor objects would be very laborious. What is needed is a structure which allows the *reuse* of DCTs and PspProcessors.

Remembering that each DCT defines a synthesis structure within a device, there can be many *different* sets of PspProcessors which can provide access to these structures to a composer. In the example Instrument in the above figure, there are two PspProcessors which derive input values for the six inputs of the DCTs shown. Thus, a composer would only need to specify *two* parameters (Psps) for a ScoreEvent, yet have *six* parameters provided as low-level inputs to the device synthesis structure (DCT).

A designer may, however, wish to create a variant of this Instrument, perhaps with *more* scoring inputs, and less processing within the PspProcessor. This would enable a composer to have more *direct* control over low-level synthesis parameters, rather than have

the PspProcessors derive values according to a pre-programmed algorithm. Such an Instrument would enable a more detailed and individual specification of parameter values to be made by a composer for each ScoreEvent, although with a concomitant loss of ease of use (as described in 6.1), as the composer must then specify more parameter values for each event.

In order to facilitate the creation of *families* of Instruments - each with the same synthesis structures (DCTs) but differing score parameters (ie PspProcessors) - E-Scape needs to allow an Instrument designer to create and store several different *sets* of PspProcessors for use with the *same* DCTs.

Hence the PspProcessors are stored as a set within a PspProcessorSet object.

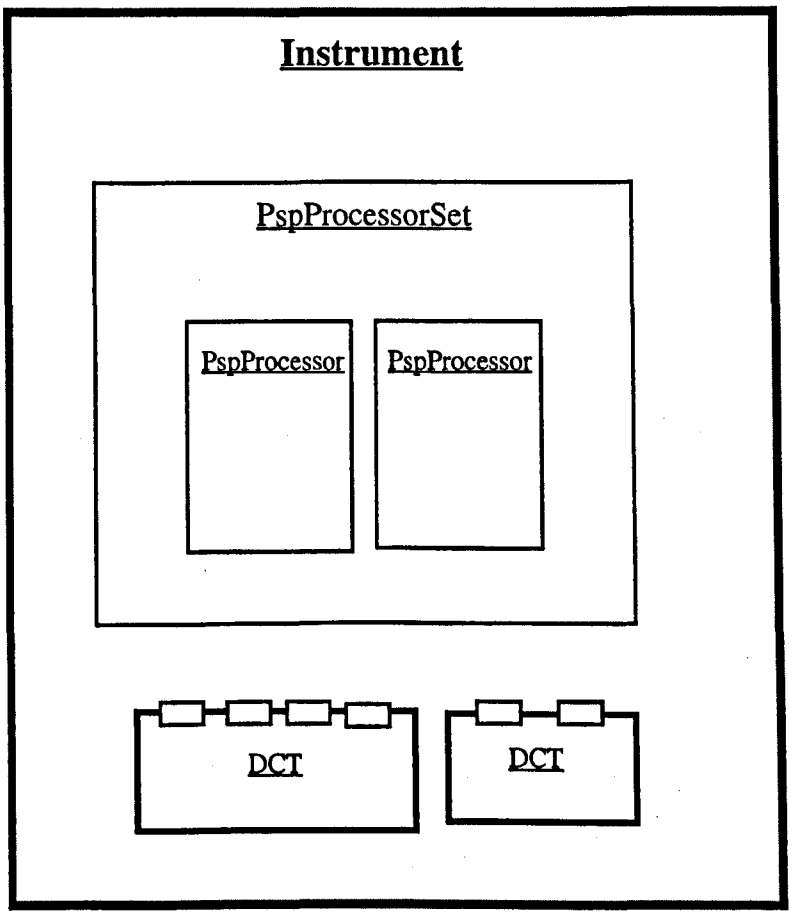


Fig. 61 Instrument structure, showing its PspProcessors contained in a PspProcessorSet

Several *different* PspProcessorSets can be created, each containing *different* PspProcessors, but all of which derive input values for the *same* synthesis structures (DCTs). Such groups of PspProcessorSets, can be stored with the DCTs (to which they act as a 'front end') within a DCTHolder object. DCTHolders are stored separately from Instruments within a library, and can be selected by a user in order to provide a template when constructing an Instrument. The new Instrument can then utilise the DCTs stored in the DCTHolder, and have a set of PspProcessors selected by the user from those available within the DCTHolder. This is illustrated in figure 62 below.

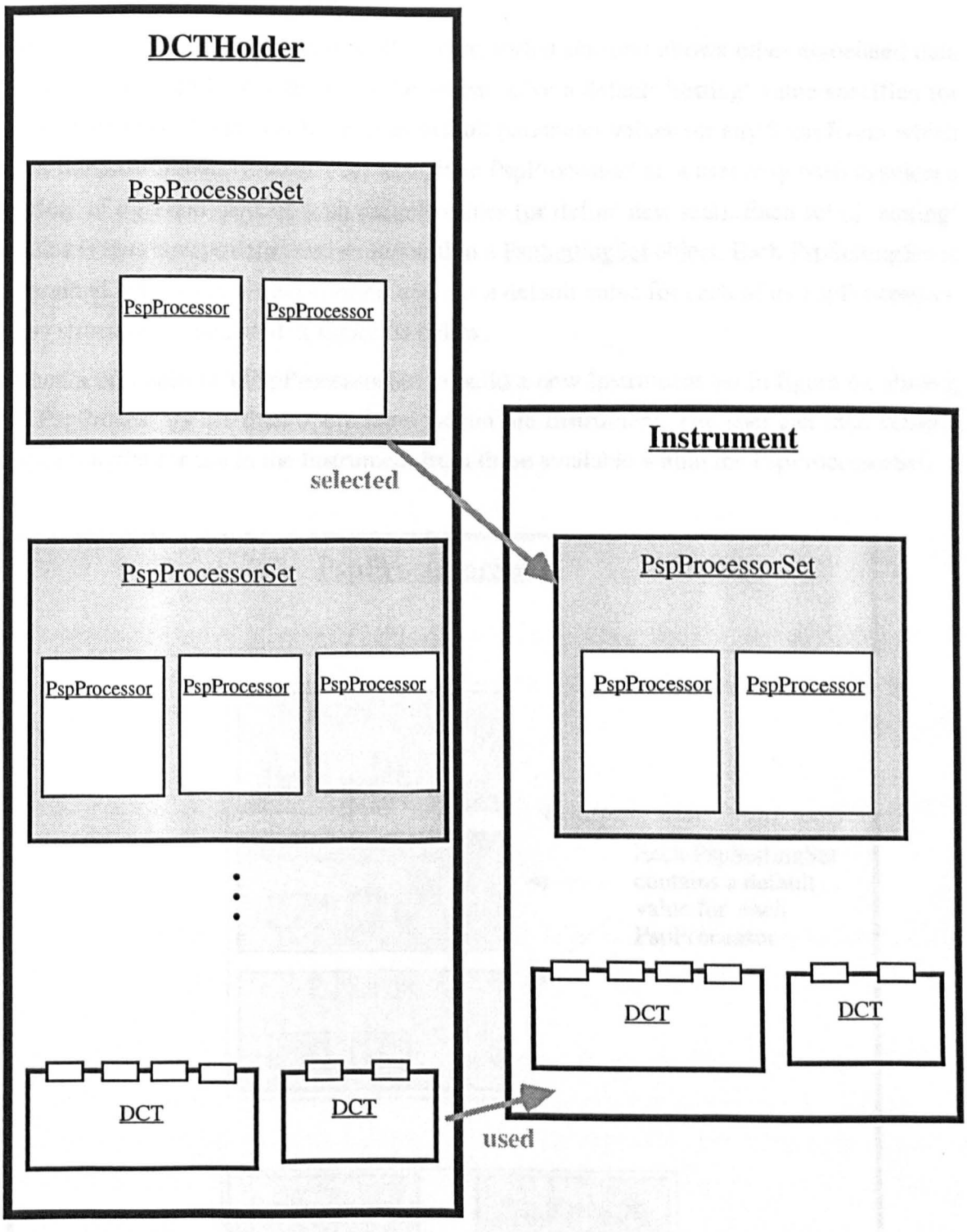


Fig. 62 Using a DCTHolder to create an Instrument-utilise its DCTs, and select one of its sets of PspProcessors

9.1.2 PspProcessorSets

Grouping PspProcessors into sets (PspProcessorSet objects) allows other associated data to be stored. Each PspProcessor in the set can have a default 'setting' value specified for each of its Psps. These will be used as default parameter values for any ScoreEvent which uses this Instrument. However, for any given PspProcessorSet, a user may wish to select a variety of *different* sets of such default values (or define new sets). Each set of 'setting' values is thus encapsulated and stored within a PspSettingSet object. Each PspSettingSet is contained within a PspProcessorSet, and has a default value for each of its PspProcessors. This structure is illustrated in figure 63 below.

When a user selects a PspProcessorSet to build a new Instrument (as in figure 62 above), its PspProcessors are then operational within the Instrument. The user can then select a PspSettingSet for use in the Instrument from those available within the PspProcessorSet.

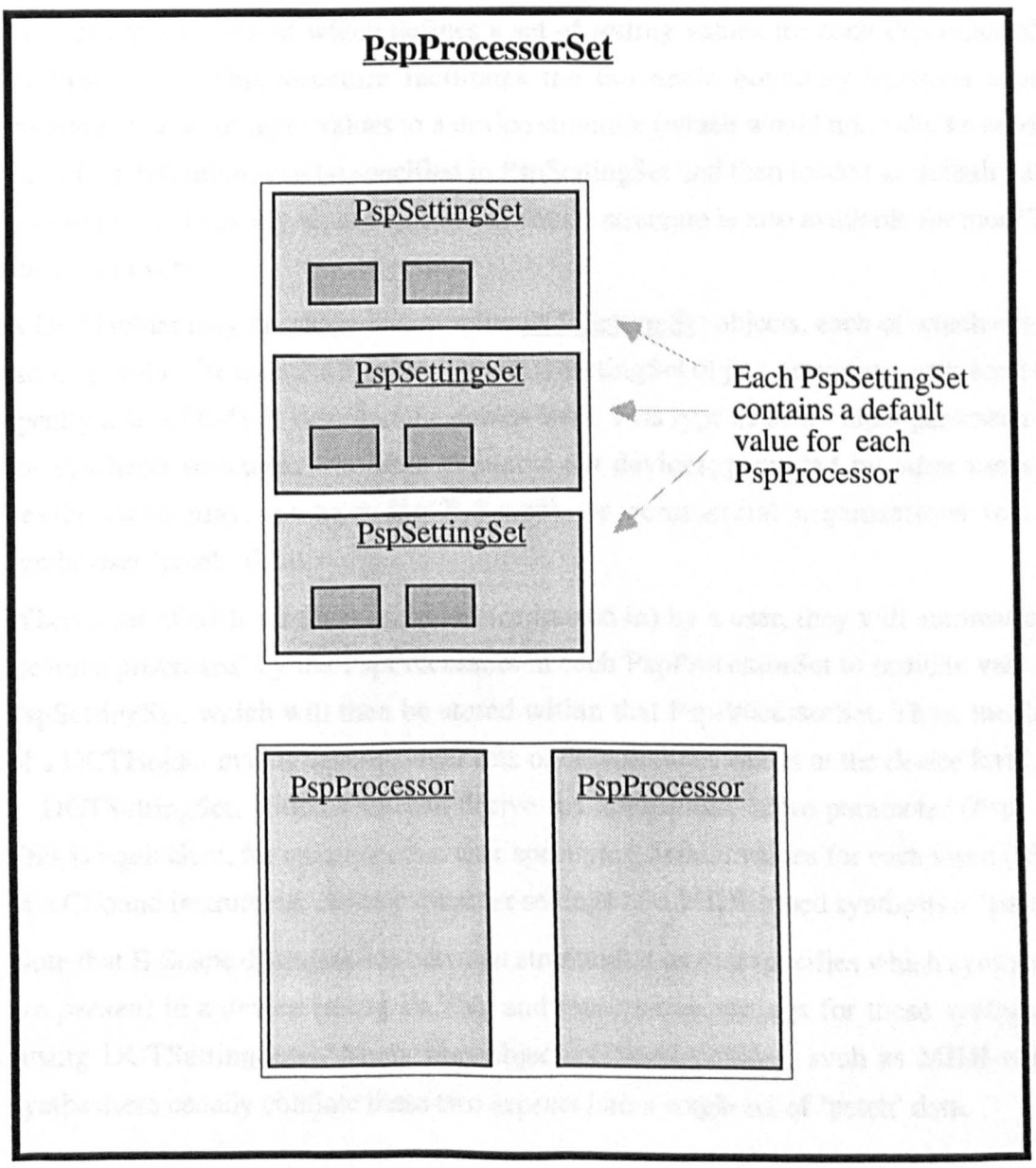


Fig. 63 PspProcessorSet object structure

9.1.3 DCTHolders

A DCTHolder acts as an Instrument construction kit, enabling an Instrument to be built out of components for different purposes. It contains one or more DCT objects, each of which defines a structures of one or more units on a particular type of device.

The designer of a DCTHolder may then provide different groups of PspProcessors which provide parameter inputs (Psp objects) to be presented to a composer. Each group of PspProcessors is contained within a PspProcessorSet object.

PspProcessors already have default values for all of their Psp inputs, but alternative sets of default values can be imposed. These are stored in a PspSettingSet - which contains one value for each Psp of the PspProcessors in the set. These sets of values can either be newly specified by the Instrument designer, of selected from amongst the existing PspSettingSets which are stored within the PspProcessorSet.

Thus a PspProcessorSet contains its PspProcessors, plus (optionally) one or more PspSettingSets, each of which defines a set of setting values for each Psp input of those PspProcessors. This structure facilitates the moveable boundary between score and Instrument: a set of input values to a device structure (which would normally be considered part of its definition) can be specified in PspSettingSet and then loaded as default values to a score event. Thus any input value of the device structure is also available for modification during an event.

A DCTHolder may also have one or more DCTSettingSet objects, each of which contains a 'setting' value for each *DCT* input. The DCTSettingSet object provides a way for a user to specify a set of default values *at the device-level.* This type of data - input parameter values for synthesis structures - is often available for devices; provided by other users of the device (who may not be using E-Scape), or commercial organisations (eg. MIDI synthesiser 'patch' data).

When a set of such values is specified (or loaded in) by a user, they will automatically be 'reverse processed' by the PspProcessors in each PspProcessorSet to provide values for a PspSettingSet, which will then be stored within that PspProcessorSet. Thus, the designer of a DCTHolder may if desired, enter sets of default input values at the *device* level, within a DCTSettingSet, which will then derive the appropriate score parameter (Psp) values. This is equivalent, for example, to a user specifying default values for each input ('P-field') of a CSound instrument, or the parameter settings of a MIDI-based synthesiser 'patch'.

Note that E-Scape distinguishes between structural data that specifies which synthesis units are *present* in a device (using DCTs), and *input value* settings for these synthesis units (using DCTSettingSets). Many non object-oriented devices, such as MIDI-controlled synthesisers usually conflate these two aspects into a single set of 'patch' data.

Thus, a user can enter device level input data which has been created elsewhere, and have E-Scape process it into scoring parameter values. A user who is constructing an Instrument with these DCTs (taken from a DCTHolder) can then select one of these PspSettingSets, but may *not* access the DCTSettingSet directly. Thus device-level 'patch data' can be *loaded* to E-Scape, but then appears at a higher level as Psp values in a PspSettingSet.

A DCTHolder has instance variables *pspProcessorSets*, *dCTSettingSets* and *dCTs* which are assigned to the above components, as illustrated in figure 64 below.

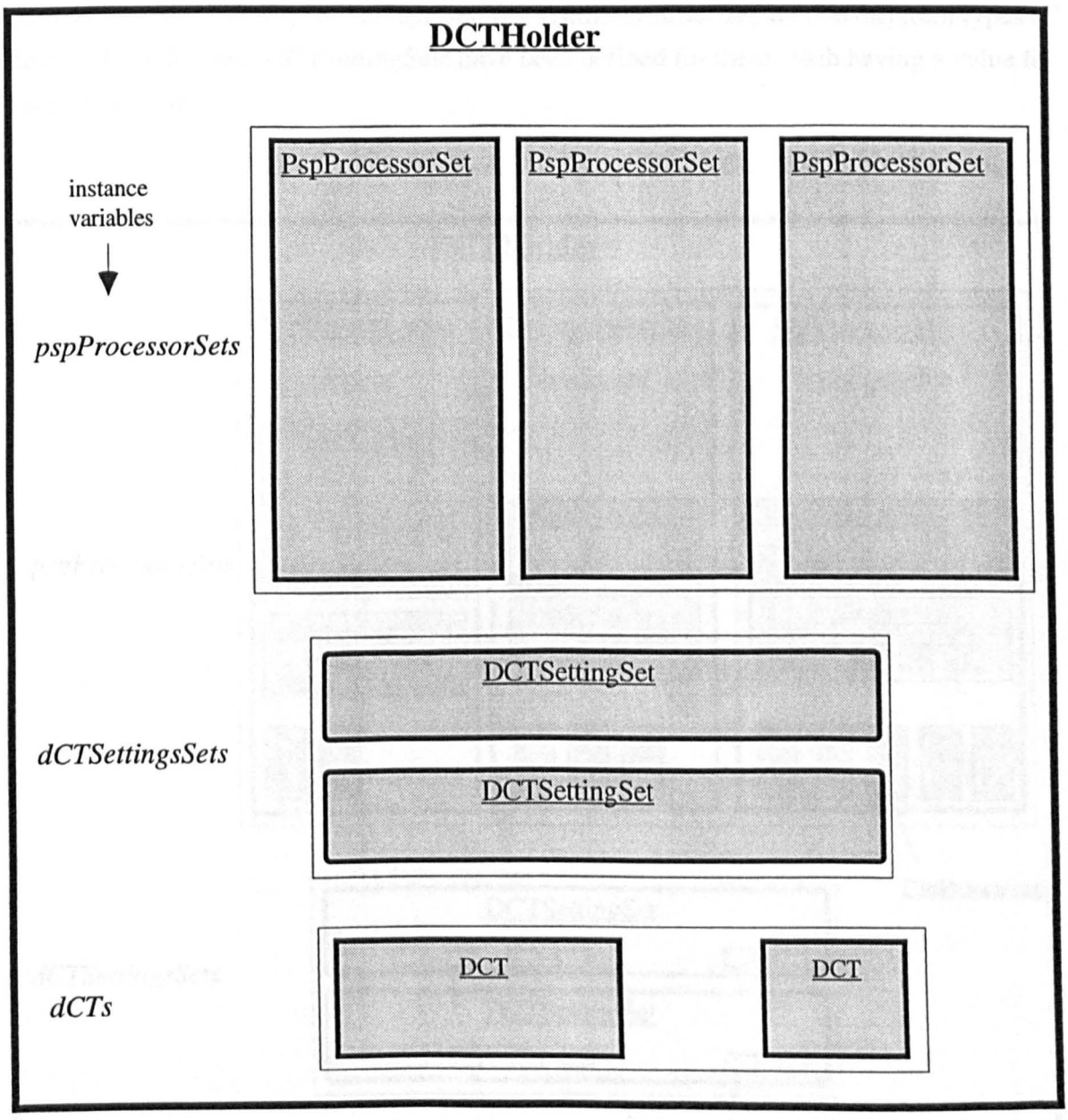


Fig. 64 DCTHolder - object structure overview

Figure 65 below shows the objects within a DCTHolder in more detail. This example DCTHolder contains three PspProcessorSets.

- The PspProcessorSet named 'simple' has two PspProcessors, which will present two (Psp) score parameters;
- The PspProcessorSet named 'moderate' has three PspProcessors, which will present four score parameters. This is because one of the PspProcessors has *two* Psps (as in the example in 8.3).
- The PspProcessorSet named 'complex' has five PspProcessors, which will present five score parameters.

The DCTHolder has two DCTs, representing synthesis structures on two different types of device. Two different DCTSettingSets have been defined for these, each having a value for each DCT input.

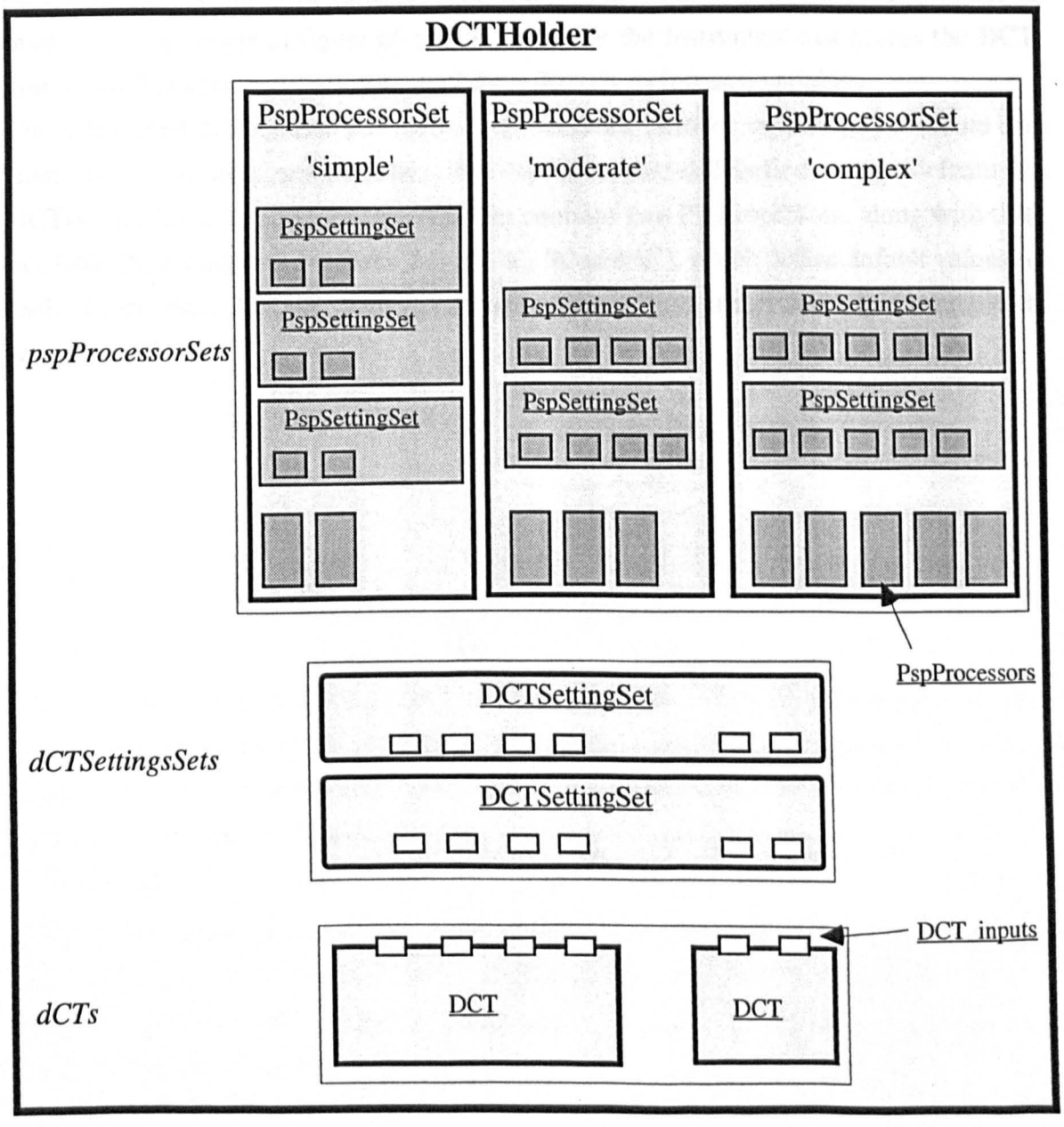


Fig. 65 An example DCTHolder - showing its detailed structure

To build an Instrument, a user can either build its structures from scratch, or select an existing DCTHolder ('template') to use, whose DCTs the Instrument can then access.

Having selected a DCTHolder, the user would then select a PspProcessorSet from those available in the DCTHolder, to provide the desired scoring parameters via its PspProcessors.

Finally, a PspSettingSet (containing default values for the Psp parameters of the PspProcessors) is selected from those available within this PspProcessorSet. If no suitable PspSettingSet is available, the user may define a new one (or edit an existing set) which is then stored in the PspProcessorSet, and is available for future use.

If suitable PspProcessors are not available, an Instrument designer may also define new ones, or assemble new combinations within PspProcessorSets. These additional objects are then automatically stored in the DCTHolder, so as to be available in future when building Instruments.

An Instrument has instance variables *dCTHolder*, *pspProcessorSet*, and *pspSettingSet*, as shown in the example in figure 66 below. Note that the Instrument can access the DCTs from its DCTHolder, and does not 'own' them directly as instance variables.
The illustrated Instrument has two PspProcessors (shown in bold). To create this Instrument, the user would select the PspProcessorSet labelled 'simple' from the DCTHolder. This 'simple' PspProcessorSet contains two PspProcessors, along with three available PspSettingSets for them (named 'a', 'b' and 'c'), which define default values for each of their Psps. The user then selects one of these PspSettingSets, in this example the set named 'c'.

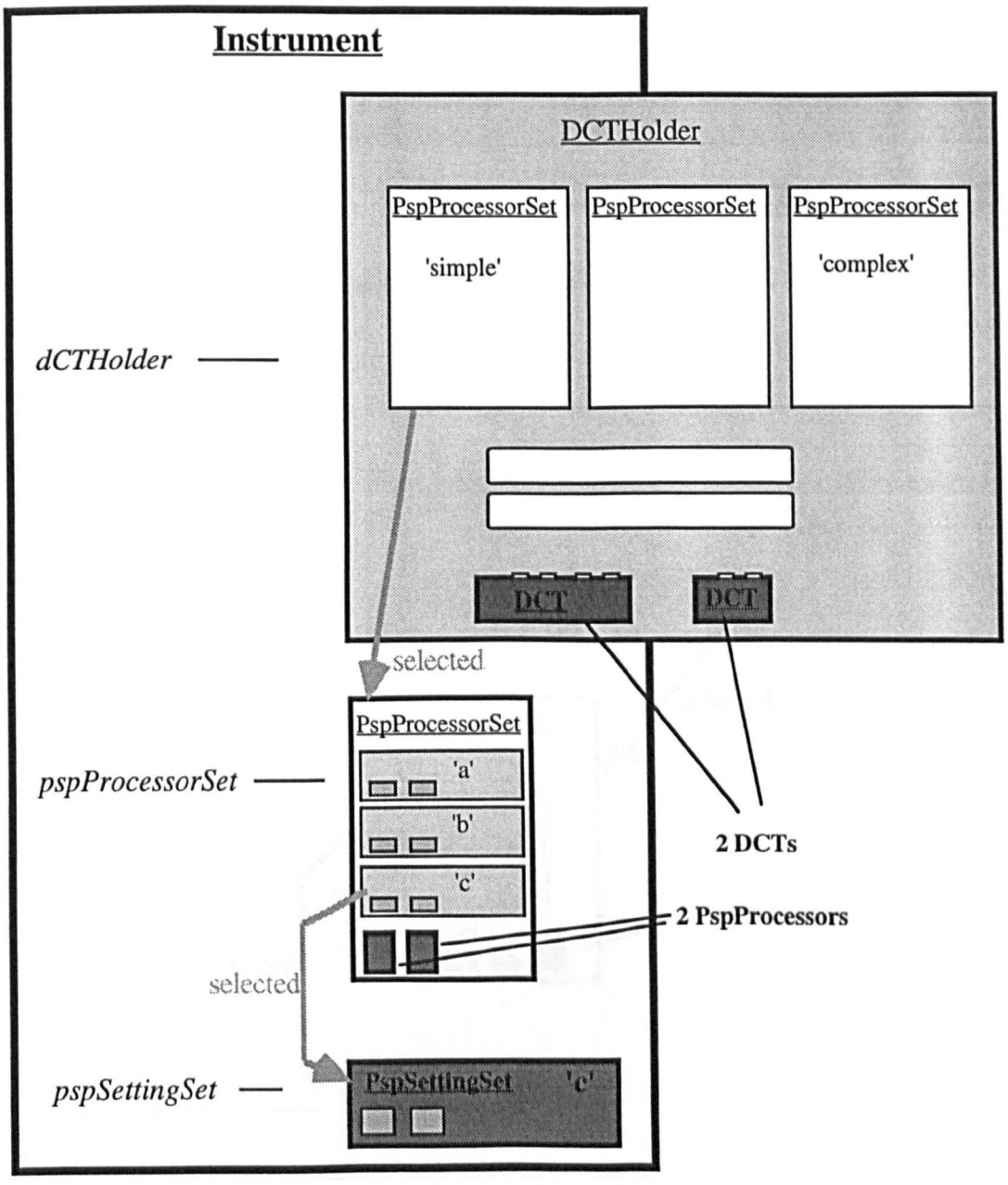


Fig. 66 Constructing an Instrument from a DCTHolder - example 1, with two PspProcessors

Another Instrument could be constructed using the *same* DCTHolder resource. Such an Instrument would then have the *same* DCTs but use a *different* PspProcessorSet, selected from those available in the DCTHolder. In the example Instrument illustrated in figure 67 below, the user has selected the right-most PspProcessorSet (named 'complex') within the DCTHolder.

This PspProcessorSet contains *five* PspProcessors, as shown. The user has then selected the PspSettingSet named 'bell' from within this PspProcessorSet. Note that it is not *compulsory* for the user to select a PspSettingSet, as default values are also defined in the Psps of each PspProcessor.

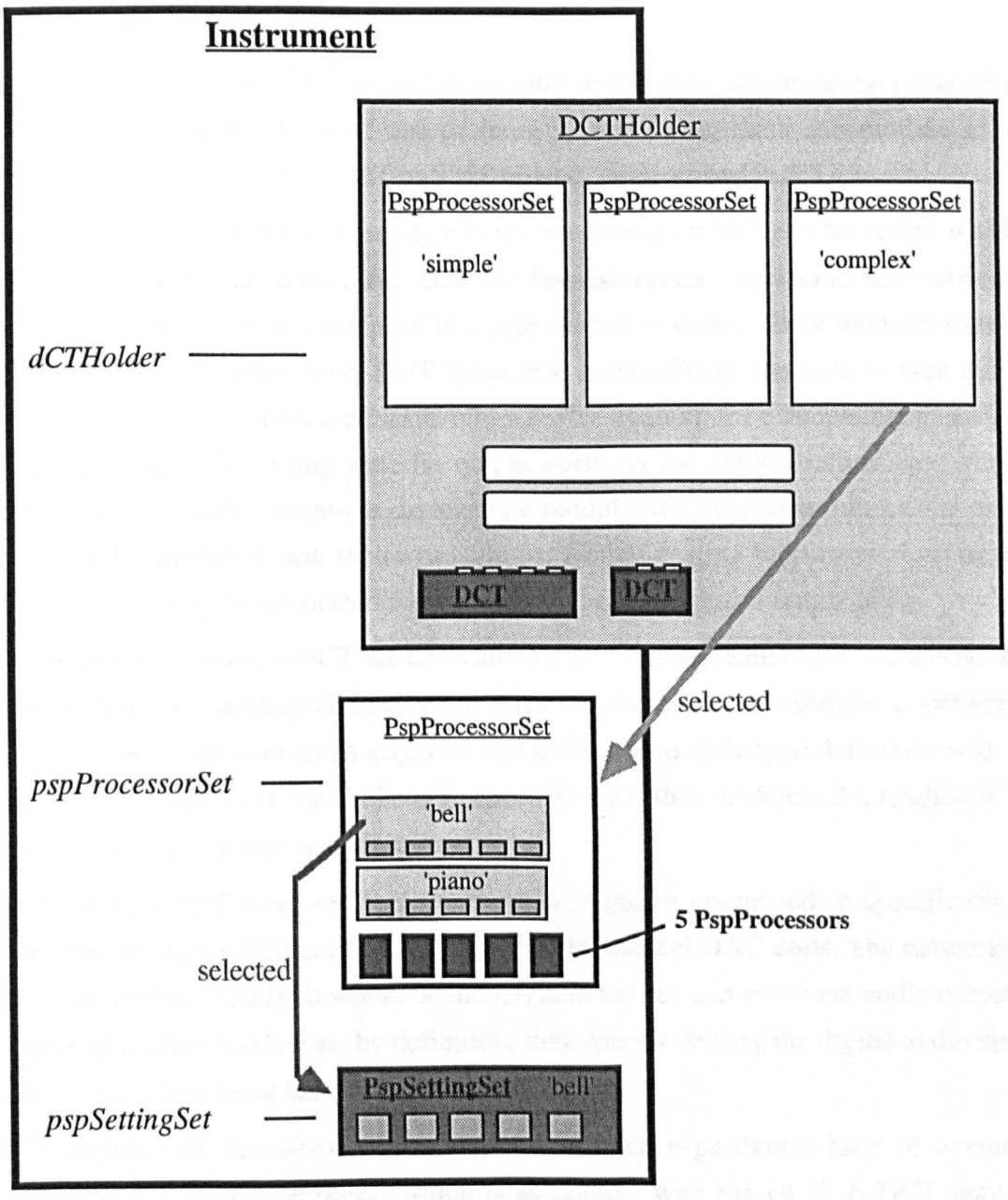


Fig. 67 Constructing an Instrument from a DCTHolder - example 2, with five PspProcessors on the same two DCTs

9.1.4 Constructing DCTs

As described in 8.4.3, each DCT describes a synthesis structure which can be installed in a single device, and is built out of one or more sub-modules. Each sub-module is of a particular module *type*, (a PrimSMT or SMT object), as described in 8.5.4.

A module type can be used as a template to create modules which can be nested within a *higher-level* module type (SMT). An SMT can later be reused - subject to any restrictions imposed by the device - as a template to create further modules. These modules can then be nested within still higher-level SMT structures. Alternatively, the module type may be used to create modules which are *locked* into a device context, for example being assigned to a specific part of it, using specific output ports on the device, or utilising various globally available device resources. In this case modules are created within a *DCT* object, and this DCT structure is *not* then available for further nesting because various *device-specific* modules may be connected within a DCT structure, such as output ports.

Modules used in creating a DCT are described by DCTSubModule objects, although they are derived from the same SMT template as other sub modules. For example, a network to perform complex FM operations might be designed as a module type definition with two outputs. These outputs are then able to be connected to other modules, if a module of this type is used within a higher-level structure.

If used within a DCT however, the two outputs might be connected to specific objects within a device, eg the 'left channel DAC and 'right channel DAC units. The network will then lose its abstract nature; it would be inconsistent to then connect these audio outputs to the inputs of another module as, by definition, they specify routing the digital audio stream to DACs which then *leave* the device as sound.

A DCT object thus describes a synthesis structure on a particular kind of device, as described by a DeviceType object which is associated with the DCT. A DCT may also (optionally) have a *particular* synthesis device (corresponding to a particular Device object) specified for its synthesis structures to be installed on.

DCT Class definition

The structure of each DCT object when instantiated is determined by the DCT Smalltalk class definition, illustrated in figure 68 below. Each instance variable (in italics on the left) will be assigned to an object or set of objects as specified.

DCT

instance variables

parent - The DCTHolder which owns this DCT

subModules - a Set of sub-modules (DCTSubModule objects) which are contained within this DCT.

inputs - a Set of DCTInputs.

dCTPrimitives - a Set of DCTPrimitive(s). These describe device-level units in the device, and are created from the above DCTSubModules (see 9.1.4.4).

deviceType - a DeviceType. All child DCTSubModules of this DCT are held within this DeviceType.

deviceID - a String. Either is 'nil', if no specific device (Device) is to be used, otherwise is the id number of the Device to be used

Fig. 68 DCT object structure

The process of constructing a new DCT can be broken down into the following stages:

9.1.4.1 Specify modules within the DCT

A DeviceType is first specified and assigned for the new DCT. One or more child 'submodules' (DCTSubModule objects) are then created, and installed within the DCT. Each submodule is created using a selected module *type* (PrimSMT or SMT object). A module type is either selected from those available in a selected category of the assigned DeviceType, or can be newly built or edited by the user at the time. E-Scape then creates a DCTSubModule using this module type as a template.

As these submodules are describing entities within a device, they must be from a category which is designated as being either *at* or *above* "device-level". As described in 8.6.1, a

category is "device-level" if it contains module types which describe *units* on a device, and is *above* "device-level" if it contains module types which describe *networks* of units. In other words, these module types must either be, or contain, modules which describe *units* within a device, rather than lower-level components (used only to *construct* a specification of a unit).

9.1.4.2 Specify inputs for the DCT

The user may then specify inputs for the DCT, in the same way as is done when creating an SMT (as described in 8.5.5.3 above). These may simply match the inputs of the child submodules, or have different connections specified. Not all the inputs of the child submodules may be connected, in which case E-Scape cannot send score-derived data to the corresponding unit input in a device. A user who is designing a DCT may deliberately want to do this in order to limit the possibilities of control - by restricting the number of data inputs available - for another user of the Instrument which contains this DCT. Different DCT objects can thus be built out of the same submodules (ie device units) but with differing degrees of accessibility to their input parameters.

9.1.4.3 Connect each DCT input to child modules

The user can then specify a connection from each DCT input to inputs of the submodules. This is illustrated in figure 69 below. These submodule inputs can be interrogated for information by a DCT input which is connected to them, and can thus inherit their characteristics, such as data range, rate, and default value.

The DCT input then 'inherits' the *messageValMap*, *deviceValMap*, *rate*, *polyphony*, and *toDeviceValRuleBlock* of the destination input.

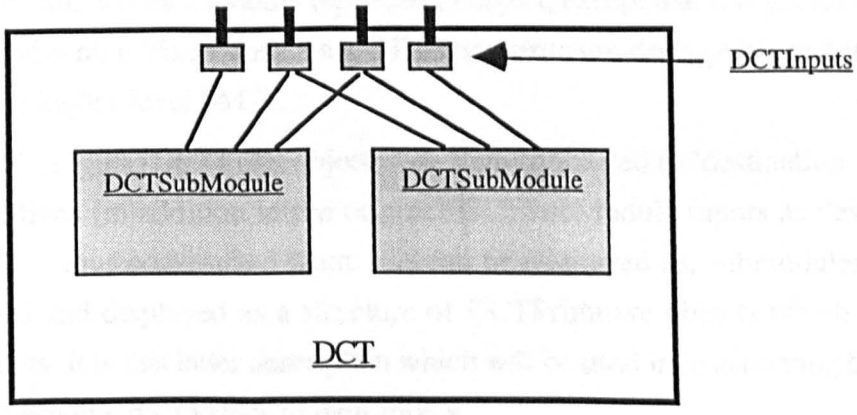


Fig. 69 An example DCT, showing its inputs connected to its submodules

9.1.3.4 Unravel the child modules to 'device-level'

The DCT describes a structure of units which may be instantiated in a device. As constructed so far, however, it consists of modules which themselves contain modules (ie above "device-level"). Thus, the DCT structure must be 'unravelled' - ie its hierarchy

flattened to a single layer of ("device-level") modules (units) which the device can understand and instantiate. Note that these modules may actually be complex, *if* the device supports such units. The process can be broken down into the following stages:

- **Unravel the child submodules to "device-level" modules.**

Each DCT is first unravelled to one or more 'device level' (Prim)SMT modules. To be 'device level' means that a module can be independently run (eg started and stopped) when instantiated as a unit in a device. For example, in the MIDAS synthesiser a device-level module would be a 'UGP' (similar to CSound's unit generators); in a Yamaha 'SY77' synthesiser it would be a 'VOICE'; in a Korg 'M1' synthesiser it would be a 'PROGRAM'; in a Roland 'D110' it would be a 'PART' or 'RHYTHM'. Thus one example DCT could, when unravelled, incorporate two 'D110' 'PARTs'; another DCT could consist of three SY77 'VOICEs'. An Instrument could if desired contain both these DCTs.

If any of the DCT's child DCTSubModules is *higher* than "device level" - ie it contains further (Prim)SubModules - then it will be recursively 'unravelled' into "device-level" sub modules; with each DCT input then having connections to these.

If a DCTSubModule is *at* "device level", then no unravelling is necessary. See figure 70 below for an example of this.

- **Convert the device-level modules to DCTPrimitive objects**

Each of these unravelled device-level submodules is then converted to a DCTPrimitive object which describes a 'unit' in a device. These DCTPrimitives will then also be stored in the DCT. An example of this is shown in figure 71 below. Note that a DCTPrimitive does *not* describe a particular instantiation of a unit in a device. This is done by another object - to be described below - when device resources are *allocated* to a score event, and instantiation of a unit takes place in a device according to the specification. A DCTPrimitive is akin to a submodule within a module type (SMT) object, except that it is guaranteed to be 'device-level', and it now 'fixed' inside a DCT device structure description, and is thus not usable in building higher-level SMTs.

Each of the DCT's inputs (DCTInput objects) are then connected to 'destination' inputs of these DCTPrimitives (in addition to the original DCTSubModule inputs as described in 9.1.4.3). A DCT is thus constructed from, and can be displayed as, submodules, but can also be described and displayed as a structure of DCTPrimitive objects which represent 'device-level' units. It is this latter description which will be used in instantiating these units on a device, and sending data values to their inputs.

Each DCTPrimitive has at least one output, which may have connections to the inputs of other DCTPrimitives. Each DCTPrimitive accesses the MessagePrototypes held in the DTSMTCategory (in its *creationMessagePrototypes* instance variable).

9.1.4.5 An example DCT

An example DCT is illustrated below in figure 70 It contains two submodules (DCTSubModule objects), both created from the same SMT 'template' which was presented as an example in 8.6.4.2. As is described in 8.6.3, its system name is built up from the child modules within it, which results in it having the unfriendly but informative (to an Instrument-builder) system name:

'((PCM-(1/3) / PCM-(2/4) -> [pmix]-)-(1&2) /

(PCM-(1/3) / PCM-(2/4) -> [pmix]-)-(3&4)) -> [output scaler]-'

As described in 8.6.3, 'user labels' may be used instead of these verbose E-Scape system names. Labels are also generated automatically, in a similar way to system names, unless the user deliberately enters a label. For the purposes of clarity within this example, the above SMT has been given a user label of 'Z'. Each submodule of this type has an id, in this case 'A' and 'B', and thus each has a default label of 'Z-A' and 'Z-B'.

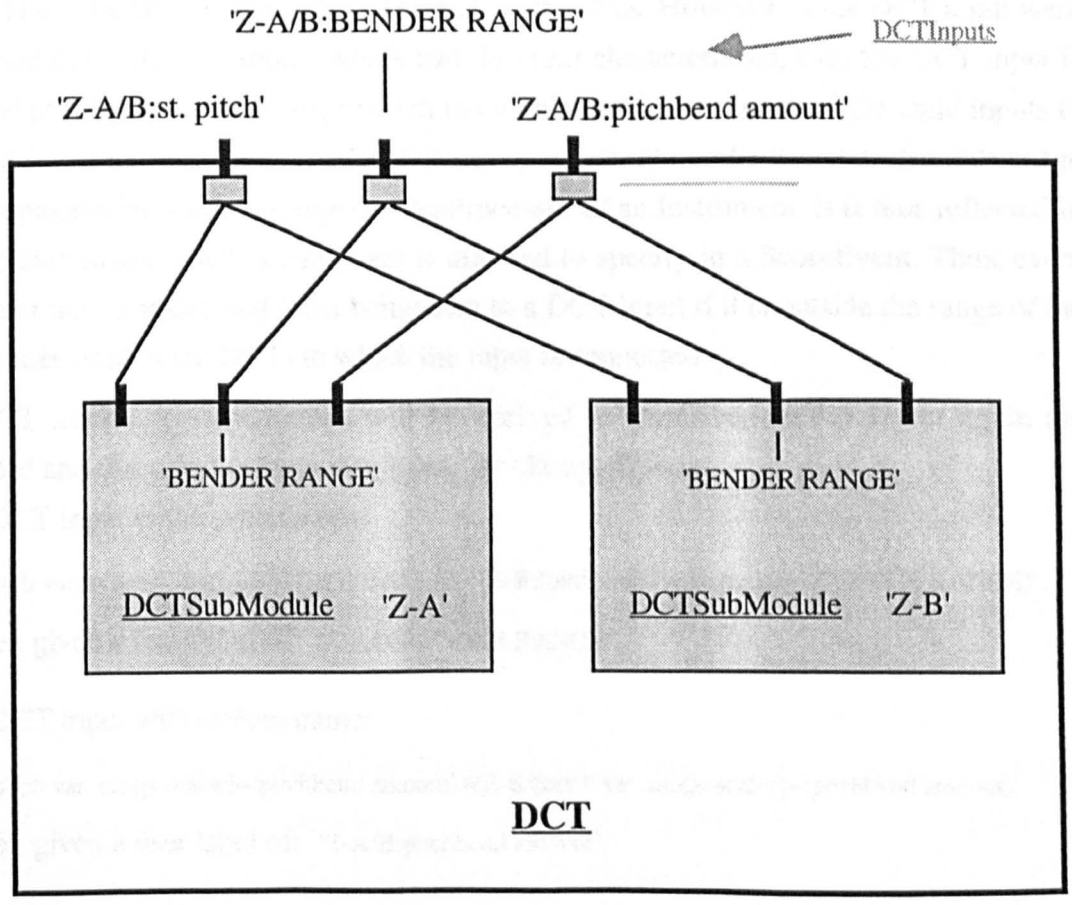


Fig. 70 An example DCT, with three inputs shown

The 'Z' SMT is defined within the category named 'PART' within the DeviceType object which describes the features of a 'D110' device. The user who built this 'Z' SMT has specified one or more MessagePrototypes (see 8.8.2) for each of its inputs. In this example, the inputs all employ MessagePrototypes defined within the Protocol named

'MIDI' in E-Scape. The maximum and minimum values have also been specified by the SMT designer using information from the 'D110' device manual as follows:

SMT input name	min. value	max. value	default value
'BENDER RANGE'	0	36	0
'st. pitch'	12	108	60
'pitchbend amount'	0	127	64

Each submodule of this type thus has inputs with these specifications, although the user is able to alter the default value, and/or modify the minimum value upwards, or the maximum value downwards.

The designer of a DCT will specify and connect its inputs. This example DCT has three inputs illustrated above, each of which is connected to *both* child submodules. In this example these inputs are both the same, and the user has *not* modified the specifications of either. Thus, the DCT inputs inherit these characteristics. However, if the DCT input were connected to submodule inputs which had *different* characteristics, then the DCT input is assigned (by default) a data range which fits within the range of each of the child inputs to which it is connected. When the DCT is incorporated into an Instrument, this data range then propagates upwards through the PspProcessor of an Instrument. It is thus reflected in the Psp data range which a composer is allowed to specify in a ScoreEvent. Thus, event parameter data is prevented from being sent to a DCT input if it is outside the range of the submodules (within the DCT) to which the input is connected.

The DCT inputs' system names will be derived as described in 8.6.3, but again are illustrated and discussed using a user label, for clarity. Thus:

• The DCT input with system name:

(Z-A:[pitch var. range scaler]-<BENDER RANGE)+(Z-B:[pitch var. range scaler]-<BENDER RANGE)',

has been given a user label of: 'Z-A/B:BENDER RANGE'.

• The DCT input with system name:

'(Z-A:[pitch var. range scaler]-<pitchbend amount)+(Z-B:[pitch var. range scaler]-<pitchbend amount)'

has been given a user label of: 'Z-A/B:pitchbend amount'.

• The DCT input with system name: '(Z-A:st. pitch)+(Z-B:st. pitch)'

has been given a user label of: 'Z-A/B:st. pitch'.

For example, the 'Z-A/B:BENDER RANGE' DCT input connects to the 'BENDER RANGE' input of *two* child submodules labelled 'Z-A' and 'Z-B'.

These submodules are then unravelled into device-level DCTPrimitive objects. In this case the submodules are already 'device level', being built from an SMT which is in the

DTSMTCategory named 'PART', as described above. Thus, the flattened structure of the DCT looks identical to its module structure in the above figure, except that DCTPrimitive objects have taken the place of the submodules, as shown in figure 71 below. The DCT inputs now connect to the inputs of the DCTPrimitives.

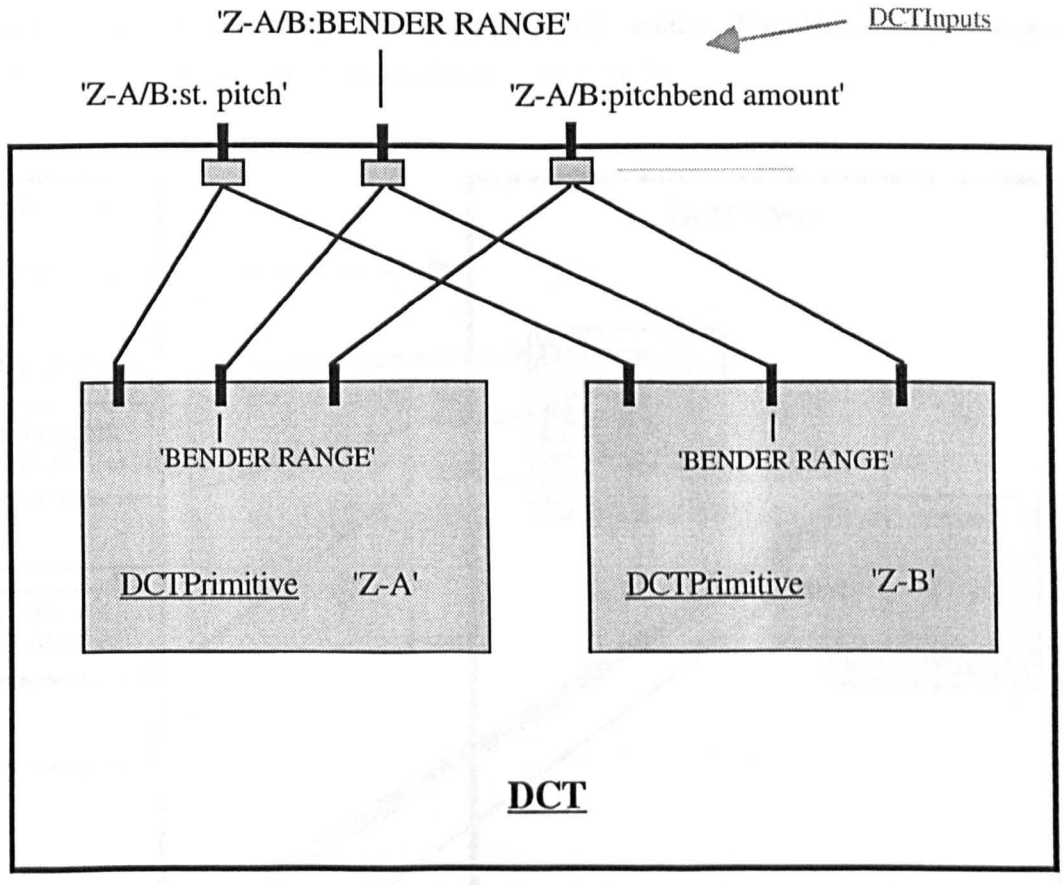


Fig. 71 The example DCT, now showing a 'flattened' structure of 'device-level' DCTPrimitives

The DCT inputs are used by a PspProcessor when assigning processed data within an Instrument, as described in 8.4.4. The DCTPrimitives within the DCT have inputs which are connected to its inputs.

Each DCTPrimitive has been derived from a module, which in turn has a template module type (SMT or PrimSMT) from which it has been derived (see 8.6). Thus a DCTPrimitive can reference a template module type object. Each input of such a module type has one or more MessagePrototypes, which have been defined by the user who has created the module type (see 8.6.2.1). These MessagePrototypes are 'inherited' by each DCTPrimitive input, and will be used subsequently (see 9.3.2) when messages are formulated to install the unit (which the DCTPrimitive describes) in a specific device.

9.2. Creation of ScoreEvents

9.2.1. Creating ScoreEvents using an Instrument

A new Score Event is constructed using a specified Instrument. A new 'blank' ScoreEvent is first created (ie with its data structures in place but unfilled), then the Instrument is assigned to the new ScoreEvent's *instrument* instance variable. The relationship between a ScoreEvent and its Instrument is shown below, in figure 72

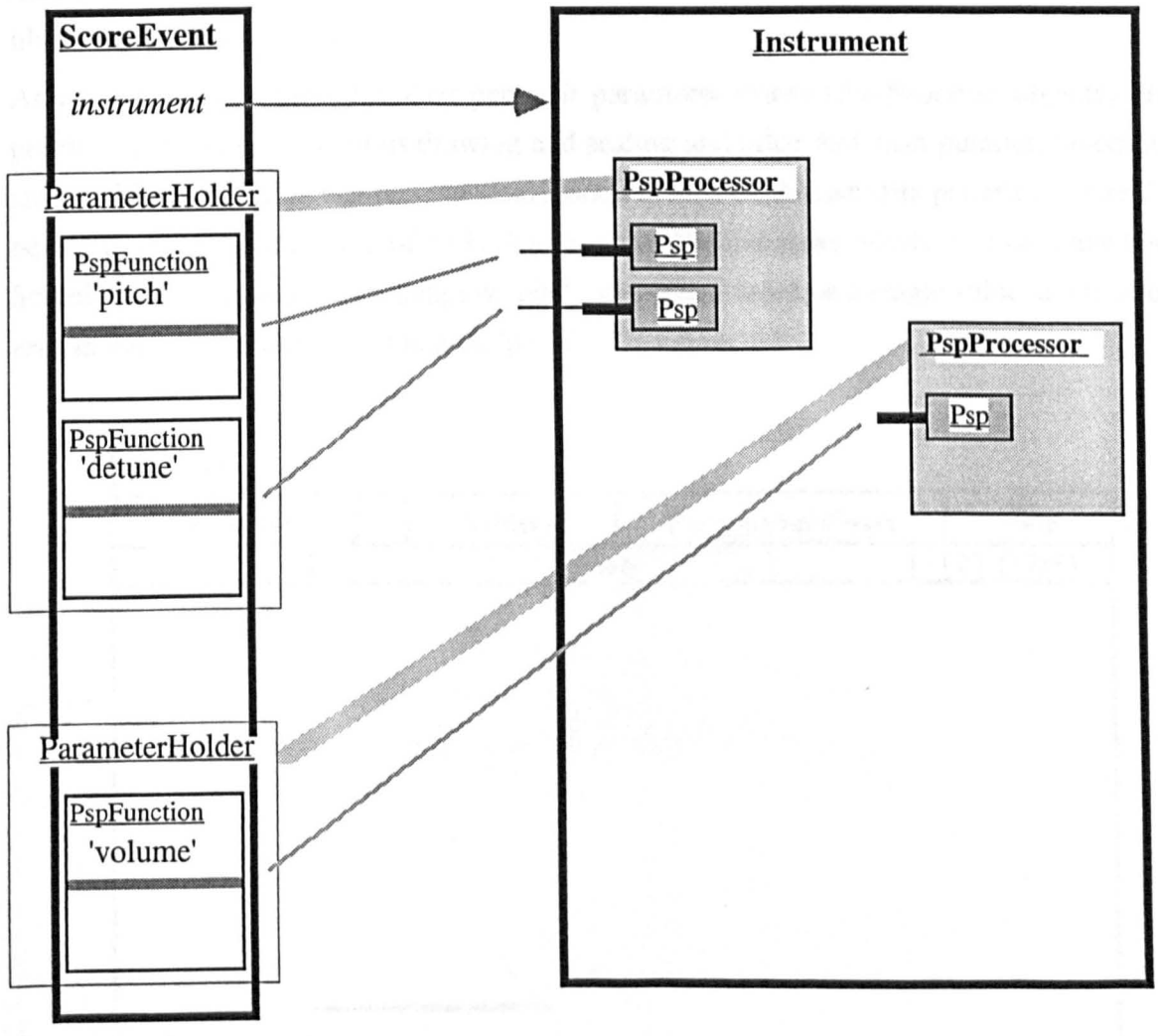


Fig. 72 A ScoreEvent's structure related to its Instrument

A ParameterHolder is then created within the new ScoreEvent, corresponding to each PspProcessor of the Instrument. Each ParameterHolder then creates one or more PspFunctions, each corresponding to a Psp of its associated PspProcessor. A PspFunction can contain one or more breakpoints, consisting of a time and value. However, when a

PspFunction is first created, it is loaded with a *single* breakpoint with a time of zero[1], and a value equal to the default value of the Psp. This value will be thus be active throughout the ScoreEvent, if no other values are specified subsequently. Thus the ScoreEvent is constructed according to the structure and data present in its assigned Instrument. A user can then subsequently enter time-varying data as breakpoints (time and value) in the PspFunction.

A ParameterHolder object thus consists of a subset of PspFunctions, whose values are all processed by a single PspProcessor in the Instrument. When a ScoreEvent is assigned to an Instrument, it will contain a ParameterHolder for each PspProcessor of that Instrument as illustrated in the above figure.

At any subsequent time, the user can edit parameter traces (PspFunction objects) via graphic or text editors. Various drawing and scaling and other data manipulation functions are provided, and traces can be saved and loaded to disk as abstract data patterns. Figure 73 below shows a screen dump of an E-Scape score (SSE) display window. It contains two ScoreEvents, both with an unchanging 'pitch' parameter value, ie a single value at a time of zero in the PspFunction object named 'pitch'.

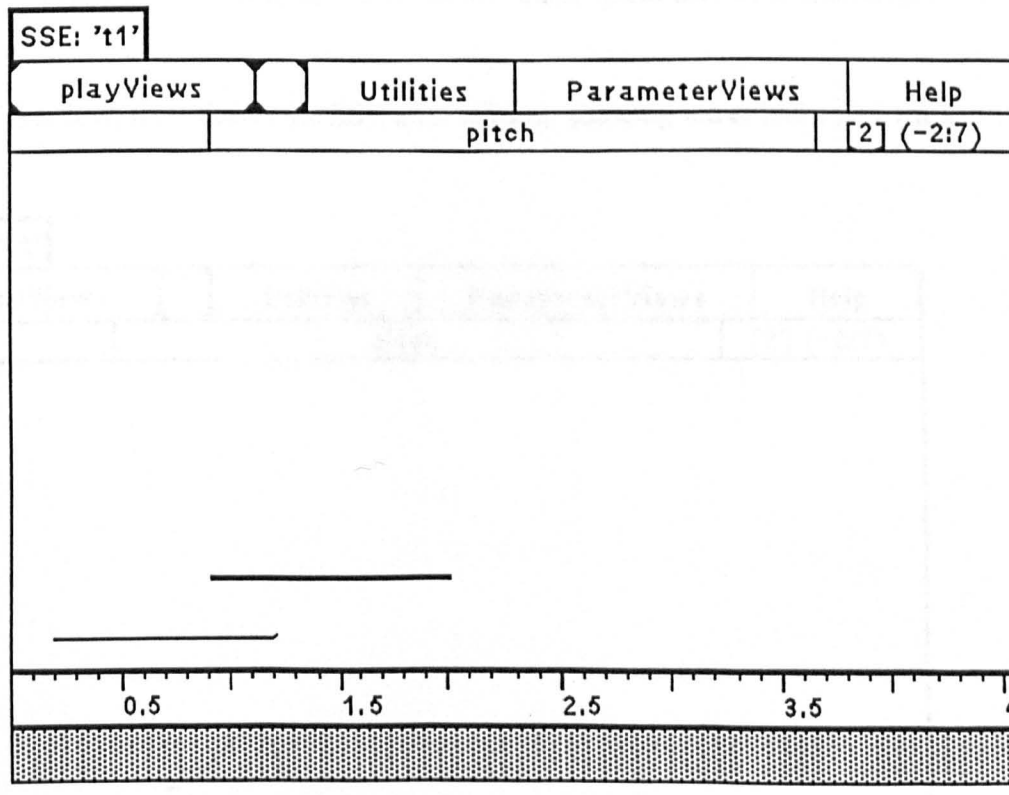


Fig. 73 An SSE with two ScoreEvents, showing 'pitch' parameter before editing

---

1 This is assuming the Psp's *rate* is not 'e' - implying that a value may only be specified for this input at the *end* of the event. If so the breakpoint will have a time set to equal the duration of the event.

The second event is then shown in the process of being edited via the graphic editor in the figure 74 below. The quantised values can be clearly seen.

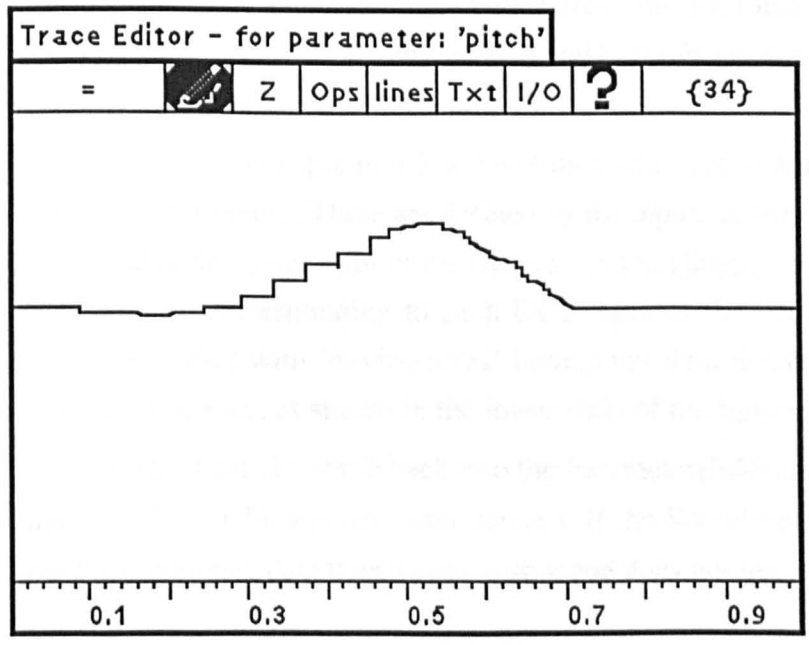


Fig. 74 Graphic editor on the 'pitch' parameter of a ScoreEvent

Figure 75 below, then shows the SSE after editing, showing the edited event trace.

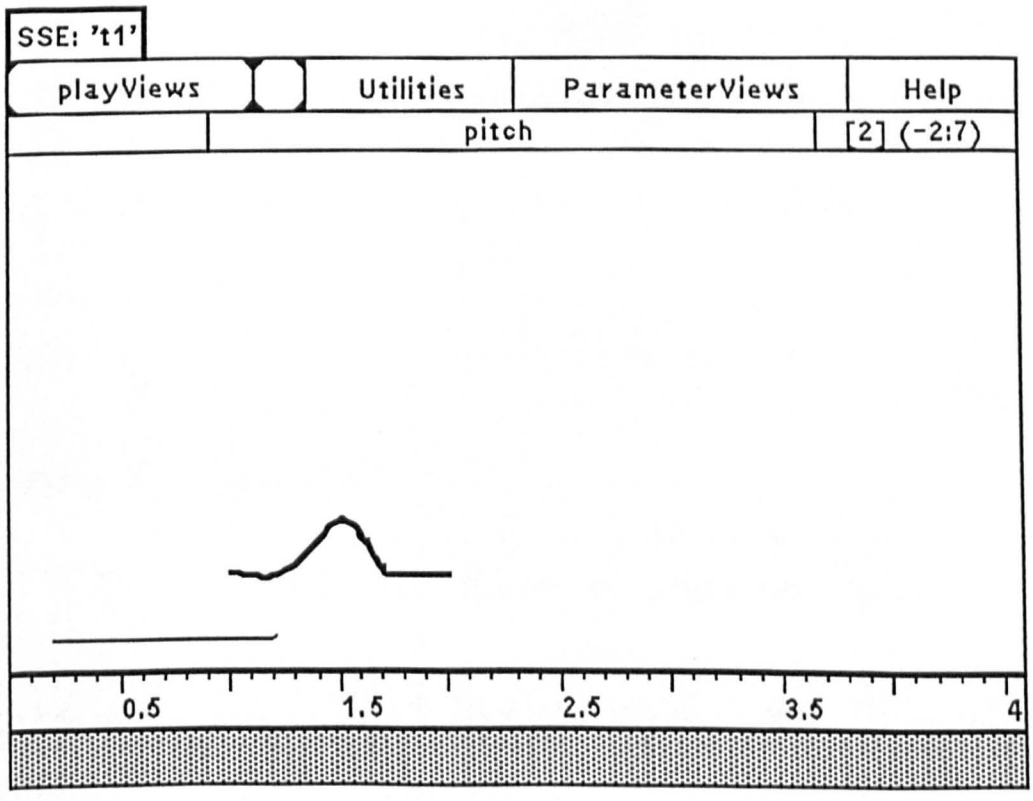


Fig. 75 The SSE display after editing the ScoreEvent, showing its altered 'pitch' parameter.

A ParameterHolder may have more than one PspFunction, data from all of which is needed by its associated PspProcessor. When a value changes in any *one* of a ParameterHolder's PspFunctions, the data in *all* of them will be needed by its associated PspProcessor, as the processing will depend on some *combination* of data from the PspFunctions. This is illustrated in the upper half of figure 76 below, where a single PspProcessor receives data from *two* PspFunctions in a ScoreEvent.

The PspProcessor will then process this into low-level data which is assigned to one or more inputs of the synthesis structure. These are defined by the inputs of the DCT object in the Instrument (illustrated in the lower right of the figure). A DCTSignal object is created within the ParameterHolder, corresponding to each DCT input of the Instrument. Each DCTSignal will later be loaded with 'device-level' breakpoint data derived from these PspFunctions by the PspProcessor, as shown in the lower right of the figure.

Thus, the low-level processed data is stored back into the ParameterHolder, and is thereby tied to the PspFunctions from which is has been derived. If the ScoreEvent is moved in time within the score, its low-level data thus moves with it and does not need reprocessing.

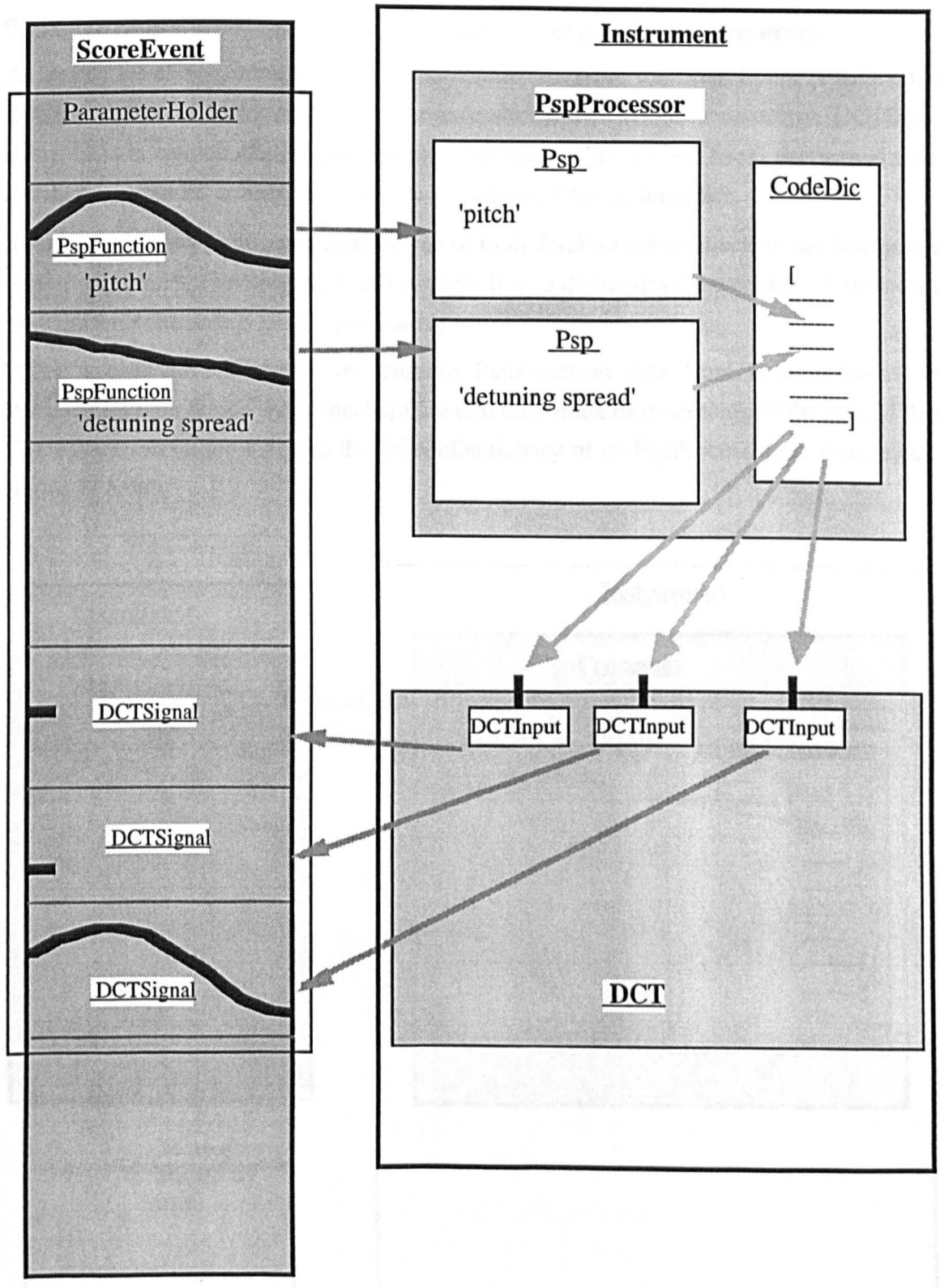


Fig. 76 Score data processed by a PspProcessor into low-level data - all associated data contained in a ParameterHolder

A ParameterHolder object thus has three chief instance variables:

- *pspFunctions* - a set of PspFunctions containing score parameter values.
- *pspProcessor* - assigned to a single PspProcessor in the Instrument, to which it sends these values.
- *dCTSignals* - a set of DCTSignals which hold the processed parameter data.

9.2.2. Score processing stage 1 - processing score parameters

As described above, composer-specified parameters from the ScoreEvent (PspFunction data) are processed into lower-level input values for device synthesis structures (DCTSignal data). This is termed stage 1 processing, with stage 2 (described later) performed when device resources are actually allocated to a score event for performance.

Stage 1 processing is implemented by one or more PspProcessor objects in the Instrument, which receive PspFunction data and process it into device-level input data. This section now presents the details of this processing.

When a user adds, changes or removes PspFunction data from a ScoreEvent, the ParameterHolder which owns this PspFunction then initiates processing of the altered data. The ParameterHolder accesses the ECodeDictionary of its PspProcessor, as illustrated in figure 77 below.

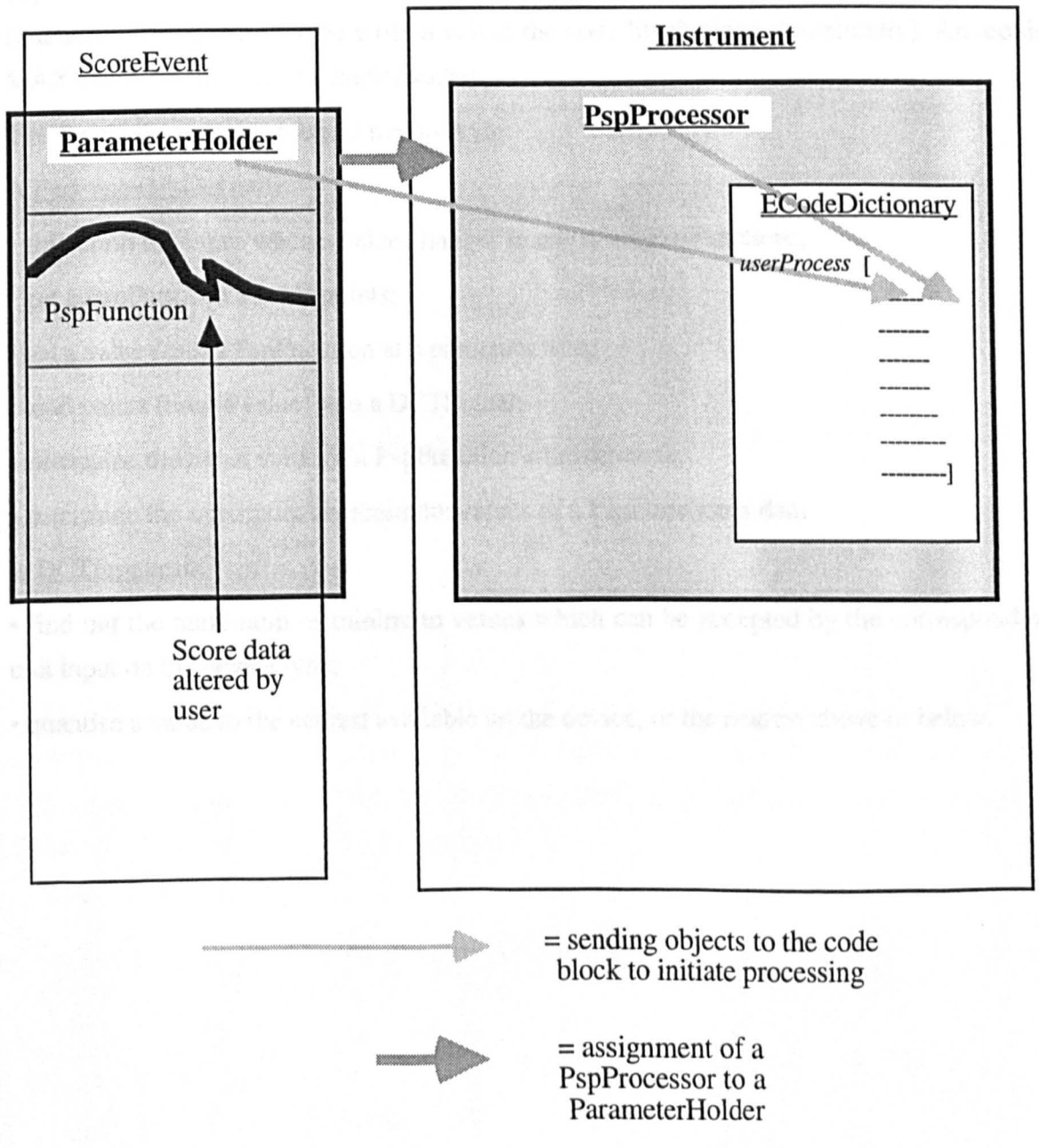


Fig. 77 Initiation of score processing stage 1

Details of the structure of the ECodeDictionary were presented in 8.4.4.2. It contains one or more named blocks of simple Smalltalk code, one of which at least (named 'userProcess') must be present. Objects can be sent as parameters to this code block, which can then be executed.

To instigate data processing, the ParameterHolder calls the 'userProcess' block in the CodeDictionary, sending itself (ie the ParameterHolder) and the PspProcessor object as block parameters. This is illustrated by the faint arrows in the above figure. The code block can then access the parameter data within the PspFunctions (of this ParameterHolder), and the DCT inputs (owned by the PspProcessor) to which it can assign the processed data (see figure 76 above). This will be described in detail in the example of 9.2.2.1.

The code block can use certain 'standard methods' which are provided in E-Scape for use by ParameterHolder and DCTInput objects. At present these have to be written as Smalltalk code within a text editing window which the user can open when creating a PspProcessor object. As described elsewhere, future plans include the provision of menu access to these functions (which will then be written is into the code block text automatically). An iconic MAX-like GUI may also be implemented.

The functions of these standard methods are:

A ParameterHolder can:

- collect *all* the times when a value changes in *any* of its PspFunctions;
- get a PspFunction's breakpoints;
- get a value from a PspFunction at a particular time;
- load points (time@value) into a DCTSignal;
- determine the mean value of a PspFunction's breakpoints;
- determine the maximum or minimum values of a PspFunction's data.

A DCTInput can:

- find out the maximum or minimum values which can be accepted by the corresponding unit input on the device type;
- quantise a value to the nearest available on the device, or the nearest above or below.

9.2.2.1 Example of stage 1 score data processing

An example will best illustrate stage 1 data processing. Figure 78 below shows an example Instrument which has two PspProcessors, and a single DCT which specifies a synthesis structure on a 'D110' device. This DCT was presented earlier as an example in 9.1.3.5. It has many inputs, three of which were shown there. In this figure, an additional input (on the right of the DCT) is shown which is assigned data from the PspProcessor shown on the top right of the Instrument diagram.

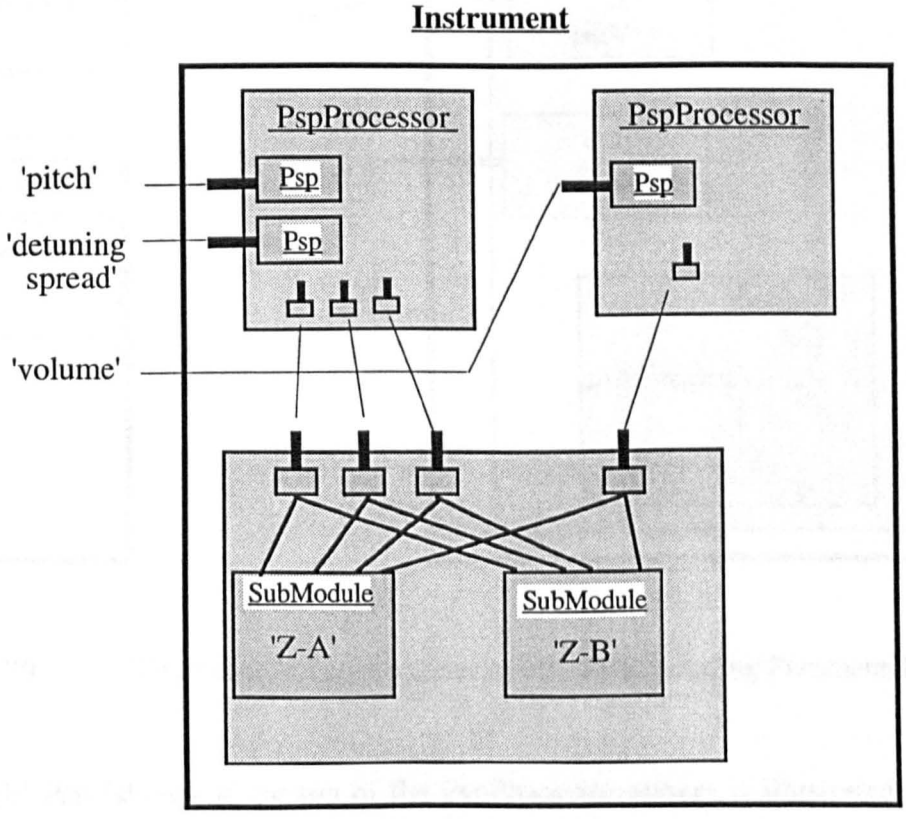


Fig. 78 Structure of the example Instrument

As described above, a ScoreEvent which uses this Instrument will have a ParameterHolder corresponding to each PspProcessor in the Instrument. The example ScoreEvent thus has two ParameterHolders, only the *first* of which will be discussed here. This corresponds to the top left PspProcessor in the above figure.

Figure 79 below shows (on the left) the ParameterHolder which corresponds to this PspProcessor (within its ScoreEvent). The ParameterHolder has two PspFunctions named 'pitch' and 'detuning spread', which correspond to the Psps of the same name in the PspProcessor (shown on the right).

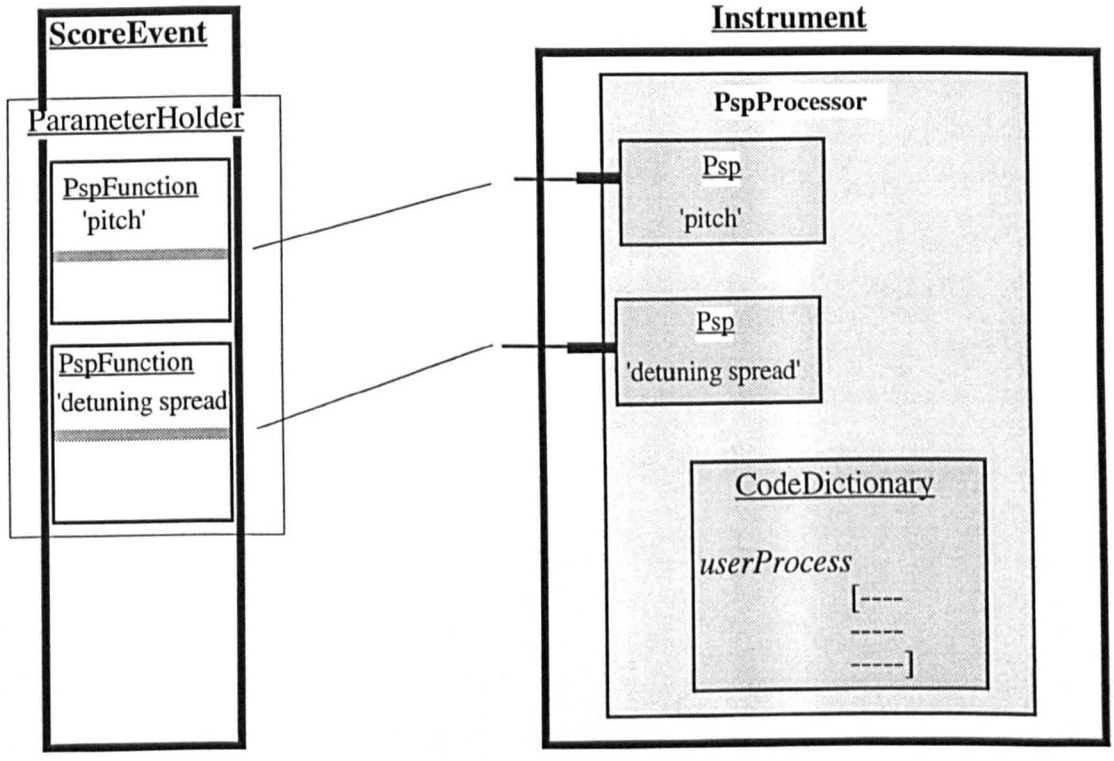


Fig. 79 The example PspProcessor and its corresponding ParameterHolder

The 'pitch' Psp (shown at the top of the PspProcessor above) is illustrated in detail in figure 80 below. The structure of a Psp object was described in 8.4.4.1. It has a BasicUnit object called 'ST from A440'. This describes the interpretation of values for the 'pitch' parameter as representing the deviation in semitones from concert A pitch (440Hz). This parameter has a minimum value of -63, and maximum value of 69, when represented in these units. The *resolution* instance variable is set by the user at 0.01, thus the 'pitch' parameter of a ScoreEvent can be specified to a resolution of 1/100 semitones, ie 1 cent. In this case, the specified values range *uniformly* from the minimum to the maximum value. However, E-Scape also provides the facility to specify non-linear and non-contiguous values, for reasons described in 6.1.4.

As an aside to this explanation, the two Unit objects ('Hz' and 'pch') should be noted in the figure. Each Unit describes the measurement of pitch in Hz and CSound 'pch' [1] notation,

---

1 'Pch' notation combines an octave value (7 = middle C) and semitones within each octave into a single number, with a decimal point separating the two values. For example 7.00 represents middle C, 7.05 represents 5 semitones above middle C, and 7.1160 represents 11 semitones and 60 cents (0.60 semitones) above middle C

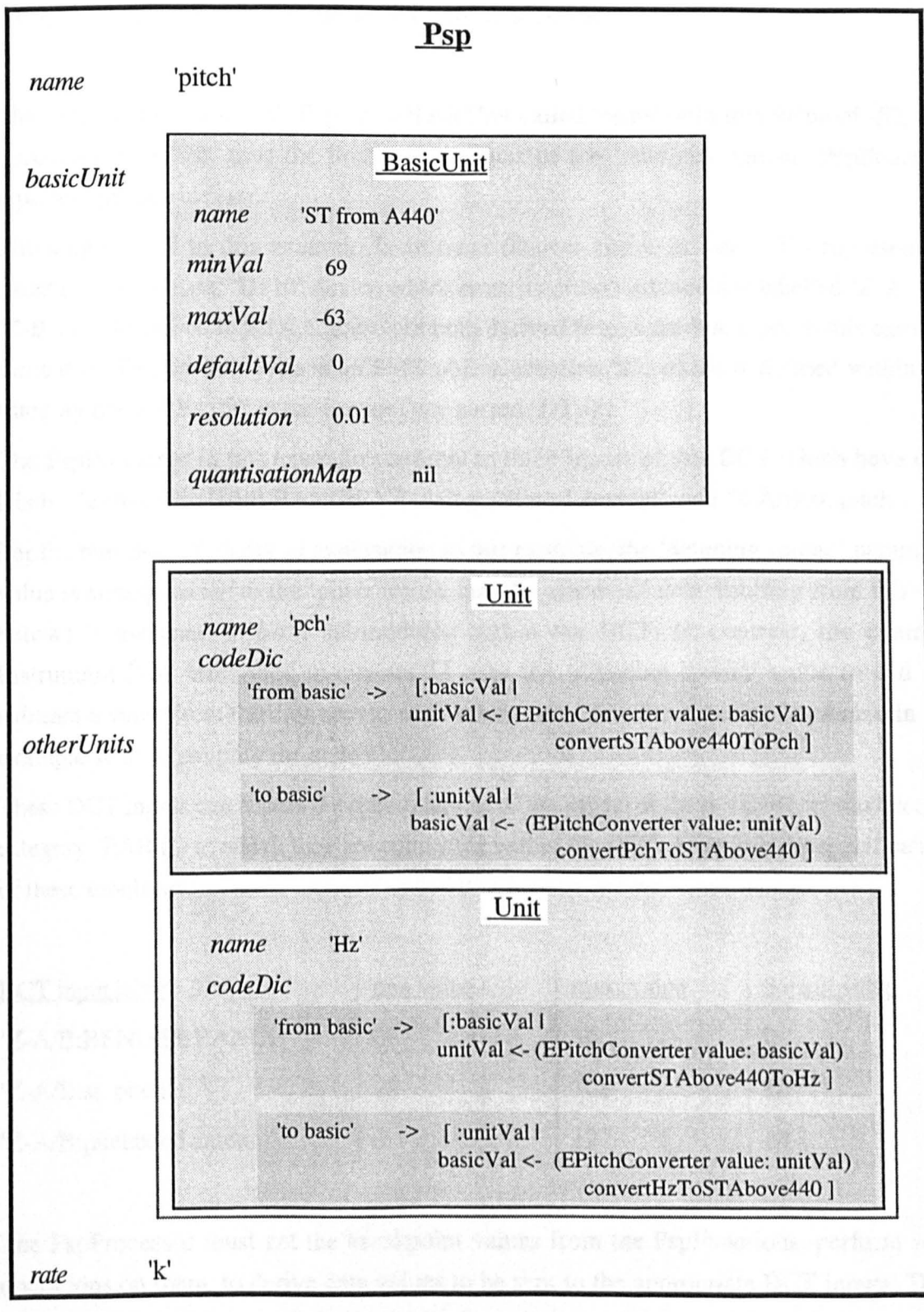


Fig. 80 The 'Pitch' Psp of the example Instrument

and has defined methods to convert values to and from the 'basic' Unit. In this case - as conversion between these units is commonly needed - E-Scape system methods are used, implemented by the specialised EPitchConverter object. A designer of a Psp would normally specify their own code for Unit conversion. Of course many of the more abstract

parameters invented by E-Scape Instrument designers may not have conventional units, but it is important to provide such facilities for those which do.

The other 'detuning spread' Psp has a BasicUnit called 'cents' with min value of -50, and maximum of + 50; thus the breakpoint values of the 'detuning spread' PspFunction represent pitches in cents.

The single DCT in this example Instrument (shown above in figure 78) represents a structure on a Roland 'D110' device which consists of two submodules labelled 'Z-A' and 'Z-B'. As described in 9.1.4.1, these are both derived from a module type, in this case the same one. This module type is an SMT object, labelled 'Z', which is defined within the category named 'PART' in the DeviceType named 'D110'.

The PspProcessor in this example connects to three inputs of this DCT which have user labels: 'Z-A/B:BENDER RANGE', 'Z-A/B:pitchbend amount', and 'Z-A/B:st. pitch'.

For the purposes of clarity of explanation in this example, the 'detuning spread' parameter value is simply *added* to the 'pitch' value, and the processed data resulting from this (see below) is assigned to *both* submodules within the DCT. In contrast, the example Instrument (1b) described in chapter 11 uses the 'detuning spread' value to add and subtract a value from the data sent to each submodule. This functionality is omitted in this example so as to simplify the code block.

These DCT inputs can access the specifications of the inputs of the device-level modules (of category 'PART') to which they are connected within the DCT. The pertinent specifications of these inputs are:-

DCT input label	min value	max. value	default value
'Z-A/B:BENDER RANGE'	0	36	0
'Z-A/B:st. pitch'	12	108	60
'Z-A/B:pitchbend amount'	0	127	64

The PspProcessor must get the breakpoint values from the PspFunctions, perform some operations on them, to derive data values to be sent to the appropriate DCT inputs. These derived values are then loaded back to the ScoreEvent to be stored as breakpoints within a DCTSignal object. The latter are similar to PspFunctions, each being associated with an input of the DCT and storing data to be sent to it.

To achieve this, the PspProcessor uses (as described previously) the 'userProcess' code block of its ECodeDictionary, as illustrated in figure 81. The code block is created from the text which is entered by the user in a text editing window. The left-hand side numbers have been included for reference purposes.

Some further information about Smalltalk syntax is required for the reader who wishes to closely follow the explanation of the code within this block:

Smalltalk messages can be of three types (messages are shown in bold):-

- 'unary' - a single string with no arguments:
  eg. **draw**, **invert**, **play**, **close**.
- 'binary' - a special symbol or symbols with a single argument:
  eg. + 4, * 4, - 4, / 4, == 4, ~= 4, & 4 etc.
- 'keyword' - one or more words, each ending in a colon and followed by an argument:
  eg. **playFrom:** 400
  playFrom: 400 **onPort:** 'MIDI'

Such messages are sent to an object which is called the *receiver* of the message, and appears to the left of the method. The receiver object then carries out the method invoked by the message (using any arguments which are passed to it). If the method results in an object being returned, it can be assigned to another variable on the left using the left arrow character. Some examples are shown below. In each, a 'receiver' object is sent a Smalltalk message.

In the first example the receiver object (assigned to the variable 'sect1') is sent the 'play' message, as a result of which it plays itself (via further functioning, not shown).

In the second example, the 'sect1' receiver object is sent the 'invert' message. In response to this it performs some operations (again not shown) after which it returns an object (perhaps an inverted copy of itself). This returned object is then assigned to the 'newS' variable.

variable	result passed	receiver object	message sent to receiver object
		sect1	play
newS	<-	sect1	invert
		sect1	playFrom: 400 onPort: 'MIDI'
newP	<-	notePitch	invert
newP	<-	notePitch	+ 4
		notePitch	playFrom: 400 onPort: 'MIDI'
newP	<-	notePitch	+ 4 / 5 * 2

ECodeDictionary

'userProcess' =>

DBlockContext

sourceString =

```
'[
"1"   :paramHolder :pspProcessor |

"2"   tPitchFunc <- paramHolder pspFunctionNamed: 'pitch'.
"3"   tDetuneFunc <- paramHolder  pspFunctionNamed: 'detuning spread'.

"4"   times <- paramHolder allTimes.
"5"   tPitchPoints <- times collect: [:t |
          t @ ((tPitchFunc valueAt: t) + ((tDetuneFunc valueAt: t) / 100) )
      ].

"6"   tCentreVal <- EFunction integerCentreOf: tPitchPoints.
"7"   tDeviation <- EFunction integerMaxDeviationOf: tPitchPoints
                                fromCentreValue: tCentreVal.

"8"   tPbDeviceInput <- pspProcessor
                                    outputNamed: 'Z-A/B:BENDER RANGE'.
"9"   tPBSensitivity <- tPbDeviceInput nearestDeviceValAbove: tDeviation abs.

"10"  pspProcessor removeAll.

"11"  pspProcessor loadVal: tPBSensitivity
                    toDCTInputNamed: 'Z-A/B:BENDER RANGE'.

"12"  tPitchPoints do: [:eachPt |
"13"      newPt <- eachPt processVal: [:val |
"14"      ((val - tCentreVal) divBySafe: tPBSensitivity) * 63 + 64 ].
"15"      pspProcessor loadPoint: newPt
                    toDCTInputNamed: 'Z-A/B:pitchbend amount'
      ].

"16"  pspProcessor  loadVal: (tCentreVal + 69)
                    toDCTInputNamed: 'Z-A/B:st. pitch'

]'
```

Fig. 81 The 'userProcess' code block in the 'pitch' PspProcessor

Temporary variables may be freely used within a block, and a convention of prefacing each with the letter 't' has been adopted to aid readability. Variables are un-typed (as are all Smalltalk variables).

This block performs the following stages of processing (the numbers referring to each line of code within the block):

(1) Two variables are passed into this block:

• the PspProcessor (within which this block exists). The block can then access the DCTInputs owned by the PspProcessor to send it output values.

• the ParameterHolder (owned by the ScoreEvent) which corresponds to this PspProcessor. The block can then access the PspFunctions owned by this ParameterHolder to get its input values.

(2) The PspFunction named 'pitch' is obtained from the DCTHolder, and assigned to a temporary variable.

(3) The PspFunction named 'detuning spread' is similarly obtained from the DCTHolder, and assigned to another temporary variable.

(4-5) For each time that *either* the 'pitch' or 'detuning spread' PspFunction value changes, the absolute pitch is calculated. This is done in the following steps:-

(4) The ParameterHolder is asked to return all the times when either of its PspFunctions changes value, using a 'standard' supplied method. This is why all the PspFunctions which correspond to Psps of a *single* PspProcessor are *grouped* within a ScoreEvent into a ParameterHolder. In this case, as the PspProcessor's output values will depend on the value of *both* these PspFunctions, a new calculation is needed whenever *either* PspFunction changes, and the PspProcessor must be able to access both.

(5) A loop is performed (the Smalltalk method 'collect: []') using all these times. At each time, the value of the 'pitch' PspFunction is obtained (using the standard 'valueAt: t' method). The value of the 'detuning spread' PspFunction is similarly obtained, divided by 100 (as its basic units are *cents*), then added to the 'pitch' value (whose basic unit is *semitones away from* concert A pitch, ie A 440Hz). In future it is planned to have a standard 'userFunction' (see 8.4.4.2) to do this processing (in lines 4-5) for all the time points.

(6-9) The breakpoint values (stored in the 'tPitchPoints' temporary variable) now represent pitches (including decimal fractions) expressed as semitones from A440. They must now be converted to values which can be later encapsulated within the appropriate message format, and sent to one or more inputs of units on the device in order to specify such a pitch.

This will involve conversion to data to be assigned to *three* DCT inputs.

• First a value for its 'Z-A/B:st. pitch' input. This sets the pitch of a note in semitones - eventually being sent to the device via a MIDI 'pitch' field of a 'note on' Message.

• Then a value for its 'Z-A/B:BENDER RANGE' input. This specifies the maximum pitch deviation (in semitones) from the specified semitone pitch, which can be effected by the full range of values sent to the third DCT input - 'A/B:pitchbend amount'.

Thus:-

(6) Determine the mean (central) value of the breakpoint pitch values (now expressed in semitones). This is an example of the use of a utility EFunction class which provides a variety of useful mathematical functions. Here it returns the mean value of a set of breakpoints (ignoring the times), quantised to the nearest integer.

(7) Another function returns the maximum deviation of a set of points from a specified value, quantised *up* to the nearest integer. This gives the maximum semitone pitch variation to send to the 'Z-A/B:BENDER RANGE' input.

For example, a set of pitch points might vary from -20.14 to +18.30. Thus, the mean integer value is -1, and the maximum deviation is 19.3, which is rounded up to 20. Thus a value of 20 should in theory be sent to the 'Z-A/B:BENDER RANGE' input. This is not the whole story, however, as this input may not actually have 20 as an allowable value. Thus:-

(8-9) Fetch the 'Z-A/B:BENDER RANGE' DCT input (remembering that the PspProcessor owns the DCTInputs it is notionally connected to), and ask it (using the standard user message 'nearestDeviceValAbove: ') for the nearest value this input will accept above this. In this case, the 'Z-A/B:BENDER RANGE' input has a range of 0 to 36 in steps of 1, thus will return 20 as an acceptable value. However, other devices may have a range of 0-7, 0-12, or 0-24, or a restricted set of values, eg 0, 1, 2, 7, 12. Thus the input must be interrogated to find the nearest value, or report an error if a value outside the input's range is being asked for.

(10) Remove all the *existing* processed values from all DCTSignals. Future improvements could selectively remove only those values within the time span which has been altered, but to search through the DCTSignal for these is likely to take up time, so the improvement in speed may be unspectacular.

(11) Load the value into a DCTSignal (see figure 76) which is associated with the DCT input named 'Z-A/B:BENDER RANGE'. Because no time is specified, the single value is stored automatically at time = 0 within the DCTSignal. See figure 84 below for an illustration of this.

(12-15) Calculate the values to send to the 'Z-A/B:pitchbend amount' input. This input controls the degree of pitch deviation from the semitone pitch specified at the 'Z-A/B:st. pitch' input. It has a range of 0 - 127, with 64 implying no deviation. 127 implies a maximum positive, and 0 a maximum negative deviation respectively of the amount specified by the 'Z-A/B:BENDER RANGE' input (eg 20 semitones in this example).

(12) For each pitch point the following is carried out:-

(13) Process the point's value (ie ignore the time), using the following Smalltalk block...

(14) Get the value's deviation from the central pitch (eg -1), and perform some simple mathematical operations to bring it within the 0-127 range, taking into account the value already sent to the 'Z-A/B:BENDER RANGE' input.

(15) Load the processes to the DCT input named 'A/B:pitchbend amount'.

(16) The 'A/B:st. pitch' DCT input uses the 'pitch' field of the MIDI 'note on' Message, with values specifying pitch in semitones (69 representing A440 on this device[1]). Thus, the value + 69 (with time = 0) is loaded to the DCT input named 'Z-A/B:st. pitch'.

Thus, connections are set up within the code block with the Psp objects which define the inputs to the PspProcessor. Each of these Psps (in the Instrument) then correlates with a PspFunction (in the ScoreEvent). Connections are also set up in the code block with inputs of the DCT in the Instrument. Each of these DCT inputs then correlates with a DCTSignal in the ScoreEvent. These correlations are shown in figure 82 below. Data processing will then be carried out according to these links.

The following figure 83 then illustrates this processing of ScoreEvent data from PspFunction data - which is entered and seen by the user - into DCT input data which is associated with particular inputs of a synthesis structure. E-Scape implements this processing as follows:

When score parameter data for either the 'pitch' and 'detuning spread' parameters is added or edited (ie values or times are altered in either PspFunction), the ParameterHolder calls the *userProcess* block in the PspProcessor's ECodeDictionary. It sends itself, along with the PspProcessor, as block parameters (:parameterHolder and :pspProcessor), as shown in line "1" of the code block above in figure 81.

This code block then performs the steps described above; taking in the values from the 'pitch' and 'detuning spread' parameters of a ScoreEvent, to derive data which are stored back into the ScoreEvent within three DCTInputSignal objects, shown bottom left.

This processed data in a DCTSignal has an assignment to a particular input of a synthesis structure by virtue of its association with a corresponding DCT input, as shown in figure 82. However, each DCTSignal still consists of simple breakpoints, and contains nothing which is specific to a *particular* device. This is because the ScoreEvent has not yet been placed in a score in conjunction with others and thus had device resources allocated to perform it. Processing of this DCTSignal data into actual messages addressed to *specific* units/addresses within a *specific* device is performed in stage 2 of score processing (to be described in 9.3.2).

---

1 Assuming that other inputs (eg fine tuning) have been left at zero deviation. Some other MIDI devices have a tuning map which can specify a specific pitch for each semitone note value. A data dump specifying a standardised relationship could be defined as part of an initialisation message sent to the device at system boot up.

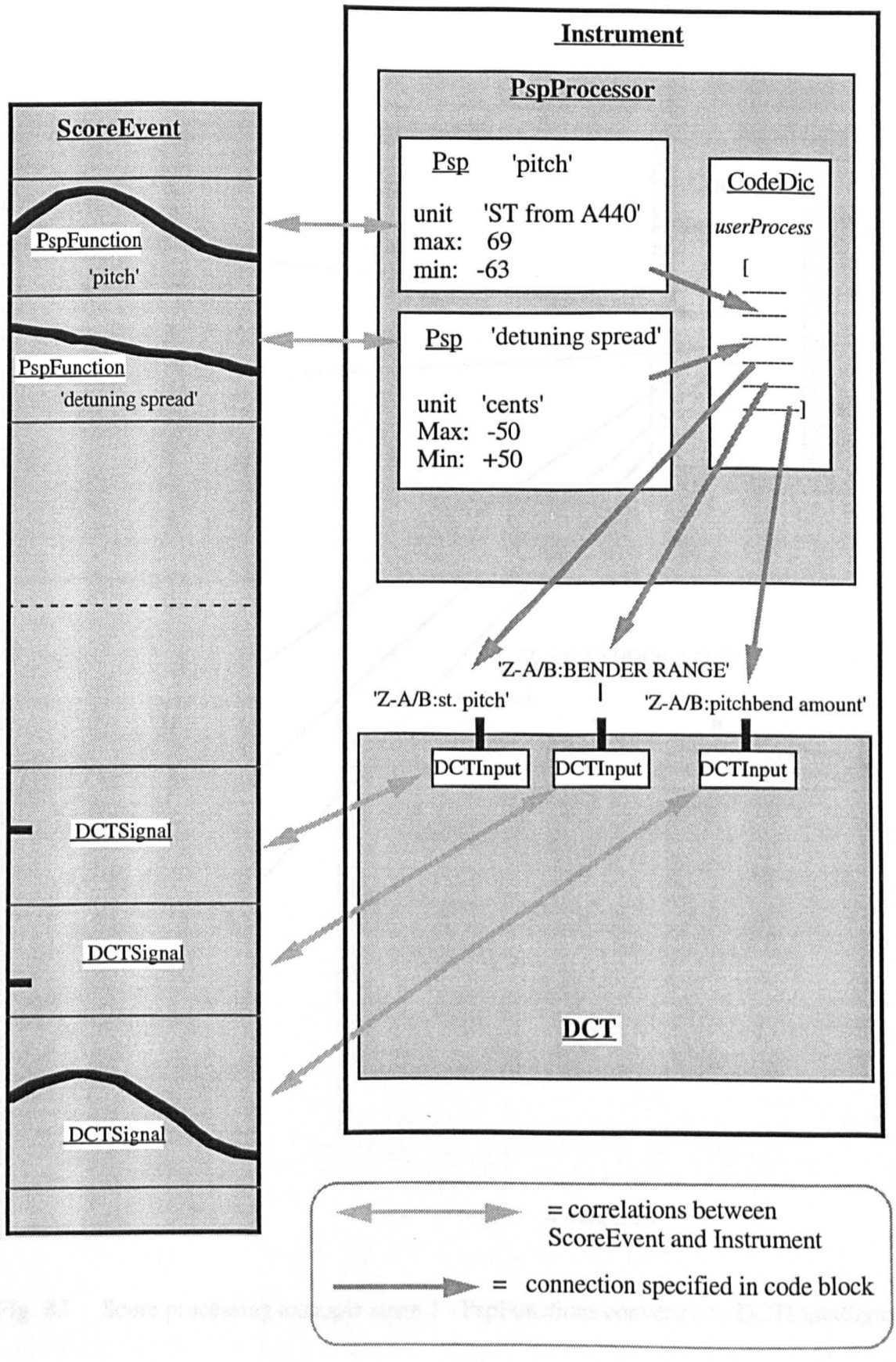


Fig. 82 Score processing example - stage 1

Connections defined within the ECodeDictionary of a PspProcessor, with resulting correlations between ScoreEvent and Instrument

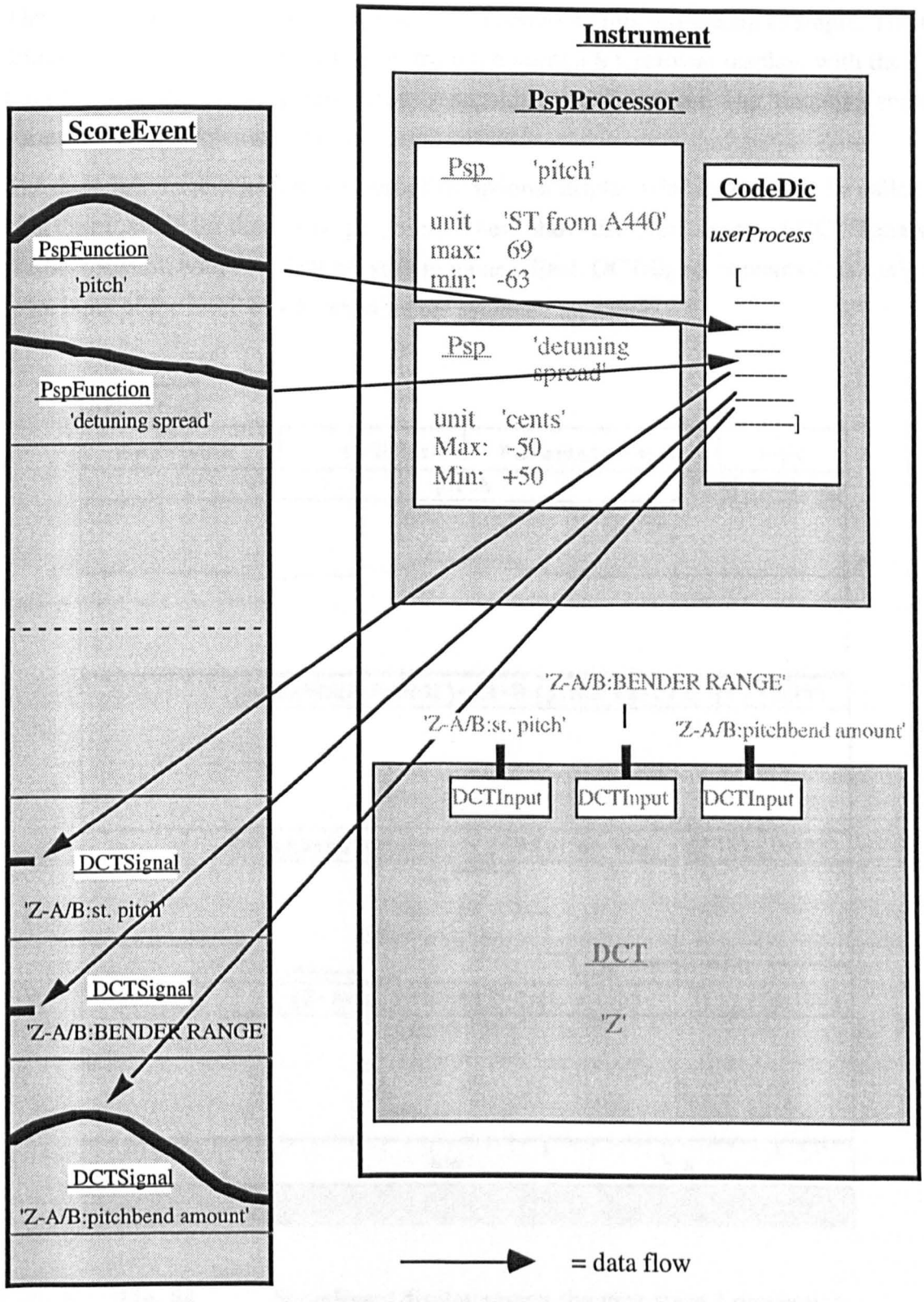


Fig. 83 Score processing example stage 1 - PspFunctions converted to DCTInputSignals

If the ScoreEvent is *moved* in time within a score, these particular device-level messages are likely to need to change, as different device resources may be allocated to perform the ScoreEvent in its new location (relative to other ScoreEvents). However, these DCTSignal values created by the stage 1 processing just described will *not* change under these circumstances. Thus it makes sense to store it with the ScoreEvent.

The screen shots in figure 84 below show the results of this processing example. The top window in the figure shows a section from the normal ScoreEvent display, with the data trace for the 'pitch' (Psp) parameter, as seen and entered by a user. The 'detuning spread' parameter is not shown in the figure.

The three lower sub-windows are part of an optional display window that may be called up by a user (mainly for diagnostic purposes). These show the three processed DCTSignals of the ScoreEvent, with their (long!) system names. Each DCTSignal contains data assigned to an input of the DCT which describes the synthesis structure.

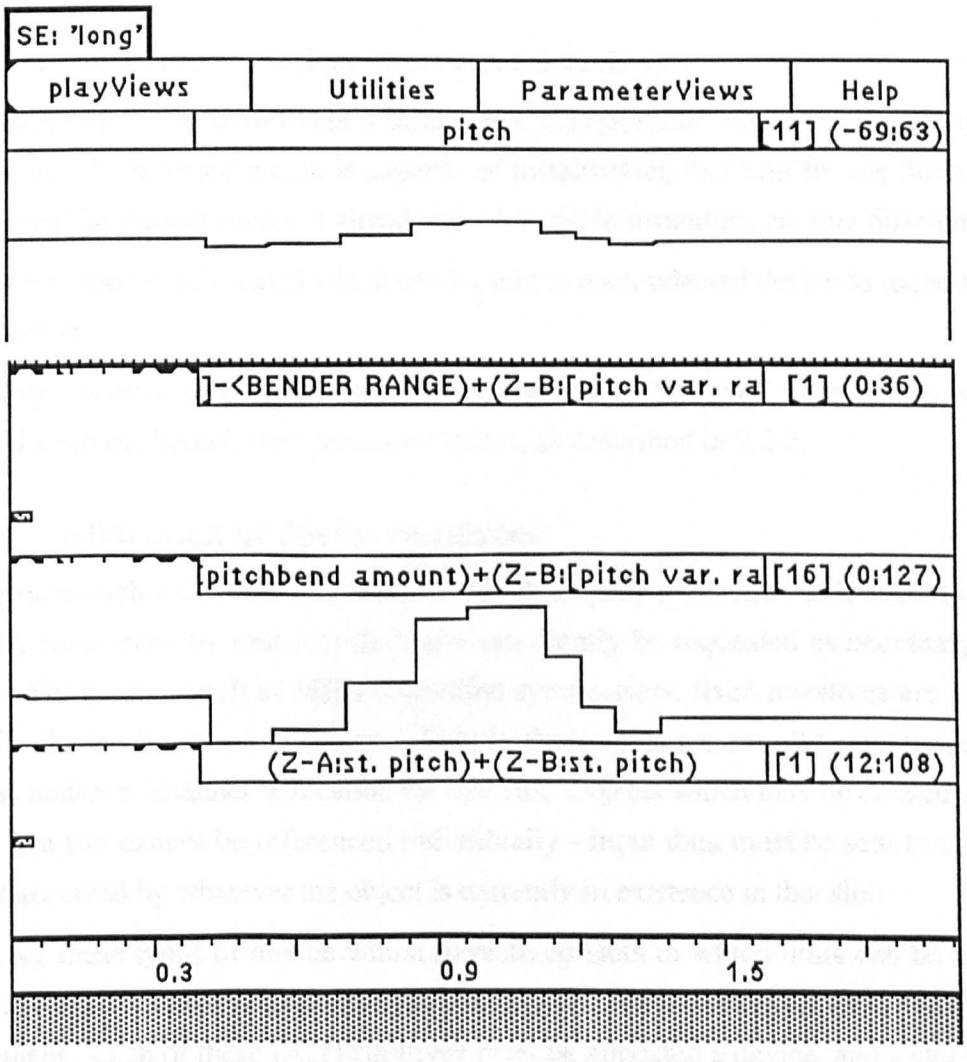


Fig. 84 ScoreEvent display screen showing stage 1 processing

The data in the 'BENDER RANGE' and 'st. pitch' DCTSignal consists of a single value, show as a small graphic block at the extreme left hand side. This is because these DCT inputs are 'i' rate (see 8.3), hence values only apply to, and affect, the *onset* of the ScoreEvent.

9.3. Preparing ScoreEvents for performance on devices

To perform (ie play or realise) a ScoreEvent, it must be loaded into a score, described by an SSE (ScoreSuperEvent) object. The ScoreEvent will have a specific time offset within - ie relative to the start of - the SSE. As described above, such a ScoreEvent has an Instrument which contains one or more DCT objects. Each of these contains a network of one or more DCTPrimitive objects, each of which describes a device-level unit in a device. Each of these units must then be *instantiated* (ie created) in an appropriate device in order to play this ScoreEvent. Instantiated units can then be instructed to start running, and have score data sent to their inputs.

Thus, three things need to be done to perform a ScoreEvent:

- For each unit in the ScoreEvent's Instrument, a *device*, and a *location* ('slot') within that device must be selected which is capable of instantiating that unit for the duration of the ScoreEvent (ie the slot must not already be allocated to instantiate another different unit.
- Messages need to be created which can be sent to each selected device to request each unit instantiation.
- Messages need to be created which can convey the device-level input values which have resulted from the ScoreEvent parameter traces, as described in 9.2.2.

9.3.1 Allocation of device resources

For devices such a CSound and MIDAS which employ a dynamic unit allocation scheme (usually referenced by unit ids) the units can simply be requested as necessary from the device. For devices such as MIDI controlled synthesisers, fixed resources are available in specific device locations or 'slots'. Units in these slots are usually referenced using an address and/or a 'channel' allocation for that slot. Objects which may be created at different times in a slot cannot be referenced individually - input data must be sent to the *slot*, and will be received by whatever the object is currently in existence in that slot.

Thus, for these types of device which have fixed slots in which units can be allocated, a more involved scheme is needed. Each ScoreEvent has a number of DCTPrimitives in its Instrument. Each of these DCTPrimitives must be allocated a device, and a slot within that device. The selected slot must be capable of instantiating the unit described by the DCTPrimitive for at least the duration of the ScoreEvent. This capability depends on whether other units are allocated in the slot during this time interval.

9.3.1.1 Allocation description objects

Each synthesis device which is connected and available to E-Scape is described by a Device object, as described in 8.6 above. An SSE which is to be performed now needs to allocate device resources for each of its ScoreEvents. It thus requires some additional objects which can *describe* allocations of devices and resources within them, to instantiate units for the duration of each ScoreEvent.

The SSE thus looks at the Instrument of each of its ScoreEvents. Each Instrument contains one or more DCT objects, each of which describes a synthesis structure on a *single* type of device, (described by a DeviceType object), as stated previously. Each Instrument can thus refer to one or more DeviceType objects which describe the types of device needed to perform its structures. So, for each of its ScoreEvents, the parent SSE looks at the DeviceType(s) of its Instrument, and examines the E-Scape system (see 9.6) to see which Device objects are available. Each Device corresponds to a *particular* individual device of this type which is presently connected to E-Scape.

Note that in an object-oriented system, this description of what the system *does* is actually matched by the activities performed at the programming level. The SSE object asks (ie sends an appropriate Smalltalk message) each ScoreEvent for its Instrument. It then sends a message to this Instrument object to ask for its DCT objects. It then asks each DCT object for its DeviceType. The SSE then asks the E-Scape system (an object called SystemResource - see 9.6)) to give it the Device objects of each of these DeviceTypes.

The SSE then creates a new ScheduledDevice object corresponding to each of these Devices. This ScheduledDevice describes the *allocation* of the resources in a device needed to perform this SSE. Each 'parent' Device has slots (DeviceDCIPrimitiveSlots) which may have 'channel', 'id', or address parameters, as described in 8.7. In the ScheduledDevice, a ScheduledDCIPrimitiveSlot object is created to correspond to each of these 'template' DeviceDCIPrimitiveSlots, as shown below in figure 85.

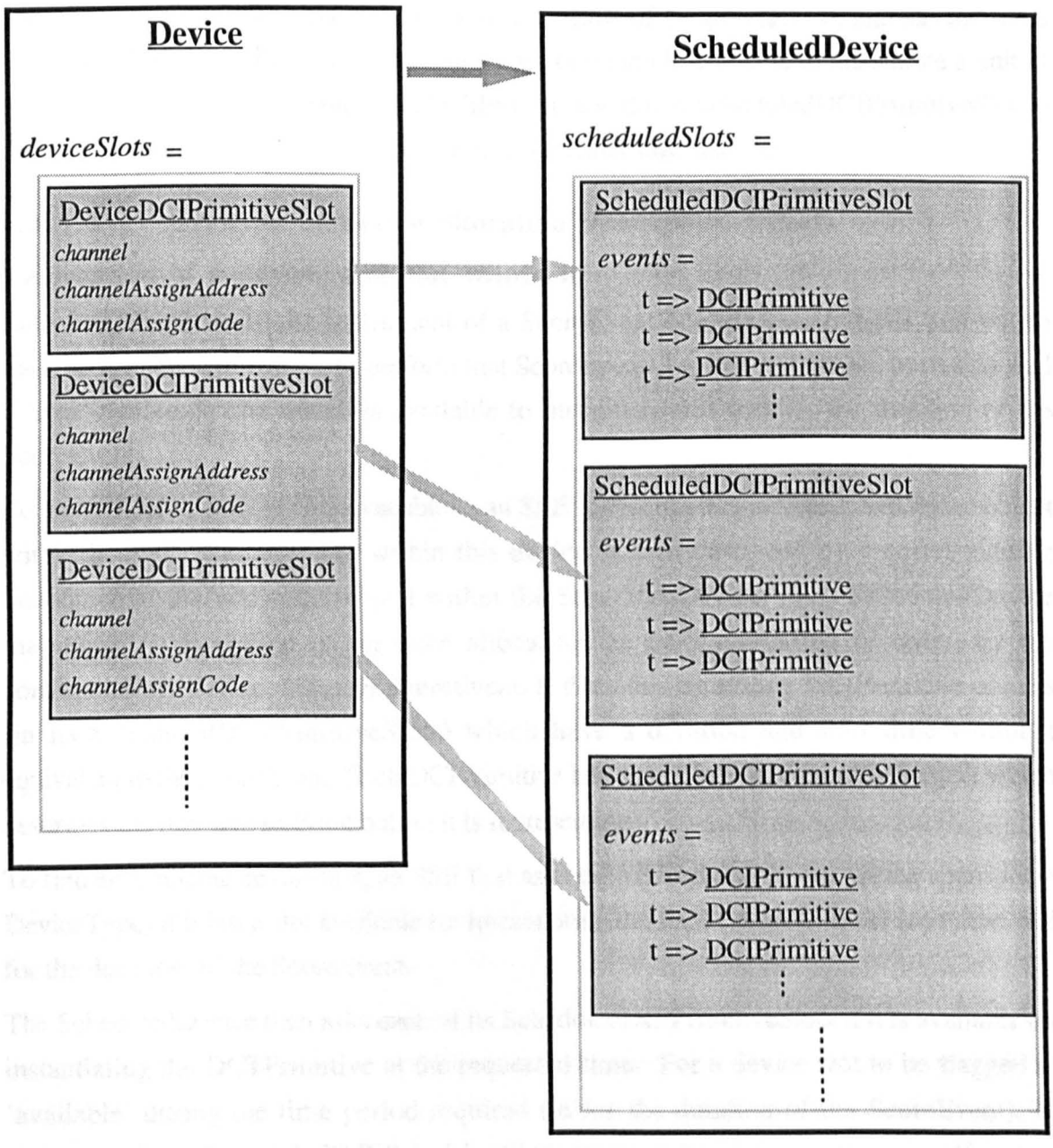


Fig. 85 Creating a ScheduledDevice for an SSE from a 'template' Device

Each ScheduledDCIPrimitiveSlot describes the *allocation* of units - for set times and durations - in the corresponding slot in the device. Each unit in an Instrument is described by a DCTPrimitive object, as described in 9.1.3.4. A corresponding DCIPrimitive[1] object is then created and stored within the slot (ScheduledDCIPrimitiveSlot) which has been allocated to instantiate it.

---

1 An etymological note: DCTPrimitive stands for DeviceConfigurationTemplatePrimitive; DCIPrimitive stands for DeviceConfiguration**Instance**Primitive. When both words are used in proximity, the **I** will be printed in bold to aid readability.

Each DCIPrimitive object is stored with the start time of the ScoreEvent, and has the same duration[1]. Thus when the SSE is looking for a slot which is available to instantiate a unit, it can use the presence or absence of a DCIPrimitive within a ScheduledDCIPrimitiveSlot to decide if a particular slot is available during a particular time interval.

9.3.1.2 Creation of device allocation description objects

• Allocation of a device, and slot within it

Each DCTPrimitive in the Instrument of a ScoreEvent describes a synthesis unit which must be instantiated in order to perform that ScoreEvent. To do this, the SSE must first find a 'free' device, ie one which is available to instantiate this unit for the duration of the ScoreEvent.

To recap 9.3.1.1, each device available to an SSE is described by a ScheduledDevice object within it. Each available *slot* within this device is then described by a corresponding ScheduledDCIPrimitiveSlot object within the ScheduledDevice. Each ScheduledDevice maintains a description of resource allocation (ie its instantiating of units) by the corresponding device, for each ScoreEvent. It does this by storing DCIPrimitive objects (in its ScheduledDCIPrimitiveSlots) which have a duration and start time within it equivalent to the ScoreEvent. Each DCIPrimitive has a template DCTPrimitive object which describes the unit whose instantiation it is representing.

To find an available device slot, an SSE first asks each ScheduledDevice (of the appropriate DeviceType) if it has a slot available for instantiating the DCTPrimitive at the start time, and for the duration, of the ScoreEvent.

The ScheduledDevice then asks each of its ScheduledDCIPrimitiveSlots if it is available for instantiating the DCTPrimitive at the requested time. For a device slot to be flagged as 'available' during the time period required (ie for the duration of the ScoreEvent), its corresponding ScheduledDCIPrimitiveSlot must either *not* contain an allocation (DCIPrimitive) for that time interval, or contain an *existing* allocation (for another ScoreEvent) which can be *shared*. The basis for such sharing of DCIPrimitives is described below.

• Allocation of a unit within a device slot

Once an available slot has been located in the device, the unit can be allocated to it. This allocation is performed in one of two ways:

(1) The more usual way is to create a new DCIPrimitive object which has a duration equal to that of the ScoreEvent which is using it, and which corresponds to the template DCTPrimitive object.

This is shown in the example illustrated in figure 86 below. A new DCIPrimitive has been created to represent the instantiation of a DCTPrimitive for a new ScoreEvent. The

---

1 DCIPrimitives can also be *shared* by ScoreEvents, and will then have a conjoint duration (described in more detail in 9.3.1.2).

ScoreEvent is likely have further DCTPrimitives, each requiring the creation of a corresponding DCIPrimitive.

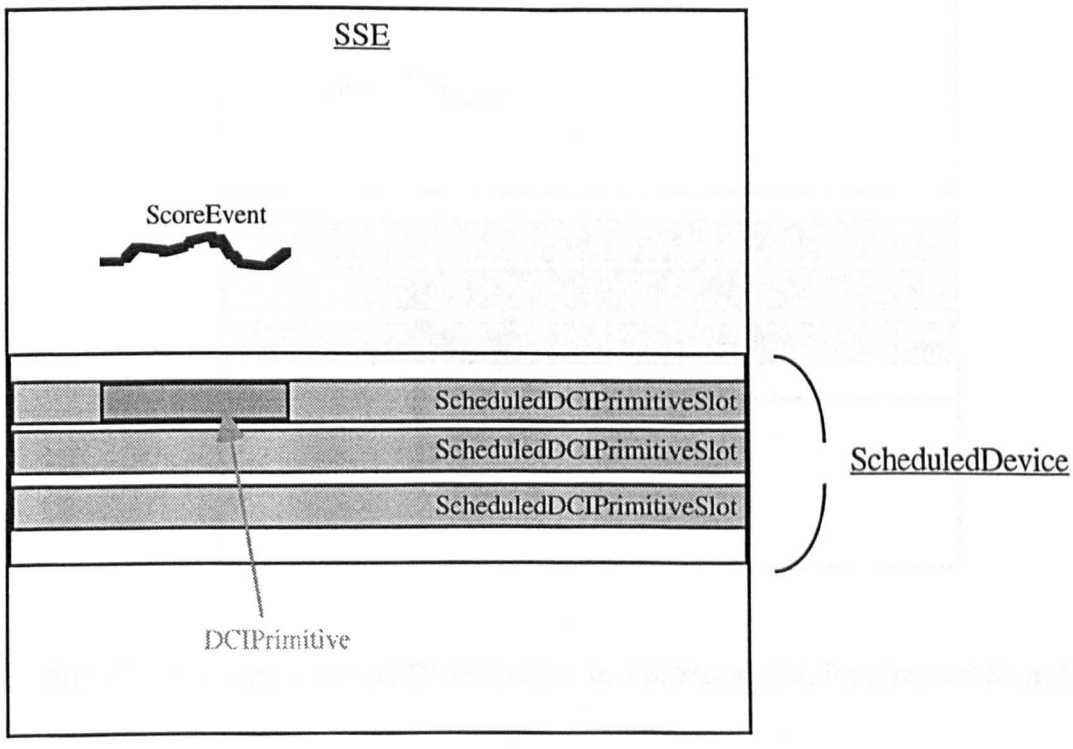


Fig. 86 Creating a DCIPrimitive to describe the instantiation of a unit for a new ScoreEvent

A second ScoreEvent is now added, as illustrated in figure 87 below. An available ScheduledDCIPrimitiveSlot is selected which does *not* contain a DCIPrimitive for the time and duration of the new ScoreEvent, and a new DCIPrimitive is created and installed in it.

Alternatively, an *existing* DCIPrimitive within the ScheduledDCIPrimitiveSlot may be able to be *re-used*. This will be the case if the unit which it represents can support the performance of more than one event at once. A unit with this capability is termed 'polyphonic', and an example is that of a unit in a MIDI controlled device, which can typically support 8, 16 or more events at the same time (see 3.1.1.4). This is subject to the constraint that *differing* input values are not sent from events being performed on the same unit.

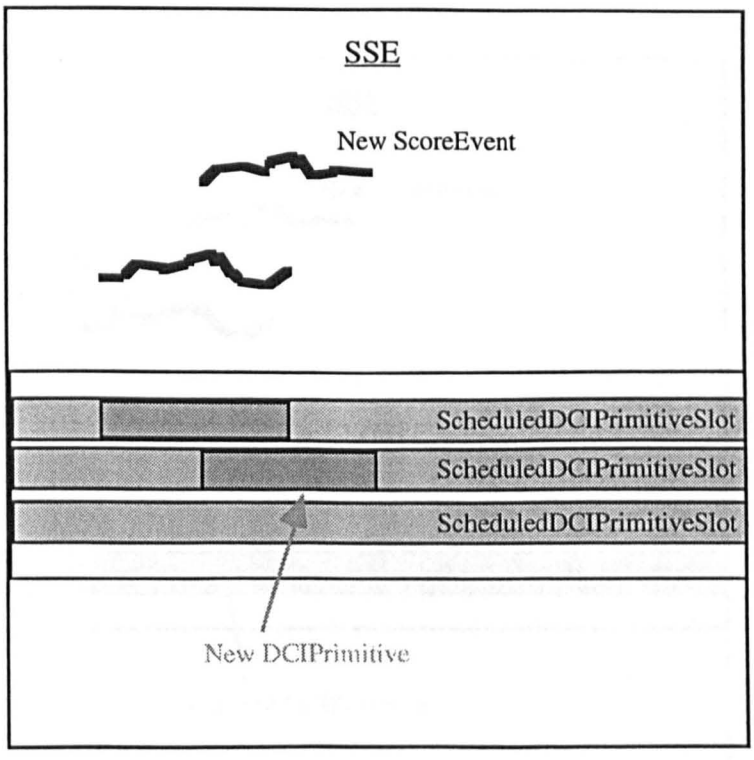


Fig. 87 Creating a second DCIPrimitive in a different slot, for a second ScoreEvent

For example, each DCTPrimitive (of category 'PART') in the above example has a 'volume' input (not shown) which uses the 'MIDI:controller' MessageType (see 8.8.3). This input is *not* 'polyphonic' which means that more than several events (which use these DCTPrimitives) can only run on the *same* device unit *if* each event has the same values for this 'volume' input.

However, as described in 3.1.1.4, some inputs can be characterised as 'polyphonic', such as the 'st. pitch' input (which uses the 'MIDI:note on' MessageType). This means that *different* values for this input may be received by a single unit. Thus, in this case, *several* ScoreEvents can run on the same DCIPrimitive, representing the *same* unit allocation in the device.

Thus, the allocation of a unit within a slot in this case will be described by incorporating the new SE's duration into that of the existing DCIPrimitive. An example of this sharing of unit allocations is shown in figure 88 below, where a new ScoreEvent is being added to an SSE. It has an Instrument (not shown) which incorporates the *same* DCTPrimitive as the existing ScoreEvent, and which is has 'polyphonic' inputs[1]. It can therefore use the *same* device unit, and therefore share the same DCIPrimitive object.

---

1 At least for the DCTPrimitive input parameters which are derived from the 'pitch' event (Psp) parameter.

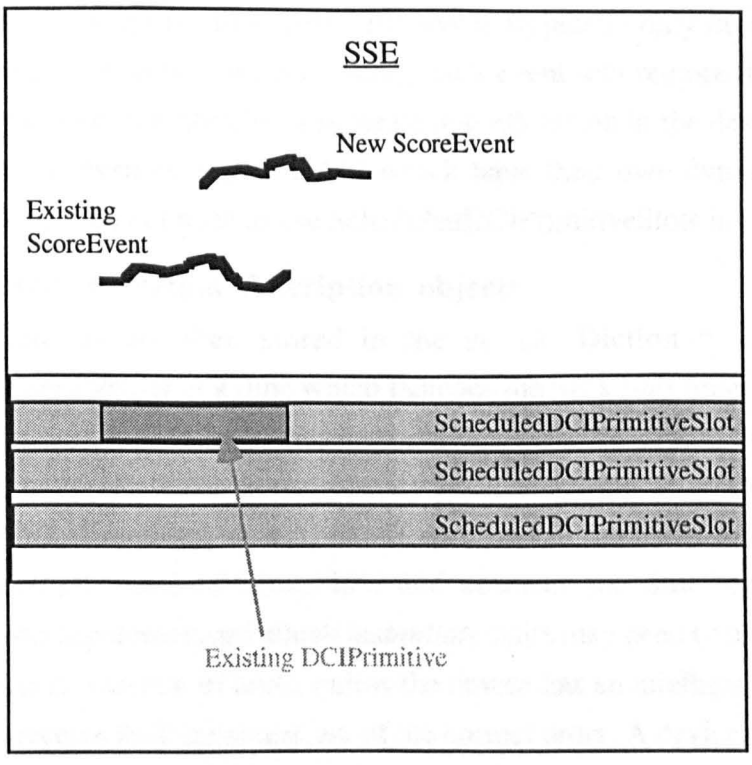


Fig. 88 Adding a new ScoreEvent which can use the same unit as an existing ScoreEvent

Figure 89 below, then shows the resulting allocation of the new ScoreEvent to the same DCIPrimitive, whose duration is then increased to encompass the new ScoreEvent.

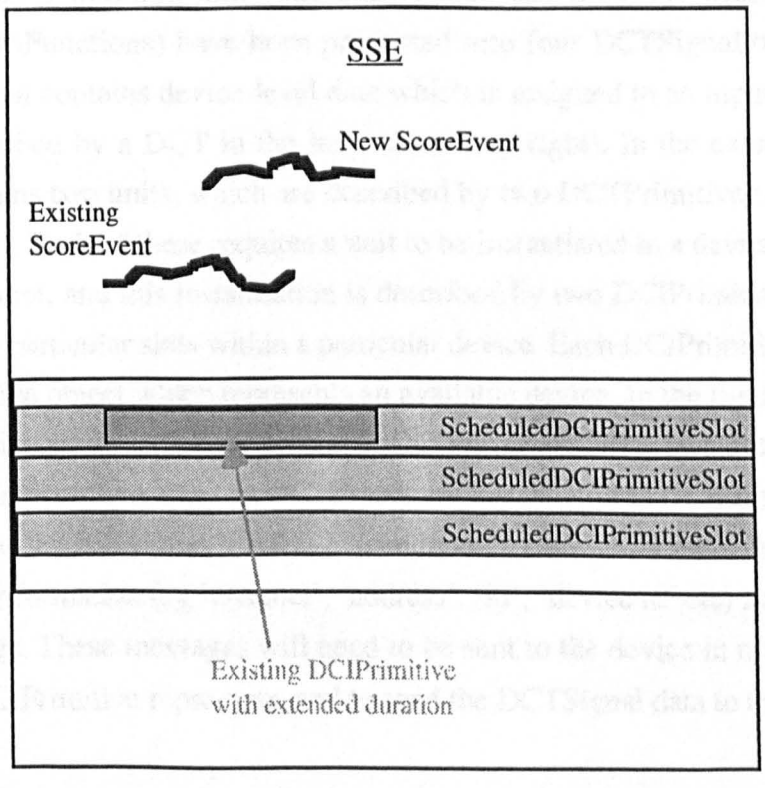


Fig. 89 Utilising an existing DCIPrimitive for a new ScoreEvent which can share the unit which it represents

In practice, in the MIDI devices discussed here, most inputs are *not* 'polyphonic' in this way. Hence, for all except the most primitive scores (typically only involving specifying semitone pitch independently for each event), each event *will* require the allocation of a *separate* DCIPrimitive, representing a separate unit allocation in the device. Note that for control of 'custom' systems (eg MIDAS) which have their own dynamic allocation of resources, E-Scape does not need to use ScheduledDCIPrimitiveSlots in an SSE.

• Storage of unit allocation description objects

These DCIPrimitives are then stored in the *events* Dictionary of the allocated ScheduledDCIPrimitiveSlot at a time which matches the SE's start time. This start time of the DCIPrimitive may need to be given a negative time offset relative to the start time of the ScoreEvent, so as to allow for device latency in instantiating or connecting units. The degree of time offset required by a device is specified in the MessagePrototypes which define the messages used to instantiate and connect the unit represented by the DCIPrimitive. MessagePrototypes which *instantiate* units may need to be sent before those which request the *connection* of units, unless the device has an intelligent message storage system and can receive such messages out of the correct order. A device could conceivably require complex timing relationships between the sending of messages (eg to instantiate then connect units into a structure), but this aspect has not been considered further in the current design.

• Summary of unit allocation by SSEs

The allocation of device resources for a ScoreEvent within an score (SSE) is summarised in figure 90 below. The ScoreEvent is shown on the top left, within an SSE. The ScoreEvent has a duration and start time within the SSE. As described above, its parameters (PspFunctions) have been processed into four DCTSignal objects (top left). Each DCTSignal contains device-level data which is assigned to an input of the synthesis structure described by a DCT in the Instrument (top right). In the example, this device structure contains two units, which are described by two DCTPrimitive objects within the DCT (far right). Each of these requires a unit to be instantiated in a device for the duration of the ScoreEvent, and this instantiation is described by two DCIPrimitive objects, which are assigned to particular slots within a particular device. Each DCIPrimitive exists within a ScheduledDevice object which represents an available device. In the figure below, a single ScheduledDevice exists with nine ScheduledDCIPrimitiveSlots which describe slots within the device. This ScoreEvent has its DCTPrimitives instantiated as DCIPrimitives within two of these, named 'PART-1' and 'PART-2' (centre left). Each DCIPrimitive can later be used to supply the information (eg 'channel', 'address', 'id', 'device id' etc) needed to construct message strings. These messages will need to be sent to the device in order both to create the unit the DCIPrimitive represents, and to send the DCTSignal data to the unit's inputs.

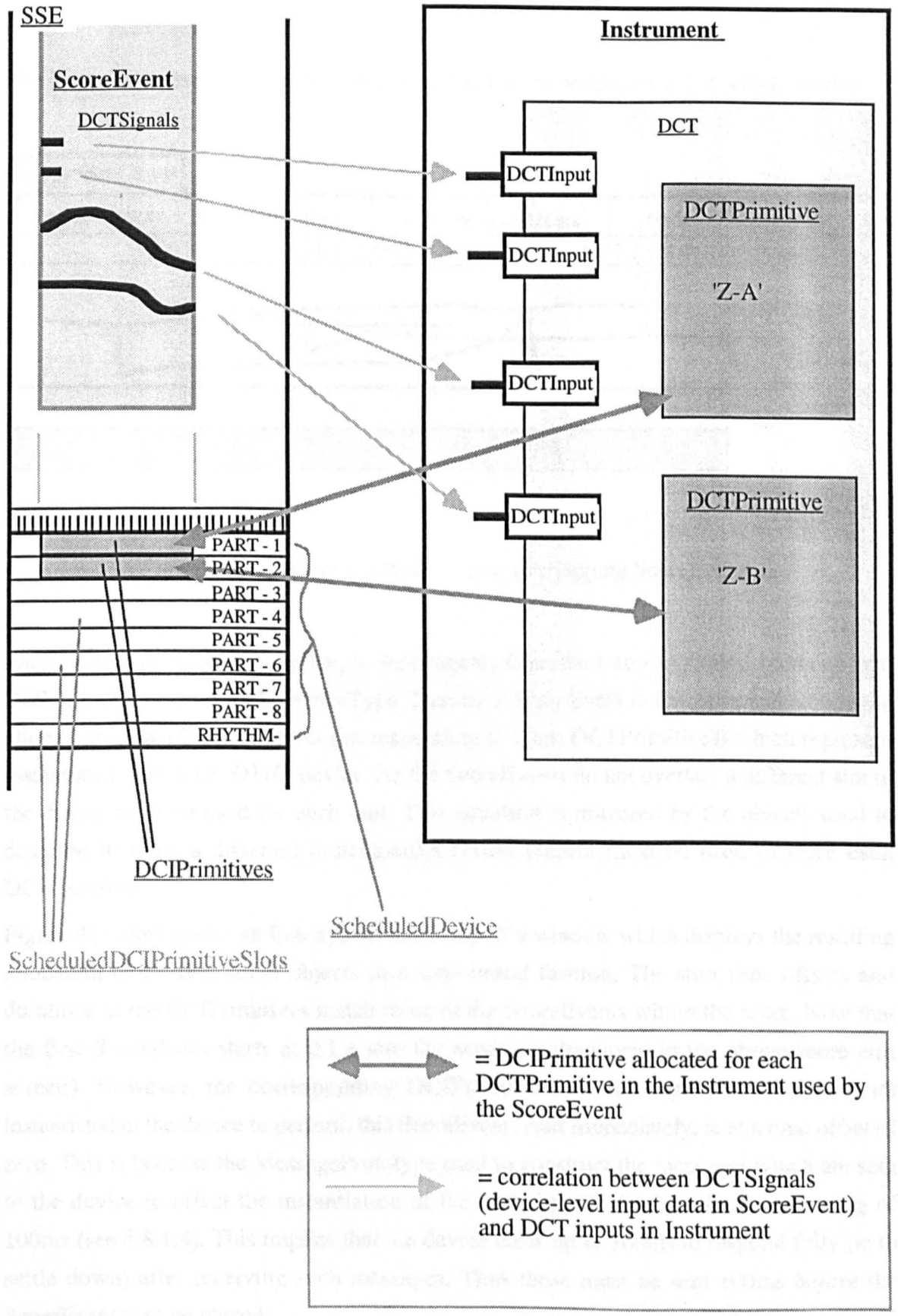


Fig. 90 Allocation of DCIPrimitives for a ScoreEvent -
one corresponding to each DCTPrimitive in the Instrument of the ScoreEvent

9.3.1.3 Allocation of device resources - example 1

The example score shown below in figure 91 has four ScoreEvents, all of which overlap.

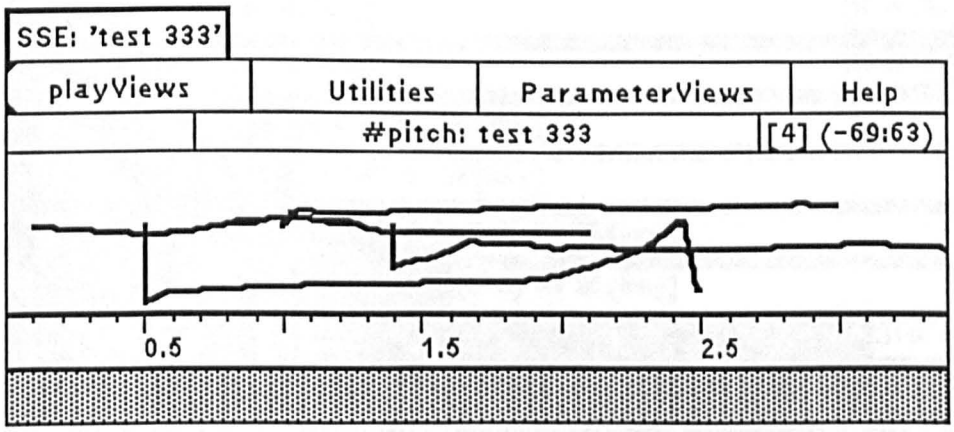


Fig. 91 Score example 1 - four overlapping ScoreEvents

Each ScoreEvent uses the example Instrument, described above, which contains two DCTPrimitives on a 'D110' DeviceType. Thus each ScoreEvent in the score will require the allocation of two DCIPrimitives (corresponding to these DCTPrimitives) which represent instantiated units on a 'D110' device. As the ScoreEvents do not overlap, a different slot in the device must be used for each unit. This situation is mirrored by the objects used to describe it: thus, a different ScheduledDCIPrimitiveSlot must be used to store each DCIPrimitive.

Figure 92 below shows an E-Scape screen dump of a window which displays the resulting allocation of DCIPrimitive objects in a time-based fashion. The start time offsets and durations of the DCIPrimitives match those of the ScoreEvents within the score. Note that the first ScoreEvent starts at 0.1 s into the score (as displayed in the above score edit screen). However, the corresponding DCIPrimitives (which represent the two units instantiated in the device to perform this ScoreEvent) start immediately, ie at a time offset of zero. This is because the MessagePrototype used to construct the messages which are sent to the device to effect the instantiation of the units have a *minTimeBetween* setting of 100ms (see 8.8.1.4). This implies that the device takes up to 100ms to respond fully (ie to settle down) after receiving such messages. Thus these must be sent 100ms *before* the ScoreEvent is to be played.

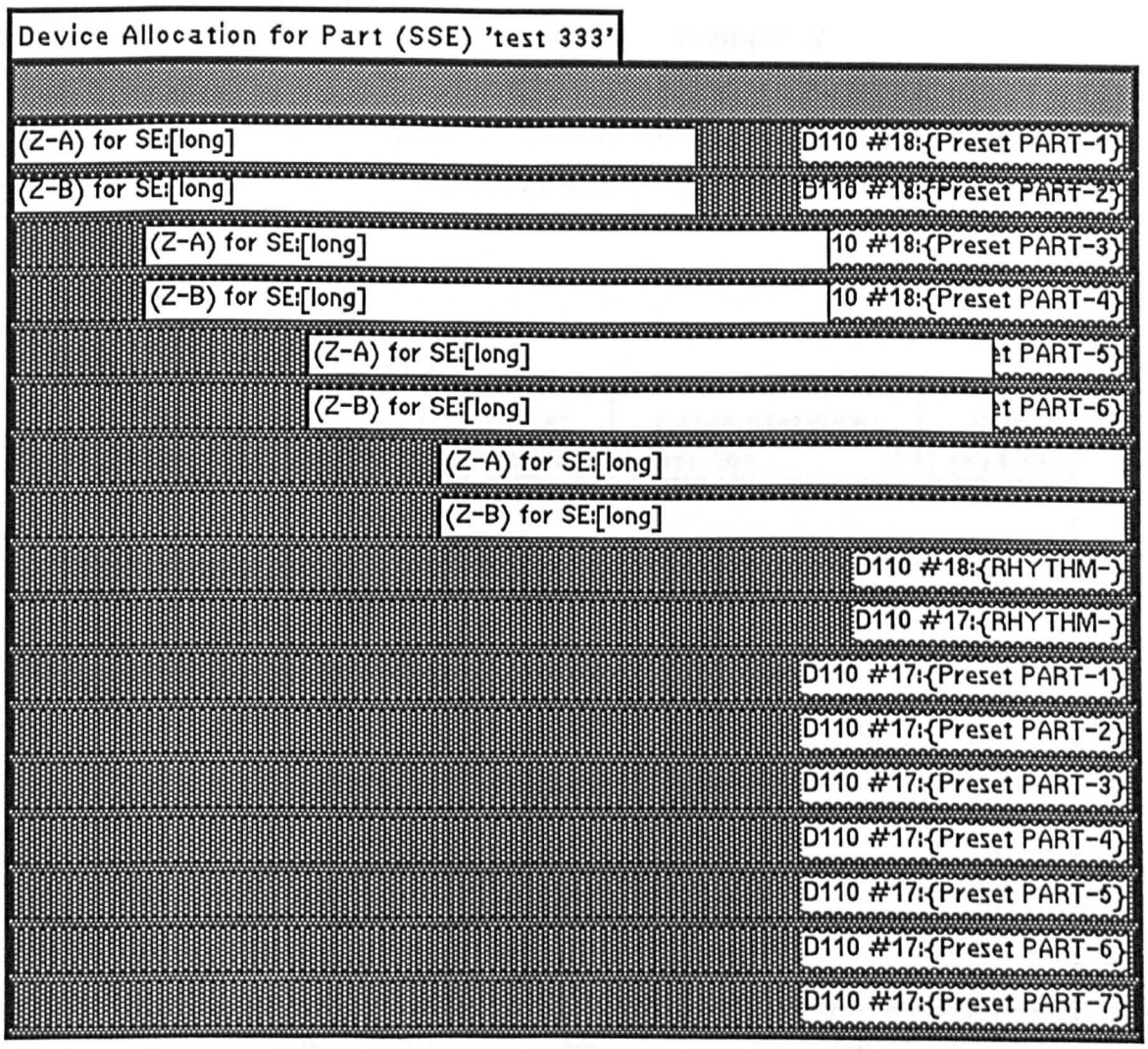


Fig. 92 Score example 1 device resource allocation display - using DCIPrimitives in *different* slots for the four overlapping ScoreEvents

9.3.1.4 Allocation of device resources - example 2

The example score shown below in figure 93 also has four ScoreEvents using the same Instrument a before. All overlap, *except* the third and fourth ScoreEvents, which can thus use the *same* ScheduledDCIPrimitiveSlot for their DCIPrimitives

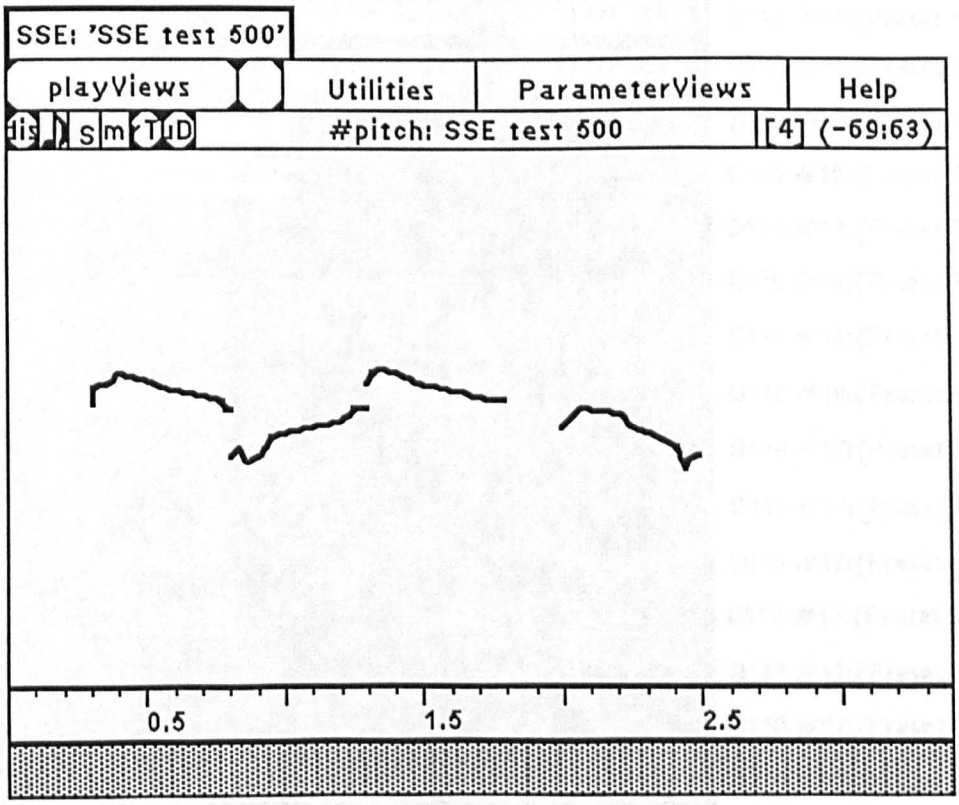


Fig. 93 Score example 2 - Four ScoreEvents
the third and fourth do *not* overlap

The resulting unit allocation display thus appears as in figure 94 below, with DCIPrimitives for the fourth ScoreEvent using the same slots as those for the third.

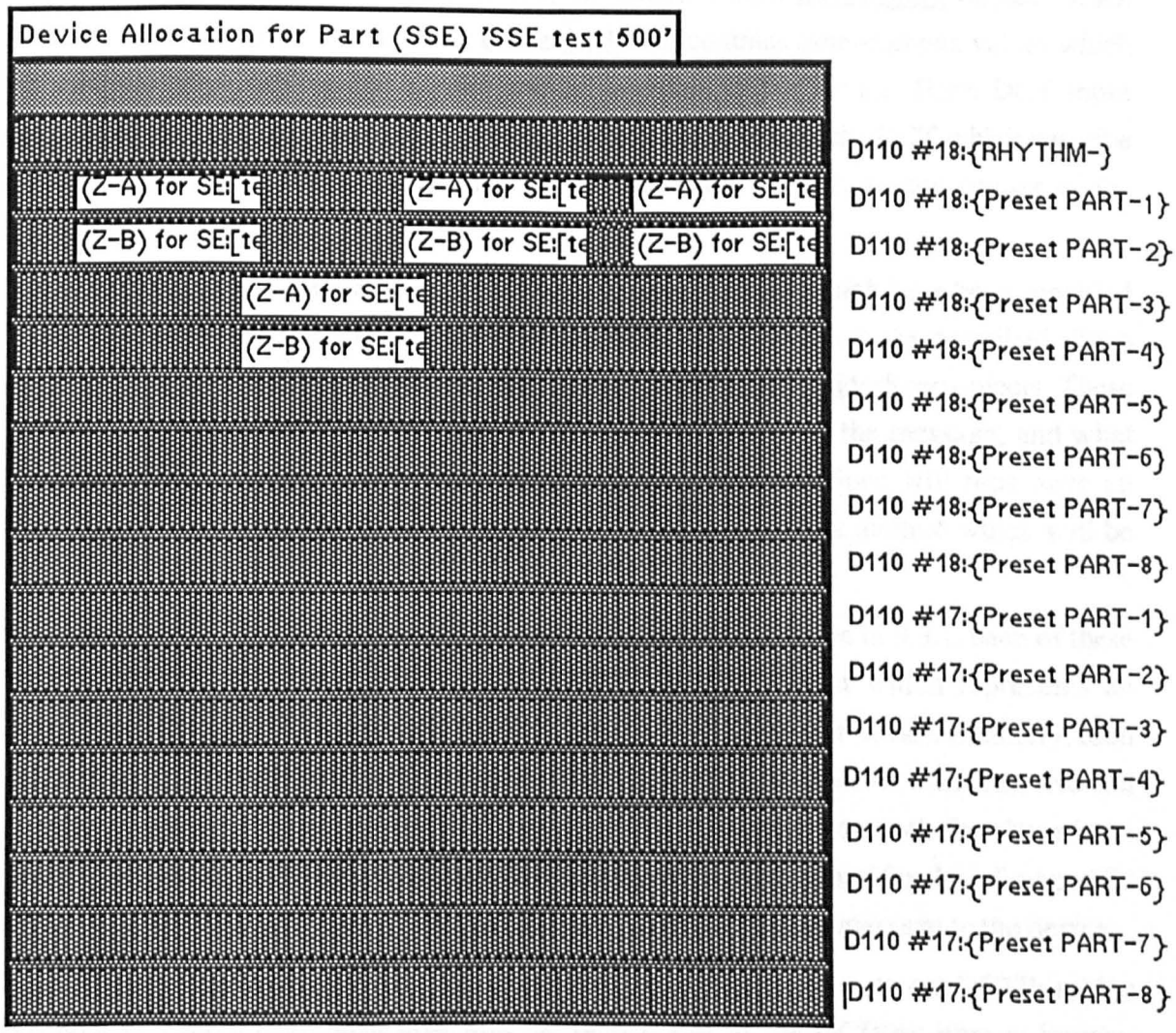


Fig. 94 Example 2 allocation display - using DCIPrimitives in the same slot for the two non-overlapping ScoreEvents

9.3.2 ScoreEvent processing stage 2 - creating device messages

As a result of the level 1 processing of score parameters (described in 9.2.2), device-level values are now present in the ScoreEvent, contained within DCTSignal objects. Each DCTSignal is associated with an input of the DCT, and contains time-stamped values which are assigned to be sent to the corresponding structure in the device. Each DCT input connects to one or more device-level DCTPrimitive objects within the DCT which describe synthesis units within the device. The connections are actually to the *inputs* of the DCTPrimitives.

Each DCTPrimitive input has access[1] to the MessagePrototypes which have been specified by the user when constructing a module type specification (see 8.6.2). As described above in 8.8.2.1, each MessagePrototype contains a number of FieldProviderSpec objects. These specify which objects are to provide the values for each field of the message, and what information is to be requested from them. Each FieldProviderSpec will thus have an assigned object which can be interrogated, as well as a Smalltalk method which will be sent to it.

As a result of the device resource allocation activity described above in 9.3.1, each of these DCTPrimitives now has a corresponding DCIPrimitive object which represents an instantiation of that DCTPrimitive in an actual location in a particular device. Similarly, each input of the DCTPrimitive now has an analogous DCIPrimitive input. A DCIPrimitive and its inputs inherit information from the 'parent' DCTPrimitive from which they have been created. However, a DCIPrimitive now has a *location* within a specific slot of a specific device, and can thus now supply all the information needed to send message to the device.

Thus device addresses which were expressed merely as *offsets* in the parent DCTPrimitive can now be finally calculated now that the device slot of the DCIPrimitive is known. Similarly, the DCIPrimitive now knows which 'channel' it can use. Thus, any MessagePrototypes which send the Smalltalk message 'channel' for example will now receive an answer from the DCIPrimitive.

9.3.2.1 Accessing a DCIPrimitive's MessagePrototypes

To perform the ScoreEvent, each DCIPrimitive now needs to do two things:-

- create one or more message which can be sent to the device to instantiate the unit which the DCIPrimitive represents;
- create messages to convey values to an input of a unit in a device which corresponds to an input of the DCIPrimitive.

This is done by the DCIPrimitive and its inputs using their MessagePrototype objects. Each of these then refers to the field list of its parent MessageType. Each field is either a fixed value, or references a FieldProviderSpec object. If the latter, it interrogates the object

---

1 A DCTPrimitive input has access to its parent DCTPrimitive input. This in turn can access the SMT or PrimSMT input from which it was created (see 8.6.2 and 8.6.4).

specified by the FieldProviderSpec and supplies a value. Each field of the MessageType is thus supplied by a value which is loaded into a DeviceEvent - a simple object which contains data values for each field in the MessageType. This low level data is subsequently sent out to the device.

9.3.2.2 Creation of DeviceEvents

As stated above, a DeviceEvent object contains the data which will be sent out to a device as a message packet in the appropriate format. This data will perform various functions such as instantiating or sending data to entities in a device.

Each allocated DCIPrimitive object represents and describes an identified unit within the device. A DCIPrimitive and its inputs each have MessagePrototype objects which build up the DeviceEvents by retrieving the requisite data for each field.

It should be noted that E-Scape development at present is concerned with level I control of devices, as many devices which implement this level of communication (at least partly) are available. At the level, DeviceEvents are sent to a device for two main purposes: to request a device to instantiate a unit, and to send data to its inputs.

9.3.2.3 Creation of DeviceEvents to send inputs values to device units

Low-level data in a DCTSignal (derived from E-Scape score parameter data) must be sent to the inputs of the units in the device which correspond to DCIPrimitive inputs.

Each DCTInputSignal in the ScoreEvent has a set of time-stamped values, which are assigned to an input of a DCT in the Instrument. This DCT input has a set of one or more connected 'destinations' within the DCT. These destinations are DCTPrimitives and *their* inputs which describe units to be instantiated in the device.

Each DCTPrimitive will now have created a corresponding DCIPrimitive within the SSE. This represents an actual *allocation* of a unit in a device, and thus has an actual id, address, channel number etc. pertaining to that allocated unit.

Thus, each DCTInputSignal can locate its associated DCT input within the DCT. It can then find the DCTPrimitives and the inputs to which the DCT input is connected. Finally it finds the DCIPrimitive inputs which correspond to these DCTPrimitive inputs.

The DCTInputSignal now sends each of its time-stamped values in turn to each of these DCIPrimitive inputs. Each DCIPrimitive input then creates DeviceEvent objects using these values, as well as other data derived from other objects (as specified in the MessagePrototype). Each DeviceEvent now contains a series of actual numbers which will be sent to the device.

A DeviceEvent is created by the DCIPrimitive input, using its MessagePrototype as follows:-

The MessageType has field names or fixed values defined in its *fieldList* set. The field names may either be designated as "supplied" in which case values for them will be supplied, or will be "derived" from such field values. Each "supplied" field will be provided with a data value by an object associated with the DCIPrimitive input. This object

is specified by the *user* object of the MessagePrototype, and can be interrogated to return a value. This *user* object is typically the input of the DCIPrimitive itself, which exists as a DCIPrimitiveInput object.

The MessagePrototype also has a number of FieldProviderSpec objects, each of which has a *valueSelector* string (eg. 'parentsChannel'). This invokes a Smalltalk message to be sent to the *user* object to interrogate it for a value. These message strings have been entered by a user when constructing the MessagePrototype - selected from one of a small group of E-Scape *standard* message strings.

A *user* DCIPrimitiveInput object can be asked for such things as its creation code(s), connected destinations, offset addresses, ids, slot id, etc. It can also be asked for the value of the DCTSignal - ie the device level input parameter value of the score event - at a particular time. The DCTSignal contains time and value breakpoints, with each different value resulting in a DeviceEvent being created.

A FieldProviderSpec may also have a *secondaryCollectionSelector* - a Smalltalk message which returns one or more objects which have some association with the *user* object. Each of *these* objects will then be sent the *valueSelector* Smalltalk message, each returning a value and creating a DeviceEvent. This was described in detail in 8.8.2.1.

9.3.2.4 Example of stage 2 processing

This processing is illustrated in the example depicted in figure 95 below. A single ScoreEvent is shown on the left within a score (SSE) object, which is using the Instrument on the right. Two DCTSignals are shown within the ScoreEvent, which are associated with the 'BENDER RANGE' and 'pitch bend' inputs respectively of a DCT within the Instrument. Each DCT contains data values which must be sent to these inputs of the structure in the device represented by this DCT. The DCT consists of two DCTPrimitives, each of which represents a unit on the device. Each DCT input shown connects to *two* inputs of DCTPrimitives within it. Thus data assigned to one of these DCT inputs must be sent to the *two* units on the device represented by these DCTPrimitives. Thus each DCTSignal value must be converted into *two* messages addressed to the two units within the device. This notional processing path is shown for the first 'BENDER RANGE' DCTSignal, and a similar pathway exists (but not illustrated) for the other 'pitch bend' signal. The other DCT inputs in this Instrument are not shown. This path indicates the conceptual processing route from DCTSignal to DeviceEvent, but the objects which are really involved are the DCIPrimitives (illustrated on the left, below the ScoreEvent). These DCIPrimitives correspond to, and correlate with, the two DCTPrimitives (right) but, as described previously, are assigned to a specific slot within a selected Device. It is their inputs which have the MessagePrototype objects described above, and which provide the values for the DeviceEvents.

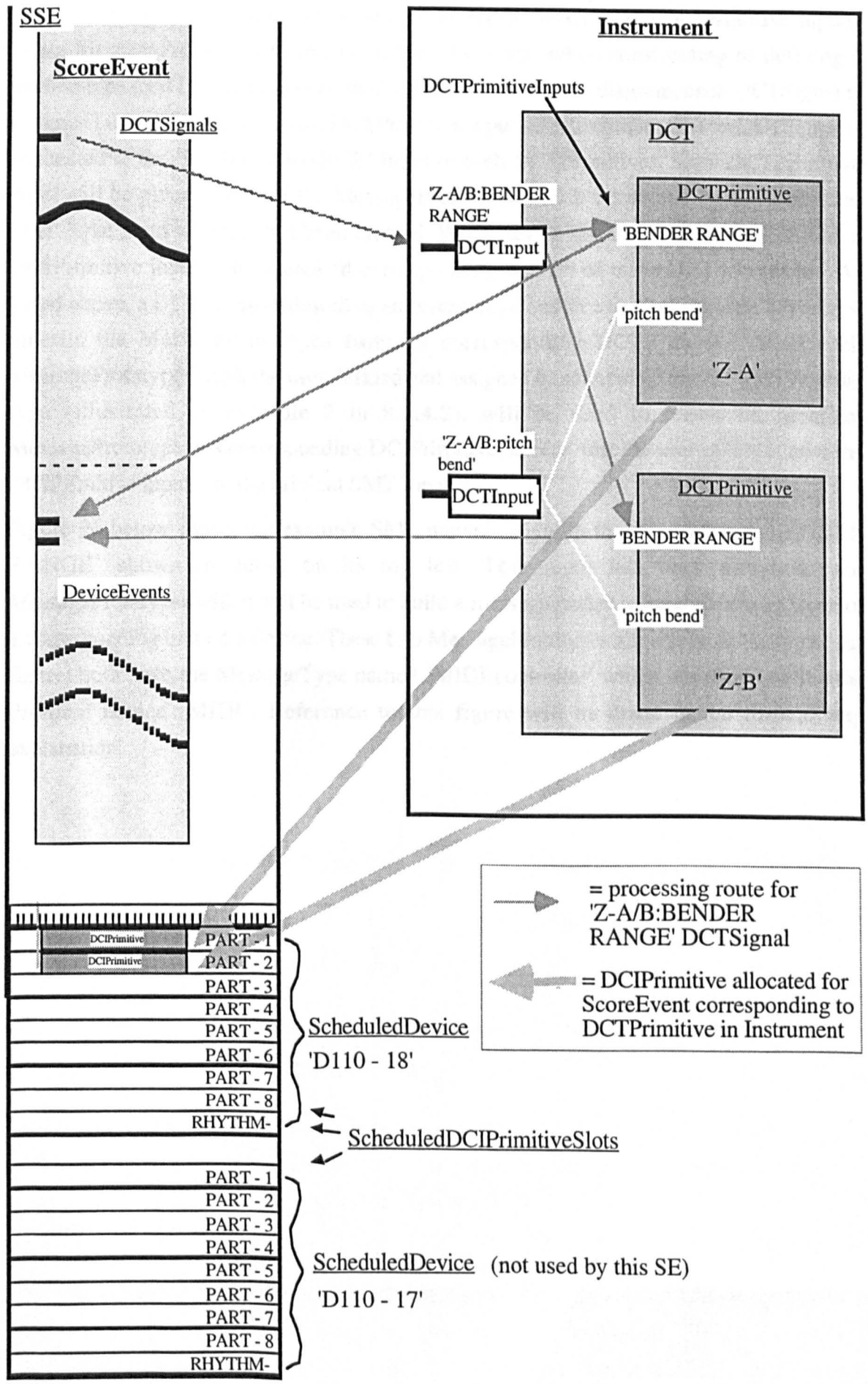


Fig. 95 Score processing stage 2 - converting DCTInputSignal data into DeviceEvents to convey unit input values to a device.

These values are provided by the MessagePrototypes owned by the DCIPrimitive inputs. These MessagePrototypes were first defined by a user when constructing or defining a module type (SMT) object, as described in 8.6.4. In the above diagram, each DCTSignal is assigned (via a DCT input) to two DCTPrimitive inputs - for example, the top DCTSignal is connected to the 'BENDER RANGE' input of both DCTPrimitives. Each DCTPrimitive input will be given a copy of the MessagePrototypes which are owned by the 'template' SMT input from which is has been created. When device resources are then allocated, a DCIPrimitive input will be created corresponding to each of these DCTPrimitives. As stated above, a DCIPrimitive describes an *instantiated* unit in a device. Each DCIPrimitive inherits the MessagePrototypes from its corresponding DCTPrimitive. Thus each MessagePrototype which the user defined and assigned to an input of the 'Z' SMT module type (illustrated as example 2 in 8.6.4.2), will be used to create an identical MessagePrototype in a corresponding DCIPrimitive, *except* that the *user* object is now the DCIPrimitive input, not the original SMT input.

Figure 96 below shows the example SMT named 'Z', with the input named 'BENDER RANGE' shown in detail on its top left. This input has been assigned *two* MessagePrototypes which will be used to build a message packet to send data to an input of a corresponding unit on a device. These two MessagePrototypes are shown at the top of the figure: both have the MessageType named 'MIDI:controller' which is defined within the Protocol named 'MIDI'. Reference to this figure will be made in the forthcoming explanation.

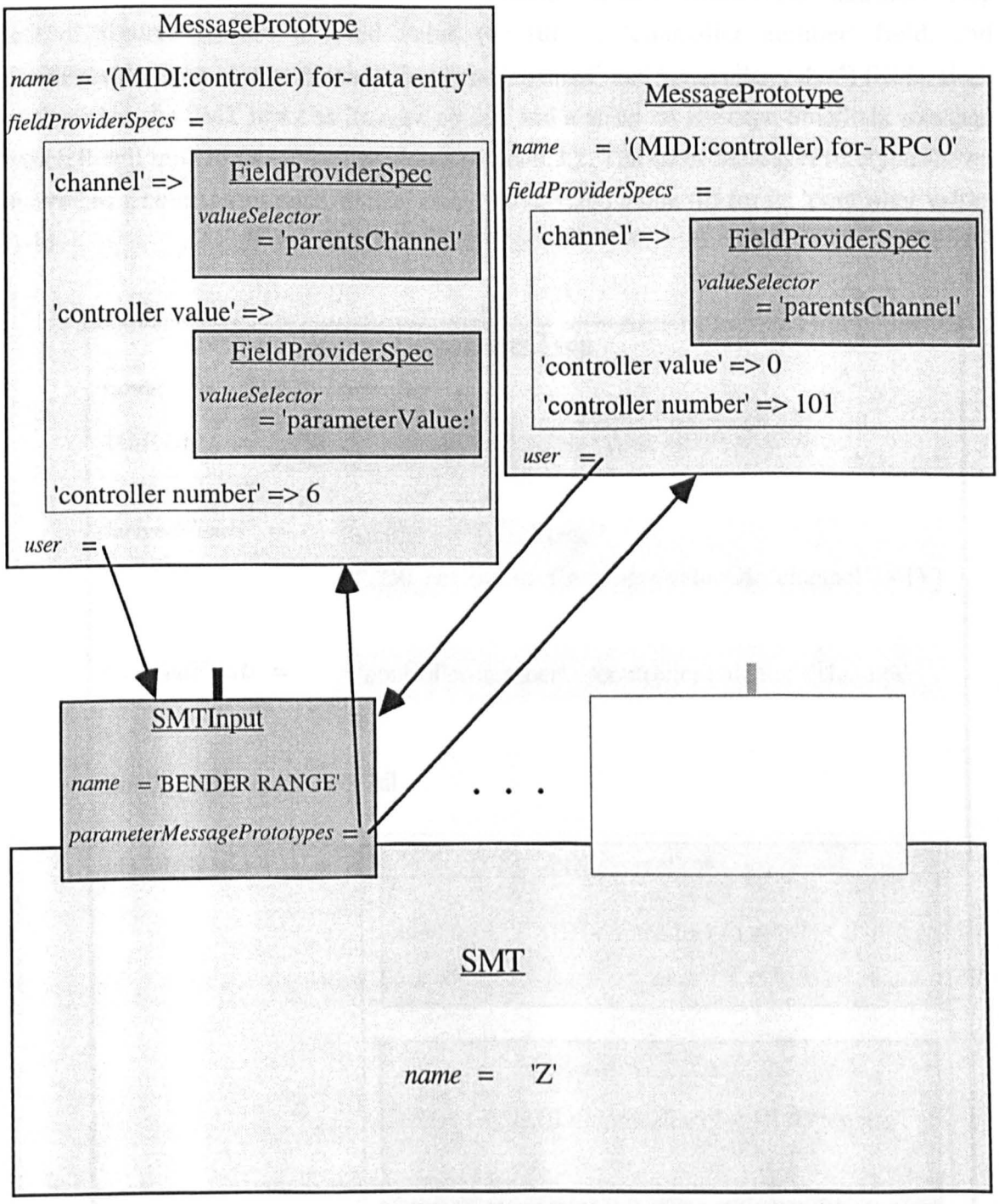


Fig. 96 The two MessagePrototypes owned by the 'BENDER RANGE' input of the example SMT

This 'MIDI:controller' MessageType (described in 8.8.3) has a specification entered by a user, as shown in figure 97 below. It has three names in its *suppliedFields*: ('controller number', 'controller value' and 'channel'). Each MessagePrototype of this type thus has three identically named entries in its *fieldProviderSpecs*, as shown in the two MessagePrototype illustrated above. Each entry will either be a fixed value, or FieldProviderSpec object - either will provide a value for the named message field.

In this example, the first MessagePrototype named '(MIDI:controller) for- data entry' (top left of figure 96) has a fixed value (6) for its 'controller number' field, and FieldProviderSpec objects for the other two ('channel' and 'controller value') fields. Each of these has the SMT input as its *user* object, and a standard E-Scape Smalltalk message which it will send to this object, as described in 8.8.2. The other MessagePrototype shown above has a similar structure, except that it has a fixed value (0) for its 'controller value' field.

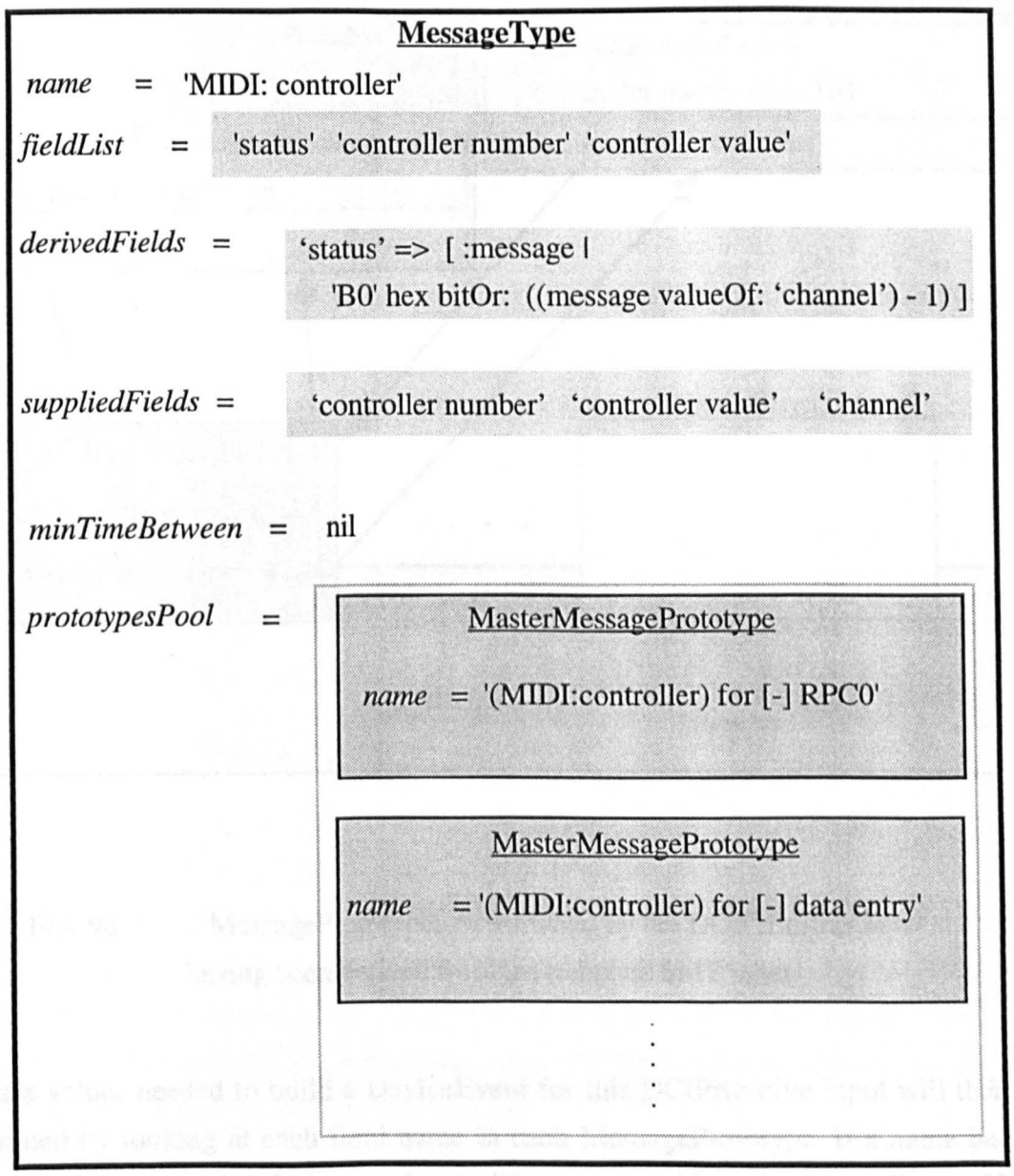


Fig. 97 The MessageType named 'MIDI:controller'

As described above, these MessagePrototypes are then 'inherited' by a DCIPrimitive input which are ultimately derived (via a DCTPrimitive input - see 9.1.4.4) from this SMT input. Figure 98 below shows these derived MessagePrototypes, now owned by a DCIPrimitive input.

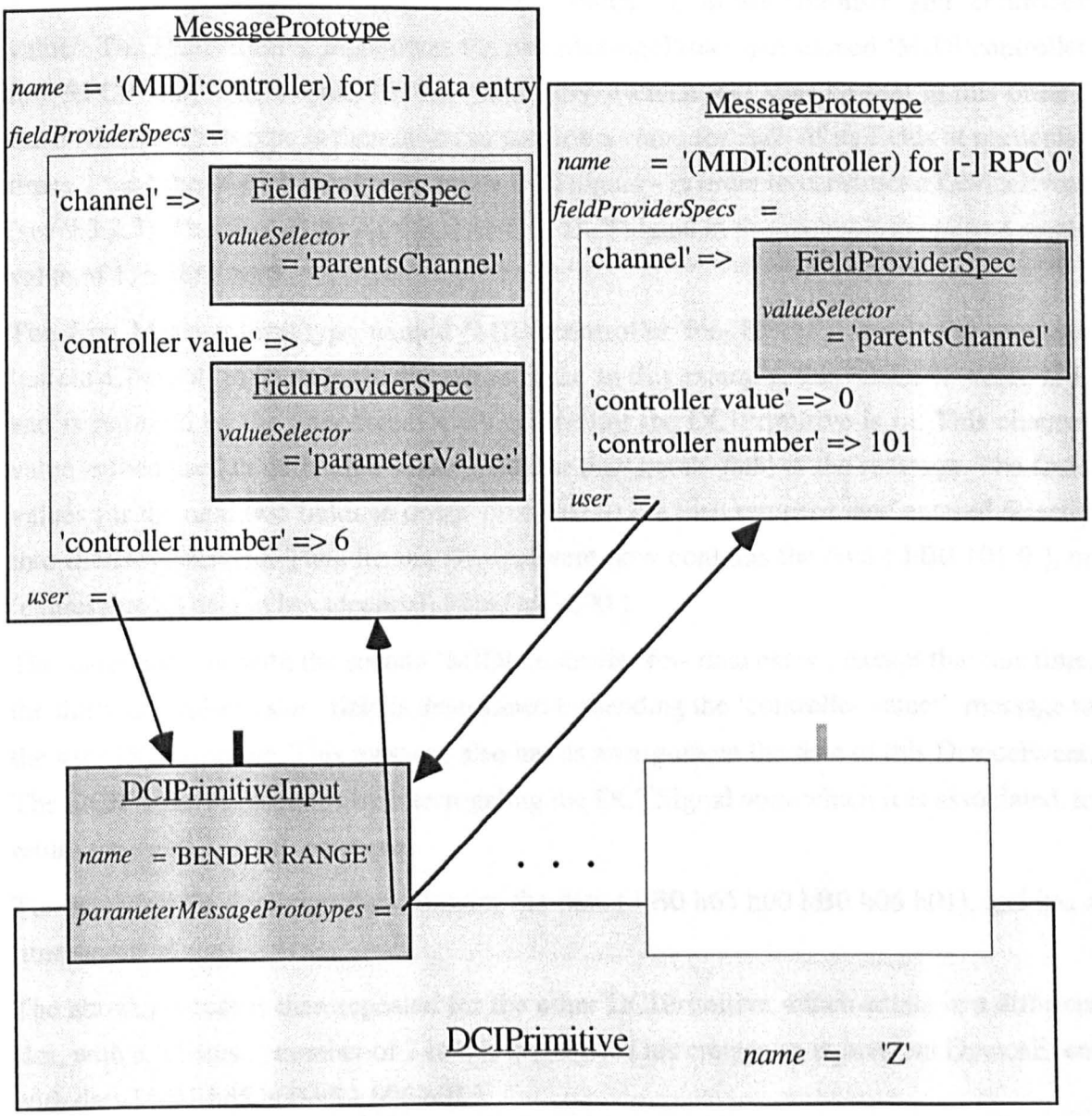


Fig. 98 MessagePrototypes now owned by the DCIPrimitive input - having been derived from the template SMT input

The data values needed to build a DeviceEvent for this DCIPrimitive input will then be determined by looking at each field name in each MessagePrototype. If a name has an associated fixed value (eg 0 above), this is directly entered into the DeviceEvent. If the name has an associated FieldProviderSpec, then its *valueSelector* Smalltalk message is sent to its *user* DCIPrimitive input object, to provide the value for this field.

For example, the value for the 'channel' variable in the example will be returned by sending the Smalltalk message 'parentsChannel' to the DCIPrimitive input. This 'channel' variable is not actually present as a field of the MessageType (ie not present in its *suppliedFields* list), *but* its value is used to *derive* a value for the 'status' field of the MessageType (as described in 8.8.1.3).

The MessageType has field names (in order) 'status', 'controller number' and 'controller value'. The DCIPrimitive input owns the two MessagePrototypes named 'MIDI:controller for- RPC0' and 'MIDI:controller for- data entry' (which will later be sent in this order). Each MessagePrototype is then asked to provide a value for each of its fields at particular times - the time of each breakpoint in the DCTSignal - in order to construct a DeviceEvent (see 9.3.2.3). The 'Z-A/B:BENDER RANGE' DCTSignal in this example contains a single value of 1, at time zero.

The first MessagePrototype named 'MIDI:controller for- RPC0' [1] sends the message 'parentsChannel' to its *user* DCIPrimitive input. In this example, this 'channel' value is 1, and is returned by the ScheduledDCIPrimitiveSlot the DCIPrimitive is in. This channel value is then used to derive the value (hB0) for the 'status' field of the message. The fixed values for the next two fields in order (101 and 0) are then returned, and entered directly into the DeviceEvent. Thus far the DeviceEvent now contains the data [hB0 101 0], or (expressing all data in hexadecimal) [hB0 h65 h00].

The same happens with the second 'MIDI:controller for- data entry', except that this time, the third 'controller value' field is determined by sending the 'controller value:' message to the *user* DCIPrimitive. This message also has as an argument the *time* of this DeviceEvent. The DCIPrimitive responds by interrogating the DCTSignal with which it is associated, to return the value 1 in this example.

The resulting DeviceEvent thus contains the data [hB0 h65 h00 hB0 h06 h01], and has a time stamp of zero.

The above process is then repeated for the other DCIPrimitive which exists in a different slot, with a 'channel' number of 2 in this example. This creates an equivalent DeviceEvent with data [hB1 h65 h00 hB1 h06 h01].

The other DCTSignal shown named 'Z-A/B:pitch bend' has a series of breakpoints, and its associated DCTPrimitive inputs have a MessagePrototype named 'MIDI:7 bit pitch bend'. Each DCTSignal value is then processed in the same way as above, to produce a DeviceEvent for each DCIPrimitive. These resulting DeviceEvents will contain data such as [hE0 h00 h40], where the last field is derived from the DCTSignal breakpoint value. Thus the ScoreEvent will then contain a set of time-stamped DeviceEvents in its *events* instance variable, with the times (shown on the left) in ms:-

0 =>DeviceEvent [hB0 h65 h00 hB0 h06 h01]
0 =>DeviceEvent [hB1 h65 h00 hB1 h06 h01]
....
0 =>DeviceEvent [hE0 h00 h40]

---

[1] The E-Scape system names of MessageProtypes, as shown thus far, have brackets round the MessageType name which forms the root of the MessageProtype name, eg '(MIDI:controller) for- data entry' has the 'MIDI:controller' parent MessageType. These brackets will be omitted for the sake of readability in this section.

0 =>DeviceEvent [hE1 h00 h40]

9 =>DeviceEvent [hE0 h00 h40]

9 =>DeviceEvent [hE1 h00 h40]

....

....

796 =>DeviceEvent [hE0 h00 h05]

796 =>DeviceEvent [hE1 h00 h05]

If the ScoreEvent has a duration of 1 second, then the input data sent by the last DeviceEvent (at 796ms) will pertain for the rest of the event.

9.3.2.5 Creation of DeviceEvents to *instantiate* units in a device

Each DCIPrimitive corresponds to an allocation requirement for a unit in a device, in order to perform a ScoreEvent. The DCIPrimitive thus has a start time and duration within an SSE which matches this ScoreEvent, and a specification (DCTPrimitive object) of the unit to be instantiated in the device. To perform this ScoreEvent, one or more DeviceEvents must be sent to instantiate the unit in the device.

This DeviceEvent is created by a MessagePrototype (or possibly several) which has been assigned for the creation of the units described by the DCIPrimitive. This MessagePrototype was specified by the user when designing the category (DTSMTCategory) of module types of which this DCIPrimitive is an example. Such MessagePrototypes are stored within the *creationMessagePrototypes* instance variable of the DTSMTCategory (as described in 8.5.3.1).

These MessagePrototypes will now interrogate the appropriate objects to provide the value for each message field in the DeviceEvent, in the same way as for the unit input values described in the previous section.

9.3.2.6 Storage of DeviceEvents

Each DeviceEvent and its time is then loaded into the ScoreEvent's *events* collection. Thus each ScoreEvent holds onto the actual data which is to be transmitted for its performance, enabling more efficient performance, with little subsequent processing being carried out as the score is playing. As DeviceEvents are created only after device resources have been found, then the score which has been successfully processed to stage 2, is then *guaranteed* to be played as it appears. If all resources are not available to achieve this, future E-Scape development will allow the user can select various options: either aborting, manually selecting between ScoreEvents which are competing for a device resource, or specifying parametric conditions to perform this selection on all events automatically. These conditions may be different for different sections or levels of the score structure. The score display appearance can be modified to show which events or parts of events will not actually play.

9.4. Performing events on synthesis devices

As described in 9.3, ScoreEvents are contained within a higher-level SSE (SuperScoreEvent) object, with each SSE then possessing a time-stamped set of ScoreEvents. The SSE will then also possess a number of ScheduledDevice objects, which each define *allocations* within a device of the synthesis structures defined by the Instruments of the ScoreEvents.

The SSE will also contain time-stamped DeviceEvents. These contain data which will be sent to the device to instantiate synthesis units, and convey data values to their inputs. The data in a DeviceEvent has been derived from parameter data in the ScoreEvent, and information in the modules specified within its Instrument, taking into account the particular device resources which the SSE has allocated to the ScoreEvent, as described in 9.3.

To play an SSE now simply requires the data in its DeviceEvents to be sent at the appropriate time-offset (defined by the time stamp of the DeviceEvent within the SSE). This data thus controls devices in real-time using 'level I' communication, as defined in 5.1. Alternatively, the DeviceEvent data can be sent, encapsulated with its time, to effect time stamped control within level I. MIDI-controlled commercial devices do not implement such time-stamped communication (having only a *partial* implementation of the level 1 proposed standard - see 11.1.5.1) but the MIDAS device, for example, does.

9.4.1 Unravelling the hierarchical structure of SSEs

An SSE can be loaded into another higher-level 'parent' SSE, within which it will then exist as a 'child' event, along with any other child SSEs or ScoreEvents. The child SSE now has to be allocated device resources *in conjunction with* the other child events in the parent SSE, in order to be performed with them. Thus the device resource allocations and DeviceEvents stored in the child SSE are no longer necessarily valid, as identical resources may have been allocated to *another* child event which overlaps this child SSE within the parent.

Thus, for performance *within a parent*, an SSE needs to have its ScoreEvents analysed with account taken of all the other SSEs or ScoreEvents which may be present at the same within the parent.

Thus the parent SSE must unravel its hierarchy of events into a single layer of ScoreEvents, as shown in figure 99 below.

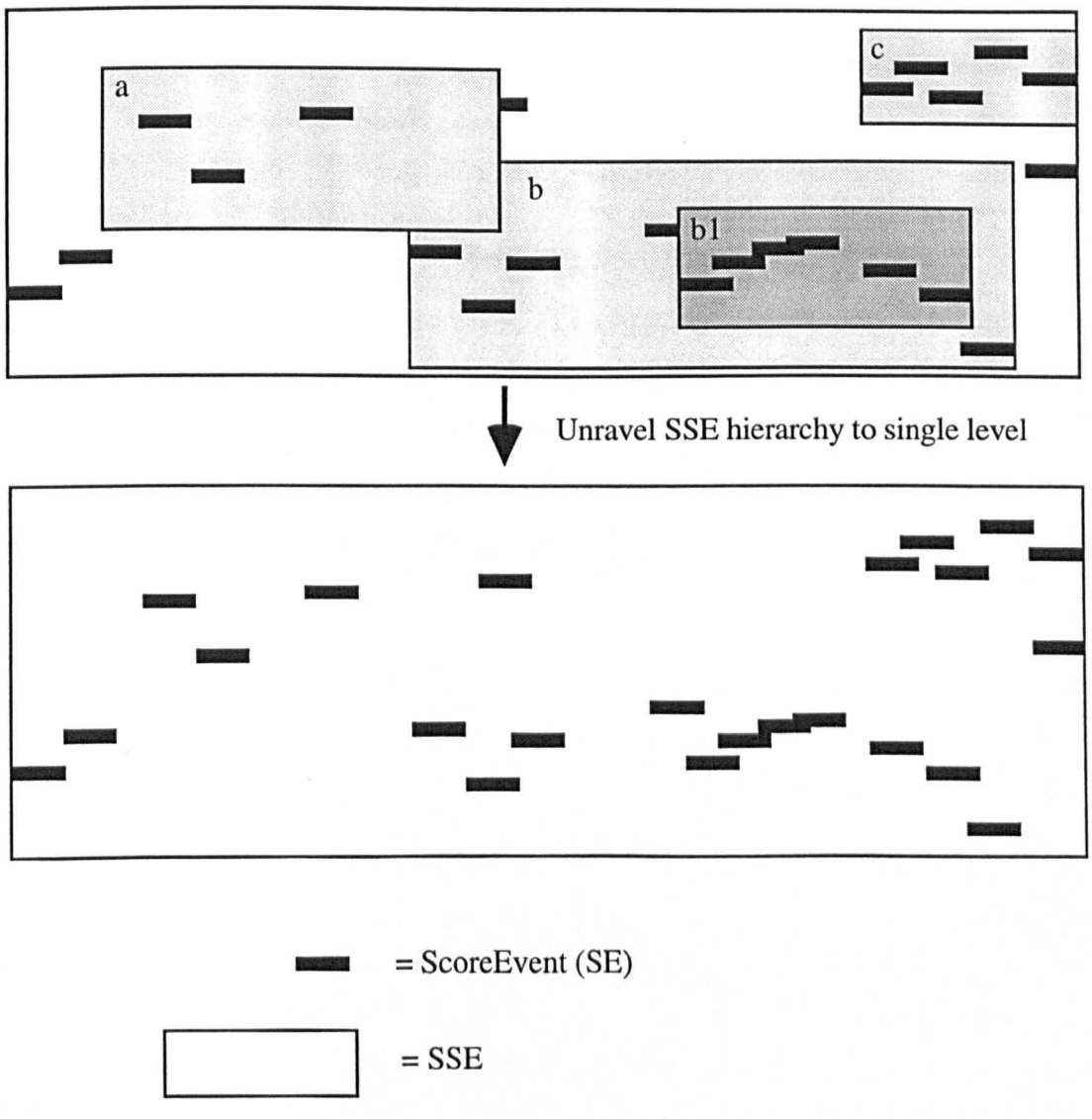


Fig. 99 Unravelling the hierarchical structure of an SSE

In the upper part of the figure the parent SSE contains ScoreEvents and three child SSEs labelled 'a', 'b' and 'c'. SSEs 'a' and 'c' contain child ScoreEvents, while the SSE 'b' contains ScoreEvents *and* a further child SSE 'b1' which in turn contains further ScoreEvents. The lower part of the figure shows the result of recursively unravelling each SSE into its child events.[1]

An 'unravelled' SSE thus contains *only* ScoreEvents, which each contain DeviceEvents for control of devices at level I, as described above. These DeviceEvents, when sent as messages to a device, can instantiate DCIPrimitives as units in the device, and can then start, stop and send input values to a unit.

---

1 It must be noted that at the time of writing, the methods which support this nesting of SSEs have not been fully implemented, but the object structures needed for this are in place.

Each 'unravelled' ScoreEvent has a time offset within its parent SSE, and its DeviceEvents in turn have a time offset within the ScoreEvent. When asked to play (ie when sent a 'play' Smalltalk message), a parent SSE first unravels its child events into a single layer of ScoreEvents. It does this by asking each of its child events (SSEs or ScoreEvents) to play at its start time offset relative to itself. This is done by sending it a 'playAtTime: t' message where t is its time offset within the parent.

If a child event is an SSE, it will in turn ask each of its child events to play at its start time offset within itself, and this continues recursively until the child event is a ScoreEvent.

When a 'playAtTime: t' message is sent to a child event which is a ScoreEvent, it then asks its child DeviceEvents to play, at *their* time offsets (time stamp).

Thus requesting an SSE to play, will result in a series of requests to DeviceEvents to play at a certain time, via the message *playAtTime: t.*

9.4.2 Playing DeviceEvents

A DeviceEvent 'plays' by transmitting its data values to the appropriate device on the appropriate interface.

9.4.2.1 Interfacing via 'RT output' system classes

Each DeviceEvent has a designated Protocol object (see 8.7), and can access a specified Smalltalk class to send real-time data out of the host computer's physical output ports. Such object classes are termed 'real-time (RT) output system classes' and each DeviceEvent has such a class assigned to its *rTOutputClass* instance variable. Each such E-Scape class uses primitive methods in itself or in another class to access the platform specific output ports.

E-Scape presently implements the ERS232 and EMidi classes. In Smalltalk-80 v2.3 on the Apple Macintosh platform, the EMidi class, for example, calls methods in system class Midi to access the low-level output routines for the MIDI port. Other versions of E-Scape running on different platforms would still have the EMidi class, which will understand the same messages. However, the EMidi class may be *implemented* differently, and may call a different low-level system class (to Midi), or invoke different methods.

Such low-level classes are relatively non specialised - simply sending raw data out of a hardware port. Each system class may actually be used to send DeviceEvents using several *different* Protocols. For example, both the 'MIDAS' and 'MIDI' Protocols have the EMidi class as one which they can use to send out messages. Each Protocol defines a set of message formats, which *may* actually be communicated using the *same* physical computer port and hardware standard.

As stated above, all such RT output system classes have been programmed for a particular platform. All must however respond to the message playBytes: atTime: onPortNamed: which has three arguments (shown below within square brackets):-

playBytes: [xx yy ... zz] atTime: [t] onPortNamed: [a port name]

The arguments are:

[xx yy ... zz] - a set containing (an arbitrary number of values (typically of 8 or 16 bits);

[t] - the time offset to send this data (relative to the start of the SSE);

[a port name] - the name of a specific port on this computer platform.

This latter port name will have been specified by the user when building a Device definition for a particular device in use by E-Scape, selected from those available for Devices of this type. It is possible that a single type of device might use more than one port. Thus, a certain type of device might have a SCSI-2 connection, through which commands to instantiate and control a certain category of unit (perhaps dealing with high data densities) can be sent. The same device may *also* have, for example, a connection from the serial port of the computer to control a different category of unit (perhaps dealing with low rate control data) on the same device.

To provide the user with the ports which are available, each output class should also be able to respond to the message 'availPortNames' by returning a list of names of the computer ports from which it can send data.

For example, the EMidi output class should respond to the 'availPortNames' message with the names: 'Modem' and 'Printer' (the names of physical output ports on the Macintosh), as MIDI data can (in theory - see below) be sent from either or both ports. The ERS232 class should also return both port names in response to the 'availPortNames' message, as again, either of these ports can be used for RS232 communication from the Macintosh.

These structures are in place in the current (v. 2.3) Macintosh platform implementation of E-Scape, but unfortunately, the low-level communication restrictions in this Smalltalk version (detailed in 7.5.1.2) mean that in practice only the Macintosh 'Modem' port is available for both these classes. The new release 4.1 Smalltalk version will allow full communication via a number of ports.

9.4.2.2 Sending DeviceEvent data

As described above, a DeviceEvent object implements the method 'playAtTime: t' where 't' is the time offset since the start of the SSE. A DeviceEvent contains a set of data values for each field of the message according to the MessageType from which it was created. These values now require sending to the outside world, via whatever port is required by each device. As stated in 8.8, a device can be controlled by E-Scape via a number of different ports, in fact each *category* of unit within a device can if necessary have a different Protocol, message format or physical port.

To transmit its data, the DeviceEvent sends its collection of data values to its RT output system class, also specifying the name of a physical output port to use. This RT output

system class is designed to format and send bytes to the outside world via primitive methods.

The Smalltalk code for the playAtTime: t method is:-

```
(self rTOutputClass)
    sendBytes: (self data)
    atTime: t
    onPortNamed: (self portName)
```

and is explained below, in two stages.

(i) Getting the RT output class of the DeviceEvent

The DeviceEvent first sends the message 'rTOutputClass' to itself to return the RT output class which it owns.[1] To implement this request, the DeviceEvent first finds out the DTSMTCategory of the DCIPrimitive in the Device which created it, and from this gets the real-time output system class. In detail:

Each DeviceEvent has a *dCIPrimitive* - the DCIPrimitive which created it. This DCIPrimitive in turn has a *parent* ScheduledDCIPrimitiveSlot (which may contain several other DCIPrimitives). The ScheduledDCIPrimitiveSlot exists in a ScheduledDevice of the SSE, and can access the 'model' DeviceDCIPrimitiveSlot from which it was created via its *modelType* instance variable. This DeviceDCIPrimitiveSlot has a *parent* - the Device which owns it, and a "category" (the *parent* DTSMTCategory of its *modelType* DCIPrimitiveSlot).

(ii) Sending data

The DeviceEvent then sends this output class the Smalltalk message

sendBytes: [bytes] atTime: [t] onPortNamed: [port name].

The three arguments to this message are:

- [bytes] - the set of data values owned by the DeviceEvent. The DeviceEvent obtains these by sending itself the message 'data'. This returns a collection of values that constitute the DeviceEvent's *data.*
- [t] - the time offset. This is simply passed on from the time specified as an argument to this 'playAtTime:' method).
- [port name] - the name of the port to send out the data from. The DeviceEvent obtains this by sending itself the message 'portName'. This gets the name of the port to use for this DeviceEvent. In detail:

  The DeviceDCIPrimitiveSlot looks at the *portNamesDic* Dictionary of its parent Device. This can have a port name to use for each unit category name. This DeviceEvent is owned by a DCIPrimitive or one of its inputs, which can access the unit category

---

1 This class is assigned to the DeviceEvent's *rTOutputClass* instance variable. This is accessed by a Smalltalk message of the same name, which is common (if potentially confusing) in Smalltalk.

(DTSMTCategory object) it is in. If the *portNamesDic* in the Device has a port name stored in association with the name of this category, then this port name is returned, otherwise the default port name of the DeviceType is returned.

9.4.2.3 Multiple Protocols

It is conceivable that a unit input might require messages to be sent using *more* than one Protocol (possibly using different RT output system classes and more than one port). E-Scape can cope with this situation by allowing a user to specify *several* MessagePrototypes when defining a module input (as described in 8.6.2.1). This would result - after Score processing - in *several* DeviceEvents being created, each with the necessary RT output system class and/or port name. This scenario, although unlikely, demonstrates the flexibility of the structures built into E-Scape.

9.5. Support for algorithmic composition in E-Scape

The term 'algorithmic composition', as used in the electroacoustic music field, can include complex generation of control parameters for an event, as well as the more normally held concept of the generation of events. Within its current structure, E-Scape can support both kinds of activity, within the *same* familiar and consistent system. At the time of writing, examples of such applications have not yet been produced.

9.5.1. The concept of 'algorithmic Instruments' within E-Scape

E-Scape also can present algorithm design within a uniform environment using the familiar score and instrument paradigm. Thus, algorithms can be specified inside an E-Scape Instrument structure, the only difference between such an 'algorithmic Instrument' and a 'normal' one being the rate and destination of the output data. A 'normal' E-Scape Instrument processes and assigns score parameter data to inputs of its DCTs which represent synthesis structures in a device (see 9.2 above). An 'algorithmic' E-Scape Instrument sends data to *newly* created score events which are then loaded into the score (SSE). Each of these new events will also have its own 'normal' Instrument to enable it to be performed, ie realised as sound.

E-Scape allows event or sonic data to be generated and stored in software by the PspProcessors in an Instrument. The bottom level will be events which are control inputs to the synthesis hardware attached. These inputs would typically control inputs to the audio-rate sound-producing processes occurring in the hardware. However, *if* the device can support such processes as are needed to implement an algorithm (eg logic gates, conditional branches, loops etc.), then the algorithm defined in the PspProcessor could be transferred (by the user) to the DCT part of the Instrument. This represents a transference of algorithmic processes from software to the device.

At the present stage of development of E-Scape, the PspProcessor algorithms are specified as a script of (restricted) Smalltalk-80 code. Eventually, the aim is to use a MAX-like iconic interface for specifying processing algorithms by graphic connection of processing units, in a similar fashion to the COMPASS computer-aided composition system (Mahling 1991).

This would mean that to the user, a PspProcessor algorithm would look identical to a DCT description (which will also have an iconic display). The same algorithm would appear in either location within an Instrument as a network of icons, although the underlying data structure of each is very different. The former is an active code block containing software functions; the latter is a descriptive structure with creation and connection codes which will be transmitted to a device to set up such a network within it.

If an Instrument's 'front-end' functionality is described by a DCT in this way, it then exists as a network of control-rate primitive signal processing units in the device (in addition to the audio-rate processes they are connected to). The E-Scape Instrument would then contain only *simple* PspProcessors, with these more complex DCT(s). The complexity now only exists in E-Scape as a *specification* of processes, rather than those executable processes

themselves. This has obvious benefits of increased speed, and a reduced data flow requirement from host to device.

A PspProcessor could in fact contain algorithms to create an entire score, and an Instrument containing it could possibly have *no* Psps. ScoreEvents using such a 'parameter-less' Instrument would not appear in a score display in the 'normal' parameter sub-windows (as shown in 8.10, figure 57, for example). They would only be displayed in the sub-window of a score display which shows each ScoreEvent as an icon, and treating the *Instrument* of each as a visible 'parameter'.

A score (SSE) could then in theory consist of a *single* ScoreEvent using this Instrument, which would simply act as a 'start' trigger for the processes in the Instrument. More typically, the algorithm in the PspProcessor *would* have some input parameters for its event-generating processes, which could receive parameter (Psp) values from the ScoreEvent.

In summary, E-Scape can specify (within an Instrument definition) algorithmic processes which can generate event data (via 'stage 1' processing - see 9.2.2). These processes can then be initiated by creating a ScoreEvent which uses this Instrument, and installing it within an SSE. If the device in use can support the signal processing entities which are needed in the algorithm, then the specification in E-Scape can exist as a *DCT* within the Instrument. A DCT is a *description* of the network of processes which are to be instantiated in hardware. The Instrument complexity (event generation and synthesis) is thus almost all in the device.

E-Scape will more typically be used with hardware which does *not* support such algorithmic processes, (eg which create events and call other lower level processes). The algorithm specification in E-Scape will then exist as a function definition in a PspProcessor within the Instrument. The PspProcessor specifies signal processing to be carried out within E-Scape when a ScoreEvent is created/edited which uses this Instrument.

The Instrument complexity is thus distributed between E-Scape host software (event generation) and the device (synthesis). Real time performance is not inhibited by host processing speed, as event generation occurs when such a score event (using an 'algorithmic Instrument') is added to the score. The only performance restriction resulting from this arrangement is that many *more* messages (several for each score event generated) will need to be sent to the hardware.

The advantage of doing ScoreEvent generation at the host level is that the resulting events can be seen and manipulated in a score display after being generated. This compares with systems which can certainly create complex musical material from a single event using a complex instrument (Bailey 1990), but whose output is not available for editing.

Thus the score / instrument boundary is movable: additional 'score events' can be generated by an Instrument, either within E-Scape or the device. The events can, if appropriate, be viewed as a single entity. If, for example, the Instrument produces a dense cluster of short

events, then it may be more useful to display only the 'source' event, and to consider the resulting set of events to be one - rather sonically complex - event.

9.5.2. Complex Instruments which generate events

If the processing occurs in the Device, then it is specified within a DCT just like any other type of Instrument. If, as is more likely with present devices, the algorithmic processing will occur in E-Scape, then two levels of event generation can be distinguished.

9.5.2.1 Generating 'sonic events' *within* a single ScoreEvent

An Instrument could contain a PspProcessor code block which may have one or more (Psp) score parameter inputs (or even have *no* inputs). It could also access a ScoreEvent's duration or start time. The code block could then create values and load them to the DCTSignals of the ScoreEvent at various times. Thus, time-varying device-level parameters could be generated for a ScoreEvent by algorithmic processes within an Instrument.

This parameter data would not *normally* be shown on the score display screen, but *may* optionally be displayed, as in 9.2.2.1 (figure 84). In addition, a future development will allow values *anywhere* within a PspProcessor's code block to be shown in the score display.

For example, an Instrument called 'Glissando' might have three 'i' rate parameters (Psps) named 'glissando step', 'start pitch' and 'pitch span'. A ScoreEvent using this Instrument can then specify values for these parameters, with the result that a glissando (a series of stepped pitch values) is played. Note that the event does *not* re-attack; it is a *single* event with a varying pitch parameter.

The code block to perform this would be created from the following user-entered text. (temporary variables within the block commencing with 't' by convention):-

```
'[ :paramHolder |
tNoOfNotes <- 'span' / 'glissando step'.    "(got from the PspFunction)"
tDur <- paramHolder sE duration.
tStepDur <- tDur / tNoOfNotes.
step <- 0.
[step > tNoOfNotes] whileFalse: [
        paramHolder
                loadVal: startPitch + (t * 'glissando step')
                atTime: step * tStepDur
                toDCTInputNamed: 'st. pitch'.
                                "simple example - only semitones in the glissando"
        step <-step + 1.
        ]
]'
```

9.5.2.2 Generating *new* ScoreEvents from a single ScoreEvent

New ScoreEvents could be generated using a special 'event-generating' type of Instrument. A ScoreEvent using such an 'event-generating Instrument' is then designated 'source', and will appear in the score display with some kind of visual indication that its Instrument may not create sound *directly*, but create *other* ScoreEvents. A code block of a PspProcessor in such an Instrument is similar in structure to the previous example above, except that the PspFunction data is not processed into data assigned to DCT inputs - the Instrument does not in fact *contain* any DCTs. The code block instead creates *new* ScoreEvents.

An example 'event-generating' Instrument might create a series of new ScoreEvents, each with a different pitch, to fit within the duration of the 'source' ScoreEvent which is using it. The 'source' ScoreEvent would also have values for its 'span' and 'glissando step' parameters. The Instrument would then calculate from these values the number of new events required and their duration, and then perform a loop to create each new ScoreEvent with this duration. In this example, the Instrument specifies a start time relative to the 'source' event start time, and a fixed Instrument (named 'X') to use for each new ScoreEvent it generates. The 'event-generating' Instrument also specifies a value for the 'pitch' parameter of the new ScoreEvent; this parameter must obviously be present on the 'X' Instrument.

The code block to perform the above functions would be created from the following user-entered text :-

```
'[ :paramHolder |
tNoOfNotes <- (valueOfPspNamed: 'span') / (valueOfPspNamed: 'glissando step').
                    " 'span' and 'glissando step' are ScoreEvent parameters"
tDur <- paramHolder sE duration.
tStepDur <- tDur / tNoOfNotes.
step <- 0.
[step > tNoOfNotes] whileFalse: [
        paramHolder
                createSEWithInstrumentNamed: 'X'
                atTime: step * tStepDur
                withDur: tStepDur
                withPspNamed: 'st. pitch'
                withValue: startPitch + (t * 'glissando step').
        step <-step + 1.
        ]
]'
```

More complex variation in each new ScoreEvent can be imagined, with the time, duration or choice of Instrument varied for each new ScoreEvent via more complex decision making or data processing algorithms. The above example has a single 'st. pitch' value specified for each new ScoreEvent (at t=0), but time-varying values could also be specified.

Recursive event generation could be performed, if the Instrument assigned to the newly-generated ScoreEvents is itself an 'event generating' Instrument. These new ScoreEvents could then, for example, be sent parameter values which are scaled down from the values received from the 'source' event. Thus, a large number of events could be produced from a single 'source' event using a single 'event generating' Instrument - for example to produce a fractal-like structure of "glissandi within glissandi". Such an Instrument would obviously need termination conditions to avoid infinite recursion.

The newly -generated ScoreEvents would then be *shown* on the score display. Thus the composer always is able to examine and edit the resulting material from such an 'algorithmic Instrument'.

9.5.3 Complex instruments which generate parameter control data

As stated above, the term 'algorithmic composition', as used in the electroacoustic music field, may include complex generation of *control parameters* for an event.

For example, a composer may wish to generate or specify a complex data pattern, which should be sent to one or more inputs of units in a synthesis device. This generation may involve mathematical operations, and/or involve decision making about data routing or selection of processes depending on several factors.

The synthesis device itself may support the kind of processes needed to do this, either as functions specified in a textual interface, or as connected graphic objects in the 'MAX' style. In this case, E-Scape can facilitate the *description* of such structures networks via the DCTs in an Instrument. However, many devices may *not* support such processes, and the required complex manipulations would have to carried out in the control software system to create the complex data which can be sent to the device.

E-Scape Instruments enable a user to perform this kind of algorithmic processing *either* by specifying the device structures needed to do it, *or* by specifying processing to be carried out in E-Scape itself, or some mix of the two. In both cases, however, the user works with the *same* Instrument structure, and the scoring parameters for it remain the same, whichever method is employed.

This concept can be illustrated by presenting the details of the two example Instruments outlined earlier in 6.2.3, both of which implement vibrato[1] and allow a composer to specify the nature of the *change* in vibrato with 'second order morphology' (Wishart 1985, p 66). This example Instrument allows the composer to specify a starting and ending vibrato *rate* (the frequency with which the pitch variation cycle repeats), and a shape which governs the

---

1 A cyclic variation of the pitch of a sound; above and below its normal value.

transition between these rates during the course of the event. The amplitude and shape of the vibrato itself are also specified but - for the sake of clarity within the example - they simply remain constant during the event.

The first Instrument is shown in figure 100 below, and specifies all the necessary processing within the device structure specification (within the DCT in the bottom half of the figure), ie as a set of connected units in the device. In contrast the PspProcessor is simple - it merely routes the score parameter (Psp) input data to the inputs of this device structure.

The details of how this processing works are not essential to this example, but a brief overview will be given: The device is assumed to already have a series of stored wave shapes. These are data tables which contain data pairs, for time and value, both of which are assumed to be within the range 0-1. The 'Table reader' unit in the device (shown in the top right of the DCT) can read such a table, and scale its time values by a specified amount. In this case this is the event duration, thus producing a succession of values over the course of the event. These values are then scaled and offset by the specified vibrato start and end speeds, and then used to control the frequency of the 'OSCIL' unit (shown bottom centre) which will produce a cyclic waveform with this varying frequency.

The output of this OSCIL then represents a time-varying frequency offset which can effectively add vibrato to the second 'OSCIL' unit on the left. This involves the intermediate conversion of data values to 'oct' units. These are units first defined in CSound (Vercoe 1986) which allow a *frequency* variation to effect a proportional perceived *pitch* variation (ie adding percentages of an *octave*, rather than absolute Hz frequencies).

A second example Instrument which performs the same function is show below in figure 101. It has exactly the same *presentation* to the composer as the first Instrument, with the same five scoring parameters (Psps) shown at the top left. As can be seen, the device structure (the DCT in the lower half of the figure) is now very simple, and all the data processing occurs via functions within the PspProcessor (at the top) in the Instrument specification.

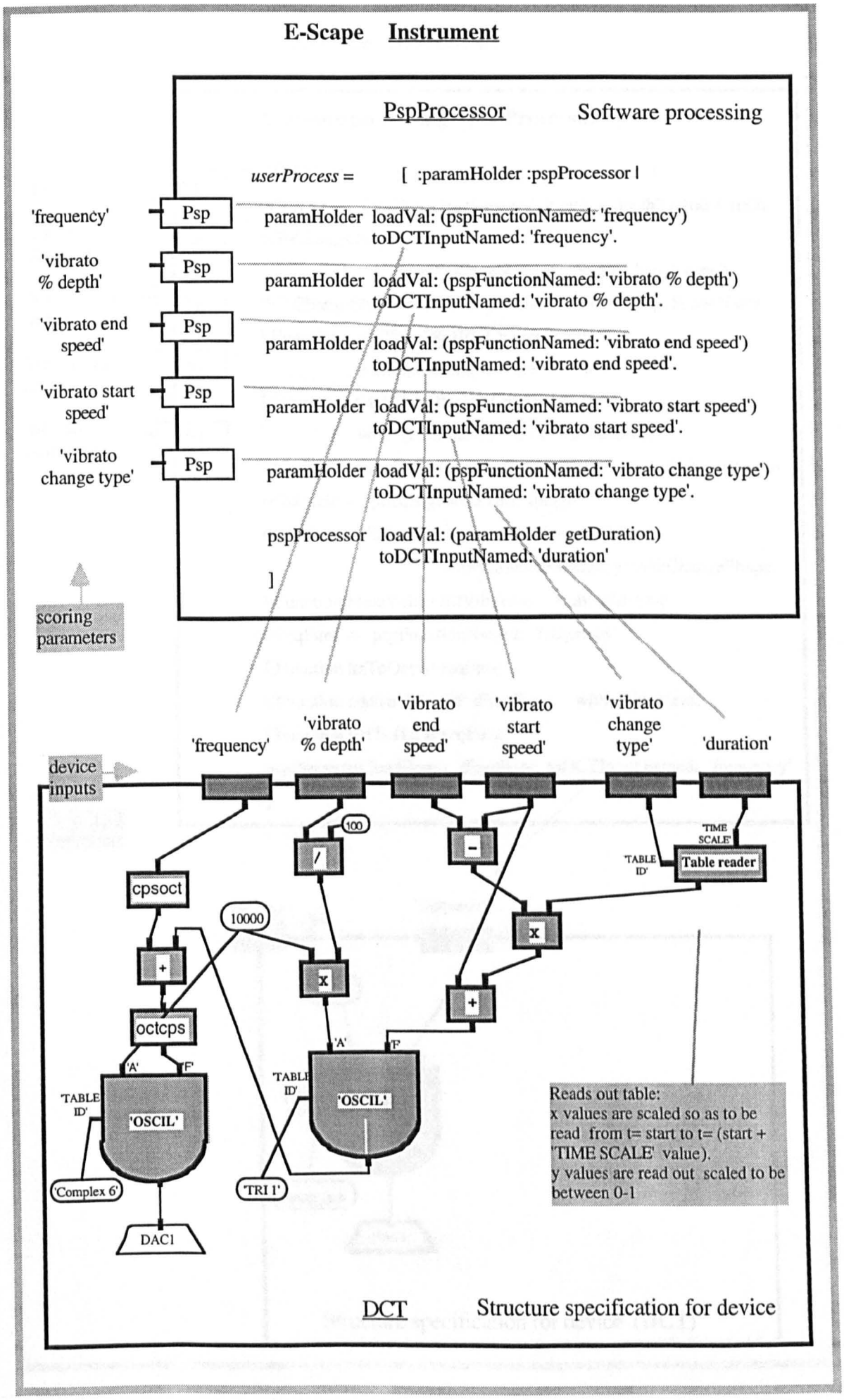


Fig. 100 Vibrato Instrument - example 1

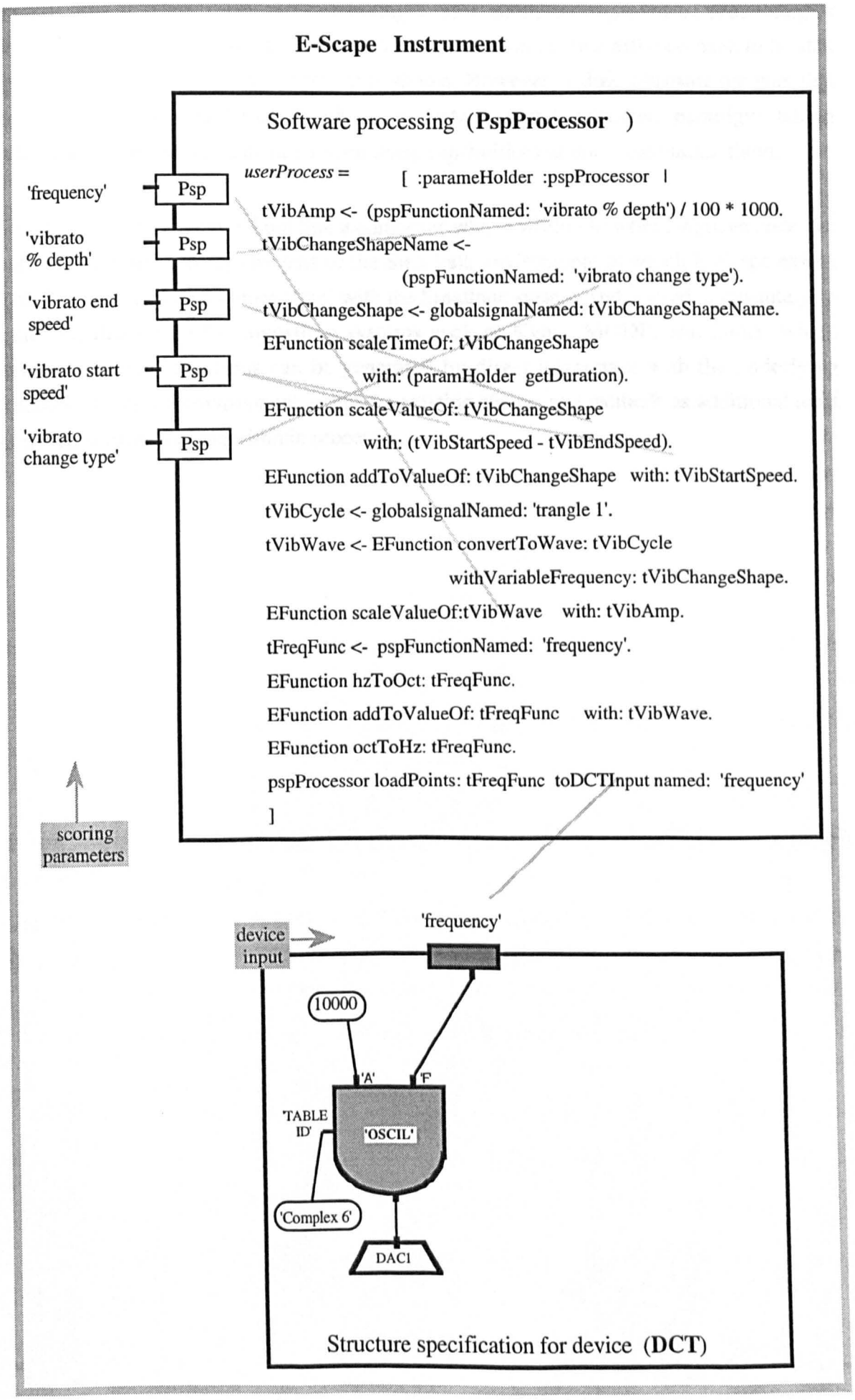


Fig. 101 Vibrato Instrument - example 2

Clearly this is not a very practical example, as a significant amount of processing is required within the E-Scape software, and a large amount of data will then need to be sent to the single input of the device structure shown. However, it does illustrate the way that complex processes can be used within a consistent score / instrument paradigm, taking advantage of the synthesis device's processing capabilities but not necessitating them.

Finally, it should be pointed out that a composer who is willing to write language code can safely access and modify any part of the Smalltalk environment in which E-Scape exists. This is a feature which comes 'free' with the Smalltalk system, and is used to advantage in other Smalltalk-based composition systems such as Kyma, MODE, and Dmix, where events or control functions can be generated by direct interaction with the underlying Smalltalk language environment, using the existing classes and methods as additional tools in the construction of algorithmic processes.

9.6 E-Scape system organisation

As described in this chapter and the previous one, the E-Scape system consists of a variety of top-level objects: ScoreEvents, SuperScoreEvents, Instruments, DCTHolders, DeviceTypes, Devices, and Protocols.

These top-level objects can be summarised as follows:

• DeviceTypes each describe the features of a type of synthesis device. Such features include its structure and address map, and the message formats used. They contain objects which describe the available types of synthesis unit and their inputs, and networks of such units.

• Devices each describe a particular device, such as its 'channel' assignments (if any), its id, or other set-up parameters.

• Instruments each define:

- a set of device structures (DCT objects);
- musically meaningful scoring parameters (Psp objects);
- processing between scoring parameters and device structures (PspProcessors).

An Instrument is built from a 'template' DCTHolder object, which contain sets of DCTs and PspProcessors.

• ScoreEvents each define the characteristics of a single sound event, such as duration, the assigned Instrument and its input parameter values.

• SSEs (ScoreSuperEvents) provide hierarchical score structures of nested time-stamped events, and device resource allocations for each event, based on the requirements defined in its Instrument.

• Protocols which define sets of types of message (MessageType objects) which can be used to send data to devices. Each MessageType also has a set of templates (MessagePrototype objects) which each specify a *use* of the message in a particular context.

These objects and their relationships are illustrated in figure 102 below.

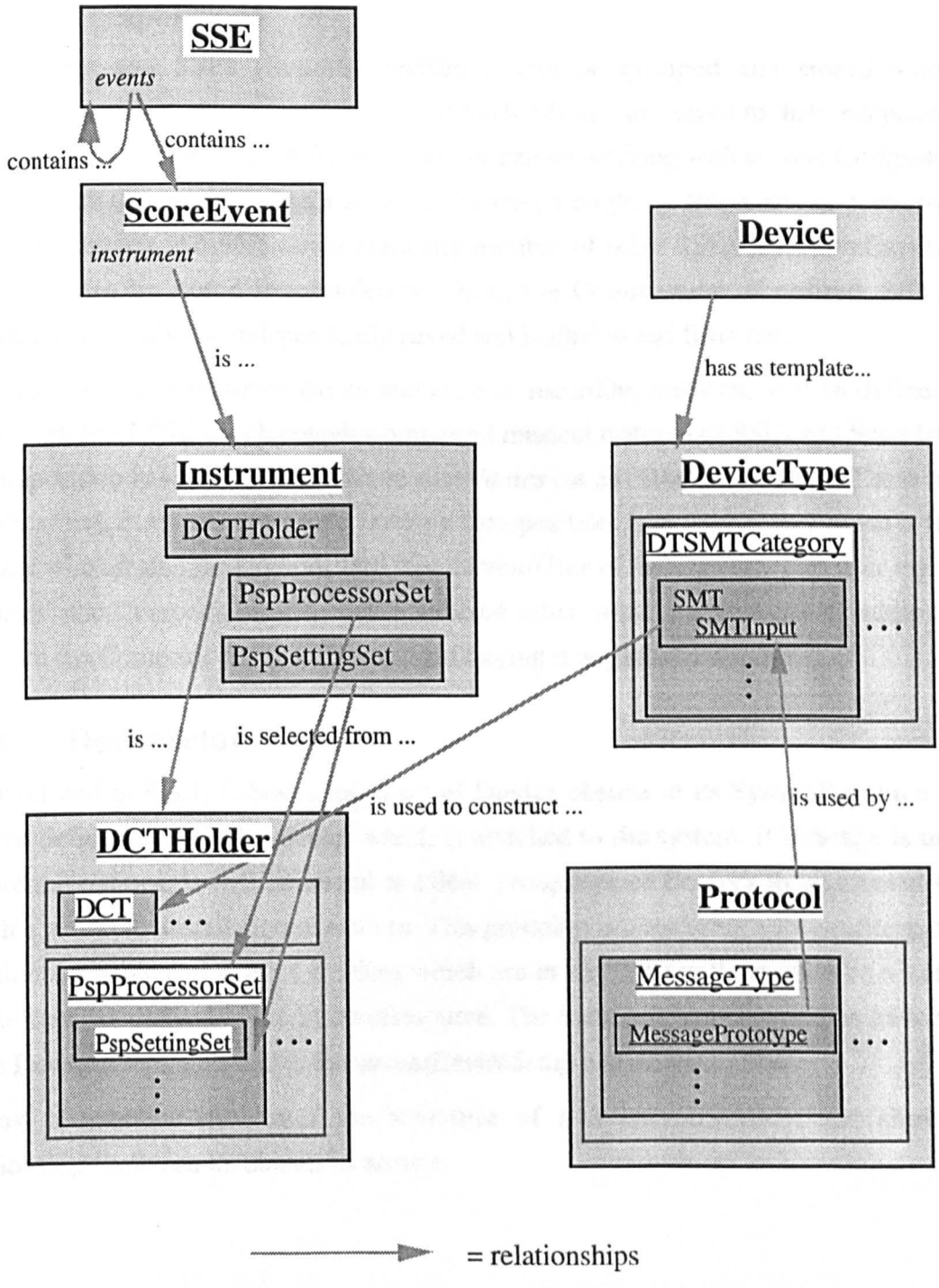


Fig. 102 The top-level E-Scape objects, and their relationships

9.6.1 The SystemResource

These objects require organising and storing within the E-Scape system, and this is done by a single SystemResource object, which stores sets of Instruments, DCTHolders, DeviceTypes, Devices, and Protocols.

A SystemResource also has a list of users, each described by a User object. Each User has a name, password, and other attributes (to be described in chapter 10). The SystemResource also has a *currentUser* instance variable, as it is designed to be a single user system.

9.6.2 Compositions

ScoreEvents and SSEs (SuperScoreEvents) can be grouped and stored within a Composition object. A Composition can be loaded from, and saved to disk independently of the E-Scape system as a whole. A composer can be working with several Compositions at once; each Composition has a set of SSEs which might be fragments or sections of a piece, or sketches. An SSE can contain any number of other SSEs and ScoreEvents, but these can also be stored independently within the Composition, if desired. SSEs and ScoreEvents can also be independently saved and loaded to and from disk.

The final piece as performed (to an audience or recording medium) will be defined in a single high-level SSE which contains organised musical material of SSEs and ScoreEvents. A Composition has a set of one or more *userNames* (as an instance variable). These names specify which users are allowed to access a Composition. The default user name is that of the user who created the Composition (the *currentUser* of the system). This user may then "publish" the Composition - to enable selected other users to access it - by adding their names to the Composition's *userNames* (or flagging it as globally available).

9.6.3 DeviceSetups

As described in 9.6.1, E-Scape has a set of Device objects in its SystemResource. Each Device defines a particular device which is attached to the system. If E-Scape is used in different locations, it will be useful to allow *groupings* of Devices to be created which match a particular installation of devices. This grouping is done using a DeviceSetup object, which has a name and a set of Devices which are in use. This will typically be a subset of all the Devices defined in the SystemResource. The SystemResource will then have one of these DeviceSetups assigned to its *currentDeviceSetup* instance variable.

Figure 103 below illustrates the structure of a SystemResource, and shows the relationships between its objects as arrows.

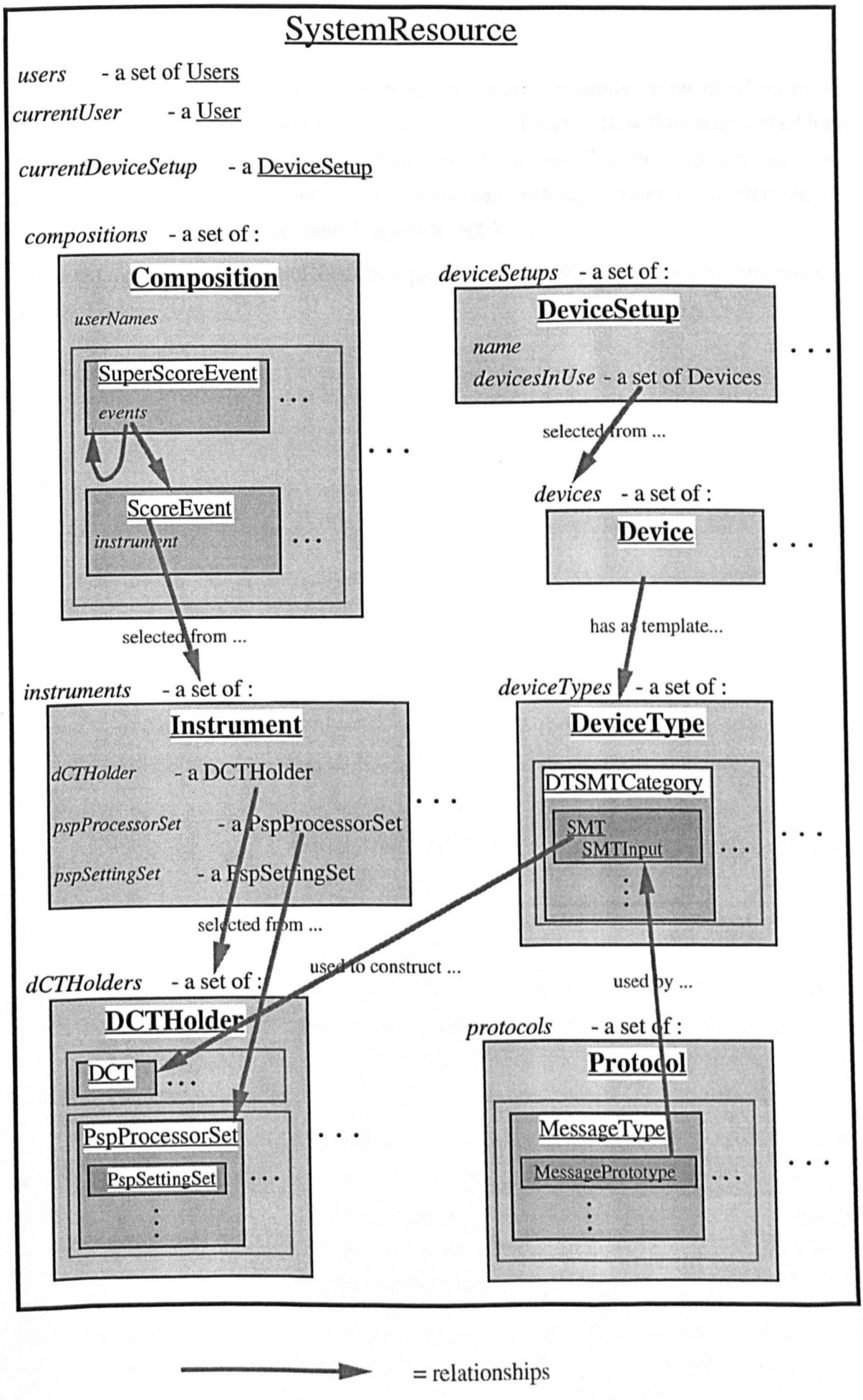


Fig. 103 The structure of the E-Scape SystemResource

9.7. Conclusion

This chapter has described the way E-Scape software functions, in terms of its major component objects, and their interaction and operation. Chapter 10 will then described how these various operations are controlled and seen by a user. Section 11.3 presents some examples of E-Scape objects which have been built, although examples of such objects have also been presented throughout chapters 8 and 9.

The compositional and control facilities provided by E-Scape are then summarised in chapter 12.

10. E-Scape in use

This chapter follows on from the previous two, and described the facilities available to users of E-Scape. E-Scape's user-interface is not yet developed enough for use by composers not versed in Smalltalk. Hence, this description of user capabilities is tied in with the operation of the software at the programming level, and is not intended to be a user manual.

It describes five levels of E-Scape user, who can operate at different levels of understanding of the system, and construct objects at different depths of complexity. A user at each level sees different objects as being *fixed* ('existing') or as being user-definable within E-Scape.

A level 1 user can create ScoreEvents, using existing Instruments selected from a library.

A level 2 user can assemble Instruments using an existing template of components, and define the parameters of connected devices (which are of an existing type selected from a library).

A level 3 user can build 'module types' which describe structures in a device, using lower-level modules (of an existing type selected from a library).

A level 4 user is able to define new DeviceTypes, which each describe *type* of synthesis device.

A level 5 user can specify new communication Protocols in E-Scape, or edit existing ones.

Reference to section 8.2, which summarises the major E-Scape objects and their relationships, may usefully be made during this chapter.

10.1 Levels of E-Scape user

Throughout the previous two chapters, much mention has been made of "the user". Many objects have been described, which can be specified or built by a user, yet mention has also been made of composers and the *lack* of a need for them to know about the structure of many of these objects.

To help a user to make sense of E-Scape, five *levels* of user have been designated. A user at each level sees different aspects of the system as immutable or as editable, and has different abilities to edit objects. At each level, different objects can either be regarded as 'given' - *fixed* entities which can be interacted with, or alternatively as user-definable - entities which may be *constructed* by a user as well as used.

Level 1 use provides the 'simplest' view of E-Scape, involving the least ability to edit objects. Level 5 provides the most 'complex' view, allowing a user to build objects which describe low-level functionality, requiring expertise and knowledge of communication protocol and device specifications. Even at this level of detail, the aim is still to *not* require a knowledge of Smalltalk programming.

It is envisaged that each user has a designated level, and can then choose (or be authorised via a password, or a UNIX-style 'super-user') to change to a higher-level. In an educational situation, many users may use the same E-Scape system at different times, and each user can have limits imposed on the range of levels he/she may access, in order to prevent confusion.

The abilities of a user at a particular level will subsume the abilities of the levels below it. Thus, a level 4 user, for example will be able to perform all the activities of levels 1 to 4, and will not then make any distinction between these levels.

The various levels of user have the following abilities:-

• A level 1 user (henceforth written as 'user-1') can create musical events (ScoreEvents) which use a specified Instrument, which can be selected from a library. Such a user can compose scores with no knowledge of the internal structure or functioning of Instruments required. This user would also use a network of devices provided, and not be able to alter the specifications of a device. However, a user-1 can *select* a set of device specifications from those already available within E-Scape.

• A user-2 can assemble Instruments using components from a selected DCTHolder template which contains matching sets of components from which an Instrument can be constructed. Each DCTHolder contains definitions of synthesis structures on one or more types of device, and will typically contain *several* sets of score parameter processing facilities. The user-2 can select a set of scoring parameters (and a set of initial parameter values, if desired) to build an Instrument.

A user-2 can also specify *which* hardware devices (of types already known to the system) are connected to the host computer running E-Scape. This user would also specify various set-up parameters of each connected device, such as its operation mode, communication channels and id.

• A user-3 can build 'module types' (SMT objects which describe structures in a particular type of device) hierarchically from other modules (SMTs), or the pre-defined low-level modules (PrimSMTs) which are part of a DeviceType specification. A user-3 can also build DCTHolder objects from which a user-2 can construct Instruments. This involves creating PspProcessors, and DCTs.

• A user-4 is able to specify new DeviceType objects in E-Scape, which each describe a new *type* of synthesis device. This description includes such things as a definition of a device's lowest-level synthesis structures (units), and the characteristics of its address map. The communication Protocols used (eg. 'MIDI') must already be defined within E-Scape, however - Protocols cannot be edited, or new ones defined.

• A user-5 can specify *new* communication Protocols in E-Scape, or edit existing ones. This specification will include the ports used, message fields and their source of data from score or Instrument objects.

The following sections in this chapter describes in detail how each level of user can set up, edit or create these various aspects of the E-Scape system.

10.2. Level 1 user activities

As stated above, a user-1 is able to select a set of Devices to use (a DeviceSetup object), and create scores (SSE and ScoreEvents) within a Composition (see 9.6.2).

10.2.1 Selecting a DeviceSetup

A user-1 can select a DeviceSetup from those available in the SystemResource (see 9.6.3) or create a new one. Each DeviceSetup object contains a set of Device objects which describe the characteristics and parameters of a synthesis device.

The user must check that the chosen DeviceSetup matches the physical situation, ie that each Device object in the DeviceSetup has a corresponding synthesis device which is active and connected to E-Scape. The parameters and settings of each device must also match the specification in the corresponding Device. The selected DeviceSetup will then become the *currentDeviceSetup* of the SystemResource.

10.2.2 Constructing a single ScoreEvent

A ScoreEvent is created by choosing an Instrument, specifying a duration, and event parameter values. These stages are as follows:

10.2.2.1 Selecting an Instrument

A ScoreEvent is created by first choosing an Instrument from a list (menu) of those which are "usable" - a *subset* of all the Instruments stored in the SystemResource. To be "usable", the necessary Device objects must be available to enable the DCTs in the Instrument to be realised. Each DCT in an Instrument has a DeviceType - describing a type of device on which the DCT's components must be realised as units. At least one Device of this type must be present in the *current* DeviceSetup, ie a synthesis device of this type must be connected to E-Scape.

If a device is disconnected, the corresponding Device object must also be removed from the current DeviceSetup. The removal of a Device may make certain Instruments (currently in use by ScoreEvents) effectively "unusable", and the user would then be invited to alter these Instruments, swap them for 'usable' ones, or remove these (now un-performable) ScoreEvents.

10.2.2.2 Specifying event duration

The duration of the ScoreEvent can be specified overtly at this stage, or be determined subsequently when the user specifies parameter values by on screen drawing with the mouse.

10.2.2.3 Specifying score parameter values

Event parameters may either have a single value at the start or end of the event, or have time-varying values, depending on the *rate* ('i', 'e', or 'k' respectively) of the Psp object which describes the parameter (see 8.4.2)

A single ScoreEvent can be displayed in a window which can show the data for one or more of the ScoreEvent's parameters, each being in a separate sub-window. Time within the ScoreEvent goes from left to right horizontally, while parameter values are displayed vertically within each sub-window. Data for each parameter may be time-varying *if* the device supports this. For example, the ScoreEvent being edited in figure 104 below has three time-varying parameters (with a rate of 'k'). Note the stepped display which reflects the quantisation applied to entered values by the Psp (see 8.4.4.1), so that the display reflects the perceived sonic parameter changes.

The ScoreEvent also has a fourth parameter named 'attack time' which has a rate of 'i'[1] and hence can only have a single data value specified at the start of the event. This value is shown as a small block on the left of the 'attack time' sub-window (the third one down in the figure).

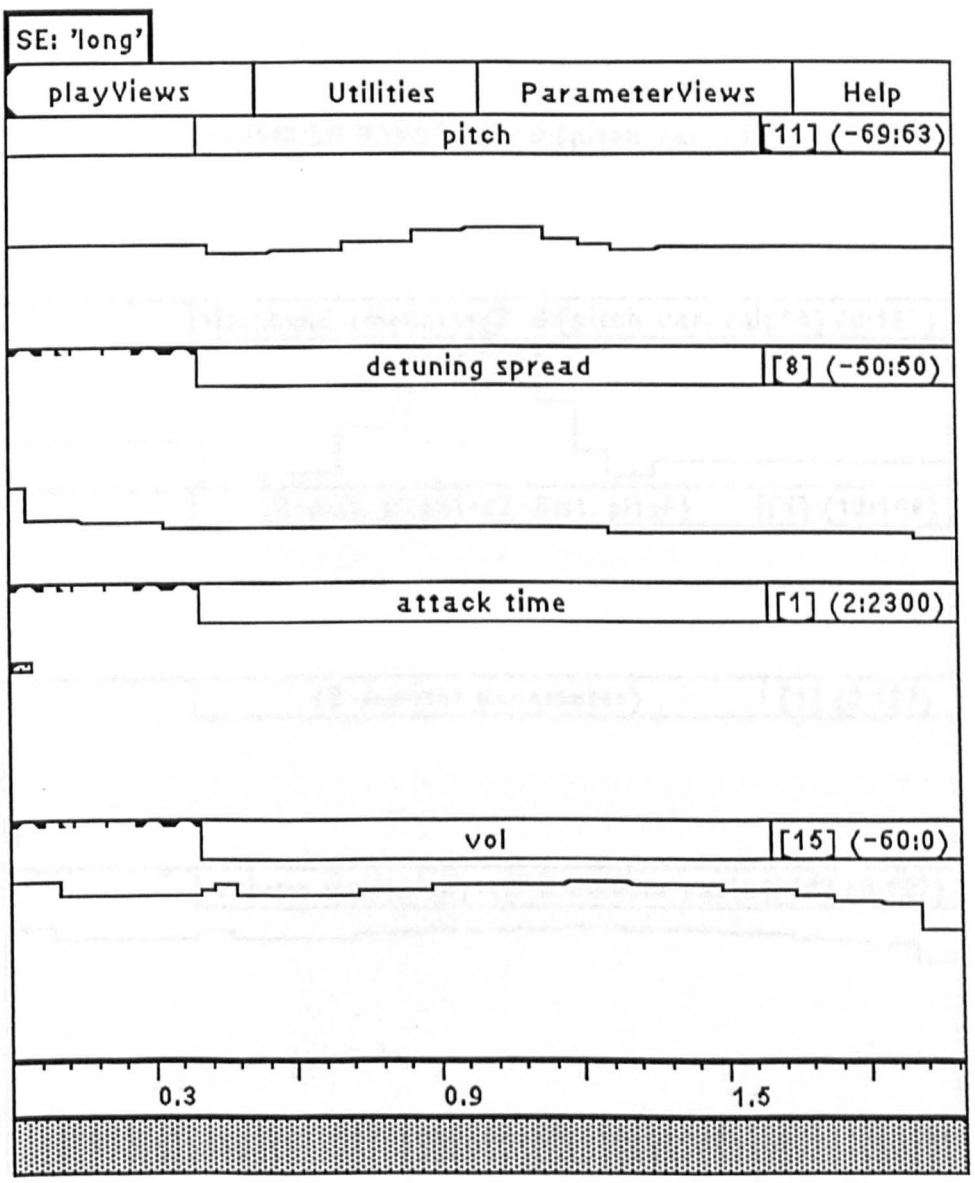


Fig. 104 A Score Event display showing parameter traces

1 The parameter (Psp) is 'i' rate because it connects - after processing - to inputs of the DCT (device structure description object) which are 'i' rate.

The Instrument of this example ScoreEvent has a single DCT - the example shown in 9.1.3.5 . This DCT contains *two* device-level modules, each corresponding to a 'PART' unit in the device. The 'pitch' and 'detuning spread' parameters are processed by a PspProcessor, whose output is connected to three inputs of the DCT labelled 'Z-A/B:BENDER RANGE', 'Z-A/B:pitchbend' and 'Z-A/B:st. pitch' (see 9.2.2.1, figure 82).

The processed parameter data which is assigned to each input of the DCT can also optionally be shown in a ScoreEvent display, as illustrated in figure 105 below. This display would normally only be used for diagnostic purposes, when designing an Instrument and its PspProcessors.

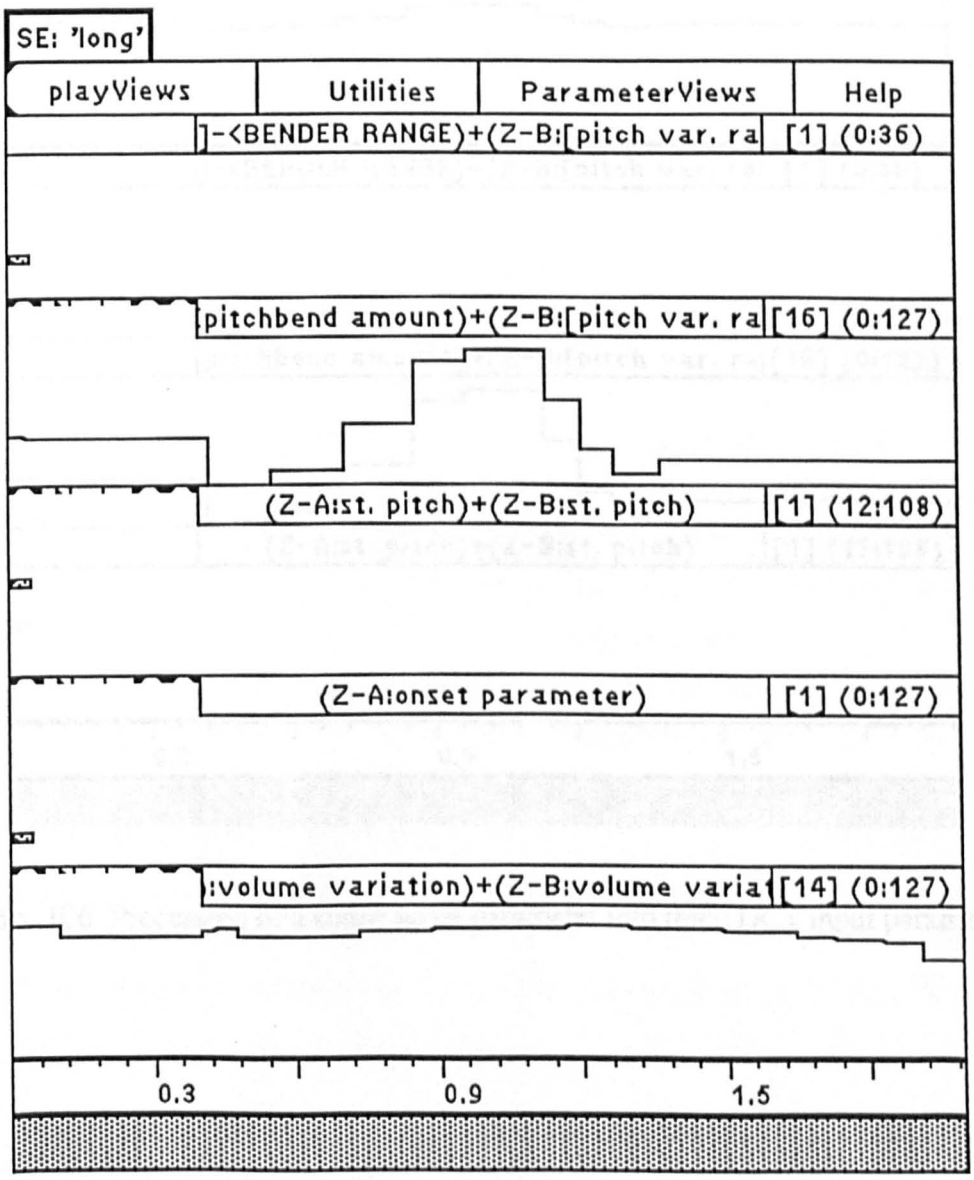


Fig. 105 A Score Event editor showing optional DCT input traces

The user would normally choose to view both the score parameter (Psp) trace, together with the DCT input traces which result from it, as shown in figure 106, below.

Note that the 'rate' of each DCT input is *not* set by the user at this level. The DCT input is connected to one or more inputs of submodules which correspond to primitive synthesis units in the device. These submodules are specified by the user as being of a certain *type*, such as an 'OSCIL'. This type specification (a PrimSMT object in E-Scape) is created by a user to match the specification of units available in a device.

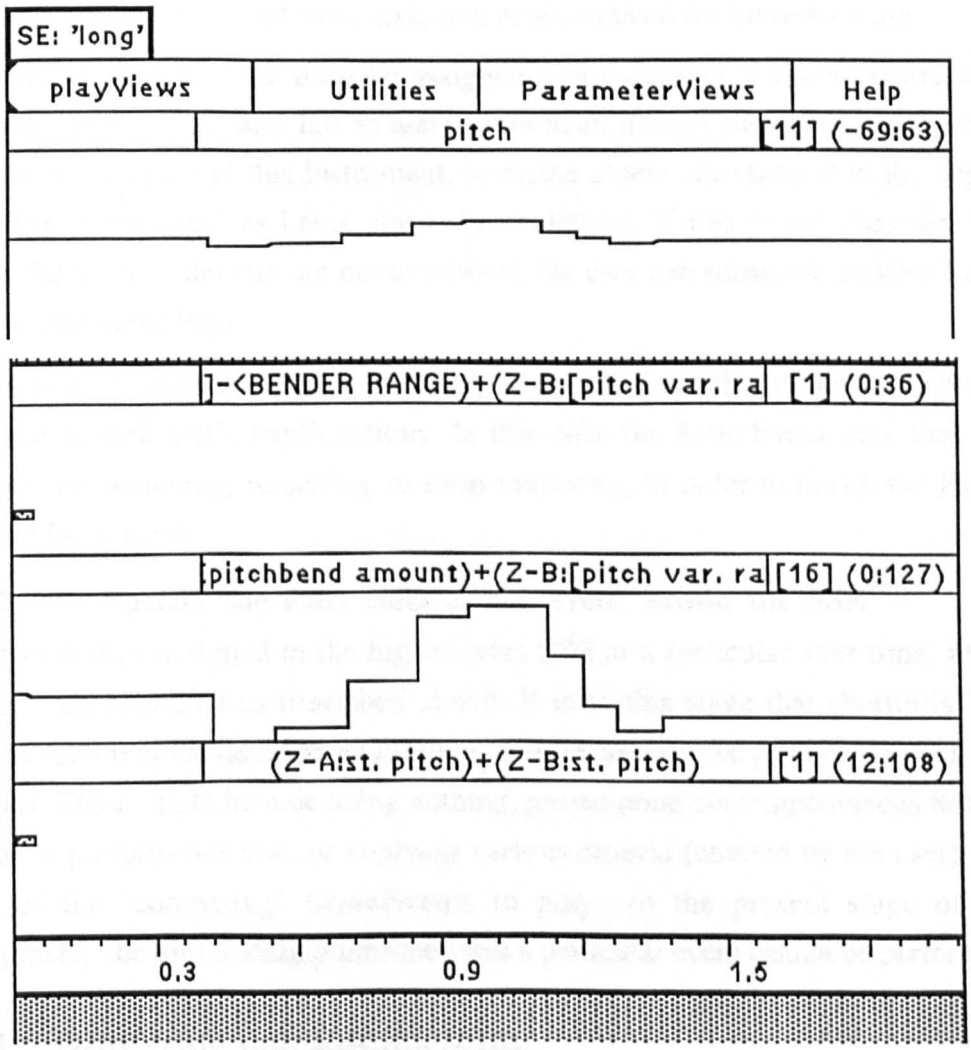


Fig. 106 Processing of a single score parameter into three DCT input parameters

10.2.3 Constructing a score (SSE)

To specify a score, a user creates a new empty SSE (SuperScoreEvent) object. Other 'child' SSEs or ScoreEvents can then be added into this parent SSE, with a specified start time within the parent. The SSE can also be edited by adding, removing or altering its child events. The process of building a new SSE consists of the following steps:

10.2.3.1 Select existing event

The user can select an event (ScoreEvent or SSE), which may be accessed from storage within a Composition, loaded from disk, or a newly-created for immediate use.

All ScoreEvents to be used must be assigned to a *currently* available Instrument. If a ScoreEvent loaded from disk has an assignment to an Instrument which is *not* present, the user is prompted to load this Instrument, or create a new one (subject to the appropriate DeviceTypes and Devices being currently available). If this is not possible, (perhaps because the original devices are not available), the user can substitute another Instrument which has the same Psps.

Alternatively an alternative Instrument may be assigned which does *not* have Psps which match the ScoreEvent's PspFunctions. In this case the ScoreEvent may then need its PspFunctions renaming, re-scaling or even removing, in order to match the Psps of the substitute Instrument.

10.2.3.1 Specify the start time of the event within the SSE

The event is then assigned to the higher-level SSE at a particular start time, and device resources are allocated as described above. It is at this stage that shortfalls in device synthesis resources are detected by E-Scape, and the user can be prompted with a choice of strategies. These might include doing nothing, pre-empting contemporaneous ScoreEvents in order to perform this one, or applying various criteria (entered by the user) to decide which of the 'competing' ScoreEvents to play. At the present stage of E-Scape development, the user is simply informed that a particular event cannot be performed.

10.2.4 Displaying and Editing SSEs

The user will first select an SSE within a Composition (see 9.6.2). In the screen shot in figure 107 below, the composition menu is shown. A collapsed display window is also shown at the bottom right, which can be clicked on to bring up an existing SSE which is being edited.

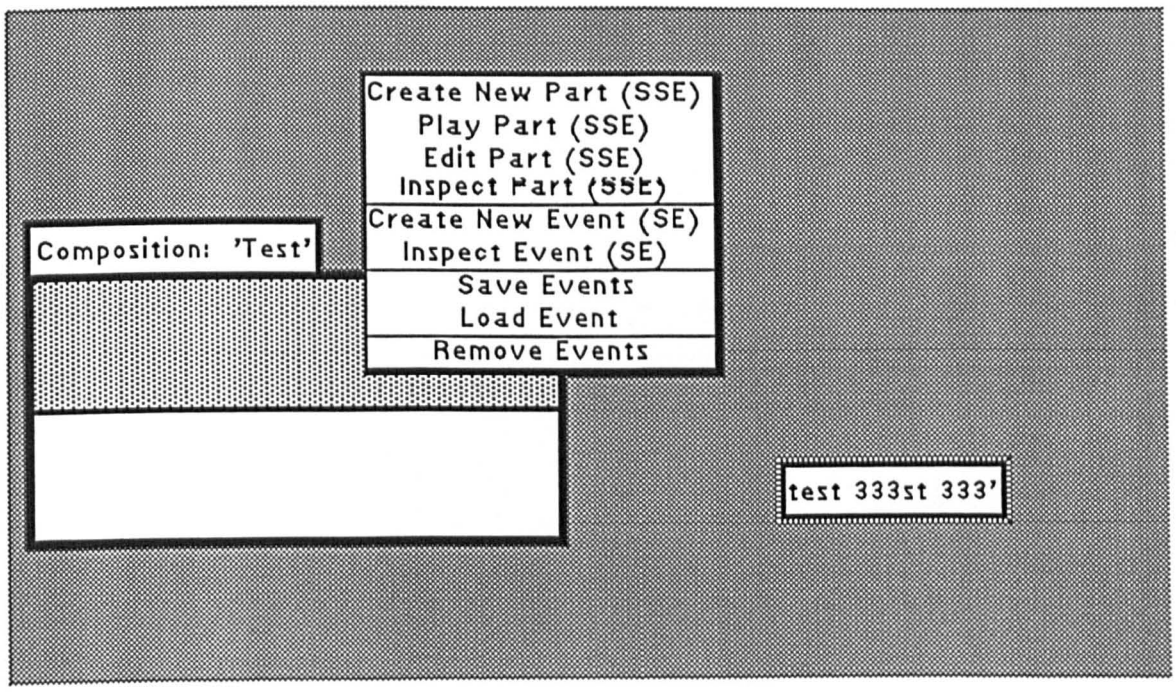


Fig. 107 Selecting an SSE to edit

The user may then open a graphic display window for a selected SSE. These displays are one of the GUI features which *have* been implemented in E-Scape, as they form a key component of one of its *innovative* design goals: to facilitate the specification and display of multiple time-varying parameters for each ScoreEvent.

Each SSE can have event parameters (Psps) specified by a user as 'default' ones which will be displayed initially. If no defaults have been specified, E-Scape checks if there is a parameter called 'pitch' and displays that, otherwise it shows the first parameter.

In figure 108 below, an SSE display is showing a single 'pitch' parameter for its events.

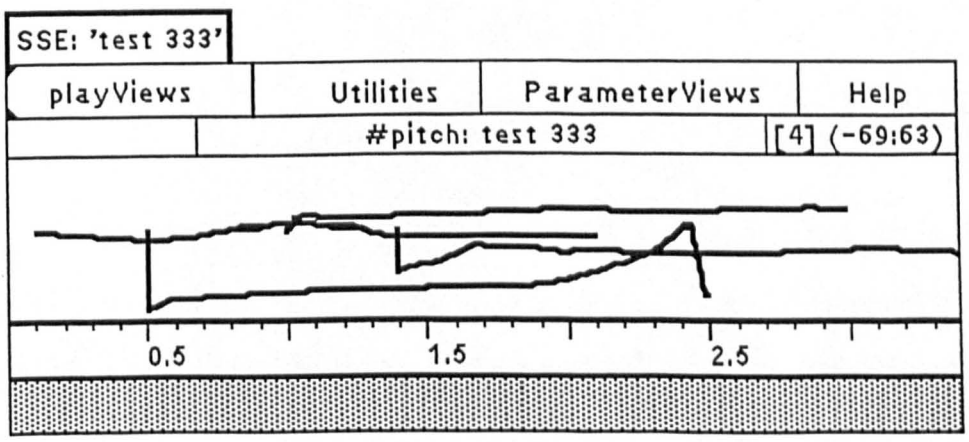


Fig. 108 SSE display's default view

The user can then choose to display any or all of the other event parameters, as illustrated in figure 109 below. These will be the superset of all parameters of all Instruments used by the ScoreEvents within the SSE. In the example illustrated, a single Instrument is used by

all four ScoreEvents, which has parameters 'pitch' (presently displayed), 'detuning spread', 'attack time' and 'vol'.

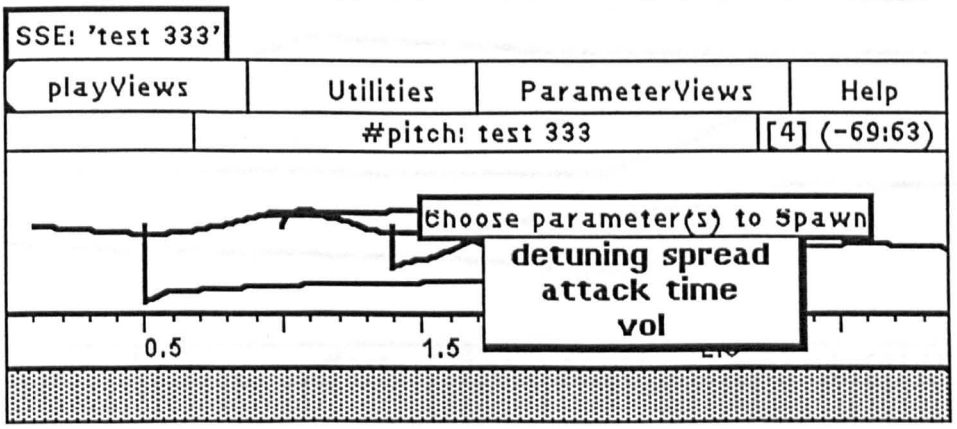


Fig. 109 Selecting additional score parameters to display from a menu

Figure 110 below shows the resulting SSE display after the user has chosen to additionally display the 'vol' parameter.

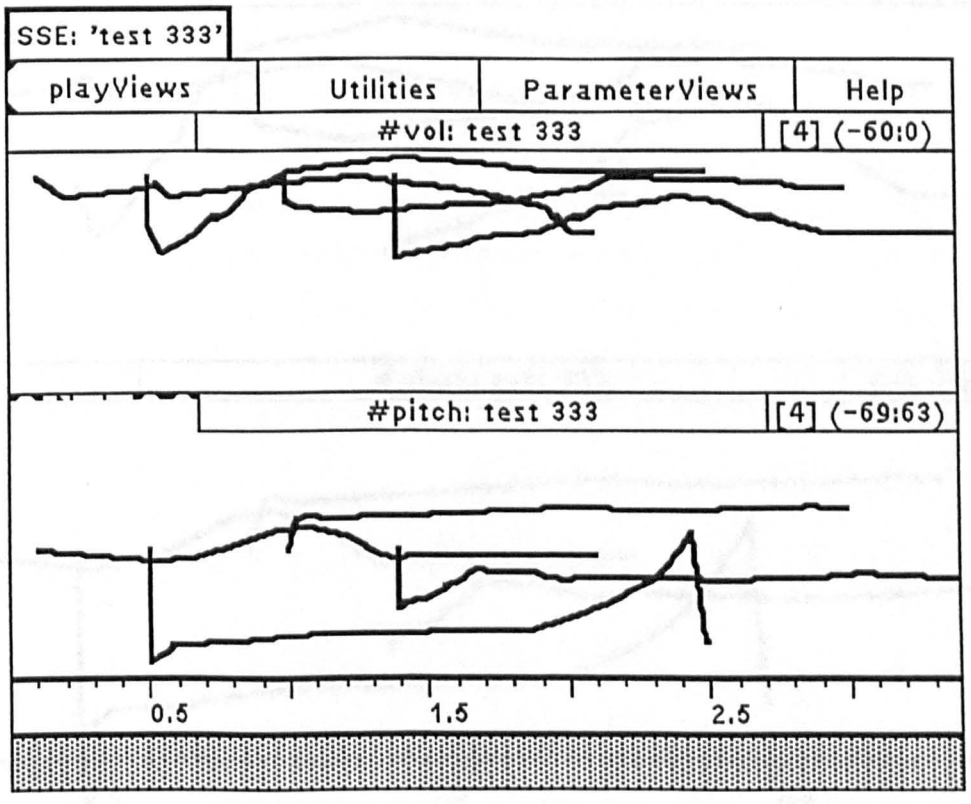


Fig. 110 Two parameters selected to be displayed for score events within the SSE

In figure 111 below, the same SSE display now has all four available Instrument parameters shown. As all the four ScoreEvents use this Instrument, they all have traces in all the parameter sub-windows.

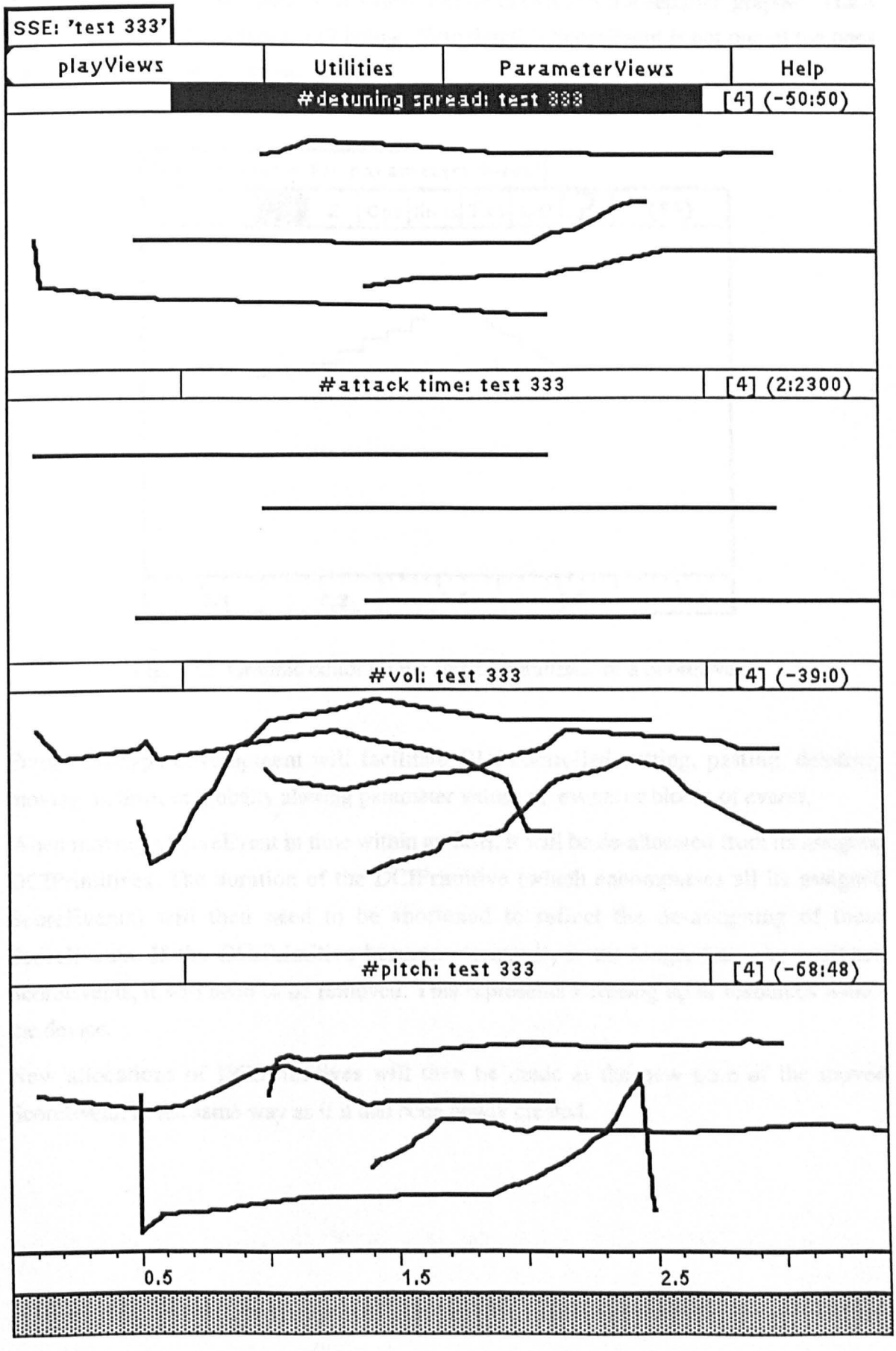


Fig. 111 All four available parameters selected to be displayed for score events

At present, ScoreEvents' parameter values can be drawn in via a separate graphic 'Trace Editor', as illustrated in figure 112 below. Note that this ScoreEvent is not one of the ones present in the previously shown SSE.

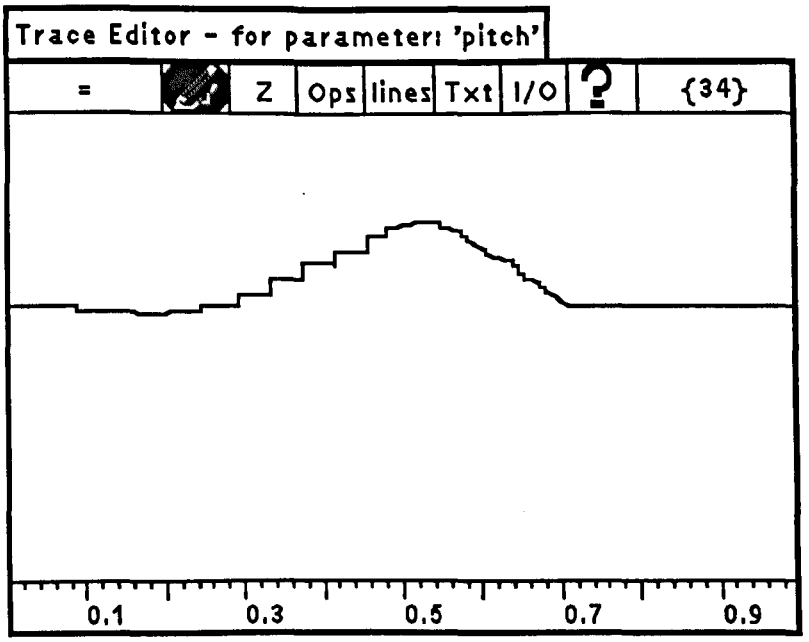


Fig. 112 Graphic editor on the 'pitch' parameter of a ScoreEvent

Future E-Scape development will facilitate GUI-controlled cutting, pasting, deleting, moving in time, or globally altering parameter values of events or blocks of events.

When moving a ScoreEvent in time within an SSE, it will be de-allocated from its assigned DCIPrimitives. The duration of the DCIPrimitive (which encompasses all its assigned ScoreEvents) will then need to be shortened to reflect the de-assigning of these ScoreEvents. If the DCIPrimitive becomes 'unused', ie no longer has *any* assigned ScoreEvents, it will need to be removed. This represents a freeing up of resources within the device.

New allocations of DCIPrimitives will then be made at the new time of the moved ScoreEvent, in the same way as if it had been newly created.

10.3 Level 2 user activities

As stated in the introduction to this chapter, a user-2 can construct Instruments at a high-level by selecting a DCTHolder object (see 9.1.2) which acts as a construction kit for an Instrument, containing sets of matched components which can be selected by the user.

A user-2 can also specify particular synthesis devices as being connected to E-Scape, as long as the *type* of device is known to E-Scape (ie defined in a DeviceType object). The user can specify the characteristics of the device (in a Device object), such as its id number, and channel settings, if any.

10.3.1 Constructing Instruments

The user first selects a DCTHolder (see 9.1.2). Each DCTHolder defines synthesis structures on one or more types of device (via DCT objects - see 8.4.3), and contains different sets of score parameter processing facilities (ie sets of PspProcessors).

Each PspProcessor presents scoring parameters (via Psp objects), and one or more sets of default values for these parameters. Each set of default values is stored in a PspSettingSet object, which has a default value for each Psp.

The user can then select one of these sets of scoring parameters for an Instrument (ie select a particular set of PspProcessors from those available), and then select a set of initial parameter values (a PspSettingSet) if desired.

The following description reiterates some of the description of Instruments given in 9.1 above, and presents the user actions which are involved in the construction of Instruments from DCTHolders. These actions consist of the following steps:

10.3.1.1 Selection of Instrument construction template (DCTHolder)

A number of DCTHolders will have been created by a level 3 user, and stored in the SystemResource (see 9.6.1). The user-2 can then select one of these DCTHolders. Each contains one or more DCT objects, which describe synthesis structures (networks of units) on a particular type of device. Each DCT corresponds to a particular DeviceType object; thus selecting a DCTHolder represents selecting a particular set of structures to be used on one or more types of device.

10.3.1.2 Selection scoring parameters (PspProcessorSet)

As stated above, the DCTHolder contains various sets of score parameter processing facilities as sets of PspProcessor objects. Each PspProcessor has a number of Psp objects defined in it, which describe scoring parameters which can be specified in a score, and processed into input data for the synthesis structures specified in the DCTs (see 10.3.1.1 above).

Each set of PspProcessors thus describes a *different* set of scoring parameters which facilitate compositional control of the same synthesis structures. Each set of PspProcessors

is stored in a named PspProcessorSet object within a DCTHolder. The user can then select one of these PspProcessorSets via a menu of their names.

10.3.1.3 Specification of default score parameter values (PspSettingSet)

The PspProcessorSet may contain one or more sets of default values for its Psps (as described in 9.1.1). Each set of default values is contained in a named PspSettingSet object. The user then selects one of these PspSettingSets, or can specify *new* values to create a new PspSettingSet. In the latter case, the new PspSettingSet will automatically be stored in the DCTHolder for future use.

10.3.2 Current user interface

The present interface embeds these Instrument construction activities within a series of surrounding Smalltalk 'example' methods, with particular names and values hard coded in, as arguments to Smalltalk messages within the example method. In future development, these arguments will be supplied from GUI dialogues or menus, but this aspect has not been developed thus far. Figure 113 below shows a screen dump of the current user interface which allows a variety of these example methods to be invoked.

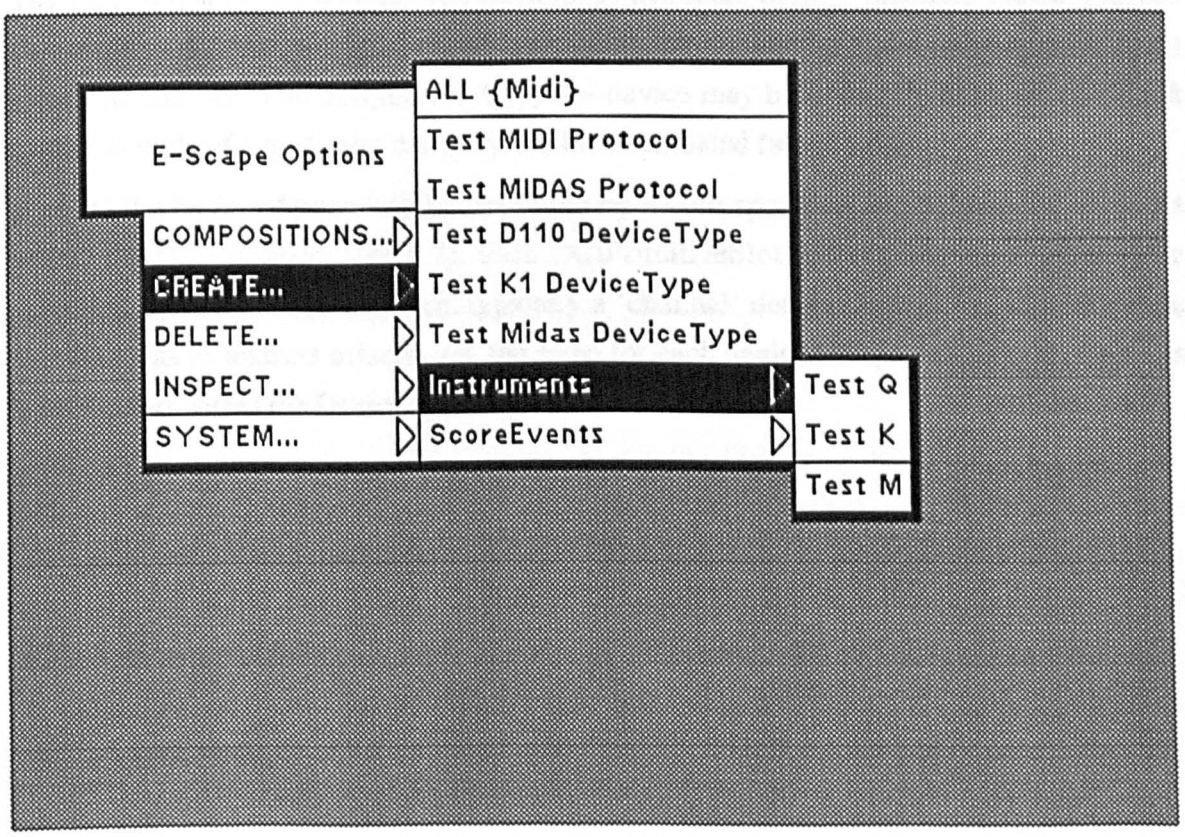


Fig. 113 Current user interface for building example Instruments

10.3.3 Creating a DeviceSetup

A DeviceType can contain many different stored Device objects, each of which describes a synthesis device with particular settings (eg id, channel numbers etc). All these devices are unlikely to be in use at once. Thus, *groupings* of devices which are connected to E-Scape can be described by DeviceSetup objects, each of which contains a subset of these Devices.

A user-2 can create a new DeviceSetup or edit an existing one.

The user creates a new DeviceSetup by defining a name for it, then adding Devices. Each Device may be newly created, or can be an existing one already defined within a DeviceType, which may then be copied and edited if necessary. Any *new* Devices defined are stored in the appropriate DeviceType as well as in the DeviceSetup being created.

The user can edit an existing DeviceSetup by adding additional Devices, or deleting or editing Devices within it.

In addition, a user-2 can describe a new device configuration by creating new Device objects (based on a DeviceType already defined in the current system).

10.3.3.1 Creating a Device

The user first selects a DeviceType from those available in the SystemResource. Various parameters can then be specified, as appropriate for the DeviceType: most will require a device *id* number to be designated. A type of device may have fixed 'slots', within which synthesis units of a particular category can be instantiated (see 8.5.2).

If so, each slot in a device will be described by a corresponding DCIPrimitiveSlot object within the Device object (see 8.7). Each DCIPrimitiveSlot will require information to be specified by the user for a device, typically a 'channel' designation. Other aspects of the slot - such as its address offsets - are the same for each device of a particular type, and thus are specified within the DeviceType object.

10.4 Level 3 user activities

As stated in the introduction, a user-3 can build synthesis structure definitions as 'module types' (SMT objects). Each module type is built out of lower-level modules which are installed as 'sub-modules' within it. These sub-modules are in turn built using a lower-level 'module type' (a SMT or PrimSMT object) as a 'template', as described in 8.6. These lower-level module types may already be defined and stored within the DeviceType, or may in turn be built by a user-3. Any newly-constructed module types are then stored as a 'library' within the DeviceType.

Such a user may *not*, however, specify the *lowest* level module types which describe primitive synthesis units on a device - and out of which all other module types are ultimately built - or specify *new* DeviceTypes or module categories within them.

A user-3 can also build DCTHolder objects (see 10.3 above), out of which Instruments can be built. This involves the construction of a DCTHolder's components: DCTs, PspProcessors, and PspSettingSets.

10.4.1 Building synthesis structure templates (SMTs)

The user can build up an SMT object which describes a synthesis structure, as a connected network of lower-level 'child' submodules, as described in 8.6.4. Each module may correspond to a type of synthesis *unit* in this type of device, or may itself contain further modules within its structure.

A user first selects a DeviceType object to work within, and selects an (existing) category for the new module type (SMT object). The user can then create and name a new 'blank' SMT object. The following user activities are then involved:

10.4.1.1 Specifying child submodules within the SMT

A sub-module of an appropriate category is created and installed within the new SMT object. The user creates each sub-module using a particular module *type* selected from a menu. Some devices may impose restrictions on which sub-modules can be used to create a module type; module types of a particular category may not necessarily be allowed to contain just *any* kind of submodule. This requires a framework to allow the specification of any restrictions on *how* module types may be constructed. Thus each category of module type can have categories specified, from which the submodules used to construct it must come.

However, some types of device (for example many MIDI-controlled ones) impose restrictions on the number and type of submodule, which are often arbitrary. To facilitate the description of such restrictions by a user within a uniform object-oriented framework has proved problematic, without introducing a large degree of complexity - both to the structure and user operation of E-Scape - with no guarantee of success in describing any type of device which may encountered in the future. These problems have resulted in the device functionality recommendations described in 4.3. For those types of device where

these kind of restrictions apply, a user must operate at level 4, and create module type specifications directly, as described in 10.5.1.2.

10.4.1.2 Specifying and connecting SMT inputs to submodule inputs

The user can then create named inputs for the new SMT, if desired, and connect each input to one or more inputs of submodules, as described in 8.6.4. This will eventually be done by MAX-like graphical linking of icons, but at present is performed by specifying arguments to example methods - specifying the name of a destination submodule, and an input of that submodule to connect the SMT input to.

There is no necessity to specify such SMT inputs or connections for them, as described in 8.6.4.3. If the user does not specify a connection to a particular submodule input, then a corresponding SMT input is automatically created and linked to it by E-Scape. Thus for an SMT where each input of its *sub-modules* is simply duplicated as an input to the SMT, no user action is necessary. This is especially useful for structures on MIDI-based devices, where overt user-specified connections between modules are rare.

10.4.2 Building an Instrument construction template (DCTHolder)

Building a DCTHolder involves the user creating a new blank DCTHolder and naming it. Its components can then be created and loaded into it. These components consist of one or more DCTs, one or more sets of PspProcessors, and sets of default values for their inputs.

10.4.2.1 Constructing a DCT

As described in 8.4.3, each DCT object describes a structure in a single type of device. Each DCT is built out of one or more sub-modules (DCTSubModule objects), and has inputs which are connected to these submodules, as illustrated in figure 114 below.

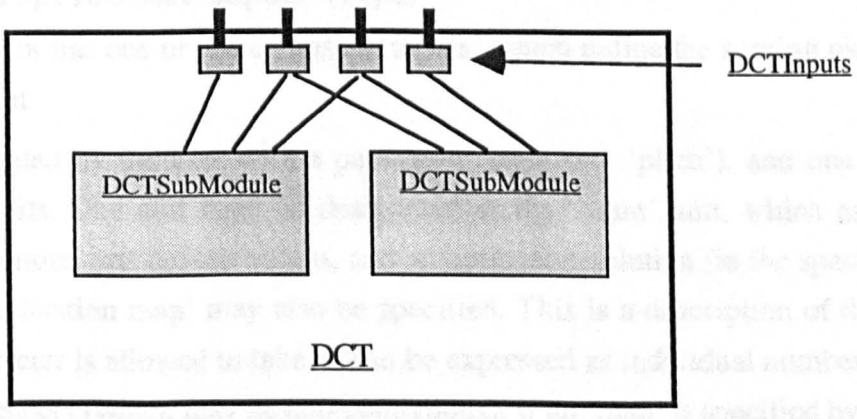


Fig. 114 A DCT showing its inputs connected to its submodules

Each sub-module is created from a particular module *type*, (a PrimSMT or SMT object) which acts as a template. These module types are themselves built up (see 10.4.1) or are defined as primitive objects by a user-4 (described in 10.5.1.2 below).

To build a DCT, a user will undertake the following activities:

- **Creating sub-modules**

(i) The user first selects the *type* of device for the DCT, by selecting a named DeviceType object. (ii) The user then selects a named module *category* in the DeviceType from those which are eligible (see 8.6.1). (iii) Finally the user creates a sub-module by selecting a named *module type* from the selected category. This module will then be contained within the DCT as a child sub-module.

Steps (ii) and (iii) can be repeated to load additional sub-modules into the DCT.

- **Creating and connecting inputs**

The user first creates inputs for the DCT structure, each with a specified name.

For each of these inputs, the user then specifies a destination submodule, and an input of that submodule to which the DCT input is then connected. As described in 9.1.3.2, a user need not necessarily specify connections to *all* the inputs of submodules.

10.4.2.2 Constructing PspProcessors

As described in 8.4.4, PspProcessors define the scoring interface of an Instrument, and its connection to device structures. Within the DCTHolder, the user can define one or more named *sets* of PspProcessors (stored in a PspProcessorSet object).

Each PspProcessor has a number of defined Psp objects which act as inputs to the PspProcessor, and describe the parameters which can be specified in a score.

A PspProcessor also has a CodeDictionary which defines how parameter data from the score is processed into input data for the synthesis structures specified in the DCTs (see 10.3.1.1). The construction of a PspProcessor involves the user specifying both these entities.

- **Specifying PspProcessor inputs (Psps)**

Each PspProcessor has one or more Psps as 'inputs' which define the scoring parameters for the Instrument.

Each Psp is created by the user with a parameter name (eg. 'pitch'), and one or more measurement units. One unit must be designated as the 'basic' unit, which must have minimum, maximum, and default values, and an optional resolution (ie the spacing of its values). A 'quantisation map' may also be specified. This is a description of the values which the parameter is allowed to take. It can be expressed as individual numbers, and/or as *ranges* of numbers (which may be non-contiguous). If no 'map' is specified by the user, E-Scape creates one automatically from the minimum, maximum and resolution settings, with uniform contiguous values.

All these settings are constrained, however, by the values which the inputs to device structures (specified by the DCTs) will accept. At present, the system merely warns the user that the specified parameter value will, after processing, result in a value which is outside the acceptable range of one or more of the DCT's inputs. Future E-Scape development will attempt to determine the parameter ranges by iterative approximation techniques to find the set of parameter value ranges which will produce acceptable

processed (device-level) DCT input values. This area is problematic, however, if multiple score parameters are processed, as there could be many different sets of acceptable input ranges which depend on each other.

- **Specifying the processing of score parameter values (ECodeDictionary)**

This processing is performed by an ECodeDictionary object (described in 8.4.4.2).

An ECodeDictionary contains a block of Smalltalk source code (named 'userProcess') which takes in score parameter data, processes it, then sends it to inputs of the device synthesis structure. This is specified by the user as Smalltalk text within a text editing window provided by the ECodeDictionary.

The ECodeDictionary can also contain other named user-defined code blocks - written to perform commonly needed operations which are needed within the 'userProcess' code block (if the requisite functions are not among the utility functions provided in E-Scape). At present, such additional functions have not been found to be necessary, but if a more complex 'userProcess' code block were required than in the present examples, they could provide a way of modularising its functionality. To create such a block, the user will first name a new block, then write its code in the same way as for the 'userProcess' block. This code block can then be used as a function within the 'userProcess' block.

The ECodeDictionary editor will provide menus of the standard E-Scape functions provided, so the user is only required to additionally know some basic structural and mathematical Smalltalk methods and constructs, as illustrated in 9.2.2 above. As explained elsewhere, the original aim was to provide for user specification of processing via iconic connection, but this aim has not been given a high priority within E-Scape development in the light of other systems (eg MAX, Unison, Patchwork) which provide such facilities. However, the aim has still been to insulate the Instrument designer / user from the Smalltalk system as a whole, and the code block text editor in the ECodeDictionary provides this insulation.

A PspProcessor sends its processed data *out* to one or more inputs of DCTs in the Instrument. When defining the 'userProcess' code block, the user can designate a DCT input to receive processed data, selected from a menu of the DCT inputs available (in the DCTs installed in this Instrument). Once a DCT input has been thus assigned, it is removed from this menu, and is then *unavailable* for connecting to other PspProcessors. This avoids possibly conflicting data being sent to a DCT input from different sections of the code block.

10.4.2.3 Specifying default parameter values (Psp 'settings')

The user can specify default values - one for each parameter (Psp) of the PspProcessors in a PspProcessorSet (see 9.1.2). These values are then stored in a user-named PspSettingSet object. A user can specify several different sets of values within a single PspProcessorSet.

10.5 Level 4 user activities

A user-4 can specify descriptions of new types of synthesis device using DeviceType objects. As described in 8.5, a DeviceType has *categories* of module types which can be created on devices of this type. Descriptions of structures or units (SMT or PrimSMT objects) can be stored in each category. Categories can also contain descriptions of the 'slots' in a device within which its units can be instantiated.

A DeviceType also describes the features of the address map (if any) within devices of this type, and the range of available id numbers which can be assigned to devices.

10.5.1 Specifying the address map

If a type of device has a fixed address map, the user is able to specify the names of *categories* of addresses in the map, and the base address within the device in each category. For example, in the 'D110' device, there are addresses categories named 'SYSTEM', 'TONE' and 'TIMBRE', and 'RHYTHM SETUP', which have base addresses of hex 010000, 040000, 030000, and 030110 respectively. Each slot in a device may then have address offsets in one or more of these categories, as described in 10.5.2.3.

10.5.2 Defining module categories

As detailed in 8.5.3, module categories are described by DTSMTCategory objects. Each category has a library of module types (PrimSMT objects), each of which describes a unit or structure of units in a device. Each category may also describe a number of slots within a device, into which units in this category can be instantiated.

A user thus defines a category by naming it, defining module types within it, then specifying each slot. The user also specifies the MessagePrototype to use for creating the message used to instantiate a unit in this category in a device.

10.5.2.1 Defining primitive module types

A user can specify the characteristics of primitive module types by creating PrimSMT objects (see 8.6.2). A category can also contain SMTs built from child sub-modules by a user-3, as described in 10.4.1.

The user first gives a name to the new PrimSMT, then specifies a *creation code*, *offset address* and *address category* for it. This information will be used when constructing a message to send to a device, in order to instantiate a unit of this type. Finally the user will create named inputs for the SMT, and specify their parameters.

- **Specify creation code**

The user first specifies a *creation code* for the new PrimSMT. For example the 'OSCIL' PrimSMT (in the 'UGP' category of the 'MIDAS' DeviceType) has a creation code of 6.

• **Specify address offsets**

If the DeviceType has a fixed address map, then each input of a PrimSMT will have an address offset within it. This offset will then typically need to be added to an offset for the PrimSMT. Thus, the user can specify an *offset address* and *address category* for the PrimSMT. The *address category* is used by E-Scape to access the correct base address of the DeviceType when calculating the absolute address to use. This will be needed when formulating a message to send data to the input of a unit (which corresponds to this PrimSMT input) in a device.

• **Specifying inputs**

The user must specify the parameters and characteristics for each input of a module type (PrimSMT object). This involves specifying various parameters for the input (as described in 8.6.2.1):-

- its *rate*, ie how often its value can be updated in the device.
- the maximum, minimum and resolution of values allowed to be sent.
- an (optional) default value.

The user also assigns a MessagePrototype which will create a message to send input values to the corresponding unit input in a device (as described in 8.8). The user may also specify an address offset for the PrimSMT input if appropriate (as described in the previous subsection).

10.5.2.2 Specifying channel assign parameters

If each slot in the device can have a 'channel' assigned by the user, then a MessagePrototype to do this can be specified by the user, and the address category used by it. For example, the 'PART' category of the 'D110' DeviceType uses a MessagePrototype named '(MIDI:Roland sys. exc. std) for- PART channel assign', and uses the 'SYSTEM' address category for channel assignment. This information will be used when constructing a message to send to assign the channel of each slot, as described below.

10.5.2.3 Specifying unit slot descriptions in a category

The user describes each slot in this category by creating a named DCIPrimitiveSlot object.

• **Slot address offset**

If the device has a fixed address map, the user can specify the address offset and address category (selected from the DeviceType's address categories).

For example, the 'PART' category of the 'D110' DeviceType has eight DCIPrimitiveSlots. The second slot (named 'PART-2') has an address offset (in hex) of h 00 01 76 in address category 'TONE', and h 00 00 10 in address category 'TIMBRE'. These offsets will be used when constructing message to send to units which are installed within this slot.

• **Channel assign address offset**

If units within the slot are accessed by a 'channel' label for the slot, and this channel can be assigned by the user, then a message will be required to communicate this channel setting to

a device. The MessagePrototype which should be used has been defined in the category itself, but the address offset to use for each slot needs to be defined by the user.

For example, the 'PART-2' slot (as above) has a channel assignment address offset of h 00 00 0E. This offset is only used to calculate the absolute address to use when formulating the channel assignment message for the slot. The 'PART' module category has the 'SYSTEM' *address* category; thus to calculate the absolute address for this slot, the DeviceType's base address for the 'SYSTEM' category (h 01 00 00) is added to this slot's offset (h 00 00 0E).

The 'channel' number for use within the message will be assigned by the user when he/she creates a new Device description using this DeviceType, as described in 10.3.3.1.

10.6 Level 5 user activities

A level 5 user can define new Protocols, or modify existing ones.

To create a new Protocol the user will give it a name, and the default time taken to for devices to respond to messages (as described in 8.8). The user will then specify a number of MessageType objects, and one or more Smalltalk 'output' system classes which will convey messages of this Protocol out of the computer platform on which E-Scape is running.

10.6.1 Defining MessageTypes

MessageTypes are described in 8.8.1. To define a new MessageType, a user first specifies (within a text editor) its field list, and supplied and derived fields. The user will then define one or more MessagePrototypes of this type (see 10.6.2). These operations are described below.

10.6.1.1 Specify field list

The user defines the message format by entering a field name or a fixed value for each field.

10.6.1.2 Specify supplied fields

The user specifies the names of the fields which will be supplied from objects (via FieldProviderSpecs - see 10.6.2.1).

10.6.1.3 Specify derived fields

The user specifies the names of the fields whose value will be *derived* from other fields, and enters a block of simple Smalltalk code (via a text editor) which will perform this (see 8.7.1.3). As stated elsewhere, the eventual aim is to facilitate the specification of such processing via an iconic GUI.

10.6.2 Defining MessagePrototypes

The user typically creates a number of MessagePrototypes within each MessageType. The user specifies a name, and is prompted to create a FieldProviderSpec object for each field which has been designated as 'supplied' in the MessageType (see 10.6.1.2).

10.6.2.1 Define FieldProviderSpecs

The user must define a FieldProviderSpec for each name in the *suppliedFields* of the MessageType (see 8.7.1.6). Each FieldProviderSpec describes which kind of E-Scape object (eg Instrument, Device or ScoreEvent) will supply a value for a message field, and how it will find out the value. Each new FieldProviderSpec automatically has the name of the field which it will supply a value for.

The user then selects the *type* of object which will supply a value for that field name. Currently, DCIPrimitive and DCIPrimitiveInput are the only types of object needed to enable all known synthesisers devices to be described. In the envisaged user interface development, objects (such as DCIPrimitive and DCIPrimitiveInput) will be selected by the

user from a menu, but presented more in 'non-programming' terms, for example using the names 'unit' and 'unit input' respectively.

The user then selects a Smalltalk message to be sent to the object to ask it to provide the field value. This message will also selected from a menu of messages, again presented as more user friendly text than the actual message string. For example, the Smalltalk message 'parentsChannel' (sent to a DCTPrimitiveInput) might be presented to the user in a menu as "channel of the device slot which the unit is in". To define a Protocol obviously requires considerable understanding and knowledge from a level 5 user, but the aim is still to avoid the need to write code when extending the system.

10.6.3 Specifying output classes

The user can select one or more 'RT output' Smalltalk classes (see 9.4.2) which can provide access to outputs of the computer for messages in this Protocol. These classes will already exist within E-Scape for each possible output port on the current platform.

10.7 Programmer-user activities

Above these five levels of user, a Smalltalk programmer can define new types of object and Smalltalk message to be sent, in order to access any parameters of objects which might be needed in the future for message building. Features or parameters of objects not conceived of in the present design may be added without affecting the way that messages are built, or the way that E-Scape is constructed.

A programmer can also define new 'RT output' system classes (see 9.4.2) to access the physical ports of a new computer platform on which E-Scape is installed. These classes can then be used immediately by Protocols, with no further restructuring required.

10.8 Conclusion

This chapter has described the capabilites of E-Scape as seen by a user, and the facilities it provides.

E-Scape allow a user to build Instruments (which define both synthesis and score parameter processing structures) from re-usable components ('module types'). A user can then construct scores (SSEs) by specifying high-level parameters for ScoreEvents using these Instruments, although without necessarily needing to know their internal structure. Various example Instruments and an example SSE are described in 11.3.

At all levels within E-Scape, a user can work in a "need to know" basis, able to construct and edit objects if desired, or simply *use* them, without any requirement to understand their construction or functioning.

IV - Assessment and Conclusions

This section (chapters 11 & 12) assesses, evaluates and summarises the achievements of the research presented in this thesis.

Chapter 11 presents an assessment of the elements of the work in this project and their degree of success in solving the problems and goals set.

Chapter 12 then sums up the project's achievements, and its contribution to the worlds of music technology and composition.

11. Evaluation and assessment

This chapter describes an evaluation which has been carried out of the various elements of the work in this thesis. This has examined three main areas:

Firstly, an assessment has been made of the feasibility of the proposals for the structure and features of computer music systems (presented in 4.1). This has involved the examination of representative existing systems, with a discussion of the changes or additions which would be necessary in order for these systems to meet the standards of behaviour proposed, and to operate as synthesis device and/or event specification sub-systems.

Secondly, the viability of the inter-system communication standards (proposed in 4.2) is been assessed by designing a detailed protocol which meets the proposed standard. This design has been for the MIDAS system, and has included a study of the internal operation of this system in order to facilitate the functionality required by the protocol.

Thirdly, an assessment has been carried out of the degree of success achieved by the E-Scape prototype event specification software system in meeting its design specification and solving the problems posed.

11.1. Assessment of system structure proposals

There are at present no synthesis systems or devices which fully implement the behaviour proposed in 4.1, ie the ability to act as a sub-system within a loosely-coupled computer music system.

However, several existing systems or devices do implement - fully or partially - some aspects of the communication and behaviour standards proposed for devices in 4.3, or may have the potential to be adapted to do so.

An assessment of whether the proposed structure is useful and reasonable has thus been carried out by examining a representative selection of existing systems, with regard to:

- their present level of implementation, if any, of the proposed structure and behaviour standards, either as devices or event specification sub-systems.
- the developments which would be needed in order for existing systems or system components to adapt or conform to the proposals;
- the feasibility of these developments being carried out in future.

11.1.1. MIDAS as a system component

The MIDAS system (see 3.2.1.9) has been designed from the outset to be controlled from a variety of external systems, ie to act as a device. As described in 11.3 below, a set of messages (the 'MIDAS' protocol), and a design for the MIDAS system's response to these messages have been created as part of this research in order to demonstrate the viability of all levels of the proposed communication standards proposals.

Messages have been defined and agreed (between the author and the MIDAS design team) on messages which implement full level I, II and III communication to MIDAS. Part of this process has involved the author in designing a device methodology to demonstrate how MII and MIDAS are able to respond to these messages within the context of their current structure and functionality.

Details of this implementation for the MIDAS device are given below in appendix 2.

11.1.2. CSound as a system component

11.1.2.1 Current situation

CSound's instrument definitions specify networks of unit generators (analogous to the 'units' referred to throughout this thesis). These can include unit generators which receive input data via MIDI (see 3.2.1.5). The CSound synthesis engine (typically running on a fast general purpose CPU within a computer host) can then instantiate a new copy of this network specification (in CSound terminology 'make allocations of an instrument') in response to a single MIDI 'note on' message. This conforms to the concepts of level II communication.

There are, however, no messages to enable an *external* control software package to set up these unit generator network specifications as in level II. A text file has to be created in its entirety, with all the unit generators and their connections specified.

11.1.2.2 Future developments

CSound could feasibly be developed to enable conformance to the proposed standards, and thus facilitate its use as a synthesis device by a variety of higher-level composition software subsystems. This would require additional features, which could either be incorporated within the main CSound program itself, or added via an overlay interpreter program which creates and writes score and orchestra files, and then calls CSound, all in response to external control messages.

At level I, this interpreter could add real-time input unit generators (such as the existing MIDI ones). It would then connect these (in the Orchestra file) to inputs of the units which require real-time input control.

At level II, the interpreter could receive instrument definition messages, and build the corresponding CSound instrument definition, which would be stored as text in a CSound 'orchestra' file. The interpreter could then use this file in response to an instrument instantiation message from the control system. For time-stamped level II operation, the interpreter could load time-stamped data values (addressed to instrument inputs) into the appropriate field of a CSound score file.

A more complex interpreter program could provide the functionality and data storage needed to implement the level III communication proposals. At level III, each *score-super-event* structure (see 5.3.3.1) to be defined in the device would be created as a new 'score' file. However, as CSound has no inherent support for hierarchical event structures, each

time a request was received to nest such an event within a *higher-level* event (see 5.3.3.2), the entire event structure would have to be rewritten as a new score file.

If a user wanted to use CSound (ie write scores and orchestra text files) to act as an event specification system, another 'interpreter' application could translate these into a subset of the messages in the communication standard. This would be useful in order to be able to play existing compositions written in CSound format within the wider world of connected systems here proposed.

11.1.3. CHANT as a system component

In CHANT (described in 3.2.1.6) a user writes a Lisp control file, or uses the Patchwork GUI system (see in 3.2.1.7) to specify synthesis processes and organise them in time.

For the CHANT system to be able to act as a device within the proposed system structure, it could incorporate an input reader process or program, which receives incoming time-stamped messages and writes a Lisp text file. It could then compile this file, on receipt of an appropriate message analogous to the MIDAS level I 'run' command.

A more complex 'front end' program could provide the functionality and data storage needed to implement the level II and III communication proposals. At level III, each *score-super-event* structure to be defined in the device would need to be created as a new Lisp file, in a similar way to CSound.

In a similar way to CSound, a Lisp file interpreter could, if desired, interpret existing files, As CHANT is based within a Lisp environment, and Lisp is a conventional general purpose language, there is more potential than for CSound to expand the capabilities of CHANT to use all the messages. Again, the main use, if any, for such a set-up would probably be to perform existing CHANT files on other devices.

11.1.4. Kyma as a system component

The Kyma system (described in 3.2.1.3) is primarily a programming environment containing editable software components which support the construction of music-oriented structures, and provide functions such as access to soundfiles, MIDI i/o and custom DSP-based hardware (Capybara).

The whole Kyma/Capybara system could thus be easily extended and adapted to incorporate functionality enabling it to respond to the proposed communication standards, and thus fit the proposed wider definition of 'device'. New Smalltalk objects and methods could be incorporated within the existing software structure of Kyma to receive and translate messages, either for real-time or time-stamped control, building up or recalling stored Sounds (see 3.2.1.3) and then performing them.

In addition, as a general programming environment, Kyma could be used to implement the recommended functionality for event specification sub-systems (described in 4.4), and send messages to a variety of devices. A more detailed discussion is given in 11.3.4.

11.1.5. MAX as a system component

MAX is also primarily a graphical programming environment with software components which support the scheduling and processing of events.

Both versions of MAX (see 3.2.1.1) possess mechanisms to translate graphically specified units and their connections into function calls to instantiate and connect algorithms on an associated device. These can be such things as 'fof' modules (written in C) for CHANT (3.2.1.6), or DSP algorithms coded for the IRCAM ISPW Workstation (3.2.1.1) on an integrated synthesis engine (running on the same or associated CPUs).

For MAX to operate as a device within the proposed framework, additional objects could be written in C to enable MAX to receive and interpret the proposed standard messages from an external system, and then effect the appropriate control over the connected device.

In addition, as a programming system, Max could be used to implement the recommended communication functionality, acting as an event specification sub-system (see 4.4), and send messages to a variety of devices. However, to build a full user interface for such a system in Max, with visual score displays would be difficult. Max at present has only limited support for time-based score display (via two fixed types of graphic drawing screen), and does not naturally support *every* kind of programming language construct. A more detailed discussion is given in 11.3.4.

11.1.6. Commercial MIDI synthesisers as a system component

MIDI-based synthesisers *do* already operate to *some* extent in the role of the 'device' as defined in 4.3, ie they can receive external control messages and act on them. Hence, a discussion of the developments necessary for such synthesisers to fully implement each level of the proposed device functionality will be preceded by an analysis of their *present* degree of conformance to the proposal.

11.1.6.1 Level I implementation

Current situation

As stated above, MIDI synthesisers at present *partially* implement the proposed level I communication standard. Time-stamped messages are not catered for, nor are any of the system scheduling functions. However, MIDI devices do implement the real-time messages in level I to varying degrees:

- **Create unit**

The nearest equivalent to a unit instantiation message is a MIDI 'program change' message. This 'instantiates' a synthesiser 'patch' - a structure plus input parameter values. This will then exist in a device 'slot' - a memory area in which stored structures can be considered to exist, ready to run. There will be a fixed number of these 'slots' (typically 8 or 16). However, this usage will require that the *description* of the device in the control software is kept updated with any changes to the stored 'patch' in the synthesiser device.

This situation is unsatisfactory, as it requires a user to constantly monitor the state of the device and the control software to ensure that they remain consistent. An analogy would be

that a CSound user could not be sure that a request for an 'oscil' unit generator had not resulted in some *other* type of unit generator being created; its definition within the system having been changed by some other user without any notification.

E-Scape's design thus includes the facility to have a *series* of creation messages for a single unit structure. These can either be specified 'manually' when such a unit is described by a user, or built up automatically when such a 'device-level' structure is *built* by the user from lower-level modules. These, being lower than 'device-level', have no meaning as independent entities within a device, but can have creation messages and codes which are 'passed up' to the higher-level structure they are built into.

- **Connect unit to another unit**

Connection of synthesis units is not supported by any known MIDI device. This should not be confused with the provision in some devices of the ability to specify a matrix of connections between modules when constructing an instrument (eg. The Oberheim 'Matrix' series, or E-mu 'Proteus' series of synthesisers). These modules are not synthesis units as defined here, as they are *below* device-level, ie are components *within* a synthesis unit. To reiterate: a 'device-level' unit is one which can be independently stopped and started.

Connection of units to such things as device outputs or global 'effects' signal processors *is* often provided, although these are not presented as independent objects.

- **Disconnect unit from unit**

MIDI devices only implement this message implicitly, as 'connections' between units (as above) are restricted in scope: the 'connection' of a new output or effects module to a unit has the effect of disconnecting the previously connected units. This lack of explicitness contributes greatly to the difficulty of universal object-oriented description of MIDI based devices.

- **Delete unit**

MIDI devices only implement this *implicitly*, in a similar way to the disconnect messages above. Because units are instantiated in fixed 'slots' in devices, instantiating a new unit within a particular slot implicitly deletes the unit previously occupying that slot.

- **Send data value to unit**

MIDI synthesisers implement the *effect* of this message fully. For example they can start and stop a synthesis unit in response to a MIDI 'note on' and 'note off' message respectively. Other unit input parameter values can be updated using a variety of MIDI 'system exclusive', 'controller', 'pitchbend' or 'aftertouch' messages (or the 'pitch' or 'velocity' fields of a 'note on' or 'note off' message). However, the way a unit is addressed is via its *location* (rather than an individual id), either within the address map of the device, or a designated 'slot' with a MIDI channel assignment.

This makes it harder for event specification software to communicate with a unit, as the *location* the unit is in then determines the content of messages sent to it. This is one of the reasons the it is proposed that devices allow external access to units via an id number.

Future possibilities within MIDI

The MIDI standard already includes the provision of the ability to transmit 'cue lists' within the MIDI Time Code (MTC) specification. This consists of a list of times (expressed in MTC or SMPTE format) and MIDI events which can be loaded to a MIDI device which supports this feature. These events will then be output from the device when this time is reached. This feature has only been implemented on a handful of synthesis devices to date - eg the Akai S1100 and 3000 series of samplers.

Future extensions to the MTC specification and the corresponding device functionality could thus facilitate the provision of scheduling functions. Messages could be sent with time-stamps, stored in the device as a cue list, and then triggered. This would facilitate the performance of highly dense scores on a distributed MIDI-based system - avoiding the bandwidth limitations of MIDI, while incurring only a small time penalty.

Synthesis units and their inputs could be referred to by id, rather than the present system of indirect mapping via channel or system exclusive address map offsets.

This could also provide increased flexibility, with devices able to provide a variable number of independent units, depending on their complexity. Rather than being limited to a fixed number (typically 8, 9 or 16 at present). This would also obviate the present limitation of 16 channels. A MIDI channel was originally envisaged to be used to distinguish between different *devices* in a network. As devices have increased in their ability to support independent units, channels have been used to distinguish between these units. A single device can now use up all 16 available MIDI channels. Addressing units within a device by an id number would mean that a device could be controlled on a single 'channel'.

These changes would also require some kind of LAN solution (see 2.6.2) to facilitate real-time operation, given the overall MIDI bandwidth restriction and the typical control data density required.

11.1.6.2 Level II implementation by MIDI

Current situation

The general concept of downloading a structure to a device is present in MIDI via system exclusive data dumps, but these have no structure, and simply consist of a set of parameters. Inputs to an instrument are not defined, but simply exist as addresses of an entity on a single device slot, and/or a specified channel number.

E-Scape does not attempt to implement this kind of communication with current MIDI-based devices. A significant part of this functionality already exists in the form of commercially available generic MIDI editing/librarian software packages such as Dr. T 'X-OR' (Waugh 1989) which allows primitive user-specification of the structure of these data dumps. These use a language (eg Dr. T 'E-OR') to specify mapping to device memory, and wrappers for messages (within the MIDI specification). However, this software requires the user to operate at a level similar to a programming language - the task is too complex and the structures too diverse for any less flexible system to cope with.

Thus the task needs to be made simpler - which is exactly the aim of the proposed level II communication standard.

Future possibilities

Individual MIDI channel messages do not have enough fields to be usable, and such machinations as having an RPC message (see 2.5.2.1) for each field would require a large number of separate messages, and considerable complexity for the software at each end of the communication link. However, some kind of MIDI system message standard - such as that used for MIDI system non real-time messages (see 2.5.2.2) - could be used to send level II format messages. The device would also then require an additional level of address indirection to route incoming object ids to the appropriate object location or local id within the device.

11.1.6.3 Level III implementation by MIDI

The down-loading of time-stamped messages to devices via MIDI MTC 'cue' lists has been described in 11.1.6.1. Such lists of actions, however, only represent a simple one-level data structure, constructed solely from standard MIDI messages. However, MTC cue lists do illustrate that the downloading of event information to a device is feasible. Such stored data can then be used by the device, with minimal communication needed during actual performance.

An extension of the range of commands within the existing MTC format could conceivably be used to send level III message types, with the device storing and organising data in the kind of way described for MII (see appendix 2.3).

11.1.7. Conclusion

The structure of many existing systems would enable them to be adapted or added to support the proposed device functionality.

Computer-based systems acting as devices could feasibly have their software adapted, or incorporate an additional 'front end' software module. This would provide the proposed external control facilities, and then communicate with the existing synthesis component of the system, in the format normally used by that component. For example, a CSound 'font end' software module could write out a 'score' and 'orchestra' file, then load these files into the existing CSound system as normal. This has the advantage that an existing synthesis system remains unchanged, enabling its continued use as a stand alone system by composers, with no difference in its operation.

Hardware synthesis systems could in theory be redesigned, but a more realistic solution is a layer of software (which may exist in a separate 'translator' device), which interfaces between such devices and a control system which is using the new communication standards. Some of this device resource allocation functionality has been demonstrated within E-Scape (see 9.3.1).

11.2. Assessment of communication standards proposals

There are at present no synthesis systems or devices which fully implement the proposed communication standards, especially at level II and III.

An assessment of the viability of these proposed communication standards - testing that the proposed standards are useful and reasonable - has thus been carried out by designing a detailed protocol with message formats for communication with the University of York Music Technology Group's MIDAS system.

MIDAS (see 3.2.1.9) consists of a network of processors ('nodes') which communicate via low-level message packets. Unit generator processes (UGPs) can be linked to form synthesis structures, and communicate with each other via MIDAS packets sent between nodes.

A software module called MII (MIDAS Intermediate Interface) provides a layer of insulation between the MIDAS system architecture and a controlling application. MII provides an intelligent interface to the low-level packets and includes a database of the addresses and states of UGPs on the ring. An event specification system will typically control MIDAS via MII, and can then address all entities in MIDAS via a 'high-level' id (Anderson 1992a), and can request functionality such as storage and scheduling of event or control data.

This design of this protocol for communication via MII has also included the structure and functioning of the components of MIDAS (see appendix 1) in order for it to be able to respond to these messages according to the proposed standard. In consultation with the MIDAS development team, a viable design has been produced by the author.

As described in 4.3, the proposed communication standards require the kind of data structure and processing complexity on the part of the receiving synthesis device which may be more appropriately handled by a separate software process or subsystem which then communicates with the synthesis engine.

The MIDAS system has been designed in accordance with this structure, with the 'front end' MII software module receiving instruction and data messages from a high-level 'event specification' software system, such as E-Scape. As described in appendix 1, MII then performs non time-critical processing of these messages, using its database of information about node and UGP locations within MIDAS. It then sends function calls to the MIDAS network of processors, via message packets. The network may also send packets back to MII in order to report errors, or return the addresses of allocated UGPs. E-Scape (or other event specification system which is controlling MIDAS) is thus insulated from device details such as addresses, node ids, UGP-type ids, and data sizes.

It is planned by the MIDAS / MII development team to implement all three levels of the proposed communication standard, as illustrated in figure 115, below. Some of MII's activities are listed on the left.

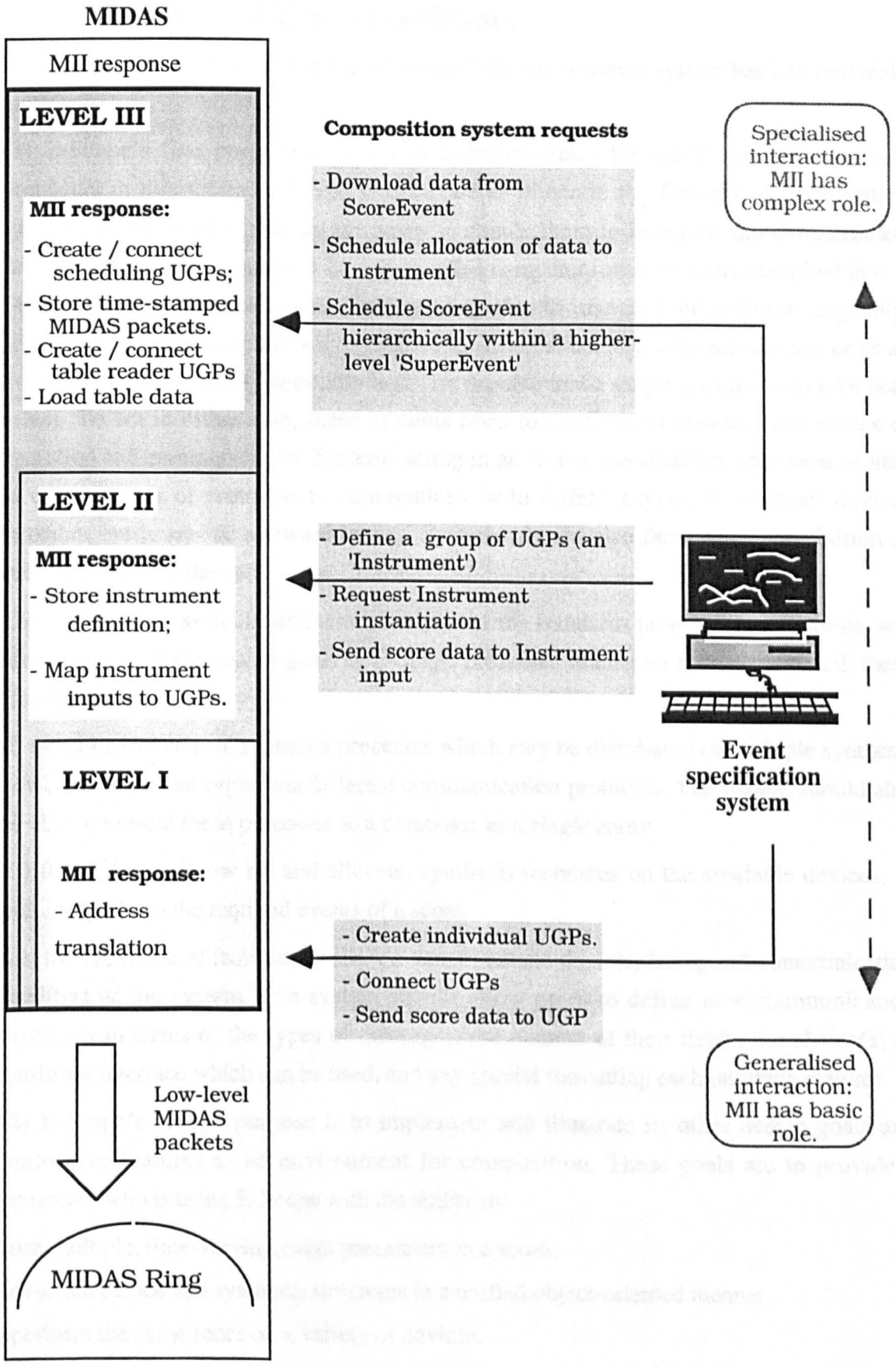


Fig. 115 Proposed implementation of all three levels of communication by MIDAS

The detailed design of an implementation for MIDAS of protocols which meet the proposed standard is presented in appendix 2.

11.3. Evaluation of E-Scape software

The design and development of the prototype E-Scape software system has had two main purposes:

(1) E-Scape's first purpose is to act as a demonstrator for aspects of the standards proposals in this thesis. E-Scape is designed to illustrate the features of composition software subsystems which are proposed to enable them to integrate and communicate within the proposed scenario of loosely-coupled computer music systems described in 4.1. This proposal involves all such systems being able to use each other interchangeably, acting as sub-systems within a larger network, acting either as a synthesis device, or as an event specification sub-system (although it is feasible that a single system could take both roles). To act in either role, these systems need to conform to standardised modes of operation and communication. Systems acting in an 'event specification' role must be able to use a variety of protocols to communicate with different types of synthesis devices (whether hardware- or software-based). Systems should also facilitate the definition of such protocols by the user.

These features of systems are included as part of the standards proposed in this thesis, and form a subset of the design goals of E-Scape presented in chapter 6. In more detail, these features are:

(i) the ability to control synthesis processes which may be distributed on multiple synthesis devices of different types, via different communication protocols. The system should also be able to present these processes to a composer as a *single* entity.

(ii) the ability to know of, and allocate, synthesis resources on the available devices, in order to perform the required events of a score.

(iii) the provision of facilities to allow a user to extend the interfacing and communication facilities of the system. The system should allow users to define new communication protocols in terms of the types of messages, the content of their fields, the choice(s) of hardware interface which can be used, and any special formatting each interface requires.

(2) E-Scape's second purpose is to implement and illustrate its other design goals and innovative features as an environment for composition. These goals are to provide a composer who is using E-Scape with the ability to:

- use multiple, time-varying event parameters in a score.
- describe device and synthesis structures in a unified object-oriented manner.
- perform the same score on a variety of devices.
- present synthesis structures on all types of device in a uniform object-oriented manner.
- present synthesis structures with perceptually meaningful scoring parameters.
- create intelligible scores which employ complex synthesis structures.
- organise and deal with score events within a hierarchical structure.
- distribute algorithmic processing structures between software system and devices.
- present algorithmic processes consistent within the normal score and instrument paradigm.

These features (detailed and justified in chapter 6), while argued as necessary in order to tackle the compositional difficulties discussed in chapter 3, are not necessarily proposed as *standard* features of all compositional systems - they are not necessarily the only way of approaching these problems. The diversity which is found within composition specification software as well as synthesis device subsystems is both necessary, inevitable and welcome.

An evaluation of E-Scape's effectiveness in fulfilling its design goals has been carried out by building a variety of example objects.

E-Scape DeviceType objects have been defined to describe different types of device. This has involved the definition of unit and structure description objects ('module types') as illustrated in 8.6, which have been stored within the DeviceType.

E-Scape Instrument objects have been created which use these structure descriptions. These Instruments have been designed to include various different PspProcessor objects which define scoring parameters and their conversion to inputs of synthesis units.

E-Scape Protocol objects have been created which define messages according to the 'MIDAS' (see appendix 2) and 'MIDI' (see 2.5) communication formats. The definition of module types (above) has included the use of MessageType objects from these Protocols.

E-Scape scores (SSE objects) have then been created using these Instruments. All data has been successfully processed, and synthesis resources in devices correctly allocated, to produce sound output (where devices are available). Communication to devices was tested in real-time only, as no devices yet implement the proposed standards for time-stamped communication, as described in chapter 5.

All these example objects have been created by specifying parameters in the way an E-Scape *user* would (rather than a programmer), as described in chapter 10.

11.3.1 Assessment of E-Scape flexibility in defining protocols

Some difficulty has been encountered in finding suitable protocols other than MIDI to use for evaluation purposes. Other synthesis systems either do not provide real time external control (*except* via MIDI!), or are in the commercial domain, and have not agreed to make their protocols available.

In consultation with the MIDAS design team, the author therefore developed a protocol with detailed message specifications for control of MIDAS by an external system communicating with it. This protocol design exercise also involved designing a scheme for the behaviour and internal *functioning* of MIDAS (and MII) in order to facilitate the appropriate *response* to these messages.

The MIDAS communication protocol has been developed for three reasons:

• To assess the *feasibility* of the proposed new communication standards for external control of devices (outlined in 4.2, with details in chapter 5). The aim is to demonstrate that the proposed communication standards are capable of facilitating instrument specification

and score performance in a real device. The protocol also acts as an exemplar for the communication recommendations.

• To illustrate the recommendations for device functionality and behaviour (outlined in 4.3) and assess their feasibility.

The low-level operation of MIDAS was already defined, and the author then designed methods of operation within MIDAS (and between it and its MII 'front end') which enabled MIDAS to implement the recommended device functionality. This has shown that the demands made on a device by the communication standards are realistic.

The MIDAS system operation also acts as an exemplar for the recommended device functionality (described in 4.3), which specifies, for example, that:

- the device should facilitate control by an external system;
- it should present itself to such an external system as having a dynamically allocated memory map (rather than having fixed slots for each unit);
- it should allow its units to be addressed by an id rather than by a 'slot' allocation.

• To enable the flexibility of E-Scape's structure and its protocol description facilities to be assessed. There was a need to use at least one protocol - preferably very different to each other - in order to facilitate the evaluation of E-Scape's flexibility using an additional (non-MIDI) protocol.

The messages defined within this 'MIDAS' protocol are designed to be sent from an external 'event specification' system (eg E-Scape) to MIDAS via its MII external control interface (see appendix 1.1)

These protocol designs for MIDAS are described in detail in appendix 2, but have not yet been able to be used in the realisation of sound output. This is because the component of MII which facilitates external access (see appendix 1.3.1) is not complete at the time of writing. However, the protocols have been fully worked out with the MIDAS / MII design team as a feasible proposition, with realistic device functionality specified for the MII / MIDAS system, as described in appendix 2.

11.3.1.1 User definition of protocols

Protocols can be defined by the user (as described in 10.6). This involves the user defining different message formats (as MessageType objects); and for each of these defining MessagePrototype objects which specify how the data for each message field is derived from score and instrument objects' parameters within the E-Scape system.

In the ideal case, this would be done without recourse to writing Smalltalk (or other) code, via a graphic specification. However, as graphic processing specification via a GUI is not central to the new aspects presented in this thesis, implementation of such a GUI is left for the future. Thus, the user is currently presented with a text editor on a small block of Smalltalk code, which presents itself with a pre-set format, and the requirement to write code for very basic mathematical operations which could be supplied in a handbook. The

user is nevertheless not required to understand the details of the Smalltalk language or programming environment.

11.3.1.2 E-Scape Protocol test 1: describing the MIDI standard

The MIDI protocol has been defined within E-Scape (as a Protocol[1] object) in the way a *user* would define it, as described in 10.6. Nothing about the MIDI standard is assumed by E-Scape, or built in to its system structures. User-defined MessageTypes have then been entered into this Protocol in the way described in 10.6, ie by supplying values to existing functions, *without* writing any Smalltalk methods. Examples of these MessageTypes are given in 8.8.3 and 8.8.5.

The user interface to do this via menus has not yet been developed, so these values have, at present, been entered as Smalltalk message arguments within an encapsulating 'test' method.

This E-Scape 'MIDI' Protocol has then been tested by constructing E-Scape Instruments which communicate via the Protocol to a variety of MIDI-controlled devices. Scores using these Instruments have been created and performed, resulting in MIDI messages being successfully output from E-Scape, and analysed.

11.3.1.3 E-Scape Protocol test 2: describing the MIDAS level I protocol

An E-Scape Protocol object which describes the MIDAS level I communication protocol (detailed in appendix 2) has been successfully defined within E-Scape. As described in 11.3.1.1, this Protocol has been entered as if by an E-Scape user, with no use of Smalltalk language. This Protocol and an example MessageType are shown in 8.8 and 8.8.4.

E-Scape Instruments which use this Protocol have then been created (see 11.3.3), and used in a score. MIDAS format messages have then been successfully created from this score, and output from E-Scape, where they have been captured and monitored.

11.3.1.4 Conclusion

The E-Scape software has been tested, by defining within it Protocol objects which describe the MIDI standard (see 2.5), and the MIDAS communication messages at level I (see 5.1). This definition has been achieved without Smalltalk programming. Thus, the protocol independence of E-Scape has been illustrated, although a *formal* proof of its ability to describe any communication protocol is beyond the scope of this project.

11.3.2. Assessment of E-Scape Instruments

E-Scape Instruments have been constructed using information supplied by a user (again without writing Smalltalk code), although this has at present been achieved by providing arguments to existing functions within an 'example' Smalltalk method which supplies these arguments, which are seen by the system as if entered from a user interface. This example method code will eventually be replaced by a user interface system, but the *functionality*

---

1 As throughout this thesis, a capitalised word indicates a E-Scape software object.

which allows a user to construct structures is present, and has been used successfully to build a variety of types of Instrument:

• Instruments using structures on several different MIDI-based synthesiser devices have been designed. Scores using these Instruments have been created, and their scoring parameters then successfully processed by these Instruments (via the Protocol) into a variety of MIDI format messages. These have been monitored, as described in 11.3.3.

• Instruments have also been built in the same manner for the MIDAS device, and specify and control its synthesis structures which are very different to those of a MIDI synthesiser, and using the completely different 'MIDAS' Protocol. Real-time messages in the agreed RS232 ASCII format (see appendix 1.3.1.2) can be sent out via the Macintosh 'modem' port to MII (the front end of MIDAS). However, at present MII does not respond to real-time messages, requiring time-stamped messages to be read from a file. Additional functionality for MII is imminent (within its development programme) which can interactively receive and act on messages. Further developments which allow it to receive data via MIDI are also planned. E-Scape already has an additional port type in place, which encapsulates MIDAS messages within custom MIDI system exclusive messages, in an agreed format.

• Finally Instruments which *combine* structures in MIDI-controlled synthesisers and the MIDAS device have been constructed. An E-Scape score which specifies and sends input data to *both* these devices from each single score parameter has also been created. This ability to control such 'mixed' Instruments - instantiating and controlling synthesis processes running on a set of *different* devices - which yet appear as a single unified structure to a composer, is a unique feature of E-Scape.

A practical difficulty with the Smalltalk application's MIDI interfacing on the Macintosh has rendered it impossible to physically send both MIDI and RS232 format data at the same time. This is because both types of data need to use the same physical 'modem' port on the Macintosh, and the low-level MIDI handling routines built into the Smalltalk virtual machine become inactive once modem port access is initiated for RS232 output (see 7.5.1.2). However, RS232 messages can be *monitored* in a display window within E-Scape (without being sent to the modem port) while MIDI data is simultaneously being sent from the port, and verified to be correct.

Alternatively if MIDI output is suspended, ASCII data has been successfully output from the Macintosh, and captured and verified on an RS232 terminal.

Eight examples of these Instruments are now presented, in order to illustrate a range of synthesis and processing structures on different devices.

11.3.2.1 Example 1a. 'D110 simple'

This Instrument (figure 116) has two PART units (within a single DCT) on a D110 DeviceType. It has three Psps: 'volume', 'pitch', and 'attack time'. All Psp values affect inputs of both PART units together, as they are joined by a common DCT input. The 'volume' and 'attack time' values are simply scaled and routed to the appropriate inputs of each PART unit, as shown. The 'pitch' values are converted into *three* values which are routed to three inputs of each PART unit as shown.

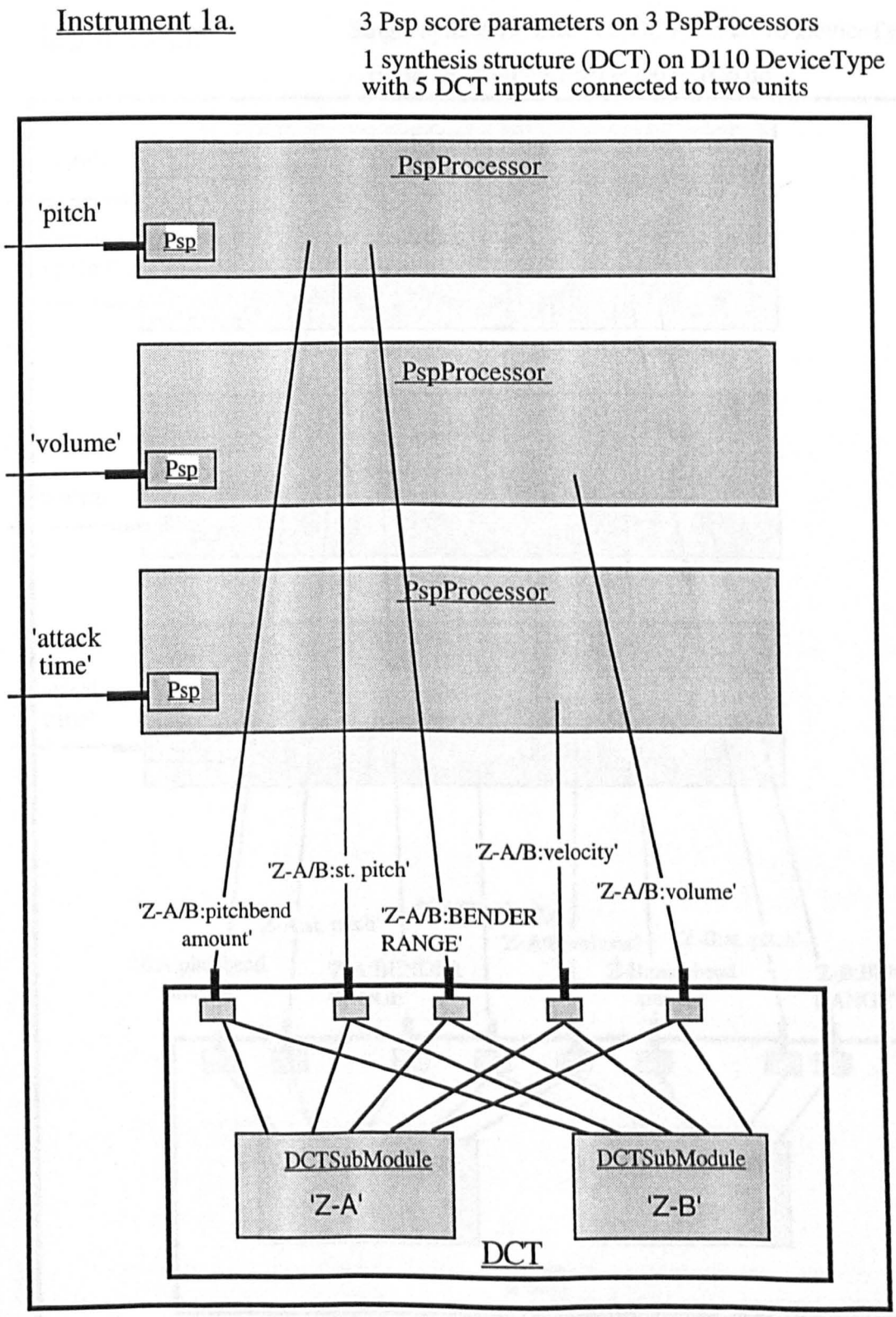


Fig. 116 Instrument 'D110 simple'

11.3.2.2 Example 1b - 'D110 intermediate'

This Instrument (figure 117) also has two PART units (named 'Z-A' and 'Z-B') on a D110 DeviceType. It has four Psps: 'pitch', 'detuning spread', 'volume' and 'attack time'.

Its structure is as above, but has an additional 'detuning spread' Psp, which allows the pitch of each PART unit to alter independently. The 'pitch' and 'detuning spread' Psps share a PspProcessor, as these two parameters interact with each other when determining unit input values.

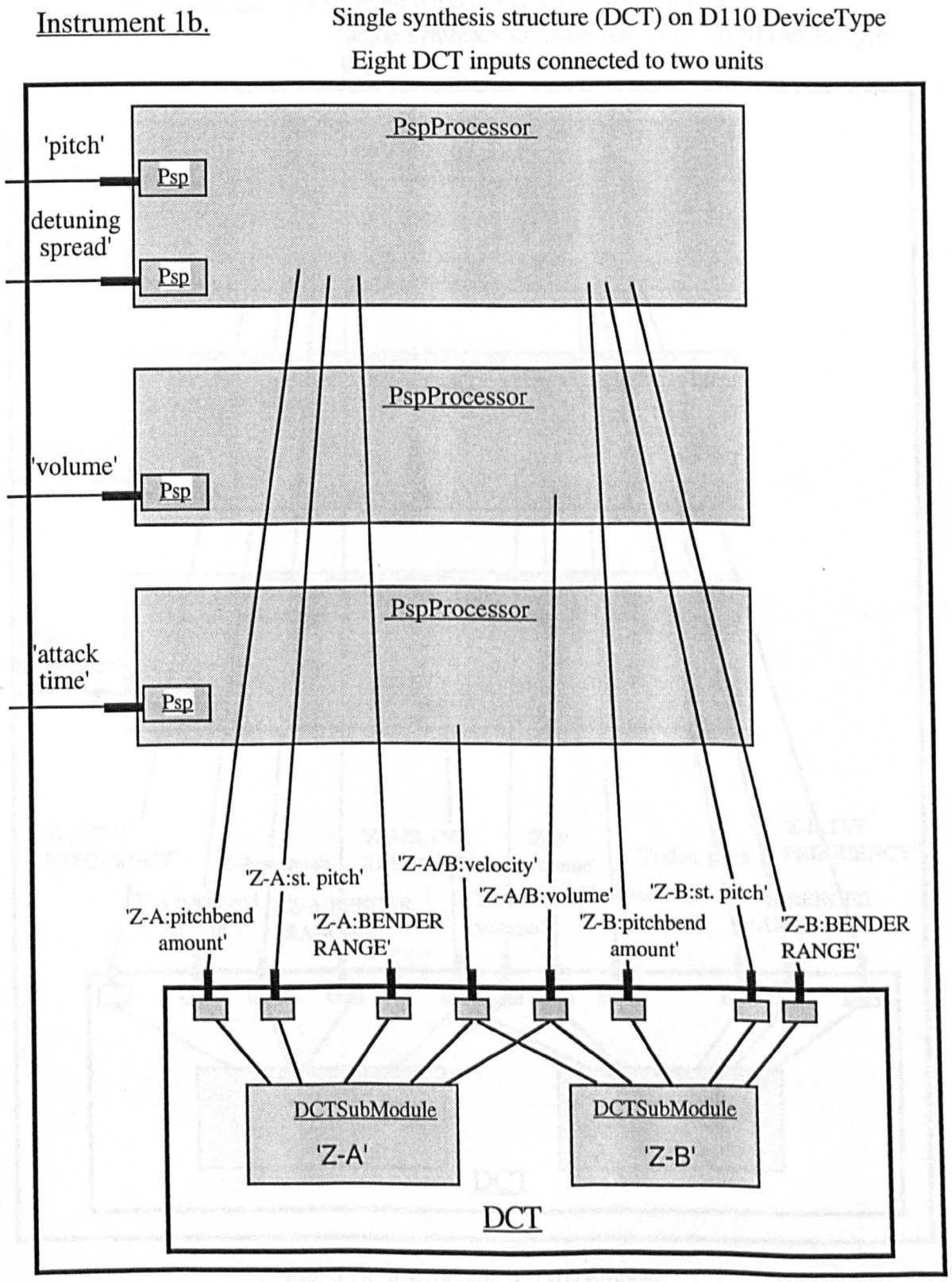


Fig. 117 Instrument 'D110 intermediate'

11.3.2.3 Example 1c - 'D110 complex'

This Instrument also has two PART units on a D110 DeviceType. It has four Psps: 'balance', 'pitch', 'detuning spread' and 'filter resonance'. Its structure is similar to example 1b, with the following differences:

- There is no 'attack time' Psp;
- Instead of 'volume' a 'balance' Psp alters each PART's volume inversely;
- A 'filter resonance' Psp alter this input on both PARTs.
- The filter frequency of both PARTs is altered as the 'pitch' varies.

Instrument 1c. 4 Psp score parameters on 3 PspProcessors
Single synthesis structure (DCT) on D110 DeviceType
Eleven DCT inputs connected to two units

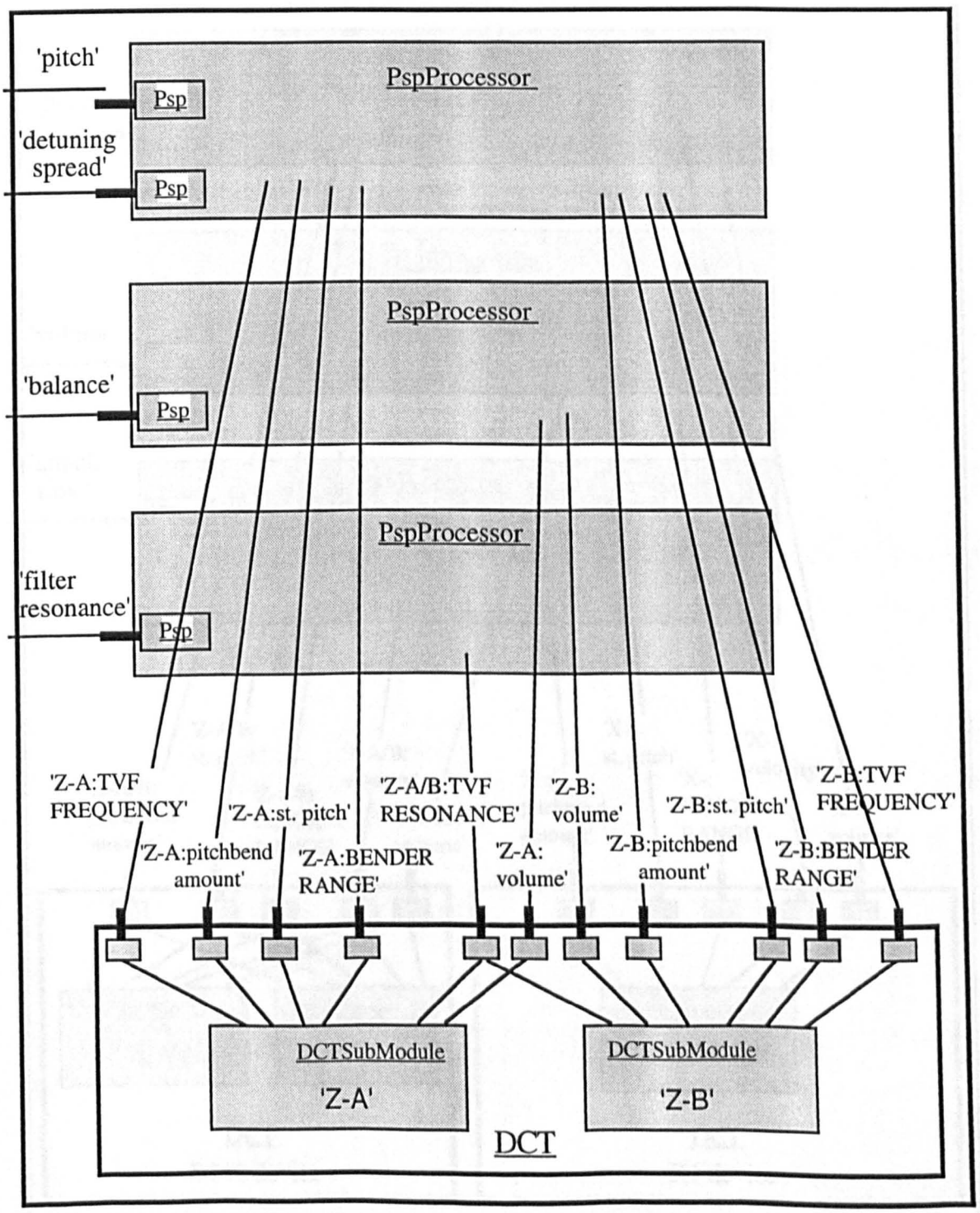


Fig. 118 Instrument 'D110 complex'

11.3.2.4 Example 2a - 'D110 and K1 simple'

This Instrument has structures (DCTs) on *two* kinds of device. It has two PART units on a 'D110' DeviceType (as in example 1), plus a single unit on a 'K1' DeviceType.

There are has three Psps: 'pitch', 'volume' and 'attack time' which are processed similarly to example 1a, but values are derived (as appropriate) and routed to *both* DCTs.

Instrument 2a. 3 Psp score parameters on 3 PspProcessors

One synthesis structure (DCT) on D110 DeviceType
Five DCT inputs connected to two units

One synthesis structure (DCT) on K1 DeviceType
Five DCT inputs connected to one unit

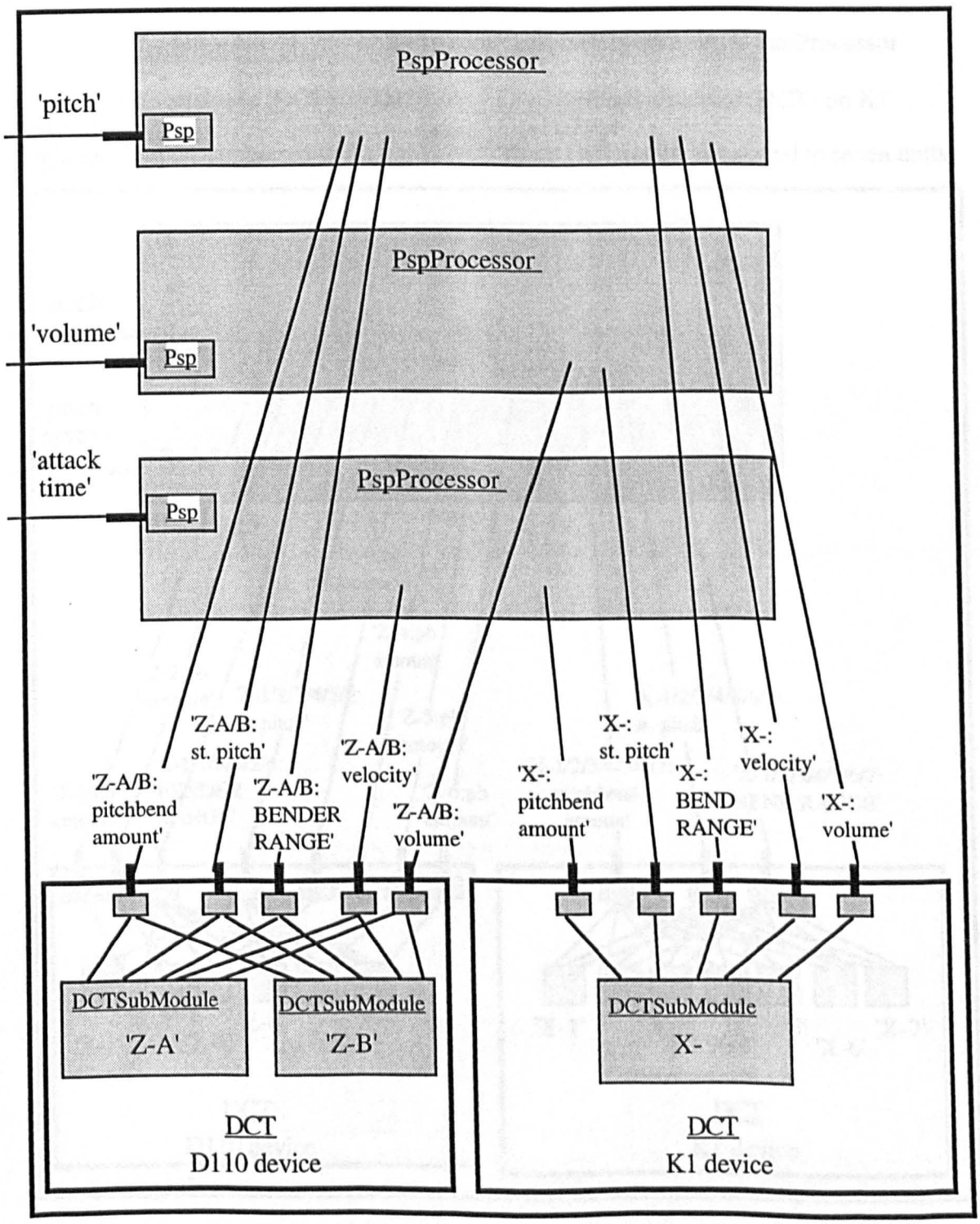


Fig. 119 Instrument 'D110 and K1 simple'

11.3.2.5 Example 2b - 'D110 and K1 complex'

This Instrument also has two structures (DCTs) on *two* kinds of device: It has *six* PART units on a D110 DeviceType, plus a *seven* units on a K1 DeviceType.

It still has two Psps: 'pitch' and 'pitch spread'. Score values for the 'pitch' Psp are processed for the K1 structure as in example 2a, but for the D110 structure, values for the 'pitchbend amount' of each unit are routed separately to each unit in the structure. This allows a degree of independent control of their pitches. Such independent values are derived by the PspProcessor, which adds or subtracts a different amount to the pitch of each unit, to a degree controlled by the 'pitch spread' Psp value.

Instrument 2b. 2 Psp score parameters on a single PspProcessor

One synthesis structure (DCT) on D110 DeviceType
Eight DCT inputs connected to six units

One synthesis structure (DCT) on K1 DeviceType
Three DCT inputs connected to seven units

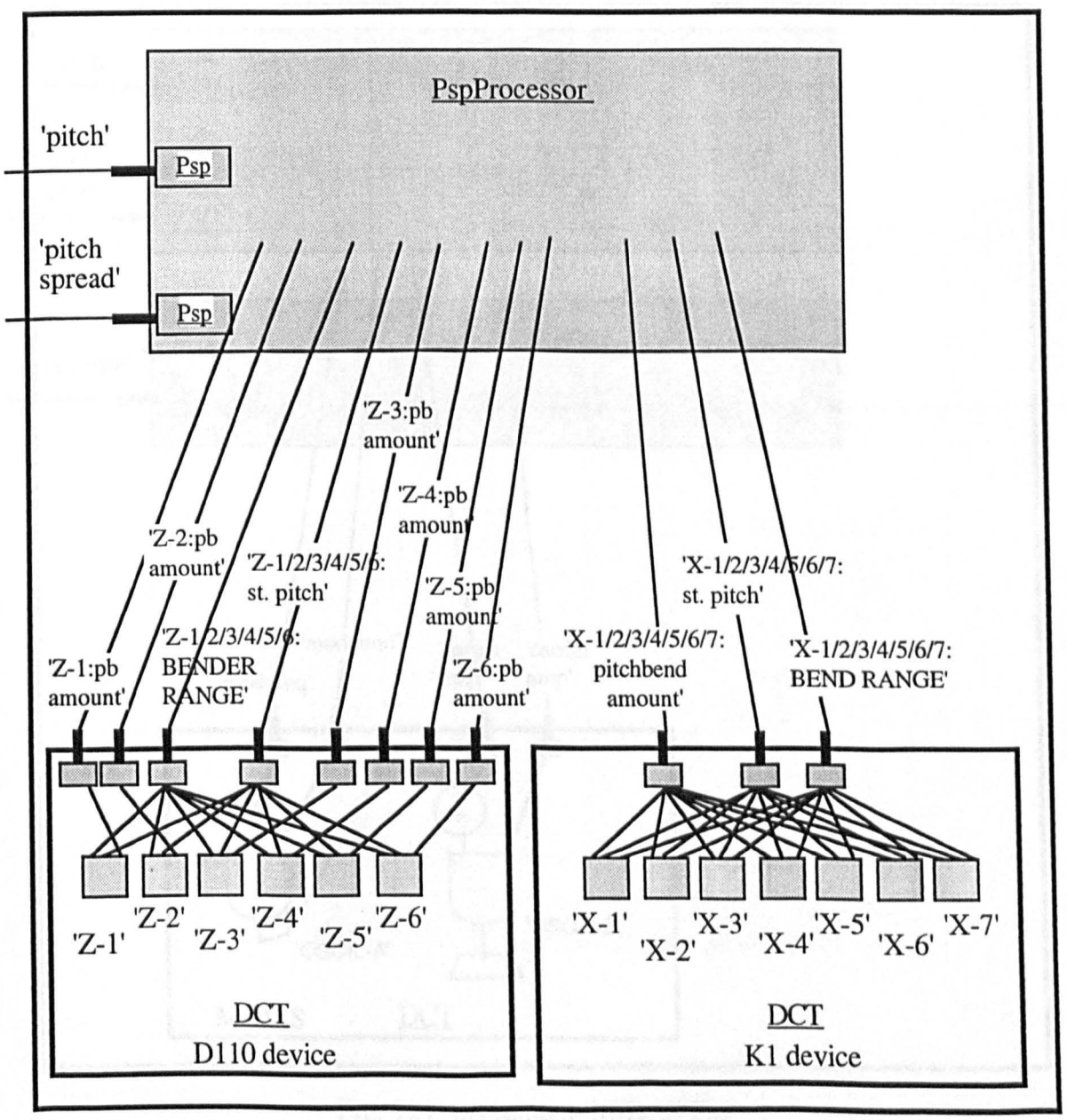


Fig. 120 Instrument 'D110 and K1 complex'

11.3.2.6 Example 3a - 'MIDAS FM'

This Instrument also has a single structure (DCT) on the MIDAS DeviceType. It has four UGP units which are connected to each other, in a simple FM configuration. Each 'OSCIL' unit is a table-reading oscillator, which defaults to a sine wave output if no wavetable is specified (as here). It has three Psps: 'pitch', 'mod index' and 'volume'.

Score values for 'volume' are scaled by a PspProcessor, then passed straight on to the structure input ('carrier amp').

Score values for the 'pitch' and 'mod index' Psps are processed by a PspProcessor, to produce values for the other three structure inputs as shown.

Instrument 3a. 3 Psp score parameters on 2 PspProcessors

Single synthesis structure (DCT) on MIDAS DeviceType
Four DCT inputs connected to four units

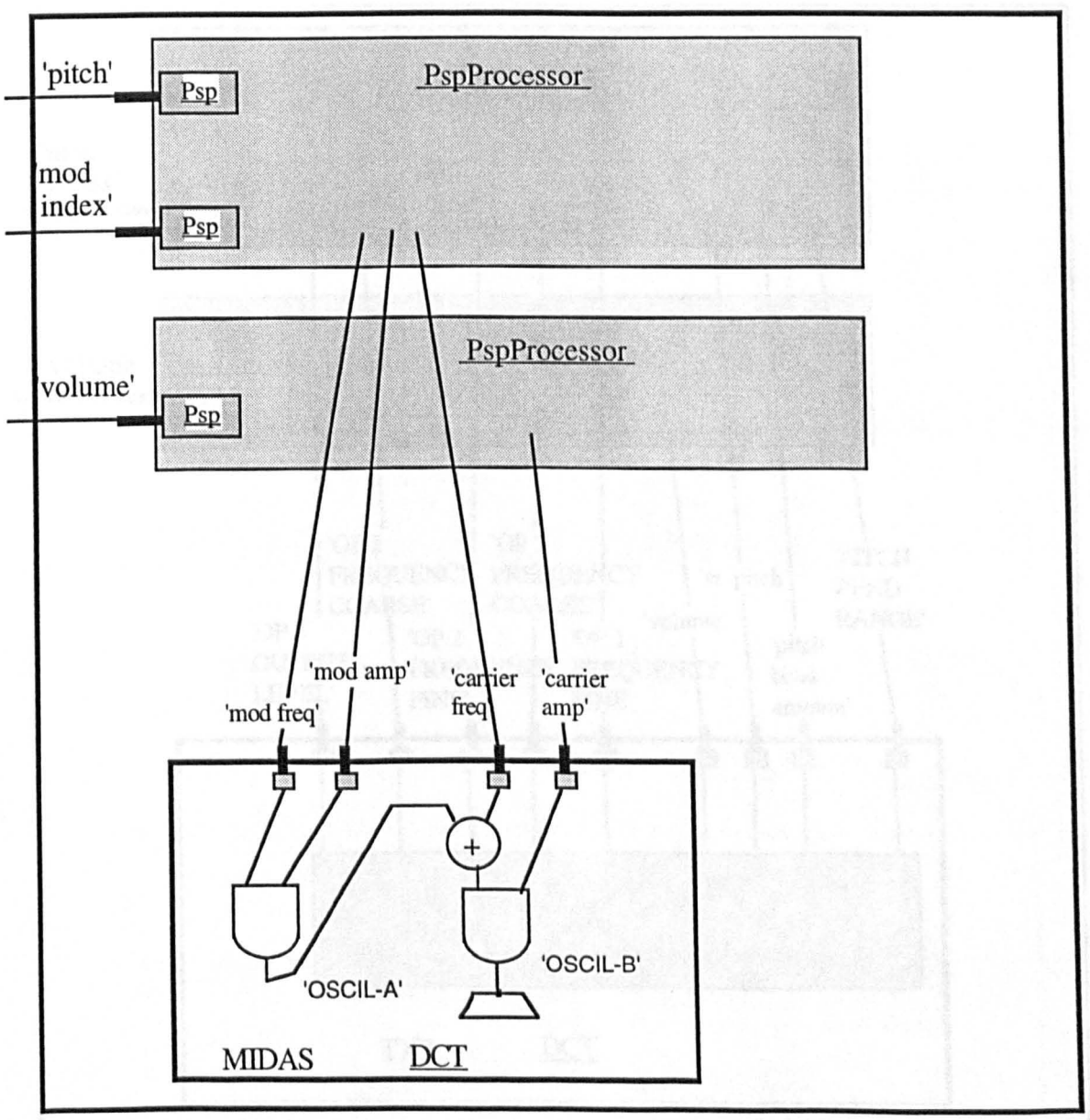


Fig. 121 Instrument 'MIDAS FM'

11.3.2.7 Example 3b - 'TX7 FM'

This Instrument also has a single structure (DCT) on the TX7 DeviceType. It has a single VOICE unit whose internal functioning facilitates up to 6 sine wave oscillators to be connected in a variety of FM configurations. The configuration used for this unit is of two oscillators in an FM configuration. It has the same three Psps: 'pitch', 'mod index' and 'volume' as example 3a.

The PspProcessor performs complex processing on the 'pitch' and 'mod index' Psps to produce values for eight of the unit inputs. These control the frequencies and amplitudes of the two oscillators. The aim is to emulate the sonic output of the Instrument in example 3a, using a completely different device.

Instrument 3b. 3 Psp score parameters on 2 PspProcessors

Single synthesis structure (DCT) on TX7 DeviceType
Nine DCT inputs connected to a single unit

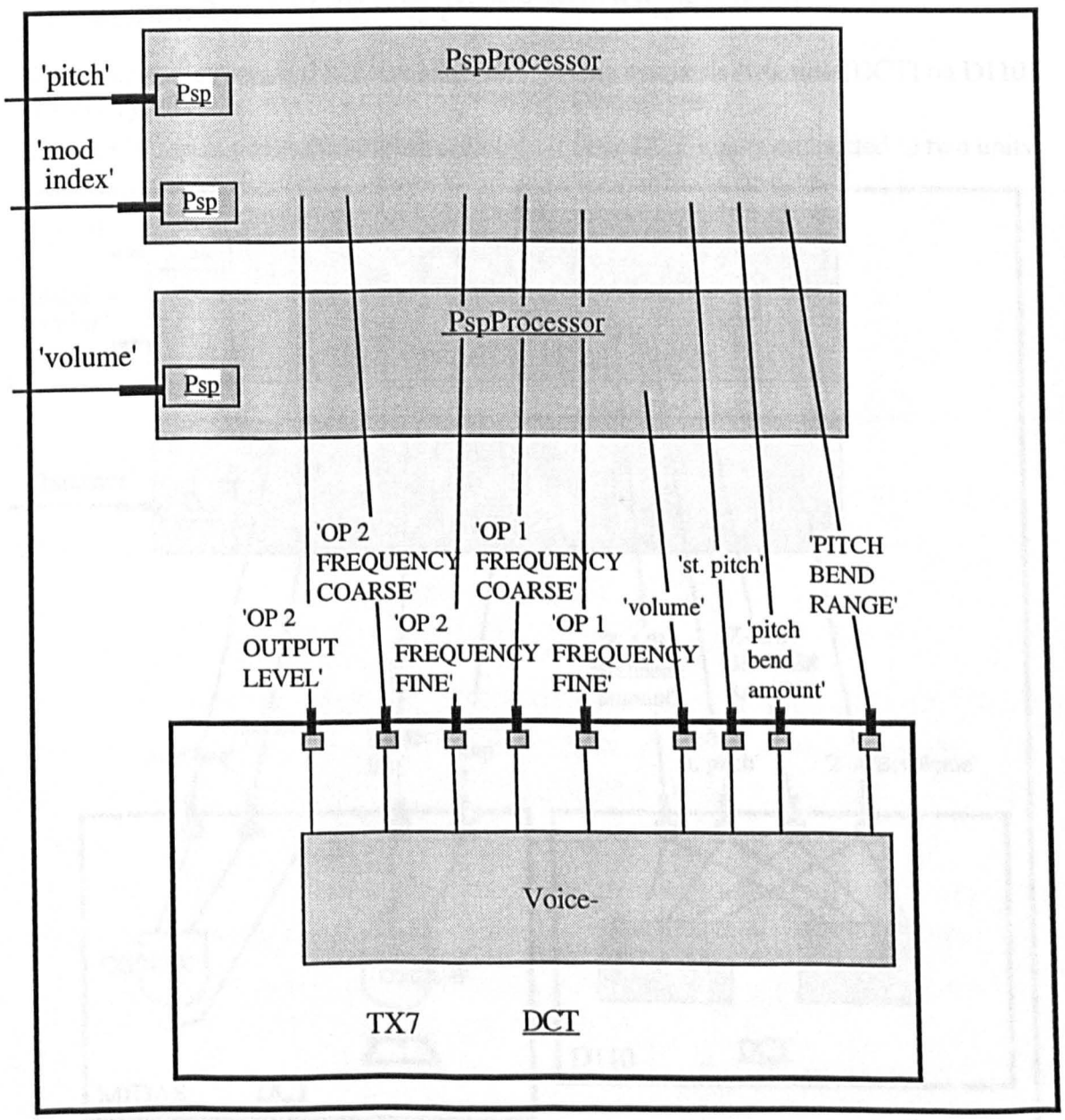


Fig. 122 Instrument 'TX7 FM'

11.3.2.8 Example 3c - 'MIDAS and D110'

This Instrument uses two structures (DCTs) on *two* completely different kinds of device which are controlled via different protocols on different ports, and employ different architectures. It uses a PART unit on a D110 DeviceType (as in example 2a), plus the same FM structure on the MIDAS DeviceType as in example 3a. It has three Psps: 'pitch', 'mod index' and 'balance'.

Score values for 'balance' are processed by a PspProcessor into inverse volume values for each DCT, then passed straight on to the structure input ('carrier amp'). Score values for the 'pitch' and 'mod index' Psps are processed by a PspProcessor, to produce values for the other three structure inputs of the MIDAS structure as in 3a, and also create the very different three input values to control the pitch of the D110 structure as in 2a. Note that this processing for the D110 structure will *not* involve the 'mod index' Psp, as the D110 structure is not FM based.

Instrument 3c. 3 Psp score parameters on 2 PspProcessors

One synthesis structure (DCT) on MIDAS DeviceType

Four DCT inputs connected to four units

One synthesis structure (DCT) on D110 DeviceType

Four DCT inputs connected to two units

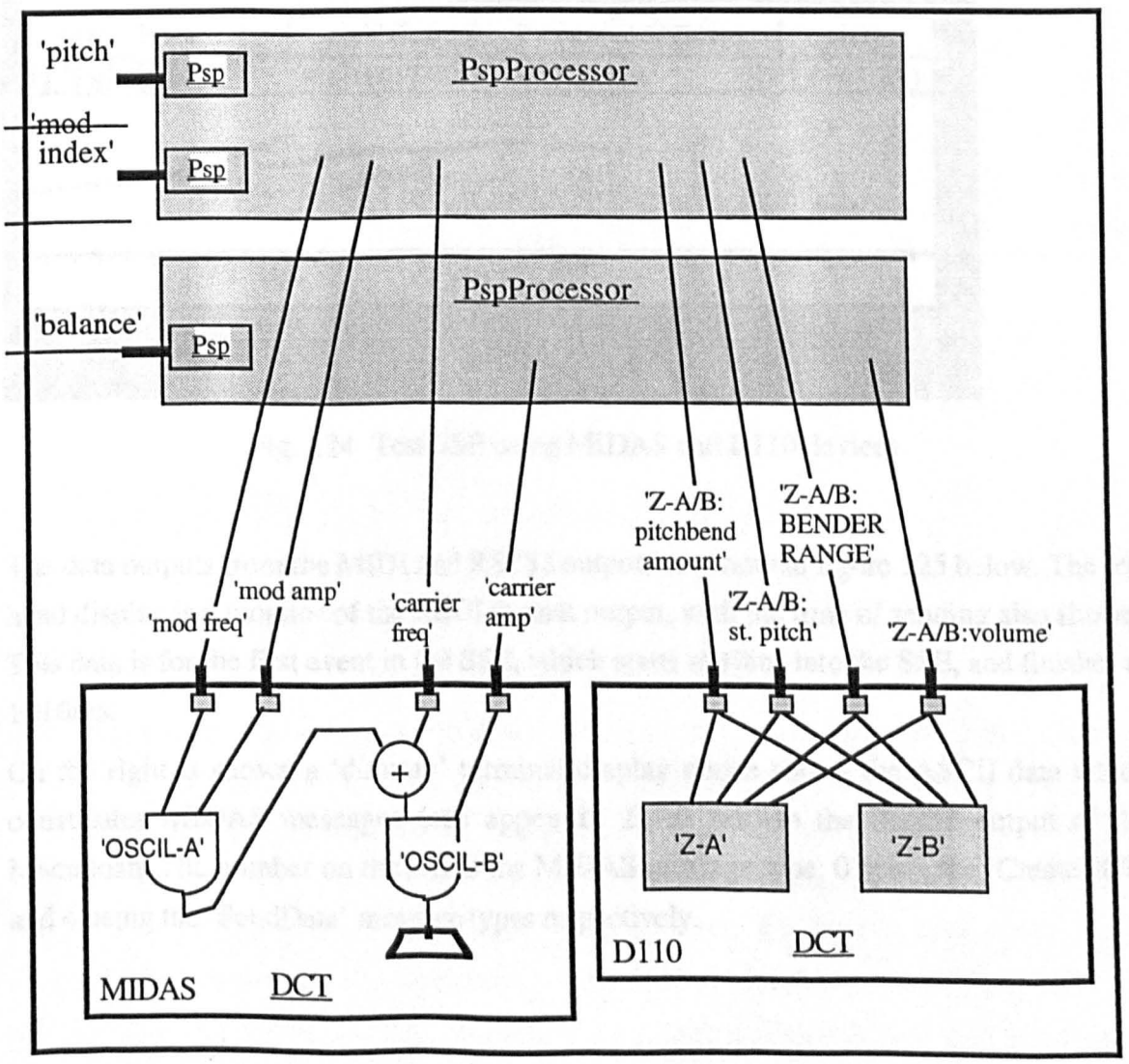


Fig. 123 Instrument 'MIDAS and D110'

Messages are sent to MIDAS from E-Scape on the Macintosh's 'modem' serial port, encoded as ASCII characters in the RS232 protocol. The use of the RS232 protocol to send real-time data is problematic, hence communication to MIDAS is envisaged using MIDI (encapsulating the MIDAS message within a MIDI system exclusive message format). When E-Scape has been converted to run on a UNIX platform, a SCSI or UNIX socket interface may be used. However, the use of the RS232 serial protocol was deliberately chosen to be able to test the concept that E-Scape espouses, ie that of being able to communicate with devices on a variety of user-definable protocols, whose specification includes one or more physical ports.

11.3.3. Testing the creation of scores in E-Scape

A variety of ScoreEvent and SSE (score) objects of varying complexity have been created using the above Instruments, via menu selection and graphical and textual data entry.

These SSEs have been successfully performed, with MIDAS messages monitored and analysed, and MIDI messages captured and tested for timing integrity. Figure 124 below shows a screen shot of an E-Scape SSE using the 'MIDAS and D110' Instrument.

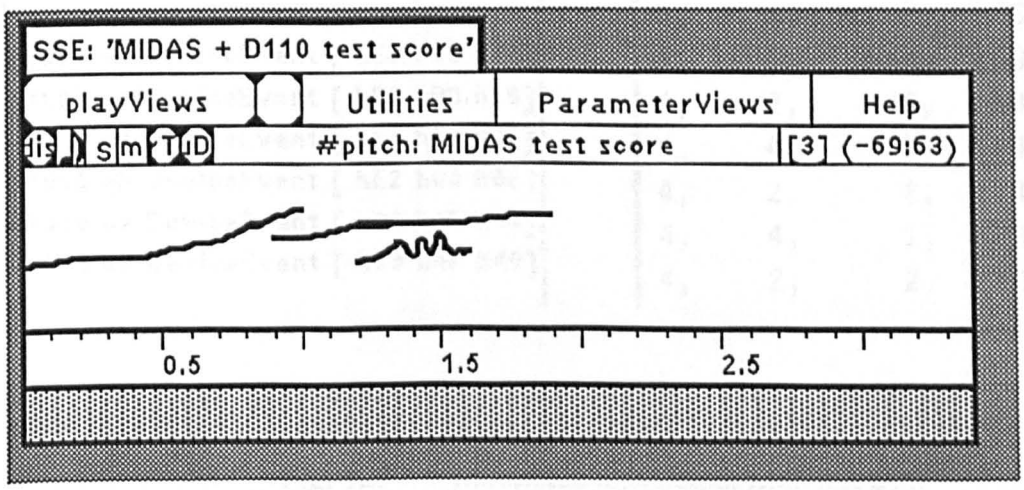


Fig. 124 Test SSE using MIDAS and D110 devices

The data outputs from the MIDI and RS232 outputs are show in figure 125 below. The left hand display is a monitor of the MIDI format output, with the time of sending also shown. This data is for the first event in the SSE, which starts at 10ms into the SSE, and finishes at 1010ms.

On the right is shown a 'dummy' terminal display which shows the ASCII data which constitutes MIDAS messages (see appendix 2), as sent to the RS232 output of the Macintosh. The number on the left is the MIDAS message type; 0 being the 'CreateUGP' and 4 being the 'SendData' message types respectively.

System Transcript

```
10 => DeviceEvent [ hB3 h65 h00]
10 => DeviceEvent [ hB3 h06 h04]
10 => DeviceEvent [ hB2 h65 h00]
10 => DeviceEvent [ hB2 h06 h04]

10 => DeviceEvent [ hE3 h00 h1A]
10 => DeviceEvent [ hE2 h00 h1A]
10 => DeviceEvent [ h93 h4F h40]
10 => DeviceEvent [ h92 h4F h00]
10 => DeviceEvent [ hB3 h07 h73]
10 => DeviceEvent [ hB2 h07 h73]
17 => DeviceEvent [ hE3 h00 h3A]
17 => DeviceEvent [ hE2 h00 h3A]
30 => DeviceEvent [ hE3 h00 h39]
30 => DeviceEvent [ hE2 h00 h39]
71 => DeviceEvent [ hE3 h00 h37]
71 => DeviceEvent [ hE2 h00 h37]
.
.
.
1007 => DeviceEvent [ hE3 h00 h10]
1007 => DeviceEvent [ hE2 h00 h10]
1010 => DeviceEvent [ hE3 h00 h0E]
1010 => DeviceEvent [ hE2 h00 h0E]
1010 => DeviceEvent [ h82 h4F h40]
1010 => DeviceEvent [ h83 h4F h40]
```

DummyTerminal

```
0,    1,    3,
0,    2,    2,
0,    3,    1,
0,    4,    1,
0,    5,    3,
0,    6,    2,
0,    7,    1,
0,    8,    1,
0,    9,    3,
0,    10,   2,
0,    11,   1,
0,    12,   1,
4,    4,    1,   148,
4,    2,    2,   148,
4,    3,    3,   2.0,
4,    4,    3,   2.0,
4,    3,    2,   3035.0,
4,    1,    2,   10500.0,
4,    4,    1,   160,
4,    2,    2,   160,
4,    4,    1,   174,
4,    2,    2,   174,
4,    4,    1,   188,
4,    2,    2,   188,
```

Fig. 125 Monitored data output from test SSE

Other examples of SSEs can be seen in sections 9.2, 9.3, and 10.2.

The relative timing of MIDI messages sent from E-Scape has been analysed using the 'Cubase' professional MIDI sequencing software. This has a maximum timing resolution of ~1.5 ms.

Subject to this resolution limitation, MIDI messages are shown to have been received from E-Scape within 1 ms of their expected (assigned) time, relative to each other. For example, an E-Scape ScoreEvent which starts 50 ms after the previous one finishes, is shown to send its 'start' (MIDI 'note on') messages 50(±1) ms after the previous event's 'stop' (MIDI 'note off') messages. This relative timing accuracy is independent of, and neglects, any *fixed* delay inherent in the Apple Macintosh MIDI ports and the communication time of MIDI messages themselves. Thus this Smalltalk implementation is shown to be adequate for control of external devices.

11.3.4 Comparison of E-Scape with existing computer music systems

This section examines whether any of E-Scape's facilities can be provided by some of the more flexible computer music systems described in chapter 3. Such systems might include Kyma (3.2.1.3), MAX (3.2.1.1), and HMSL (3.2.1.8).

For many of these systems, the answer to the question will take the form of: "yes they *could* provide some of E-Scape's facilities (but don't)". This statement becomes more intelligible if the nature of these systems is realised. These systems are primarily programming environments which provide tools and facilities at various levels to support the construction of applications and music-oriented structures. Such facilities include programming tools (eg interface builders, graphic icons etc), and editable software objects or components which can be adapted or incorporated within users' applications. These provide functionality such as access to soundfiles, MIDI i/o, control of processes on custom DSP-based hardware devices, or access to low-level sound-generating software routines.

> "A Kyma score is not a data file, but a Smalltalk-80 program.
>
> ... Kyma is primarily a language for sound synthesis and processing; it also provides basic ... tools for graphic editing of digital recordings".
>
> (Symbolic-Sound-Corporation 1992).

> "Max is best viewed as a graphical shell with a scheduler, as Miller [Puckette, Max's author] suggests". (Jaffe 1993).

These systems thus operate primarily as programming languages, which are provided in an integrated environment with high-level support for building applications or structures with musical features. Hence, such systems can be used for a wide variety of purposes.

Kyma especially, if it had been available, would have been ideal as a development environment for E-Scape. It provides all the programming facilities and advantages of Smalltalk, plus many additional application and user interface components (objects) which facilitate the building of hierarchical data structures and data processing networks, as well as control of the custom 'Capybara' DSP-based synthesis device.

Many existing systems support the creation and output of MIDI data, as well as control (via their own 'custom' function calls and protocols) of a dedicated synthesis device (whether DSP-based hardware, or software synthesis routines). Data processing structures could therefore be implemented in such systems which could effect control of MIDI-based and 'custom' devices simultaneously.

A user of such a system (who is effectively a programmer) could thus use it to *implement* some of E-Scape's structures, such as the data processing facilities which are in the PspProcessors on an E-Scape Instrument. Some of these systems can thus provide facilities which enable some or all of E-Scape's features to be implemented by a programmer, simply by virtue of their being programming languages. However, this is not the same as saying that they *provide* E-Scape's functionality as they stand.

Kyma, for example, is seen as a usable *application* (by a non-programming user) rather than as a programming environment which provides high-level facilities and a user interface for sound processing and scheduling operations.

As they stand, however, these systems do not have the *structure* which E-Scape provides to a non-programming user for composition with synthesis structures on different devices, without *needing* to be involved in programming or building such structures or processing. They do not implement to idea of integrated control of such distributed structures in a transparent manner. There is also no mechanism to allow a high-level MAX *user* to specify how these messages are formulated for different devices, ie no ability for a user to define new communication protocols, without employing low-level programming.

E-Scape thus exists at a higher level to these types of system; they are flexible *programming and working environments* with music and sound-oriented software tools, while E-Scape is a specific *application* (albeit a very flexible one, which can provide programming facilities if desired). E-Scape approaches the provision of composition facilities in a *particular* way - its provision of distributed Instruments with a transparent scoring interface is a concept, which could actually be implemented within a variety of programming environments.

Although they all have their individual strengths and approaches, existing systems quite rightly attempt to provide a generality of approach in order to support a wide range of compositional activities and concepts. If the proposed inter-communication standards were in place, then there would be more of a place for systems which concentrate on a particular specialised approach, rather than attempting to be general. E-Scape (neglecting its user-expandable aspects) would thus be one such specialised system in this scenario.

11.3.5 Assessment of the description of device structures by E-Scape

The survey of existing MIDI-based devices carried out as part of the design process for E-Scape has (practically, and necessarily) not been able to be exhaustive, but it has resulted in a seemingly never-ending discovery of structural differences in each synthesiser model examined; there is a non-closed problem domain. The presentation and analysis of the structure of current synthesiser devices would be a thesis in itself, and be quickly outdated, and has therefore not been attempted in this project.

Unfortunately, many present generation devices have not been designed along object-oriented lines, and are most certainly *not* object-oriented in their structure or mode of control. Hence, this goal has only been partially met, and the E-Scape design has encountered considerable difficulties in matching this object-oriented description to the often arbitrary structural paradigms of commercial MIDI-controlled synthesisers. This has proved to be a difficult task, which threatened to absorb a significant proportion of research effort.

Many relatively complex structures have had to be included within E-Scape in order to cope with the definition of structures (often with seemingly arbitrary restrictions) within MIDI-based devices, and the formulation of messages to be sent to them. These aspects are most definitely *not* structured in an object-oriented way, and the design of the E-Scape structures

needed to describe these aspects included some rather convoluted structures and methods in order to enable their description and specification in a uniform and consistent object-oriented manner.

Thus, the diversity of device structure and operation discovered has led to the conclusion that to facilitate control and description of *every* feature of every device would require a system of such vast complexity as to be as complex as a programming language, or require structures and methods so complex as to be unusable in the practical composer-centred environment that the system is designed around. Any simpler system will not be able to guarantee catering for any new device produced.

Thus, this aspect of E-Scape has not been pursued to this deep level, and has been replaced by the proposals for standards of device behaviour under external control outlined in chapter 4.

However, these difficulties did result in the formulation of the recommendations for future system organisation and device design specifications outlined in chapter 4.

- **Devices with a 'unit' structural paradigm**

In the academic computer music research world, a CSound 'unit generator' structure has been adopted in recent years as the de facto underlying model for the implementation of synthesis methods. Devices which conform to this 'unit' model, have primitive processing algorithms or units which can be created and linked, and have data sent to their inputs.

E-Scape's object-oriented model of connected synthesis 'units' thus matches the structure of almost all computer music synthesis systems in use today, and has many other advantages (as documented in 7.3 above).

E-Scape can also allow a user to deal with structures conforming to this unit paradigm when using devices (mainly MIDI-based commercial synthesisers) which are *not* overtly object-oriented in their structure of operation.

For such devices, E-Scape's structure allows the user to describe and specify synthesis or signal processing structures (either at the level of the lowest unit 'device-level' entities, or at a higher level). It also allows a knowledgeable user to define a protocol consisting of message types to instantiate and connect 'device-level' entities, and to send data to their inputs.

- **Devices with no 'unit' structural paradigm**

For devices with other structural paradigms, (mainly commercial MIDI-controlled synthesisers) E-Scape's design attempts to provide the facility for a user to specify new device type specifications and define new protocols for communication with them. To do this (and for other reasons described in 6.1 above) E-Scape attempts to describe all such devices using the same object-oriented 'unit' model.

Many of the more recent MIDI-controlled commercial synthesiser devices are 'multi-timbral'. This means that such a device can be set up in a mode (typically named 'multi', 'combi' or 'edit') which allows it to be treated as a number of relatively independent 'slots' - locations in the device which have an address offset and channel assignment, and within

which synthesis entities can be operated. E-Scape has then been able to use these entities (which may be primitive units, or more complex structures) as 'raw' resources to enable the building of more complex structures (albeit within the limitations and often arbitrary restrictions which exist within such devices). There would be a problem using a device in this way if independent settings were *not* available when used in 'multi mode'. For example, if there is only a *global* 'pitch bend sensitivity' setting on a commercial MIDI-controlled synthesiser. At all times one is fighting the fact that such devices most definitely do *not* present themselves or their functionality in an object-oriented 'unit' fashion.

The difficulties encountered in this aspect of device structure need to be placed in context:

As stated above, this thesis presents a set of recommendations for the structuring of electroacoustic composition systems which accommodate different kinds of synthesis device, along with communication protocol and device behaviour standards. The implementation of E-Scape is presented partly as a demonstrator for the proposed structural standard.

Pragmatically, however, it would be desirable for such systems to be able to contain MIDI-controlled devices and 'custom' system devices (constructed from DSP components for instance) which do *not* fully implement the device behaviour recommendation, and which are controlled by other 'non-standard' protocols.

Clearly, *existing* systems (eg current MIDI devices) will not have been designed to conform to the proposed recommendations for device behaviour. However, despite its non-suitability as a communication protocol fitting the recommended device communication standards, MIDI nevertheless provides a good *test case* for the *flexibility* of E-Scape as a high-level event specification (composition) system which has been designed to facilitate control of a variety of devices, using a variety of protocols.

Hence, the way in which it has proved possible to include these non-conforming devices has provided a good test of the flexibility of the design of E-Scape. A user is able to define complex structures within E-Scape in an object-oriented way from lower-level modules, which themselves may be complex. These structures can then be specified to, and instantiated in, a device, while facilitating the allocation of the device's resources in a flexible way.

E-Scape has had success with extending both the consistency of description and control flexibility of synthesis devices (see examples in 11.4). However, as described above, the complexities experienced with describing MIDI-controlled commercial synthesiser devices in such a way, coupled with the lack of any standard external communication provision by custom synthesis systems, has resulted in a perceived need for new standards for synthesis device behaviour (whatever their internal structure) and communication provision.

11.4 Conclusion

The system organisation, functionality and inter-communication proposals are capable of being implemented (or in the case of MIDAS, *have* been implemented) by feasible modifications to existing computer music systems.

E-Scape demonstrates the functionality required in order for a system to be incorporated as a sub-system within the proposed system organisation, and it functions correctly within its design specification (subject to the caveats discussed in 11.3.5).

E-Scape Protocols and DeviceTypes have been successfully defined, Instruments have been built using these DeviceTypes, and scores (SSEs) have been created and performed using these Instruments.

12. Conclusions

This chapter first summarises the aims of this research project which divide into three main areas. It then presents a summary of the results achieved in each of these areas.

12.1. Summary of research aims

The initial aim of this project was first to examined the nature, structure and operation of computer-based composition and performance systems, in order to investigate and analyse the difficulties faced by the builders of such systems and the composers who use them.

The focus in this investigation was on the electro-acoustic aspects of music composition which are uniquely enabled by computers and synthesis systems. This work is presented in chapters 2 and 3.

Having analysed the problems experienced by current systems, the main aim was then to address these problems and propose some possible solutions. These proposed solutions are presented in chapter 4.

This work led to the formulation of three linked subordinate goals, which together approach a solution to the problems.

These sub goals were:-

1. The design of a *structure* for interrelating computer music systems so they can interact within a linked system. This includes a proposal for the functionality which systems should present to an external system.

2. The design of a set of *communication* protocols by which such interaction between systems can occur, to effect control of each other. Details of this design are presented in chapter 5.

3. The design and development of a prototype software composition system (named 'E-Scape') for the purpose of: (i) illustrating the features required by systems in order to participate in the proposed system organisation, and (ii) demonstrating other design features of composition software which can enable composers to overcome the difficulties encountered.

12.2. Summary of research results, and original contribution to knowledge

This project has resulted in the design of a proposed set of new standards for the structure, control and inter-communication of computer-based music composition and synthesis systems. These structure and communication standards, if adopted, would help to achieve the *integration* of computer music composition software and synthesis devices, both today and in the future. This will reduce system rigidity, obsolescence and specificity, with a resultant reduction in costs with regard both to equipment and software development time.

This will result in a composer being able to use the synthesis or compositional control aspects of existing systems as interchangeable components within a larger-scale, loosely-coupled system. Such a system can then be constructed and customised by composers to suit the kinds of activities they want to engage in, their experience of compositional methods and languages, and their synthesis requirements for performance.

The results can thus be divided into three areas:

1. Proposals for the organisation and functionality of computer music systems, namely that all systems should be able to operate in a *synthesis device* role in a larger-scale network of disparate systems. To do this they need to respond to external control, via standardised types of request, from another system acting in an 'event specification' role.

2. Standards proposals for communication protocols to facilitate this inter-system communication, ie the specific information required to be conveyed by messages between systems.

3. The design and implementation of a prototype composition software system, which acts both as a demonstrator for these proposals, and illustrates other features of composition systems which approach a solution to the problems encountered.

12.2.1. Computer music systems organisation proposals

Several problems which face computer music systems in general (see 3.3-3.5) have been tackled, the chief of these are briefly re-iterated here.

Many systems have 'event-specification' sub-systems which are tied to, or integrated with, a single sound generation subsystem, and may be limited in the synthesis methods available. This has resulted in a lack of flexibility to expand the capabilities of systems as new synthesis devices become available. These systems exist in isolation, dependent on the maintenance and adaptation by institutions of their software and hardware components. This larger scale problem looms over the development and introduction of any new specific computer-based music composition system: namely how to ensure that existing knowledge and systems are maximally utilised, and that the effort invested in gaining expertise and experience in a new system will not be wasted in years to come, as it too is forced to be scrapped. For example, Lippe comments on the IRCAM institution's problems in this area.

> "IRCAM cannot afford to continue porting its constantly growing number of compositions and research projects to new platforms. That is why, among our stated goals for the future, the development of a technology-independent system is important" (Lippe 1993).

Systems which access external sound generating facilities via a communication protocol will require new or updated software in order to cope with new or augmented protocols which may be introduced.

Systems cannot usually distribute a synthesis process over multiple types of device, or often, even over more than one device of the same type.

The specification of event parameters by a composer requires him/her to have a detailed knowledge of the internal workings of the synthesis processes (and their control inputs) which will realise that event.

These problems, and the others discussed in chapter 3, have resulted in a proposal that computer music systems should be able to be used interchangeably in a larger-scale network, yet not be forced to sacrifice the individual approaches and philosophies which lie behind their provision of compositional and sound structuring facilities.

In order to facilitate these proposals, both computer music systems (designed to provide integrated composition and synthesis facilities in a stand-alone manner) and synthesis devices (designed to be controlled from other systems) should be able to *act* as devices. They thus can take on the role of modules, sub-systems or nodes in a larger-scale distributed, loosely-coupled system. Thus, it is recommended that all synthesis devices, or systems which incorporate a synthesis component, should also be controllable from an external system in a manner which conform to the proposed communication standards (see 12.2.2). This will enable them to be utilised by an external system as a synthesis device resource,

This will enable a composer to build a composition and performance system out of components appropriate to that individual: a favourite score language, a favourite graphic instrument editor (if not provided in the same software package), and a selection of synthesis devices or systems of various ages - all running on a variety of platforms.

All such devices should be able to:

- be accessed in an object-oriented manner, with unit processes or (optionally) networks uniquely identified. This will facilitate the construction and control of complex structures in a simplified, modular fashion.
- respond to real-time *and* time-stamped control. This will enable processes of arbitrary complexity to be specified and controlled.
- store unit network definitions, data tables, hierarchical event structures, and score parameter patterns. This will enable large and/or complex scores to be performed by downloading data from a controlling event specification system.

This will mean that a device's synthesis processes can be accessed as units, which can be instantiated and controlled within user-definable structures by remotely generated messages.

This is not to say that *all* features of a device or system should be controllable from outside, or that it should not incorporate software for high-level control within itself. For systems which incorporate both synthesis and compositional control components, it is recommended that the *interface between the two components should be opened up to outside systems.* Although internal communication between a system's components is likely to involve low-level (custom) communication links, it should also be able to receive and interpret standardised kinds of messages from *external* control systems, which will have a defined effect on the device.

Various existing systems have been examined, and modifications suggested which would enable them to participate in this way in a larger community of inter-communicating systems.

12.2.2. Communication standards proposals

The aim of proposing these communication standards is to enable computer music systems to operate within a larger loosely-coupled system, and be able to exert control over each other.

Computer music systems should be able to act within a larger system structure in one or both of two roles: either as a 'device' sub-system (being controlled as a slave from another system), or as an 'event specification' sub-system (able to control another system).

A device should be able to receive and act on standardised kinds of messages which will have a defined effect on it. Thus there are not only a standard set of *types* of command, but a standardised *meaning* for them.

The aim has been to avoid narrow prescriptive standards, so as to facilitate the maximum diversity between systems' internal functioning, only requiring them to understand a common *meaning* each defined of message. Thus, the conformance of messages to a *particular* protocol is not necessarily being proposed; such a standard would be too restrictive and unwieldy, and restrict development of new devices and synthesis methods. Messages may well be conveyed on one or more of a variety of hardware interface standards, and the fields may be in various orders, and contain data of different sizes. Composition systems - such as E-Scape - which aim to control such systems as 'devices' (as defined in 4.3), will thus still require a degree of flexibility in the formulation of messages to be sent to devices. As stated in 12.1, one of the goals of the E-Scape system was to demonstrate the kind of functionality that will be required for computer music systems which aim to integrate into the system inter-communication scenario proposed in chapter 4. What then *is* proposed is that the general features and organisational model of devices - as seen by external users - should be standardised.

For example, devices should be able to respond to requests to have a synthesis unit instantiated (this could be likened to a MIDI program change command, recalling a structure

or object from memory and installing it in current memory area ready for running). In addition this message should contain a 'user id' which can be used for future references to the unit, and an 'instrument id', allowing units to be grouped. Messages to connect units should also be understood, if such flexible structuring is implemented on a device. Other commands can either request an action (eg 'start') from all units with a particular instrument id, or send data to particular inputs of particular units. This provides a flexibility not provided at present from MIDI-controlled synthesisers, and as previously discussed, a standard way of interfacing custom systems to any control software system which can implement the appropriate message format.

Devices should be controllable both in real-time (if their processing speed allows), as well via time-stamped communication from an external system. The ability to load score structures and data into a device should also be provided. Three levels of communication are recommended. Some elements of the first two are implemented on the majority of MIDI-controlled commercial synthesisers; the third level only so far in the design of the MIDAS system:

12.2.2.1 Level I communication

Level I messages allow control software to directly create, and connect synthesis units in a device, to start and stop them, and to send data to their inputs. Messages can be sent to be executed immediately, or with a time stamp.

Before or during score playback, the control software can then send messages to:

- instantiate synthesis units;
- (optionally) connect them into structures if the device supports such connections;.
- specify a control parameter value for an input of a synthesis unit. There should be control parameters to individually start and stop a unit running in the device.

All messages have a time-stamped variant. Some devices may *only* be able to respond to real-time messages (acting on it immediately), while others may only be able to receive time-stamped messages, building them into a schedule or file. This schedule would then be started in response to the 'run device' system message.

12.2.2.2 Level II communication

Level II messages allow control software to define *instrument-templates* within the device. Each is a specification of networks of connected synthesis units. These have defined inputs which connect to one or more of these units.

The controlling 'event specification' (composition) software can then request the creation of *instances* of such an *instrument-template*, and send score data to its inputs, rather than to individual units within it.

12.2.2.3 Level III communication

Level III messages will allow control software to download and schedule hierarchical score data to a device in advance of performance. This performance can then be initiated or

controlled by relatively few messages to the device. Thus complex score structures can be performed without encountering communication bandwidth limitations.

12.2.2.4 Implementation design for the MIDAS system

A design has been produced for the implementation of these standards proposals by the MIDAS system, which can act as a device (as defined in this thesis) and respond to messages from an external control system. This work has involved a detailed specification of message types, and a suggested design for the MIDAS system's organisation and functioning necessary to carry out the proposed device functionality and behaviour. MIDAS is still under development, hence some of its low-level details may alter, but the design has demonstrated how the MIDAS 'device' can legitimately implement this functionality within its existing structures.

12.2.3. The E-Scape software system

The proposals for system organisation presented in this thesis have been illustrated by the E-Scape event specification (composition) software. E-Scape implements the recommended system features, and has been able to integrate and control a range of devices with its flexible communication facilities.

The design goals of for E-Scape have largely been achieved in a prototype software implementation. These design goals can be summarised in two categories:

12.2.3.1 E-Scape design goals (1)

Recommendations have been presented in 4.4.1 for the functionality which 'event specification' subsystems need to provide in order for them to be able to function and communicate effectively within the proposed system scenario. One of the main aims of the E-Scape prototype is to demonstrate these features, which are:

- Protocol-independence, and the facilitation of protocol definition by a user
  - the provision of facilities to allow a user to extend the interfacing and communication facilities of the system. The system should allow users to define new communication protocols in terms of the types of messages, the content of their fields, the choice(s) of hardware interface which can be used, and any special formatting each interface requires.
- Support for distributed instruments across devices and device *types*
  - the ability to control synthesis processes which may be distributed on multiple synthesis devices of different types, via different communication protocols. The system should also be able to present these processes to a composer as a *single* entity.
- Provision of allocation of device synthesis resources with well-defined score performance
  - the ability to know of, and allocate, synthesis resources on the devices available , in order to perform the required events of a score, or else inform the composer of events which are unable to be realised.

12.2.3.2 E-Scape design goals (2)

Other design goals have been set for E-Scape, in order to address some of the difficulties of current systems outlined in chapter 3. These features are not necessarily proposed as *standards*, although the author would certainly contend they are desirable. However, as argued elsewhere in this thesis, the diversity of approach employed by computer music systems - more particularly by their 'event specification' subsystems - is both necessary and desirable. No one system can realistically facilitate every way of working, and every manner of conceptualising and creating musical and sonic structures.

These design goals are:

- The ability to use multiple, time-varying event parameters in a score.
- The ability to present synthesis structures to a composer with perceptually meaningful scoring parameters.
- The ability to describe device and synthesis structures in a unified object-oriented manner.
- The ability to perform the same score on a variety of devices.
- The ability to present and describe synthesis structures on different types of device in a uniform object-oriented manner.
- The ability to create intelligible scores which employ complex synthesis structures.
- The ability to organise and deal with score events within a hierarchical structure.
- The ability to distribute algorithmic processing structures between software system and devices.
- The ability to define and use algorithmic processes in a manner consistent with the normal score and instrument paradigm.

These aims were presented in detail in chapter 6.

To reiterate: although the design goals (1) above *are* recommended as standard features of event specification subsystems (to enable computer music systems to integrate into the proposed system scenario), the other features presented as E-Scape design goals (2) are *not* necessarily proposed as *standard.*

The E-Scape system design provides for all these features, and most have been successfully implemented in a working software prototype. The following sections present a summary of the innovative features of the E-Scape software which have fulfilled the above goals.

12.2.3.3 Device communication and control

One of the goals of the design of the E-Scape system was to demonstrate how a software control system can interface to, and allocate resources within any kind of device, and control them in real-time. This obviously has theoretical limitations of communication bandwidth and host processing speed, but with the constant increase in both the absolute speed, and the performance/price ratio of systems and communications links, this limitation will become less practically important in the course of time.

This aspect of the work has been successful, as described in 11.3.1, and messages have been output from an E-Scape score using both MIDI and MIDAS protocols to control synthesis structures on several devices at once (see 11.3.3).

12.2.3.4 Unified, object oriented description of devices

Significant effort has been spent in attempting to build a structure within E-Scape to allow users to build descriptions of all existing devices in a uniform, object-oriented manner, and specify synthesis structures within them.

E-Scape's structure allows the user to describe and specify synthesis or signal processing structures (either at the level of the 'device-level' units, or at a higher level, built out of such units). It also allows a knowledgeable user to define a protocol consisting of message types to instantiate and connect 'device-level' entities, and to send data to their inputs.

As described in 11.3.4, efforts to facilitate object-oriented description of lower-level structures (ie structure *within* device-level units) has proved problematic, as there are a large number of different underlying structural paradigms in use by MIDI-controlled devices, and the presentation of these models is not standardised - the type and structure of the messages used to create the same sort of synthesis structures varies from device to device. An additional problem has been the (often arbitrary) restrictions many devices place on the *combinations* of units they can support.

However, notwithstanding the difficulties encountered, this has been successful in terms of allowing a user to describe 'device level' synthesis structures which exist on different devices in the same way.

12.2.3.5 Protocol independence

E-Scape is expandable in the range of devices it can control, with the ability for an E-Scape user to define new message protocols. This facility is necessary if E-Scape is to be used within the proposed standard communication environment, able to transmit messages on any of the available computer ports, in order to interface to different devices via their own particular message format.

Protocols can be defined by the user. This involves defining various types of message (as described in 10.6), and specifying how the data for each of their fields is derived from score and instrument objects' parameters within the E-Scape system.

If necessary, different Protocols may be defined and used for different categories of synthesis structure on the *same* type of device, or even for different inputs of the *same* structure. For example, a particular device might use two different sets of message types communicating over different interfaces; it might require messages of a certain format to be sent over a SCSI interface to transmit control-rate data, while also using (say) MIDI to start and stop instruments playing. E-Scape could cope with controlling this device using two different Protocols.

12.2.3.6 Allocation of synthesis resources for instruments

E-Scape has been able to automatically allocate synthesis resources on available connected devices by analysing the score requirements. This process has been described in detail in 9.3.1.

Several MIDI-controlled commercial synthesisers have been used as 'raw' synthesis resources, with E-Scape Instruments defining multiple independent structures within them. E-Scape can detect where a score is un-playable on the currently connected devices, either because insufficient units are available for the ScoreEvents present, or because a score parameter is outside the allowed range. This ability to map score requirements to the available device resources is a major strength of E-Scape.

12.2.3.7 Modular design and construction of Instruments

E-Scape allows Instruments to be built from pre-constructed lower-level modules. This has allowed structures to be designed then *reused* in different higher-level structures. Data inputs for these higher-level structures can be created and connected automatically by the system, and can inherit the characteristics of their destinations lower-level modules. This allows structures to be created more quickly and easily than in other modular system such as 'MAX' (see 3.2.1.1).

Many systems - such as MAX, 'ANIMAL' (Lindemann 1990; Lindemann 1991), 'Patchwork' (see 3.2.1.7), 'Katosizer' (Blythe 1986) and 'Unison' (Bate 1992) - allow networks of processes to be specified and connected on a device via a graphic representation of icons and lines connecting their inputs and outputs. In the prototype E-Scape system, *graphic* iconic presentation of these structures has therefore *not* been implemented at this stage, as this aspect is not new. The porting of E-Scape to a new Smalltalk-80 version ('Object works 4.1') with a different graphics implementation is imminent (see 7.5), and means that all current graphics code will require rewriting.

12.2.3.8 Presentation and processing of high-level scoring parameters

Instruments present scoring parameters to a user, and process them into device-level messages assigned to a larger number of inputs to the synthesis structures in each device.

An Instrument can thus present a conceptually simple scoring interface, which requires no knowledge of lower levels within its structure, yet can specify and control complex networks on a variety of device types. For example, MIDI-based synthesisers and custom DSP devices can both be used within the same system in an *integrated* way, ie the user need not realise the differences between the attached devices (apart from their capabilities).

This facilitation of unified and simplified score control over arbitrarily complex synthesis structures, is another major strength of E-Scape.

12.2.3.9 Provision of multiple time-varying event parameters

Any or all of the score parameters of an E-Scape Instrument can be displayed in a graphic score editor as time-domain traces. This allows graphic and/or textual specification by a user of time-varying parameters for each event. The ability to specify and display such multiple parameters for each individual event is an innovative aspect of E-Scape.

12.2.3.10 Distributed Instruments

E-Scape allows synthesis structures to be presented in a *uniform* object-oriented manner different types of device.

Instruments can be built from modules which specify units which are *distributed* over several different synthesis devices which can be of different types. Complex synthesis structures can be created (and scores created for them) using devices which *individually* do not support such complexity.

This facilitation of transparent high-level control over multiple synthesis structures in many different devices (for example, a mixed MIDI-based and 'custom' system) is another innovative feature of E-Scape.

12.2.3.11 Distribution of processing between E-Scape and synthesis device

Data processing within an Instrument can be performed within E-Scape's PspProcessors, or specified as a processing structure within synthesis devices. However, the devices available at present do not support the logical elements (units) needed to do this.

12.2.3.12 Algorithmic composition facilities

The methods needed to build algorithmic Instruments have not yet been fully implemented in E-Scape. However, the structures to enable the design presented in 9.5 to be implemented are in place within the existing Instrument structure. E-Scape's structure will allow events or control data to be generated algorithmically *within* the existing Instrument and ScoreEvent structure, rather than being a separate concept and activity.

12.2.3.13 Extraction and application of abstract gestural data

One of the original design goals of E-Scape was to facilitate the abstraction of time-varying data from any parameters of any group of score events at any level in a hierarchy, and its subsequent applications to modify other events

DMIX (Oppenheim 1992) has been found to provide many elements of this facility - able to extract abstract data from the time or parameter values of events at the bottom level of the event hierarchy, and apply it to modify parameters of other events in a flexible way. Hence this aspect has been left un-implemented at present in the prototype E-Scape system, with development effort being focused on the innovative aspects of the design.

It may be thought that such facilities would be available to E-Scape via object-oriented inheritance from Dmix, but E-Scape's events have a different structure. Hence, *some* adaptation to Dmix methods would be necessary to provide such functionality.

12.3. Further work

As discussed above, several design features of the E-Scape system have been left unimplemented, although fully worked out as a design study.

Level II and III communication have not yet been implemented, as no device as yet supports such standards. The MIDAS/MII system is planned to implement such standards (Anderson 1992a), and E-Scape development will be able to follow; its object-oriented structure and development environment making this feasible.

Messages proposed for intra-system communication have focused on the control-rate event-level parameters which result in audio-rate data in the target synthesis device. Thus no attempt has yet been made to deal with the editing or communication of audio rate data, or the control of soundfile manipulation within a device.

The design for algorithmic event generation (including the use of globally available data patterns) within E-Scape Instruments need to be implemented.

These omissions will form the basis of the future development of the E-Scape software system. Extensions to E-Scape in order to implement the proposed level II and III communication standards will be undertaken in tandem with development of the MIDAS / MII system, in order to demonstrate its viability. It is anticipated that further alterations and extensions will be made as a result of this work.

In addition, further work is planned by the author for the development of E-Scape's *user interface*. The object-oriented structures in E-Scape will allow a relatively safe extension of the user-interface, with higher-level operations under user control.

An initial design has been undertaken by the author which allows the user to construct and control high-level musical activities within the system, with the ability to define and use method templates, which could also be imported from other composers.

This user interface design will allow novice users to safely use the system, making the choices needed and no more. It also facilitates access and control by disabled users with customisable control and presentation features.

This design has formed the basis of a funding application under the EC TIDE initiative by an international partnership of five organisations, to develop E-Scape as a flexible composition system able to be learnt and used by anyone, with the ability to use whatever devices are available. The details of this design study are given in Appendix 4.

12.4. Conclusion

The work in this thesis can make a contribution in various areas of music related activity:

(i) The proposals for the organisation and functionality of a network of inter-communicating systems will be of interest to computer music system software designers, who are often based in institutions who are both the developers and users of such systems. The proposals will help systems become less dependent on specific hardware, and less prone to obsolescence.
Recommendations are also made with regard to the functionality required, in order for computer systems to be able to operate as synthesis devices (sub-systems) within such a network. These will be of interest to commercial synthesiser hardware manufacturers, whose devices at present are almost all MIDI-controlled. The adoption of a wider-ranging set of standards for the behaviour of devices, (or the modification of the existing MIDI standard, as suggested in 11.1.6) will make it economic to construct and market more capable, flexible or specialised devices than at present, with a far greater potential user-base than exists for present large computer music systems. Other markets will be opened up for dedicated software-based synthesis devices, implemented as software running on general purpose computers.
This more wide-spread availability will vastly increase the flexibility of usage of such devices and extend the range of musical activities and compositional styles in *common* use beyond the current mainly rock/pop idiom, into more electro-acoustic areas.

(ii) As stated above, one of the proposals made by this thesis is that composition (event specification) systems should be able to control different types of synthesis device within an integrated structure, as demonstrated by E-Scape. In addition, E-Scape facilitates the transference of all or part of a data processing algorithm into the composition software.
These features will enable composers to employ more flexible and complex synthesis structures and processes using the devices available to them.

(iii) Two key features of E-Scape for a composer lie in its presentation of the task of building, and of then controlling, complex synthesis structures.
A composer can build or modify a structure by manipulating lower-level modules. Modules present an informative interface to a user via their input specifications, and their internal functioning or structure need not be known.
A composer is able to create scores using such complex synthesis structures, again without necessarily needing to know or understand their internal construction. Their operation can be controlled from a score in an intelligible manner, using a variety of simplified ('perceptual') input parameters which can be specified for events in the score.
This capability will enable more traditionally-grounded composers to use their orchestration skills in working easily and transparently with complex sound generating processes, which at present requires an in depth expertise in their construction and functioning.

This in turn will encourage and enable a wider range of composers to expand the scope of their musical activities and work with computer-generated sounds.

The fact that such simplified event parameters can be feasibly specified and understood within a *score* will encourage the usage of written (or drawn) scores by composers of electro-acoustic music. This may have the effect of increasing the communication and shared understanding of such compositional methods, and enable the musical structures and thought processes involved in electro-acoustic composition to approach the stature and communicability of those conveyed by a conventionally orchestrated score.

Appendices

Appendix 1: The MIDAS device and its relevant operational aspects, in response to the proposed communication standards

This appendix presents a general description of the University of York Music Technology Group's MIDAS system. This will be necessary in order to understand the MIDAS communication protocol described in Appendix 2.

1.1. MIDAS structure

MIDAS (described in 3.2.1.9) has a network of processors ('nodes') connected in a 'ring' (not necessarily a topological one) which communicate via defined protocols. It can implement synthesis structures using unit generator processes (UGPs) which can be linked to form complex networks, and which communicate with each other via messages (MIDAS packets) on the MIDAS ring.

Scheduler-UGPs provide the means for the device to store time-stamped packets destined for UGPs. Packets can start other *Scheduler-UGPs* running, thus time-based data structures can be facilitated.

An interactive subsystem called the 'MIDAS Intermediate Interface' (MII) provides a layer of insulation between the precise MIDAS system architecture (including address and processor locations and resource allocation) and a controlling application. It provides an intelligent interface to the low-level messages and includes a database of the addresses and states of UGPs on the ring. A high-level system can thus address all entities in MIDAS via a 'high-level' id (Anderson 1992a), and can request functionality such as storage and scheduling of event or control data.

Messages can be sent to MII in time-stamped or 'immediate' form. The latter are acted upon immediately by MIDAS to generate sound; the former may be sent and processed at a lower rate than is necessary for sound output, with synthesis then occurring off-line.

1.2. MIDAS low-level operation

1.2.1 Scheduler-UGPs

A *Scheduler-UGP* 'stores'[1] MIDAS packets, each having an associated 'send time' and a label. This label can enable the packet to be identified and removed later. For example, a 'start' data-packet will have the *score-event* id as its label, and a 'table assignment' data-packet will have the Instrument input id as a label.

Packets can be added to a *Scheduler-UGP*, and will be sorted by it into time order. Packets with a designated label can also be removed from it.

---

1 Most likely it accesses a table containing this data.

A *Scheduler-UGP* has three *UGP-inputs:-* 'start time', 'stop time' and 'start/stop'. When it receives a 'start' data-packet (sent to its 'start/stop' input) it starts running, first jumping its internal clock to its (previously loaded) 'start time' (default = 0). It will then send out its stored packets when its internal clock time becomes larger than the 'send time' associated with the packet. Each packet contains an address to which it will be sent.

When a *Scheduler-UGP* either receives a 'stop' data-packet, or its internal clock becomes greater than the (previously loaded) 'stop time', no further stored 'start' packets will be sent out. It will then scan through its list of remaining packets looking for 'stop' packets, which it sends out immediately.

Other types of *Scheduler-UGP* may be constructed, eg with 'looping' or 'reversing' abilities.

1.2.2. Wave or breakpoint data

Wave or breakpoint data is contained in data tables, which have an id.

A data table is stored on any node in MIDAS which is running *UGPs* which have been specified to access that table.

Any *UGP* which uses a data table (typically 'OSCIL' and 'TABLE READER' *UGPs*) will have an input called 'table id' which specifies the table to access. This table id can be changed (at audio rate) like any other *UGP* input, allowing a *UGP* to sequentially access different tables as it runs.

NB1. An 'OSCIL' *UGP* could be sent the 'table id' of a data table which is *designed* to be read by a 'TABLE READER' *UGP* (ie it contains breakpoint time and value pairs). The OSCIL would therefore read this table as a set of sequential wave values, which would probably produce noise. This would be a mistake by the user, but is perfectly legitimate in MIDAS.

NB2. Some *UGPs* may access more than one data table at once, eg the 'ADDING OSCIL' *UGP-type* (an imaginary example) which uses two waveforms would have two *UGP* inputs (called 'wave-1 table id', and 'wave-2 table id').

1.2.3. Schemes for specifying time data

Time data needs to be specified for table breakpoints and in *Scheduler-UGPs*.

Using absolute times, with 16 bits per time value, mapped to eg 1ms per value, implies a 65.535s max. duration for a single *score-event* which *may* be inadequate, for certain types of composition.

To cope with this problem, there are various alternatives for storage of time data:-

• Lessen the resolution to enable the specification of a longer *score-event* duration (eg 2ms per value => 2' 11" max. duration). But this precludes a long *score-event* which *also* requires high time resolution.

• Use a larger data size (eg 24 or 32 bits). This may be wasteful of memory, in the many cases where such precision is unnecessary.

• Use delta times[1], enabling any length of *score-event* to be specified, with any time resolution. *Scheduler-UGPs* and 'TABLE READER' *UGPs* would need to interpret this data appropriately.

• Use a variable data size for times, specified in a header (eg times could be stored as 1, 2, 3 *or* 4 bytes). Again *Scheduler-UGPs* and 'TABLE READER' *UGPs* would need to read this information from the header and interpret the data accordingly.

It may often be the case that a single byte is sufficient to specify breakpoint times (eg for very short events). In addition the breakpoint *value* may also be able to be specified in one byte, if higher precision is unnecessary, which could result in considerable memory saving.

---

1 Where only the time difference since the previous event is recorded. This scheme incurs difficulties when a score event is moved in time within a score, or removed from it.

1.3 MII (MIDAS Intermediate Interface)

1.3.1 MII structure

MII is an application which interfaces directly to the MIDAS ring. It can receive 'high-level' messages from a composition / 'event specification' system which is controlling MIDAS as a device, or have a user type commands directly into it. It then formulates and sends the low-level MIDAS message packets onto the MIDAS ring. MII can be considered to consist of two components:

• a 'front end' - which facilitates direct user input via a text editor;

• a 'main section' - which receives data, either from the 'front end', or via messages (as described in Appendix 2) from external systems.

1.3.1.1 The 'MII front end'

The 'MII front end' allows the user to manually type in commands, names and ids as text strings. It then converts these to ASCII symbols which it sends to the 'MII main section'. The same commands, names and ids will also be present within E-Scape, whose output section also converts them to ASCII symbols. Thus, the 'MII main section' (see below) receives the same symbol stream whether originating from E-Scape or the 'MII front end'.

• Messages are text strings which may be typed into the 'MII front end' user interface.

• Each message has a number of fields. Fields are delineated by commas (",") which may be followed by optional white space (space or tab characters).

• The contents of fields (as seen at the E-Scape user's end or 'MII front end') are *names* (a string of characters), *ids* (integers), or *data* integers.

• Within names, any characters or spaces can be used, except the comma and the semicolon, with upper and lower case letters treated as identical.

• The first field is always a *command* name, followed by a variable number of arguments (names or ids).

• Argument names (text strings) will be *UGP-type* names, and *UGP-input* names. Each of these is "#defined" as a unique integer.

• Each message ends with a *carriage_return* (invisible in this document). If the message needs to continue on a new line, a comma (plus optional white space) may *precede* the carriage return, which will then *not* be interpreted as the end of the message.

• All characters on a line after a semicolon (";") will be ignored. Thus a comment may follow a semicolon.

1.3.1.2 The 'MII main section'

This is the part of MII which formulates messages to be sent to MIDAS. It does such things as adding node ids and translating between high-level ids and MIDAS *UGP* ids.

The 'MII main section' receives integers, either from the 'MII front end' (which has encoded the above message strings), or from E-Scape or other high-level system. The integers arrive as streams of ASCII characters delineated by a comma character, (plus optional white space, which is ignored).

These ASCII characters are transferred to the 'MII main section' in a variety of ways:-

- via a disk file.
- via direct serial transfer, such as encoded MIDI, or RS232
  eg1. E-Scape on Mac <-> MII main section on Atari or UNIX system;
  eg2. MII front end on Atari <-> MII main section on UNIX system.
- via a pad - eg. E-Scape (or MII front end) on Mac, <-> MII main section on UNIX system.
- via UNIX sockets
  eg. E-Scape / MII front end <-> MII main section *both* on UNIX system.

1.3.2 MII data structures

MII holds a *score-event* id for every Score-Event in the high-level composition control system (eg. E-Scape).

Each *score-event* id is mapped to:-

- The *Scheduler-UGP* id used for the *score-event* in MIDAS.
- The *instrument-template* id and instance id of the instrument instance assigned to the *score-event.* This instrument instance id then references:-
  - a list of *UGP* ids;
  - a set of *instrument-template* input names (from the *instrument-template*). Each of these inputs then references:-
    - a list of *UGP* ids plus *UGP-input* addresses/party ids for each. These are the destinations fed from this *instrument-template* input in the *instrument-template* specification.
    - the id of a 'TABLE READER' *UGP* associated with the instrument instance for this instrument instance;
    - the id of a MIDAS breakpoint table[1] (for this *score-event*).

---

1 This is also referenced by an E-Scape table id.

Appendix 2: Communication protocol design for external control of the MIDAS system

Appendix 2 presents the detailed design of an implementation for MIDAS of protocols which meet the communication standards proposed and described in chapter 5. It assumes knowledge of the information presented in appendix 1.

In the message examples given below, the message types, UGP types, and input ids are in many cases shown as *names* for clarity. Such names might appear in the user interface of an event specification system (which is using these messages to control MIDAS), or in the text entry 'front end' of MII (see Appendix 1.3.2.1). These names will have numerical equivalents in the actual messages transmitted. UGP types and UGP inputs have names which are defined by as standard by MIDAS, and these by convention are capitalised. Thus, for example, the first example message (in 1a. below) would be displayed as [CreateUGP, 1, OSCIL] but would actually sent to MII as the data 0, 1, 6.

2.1. Design for implementation of level I communication by MIDAS

2.1.1 Creating, deleting, and sending data to UGPs in MIDAS

These are five types of message in this category, each with a time-stamped variant.

1a. Create UGP

This message requests MIDAS to create a UGP of a specified type, with a specified 'high-level' id number. This id is then stored within MII, and will be used for all further reference to this UGP by the controlling 'event specification' software.

MII passes on the creation request to the appropriate MIDAS node(s), by sending out an appropriate MIDAS message packet. A node which is able to successfully service the creation request will return its id, and the UGP address within it to MII. MII then maps this MIDAS address to this high-level id.

MII returns a simple "error" message to the controlling event specification software if the creation request cannot be accommodated by MIDAS.

Message format

Fields	*Comments*
CreateUGP	The message type
UGP id	The 'high-level' reference id for this *UGP*. The event specification system will use this id in all subsequent references to this *UGP*.
UGP-type name	The type of the *UGP* to be created.
[*UGP* instantiation parameters..]	One or more optional parameters, which specify the characteristics of the *UGP*, eg the number of inputs for an N-MIXER *UGP*.

Message Format			
CreateUGP,	*UGP* id,	*UGP-type*	[, instantiation parameters]
Examples			
CreateUGP,	1,	OSCIL	
CreateUGP,	2,	OSCIL	
CreateUGP,	3,	N-MIXER,	2

In response to this, MII will generate a MIDAS "CreateUGP" message packet, and send it out on the MIDAS ring. If no MIDAS node can accommodate the message, MII will be able to report back a general 'error' condition to E-Scape. Future developments of the communication standard may allow a variety of different error messages to be returned by a device (see chapter 5).

1b. Create UGP with time stamp

This message requests the creation of a UGP at a specified future time (measured in ms). Its format is as in (1a) above, plus the supplied time value.

In response to this, MII will create a MIDAS 'CreateUGP' message, and schedule it by loading it into a *Scheduler-UGP*[1] within MIDAS with the specified time. This MIDAS message will subsequently be activated by the *Scheduler-UGP* at the specified time offset, after MII has received a 'run' message. It will then be acted upon by MIDAS in the same way as if received directly as in (1a).

Message Format				
CreateUGPAtTime,	Time,	*UGP* id,	*UGP-type*	[, instantiation parameters]
Examples				
CreateUGPAtTime,	0,	1,	OSCIL	
CreateUGPAtTime,	1500,	2,	OSCIL	
CreateUGPAtTime,	1500,	3,	N-MIXER,	4

[1] See appendix 1.1.1

2a. Connect UGP to another UGP

This message requests the connection of a specified output of a specified UGP (using its high-level id) to a specified input of another UGP (also using its high-level id).

In response, MII sends the corresponding MIDAS message packet, having translated (via its database) the specified high-level UGP ids, and input and output ids into MIDAS addresses.

Message format

Fields	*Comments*
ConnectUGP	The message type
Source *UGP* id	The id of the *UGP* the connection is from.
Source *UGP-output* name	The name of the *UGP-output* the connection is from.
Destination *UGP* id	The id of the *UGP* the connection is to.
Destination *UGP-input* name	The name of the *UGP-input* the connection is to.

Message Format				
ConnectUGP,	Source *UGP* id,	Source *UGP-output* name,	Destination *UGP* id,	Destination *UGP-input* name
Examples				
ConnectUGP,	1	'A-OUT',	2	'AMP'
ConnectUGP,	2	'A-OUT',	3	'IN-1'

2b. Connect UGP to another UGP with time stamp

The message is as (2a) above, plus a supplied time value for scheduling. MII loads the corresponding MIDAS message into a *Scheduler-UGP* in MIDAS, as for (1b) above.

Message Format					
ConnectUGP,	Time,	Source *UGP* id,	Source *UGP-output* name,	Destination *UGP* id,	Destination *UGP-input* name
Examples					
ConnectUGPAtTime,	0,	1,	'A-OUT',	2,	'AMP'
ConnectUGPAtTime,	1500,	2,	'A-OUT',	3,	'IN-1'

3a. Disconnect UGP from UGP

Disconnect a specified output of a specified UGP (using its high-level id) from a specified input of another UGP. MII then sends the corresponding MIDAS message packet, having translated the ids into MIDAS addresses.

Message format

Fields	*Comments*
DisconnectUGP	The message type
Source *UGP* id	The id of the *UGP* the connection is from.
Source *UGP-output* name	The id of the *UGP* output the connection is from.
Destination *UGP* id	The id of the *UGP* the connection is to.
Destination *UGP-input* name	The id of the *UGP-input* the connection is to.

Message Format				
DisconnectUGP,	Source *UGP* id,	Source *UGP-output* name,	Destination *UGP* id,	Destination *UGP-input* name
Examples				
DisconnectUGP,	1,	'A-OUT',	2,	'FREQ'
DisconnectUGP,	2,	'A-OUT',	3,	'IN-1'

3b. Disconnect UGP from UGP with time stamp

The message is as (3a) above, plus a supplied time value for scheduling. MII loads the corresponding MIDAS message into a *Scheduler-UGP* in MIDAS, as for (1b) above.

Message Format					
DisconnectUGP,	time	Source *UGP* id,	Source *UGP-output* name,	Destination *UGP* id,	Destination *UGP-input* name
Examples					
DisconnectUGP,	4000,	1,	'A-OUT',	2,	'FREQ'
DisconnectUGP,	6000,	2,	'A-OUT',	3,	'IN-1'

4a. Delete UGP

Delete a specified UGP (using its high-level id). MII then sends the corresponding MIDAS message packet, having translated the id into a MIDAS address.

Message format

Fields	*Comments*
DeleteUGP	The message type
UGP id	The id of the *UGP* to be deleted.

Message Format	
DeleteUGP,	*UGP* id
Example	
DeleteUGP,	1

4b. Delete UGP with time stamp

The message is as (4a) above, plus a supplied time value for scheduling. MII loads the corresponding MIDAS message into a *Scheduler-UGP* in MIDAS, as for (1b) above.

Message Format		
DeleteUGP,	time,	*UGP* id
Example		
DeleteUGP,	5000,	1

5a. Send data value to UGP

This message sends a data value to a specified input of a specified UGP, using the high-level ids (as specified in messages 1). MII then sends the corresponding MIDAS message packet, having translated the id into a MIDAS address. A data value may start or stop a UGP if addressed to the appropriate input.

Message format

Fields	*Comments*
SendValueImmediate	The message type
UGP id	The id of the *UGP* to be sent to.
UGP-input name	The name of the *UGP* input to be sent to.
Data Value	The value (number) to be sent.

Message Format			
SendValueImmediate,	*UGP* id,	*UGP-input* name,	Data Value
Examples			
SendValueImmediate,	1,	'AMP',	15000
SendValueImmediate,	2,	'AMP',	16000
SendValueImmediate,	2,	'FREQ',	650

5b. Send data value to UGP with time stamp

The message is as (5a) above, plus a supplied time value for scheduling. MII loads the corresponding MIDAS message into a *Scheduler-UGP* in MIDAS, as for (1b) above.

Message Format				
SendValueWithTime,	Time,	*UGP* id,	*UGP-input* name,	Data Value
Examples				
SendValueWithTime,	0,	2,	'AMP',	15000
SendValueWithTime,	0,	1,	'AMP',	16000
SendValueWithTime,	560,	2,	'AMP',	17000
SendValueWithTime,	560,	2,	'FREQ',	600

2.1.2 MIDAS system messages

System messages set up various MIDAS operation parameters, or start or stop its scheduling processing running, when used in 'time stamped' mode (the 'b' message types above). There are six message in this category at present.

1. Run MIDAS

Start the MIDAS Master *Scheduler-UGP* running which will then send out each time-stamped packet stored in it at the appropriate time. The current system clock time is reset to zero, thus MIDAS will start from the first message in the *Scheduler-UGP* .

Message Format
RunMidas

2. Stop MIDAS

Stop ('pause') the MIDAS system clock, which causes all UGPs to stop running. This includes the master *Scheduler-UGP*, which thus suspends the processing of its scheduled messages.

Message Format
StopMidas

3. Continue MIDAS

Restart the MIDAS system clock (*without* resetting the system clock time to zero). This restarts the master *Scheduler-UGP* running[1].

Message Format
ContinueMidas

1 The correct system 'data state' may not be present when the system is run from an arbitrary time. Thus, whether it is useful or workable to start from a place other than the beginning is up to the high-level user.

4. Set MIDAS system time

MII sends a message to set the MIDAS system clock time to the specified value, in ms. A 'Continue device' message will then start processing of messages stored in the master *Scheduler-UGP* with time-stamps later than this time.[1]

Message Format	
SetMidasTime,	*Time*
Example	
SetMidasTime,	10250

5. Reset MIDAS

This message will perform the same actions as for the 'Stop device' message (above), and also empty the master *Scheduler-UGP*. Any subsequent time-stamped messages received will then be loaded to it as part of a new performance.

Message Format
ResetMidas

6. Set MIDAS system sample rate

This message effectively sets the conversion ratio between absolute elapsed time, and MIDAS' system clock 'tick'. Each tick results in the processes necessary to computer another output sound sample being triggered.

Message Format	
SetMidasSampleRate,	*Rate (Hz)*
Example	
SetMidasSampleRate,	44100

2.2. Design for implementation of level II communication by MIDAS

The level II communication standard incorporates all the functionality and messages of level I, as well as additional facilities to specify and store a synthesis structure specification in device memory. This structure specification, as stored in a device, is termed an *instrument-template*. It should not be confused with an Instrument - a software object existing in E-Scape, *part* of whose function is to describe such a device structure.

---

1 As previous footnote.

These stored configurations (and optional default input values) may also be specified to the device via other control systems, or even via an interface directly integrated with the device itself. For example, most MIDI-based synthesiser devices can have (albeit very limited) structures[1] specified from their front panel controls, or via a number of editing software packages communicating to the via MIDI. The MIDAS device can have structures specified via textual input into MII, or via the graphic 'Canute' editor program (which currently runs on a Silicon Graphics 'Indigo' computer).

Before or during score performance, the controlling event specification software can request the instantiation of a synthesis structure specification in a device. To do this it will send appropriate messages which recall the specification from stored memory within the device, and create an active structure of synthesis units.

2.2.1 Specifying *instrument-templates* in MIDAS

An *instrument-template* structure describes a network of *UGPs* in MIDAS, plus (optional) default input values for them.

The *instrument-template* consists of:

- a number of connected *Potential-UGPs*;
- any default initialisation values to be loaded to them;
- table data to be used by all instances of this *instrument-template*. The *instrument-templates*, once created, are stored in MII.

A *'Potential-UGP'* is an entity within an *instrument-template*, which signifies that the corresponding UGP does not yet exist, but is held in template form. The UGP it describes will be instantiated when the *instrument-template* is instantiated.

There are six message types within this category:

---

1 These structures form part of a MIDI synthesiser 'patch' or 'program'.

1. Start *instrument-template* definition.

This message requests that that a new *instrument-template* is to be defined. The specification is actually stored in MII, but MII and MIDAS do not appear as distinct entities to an external system, thus the controlling system sees an *instrument-template* simply as being stored somewhere 'in MIDAS'.

The *instrument-template* has a 'high-level' id specified which is used subsequently by the event specification system to refer to it.

If an *instrument-template* with this id has already been defined, MII erases all previous information about the *instrument-template*. MII then sets up and initialises the necessary data structures to store the *instrument-template* definition data.

Message Format

Fields	*Comments*
StartInstrumentTemplate	The message type
instrument-template id	The id of the *instrument-template* being built up.

Message Format	
StartInstrumentTemplate,	*instrument-template* id
Example	
StartInstrumentTemplate,	'Complex Am'

2. Specify a *Potential-UGP* within an *instrument-template*.

This message indicates that a *Potential-UGP* of a specified type is to be defined within the *instrument-template*. The *Potential-UGP* is given an 'high-level' id within the *instrument-template* which enables the corresponding instantiated *UGP* to be subsequently referred to by the event specification system.

One or more parameters pertaining to the eventual instantiation of the *UGP* may also be specified, eg the 'N-MIXER' *UGP-type* needs the number of inputs specifying when it is instantiated.

Message Format

Fields	*Comments*
SpecifyPotentialUGP	The message type.
instrument-template id	The id of the *instrument-template* being built up.
Potential-UGP id	A unique number identifying this *Potential-UGP* within the *instrument-template*. E-Scape will subsequently refer to this *UGP* by this number.
UGP-type	The id of the *UGP-type*.
[, *UGP* instantiation parameters,..]	One or more optional parameters which specify the characteristics of the *UGP*, eg the number of inputs for an N-MIXER.

Message Format				
SpecifyPotentialUGP,	*instrument-template* id,	*Potential-UGP* id,	*UGP-type* id	[, *UGP* instantiation parameters]
Examples				
SpecifyPotentialUGP,	'Complex Am',	1,	'OSCIL'	
SpecifyPotentialUGP,	'Complex Am',	3,	'N-MIXER',	2

3. Specify a connection between *Potential-UGPs*.

This message specifies a connection from a ('source') output of a *Potential-UGP* to a ('destination') input of another *Potential-UGP*. A single *Potential-UGP* output may be connected to the inputs of *several Potential-UGPs* by sending a series of these messages.

A *Potential-UGP* can be a MIDAS sound output port eg 'DAC-1', DAC-3' [1]. Only *one* of each port may be used in a single *instrument-template* - if more than one source is desired to be sent to the *same* output port, an N-MIXER should be specified overtly within the *instrument-template*, and *its* output connected to the output port.

Message Format

Fields	*Comments*
SpecifyUGPConnection	The message type.
instrument-template id	The id of the *instrument-template* being built up.
Source *Potential-UGP* id	The id of the *source Potential-UGP*.
Source output id	The id of the output of the source *Potential-UGP*.
Destination *Potential-UGP* id	The id of the *destination Potential-UGP*.
Destination input id	An id identifying the input of the destination *Potential-UGP*.

Message Format					
SpecifyUGPConnection,	*instrument-template* id,	Source *Potential-UGP* id,	Source output id,	Destination *Potential-UGP* id,	Destination input id
Examples					
SpecifyUGPConnection,	'Complex Am',	1,	'A-OUT',	6,	'IN-1'
SpecifyUGPConnection,	'Complex Am',	5,	'A-OUT',	6,	'IN-2'
SpecifyUGPConnection,	3,	6,	1,	4,	2

NB. MIDAS already knows about the input and output ids of each *UGP-type*. For example it knows that if a UGP is of type '2-MIXER, it will have *UGP-input* ids as follows:-

input id = 01 (eg) indicates the 'IN-1' input; input id = 02 (eg) indicates the 'IN-2' input.

---

1 The names of MIDAS output ports are defined in the system, and hence have upper case names by convention.

4. Specify an *instrument-template* input

This message specifies an *input* to the *instrument-template* structure being defined. This input can then be connected to *Potential-UGP* inputs within the *instrument-template.*

When an instrument has been instantiated from this *instrument-template,* data value messages can be sent to an instrument input; MII will then route these values to any UGP inputs specified as being connected to the instrument input.

Message Format

Fields	*Comments*
SpecifyInstrumentTemplateInput	The message type
instrument-template id	The id of the *instrument-template* being built
instrument input id	The id of the *instrument-template* input

Message Format		
SpecifyInstrumentTemplateInput,	*instrument-template* id,	*instrument* input id
Examples		
SpecifyInstrumentTemplateInput,	'Complex Am',	3
SpecifyInstrumentTemplateInput,	'Complex Am',	'warble'

5. Specify a connection from an *instrument-template* input to a *Potential-UGP*

This message specifies a connection from a ('source') *instrument-template* input to a ('destination') input of a *Potential-UGP* existing within the *instrument-template.* A single *instrument-template* input may be connected to the inputs of *several* UGPs by sending a series of these messages.

Message Format

Fields	*Comments*
SpecifyInputConnection	The message type
instrument-template id	The id of the *instrument-template* being built up.
instrument-template input id	The id of the source *instrument-template* input.
Destination *Potential-UGP* ID	The id of the destination *Potential-UGP.*
Destination *UGP* input id	The id of the input of the destination *Potential-UGP.*

Message Format				
SpecifyInputConnection,	*instrument-template* id,	*instrument-template* input id,	Destination *Potential-UGP* id,	Destination *UGP* input id
Examples				
SpecifyInputConnection,	Complex Am,	'am waveform',	5,	'TABLE-ID'
SpecifyInputConnection,	Complex Am,	'nuance',	5,	'A'

6. Specify an initialisation value for an input of a *Potential-UGP*

This message specifies a data value which is to be sent to an input of a UGP when it is first instantiated. MII then stores this value, and will send it to the corresponding input of the corresponding *UGP* when it instantiates an *instrument-instance*. This will be appropriate for *UGP* inputs which do not change from one *score-event* to the next. An external system (eg E-Scape) will *not* subsequently be able to access this input, unless a connection from an *instrument-template* input is also set up (by message 6).

Message Format.

Fields	*Comments*
SpecifyUGPInitialisationValue	The message type.
instrument-template id	The id of the *instrument-template* being built up.
Potential-UGP id	The id of the *Potential-UGP*.
UGP input id	The id of the input of this *Potential-UGP*.
Initialisation value	The value assigned to be sent to this *UGP-input* when this *instrument-template* is first instantiated in Midas. NB. A value can be expressed as a *name* in the high-level system (eg an E-Scape table name) as in the example below.

Message Format				
SpecifyUGPInitialisationValue,	*instrument-template* id,	*Potential-UGP* id,	*UGP* input id,	Initialisation value
Examples				
SpecifyUGPInitialisationValue,	Complex Am,	3,	'TABLE-ID',	'7-odd'
SpecifyUGPInitialisationValue,	Complex Am,	4,	'TABLE-ID',	23

2.2.2 Creating and deleting instrument instances in MIDAS

Instances of a structure conforming to a defined *instrument-template* (see 2.2.1) can now be created or deleted. Two message types are used:-

1. Create *instrument-instances*

This message requests the creation (instantiation) of a specified number of instrument instances using the specified *instrument-template.*

The number of instances requested equals the 'polyphony' , ie the number of 'voices' of this *instrument-template* which need to be created and run simultaneously to cope with the maximum number of simultaneous *score-events* assigned to that *instrument-template*[1].

MII responds by creating a number of *instrument-instances* (of this *instrument-template*) in MIDAS. For each instrument instance, MII will:-

• Allocate an instrument instance id. This is stored in MII in association with the id (address in MIDAS) of each *UGP* to be created. Thus, all the *UGPs* in MIDAS which correspond to a single instrument instance can subsequently be referenced by a high-level system (eg E-Scape) via this id.

• Create and connect *UGPs* in MIDAS according to the specification in the *instrument-template* which has previously been created and stored in MII. There are two stages to this:-

- Create *UGPs*, which then have an id (address) in MIDAS. Which *UGPs* are to be created is defined by the corresponding *Potential-UGPs* in the *instrument-template.*
- Connect the *UGPs*, by assigning one or more 'destination' UGP input ids to the output of each UGP. Which connections are to be made between *UGPs* is specified by the connections of the corresponding *Potential-UGPs* in the *instrument-template.*

• Create and connect an additional 'N-MIXER' *UGP* in MIDAS corresponding to each MIDAS output port UGP (eg 'DAC-1') in the *instrument-template.*

Each N-MIXER output is then connected to the named MIDAS output port. The 'N' parameter (ie how many inputs to specify for the 'N-MIXER' *UGP*) is equal to the specified polyphony, ie the number of instances requested. Each N-MIXER then receives the output from the N instrument instances created.

NB. If some instances of this *instrument-template* have *already* been created (by a previous message of this type), then the old N-MIXER associated with these existing instances will be deleted; a new N-MIXER with the new (larger) number of inputs will then be created, and connected to the old and new instrument instances.

---

1 If the number of simultaneous *score-events* specified to play using this *instrument-template* is higher than MIDAS can manage in the time allocated, some report back can be made to MII, and thence back to E-Scape.

Message Format

Fields	*Comments*
CreateInstrumentInstances	The message type.
instrument-template id	The id of the *instrument-template* to be used.
Number of instances	The number (N) of instances requested to be created with this *instrument-template*.
	These instances can then be referred to by E-Scape, using the instance id numbers in conjunction with this *instrument-template* id. These instance id numbers are allocated consecutively from n to n+N, where n is the number of *existing* instances of this template. Both E-Scape and MII need to keep track of these id numbers.

Message Format		
CreateInstrumentInstances,	*instrument-template* id,	Number of *instrument-instances*
Example		
CreateInstrumentInstances,	'Complex Am',	3

2. Delete a specified *instrument-instance*

This message requests the deletion of an instrument instance, specified by its instance id, and its *instrument-template* id. MII then sends a MIDAS deletion message to each UGP in this instrument instance.

Message Format.

Fields	*Comments*
DeleteInstrumentInstance	The message type
instrument-template id	The id of the *instrument-template*.
instrument instance id	The instrument instance id (of this *instrument-template*).

Message Format		
DeleteInstrumentInstance,	*instrument-template* id,	*instrument-instance* id
Examples		
DeleteInstrumentInstance,	'Complex Am',	1
DeleteInstrumentInstance,	'Complex Am',	2
DeleteInstrumentInstance,	'Complex Am',	3

2.2.3 Sending data values to instruments in MIDAS

Data values can be sent to a specified input of an instrument instance

One message type is used:-

1. Send an input value to an instrument instance in MIDAS

This message sends a single data value to a specified input of a specified instrument-instance in MIDAS. As in the previous message, instrument instances are referred to using an id for the *instrument-template* and the id of the instance *of* that template.

MII then sends this value in a MIDAS packet to the inputs of the UGPs connected to this instrument input.

Message Format

Fields	*Comments*
SendInstrumentInputValue	The message type
instrument-template id	The id of an *instrument-template* in the device.
instrument-instance id	The id of an *instance* of this template within the device
instrument input id	The id of the instrument input the value is to be sent to.
value	The input value to be sent

Message Format				
SendInstrumentInputValue,	*instrument-template* id,	*instrument-instance* id,	instrument input id,	value
Examples				
SendInstrumentInputValue,	'Complex Am',	9,	'nuance',	3

2.3. Design for implementation of level III communication by MIDAS

As described in section 5.3, a composition ('event specification') system will possess score structures (termed *score-super-events*) which consist of a series of lower-level child events, each with a time offset within the parent. If the event specification system facilitates *object-oriented* hierarchical score structuring (see 7.3), then these child events may themselves be *score-super-events*, or be note-like events (which do *not* contain further child events). These low-level 'note-like' events are here termed *score-events* [1], and nested structures containing child events are termed *score-super-events.*

As required by the level III communication standard, MIDAS allows the creation of data structures which allow score structures and their parameters to be *downloaded* into it from the event specification system. In addition, events can be scheduled and stored in a nested structures (containing time-stamped events) within MIDAS.

Level III communication to MIDAS involves the use of the Level II messages described above to define *instrument-templates* and request the creation of *instrument-instances* within MIDAS.

When creating instrument instances, MII responds with the same actions as at level II (see 2.2.2), ie creating and connecting UGPs as specified in the *instrument-template.* If operating at level III, however, MII's response involves two additional steps:-

- Creating a 'TABLE READER' *UGP* corresponding to each input of the new instrument instance.
- Connecting this 'TABLE READER' *UGP* to the appropriate *UGPs*, ie those connected to this input of the instrument instance.

Within level III, additional messages are then used for the following purposes :-

- Defining and deleting *score-event* structures within MIDAS;
- Creating and deleting *score-super-event* structures within MIDAS;
- Starting and stopping these event structures within MIDAS;
- Sending real-time data values to event structures in MIDAS;
- Creating and deleting data tables within MIDAS.

---

1 Again, note that *score-event.* is hyphenated and in lower case, and should be distinguished from the specific names used by systems. For example, E-Scape uses a ScoreEvent object to describe this structure, while Dmix uses a NoteEvent (again note the capitalisation indicating a software object).

2.3.1 Defining *score-events* to MIDAS

Three message types are used to create a data structure in MIDAS which corresponds to a *score-event* in the composition system (eg E-Scape).

This data structure consists of:

- an assignment of an existing instrument instance, along with its newly-created 'TABLE READER' *UGPs,* each assigned to its associated instrument input. Each TABLE READER is connected to the *UGPs* in the instrument instance according to the corresponding connections of the *instrument-template*'s inputs.
- a *Scheduler-UGP* containing data-packets, addressed to the 'table assignment' input of each TABLE READER. This input specifies which data table will be read by the UGP (see 2.3.6).
- one or more tables loaded with breakpoint data from the *score-event*.

Details of these structures are given below.

Three message types are used in this category:

1. Start *score-event* definition

This message requests the definition in MIDAS/MII of a new data structure to correspond with a specified *score-event*, using a specified instrument instance. This is referenced by its high-level *instrument-template* id, and instrument instance id (see 2.2.2).

In the proposed level III communication standard, is the responsibility of the event specification system (eg E-Scape) to select a 'free' or 'potentially free' *instrument-instance* from the available pool (instantiated earlier in MIDAS by message 2.2.2(1)).

A 'free' instrument instance is as yet unassigned to any *Score-Event.* This will be the case for the first N *Score-events* using this *Instrument-template*, where N is the polyphony (number of instrument instances). A 'potentially free' instrument instance is one which is already assigned to one or more *Score-events*, which are *not* scheduled to overlap in time with the new *Score-event*, ie they will all stop before the new one is to start, or start after the new one is to stop.

For the event specification system to select the appropriate instrument instance to assign, it needs to use the start and stop times of the *score-event*, plus its knowledge of the start and stop times of other *score-events* which are *already* allocated to instrument instances (of the same *instrument-template*).

Message Format

Fields	*Comments*
StartScoreEventDefinition	The message type.
score-event id	A unique number identifying an E-Scape *score-event.*
instrument-template id	The *instrument-template* id to be used by the *score-event.*
instrument instance id	The instrument instance id (of this *instrument-template*).

Message Format			
StartScoreEventDefinition,	*score-event* id,	*instrument-template* name,	instrument instance id
Examples			
StartScoreEventDefinition,	23,	'Complex Am',	2
StartScoreEventDefinition,	1209,	'Complex Am',	6

MII responds by creating a *Scheduler-UGP* for the specified *score-event* id. MII then associates the MIDAS id of this *Scheduler-UGP* with the specified high-level (E-Scape) *score-event* id. Hence a request from E-Scape to play a *score-event* can subsequently be accomplished by MII sending a 'start' data-packet to the corresponding *Scheduler-UGP* (to its 'start/stop' input).

2. Assign a table to an Instrument input

This message specifies the id of a *data table* within MIDAS which is to be linked to a specified instrument input, for a specified *score-event*. The following is specified:-

- a table id;
- an *instrument-template* input id;
- the *score-event* id.

A table with this id will be required to be known within MII (specified via messages in 2.3.8), although not necessarily at this stage. The *score-Event* id is mapped by MII to the corresponding MIDAS data structure as described above.

MII responds by scheduling the specified table to be read by the appropriate 'TABLE READER' *UGP* at the start of the *score-event*. This is the 'TABLE READER' which corresponds to the specified instrument input id. This 'TABLE READER' *UGP* is part of the instrument instance assigned to the specified *score-event* when it was first defined (by message 2.3.1(1)). It is likely to be scheduled to read different tables during the performance, as the *instrument-instance* is allocated to different *score-events*.

However, each 'TABLE READER' *UGP* may only read a single table *during* a *score-event*, and must start reading it immediately. Otherwise, the TABLE READER will output the final breakpoint value from the *last* table it read (when this Instrument instance was playing its previously assigned *Score-event*). This is because a TABLE READER is a special kind of *UGP* which reads (via its table) varying data from a parameter of a *score-event*. Thus it is nonsensical for an 'TABLE READER' *UGP* to change tables during a *score-event*, as each table contains the parameter data of a *score-event*.

However, this must not be taken to imply that the user cannot specify a change of table during a *score-event* for a 'normal' *table-reading UGP*, ie one which corresponds to a 'TABLE READER' *UGP-type* within an *instrument-template*[1]. A user could specify such a 'normal' table-reading *UGP-type* to exist within an *instrument-template* structure, with its 'TABLE ID' input connected to an input (named say 'table no.') of the *instrument-template*. This 'table no.' input could then be connected (via a PspProcessor in the E-Scape Instrument object) to a parameter in the score which *may* be changed in the course of a *score-event*.

Message Format

Fields	*Comments*
AssignTableToInput	The message type
score-event id	A unique number identifying an E-Scape *score-event*.
instrument input id	The id of an input of the *instrument-template* used by this *score-event*.
Table id	The id (name or number) of the table to be assigned

Message Format			
AssignTableToInput,	*score-event* id,	instrument input id,	Table id
Examples			
AssignTableToInput,	1209,	'am waveform',	'triangle'
AssignTableToInput,	1209,	'mod waveform',	1

MII responds by:

- 'Un-scheduling' any existing 'table assignment' data-packet (with the specified *instrument-template* input name as a label), by removing it from the *score-event's Scheduler-UGP*.
- Creating a 'table assignment' data-packet containing the id of the new table, addressed to the 'TABLE READER' *UGP*.
- Loading the following information into a slot in this *score-event's Scheduler-UGP*:-
  - the 'table assignment' data packet;
  - a 'send time' of zero, (ie immediately);
  - the *instrument-template* input name (as a label).

During a subsequent performance, this *Scheduler-UGP* will send the 'table assignment' data-packet to the 'TABLE READER' *UGP* (into its 'table id' input) at the start of this *score-event,* which will then be set up to read from this table.

3. Finish *score-event* definition

This message informs MII that all instrument inputs have had values or tables specified for this *score-event,* and specifies the duration of the *score-event* in ms.

Message Format

Fields	*Comments*
FinishScoreEvent	The message type
score-event id	A unique number identifying an E-Scape *score-event*
Duration	Duration (ms)

Message Format		
Finish *score-event,*	*score-event* id,	Duration
Example		
Finish *score-event,*	90,	1200

MII responds by scheduling the starting and stopping of the designated instrument instances to match the *score-event's* duration. *UGPs* within the instrument instance will be started running by sending a 'start' data-packet to the appropriate *UGPs* within it.

To do this scheduling, MII:

- Creates one or more MIDAS 'start' data-packets, addressed to the appropriate UGP(s) in the instrument instance;
- Creates 'stop' data-packet(s), again addressed to the appropriate UGP(s) in the instrument instance;
- Stores the 'start' and 'stop' data-packets in the *Scheduler-UGP,* with time offsets of zero, and the specified duration respectively.

2.3.2 Deleting *score-events* in MIDAS

Two message types are used to remove the MIDAS *score-event* data structures:-

1. Un-assign a data table from an instrument input

This message requests the unlinking of the table assigned to the specified instrument input for the specified *score-event*. This message is the inverse of 2.3.1(2).

Message Format

Fields	*Comments*
UnassignTableFromInput	The message type
score-event id	The high-level id of the *score-event* structure in MIDAS
instrument input id	The id of the *instrument-template* input to be unassigned for this *score-event*

Message Format		
UnassignTableFromInput,	*score-event* id,	instrument input id
Example		
UnassignTableFromInput,	90,	'nuance'

MII responds by:

- Removing the existing 'table assignment' data-packet (addressed to the specified instrument input) from this *score-event's Scheduler-UGP* in MIDAS
- Storing another 'table assignment' data-packet with the value zero in the *score-event's Scheduler-UGP* with a 'send time' of zero (and again with the specified input id as label). When subsequently sent, this indicates that the 'TABLE READER' *UGP* should not read any table. If this packet were not sent at the start of this *score-event*, the TABLE READER would start reading the table it was assigned to for the *previous score-event*.

The event specification system must then assign *another* table (via message 2.3.1(2)) to this instrument input before playing this *score-event*.

2. Delete a *score-event.*

This message requests MIDAS to remove the data associated with the specified *score-event.* This message is the inverse of 2.3.1(1).

Message Format

Fields	*Comments*
DeleteScoreEvent	The message type
score-event id	The high-level id of a *score-event*

Message Format	
DeleteScoreEvent,	*score-event* id
Example	
DeleteScoreEvent,	90

MII responds by deleting the *score-event's Scheduler-UGP* and its contents (a table of times and packets).

2.3.3 Creating and deleting *score-super-events* in MIDAS

As stated at the start of 2.3, data structures can be created in MIDAS to facilitate the hierarchical scheduling of event structures. The event specification system is thus able to download such an event structure (eg an E-Scape SSE object) to MIDAS in advance, and then start it playing with a single message.

This downloading has obvious benefits in easing communication bandwidth restrictions between the event specification system and the devices it is controlling, but does incur a time delay while downloading occurs. There is also a resulting lack of score playback flexibility: the score is effectively 'compiled' into a single parent *score-super-event,* and can then only be played in its entirety - ie with *all* its event components - albeit with the ability to start and stop at arbitrary times within it.

Four message types are used in this category:-

1. Define a new *score-super-event*

This message requests the definition of a new data structure in MIDAS, to correspond with the specified *score-super-event* id, and hold a specified maximum number of child events.

Message Format

Fields	*Comments*
DefineScoreSuperEvent	The message type.
score-super-event id	The id of the new E-Scape *score-super-event.*
Max. no. of events	The maximum number of child events the user is likely to load into this parent *score-super-event.*

Message Format		
DefineScoreSuperEvent,	*score-super-event* id,	Max. no. of events
Example		
DefineScoreSuperEvent,	2,	100

MII responds by creating a *Scheduler-UGP* for the specified *score-super-event* id. It associates the MIDAS id of this *Scheduler-UGP* with the specified *score-super-event* id. Hence a subsequent request from E-Scape to play a *score-super-event* can be accomplished by MII sending a 'start' data-packet to this *Scheduler-UGP.*

The size of the *Scheduler-UGP* is set to match the specified maximum number of child events. This is less of a burdensome restriction than may at first appear, as these child events could *also* be *score-super-events* (each with their *own Scheduler-UGP* and child events).

In this way, large hierarchical score structures can be downloaded to MIDAS to mirror their structure as described in E-Scape or other composition system.

2. Schedule an event

This message requests MIDAS to schedule ('compile') a specified child event (*score-event* or *score-super-event*) within a specified parent *score-super-event.* The child event has a specified start time offset, relative to the *score-super-event* (see appendix 1.1.3 for schemes for specifying time data in MIDAS).

NB. These schedule request messages need *not* be sent in the time order of *score-events* within the *score-super-event.* MIDAS effectively sorts these, as data-packets stored in a *Scheduler-UGP* are organised by it into time order.

Message Format

Fields	*Comments*
ScheduleEvent	The message type
Event id	The id of an E-Scape event to be *event-scheduled* within the parent *score-super-event.*
score-super-event id	The id of the parent *score-super-event.*
Start time	The time (in ms) the event is to start within the *score-super-event.*

Message Format			
ScheduleEvent,	Event id,	*score-super-event* id,	Start time
Examples			
ScheduleEvent,	33,	2,	4040

MII responds by:

- Creating a 'start' data-packet addressed to the *Scheduler-UGP* of the event to be scheduled.
- Storing this data-packet in the *Scheduler-UGP* of the parent *score-super-event,* along with the specified start time, and a label consisting of the id of the event (*score-super-event* or *score-event*) to be scheduled. The *Scheduler-UGP* sorts its packets into their time order.

NB. No 'stop' data-packet is needed as the event has any necessary 'stop' packet(s) stored in its assigned *Scheduler-UGP.*

3. Un-schedule an event

This message requests MIDAS to remove ('uncompile') a specified event from a specified parent *score-super-event.* This is the inverse of message 2.3.5(2) above.

Message Format

Fields	*Comments*
UnScheduleEvent	The message type.
Event id	The high-level id of the event to be un-scheduled.
score-super-event id	The id of the parent *score-super-event* this event is within.

Message Format		
UnScheduleEvent,	Event id,	*score-super-event* id
Example		
UnScheduleEvent,	33,	2

MII responds by removing the appropriate 'start' data-packet (ie the one which has the specified event id as a label) from the *Scheduler-UGP* of the parent *score-super-event.*

NB. This event will still exist in MIDAS, and may still be started 'manually' (by message 2.3.6(1) below).

4. Delete a *score-super-event*

Request the removal/un-assignment in MIDAS of the data structures which correspond with the specified *score-super-event* id.This is the inverse of 2.3.5(1).

Message Format

Fields	*Comments*
DeleteScoreSuperEvent	The message type.
score-super-event id	The id of the parent *score-super-event.*

Message Format	
DeleteScoreSuperEvent,	*score-super-event* id
Example	
DeleteScoreSuperEvent,	2

MII responds by deleting the *Scheduler-UGP* owned by the specified *score-super-event.*

2.3.4 Starting and stopping events in MIDAS

A *score-event* or *score-super-event* stored in MIDAS can be requested to start or stop. Two message types are used:-

1. Start an event immediately

This message requests that MIDAS starts the event with the specified id playing from its beginning, or (optionally) *from* a start time or *to* a stop time within it.

Message Format

Fields	*Comments*
StartEvent	The message type.
Event id	The high-level id of the event to be started.
[Start time]	Optional start time within the event to start playing from (ms).
[Stop time]	Optional stop time within the event for it to stop playing (ms).

Message Format			
StartEvent,	Event id	[, Start time]	[, Stop time]
Examples			
StartEvent,	56		
StartEvent,	56,	500,	1200

MII responds by sending one or more packets to the event's *Scheduler-UGP* :-

- (optionally) a data-packet containing the start time (sent to its 'start time' *UGP-input.*);
- (optionally) a data-packet containing the stop time;
- a 'start' data-packet.

2. Stop an event immediately

This message requests that MIDAS immediately stops the event with the specified id playing.

Message Format

Fields	*Comments*
StopEvent	The message type.
Event id	The high-level id of the event to be stopped.

Message Format	
StopEvent,	Event id
Example	
StopEvent,	56

MII responds by sending a 'stop' data-packet to the event's *Scheduler-UGP*.

2.3.5 Sending data values to *score-events* in MIDAS

A single message type is used to send individual data values to a *score-event* in MIDAS:

1. Send input value

This message is used to send a single data value for a specified instrument input for a specified *score-event* (previously defined in MIDAS). The message can be sent at any time - the data could be a value entered ('played in') to E-Scape in real-time, or from a breakpoint function of a ScoreEvent object in E-Scape.

Thus, if only a few values are specified for a ScoreEvent parameter in E-Scape, this message will be preferable (involving less MIDAS processing and data storage) to the assigning of *tables* (by message 2.3.1(2)) to instrument inputs.

Message Format

Fields	*Comments*
SendInputValue	The message type.
score-event id	A unique number identifying the E-Scape event to be started.
instrument input id	Id of the *instrument-template* input the value is to be sent to.
value	The input value to be sent.

Message Format			
SendInputValue,	*score-event* id,	instrument input id,	value
Examples			
SendInputValue,	90,	'nuance',	3

MII responds by sending a data-packet containing the specified value to the 'single value' input of the 'TABLE READER' *UGP* which corresponds to the specified *instrument-template* input. This UGP is part of the instrument instance which has been allocated to the specified *score-event* within MIDAS.

The 'single value' input loads a value into the TABLE READER, which, when running, will then output this value to the *UGPs* connected to it within its instrument. This input value will persist until another input value is sent, or the TABLE READER starts reading an assigned table.

If this *score-event* is already playing, then the 'TABLE READER' *UGP* will already be running - and be reading a previously-assigned table. In this case, this new input value will *replace* the current value as read the table. It will remain active until the next breakpoint is read from the table by the *UGP*.

2.3.6 Creating and deleting data tables in MIDAS

Data can be stored in tables in MIDAS, with a high-level id specified to MII.

E-Scape can subsequently refer to the table by this id, which will be referenced by MII to the id of the table as stored in MIDAS.

MII can send this MIDAS table id in a data-packet to the 'table id' input of a table-reading *UGP* (eg. of *UGP-type* 'TABLE-READER' or 'OSCIL').

Three message types are used:-

1. Load table data

This message loads a specified number of values to MIDAS, to be stored in a table with a specified high-level reference id.

Values can be grouped singly (eg. for a wavetable), or into twos (eg. for breakpoint functions), or threes or more. The format will only be important when the data is read and interpreted by a *UGP* in MIDAS.

The data consists of the number of values to follow, followed by a set of values. This is typically used for wavetable data, although any format of data can be used.

Message Format

Fields	*Comments*
LoadTable	The message type.
Table id	The high-level id identifying this table in MIDAS.
Number of values	The number ('N') of data values to follow.
value 1	Data for the first value (an integer).
[, value 2 ... value N]	Data for additional values.

Message Format							
LoadTable,	Table id,	Number of values,	value 1	[, value 2	, ...	, ...	, ..., value N]
Example							
LoadTable,	Table id,	Number of values,	value 1	, value 2	, value 3	, value 4	, ..., value 512
LoadTable,	'looped-att',	512,	0,	-3,	-12,	-25,	, ..., 0
LoadTable	2,	128,	0,	-8,	-23,	-45,	,... , 25

2. Load breakpoint table data

This message loads a specified number of *pairs* of data values into a table with a specified high-level id. This message is a special case of message (1) above, with values grouped *explicitly* into pairs. It is used to make clearer the loading of breakpoint tables, as these are so common. Note that a breakpoint table *could* also be loaded using message (1), see appendix 1.1.2.

The table data consists of:-

- the number of breakpoints to follow;
- a set of breakpoint (time + value) pairs, with time expressed in ms.

Message Format

Fields	*Comments*
LoadBreakpointTable	The message type.
Table id	The high-level id identifying this table in MIDAS.
Number of breakpoints	The number ('N') of data pairs to follow.
Breakpoint-1 data	Data for the first breakpoint. It consists of two data items usually signifying time and value.
[Breakpoint-2 data, ... Breakpoint-N data]	Data for optional additional breakpoints.

Message Format								
LoadBreakpointTable,	Table id,	Number of breakpoints,	Bp-1 Time,	Bp-1 Value,	Bp-2 Time ,	Bp-2 Value,	... Bp-N Time,	Bp-N Value
Example 1								
LoadBreakpointTable,	Table id,	Number of breakpoints,	Bp-1 Time,	Bp-1 Val,	Bp-2 Time ,	Bp-2 Val,	Bp-3 Time,	Bp-3 Val,
LoadBreakpointTable,	39,	3,	0,	3,	50,	7,	75,	7
Example 2								
LoadBreakpointTable,	Table id,	Number of breakpoints,	Bp-1 Time,	Bp-1 Val,	...Bp-3 Time,	Bp-3 Val,	...Bp-7 Time,	Bp-7 Val,
LoadBreakpointTable,	'triangley',	7,	0,	1,	...560,	2,	..1200,	0

3. Delete table data

This message requests the deletion of a data table with the specified user id within MIDAS.

This message is the inverse of (1) or (2) above

Message Format

Fields	*Comments*
DeleteTable,	The message type.
Table id	The high-level id identifying this table in MIDAS.

Message Format	
DeleteTable,	Table id
Example	
DeleteTable,	39
DeleteTable,	'triangley'

2.4. Performance and editing on MIDAS

2.4.1 Performance options

The design for an implementation of MIDAS' response to level III messages is here presented.

Usually, E-Scape will create *score-event* structures in MIDAS in advance, by pre-loading *score-event* data from the event specification (composition) system and allocating it to appropriate instrument instances in advance.

These *score-event* structures stored in MIDAS can then be triggered in two ways:-

- triggered 'on the fly' by E-Scape as it plays the score;
- scheduled ('compiled') into a parent *score-super-event* which can then be started with a single message to MIDAS.

In either case, the user can perform 'interactive scoring' (Hunt 1991), altering one or more score parameters 'live' as the score plays, using message 2.3.5.

Score data may be sent in two ways:-

- Parameter data from the *score-event* may be loaded[1] to a *score-event* (by message 2.3.1(2)), then immediately used;
- Single values can be sent to a *score-event* in MIDAS (message 2.3.5(1)).

This facilitates the option of real-time performance control of MIDAS, eg from a keyboard or wind controller.

2.4.2 Performance examples

Controlling the playing of events already present in MIDAS uses messages in 2.3.4.

An event can be a *score-event or* a *score-super-event* structure in MIDAS. Both structures may be scheduled (nested) within a parent *score-super-event*, but can still be accessed and controlled independently as well, each having its own *Scheduler-UGP* in MIDAS[2].

Example1

To play an event all the way through:

E-Scape would specify the event id with *no* 'start time' or 'stop time' arguments in the message.

---

1 The parameter data will be loaded into a table in MIDAS. A 'TABLE READER' *UGP* will then be created (and connected to the instrument input), and reads this table.

2 MII keeps a record of each *score-event* id, which it maps to an instrument instance id, table ids, and the id of the assigned*Scheduler-UGP* in MIDAS.

MII also maps each *score-super-event* ID to a *Scheduler-UGP* id.

MII would then send only a single 'start' data-packet to the event's *Scheduler-UGP*. This would then start running from the beginning to the end (ie from its internal clock set to zero, until it has sent out its last stored packet).

Example 2a

To play a *segment* of a *score-super-event* - ie a block of consecutive child events within a specified time span[1]:-

E-Scape would send message 2.3.4 (1), specifying-

- the *score-super-event* id;
- the start time of the block within the *score-super-event*;
- the stop time of the block (again within the *score-super-event*).

MII would then send the following data to the appropriate inputs[2] of the *Scheduler-UGP* assigned to this *score-super-event*:-

- the 'start time';
- the 'stop time';
- a 'start' data-packet.

The *Scheduler-UGP* would then start running from this 'start time' to the 'stop' time'.

Example 2b

To play a *segment* of a *score-event*[3] - a set of contiguous sections of time-varying parameter data within a specified time span:-

E-Scape would send message 2.3.4 (1), specifying:-

- the *score-event* id;
- the start time (within the *score-event*) of the segment;
- the stop time (again within the *score-event*) of the segment.

MII would then send the following data to the *Scheduler-UGP* assigned to the *score-event*:-

- the 'start time';
- the 'stop time';
- a 'start' data-packet.

---

1 These child events will have been previously scheduled within the *score-super-event.*

2 For example, the 'start time' data-packet will be sent to the 'start time' input of the *Scheduler-UGP.*

3 The *score-event* may have been previously scheduled within a *score-super-event*, but can still be accessed and played independently as well.

The *Scheduler-UGP* would then start running from this 'start time' to the 'stop' time'. The appropriate *UGPs* in the instrument instance playing this *score-event* will then be sent a 'stop' data-packet at this time, to stop them running.

2.4.3 Editing Examples

A *score-event* can be scheduled ('compiled') into a *score-super-event* structure in MIDAS, or un-scheduled ('deleted') from it at any time.

Example 1

To move a *score-event* in time within the same *score-super-event* :

MII will un-schedule the old *score-event* then schedule it again with a new start time.

Example 2a

To move a *score-event* from one parent *score-super-event* to another:

MII invokes the same procedure as in example 1, except that it then *schedules* the *score-event* into the *new* parent *score-super-event.*

Example 2b

To move a *score-super-event* from one parent *score-super-event* to another:

MII uses the same procedure as in example 2a.

Example 3

To edit the input parameter data of a Score-event:-

E-Scape will request MII to un-assign the data table containing the old parameter data from the *score-event* in MIDAS, and then delete the table.

E-Scape will then request MII to load the edited data into a new table, and then assign the table to the *score-event* structure in MIDAS.

Appendix 3: Selected papers published

Perceptual Parameters - Their Specification, Scoring and Control within two Software Composition Systems

International Computer Music Conference, Tokyo, Japan. 1993

Tim Anderson,

Music Technology Group
University of York
YORK YO1 5DD, UK
tma@ohm.york.ac.uk

Daniel V. Oppenheim

CCRMA
Stanford University, Stanford,
CA 94305
dan@ccrma.stanford.edu

Composers think in terms of perceptual concepts whereas synthesis requires the specification of physical device parameters. Composers would like to work by defining and specifying perceptual parameters within a given sound, rather than synthesis parameters. This requires a system to translate from such user-defined perceptual aspects to the larger numbers of parameters which control the synthesis of that aspect. We describe the implementation of this concept within two compositional systems: DMIX and E-Scape.

The Concept of Perceptual Mapping

Composers think about their music in perceptual terms whereas computers control synthesis parameters that typically model acoustic parameters. The correlation between human perception and the related acoustic parameters is complex and not clearly understood. This raises many problems for composers, particularly when working with the timbral aspects of a musical event. A simple example will clarify this. Let us assume a composer is working with a sound, and finds within it two qualities he would like to work with at the compositional level. He names the first 'smooth' and the second 'grainy'. Note that unlike parameters such as pitch or duration, smooth and grainy are meaningless unless associated with the specific sound and the musical context in which it is embedded. He would now like to work by specifying and controlling the perceptual aspects of the sound that he defined.

Changes in such a perceptual aspect may require corresponding changes in many synthesis parameters. Furthermore, the manner of change in each synthesis parameter might be

different. The composer must first discover how each synthesis parameter should change in relation to a desired perceptual change. Then he can control the synthesis by specifying the correct physical-parameter values. It can be extremely tedious to input this data via a score, even if using a high-level programmable music description language, such as Quill [Oppenheim, 1990]. Real-time control of many parameters is also hard to accomplish as it requires exceptional performance skills, especially if parameters change independently of each other. Clearly, a mechanism is needed that will allow composers to specify and control only the perceptual aspects they define and hide the complex non-linear mapping into the many corresponding synthesis parameters.

Perceptual Mapping Design

An important design consideration is the location of such perceptual-mapping within the overall system—whether in the code defining the synthesis instrument, in the high-level sound-object (such as a note-event), or in an intermediate layer between the two. In this paper we describe the implementation of such mechanisms in two composition systems, both implemented in Smalltalk-80: DMIX [Oppenheim, 1993a and 1993b] and E-Scape [Anderson, 1990 and 1992]. Both systems enable users to define the perceptual aspects within a sound that they are interested in and then control it via a score; DMIX also enables real-time control.

E-Scape implements the first approach to perceptual mapping, having 'Instruments' which possess an intermediate PspProcessor object within which users can define perceptual parameters. Perceptual data from high-level ScoreEvents is then processed into data for input to synthesis structures. The relationship between perceptual and device parameters is defined by writing (programming) short scripts. This approach requires E-Scape to incorporate the instrument definition of synthesis structures. Hence it will be inappropriate for other systems which do not incorporate such definitions.

DMIX employs an abstract OneToMany object that can be placed in a note-event for real-time control, in various intermediate software levels, or even used merely as a compositional tool for its mapping capabilities. If incorporated into a note-event, there will be a OneToMany for each perceptual parameter. Whenever the Perceptual Parameter changes its value, each of its outputs will update the appropriate synthesis parameter in real-time. The user can specify the perceptual parameters interactively via an interface that provides real-time audio and graphical feed-back.

Implementation in DMIX—PeRRY

PeRRY is a simple mechanism that maps one (perceptual) input into any number of outputs. Each time PeRRY receives an input it calculates a new value for each of its outputs and updates it. The transform function between the input and each output is unique and can be a Function (table), DMIX Modifier (much more flexible than a table), or Smalltalk BlockClosure (i.e. any algorithm expressed in Smalltalk syntax). A graphic interface aids in interactively defining and testing each output individually. Thus, the composer may vary several states in the perceptual entity (a process we term *teaching),* and

PeRRY will then interpolate to derive intermediate values. Another important feature is that each output can *itself* be a OneToMany, allowing the construction of hierarchies of transformations.

Design and User Interface

Figure 1 displays a graphic editor that can be used to define, set and test a PeRRY. It is implemented in DMIX via two classes: OneToMany and PhysicalNode. The OneToMany has two instance variables: a *value* - this is the perceptual parameter - (1, numbers in brackets refer to the numbers in the two figures) and a *net* (2). The *net* is merely a collection of PhysicalNodes, one for each synthesis parameter. Each PhysicalNode has three instance variables: *name*, *mapper*, and *setEvent*. The *mapper* maps a given perceptual value into the synthesis parameter in the *setEvent*. The mapper can be a table, a Function (6, 7), or a user-defined algorithm. It can be edited by clicking on the button. The *setEvent* updates the synthesis device as soon as it receives a new synthesis parameter. Typical types (classes) of *setEvent* are MidiSystemExclusive, MidiController, MidiNote, DSPNote, or DSPUpdate. This arrangement allows for interactive testing and setting of each PhysicalNode, provided that real-time synthesis is available.

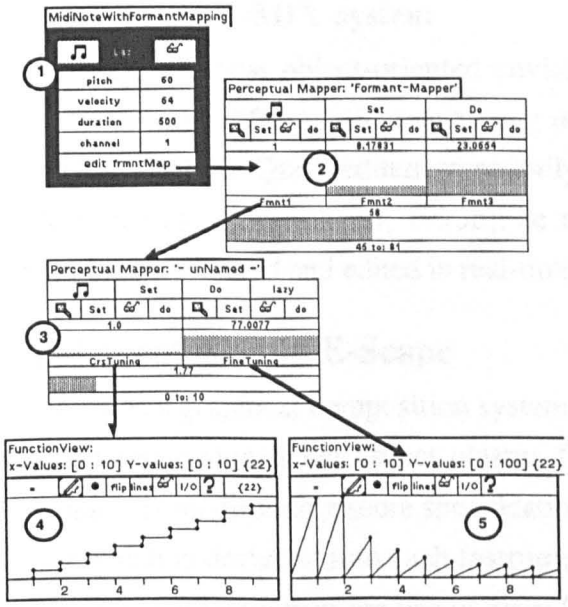


Fig 1: The user interface

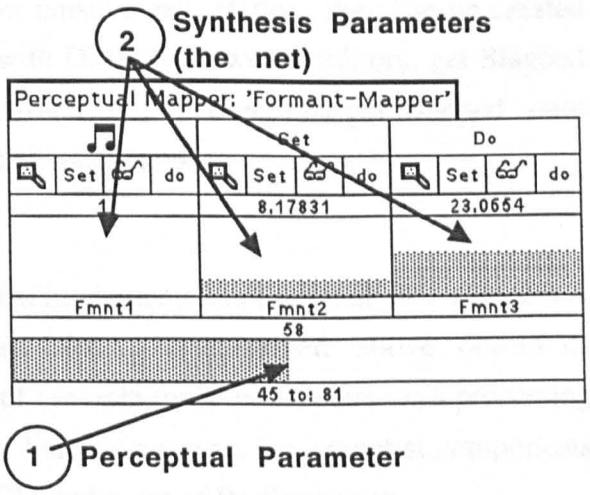


Fig 2: Mapping pitch to formant frequencies

The user can set the perceptual parameter value by moving the slider (1). As he does so the sliders of each PhysicalNode in the net also moves to display their new synthesis parameters, and sound is produced to provide real-time feedback. After setting a perceptual parameter the user can fine-tune each PhysicalNode; when satisfied with the result he can click on the 'set' button. We term this process *teaching the net.*

Example: Formant Synthesis in MIDI

Figure 2 is an example of using a PeRRY to implement formant synthesis of the human voice via MIDI on a Yamaha SY-77 synthesizer. We choose this example because it is

extremely hard to do via MIDI. The human voice is synthesized by having three formants at fixed frequencies. In FM this is can be modeled with three carriers, that emulate the three formants, and one modulator that determines the fundamental frequency. The modulator's frequency is determined by the MIDI key number and the trick is to have the carrier frequency-ratio be inversely proportioned to the fundamental frequency so that formats stay at fixed frequencies. Needless to say, the Yamaha SY-77 does not allow such a flexible frequency setting of the carriers so this method has a limited acceptable range of roughly 1 to 4 semi-tones. Using PeRRY we were able to extend the range extensively by having a OneToMany calculate a frequency-ratio for each formant in relation to the note-event's pitch (MIDI key-number). PeRRY makes the needed calculations and sends SystemExclusive messages to set the frequency-ratio of each carrier before the actual Midi-note is played. Note that here the *setEvent* in the first OneToMany (4) is itself a OneToMany (5) that automatically deals with the hardware requirement to set the carriers' frequency-ratio via two parameters: coarse and fine frequency. The first OneToMany (4) calculates the desired frequency-ratio and the second (5) brakes this value into the two SY-parameters and sends the correct SystemExclusive messages. The transfer functions are numbered 6 and 7.

PeRRY in the DMIX system

As DMIX is a true object-oriented environment, using music-events with perceptual parameter is no different than using any other music-event. Hence, they can be created algorithmically in Quill, edited graphically with DMIX's powerful editors, get Slapped onto Functions [Oppenheim, 1993b], be modified by Functions that get Slapped onto them, be transformed and edited in real-time, and much more.

Implementation in E-Scape

E-Scape is a graphical composition system also implemented in Smalltalk-80. Hence it is able to reuse significant object classes from DMIX. As described above, one of its emphases is on providing score specification of multiple sonic parameters via a processing layer which is designed into each Instrument. For this purpose, the essential components of an E-Scape Instrument are one or more DCTs and a set of PspProcessors.

A DCT is a specification of one or more 'device-level' synthesis modules (processes equivalent to a Music-N unit generator) on a single device. Each synthesis module can be controlled independently. Examples of such modules would be a 'UGP' in the MIDAS synthesizer [Kirk, 1990] or a 'VOICE' in a Yamaha SY77 synthesizer. A DCT can be constructed by a user out of higher-level modules, which are then unravelled hierarchically.

Each PspProcessor specifies one or more perceptual parameters which are available to a composer for scoring, and translate these into 'device-level' synthesis input parameters within a DCT.

An example Instrument could contain three DCTs and two PspProcessors. The DCTs could contain ten MIDAS 'UGP', two D110 'PART' and four SY77 'VOICE' modules

respectively. In this example, one PspProcessor provides two (interelated) Perceptual Parameters ('pitch' and 'random spread') and the other provides one ('energy'). They will then translate values of these parameters to various DCT inputs (which are connected to various inputs of their device-level modules).

A PspProcessor has *inputs*, *outputs* and a *userProcess*. Its inputs effectively describe the Perceptual Parameters which a composer may specify for a score event that uses this Instrument. Each input defines the range and default value of a parameter as well as providing one or more different units which can be selected by the user. Each output of the PspProcessor connects to one or more inputs of a DCT.

A PspProcessor's *userProcess* contains user-entered Smalltalk source-code which is compiled to a code block (see example below). The code can include specialized system functions which either provide dedicated functions (such as reading data from one or more perceptual parameters of a score event, and assigning data to an available input of a DCT), or utility functions for analyzing or processing data (eg find the integer center value of a set of points, or quantize a value to the nearest available on the device). New functions can also be created by a user.

Creating ScoreEvents using an E-Scape Instrument

A new Score Event is constructed using a specified Instrument. Each PspProcessor in the Instrument creates a holder to contain 'input' Perceptual Parameter values (which may be time-varying). It also creates a holder to contain 'output' processed data assigned to the input of a DCT.

When Perceptual Parameter values are entered or changed in the score, the PspProcessor calls its *userProcess* block, sending itself as a block parameter (:parameterHolder). The block can then access the 'input' Perceptual Parameter data, and 'output' DCT inputs.

Example: Pitch processing by a PspProcessor

The Smalltalk code entered by a user for a simplified example *userProcess* is given in figure 3. The PspProcessor has two Perceptual Parameters named 'pitch' (whose basic units are 'semitones from A440'), and 'detuning spread' (whose basic units are cents). It has three connected DCT inputs (named 'BENDER RANGE', 'pitchbend amount' and 'st. pitch') which are each connected to several modules within a device.

Each input parameter may be time-varying, thus whenever either Perceptual Parameter changes, the block calculates the absolute pitch by adding the 'pitch' value to the 'detuning spread' value / 100. It then determines the integer mean value, quantized to the nearest allowable value of the DCT input above this on the device. This value is then loaded (with time = 0) to the output data holder for the 'BENDER RANGE' DCT input. Similar processing results in values which are loaded to holders for the other DCTInputs. When the score is played, this data will be sent to the appropriate modules within the DCT as instantiated on a device (inserted within the assigned message for that input of that device).

```
NB. Temporary block variables commence by convention with lower case 't'

[:pspProcessor |
tPitchFunc <- (pspProcessor pointsForPerceptualParameterNamed: ('pitch')).
tDetuneFunc <- (pspProcessor pointsForPerceptualParameterNamed: ('detuning spread')).
tPitchPoints <- pspProcessor allTimes collect: [:t |
    t @ ((tPitchFunc valueAt: t) + ((tDetuneFunc valueAt: t) / 100) ) ].
tCentreVal <- EFunction integerCentreOf: tPitchPoints.
tDeviation <- EFunction integerMaxDeviationOf: tPitchPoints fromCentreValue: tCentreVal.
tPBSensitivity <- (pspProcessor outputNamed: 'BENDER RANGE')
                    nearestDeviceValAbove: tDeviation abs.
pspProcessor removeAll.
pspProcessor loadVal: tPBSensitivity toDCTInputNamed: 'BENDER RANGE'.
tPitchPoints do: [:eachPt |
    newPt <- eachPt processVal: [:val |
        ((val - tCentreVal) divBySafe: tPBSensitivity) * 63 + 64 ].
    pspProcessor loadPoint: newPt toDCTInputNamed: 'pitchbend amount'].
pspProcessor loadVal: (tCentreVal + 69) toDCTInputNamed: 'st. pitch']
```

Figure 3: code entered in a *userProcess* for pitch-processing

Conclusions

The implementations in both DMIX and E-Scape enable composers to specify complex timbral changes using musically meaningful Perceptual Parameters. Each approach has strengths and limitations.

E-Scape's approach to specifying the process between score parameters requires the user to type code (albeit a restricted subset). However it allows quite complex algorithmic mechanisms to process, if desired, several Perceptual Parameters which interact; to analyze time-varying score parameters, and interrogate device specifications.

The approach in DMIX whereby the user 'teaches' a OneToMany is more intuitive, far easier to use and provides real-time audio and visual feedback. It is especially useful for specifying mappings which are more empirical (eg the SY77 example above) and which would be highly laborious to enter in the E-Scape implementation. However, the DMIX implementation supports a less flexible relationship between inputs and outputs. The ability

to replace the function mapping with a code block facilitates more complex mappings, but still assumes a tree structure starting with a single Perceptual Parameter, and provides no access to device input settings or to other Perceptual Parameters.

It is planned to incorporate the DMIX OneToMany object into the E-Scape PspProcessor object, in order to use the more intuitive DMIX implementation in cases where there is only a single Perceptual Parameter to be processed. Parameter data can be sent to a DMIX OneToMany owned by the PspProcessor. Each of its bottom-level PhysicalNodes could then have a DCT input as its *setEvent* and load mapped data into the corresponding holder. The task of creating *userProcess* blocks in E-Scape will also be made easier with a menu-driven interface allowing functions, input and output names to be selected by the user.

Bibliography

[Anderson, 1990] Anderson, T.M. E-Scape: An extendable Sonic Composition and Performance Environment. Proc. ICMC Glasgow 1990. ICMA.

[Anderson, 1992]. Anderson, T.M., Kirk, P.R. Electroacoustic Scoring with Phase-vocoding Instruments using the E-Scape composition system. Proc. ICMC San Jose 1992. ICMA.

[Kirk, 1990] Kirk, P.R., and Orton, R. MIDAS: a Musical Instrument Digital Array Signal Processor. Proc. ICMC Glasgow. ICMA.

[Oppenheim, 90] QUILL: An Interpreter for Creating Music-Objects Within the DMIX Environment, Proc. ICMC, Montreal, Canada.

[Oppenheim, 93a] Oppenheim, Daniel V. DMIX—A Multi Faceted Environment for Composing and Performing Computer Music: its Design, Philosophy, and Implementation. Proceedings of the SEAMUS Conference, Austin, Texas; also in proceedings of the Arts and Technology Symposium, Connecticut College, Connecticut.

[Oppenheim, 93b] Oppenheim, Daniel V. Slappability: A New Metaphor for Human Computer Interaction. Proc. ICMC, Tokyo, Japan.

Electroacoustic Scoring with Phase-vocoding Instruments using the E-Scape composition system

Internation Computer Music Conference, San Jose, USA. 1992

Tim Anderson
Music Technology Group
University of York
YORK YO1 5DD UK
tma@ohm.york.ac.uk

Abstract

This paper presents the compositional possibilities for scoring using E-Scape [Anderson 90] instruments which incorporate phase vocoding and complex manipulations of phase-vocoder analysis files. These are presented in the same manner as instruments incorporating synthesis or other sound manipulation processes.

An instrument can incorporate graphical displays of the resultant sound spectrum at various stages within it. These are provided by using various frequency spectra display modules in the University of York MIDAS system [Kirk 90] , at present running on an Silicon Graphics Indigo node.

An illustration will be presented of a phase vocoded analysis file processing instrument, and graphical score which uses it. The composer is provided with a direct visual mapping of the score to the processing parameters used in each event, within the E-Scape graphical electroacoustic scoring system.

Spectral manipulation techniques involving the processing of phase vocoder analysis files have been successfully employed in electroacoustic composition (Wishart 88).

These files consist of amplitude and phase values in a number of frequency channels, in each of a number of time windows or frames.

To manipulate such files, composers have either had to specify command line parameters to their own (or others') processing programs, or use the spectral data type unit-generators in CSound version (xx). These enable :

- the imposition of a (static) frequency spectrum envelope;
- the (proportional) mixing of two spectra (again static);
- the differentiating of successive frames;
- the display of the analysis data;
- the filtering of each channel;

These additions to CSound are welcome, but suffer the disadvantage of being relatively high-level operations, as necessitated by the need for CSound to be highly optimised for speed on a single processor. Also not surprisingly, all control parameters are i-rate.

In the E-Scape composition system, [Anderson, 90] a composer can design instruments graphically from modules, which when unravelled, specify a network of primitive processes on a variety of synthesis systems. One such is the MIDAS system under development at the University of York [see companion papers.....92, Kirk & Orton 90]. This enables a network of primitive processes ('UGPs') to run on an arbitrary number of processors, which communicate data via a LAN.

Several UGPs (primitive processes) are under development to enable windowing, FFT analysis and resynthesis processes to be constructed by a high-level user. UGPs to perform low-level data processing operations are also planned, capable of accepting continously variable control data. Thus a composer can design an E-Scape instrument which processes analysis files in arbitrarily complex ways (and converts them back to time domain sound).

The E-Scape system also allows time-varying control (via the graphic score) of any allowable input parameter of an instrument.

Each instrument parameter can have a score pattern which then provides visual feedback about the sound manipulation. Frequency domain displays of the data at any stage in the signal path can be specified, by linking in display UGPs running on a MIDAS graphics node.

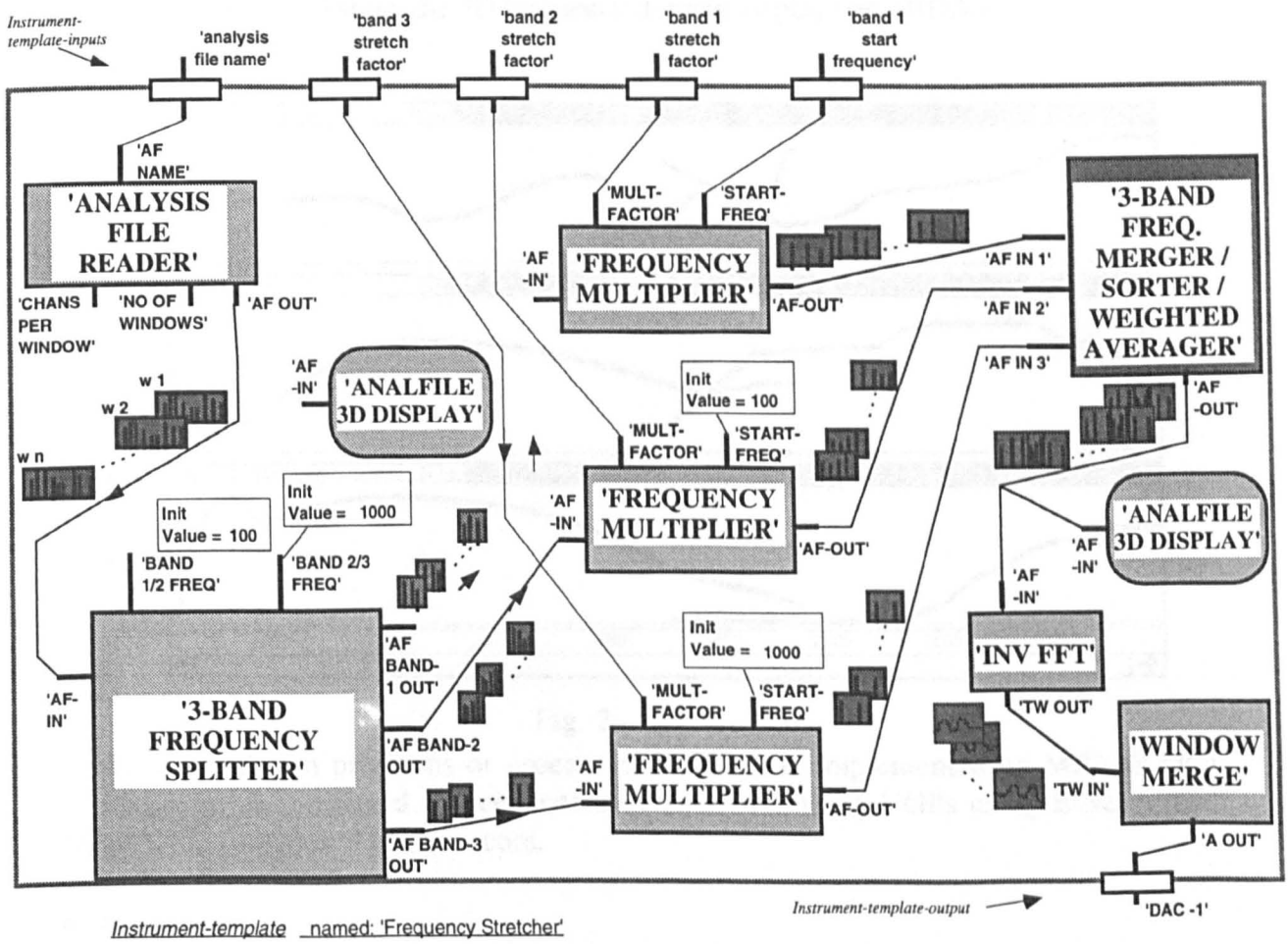


Fig. 1

A simple example instrument is shown in fig 1, with the signal path running from top left to bottom right. An analysis-file name can be specified from the score in this case, providing a crude capability to migrate from one sound to another (the smoothness depending on the window overlap). Each window of data is then split into three frequency bands, which are then independently processed.

Each band is 'stretched' by multiplying each phase value by a variable factor specifed from the score. If the resulting phase exceeds $+\pi$, it wraps round to $-\pi$ and also moves to the next frequency channel. A frequency below which no stretching will be performed is also specified, with band-1 able to have this parameter controlled from the score.

The top right module performs an average (weighted according to amplitude) of the phase of any spectral components which exist in the same channel. This is necessary as components in band-1 may have been multiplied so as now to occupy one of band-2's channels, in which there still may exist a component. Again the result may need to be shifted into another band if the resulting phase $< -\pi$, or $> +\pi$.

Finally inverse-FFT and window-merging UGPs produce output sound.

Data from the start and end of the processing chain is also specified to be sent to two 3D frequency domain display UGPs running on a MIDAS graphics node. The modules shown may be UGPs, or may be constructed out of simpler primitive UGPs, but in either case the composer may connect and use them as modules.

The simplified score example changes the stretch factors during the course of the three score events. The composer can see any time domain changes directly in the score, while a separate screen would show the 3D frequency domain displays on MIDAS.

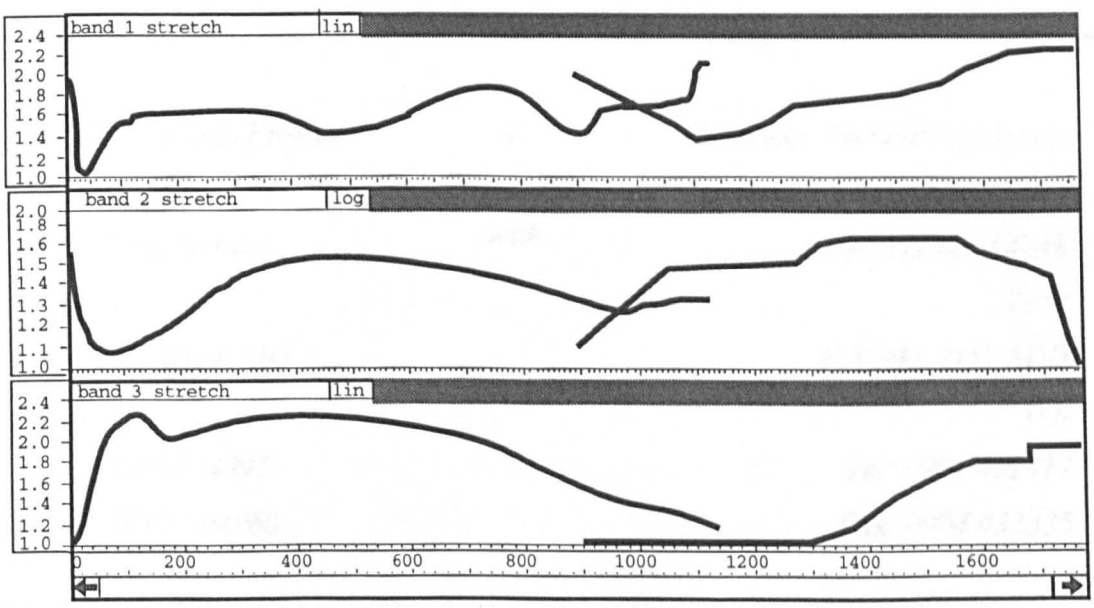


Fig. 2

Other manipulation programs or processes, can also be implemented on MIDAS by a composer in an integrated and consistent manner by linking UGPs using E-Scape and playing via a composed graphic score.

References

[Anderson 90] Anderson, T.M. "E-Scape: An Extendable Sonic Composition and Performance Environment" Proc. ICMC, Glasgow, 1990.

[Kirk 90] Kirk, P.R., Orton, R. "MIDAS: A Musical Instrument Digital Array Signal Processor". Proc. ICMC, Glasgow, 1990

[Wishart 90] Wishart, T, "The Composition of Vox-5". CMJ, Vol 12, No 4.

Appendix 4: E-Scape user-interface development design study

- a proposal by a five-partner consortium for the E.C. TIDE Project

Tim Anderson
tma@ohm.york.ac.uk

Drake Research Project
3 Ure Lodge
Ure Bank Terrace
Ripon
N.Yorks HG4 1JG
UK
Tel: 0765 604993
Fax: 0765 604960

in cooperation with

The Music Technology Group
Department of Electronics
University of York
York
N.Yorks YO1 5DD
UK
Tel: 0904 432417
Fax: 0904 432335

Overview

It is proposed to develop a Customisable User Interface Construction System to enable a person with any degree of physical disability to control a state of the art Computer-based Music Production System with high productivity.

This Music Production system allows a user to create music of any degree of complexity, or control recording studio technology. Users of the system will be able to participate fully in commercial music production and recording, and gain employment opportunities in the creation of synthesised music for the advertising, film and corporate film market, as well as an the opportunity to develop a career as a composer.

The User Interface System will be tested in active use with the Music Production System by a number of severely physically disabled composers involved with this project. Although having a track record of achievement in producing music for public performance, these composers currently find it impossible to achieve comparable productivity to able-bodied composers because of their constrained physical abilities.

Production methods in use in the commercial music environment (in which these composers are aiming to compete), use computer systems to control composition, sound creation and recording processes. State of the art software for such computer systems without exception uses graphic WIMP interfaces to allow the user to cope with the complexity of the task.

Software used in other employment areas is also increasingly following this path, but composition systems provide a paradigm of a complex system in which many different kinds of tasks need to be performed, on data structures which can be elaborate, yet arbitrary.

The Interface System can also be used with other suitable structured Application Software. This structure is also proposed as a standard, such that disabled users will have access to all kinds of Application Software and be enabled to participate fully and productively in all work areas.

The Interface Construction System

This Interface System is an innovative new development which will interface with the existing state of the art Composition system (see below). It addresses the accuracy and productivity problems which disabled users face in controlling and actively using software with existing disabled interfaces

Disabled users of existing systems face two main problems which this System will remedy:-

Problem (1)

There is a mismatch between the physical actions which are needed to operate a music production system, and the type of controlling movements a disabled user is able to make. The movement capabilities of a disabled user are very different to those of an

able-bodied user in terms of timing accuracy, positional control, speed and repeatability

Present bolt on interface systems for disabled computer users employ conventional disabled access aids, eg single switch overlays, head pointers, macro recorders, 'sticky keys', or adapted tracker balls with scaling. However, they attempt to help a disabled user to conform to the kinds of input required by the computer (eg pressing combinations of keys, moving a mouse pointer) rather than adapting to the types of control which a user may more easily make.

For example, to carry out a relatively simple task, a software system may require a user to hold a key down, while also holding a mouse button down, and moving it precisely from one point to another, keeping it steady while then releasing both other keys.

On the other hand the user may be able to reliably move the mouse into perhaps two or three zones on the screen, and be able to only hit a group of keys on the console. Communication is still possible but needs a more elaborate mapping than provided by present interface adaptation systems.

Solution to problem (1)

The system intelligently processes movements from any variety of user actions to create one or more 'Virtual Switches'. These can then be used to control any aspect of any stage of an activity... (more details below)

Problem (2)

The mode of operation of the system under control requires a user to initiate a large number of actions, each of which may have many component actions. For physically disabled users, each control action can take a long time and consume much effort, and it is easy to accidentally select an undesired option. There is thus great scope for losing track of what one is doing, and an accidental movement at any stage may involve starting the whole activity again. The whole way of working with a complex system with infinite combinations of operations possible and many repetitious operations often required, may be inappropriate for a user to whom each movement is difficult to control.

Solution to problem (2)

What is need is some kind of structure for actions, such that a user can initiate a selected activity, exercise as much choice as is needed at any stage of that activity, but remain safely 'locked in' to it; at any point only legitimate actions and decisions are possible.

The proposed Interface Construction System is more integrated with the host system than a simple overlay. It can access quite primitive operations within the system directly, rather than having to attempt to control it from the surface in the same way an (able bodied) user would.Thus it can construct its own ways of ordering actions within the host, and provide complex sequences of operations, termed Activities, but with user control and choice at any stage.

This system is thus ideal for disabled users, as a large number of actions can be selected and carried out by stepping through actions and choosing options using only a few control inputs - minimally with only a single switch for input. A disabled composer can try out and control complex or repetitious processes quickly and easily.

It is also important that the activity structure is editable by users themselves. They can construct their own activities which require little user input apart from the crucial high-level compositional decisions. If a user's motor skills improve with practise, activities can be expanded to require more control inputs (with the resulting increase in navigational flexibility). As a user's compositional experience increases, they can adapt or replace activities to allow for more user choices.

The 'E-Scape' Composition System

This system (called 'E-Scape') has been designed and developed at the University of York over the last four years. It allows a user to enter and manipulate musical scores which are then able to be played on electronic synthesised[1] instruments, which are now of sufficient quality to emulate acoustic instruments, if desired, or go beyond their limitations to create evocative soundscapes. Most of the music heard in advertising, and a large proportion of that used in film sound-tracks for example, is produced entirely using synthesiser-generated sound. Alternatively, or in addition, a conventional printed score can be produced and played by musicians using acoustic instruments.

Most of the functionality of existing 'state of the art' composition software is included, such as entry of notes (step or live recording, or drawing), editing (cut, paste, copy), altering note or continuous instrument parameters, building up hierarchical structures, and playing on synthesiser devices.

In addition, a composer can specify a number of musically meaningful sonic parameters for each musical event. S/he is able to specify timbral details if desired, but remains insulated from the specifications of synthesis devices in use and details of the sound producing arrangement within them - a composer does not also have to be a sound designer, only a sound controller. Instruments can be constructed which are distributed over many different devices if desired, allowing complex (and hence musically interesting) sounds to be produced.The same score can be realised (played) on many different devices, allowing music to be less dependent on particular devices, and hence still playable in years to come. New synthesis devices can also be accommodated and added to the system.

Thus the 'E-Scape' system as a whole is highly flexible - capable of supporting many different kinds of compositional activity. It is also adaptable to new developments - capable of using many different kinds of synthesiser device. It is thus an ideal basis on which to build the proposed Interface Construction System.

---

1 'Sampled' sounds are also subsumed into the term 'synthesised'

Outcomes of the project

The main outcome will be the existence of an advanced Music Production (Composition and Sound Creation) Software System, in active use by disabled composers. They will be controlling the system using the Customisable User Interface System described herein. Both the composition system which will be fully tested, and evaluated for

In addition to this, the Interface System will be usable with other Application Software (eg Desktop Publishing, Film Editing, Financial Management, Information Retrieval systems). This is possible, as the Interface System (and the Music System) are implemented in an Object Oriented software environment. Thus, appropriate changes to any suitably structured Application System can allow it to be controlled in the same manner. This uniformity of use interface, no matter what the underlying interface of the controlled system, is a great advantage, as any user proficient in its use can immediately start to use another, otherwise disparate system. Thus the Interface System will be available for use with other software, either directly (within the Smalltalk-80 environment in which it is written) or as a design specification which can be implemented on other platforms.

It is proposed that this Interface System design, and the structures implemented by Application Software to be controlled by it should be evaluated by the appropriate bodies and developed as a Europe-wide standard for Software Interface Design. The Standard would consist of two main elements:

i) The structure of the Interface Construction System should be the standard specification for similar systems implemented on other appropriate software and hardware environments in order to be connectable to state of the art software in all areas, making all such systems available for high-productivity work by disabled people.

ii) The low-level access provided by the example Music Production System to all aspects of its functioning should be a standard requirement for all such software, such that it can be used with the Interface Construction System.

Summary

Composition using a computer system requires a user to perform many complex tasks or actions which manipulate musical data. These tasks may involve repetition and decision-making at each stage.

The design, uses and implementation of a Customisable User Interface System is now described. The System allows a user to structure related tasks into coherent activities - sequences of actions to achieve specific goals. The Interface System is initially described in the context of controlling a computer-based composition system, as an excellent example of a highly complex and multi-faceted application environment.

Users can undertake compositional processes and make musical decisions at a high or low level by choosing an appropriate activity, within which options and paths of action (which may be cyclic) may be selected.

In addition to its primary goal of providing access to any kind of disabled user, the system facilitates a wide variety of other usages (with corollaries in other types of Application System) :

- Composition students can be guided through a compositional process, progressing to activities which demand more decisions, or which use the methodologies of a particular composer.
- Composers can construct their own activities which reflect their own preferred working methods.
- People with learning difficulties can enjoy creating music, within a restricted 'safe' environment.
- Blind users are enabled to control a (WIMP) composition system (as all interaction may be via non-mouse input).

Introduction: Activities as structured actions

Composition using a computer system involves the execution of a plethora of compositional actions to facilitate the entry, processing, editing, and playing of musical data. A user of such a system needs to perform a large number of different data manipulation and control actions. These actions may be complex, consisting of many repeated operations, with choices needing to be made at each stage.

This user interface construction system is designed to overlay the existing 'E-Scape' composition system (Anderson 1990, 1992) and allow a composer or teacher to structure related compositional actions or actions into coherent activities.

Within an activity, a user (who may be an expert or novice composer) may step through component actions in order, or select different paths (branches, repetitions, jumps etc) through the activity. Each action within the activity may also present a user with choices, (which may be deliberately limited to a greater or lesser extent). An action could actually be another activity, hence activities of arbitrary complexity can be constructed hierarchically.

Example uses

The User Interface System, when used to control the Composition Application System, has several other modes of utility, which are provided in addition to the provision of disabled access:-

1. Music educators can construct sets of activities which guide a student through a selected compositional process. Students can thus learn by observing and participating in an active process. Activities can be more or less prescriptive - allowing varying degrees of user control and choice.

(a) Students can start with 'beginners' activities where there is little user choice required. Activities with **no** choice at any stage could also allow a student to merely step through the process, observing the effect of each action.

(b) As students gain more experience and confidence, they can gradually progress to successive activities which allow them to exercise more control - having to make more compositional decisions within each action, and having more freedom to navigate between actions within the activity.

(c) At a more advanced stage, a composition student may wish to investigate or use the methodologies of a particular composer under study. Suitably designed activities can allow a student to compose within the framework of a particular compositional methodology while allowing partial or full decision making. Students can also customise their own environment as they progress; altering existing activities or constructing new ones.

2. A composer can construct activities which reflect his/her own preferred ways of working. A series of activities could be constructed to support each of the composer's compositional methodologies. Composers could also try out compositional frameworks supplied by others.

3. People with learning or perceptual difficulties, or diminished mental acuity (perhaps because elderly?) can still enjoy, and be stimulated by, the creation of music. A restricted 'safe' environment can be provided, with Activities designed which allow such a user to make musical decisions at a high level. Activities can proceed with little further user interaction; for example a user can select a melody from a library, choose the kind of accompaniment style or change the tempo and key etc. All decisions produce some kind of musical output, and hence maintain motivation and interest. Users can progress to more interactive activities as their skill and confidence grow.

4. Blind users are enabled to control a graphic (WIMP) composition system because all interaction with the system **may** be via non-mouse input (eg switch or keypress). Menu items can have sonic or vocal feedback, as can each basic system action (eg the basic action 'select next score event' could not only announce this via a speech synthesiser, but also play the event, and/or announce various parameter values). Some additional work is envisaged for this application.

Constructing an Activity

Many activities are provided as standard in the system, but the user is free to construct any number of additional activities, or modify existing ones.

An 'Activity' consists of a list of 'actions'. There are seven types of action:-

- a 'System Action' - a primitive operation which is built into the system and called by a specified name.
- another 'Activity' - ie another list of actions.
- a 'Jump' - move to elsewhere in the list.
- a Value Selector - allows the user to select a numeric value, via various menus, scroll bars, or typing
- a Text Selector - allows the user to enter text, either by 'normal' typing, or by scrolling a grid of letters
- a 'Switch Selector' - allows the user to directly select between two or more actions (of any type).
- a 'Menu Selector' - allows the user to scroll and select between actions displayed on screen as a menu. In addition, the performance of any of these actions can be made conditional on the value of a specified user variable.

The path taken through an Activity can thus be different depending on user choices from Menu or Switch Selectors, and the values of variable previously set.

Note that the process of constructing a new activity is **itself** an activity. The activity includes a Menu Selector from which a user can select System Actions such as 'Add new menu item', and further special Menu Selectors which display, for example, all possible Jumps in the present list, all available Activities, or all System Actions.

Controlling an activity: Menu Selectors and Switch Selectors

Menu Selectors and Switch Selectors facilitate user choice of actions (ie a path) within an activity. Text and Value selectors facilitate user specification of text or numeric values.

Selectors can be controlled by Virtual Switches and/or Virtual Controllers which exist in software. Both process incoming data from a variety of physical inputs. This data can be such things as MIDI messages (eg from keyboard notes, control wheels, pedals etc), any computer console key, the mouse or trackerball position and button state, and any type of data received into any other computer input port. These are various types of Virtual Switch (eg '1-way' '2-way', '3-way' or more) which can be set to a particular active states depending on the incoming data. A Virtual Controller produces values (with a range and increment) also derived from incoming data. A full explanation of the operation of Virtual Switches and Controllers is given later in this document.

A Switch Selector has one (or more) Virtual Switches, each state of which initiates an action. It can also optionally be displayed on the screen, with each action having a graphic

button which can be clicked on by the mouse cursor (as in a conventional WIMP system) to initiate the action.

A Menu Selector displays a menu of actions on screen, and may display submenus hierarchically if desired. It is typically controlled by either: (1) a 2-way Virtual Switch (one way scrolling down an item each time it is activated, the other selecting the current item). Switches with more ways can provide faster control (eg scroll either way, exit, cancel etc); (2) a Virtual Controller (with range ≥ the number of items in the menu) to scroll, plus a 1-way (or more) Virtual Switch to select; (3) a mouse, 'as normal'

There are various types of Value Selector and Text Selector employing menus, sliders, 2-D grids or dialogue boxes, under the control of mouse, Virtual Switches, or Virtual Controllers.

Note that most Selectors can be controlled by the appropriate Virtual Switches or Controllers and on-screen mouse actions at the same time if desired[1]. This facilitates use of the system by, for example, a composition tutor who is a disabled switch user at the same time as an able-bodied student using the mouse.

Interfacing to the host System: System Actions

Each System Action is a primitive operation (function) which the host system (eg the composition system) can perform. Each System Action has a name, and these names are the only part of the interface system which would need altering if building Activities for controlling a different software system.

Typical Activities in the E-Scape composition environment involve selecting, playing and editing events. Much of this functionality is effected via low-level System Actions which can get and set the values of various parameters via system supplied menus or dialogue boxes. These also use selected Virtual Switches or Controllers, either specified with global defaults or separately for different System Actions.

An Activity to edit the volume of a note, for example would first call another Activity to select the desired event, then a System Action to present a menu from which the user can select the desired parameter. Finally the user is given a Menu Selector from which to select a System Action to increment, decrement, set a value, play or exit. Detailed examples are given later.

Examples of some of the simpler System Actions are given below. The name of each System Action appears in menus when an Activity is being constructed.

---

1 As long as mouse actions are not being used as input data states for a Virtual Switch. If this is the case, then the mouse will NOT be simultaneously available for 'direct' control. The use of mouse inputs for a Virtual Switch can be turned on and off dynamically by a system flag (using another Virtual Switch to trigger a System Action which sets it).

System Action Name	Underlying function (unseen by user)
'Choose score'	*Current-Score* <- menu choose score
'Go to start of score'	*Current-Time-Position* <- 0
'Increment time'	*Current-Time-Position* <- *Current-Time-Position* + *Time-Increment*
'Choose Time Increment'	*Time-Increment* <- menu get time value
'Go to next event'[1]	*Current-Event* <- next event after *Current-Event*
'Go to previous event'	*Current-Event* <- next event before *Current-Event*
'Select event'	*Selected-Events* replace all with: *Current-Event*
'Deselect event'	*Selected-Events* remove: *Current-Event*
'Add selected event '	*Selected-Events* add: *Current-Event*
'Play event'	Play *Current-Event*
'Play'	Play from *Current-Time-Position*
'Set block start'	*Displayed-Block-Begin-Time* <- *Current-Time*
'Set block end'	*Displayed-Block-End-Time* <- *Current-Time*
'Clear block'	Clear *Displayed-Block*
'Delete block'	Delete *Displayed-Block*
'Copy block'	Copy *Displayed-Block* to *Block-Buffer*
'Copy block with filter'	Copy *Displayed-Block* to *Block-Buffer* using *Current-Filter*
'Paste block and replace'	*Displayed-Block* <- paste *Block-Buffer* starting at *Current-Time* (with replace)
'Paste block and insert time'	*Displayed-Block* <- paste *Block-Buffer* starting at *Current-Time* (with insert)
'Paste block and merge'	*Displayed-Block* <- paste *Block-Buffer* starting at *Current-Time* (with merge)
'Choose parameter'	Current-Parameter <- menu select parameter from: *Current-Event*
'Choose parameter from block'	Current-Parameter <- menu select parameter from: *Displayed-Block*
'Increment parameter'	Increment *Current-Parameter* of *Current-Event*
'Decrement parameter'	Decrement *Current-Parameter* of *Current-Event*
'Set flag x to True'	*x* <- Set system variable as: true
'Set flag x to False'	*x* <- Set system variable as: false

1 If there is more than one event at the same time, then each is selected in turn.If there is *no* next event (ie at the end of this score), or no events are present, then an 'error' / 'completion' state is reached, and the System Action within this forces an exit from the Activity it is in (however deeply-nested)

Example Activities

Two example Activities are illustrated: The first is at a relatively low-level, and is more useful to a disabled switch-user. The second more complex, and might be used by any composer. It also demonstrates the concept of building complex activities from more basic ones.

Example 1 - Activity named 'Select event- forwards from current position'

This example Activity allows the user to choose an event from the *Current-Score* starting at the current position. This event will then be 'selected' ie highlighted as the event on which any further operations will be carried out. Other more complex Activities can allow the user more choices (eg which score, which instrument), and/or allow selection of more than one event.

This Activity uses a *Switch Selector* named 'Yes or No' and a *Menu Selector* named 'Scroll or Select'. The *Switch Selector* has two options ('YES' and 'NO'), hence must be assigned a 2-way Virtual Switch (with each option assigned to an active state of the Virtual Switch). The *Menu Selector* also has two options (scroll-down and select), hence must also be assigned a 2-way Virtual Switch.

These *Selectors* can be defined in the system already to use a default Virtual Switch, but each user is likely to want to assign their own. For example, Dave may use the 'Clicked or not: mouse or notes' Virtual Switch (example 2, below), whereas Steve may use the 'Low or High: Console or MIDI' Virtual Switch (example 1, below).

In this example, the Virtual Switch named 'Clicked or not: mouse or notes' is used for both *Selectors*. For the *Switch Selector*, the 'YES' option is assigned to the 'Clicked' state, and the 'NO' option to the 'Not clicked' state. For the *Menu Selector*, a state of 'Not clicked' is causes the menu to scroll to the next item, and a state of 'Click' makes the menu performs the current action.

Note that these <u>same</u> *Selectors* may also be used within many <u>other</u> activities.

On initiating the Activity, it first moves to the next event after the current one and plays it. If the user then does <u>not</u> click, it continues to step forward to the next event and play it (by continually jumping back to step [1]). If the user clicks, a menu appears which then scrolls until the user clicks again. Depending on the option selected, it can then go back or forward just one event (and play it), revert to stepping forward, cancel, or finish. The current event will then be 'selected' (highlighted).

Activity Definition

Step Action

[1] System *Action:* 'Go to next event' (goes to first event at or after *Current-Time* if there is no *Current-Event*)

[2] *System Action:* 'Play event'

[3] *Switch Selector:* 'Yes or No'

option		action(s)
'NO'	->	*Jump:* [1]
'YES'	->	*Menu Selector* : 'Scroll or Select'

Menu item		action(s)	
'Move on'	->	*Jump:*	[1]
'Move on one'	->	*System Action:*	'Go to previous event'
		System Action:	'Play event'[1]
'Move back one'	->	*System Action:*	'Go to next event'
		System Action:	'Play event'
'Play'	->	*System Action:*	'Play event'
'OK'	->	*Jump:*	[4]
'Cancel'	->	*Jump:*	[END]

[4] *System Action:* 'Select Event'

[END]

NB1. A menu action (if not a Jump) returns when completed to continue the operation of the menu it is held by.

NB2. A menu 'cancel' option can be provided by having a menu action which is a jump to the next item after the menu.

---

1 On successful completion, a System Action returns to the menu it is held by (if any). Thus the only way to exit a menu (apart from an error/completion condition - see above) is to have a menu action which is a Jump.

Example 2 - Activity named 'Copy block with conditions'

For illustrative purposes, the activity has been constructed from relatively many steps. In reality, many of these would be nested within other Activities.

Steps 1-5: Choose a (source) block from which events are to be extracted. In this example it is defined by all those events between two time points, but another Activity allows conditional loading of events to a block.

Step 6: Build or load a 'condition filter'. This is an E-Scape entity which facilitates the conditional copying of events whose parameters meet the conjunction of conditions in the filter.

Step 7: Copy events from the block using this condition filter.

Step 8: Choose destination score and time within it.

Step 9: Paste copied events into a chosen score starting at the chosen time.

Activity Definition

Step	Action		
[1]	*System Action:*	'Choose score'	
[2]	*Activity:*	'Select time - from start'	
[3]	*System Action:*	'Set block start'	
[4]	*Activity:*	'Select time - from current position'	
[5]	*System Action:*	'Set block end'	(a block is now defined)
[6]	*Activity:*	'Get selection filter'	(load from library, or construct new one)
[7]	*System Action:*	'Copy block with filter'	
[8]	*Activity:*	'Select score and time'	
[9]	*System Action:*	'Copy block'	

Example 3 - Activity named 'Select score and time'

[1] *System Action:* 'Store System Variable: *Current-Score* as: temp variable: *S*'

[2] *System Action:* 'Choose score' (this is now *Current-Score*)

[3] If (*S* = *Current-Score*)

Activity: 'Select time - from current position'

Else

Activity: 'Select time - from start' (this is now *Current-Time*)

Functionality of Virtual Switches

As described above, a Virtual Switch has an action associated with each of its active states, and is initiated when that state is achieved. It is set to a particular state by processing incoming data of any type (eg Midi, Ascii, Mouse) received. Each 'active' state of the Virtual Switch has an 'activating condition' which tests for the presence of one **or more** particular input data states. Each data state consists of a particular value (or range of values), for a particular data field of a particular data type on a particular input port. For an activating condition to be met may require several input data states to all be present, or just one, or some other logical combination of input data states.

Virtual Switch Example 1

A 2-way Virtual Switch (named 'Low or High: Console or MIDI') has two active states (labelled 'High' and 'Low'). The 'Low' state for example has an 'activating condition' which will be met if **any one** of the following input data-states is present:-

'Low'	***Data State or***	***Data State or***	***Data State or***	***Data State***
port	console	console	console	MIDI
data type	ascii	ascii	ascii	Midi controller
field	-	-	-	channel
min value	'a'	'q'	'v'	1
max value	'h'	't'	'z'	2
field				controller no.
min value				1
max value				(1)
field				controller value
min value				0
max value				63

Activating Condition for the 'Low' Switch State

If any of these four data states are present (ie a console keypress between ascii 'a'-'h', 'q'-'t', 'v'-'z' or a MIDI mod wheel controller message on channel 1 or 2, with value below 64) then the activating condition will be met and the 'Low' active state will be triggered.

As seen by the user, the 'Low' state is triggered by pressing any of the letter keys on the left side of the computer console (to the left of the line between the 'Y', 'H' and 'B' keys), or moving a MIDI modulation wheel (on channel 1) to anywhere below its half way (64) position. The 'High' state (not shown) would be triggered for example by pressing any of the letter keys on the right side of the console (to the right of the line between the 'U' and 'M' keys) or moving the modulation wheel to above its half way position.

Note that 'activating conditions' for different active states of a switch may be simultaneously met, so that a Virtual Switch could be in several input states at once (an analogy would be a 4-way joystick in its N-W position, which will then be in its 'N' and 'W' states).

Thus each active state of a Virtual Switch has an associated 'activating condition' which may or may not be met at a particular time. Other parameters then determine how it responds to this and which 'active state(s)' it is actually then set to.

Gate times

Each active state has a user-specified 'gate time'. This is the time during which, once a state is activated, it cannot be **re**activated (by the continued presence of the 'activating condition'). The Switch **can** be reset (ie so it is again able to be triggered to an 'active' state) before its gate time has elapsed, if the activating condition becomes false. Note that if the gate time is set to ∞, then the activating condition for that active state **has** to go false it can be triggered again.

For example, both 'Low' and High' states of the Virtual Switch above could have a 'gate time' of ∞. When a user presses one or more keys on the left of the console the Virtual Switch would go to state 'Low' [1] (and trigger whatever action is associated with this state, whether an action at top-level, or within a Switch or Menu Selector). However, because the 'gate time' = ∞, the Switch cannot be triggered to active state 'Low' **again** until all keys are released, thus the Switch will **not** go on again if keys are continually held down.

Virtual Switch Example 2

Another 2-way Virtual Switch (named 'Clicked or not: mouse or notes') has two active states (labelled 'Clicked' and 'Not clicked') and a gate time of 1s. The 'Clicked' state has a gate time of ∞ and an 'activating condition' which will be met if the following input data-states are present:-

---

1 After going to a particular state , and triggering some action, that state is flushed and will not trigger again.

'Clicked' *Gate time = ∞*	***Data State*** *or*	***Data State***
port	mouse	MIDI
data type	button	Midi note on
field	left	channel
min value	#down	1
max value	(#down)	(1)
field		pitch
min value		37
max value		127

Example 2: Activating Condition for the 'Clicked' Switch State

Thus if a MIDI note on channel 1, with pitch above 37 is pressed, or the left mouse button is down, then the 'Clicked' state is triggered. As the 'gate time' = ∞, then the 'Clicked' state can only be reactivated after the activating condition becomes false (ie the mouse button is up and no note above 37 is pressed). Thus holding the mouse button down only triggers the Virtual Switch to the 'Clicked' state once.

The 'Not clicked' state has an 'activating condition' which will be met if the following input data-states are present:-

'Not clicked' *Gate time = 1s*	***Data State*** *and not*	***Data State***
port	mouse	MIDI
data type	button	Midi note on
field	left	channel
min value	#up	1
max value	(#up)	(1)
field		pitch
min value		37
max value		127

Example 2: Activating Condition for the 'Not clicked' Switch State

Thus if a MIDI note is not being pressed, and the left mouse button is up, then the 'Not clicked' state is triggered. After the 'gate time' of 1s has elapsed, then the 'Not clicked' state can be reactivated if the activating condition is still met (ie the mouse button is still up and no note above 37 has been pressed). Thus if the user does nothing, this Virtual Switch will continually trigger in state 'Not clicked'. If a Menu Selector uses this Virtual Switch, with scrolling down triggered by this 'Not clicked' state, then the menu will scroll down once a second.

Toggle Switches

If several active states have the same 'activating condition' then each time the condition is met, the Virtual Switches activates alternate states in a cycle. Thus, for example, a Virtual Switch could have two active states 'ON' and 'OFF' which are both activated by the 'activating condition' that MIDI notes below 36 are pressed. This Switch will then act as a toggle switch, switching between 'ON' and 'OFF' successively each time a note less than 36 is pressed.

Dynamic Activating conditions

The assignment of physical input data states to each 'activating condition' can be dynamic, ie a data state may be conditional on the value of some System Variable. Such a variable can be altered by a System Action which itself is triggered by another Virtual Switch. Thus for example 2 above, the MIDI note on data state could be conditional on whether a System Variable (named *UseNotes*) is set as true or false[1]. If this variable is true, then notes will trigger the Switch, otherwise not. Thus notes in this example could then be used for 'normal' note event music entry. The active state definitions for this Switch would then have "[if *UseNotes*]" appended to the last MIDI data state column.

Functionality of Virtual Controllers

A Virtual Controller has one or more input data state, usually only one of which will use at a time. Each state consists of a particular value (or range of values), for each data field of a particular data type on a particular input port. In addition,one of the fields is designated as 'variable', ie as determining the Controller's output values. The polarity can be specified as '+' (normal) or '-' to reverse the high and low ends of the range.

Virtual Controller Example

This Virtual Controller (named 'MIDI notes-1 or mouse up') produces output derived either from the mouse vertical position, or from the pitch of notes played on a connected MIDI controller[2], depending if the System Variable *UseNotes* is set true or false. In addition the value (position) of a MIDI modulation wheel (controller #1) on any channel[3] is read. In this example (a portable MIDI keyboard), the modulation wheel is mounted upside down, hence polarity is set to '-'.

---

1 System Variables are set by System Actions which themselves are triggered by a Virtual Switch, eg example 1.

2 This controller will typically be a keyboard, but may also be a MIDI guitar, wind controller, ultrasonic distance sensor (eg EMS Soundbeam), or other sensor (eg York University's 'MIDI creator' box)

3 Because any value (1-16) in this field is acceptable, it does not require specifying in the Virtual Controller definition.

	Data State *or* *[if not* **UseNotes***]*	*Data State* *or*	*Data State* *[if* **UseNotes***]*
port	mouse	MIDI	MID
type	position	Midi controller	Midi note on
polarity	+	-	+
variable field	y	controller value	pitch
field	y	controller number	channel
min value	0	1	1
max value	864	(1)	(1)
field		controller value	pitch
min value		0	37
max value		127	72

If MIDI notes are in use (*UseNotes* = true), then pitches between C#1 (note 37) and C4 (note 72) produce 36 values, which is thus its maximum range. This range will be scaled down to suit the range of values required in the Menu or Value Selector (eg) it is assigned to. For example if this Controller is used by a Menu Selector with 8 items, then its output will automatically be scaled by the Menu.[1].

Note that this Virtual Controller cannot be used with a Menu Selector of more than 36 items (unlikely!) or a Value Selector with range more than 36 (although it could be if the third 'MIDI note on' column were omitted).

---

1 In the object oriented software environment, the Menu asks the Controller for its max and min values, and then can scale incoming data appropriately. The Controller does not perform this scaling itself, as it may be used by many different Menus etc which require different scaling factors

Initiating an activity

The system allows a user to initiate an activity in a variety of ways.

A graphic button can be created which when pressed (by a mouse button click) launches an activity. This activity could be a single simple action (eg 'play') or could start with a menu of other nested activities (and would then be seen by the user as a 'menu button'). A button can be of any size, and contain text and/or graphics which can be selected by the user from a library when creating the button, or drawn in.

A 'virtual switch' and one of its 'active states' can also be assigned to an activity and when activated launches the activity as above.

Implementation and further work

The system is being implemented within the Smalltalk-80 software system (Goldberg 1989), and is being used to control the 'E-Scape' composition environment (Anderson 90, 92). However, the only connection between the controlling system and E-Scape is via primitive 'system actions' - basic function names specifying basic operations which E-Scape can perform. Thus it could be integrated easily with any other application (implemented on the same Smalltalk-80 software platform) which has primitive system actions ('hooks') provided in a similar way; only these system actions would need altering.

Future AI development could allow the system to analyse a user's way of working, and present a selection of their most likely next action. This is similar to the way in which current speaking devices for the disabled predict the user's next letter or word, thereby reducing the amount of user interaction required. Alternatively the system could suggest some musically useful next actions, based on analysis of the activities of an expert composer.

Thus an activity would not be a fixed network of paths through actions, but adapt its presentation of action choices taking into account previous action selections.

References

Anderson, T.M. (1990). E-Scape: An extendable Sonic Composition and Performance Environment. *Proc. ICMC Glasgow 1990.* ICMA.

Anderson, T.M. (1992). Electroacoustic Scoring with Phase-vocoding Instruments using the E-Scape composition system. *Proc. ICMC San Jose 1992.* ICMA.

Goldberg, A., & Robson, D. (1989). *Smalltalk-80: the language.* Addison-Wesley

Appendix 5: The MIDI standard

5.1. Hardware

MIDI employs a serial interface using a 5mA current loop, with 'current on' as logic zero. It has a rate of 31.25 (± 1%) kbaud, with 10 bits per serial byte (8 data, one start, and one stop bit), implying ~320µs per byte. It is unidirectional, thus requiring two connections for bi-directional communication.

5.2. Message definitions

MIDI defines a set of message types, which can be used in various combinations.

5.2.1. Channel messages

'Channel messages' can address synthesis structures in devices via a 'channel' number, of which there can be sixteen on any one MIDI link. The first 'status' byte of a channel message has bit 7 set, and contains the channel number, and message type. It is followed by one or two data bytes (with bit 7 not set), which contain data for one or two fields within the message. A 3 byte channel message will take a minimum of 960 µs to transmit, ie ~1ms.

If several messages with the *same* 'status' byte (ie with the same message type, and channel number) follow each other, the standard specified that the identical status bytes after the first can be omitted, thus reducing the bandwidth requirement by 1/2 or 1/3. This mode of operation is known as 'running status'.

The following types of channel message are defined in the MIDI standard, and are presented below.

The status field (byte) is shown first in the form *h*n* , where *h* indicates hexadecimal notation, * is a nybble which indicates the message type (eg h9 = 'note on'), and *n* is a nybble indicating the MIDI channel number between 0 and 15 (0-hF).

One or two data fields (bytes) then follow, each shown as a pair of letters which give an indication of its function, eg kk = key number, vv = velocity, dd = data value.

- **Note on (h9n kk vv)**

This message starts a synthesis structure in a device playing, with its data fields specifying integer 'pitch' and 'velocity' values with a maximum range of 0 to 127.

The 'pitch' field (kk) conveys a semitone 'key' number (showing MIDI's keyboard origins) with 69 as A440Hz. Many synthesiser devices can, however, be set up to interpret each key number as an *arbitrary* pitch via a table stored in the device.

The 'velocity' field (vv) is usually interpreted as influencing some aspect of the synthesis process which is initiated by the 'note on' message, although some (older) devices will even simply ignore it.

• **Note off (h8n kk vv)**

This message is designed to match a corresponding 'note on' message, to command the playing of a note to cease. It also has 'pitch' (kk) and 'velocity' (vv) fields, the latter name derived from the speed at which a key is released during keyboard performance.

The 'pitch' field (kk) conveys no new information, being used as the 'match' field by the device in order to identify which note to stop. Thus only a single note of any one pitch can be playing, on any one channel.

This message does not necessarily *stop* a synthesis structure in a device operating (playing), but instructs it to move to its 'release' phase of operation which culminates in its termination. The 'release time' - the time elapsed between receipt of this message and note cessation is again not specified by the message, and depends on settings within the synthesis process. The 'velocity' field (vv) may be used to affect parameters of the 'release' phase, most typically this release time.

It should be noted here that the note *on* message can also be used to convey a 'note off' command by using a velocity field value of zero, thus foregoing the use of a 'release velocity' value, which few present devices respond to, or transmit.

• **Controller (hBn cc dd)**

This message has two data fields: 'controller number' (cc) and 'controller data value' (dd), again each with a 0 to 127 integer range. The 'controller number' (henceforth abbreviated to '#') field allows a number of independent control messages to be identified and used on the same channel, although again such messages will affect all notes on this channel.

A growing number of controller numbers have been defined with a name which conveys their intended use, eg controller #6 is named 'data entry'; controller #64 is named 'damper pedal on/off'.

For most of these controllers, no standard responses are defined. For example controller #1 ('modulation wheel'), or controller #4 ('foot controller') may have any number of effects on the sound output, depending on settings within the synthesis structure in use, or globally in the device.

Many controller numbers are presently left undefined, and most MIDI-based software does therefore allow new controller numbers to be used and named by the user.

The controller value gives a 7 bit range, but a second 'matching' message (with its 'controller number' field offset by 32) can optionally be sent to convey 7 more bits of resolution as an LSB. Again, whether or not a device responds to this additional LSB-providing message, or indeed to any particular controller number is *not* standardised.

The number of controller numbers is finite (128), and thus there is a facility to add an additional 16,384 controller types via a scheme known as 'Registered Parameter Controllers' (RPC).

Controller number 101 is used to send a value which indicates a registered parameter number to a device (0-127). If more parameter numbers than this are required, controller

number 100 can send an LSB value afterwards to provide 16,384 different registered parameter numbers. The MIDI ('data entry') controller message (with controller #6) can then be used to send *values* for this set parameter number.

Three parameter numbers have been registered as standard with the IMA so far: for sending values for pitch bend sensitivity' (see below), and the global coarse and fine tuning of a device. An example of this facility, implemented on the Roland 'D110' synthesiser device (Roland 1988), is used in an example instrument in chapter 8.

A similar scheme operates with controllers 99 and 98, this time the parameter numbers specified being 'unregistered'. For example, the Oberheim 'Matrix 1000' synthesiser device uses these unregistered parameter numbers for all 133 of the input parameters to its synthesis structure (Oberheim 1988). While this means that each manufacturer sets their own definitions, it can be an effective way of controlling complex or device specific parameters without resorting to the longer and more complicated 'system exclusive' message type (see below).

- **Channel Aftertouch (hDn dd)**

This message has a single 'value' field (dd) with a range of 0-127 (this time with *no* facility to extend the range to 14 bit resolution). It is similar in effect to a controller message, able to convey time-varying data values which can affect synthesis processes for *all* the notes playing with this channel.

Its name derives from the originally expected source of the message, pressure applied to a keyboard key *after* it has been fully depressed. However, as with the 'controller' message, the sonic *meaning* and *effect* of the message is undefined, depending on a device's internal settings.

- **Pitchbend (hEn ll mm)**

This message can alter the pitch of all notes playing on this channel. It has two data bytes, MSB (mm) and LSB (ll) giving a potential 14 bit resolution, but most present devices only respond to the MSB value, to give 7 bit resolution, although some more recent devices have started to increase this resolution.

Again, however, the *response* of a device is not specified - a few do not respond at all, or alter the pitch by an amount which depends on a setting within the device (usually termed 'bender range' or 'pitch bend sensitivity'). Values can typically be set to integers between 1 to 12, 24 or 36, and indicate the positive pitch variation in semitones when the maximum 'pitch bend' value of 127 (or 16383 using 14 bit resolution[1]) is received.

The value for which *no* pitch variation is effected is actually the half way point in this 'pitch bend' value range, ie 64, (or 8064), with zero giving maximum negative variation.

The 'pitch bend sensitivity' or 'bender range' setting can often be transmitted to the device via MIDI, and this capability is a de facto standard on recent devices.

---

1 $16383 = 128^2 - 1$.

• **Program change (hCn pp)**

This message has a single data field, which can select one of 128 'programs' or 'patches' - synthesis units or networks - whose specifications are stored in a device. The selected 'program' is then installed in an active memory area in the device (with a designated MIDI channel). The synthesis structure can be played (ie set running) and sent parameter data by messages on this channel, or addressed directly within the device's memory map, via system exclusive messages (see 2.5.2.2). A major shortcoming is the limit of 128 instruments, as many, even cheap, MIDI devices now have more than this. The new recent definition of MIDI controller #0 as a 'bank select' message gets round the problem, but is inelegant.

• **Polyphonic aftertouch (hAn kk dd)**

This message is the only one which allows a continuously varying parameter value (dd) to be sent which is assigned to an individual note, again using the 'pitch' field (kk) for matching. Very few devices at present support this message.

5.2.2. System messages

These perform various tasks to do with synchronising devices, requesting devices to tune themselves (a throwback to analogue voltage controlled devices), or resetting to a default state. These aspects will not be discussed here.

The most important category of system message is the 'System Exclusive' message. This starts with a status byte of hF0 (there is no channel number), then has a 7 bit id number, followed by any number of 7 bit bytes (ie the MS bit not set), finishing with a final hF7 byte.

The id number can be:

• h7D, indicating non-commercial use.

• h7E or h7F, indicating non-real time and real-time extensions to the MIDI standard which cannot be accommodated within the channel message format. An example of the former are the various messages within the MIDI sample dump standard (SDS), which allow digital sample data to be transferred over a MIDI link. An example of the latter are MIDI Time Code (MTC) messages - a way of encoding SMPTE-type absolute time information for device synchronisation purposes.

• a "manufacturer's id" - each is a number assigned to a specific commercial manufacturer of MIDI-based devices. The format of the data following this manufacturer's id is not specified, and is left up to each manufacturer. Many of the more subtle and complex timbral modifications to a sound produced by a MIDI device can be controlled by these messages, and any composition system which aims to use such devices must provide a way of defining the fields and kind of data needed for such messages.

5.3. General MIDI

This recent extension to the MIDI standard provides for some degree of standardisation for synthesis devices, both in terms of the facilities they must provide, and their response to certain messages. Devices which conform to the 'General MIDI' standard must respond on all 16 MIDI channels, with the ability to have a different program (ie sound) assigned to each channel, and to be able to play at least 24 simultaneous notes in total. Channel 10 is specified to have *percussion* sounds, with a designated type of sound for each MIDI note (key) number. Its other main features are an association of each program change number with a particular identifiable sound type. For example a program change number of 1 should result in an 'acoustic piano' type of sound being installed in the device.

Definitions

Object-oriented terms

Abstract class	An object class which is not designed to be instantiated itself, but to provide state and functionality definitions to enable *subclasses* to be defined.
Browser	A specialised software windowed facility within a GUI-based OOPS, which provide a view of particular aspects of the system, eg the Class definition data.
Class	The definition template within an OOPS from which objects can be instantiated.
Dynamic typing	The concept of only taking account of an object's data type during software system running.
Dynamic binding	The concept of only selecting which method to use at run time, when an object's type is determined.
Encapsulation	The concept of incorporating both state and functionality within an object, to which access is then only provided by specific methods.
Hierarchy (of classes)	The structuring of object classes into relationships where classes can inherit state (instance variables) and functionality (methods) from a common superclass ancestor.
Inheritance	The use of an ancestor superclass to create a new variant class which derives any or all its features from it, but will typically modify or add new features.
Instance variable	A variable defined in an object class which will be owned by any objects instantiated from the class. Each instantiated object then has its own individual assignments of its instance variables.
Instantiation	The creation of objects from a template class, which defines the methods each object can understand.
Polymorphism	The ability to send the same message to objects of different classes, which may then each invoke a *different* method.
Method	A set of behaviour (like a function) which a Smalltalk object can perform when invoked by sending it the concomitant message.
Message	A language construct in Smalltalk which invokes the performance of a method.
Subclassing	The creation of a class which inherits the structure and behaviour of another class - a superclass.

Other terms with specific meanings

module	A network of synthesis units - instantiated in a device, or described within a high-level control composition system - which are treated as a composite.
device	An entity which produces sound, usually via digital processes. It may or may not incorporate its own user interface control system, but must include a facility for external control by another *discrete* system, ie must be able to be treated as a *separate* entity. NB. A device should not be confused with the capitalised 'Device'. The latter is a Smalltalk class of object within the E-Scape software which *describes* various features of a device.
unit	A primitive synthesis process within a device which cannot be broken down (as far as the outside user is concerned) into lower-level entities.
instrument	A network of units as instantiated in a device. NB. An instrument should not be confused with the capitalised 'Instrument' . The latter is a Smalltalk class of object within the E-Scape software which describes a more complex structure, only part of which correlates with an instrument in a device.

Particular terms which are related and should be distinguished

'device' / Device

The word 'Device' (in uppercase) signifies a class of software object in E-Scape.

This must be distinguished from the word 'device' (in lowercase) which signifies an entity that runs synthesis processes. A device may consist of hardware or software which is physically or notionally discrete from the E-Scape(or other) compositional control software subsystem. Thus, a Device object *describes* a device.

'unit' / 'potential unit' / DCTPrimitive / DCIPrimitive

Following the above upper case convention, 'DCTPrimitive' and 'DCIPrimitive' also denote classes of software object in E-Scape, while 'unit' and a 'potential unit' denote entities which exist on a device.

• A 'unit' is a primitive synthesis process running on a device, which is the constituent part of an instrument (see above).

• A 'DCTPrimitive' object *describes* a unit specification within the context of other units in an instrument. A DCTPrimitive does not correlate *directly* to a unit, as the latter is an instantiated entity, whereas a DCTPrimitive[1] object is only a description of the *specification* of a unit.

• A 'DCIPrimitive' object is a description of an *instantiated* unit in a device (cf. a DCTPrimitive describes the unit's *specification*).

• A 'potential unit' is a data structure entity which is stored in a device. It is similar in function to a DCIPrimitive - ie is a description of an *instantiated* unit in the device - but is stored in the device itself, rather than in E-Scape software.

It should be noted that units and potential units will exist in devices under the proposed standards, whether controlled from E-Scape, or some other compositional software subsystem. All Smalltalk objects only exist in E-Scape, however: other compositional software might implement the same functionality, but will employ different structures or objects to achieve this.

'instrument-template' / DCT

In both Level I and II, a DCTPrimitive exists within a DCT in E-Scape, as does a unit in a device.

At level I, when a DCT is allocated (instantiated) in a device, each of its component DCTPrimitives is individually allocated by E-Scape as a unit in the device. This allocation of a DCTPrimitive is described within E-Scape by a DCIPrimitive (which contains details of its address/ slot etc within the device).

---

1 'DCTPrimitive' is an abbreviation for "DeviceConfigurationTemplatePrimitive", ie is a primitive component of a DCT object.

In level II a DCTPrimitive is first *downloaded* to the device as a component of an instrument-template, within which it exists as a potential-unit. Now, when a DCT is allocated in a device, E-Scape can treat it as a composite entity, and it can allocate the DCT within the device as an instrument. The *device* then allocates the component units using its stored specification instrument-template. At level II, E-Scape thus has no need to keep track of individual allocations of component units, thus does not need to create DCIPrimitive objects. The DCTPrimitives which comprise a DCT are used only at the earlier stage, when the instrument-template was described to, and stored by the device. Compare this with level I, where each DCTPrimitive has to be allocated in the device by E-Scape individually, and it therefore has to keep track of these allocations using DCIPrimitives.

To summarise the difference, at level I E-Scape maintains a description of the allocation within the device of *individual* units (using DCIPrimitive objects), whereas at level II it first describes the construction of an instrument to the device, but the subsequently only needs to maintain a description of the allocation within the device of the instrument as a *whole.*

'instrument' / Instrument

• The word 'Instrument' (in uppercase) signifies a class of software object in E-Scape.

• This must be distinguished from the word 'instrument' (in lowercase) which signifies a network of one or more synthesis processes (units) which are instantiated on a device.

An Instrument software object is not directly analogous to an instrument (as a Device is to a device). An Instrument object contains further objects (DCTs), each of which describes an instrument structure on a device. When an allocation of an Instrument is made, an instrument is instantiated on a device corresponding to each of these DCT objects.

Thus an Instrument object not only contains *specifications* of instruments, but in addition contains objects (PspProcessors) which specify and process higher-level scoring parameters.

There are a network of relationships between the DCTPrimitive and DCIPrimitive objects, and the 'potential-unit' and 'unit' device entities:

A DCTPrimitive relates to a 'unit' on a device, either via an intermediary DCIPrimitive object, or via a 'potential unit' entity on a device, depending whether Level I or Level II communication from E-Scape to the device is in use:-

In Level I communication, all units within an instrument have a description of their allocation stored within the E-Scape software as a 'DCIPrimitive' object.

A DCIPrimitive is created from a corresponding 'template' DCTPrimitive, when a score event is allocated resources within a device. The event will have one or more DCTs, and each DCTPrimitive it contains is allocated resources within a device, which is described by a DCIPrimitive. This DCIPrimitive corresponds directly to a unit within a device, and contains details of the device to be used, as well as the unit's id, or address if any within the device. The DCIPrimitive also contains Messages which will effect the instantiation of the unit in the device when sent out from E-Scape.

Glossary

56000 / 56001	A make of DSP chip
ADC	Analogue to digital converter
AES/EBU	A digital data transfer standard
algorithm	A process which produces output states or data, from input states or starting conditions
Analogue	A process which has *continuous* parameters, as opposed to a quantised digital description. Also refers to synthesis devices which use such electronic (usually voltage controlled) components
CEMAMu	The Centre d'Etudes de Mathématique et Automatique Musicales (The Centre for Studies of Mathematical and Automated Music) in France.
Channel Aftertouch	A message type within the MIDI protocol
channel	A labels which pertains to and identifies a particular message or a particular location within a device
Common Practice Notation (CPN)	The traditional western music notation for representation of musical compositions.
D110	A MIDI-controlled synthesiser device
DAC	Digital to Analogue Converter - converts a stream of digital samples to the sound signal they describe
device	Any entity (hardware or software) which can be controlled externally, to create sound output
DSP	Digital Signal Processing. Also refers to VLSI chips whose operation and structure is optimised for such processing
electroacoustic	A type of music where the structural emphasis is on the microstructural details within the sound and its evolution
Ethernet	A common LAN
FM	Frequency Modulation - in the context of this thesis it refers to a particular synthesis technique where the modulating frequency is in the audio range
FOF	Synthesis employing a 'Formant wave function' algorithm, which builds a sound from a controlled, pitch-independent formant spectrum.

GUI	Graphic User Interface - computer interface based on a bit-mapped screen with graphic presentation - see WIMP
Hybrid	A system with digital control of analogue synthesis processes
IRCAM	The French Computer Music Research Institute
ISPW	The IRCAM Signal Processing Workstation
JEIDA	The Japanese Electronic Industry Development Association
LAN	Local Area Network
loose coupling	A system organisation where communication is between heterogeneous nodes via interpreted protocols
MADI	Multi-channel Audio Digital Interface - a communications protocol
MIDI Machine Control	A set of MIDI system messages facilitating control of tape recorder operation
MIDI	Musical Instrument Digital Interface (see 2.5)
MIDI sample dump standard	Part of the MIDI protocol - facilitate the transfer of digital audio data via a MIDI connection
MIDI Standard Files	A standard file format allowing interchange of time-stamped MIDI events between applications
MIDI Time Code	Part of the MIDI protocol, which facilities synchronisation and cueing of events within devices. Usually abbreviated to MTC.
MII	Midas Intermediate Interface - an application overlay to the MIDAS system which facilities interaction with a user and high-level control by another system
MTC	see MIDI Time Code
OOPS	Object Oriented Programming System (eg Smalltalk, Objective C) - see chapter 7
PCMCIA	The Personal Computer Memory Card International Association
Pitchbend	A message type within the MIDI protocol
Polyphonic aftertouch	A message type within the MIDI protocol
polyphony	The number of simultaneous events which a single device synthesis process can play
Registered Parameter Numbers	Used by MIDI RPC messages
RPC	Registered Parameter Controller - a MIDI controller message
RS232	A serial interface standard

Running status	The omission of identical consecutive status bytes from a MIDI message stream
SCSI	Small Computer Systems Interface - a standardised fast parallel data transfer interface in widespread use
SMPTE	Society of Motion Picture and Television Engineers. The abbreviation almost invariably refers to the time code synchronisation standard designed by this body.
soundfile	A data file on disk, often contiguous, which describes digital sound sample data
spectral morphology	The aspect of sound and musical structure which concentrates on the shaping in time of the entire range of frequencies (partials) which constitute a sound.
System Exclusive	A message type within the MIDI protocol
UNIX	A multi-tasking operating system in widespread use
Unregistered Parameter Number	Used by MIDI RPC messages
VLSI	Very Large Scale Integration - refers to silicon chips which incorporate highly condensed electronic circuits
WIMP	Windows, Icons, Menu, Pointer - describes the GUI-based software systems which employ on screen graphic windows, and allow a user to control an on-screen pointer to access graphic icons and menus to control the system.

List of References

Abbott, C. (1985). 'Introduction'. in *ACM Computing Surveys* 17(2) 1985.

AES (1985). 'Serial Transmission Format for Linearly Represented Digital Audio Data'. in *Journal of the Audio Engineering Society* 33(10) 1985. 976-984.

Althoff, J. C. (1981). 'Building Data Structures in the Smalltalk-80 System'. in *Byte* 6(8) August 1981. 230-278.

Andersen, D. M. (1991). 'Portability and the GUI'. in *Byte* 16(12) 1991. 221-226.

Anderson, D. P. (1987). 'Synthesizer Management Based on Note Priorities'. *Proc. ICMC*, Illinois. 1987.

Anderson, D. P. and Kuivila, R. (1991). 'Formula: A Programming Language for Expressive Computer Music'. in *Computer* (July) 1991. 12-21.

Anderson, D. P. and R.Kuivila (1986). 'A Model of Real-time Computation for Computer Music'. *Proc. ICMC*, The Hague. 1986.

Anderson, T. M. (1990). 'E-Scape: An Extendable Sonic Composition and Performance Environment'. *Proc. ICMC*, Glasgow. 1990.

Anderson, T. M. and Howard, D. M. (1991-92). 'The New Sounds of Music'. in *Ability - The Journal of the British Computer Society Disability Programme* (3) 1991-92. 9-12.

Anderson, T. M. and Kirk, P. R. (1992b). 'Electroacoustic Scoring with Phase-vocoding Instruments using the E-Scape composition system'. *Proc. ICMC*, San Jose. 1992.

Anderson, T. M. and Kirk, P. R. (1993a). 'A Customisable User Interface System for the Structuring of Compositional Tasks'. *Proc. Workshop on Music Education: An Artificial Intelligence Approach, AI-ED 93*, Edinburgh. 1993.

Anderson, T. M., Kirk, P. R., Hunt, A., McGilly, P., et al. (1992a). 'From Score to Unit Generator: A Hierarchical View of MIDAS'. *Proc. ICMC*, San Jose. 1992.

Anderson, T. M. and Oppenheim, D. V. (1993b). 'Perceptual Parameters - their specification, scoring and control within two software composition systems'. *Proc. ICMC*, Tokyo. 1993.

Anderton, C. (1987). 'The MIDI protocol'. *Proc. AES 5th International Conference - Music and Digital Technology*, Los Angeles. 1987.

Andrenacci, P., Favreau, E., Larosa, N. and Prestigiacomo, A. (1992). 'MARS: RT20M/EDIT20 - Development Tools and Graphical User Interface for Sound Generation Board'. *Proc. ICMC*, San Jose, CA. 1992.

Assayag, G. and D.Timis (1986). 'A ToolBox for Music'. *Proc. ICMC*, The Hague. 1986.

Bailey, N., Manning, P., Purvis, A., Bowler, I., et al. (1990). 'CSound Inside and Out: Today's Software, Tomorrow's Computers'. in *Musicus* 2(i/ii) 1990. 51-58.

Baisnée, P.-F., J-B.Barrière, M-A.Dalbavie, J.Duthen, et al. (1988). 'Esquisse: A Compositional Environment'. *Proc. ICMC*, Cologne. 1988.

Balaban, M. (1988). 'A Music-Workstation based on Multiple Hierarchical Views of Music'. *Proc. ICMC*, Cologne. 1988.

Ballista, A., Casali, E., Chareyron, J. and Haus, G. (1992). 'A MIDI/DSP Sound Processing Environment for a Computer Music Workstation'. in *Computer Music Journal* 16(3) 1992.

Banger, C. and B.Pennycook (1983). 'Gcomp: Graphic Control of Mixing and Editing'. in *Computer Music Journal* 7(4) 1983. 33-39.

Barrière, J.-B., Iovino, F. and Laurson, M. (1991). 'A new CHANT Synthesiser in C and its control Environment in *Patchwork*'. *Proc. ICMC*, Montreal. 1991.

Bate, J. A. (1992). 'Max + Unison - Interactive Control of a Digital Signal Multiprocessor'. *Proc. ICMC*, San Jose, CA. 1992.

Beck, K. (1992). 'Why study Smalltalk Idioms?'. in *The Smalltalk Report* 1(7) 1992. 23-25.

Berg, P. (1985). PILE - A Language for Sound Synthesis'. in Roads, C. and Strawn, J., Ed. *Foundations of computer music.* MIT Press.

Berg, P., R.Rowe and D.Theriault. (1979). 'SSP and Sound Description'. in *Computer Music Journal* 2 1979. 25-35.

Blythe, D., J.Kitamura, D.Galloway and M.Snelgrove (1986). 'Virtual Patch-Cords for the Katosizer'. *Proc. ICMC*, The Hague. 1986.

Bôcker, H.-D. and A.Mahling (1988). 'What's in a note?'. *Proc. ICMC*, Cologne. 1988.

Bokkel, A. t. (1986). 'The Errant Syncretizer'. *Proc. ICMC*, The Hague. 1986.

Boynton, L., J.Duthen, Y.Potard and X.Rodet (1986a). 'Adding a Graphical User Interface to FORMES'. *Proc. ICMC*, The Hague. 1986.

Boynton, L., P.Lavoie, Y.Orlarey, C.Rueda, et al. (1986b). 'A LISP-Based Music Programming Environment for the Macintosh'. *Proc. ICMC*, The Hague. 1986.

Bregman, A. S. (1990). *Auditory Scene Analysis.* MIT Press.

Burk, P. (1991). 'The Integration of Real-Time Synthesis into HMSL'. *Proc. ICMC*, Montreal. 1991.

Buxton, B. (1987). 'Masters and Slaves Versus Democracy: MIDI and Local Area Networks'. *Proc. AES 5th International Conference - Music and Digital Technology*, Los Angeles. 1987.

Buxton, W., Fogels, E. A., Fedorkow, G., Sasaki, L., et al. (1978). 'An Introduction to the SSSP digital synthesiser'. in Roads, C. and Strawn, J., Ed. *Foundations of computer music.* MIT Press.

Buxton, W., Sniderman, R., Reeves, W., Patel, S., et al. (1979). 'The evolution of the SSSP score editing tools'. in Roads, C. and Strawn, J., Ed. *Foundations of computer music.* MIT Press.

Chamberlin, H. (1980). *Musical Applications of Microprocessors.* Hayden Book Co. Inc.

Cole, H. (1974). *Sounds and Signs.* Oxford University Press.

Collinge, D. J. and D.J.Scheidt (1988). 'Moxie for the Atari ST'. *Proc. ICMC*, Cologne. 1988.

Cook, S. (1993). 'Object Market'. in *Consultants' Conspectus* April 1993. 2-5.

Cox, B. J. (1986). *Object-oriented programming - An Evolutionary Approach.* Addison Wesley.

Czeiszperger, M. (1988). 'A Multiple Workstation Environment for Joint Computer Music / Computer Graphics Production'. *Proc. ICMC*, Cologne. 1988.

Dannenberg, R. B. (1986a). 'A Structure for Representing , Displaying, and Editing Music'. *Proc. ICMC*, The Hague. 1986.

Dannenberg, R. B., Fraley, C. L. and Velikonja, P. (1991). 'Fugue: A Functional Language for Sound Synthesis.'. in *Computer* (July) 1991.

Dannenberg, R. B., P.McAvinney and D.Rubine (1986b). 'Arctic: A Functional Language for Real-time Systems'. in *Computer Music Journal* 10(4) 1986. 67-77.

Decker, S. L., Kende, G. S., Schmidt, B. L., Ludwig, M. D., et al. (1986). 'A Modular Environment for Sound Synthesis and Composition'. in *Computer Music Journal* 10(4) 1986. 28-41.

Degazio, B. (1988). 'The Development of Context Sensitivity in the Midiforth Computer Music System'. *Proc. ICMC*, Cologne. 1988.

Depalle, P. and Rodet, X. (1990). 'U.D.I. A Unified DSP Interface for sound signal analysis and synthesis'. *Proc. ICMC*, Glasgow. 1990.

Desain, P. (1986). 'Graphical Programming in Computer Music: A Proposal'. *Proc. ICMC*, The Hague. 1986.

Desain, P., Honing, H., Dannenberg, R., Jacobs, D., et al. (1993). 'A Max Forum'. in *Array* 13(1) 1993. 14-20.

Deutsch, L. P. and Goldberg, A. (1991). 'Smalltalk - Yesterday, Today, and Tomorrow'. in *Byte* 16(8) August 1991. 108-115.

Diener, G. (1988). 'TTrees: An Active Data Structure for Computer Music'. *Proc. ICMC*, Cologne. 1988.

Diener, G. (1989). 'TTrees: A Tool for the Compositional Environment'. in *Computer Music Journal* 13(2) 1989. 77-85.

Dobrian, C. and Zicarelli, D. (1990). *MAX (An Interactive Graphic Programming Environment) - Manual.* Ircam / Opcode Systems, Inc.

Dodge, C. and Jerse, T. A. (1985). *Computer Music: Synthesis, Composition and Performance.* Macmillan.

Duff, C. and Howard, B. (1990). 'Migration Patterns'. in *Byte* 15(10) October 1990. 223-232.

Duthen, J. and Laurson, M. (1990). 'A Compositional Environment based on PreFORM II, Patchwork and Esquisse'. *Proc. ICMC*, Glasgow. 1990.

Duthen, J. and Y.Potard (1987). 'Le_Loup, an Object-oriented Extension of Le_Lisp for an Integrated Computer Music Environment'. *Proc. ICMC*, Illinois. 1987.

Dyer, L. M. (1986). 'MUSE: An Integrated Software Environment for Computer Music Applications'. *Proc. ICMC*, The Hague. 1986.

E-mu (1989). *Proteus Digital Sound Module Operation Manual.* E-mu Systems, Inc.

Eckel, G., X.Rodet and Y.Potard (1987). 'A Sun-Mercury Music Workstation'. *Proc. ICMC*, Illinois. 1987.

Emmerson, S. (1989). Composing Strategies and Pedagogy'. in Clarke, E. and Emmerson, S., Ed. *Music, Mind and Structure*. Harwood Academic Publishers. 133-144.

Emmerson, S. (1991). 'Computers and Live Electronic Music: Some Solutions, Many Problems'. *Proc. ICMC*, Montreal. 1991.

Favreau, E., M. Fingerhut, O. Koechlin, P. Potacsek, et al. (1986). 'Software Development for the 4X Real-time System'. *Proc. ICMC*, The Hague. 1986.

Flurry, H. S. (1989). 'An Introduction to the Creation Station'. in *Computer Music Journal* 13(2) 1989. 56-70.

François, J.-C., X.Chabot and J.Silber (1987). 'MIDI Synthesizers in Performance: Realtime Dynamic Timbre Production'. *Proc. ICMC*, Illinois. 1987.

Free, J. (1987). 'Towards an Extensible Data Structure for the Representation of Music on Computers'. *Proc. ICMC*, Illinois. 1987.

Free, J. and P.Vytas (1988). 'The CAMP Music Configuration Database - Approaching the Vanilla Synthesizer'. *Proc. ICMC*, Cologne. 1988.

Free, J., P.Vytas and W.Buxton (1986). 'Whatever happened to SSSP?'. *Proc. ICMC*, The Hague. 1986.

Freed, A. (1986). 'MacMix: Mixing Music with a Mouse'. *Proc. ICMC*, The Hague. 1986.

Fry, C. (1984). 'Flavors Band: A Language for Specifying Musical Style'. in *Computer Music Journal* 8(3) 1984. 20-34.

Goebel, J. (1991). 'My Dream (Machine?)'. in *Computer Music Journal* 15(4) 1991. 47-50.

Goldberg, A. (1984). *Smalltalk-80: The Interactive Programming Environment*. Addison Wesley.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison Wesley.

Graham, B. (1980). *Music and the Synthesiser*. Argus Books.

Greenberg, G. (1986). 'Computers and Music Education: A Compositional Approach'. *Proc. ICMC*, The Hague. 1986.

Greenberg, G. (1988). 'Composing with Performer Objects'. *Proc. ICMC*, Cologne. 1988.

Griffiths, P. (1979). *A Guide to Electronic Music*. Tha mes and Hudson.

Grossman, G. (1987). 'Instruments, Cybernetics, and Computer Music'. *Proc. ICMC*, Illinois. 1987.

Haus, G. (1983). 'EMPS: A System for Graphic Transcription of Electronic Music Scores'. in *Computer Music Journal* 7(3) 1983. 31-36.

Hebel, K. J. (1987). 'Javelina: An Environment for the Development of Software for Digital Signal Processing'. *Proc. ICMC*, Illinois. 1987.

Helmuth, M. (1990). 'PATCHMIX: A C++ X Graphical Interface to Cmix'. *Proc. ICMC*, Glasgow. 1990.

Helmuth, M. (1992). 'Timbral Composition and Gesture'. *Proc. Science and Philosophy in Tomorrow's Music*, Delphi, Greece. 1992.

Hunt, A. (1991). Personal Communication

Hunt, A. D. and Kirk, P. R. (1988). 'MidiGrid: A new musical performance & composition system'. *Proc. Institute of Acoustics*, Windemere. 1988.

Hunt, A. D., Kirk, P. R., Abbotson, M. and Abbotson, R. (1992). 'MidiGrid: computer-based instrument'. *Proc. Music Therapy in Health and Education in the European Community*, Cambridge. 1992.

Hunt, A. D., Kirk, P. R. and Orton, R. (1990). 'MidiGrid: An Innovative Computer-based Performance and Composition system'. *Proc. ICMC*, Glasgow. 1990.

IMA (1988). *MIDI 1.0 Detailed Specification, version 4.0.* International MIDI Association.

IMA (1991). *General MIDI Level 1 Specification.* International MIDI Association.

Jaffe, D. (1989a). 'Overview of the NeXT Music Kit'. *Proc. ICMC*, Ohio. 1989.

Jaffe, D. and L.Boynton (1989b). 'An Overview of the Sound and Music Kits for the NeXT Computer'. in *Computer Music Journal* 13(2) 1989. 48-55.

Jaffe, D. A. (1991). 'Musical and Extra-musical Applications of the NeXT Music Kit'. *Proc. ICMC*, Montreal. 1991.

Jaffe, D. A. (1993). 'More reactions to the "Mins of Max"'. in *Array* 13(2) 1993. 10-11.

Kaehler, T. and Patterson, D. (1986). 'A Small Taste of Smalltalk'. in *Byte* 11(8) August 1986. 145-159.

Kaehler, T. and Patterson, D. (1986). *A Taste of Smalltalk.* W.W. Norton & Co.

Kahrs, M. and Killian, T. (1992). 'Gnot Music: A Flexible Workstation for Orchestral synthesis'. in *Computer Music Journal* 16(3) 1992. 48-56.

Katrami, A. I., R, K. P. and Myatt, A. (1991). 'A Phase Vocoder Graphical Interface for Timbral Manipulation of Cellular Automata and Fractal Landscape Mappings'. *Proc. ICMC*, Montreal. 1991.

Katrami, A. I., R, K. P., Orton, R. and Hunt, A. D. (1992). 'Deconstructing the Phase Vocoder'. *Proc. ICMC*, San Jose. 1992.

Kirk, P. R. and Hunt, A. D. (1993). 'Graphical Performance Interfaces for Computer Music Instruments'. *Proc. Eurographics UK*, York. 1993.

Kirk, P. R. and Orton, R. (1990). 'MIDAS: A Musical Instrument Digital Array Signal Processor'. *Proc. ICMC*, Glasgow. 1990.

Koblyakov, L. (1992). 'Score/Music Orientation: An Interview with Robert Rowe'. in *Computer Music Journal* 16(3) 1992. 22-31.

Koenig, G. M. (1983). 'Aesthetic Integration of Computer-Composed Scores'. in *Computer Music Journal* 7(4) 1983. 27-32.

Krasner, G. E. (1980). 'Machine Tongues VIII: Design of a Smalltalk Music System'. in *Computer Music Journal* 4(4) 1980. 4-14.

Krasner, G. E., Ed. (1983). *Smalltalk-80: Words of History, Bits of Advice.* Addison-Wesley.

Krasner, G. E. and Pope, S. T. (1988). 'A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80'. in *Journal of Object Oriented Programming* 1(3) 1988. 35-40.

LaLonde, W. R. and Pugh, J. R. (1989). 'Designing Is Hard: Object-Oriented Software is Different!'. in *Journal of Object Oriented Programming* 2(2) 1989. 47-58.

LaLonde, W. R. and Pugh, J. R. (1990b). *Inside Smalltalk (Vol 2).* Prentice Hall.

LaLonde, W. R. and Pugh, J. R. (1990a). *Inside Smalltalk (Vol. 1).* Prentice Hall.

LaLonde, W. R. and Pugh, J. R. (1990c). 'Smalltalk as the first programming language: the Carleton experience'. in *Journal of Object Oriented Programming* 3(4) 1990. 60-65.

Laurson, M. and Duthen, J. (1989). 'Patchwork: a Graphic language in Preform'. *Proc. ICMC*, Ohio. 1989.

Lengeling, G. and Adam, C. (1992). *Notator 3.1.* EMagic GmbH.

Lent, K., Pinkston, R. and Silsbee, P. (1989). 'Accelerando: A Real-Time, General Purpose Computer Music System'. in *Computer Music Journal* 13(4) 1989. 54-64.

Lindemann, E. (1990). 'Animal - A Rapid Prototyping Environment for Computer Music Systems'. *Proc. ICMC*, Glasgow. 1990.

Lindemann, E. and de Cecco, M. (1991). 'Animal: Graphical Data Definition and Manipulation in Real Time'. in *Computer Music Journal* 15(3) 1991. 78-100.

Lindemann, E., Starkier, M. and Dechelle, F. (1990). 'the IRCAM Musical Workstation: Hardware Overview and Signal Processing Features'. *Proc. ICMC*, Glasgow. 1990.

Lindemann, E., Starkier, M., Smith, B. and Dechelle, F. (1991a). 'The Architecture of the IRCAM Musical Workstation'. in *Computer Music Journal* 13(3) 1991.

Lippe, C. (1993). 'What's NeXT'. in *Array* 13(2) 1993. 20-21.

Lippe, C. and Puckette, M. (1991). 'Musical Performance using the IRCAM Workstation'. *Proc. ICMC*, Montreal. 1991.

Lohner, H. (1986). 'The UPIC System: A User's Report'. in *Computer Music Journal* 10(3) 1986. 42-49.

Loy, D. G. (1986). 'Designing a Computer Music Workstation from Musical Imperatives'. *Proc. ICMC*, The Hague. 1986.

Loy, G. (1985a). 'Musicians make a standard: The MIDI phenomenon'. in *Computer Music Journal* 9(4) 1985. 8-26.

Loy, G. (1989). Composing with Computers - a Survey'. in Mathews, M. and Pierce, J., Ed. *Current Directions in Computer Music Research.* MIT Press.

Loy, G. and Abbott, C. (1985b). 'Programming Languages for Computer Music Synthesis, Performance, and Composition'. in *ACM Computing Surveys* 17(2) 1985. 235-265.

Loy, G. and Freed, D. J. (1992). 'The Frox Digital Audio System'. *Proc. ICMC*, San Jose, CA. 1992.

Mahling, A. (1991). 'How to Feed Musical Gestures into Compositions'. *Proc. ICMC*, Montreal. 1991.

Marino, G., Raczinski, J.-M. and Serra, M.-H. (1990). 'The New UPIC System'. *Proc. ICMC*, Glasgow. 1990.

Marino, G., Raczinski, J.-M. and Serra, M.-H. (1992). 'A Description of the UPIC system'. *Proc. Science and Philosophy in Tomorrow's Music*, Delphi, Greece. 1992.

Mathews, M. (1969). *The Technology of Computer Music*. MIT press.

Mauchly, J. W. and Charpentier, A. J. (1987). 'Practical Considerations in the Design of Music Systems using VLSI'. *Proc. AES 5th International Conference - Music and Digital Technology*, Los Angeles. 1987.

McGilly, P. (1992). University of York MSc Thesis. 'The MIDAS System - Appendix: Application Programmer's Guide'.

Mellinger, D. K., G.E.Garnett and B.Mont-Reynard (1989). 'Virtual Digital Signal Processing in an Object-Oriented System'. in *Computer Music Journal* 13(2) 1989. 71-76.

Minnick, M. (1990). 'A Graphical Editor for Building Unit Generator Patches'. *Proc. ICMC*, Glasgow. 1990.

Moog, R. A. (1965). 'Voltage controlled electronic modules'. in *Journal of the Audio Engineering Society* 13(3) 1965.

Moore, F. R. (1982). 'The Computer Audio Research Laboratory at UCSD'. in *Computer Music Journal* 6(1) 1982. 18-29.

Moore, F. R. (1987). 'The Dysfunctions of MIDI'. *Proc. ICMC*, Illinois. 1987.

Morgan, N. (1993). 'Symbolic Composer - a description and evaluation'. in *Musicus* 3(i) 1993.

Nieberle, R. C., Rothkamm, F. H., Verwiebe, M., Modler, P., et al. (1988). 'The CAMP system: An Approach for Integration of Realtime, Distributed and Interactive Features in a Multiparadigm Environment'. *Proc. ICMC*, Cologne. 1988.

Nielsen, J. and Richards, J. T. (1989). 'The Experience of Learning and Using Smalltalk'. in *IEEE Software* May 1989. 73-76.

Nottoli, G. and Galante, F. (1986). 'SOFT MACHINE: a real time fully programmable computer music system'. *Proc. ICMC*, The Hague. 1986.

Oberheim (1988). *Matrix-1000 Analog Sound Module Owner's Manual*. ECC Development Corp.

Oppenheim, D. V. (1986). 'The Need for Essential Improvements in the Machine-Composer interface used for the Composition of Electroacoustic Computer Music'. *Proc. ICMC*, The Hague. 1986.

Oppenheim, D. V. (1989). 'Dmix: An Environment for Composition'. *Proc. ICMC*, Ohio. 1989.

Oppenheim, D. V. (1990). 'Quill: An Interpreter for Creating Music-Objects within the Dmix Environment'. *Proc. ICMC*, Glasgow. 1990.

Oppenheim, D. V. (1991). 'Towards a better Software-Design for Supporting Creative Musical Activity'. *Proc. ICMC*, Montreal. 1991.

Oppenheim, D. V. (1992). 'DMIX - A Multi-faceted Environment for Composing and Performing Computer Music'. *Proc. Science and Philosophy in Tomorrow's Music*, Delphi, Greece. 1992.

Orlarey, Y. (1986). 'MLOGO: A MIDI Composing Environment for the Apple IIe'. *Proc. ICMC*, The Hague. 1986.

Orton, R. (1989). A Brief History of the Composer's Desktop Project'. in Endrich, T., Ed. *CDP Yearbook 1989*. CDP Ltd.

Orton, R. and Kirk, P. R. (1992). 'Tabula Vigilans'. *Proc. ICMC*, San Jose, CA. 1992.

Palmieri, G. and Sapir, S. (1992). 'MARS: Musical Applications'. *Proc. ICMC*, San Jose, CA. 1992.

Pape, G. (1992). 'Some Musical Possibilities of the new UPIC System'. *Proc. Science and Philosophy in Tomorrow's Music*, Delphi, Greece. 1992.

Pearson, M. (1993). 'Interim DPhil Research Report'. Music Technology Group, University of York

Pennycook, B., Kulick, J. and Dove, D. (1987). 'Music workstations: Real-time Subsystem Design'. *Proc. AES 5th International Conference - Music and Digital Technology*, Los Angeles. 1987.

Pennycook, B. W. (1985). 'Computer Music Interfaces: A Survey'. in *ACM Computing Surveys* 17(2) 1985. 267-289.

Pinkston, R. (1991). 'The University of Texas Accelerando Project: An Update'. *Proc. ICMC*, Montreal. 1991.

Polansky, L., Rosenboom, D. and Burk, P. (1987). 'HMSL: Overview (v. 3.1) and Notes on Intelligent Instrument Design.'. *Proc. ICMC*, Illinois. 1987.

Pope, S. T. (1986). 'The Development of an Intelligent Composer's Assistant - Interactive Graphic Tools and Knowledge Representation for Music'. *Proc. ICMC*, The Hague. 1986.

Pope, S. T. (1987). 'A Smalltalk-80 based Music Toolkit'. *Proc. ICMC*, Urbana, Illinois. 1987.

Pope, S. T. (1989). 'Machine Tongues XI: Object-Oriented Software Design'. in *Computer Music Journal* 13(2) 1989. 9-22.

Pope, S. T. (1992b). 'The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers'. in *Computer Music Journal* 16(3) 1992. 73-91.

Pope, S. T. (1992a). 'The SmoKe music representation, description language, and interchange format'. *Proc. ICMC*, San Jose. 1992.

Puckette, M. (1988). 'The Patcher'. *Proc. ICMC*, Cologne. 1988.

Puckette, M. (1991b). 'Combined Event and Signal Processing in the MAX Graphical Programming Environment'. in *Computer Music Journal* 15(3) 1991. 68-77.

Puckette, M. (1991a). 'FTS: A Real-time monitor for Multiprocessor Music Synthesis'. in *Computer Music Journal* 15(3) 1991. 58-67.

Puckette, M., Lippe, C. and Waxman, D. (1991c). *ISPW Max Reference Manual*. IRCAM.

Punch, B. (1991). 'An Algorithmic Approach to Composition based on Dynamic Hierarchical Assembly'. *Proc. ICMC*, Montreal. 1991.

Raczinski, J.-M. and G.Marino (1988). ' A Real-Time Synthesis Unit'. *Proc. ICMC*, Cologne. 1988.

Risset, J.-C. (1985). Digital Techniques and Sound Structures in Music'. in Roads, C., Ed. *Composers and the Computer*. Kaufmann.

Roads, C. (1985c). 'A.I. and Computers in Music'. in *ACM Computing Surveys* 17(2) 1985.

Roads, C. (1985b). 'The Realisation of 'nscor''. in Roads, C., Ed. *Composers and the Computer*. Kaufmann.

Roads, C. (1989).*The Music Machine: Selected readings from the Computer Music Journal*. MIT Press.

Roads, C. and Strawn, J. (1985a). *Foundations of computer music*. MIT Press.

Robson, D. (1981). 'Object-Oriented Software Systems'. in *Byte* 6(8) August 1981. 74-86.

Rodet, X. (1991). 'What Would We Like to See Our Music Machines Capable of Doing?'. in *Computer Music Journal* 15(4) 1991. 51-54.

Rodet, X. and Cointe, P. (1984). 'FORMES: Composition and Scheduling of Processes'. in *Computer Music Journal* 8(3) 1984. 32-50.

Rodet, X. and G.Eckel (1988). 'Dynamic Patches: Implementation and Control in the Sun-Mercury Workstation'. *Proc. ICMC*, Cologne. 1988.

Roland (1988). *D110 Synthesiser operating manual*. Roland Corp.

Rolnick, N. B. (1986). 'A Performance Literature for Computer Music: Some Problems from Personal Experience'. *Proc. ICMC*, The Hague. 1986.

Rona, J. (1989). 'The Medialink LAN'. in *Keyboard* December 1989. 54-59.

Russ, M. (1992). 'Tomorrow's World'. in *Audio Media* (20) 72.

Scaletti, C. (1989). 'The Kyma/Platypus Computer Music Workstation.'. in *Computer Music Journal* 13(2) 1989. 23-37.

Scaletti, C. (1991). 'Lightweight Classes Without Programming'. *Proc. ICMC*, Montreal. 1991.

Scaletti, C. (1992). 'Polymorphic transformations in Kyma'. *Proc. ICMC*, San Jose. 1992.

Scaletti, C. A. (1987). 'Kyma: An Object-oriented Language for Music Composition'. *Proc. ICMC*, Illinois. 1987.

Scaletti, C. A. and Johnson, R. E. (1988). 'An Interactive Environment for Object-oriented Music Composition and Sound Synthesis'. *Proc. OOPSLA*, 1988.

Schmidt, B. L. (1987). 'Natural Language Interfaces and their Application to Music Systems'. *Proc. AES 5th International Conference - Music and Digital Technology*, Los Angeles. 1987.

Schmucker, K. (1986). *Object-Oriented Programming for the Macintosh*. Hayden Book Co.

Scholz, C. (1991). 'A Proposed Extension to the MIDI Specification Concerning Tuning'. in *Computer Music Journal* 15(1) 1991. 49-54.

Schottstaedt, B. (1983). 'Pla: A Composer's Idea of a Language'. in *Computer Music Journal* 7(1) 1983. 11-20.

Shimony, U., S.Markel and J.Tal (1988). 'Icon Notation for Electroacoustic and Computer Music'. *Proc. ICMC*, Cologne. 1988.

SIGPLAN (1988). 'Common Lisp Object System'. in *SIGPLAN notices* (September 1988) 1988.

Smalley, D. (1986). 'Spectro-morphology and Structuring Processes'. in Emmerson, S., Ed. *The Language of Electroacoustic Music*. Macmillan Press. 61-96.

Smith, J. O. (1989). 'Unit Generator Implementation on the NeXT DSP chip'. *Proc. ICMC*, Ohio. 1989.

Smith, J. O. (1991). 'Viewpoints on the History of Digital Synthesis - Keynote Paper, ICMC-91'. *Proc. ICMC*, Montreal. 1991.

Smith, J. O., Jaffe, D. and Boynton, L. (1989). 'Sound and Music on the NeXT Computer'. *Proc. AES 7th International Conference - Audio in Digital Times*, Toronto. 1989.

Starkier, M. (1986). 'Real-time Gestural Control'. *Proc. ICMC*, The Hague. 1986.

Strasburger, H., Kohler, S. and Radauer, I. (1990). 'Score Input to CSound via the MIDI keyboard'. *Proc. ICMC*, Glasgow. 1990.

Stroustrup, B. (1987).*The C++ Programming Language*. Addison Wesley.

Symbolic Sound Corporation (1992). *The Kyma Language for Sound Specification*. Symbolic Sound Corporation.

Tanenbaum, A. S. (1984). *Structured Computer Organisation*. Prentice-Hall International, Inc.

Tello, E. R. (1987). 'Smalltalk/V'. in *Dr. Dobbs's Journal* September 1987.

Tesler, L. (1981). 'The Smalltalk Environment'. in *Byte* 6(8) August 1981. 90-147.

Tesler, L. (1986). 'Programming Experiences'. in *Byte* 11(8) August 1986. 195-206.

Thomas, D. (1989a). 'The Time/Space Requirements of Object-oriented Programs'. in *Journal of Object Oriented Programming* 2(2) 1989. 71-73.

Thomas, D. (1989b). 'What's in an Object?'. in *Byte* March 1989. 231-240.

Tipei, S. (1987). 'Maiden Voyages: A Score Produced with MP1'. in *Computer Music Journal* 11(2) 1987. 49-58.

Tobenfeld, E. (1984). 'A Keyboard controlled sequencer'. in *Computer Music Journal* 8(4) 1984.

Tobenfeld, E. (1993). *KCS Omega II*. Dr. T's Software.

Tollinen, P. and Morgan, N. (1993). *Symbolic Composer*. Tonality Systems.

Toop, R. (1985). 'Brian Ferneyhough in Interview'. in *Contact - a Journal of Contemporary Music* (29) 1985. 4-19.

Truax, B. (1986). 'Computer Music Language Design and the Composing Process'. in Emmerson, S., Ed. *The Language of Electroacoustic Music*. Macmillan Press. 155-174.

Truax, B. (1986). 'Real-Time Granular Synthesis with the DMX-1000'. *Proc. ICMC*, The Hague. 1986.

Udell, J. (1990). 'Smalltalk-80 enters the Nineties'. in *Byte* 15(10) October 1990. 138-142.

Varèse, E. (1966). 'The liberation of sound'. in Boretz, B. and Cone, E., Ed. *Perspectives on American Composers*. W.W. Norton & Co.

Vercoe, B. (1986). *CSOUND Reference Manual*. MIT Press.

Vercoe, B. and Ellis, D. (1990). 'Real-Time CSound: Software Synthesis with Sensing and Control'. *Proc 'CMC*, Glasgow. 1990.

Viara, E. (1991). 'CPOS: A Real-time Operating system for the IRCAM Musical Workstation'. in *Computer Music Journal* 15(3) 1991. 50-57.

Viara, E. and Puckette, M. (1990). 'A real-time operating system for computer music'. *Proc. ICMC*, Glasgow. 1990.

Walraff, D. (1979). 'The DMX-1000 signal processing computer'. in *Computer Music Journal* 3(4) 1979. 44-49.

Waugh, I. (1989). 'X-OR (review)'. in *Music Technology* November 1989. 32-36.

Wegner, P. (1989). 'Learning the Language'. in *Byte* 14(3) March 1989. 245-253.

Wessel, D., D.Bristow and Z.Settel (1987). 'Control of Phrasing and Articulation in Synthesis'. *Proc. ICMC*, Illinois. 1987.

Wessel, D., Lavoie, P., Boynton, L. and Orlarey, Y. (1987). 'MIDI-LISP: A LISP-Based Programming environment for MIDI on the Macintosh'. *Proc. AES 5th International Conference - Music and Digital Technology*, Los Angeles. 1987.

Westfall, L. (1989). 'The LAN solution'. in *Keyboard* November 1989.

Winckel, F. (1967). *Music, Sound and Sensation*. Dover Publications, Inc.

Wirfs-Brock, R. (1992). 'Object-Oriented Design: Becoming more predictiable'. in *The Smalltalk Report* 1(6) 1992. 8-10.

Wirfs-Brock, R. (1992). 'Object-Oriented Design: Determining object roles and responsibilities'. in *The Smalltalk Report* 1(4) 1992. 6-8.

Wishart, T. (1985). *On Sonic Art*. Imagineering Press.

Wishart, T. (1986). Sound Symbols and Landscapes'. in Emmerson, S., Ed. *The Language of Electroacoustic Music*. Macmillan Press. 41-60.

Xenakis, I. (1971). *Formalised Music*. Indiana University Press.

Xerox Learning Research Group (1981). 'The Smalltalk-80 System'. in *Byte* 6(8) August 1981. 36-48.

Yavelow, C. (1985). 'Music software for the Apple Macintosh'. in *Computer Music Journal* 9(3) 1985. 52-67.

Yavelow, C. (1986a). 'The Impact of MIDI upon Compositional Methodology'. *Proc. ICMC*, The Hague. 1986.

Yavelow, C. (1986b). 'MIDI and the Apple Macintosh'. in *Computer Music Journal* 10(3) 1986. 11-47.

Bibliography

Books

Bregman, A. S. (1990). Auditory Scene Analysis. MIT Press.

Chamberlin, H. (1980). Musical Applications of Microprocessors. Hayden Book Co. Inc.

Clarke, E. and Emmerson, S., Ed. (1989). Music, Mind and Structure. Harwood Academic Publishers.

Cole, H. (1974). Sounds and Signs. Oxford University Press.

Cox, B. J. (1986). Object-oriented programming - An Evolutionary Approach. Addison Wesley.

Dobrian, C. and Zicarelli, D. (1990). MAX - Manual. Ircam / Opcode Systems, Inc.

Dodge, C. and Jerse, T. A. (1985). Computer Music: Synthesis, Composition and Performance. Macmillan.

Emmerson, S., Ed. (1986). The Language of Electroacoustic Music. Macmillan Press.

Endrich, T., Ed. (1989). CDP Yearbook 1989. CDP Ltd.

Goldberg, A. (1984). Smalltalk-80: The Interactive Programming Environment. Addison Wesley.

Goldberg, A. and Robson, D. (1983). Smalltalk-80: The Language and its Implementation. Addison Wesley.

Graham, B. (1980). Music and the Synthesiser. Argus Books.

Griffiths, P. (1979). A Guide to Electronic Music. Thames and Hudson.

Kaehler, T. and Patterson, D. (1986). A Taste of Smalltalk. W.W. Norton & Co.

Krasner, G. E., Ed. (1983). Smalltalk-80: Words of History, Bits of Advice. Addison-Wesley.

LaLonde, W. R. and Pugh, J. R. (1990). Inside Smalltalk (Vol. 1). Prentice Hall.

LaLonde, W. R. and Pugh, J. R. (1990). Inside Smalltalk (Vol 2). Prentice Hall.

Mathews, M. (1969). The Technology of Computer Music. MIT press.

Mathews, M. and Pierce, J., Ed. (1989). Current Directions in Computer Music Research. MIT Press.

Roads, C., Ed. (1985). Composers and the Computer. Kaufmann.

Roads, C. (1989). The Music Machine: Selected readings from the Computer Music Journal. MIT Press.

Roads, C. and Strawn, J., Ed. (1985). Foundations of computer music. MIT Press.

Schmucker, K. (1986). Object-Oriented Programming for the Macintosh. Hayden Book Co.

Symbolic Sound Corporation (1992). The Kyma Language for Sound Specification. Symbolic Sound Corporation.

Stroustrup, B. (1987). The C++ Programming Language. Addison Wesley.

Tanenbaum, A. S. (1984). Structured Computer Organisation. Prentice-Hall International, Inc.

Boretz, B. and Cone, E., Ed. (1966). Perspectives on American Composers. W.W. Norton & Co.

Winckel, F. (1967). Music, Sound and Sensation. Dover Publications, Inc.

Wishart, T. (1985). On Sonic Art. Imagineering Press.

Xenakis, I. (1971). Formalised Music. Indiana University Press.

Conference Proceedings

Proceedings of the AES 5th International Conference - Music and Digital Technology, Los Angeles. 1987.

Proceedings of the AES 7th International Conference - Audio in Digital Times, Toronto. 1989.

Proceedings of the Eurographics UK Conference, York. 1993.

Proceedings of the Institute of Acoustics Conference, Windemere. 1988.

Proceedings of the International Computer Music Conference, The Hague. 1986. ICMA

Proceedings of the International Computer Music Conference, Illinois. 1987. ICMA

Proceedings of the International Computer Music Conference, Cologne. 1988. ICMA

Proceedings of the International Computer Music Conference, Ohio. 1989. ICMA

Proceedings of the International Computer Music Conference, Glasgow. 1990. ICMA

Proceedings of the International Computer Music Conference, Montreal. 1991. ICMA

Proceedings of the International Computer Music Conference, San Jose. 1992. ICMA

Proceedings of the International Computer Music Conference, Tokyo. 1993. ICMA

Proceedings of OOPSLA, 1988.

Proceedings of the Science and Philosophy in Tomorrow's Music Conference, Delphi, Greece. 1992.

Proceedings of the Workshop on Music Education: An Artificial Intelligence Approach, AI-ED 93, Edinburgh. 1993.

Journals and Periodicals

ACM Computing Surveys 17(2) 1985.

Audio Media, SOS Publications

Byte Aug 81; Aug 86; Mar 89; Oct 90; Aug 91; Dec 91.

Computer Jul 91.

Computer Music Journal.

Dr. Dobbs's Journal Sept 87.

IEEE Software May 89.

Journal of the Audio Engineering Society.

Journal of Object Oriented Programming. SIGS Publications, Inc., New York

Musicus - Computer Applications for Music Education. CTI Centre for Music

SIGPLAN notices Sep 88.

The Smalltalk Report. SIGS Publications, Inc., New York

Index