

2. Sound in human-computer interaction

Black is externally the most toneless colour, against which all other colours, even the weakest, sound stronger and more precise. Not so with white, compared with which the sound of nearly all other colours becomes dulled, while many dissolve completely, leaving nothing but a weak, paralyzed echo behind them.

– Kandinsky: *On the Spiritual in Art*

2.1. Introduction

With the continued development of visual display technology have come increasingly sophisticated tools and methods for providing information to the computer user. The computer front-panel controls were superseded by keyboards connected to teleprinters. The printers were replaced by cathode-ray-tube monitors. In turn, these early text only monitors were refined to allow graphical images to be displayed. Then colour technology replaced the monochrome screens. The low resolution colour displays were superseded by higher and higher resolution devices. Today we are able to place on our desk tops machines which can render photo-realistic images in millions of colours and manipulate these images to produce animated sequences.

As a result of all this increased capacity there have been developed many applications that attempt to make use of the technology's capabilities. The graphical user interface (GUI) has replaced the text-based display in all but a handful of cases. Pro-

Programming language development environments routinely employ colour syntax highlighting to make reading program text easier.

One of the main areas to benefit from advances in graphical technology has been that of software visualisation. Visualisation tools assist users in finding their way through large databases, making sense of complex data sets and comprehending the run-time behaviour of programs. This chapter briefly describes some of the attempts to help users visualise information and discusses how the emergence of improved computer audio technology is helping to improve human-computer interaction.

2.2. Understanding software through visualisation

In formal research the term ‘software visualisation’ suggests the idea of investigation using the visual sense alone. However, the aim of software visualisation is simply to improve the understanding of software [54]. In this field of study much has been made of the ability of graphical techniques to assist users and programmers in understanding and interpreting the workings of software.

Baecker and Buchanan [11] showed that systems dating back even to the 1960s claimed to use visualisation techniques. Animation, one of the more popular techniques for demonstrating how algorithms and programs function has been developed for visualisation purposes since 1966 (see the work of Knowlton [90], Baecker [9] and Hopgood [79]). Baecker’s colour animated sound film *Sorting out Sorting* [10] is a good example of animation techniques being used to assist people in understanding how sorting algorithms work.

Baecker and Buchanan [11] developed a visualisation system called LOGOmotion which was a set of commands to allow program events to be visualised using graphical windows. An interesting conclusion they draw is that visualisation can be used in teaching, for animation can convey aspects of a program’s execution which static representations cannot. From this work Baecker and Buchanan drew a number of conclusions:

- Few visualisation systems can be applied to arbitrary programs; they are mostly restricted to a certain language or a particular application area;
- Visualisations included both static and animated displays;

- Generally, arbitrary visualisations are not possible; the set of display systems is fixed or only partially configurable, meaning that only certain aspects of certain types of program may be visualised;
- Visualisation systems are typically *invasive*, that is, they require modification of the source program to achieve their results.

A comparison of five systems is given by Baecker and Buchanan as Table 2.1. The systems in question are: London and Duisberg's Animus [100], Brown and Sedgewick's Balsa II [38], Bentley and Kernighan's Movie & Stills [14], Eisenstadt and Brayshaw's Transparent Prolog Machine (TPM) [60] and Baecker and Buchanan's LOGOmedia [11].

	Animus	Balsa II	Movie & Stills	TPM	LOGOmedia
Domain of use	SmallTalk	Pascal	Any	Prolog	Logo
Special purpose knowledge and built-in visual enhancements	Some	Very many	None	Many	Some
Animation possible	Yes	Yes	Yes	Yes	Yes
Extensible: visualisations can be programmed	Yes	Yes	User-tailorable	No	Yes
Monomorphic: host language used to define visualisations	Yes	Yes	No	No	Yes
Unobtrusive: visualisations required and defined without altering source code	No	No	No	Yes	Yes

Table 2.1 Comparison of program visualisation systems

Pancake and Utter [127] discussed several models of visualisation used in a range of parallel debuggers. Though useful, limitations in the interface design tended to make the systems clumsy to use.

Baker and Eick [12] carried out work which involved visualising statistics associated with code that is divided hierarchically into subsystems, directories and files. Again, like many others, they used animation to display the historical evolution of code. They identified three guidelines for visualising large software systems:

1. The structure of the display should reflect the structure of the software.

2. Individual components should be visually composable, comparable and decomposable.
3. Animation can be used to depict the evolution, or change history, of the software.

2.3. Sound as an interaction medium

With the exception of games and so-called multimedia applications the audio channel remains little used in most computing applications. This is especially true in the area of HCI. We have compared the role of the HCI designer to that of the music composer [7] for both are attempting to provide to their respective users access to their work and to present their work as an integrated and self consistent whole. However, unlike composers, many of whom have complemented their musical performances with visual displays, the HCI designers have for the most part ignored the audio channel. A browse through the most popular journals, conference proceedings and edited books dedicated to the study of computer-human interaction reveals a dearth of sound-related research. Where sound is employed its usage is often trivial (such as the arbitrary association of sound files with system events in Microsoft's Windows-95 operating system). For many users such application of sound offers nothing more than entertainment whose novelty value soon wears off. More useful is the practice employed by some electronic mail systems of signalling the arrival of new mail by an audible tone or sound effect. However, when compared with the sophistication of some visual display techniques the current use of sound in the interface seems meagre.

Nevertheless, sound offers much to the process of human-computer interaction. Since Bly's original thesis [19] on the use of sound in interfaces, a few researchers (particularly in visualisation) have been keen to propose that sound be used in software visualisations [11, 24, 37, 52, 82, 148].

Though many people tend not to think in terms of sound, auditory signals do help us to build mental images. Objects in the real world have sounds associated with them. We can estimate the size of a dog by hearing its bark. By listening to the sound of a racing car through stereo headphones, one easily imagines the car going round the racetrack; we do not need to see the car to build a mental image of it.

The lack of attention paid to the audio channel in HCI (and in visualisation in particular) is paralleled by the imbalance between the study of visual and auditory imagery in the cognitive sciences. Crowder and Pitt state:

“But this absolutely thundering silence on imagery in music is in contrast with two factors. First, in visual cognition and perception, the issue of imagery has become increasingly central in the last two decades. Whole books on the topic have appeared, as has a journal, and in the larger domain of the cognitive sciences, imagery is recognised as a defining problem. One of us has commented elsewhere on the lag between visual and auditory scholarship.” [49]

Crowder and Pitt define imagery as being the “*activation of the same central neural systems that played a role in the original event, but this time in the absence of the original sensory activity*” [49] (e.g., the itching and pain in phantom (amputated) limbs). A “*useful model for memory in general is the persisting activity of neural centres involved in the original experiences. Imagery, as a form of memory, would fall within that position*” [49].

For most people, the notion of auditory imagery is easily accepted. Halpern [72] reports that people tend to ask how they can *stop* tunes from running through their head. The imagining of the sound of chalk or fingernails being scraped down a blackboard conjures up a very vivid auditory image for most of us.

In software visualisation, mappings are made between software attributes and graphic devices (such as graphs, spectra etc.). Such mappings provide a framework for users to construct mental images of the states and structures of the intangible, or unmalleable, features of software. This often works extremely well for components whose features map naturally onto spatial media (such as tree diagrams for genealogy reports or dynamic data structures). Having constructed the mapping between, say, a linked-list data structure and a tree diagram, the computer science student forever has a visual analogue of the data structure which can be abstracted and used to form mental images without recourse to pen and paper. So trained, when asked to visualise a node being added to the list (Figure 2.1), the student can form an image in their head and so draw a picture showing the new state of the data structure. This visual representation has no direct connection with the actual data structure, it is merely an apt analogue or metaphor (after all, we talk of *binary tree structures*).

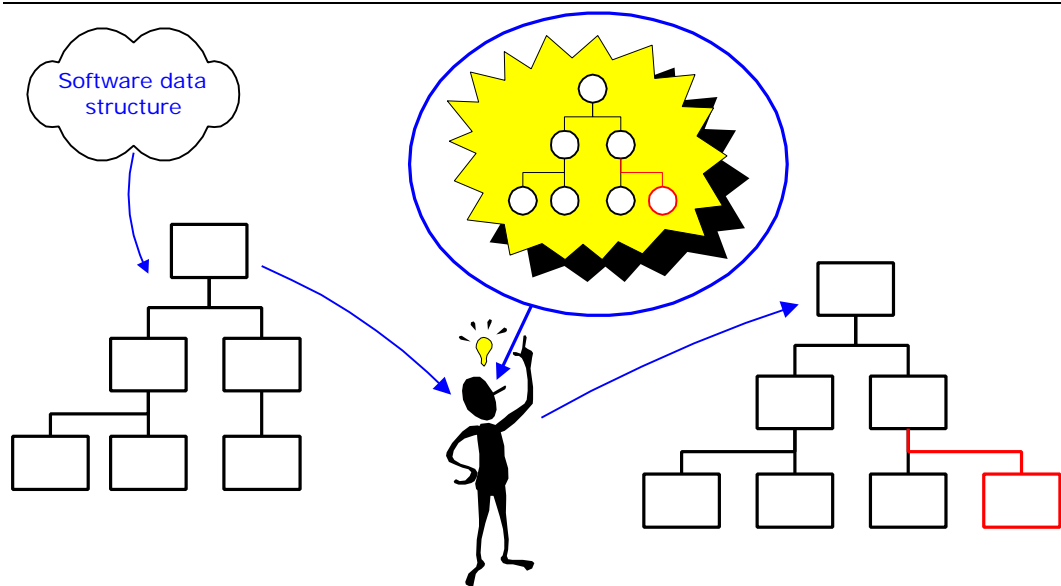


Figure 2.1 Metaphor in visualisation

Using the tree metaphor for data structures aids understanding. Once a structure is visualised as a tree then it is easy to visualise nodes being added and removed. The tree metaphor is widely used in computing for modelling program and data structures.

The aspects of software in which the programmer is interested tend not to be real world objects and thus have no real world auditory signatures. How then can sound be successfully employed in human-computer interaction? One need only look to the works of some famous composers to see how abstract ideas can be effectively expressed through sound or music. Stravinsky's *Rite of Spring* very clearly conjures up images of primal pagan ritual; many people find the work unsettlingly convincing.

When describing the process of visualisation we talk about seeing things with the mind's eye and not about hearing things with the mind's ear. Of course, while most people would say, when they understand you that they 'see what you mean', others will say that they 'hear what you mean' (and still a few more will feel it). When talking of our mental processes we use visual metaphors. Although sound is not visible we are still able to construct mental images when presented with particular sounds or pieces of music.

More pertinent, perhaps, is the use of multi-sensory methods to teach dyslexic children. Because dyslexics have difficulties seeing words and letters correctly, sight, sound and touch are used to help them to see, hear and feel the ideas being taught. That sound can be used in the teaching of ideas strengthens the case for the use of sound in the programming environment.

Therefore, it makes sense to use sound if it possesses properties that facilitate software comprehension or make the process of interacting with computers easier. Brown, Newsome and Glinert [39] showed that complex auditory cues could be used to replace cues more traditionally presented visually. They undertook a study to determine whether information that is typically presented visually could be communicated effectively using sound. The experiment centred around subjects being able to locate a target character string on a computer screen using a mixture of aural and visual cues. The results showed that subjects were as successful with audio cues as they were with visual cues. Furthermore, they found that the human brain can extract multiple messages from a sound very quickly and then act on the information given. This property of sound will be examined in more detail later.

The motivation behind their research was to see whether sound could be used to reduce visual workload. As user interfaces become more and more sophisticated, so more and more information can be presented on the monitor screen. The problem is one of visual overload where the user can no longer take in all the information being presented. Such research suggests that sound is a useful tool in the presentation of information to users.

By way of summary, Cohen [45] offers the following reasons for adopting sound to notify users of events:

- Audio does not take up screen space.
- Audio fades into the background but users are alerted when it changes (q.v. chapter 4).
- People can process audio information while simultaneously engaged in an unrelated task (e.g. listening to the radio whilst writing a paper).
- The well-known cocktail party effect [8] (the ability to selectively attend to one conversation in the midst of others in a crowded room) allows users to monitor multiple background processes via the audio channel so long as the sounds attributed to each process can be distinguished.
- Most direct manipulation tasks are visual leaving the audio channel free.

2.4. Auditory display

During the past ten years or so a body of research dedicated to the use of sound in HCI has been building up steadily. This new field of research, termed *auditory display*, was described in 1994 by Kramer as embryonic [92]. The name derives from the attempt to use sound to present information that has previously been restricted to the visual medium. We present Figure 2.2 as an overview of the field's contributing disciplines.

Though the field of auditory display is a young one in terms of formal research effort, Cohen [45] argues that it was the composer John Cage who first put forth the principles of the field in the 1950s. Cohen cites Cage's works *Music of Changes* (1952) and *Reunion* (1968) as early examples of data sonification. In *Music of Changes*⁵ the score was composed by mapping the results of coin tosses to pitch, duration, amplitude and timbre. Even changes in tempo and the number of measures in a given section were controlled by the data generated from coin tosses. *Reunion* developed the idea by using photoelectric switches on a chessboard to trigger the playing of different pieces of music. Whether Cage intended to communicate information regarding data sets by music or whether he merely used data as a mechanism for the creation of new music (i.e., was music a by-product or the intentional product) is a moot point; what is interesting is that Cage considered that the relationship between music and data can be exploited.

Though new and with few participants (in comparison to more established disciplines) the types of research carried out in auditory display are fairly diverse. However, they can be said to fall broadly into two camps: the use of audio in the interface and the use of sound in visualisation. This second category can be subdivided into two further classifications: data sonification/audification⁶ and algo-

⁵ See Cage's book *Silence* [35] for a description of how he composed the *Music of Changes*.

⁶ *Sonification* is the process of mapping properties of data or events to sounds in an attempt to represent the data or events in the audio channel. *Audification* is similar but rather than using mappings, data are played back directly, e.g., scaling seismic data up until their values lie in the audible frequency range. For more complete definitions refer to the glossary in appendix A.

rhythm/program auralisation⁷ (also sometimes called audiolisation). The first of these is concerned with the use of sound to represent data whilst auralisation aims to assist the comprehension of software by representing its state audibly thereby creating an auditory image.

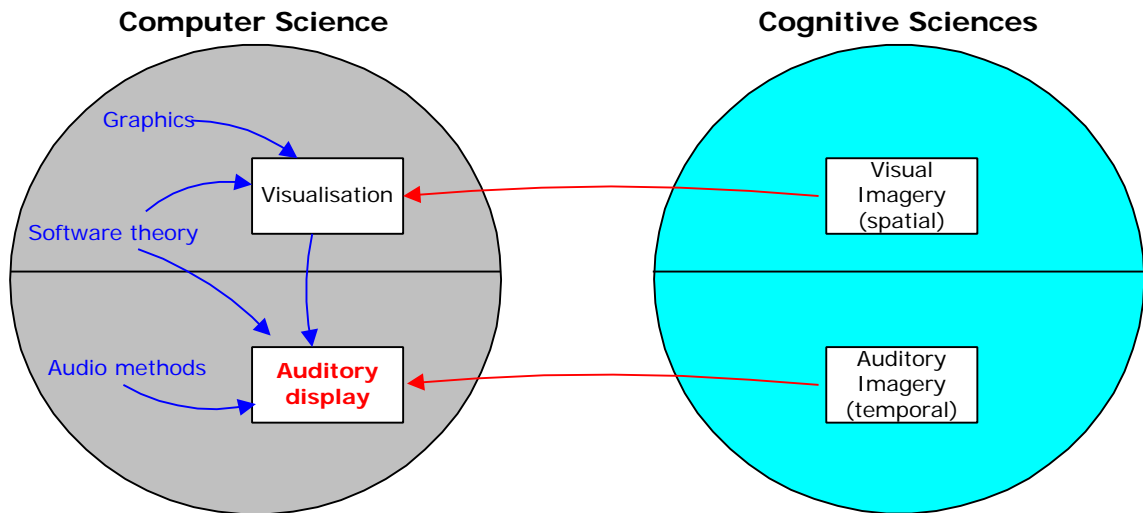


Figure 2.2 The emerging discipline of auditory display

Much effort has gone into graphical visualisation research in computer science and this work is informed by complementary research in cognitive science. More recently, the idea of auditory imagery has attracted attention in the cognitive fields. Auditory display draws upon this work and upon work in the audio engineering/sound production fields to allow the communication by sound of information and software features.

We suggest a general taxonomy of auditory display may be constructed along these lines (see Figure 2.3). Auralisation, the area of concern of this thesis, is shown in Figure 2.3 in bold type. There is an argument to make auralisation a sub-category of sonification. Indeed, Kramer [93] observes that auralisation is often used as a synonym of sonification (which is probably because of its kinship with visualisation). However, we have made the distinction between auralisation and sonification in defining the taxonomy because we feel that sonification is concerned with the auditory display of generic data, whilst auralisation is more properly about the visualisation of programs and algorithms which may involve the auditory display of certain data associated with a program. In the case of auralisation the data concerned are more usually the internal items of the program.

⁷ *Auralisation*, a term suggested by Brown and Hershberger [37], typically refers to the mapping of program data to sound and is based on the execution of the program or algorithm in question. Refer to the glossary for a more complete definition.

The boundary between interface-related and visualisation-related applications is not always wholly clear. Whereas some projects very clearly fall into one or another category, such as Edwards' audio-enhanced word-processor [58] which is obviously an application of audio in the interface, others do not. The LogoMedia system, described by DiGiano and Baecker [53], offers a form of program auralisation, but also extends the interface by adding sound effects to the process of keying in source code. Of the various strands of auditory display the majority of research effort has gone into interface applications and data sonification.

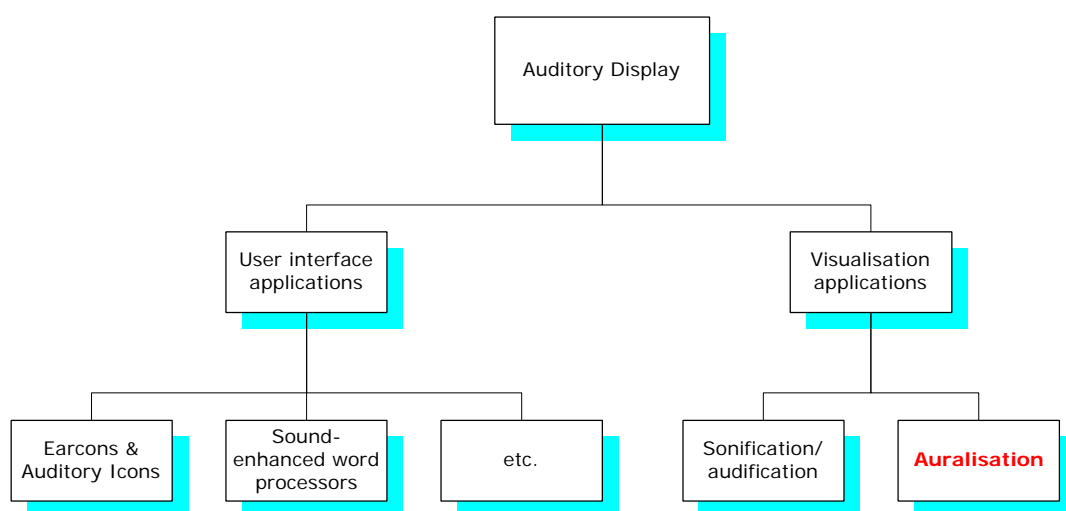


Figure 2.3 Taxonomy of auditory display

This classification is by no means complete, rather it offers exemplars of the two main strands of auditory display research—applications in the human-computer interface and visualisation techniques.

2.4.1. User interface applications

There are several good examples of applications of sound in the user interface. Edwards [58] applied audio technology to develop an interface for a word-processor for those with visual impairments. His *Soundtrack* system adapted a mouse-based interface into one which employed audio techniques. The system used a combination of synthesised speech and square waves of differing musical pitches. When a menu was selected by a mouse click the interface 'spoke' the menu's name. Moving the mouse up and down the menu's options caused changes in the pitch of a note. Evaluation gave broadly favourable results, though subjects had some difficulties using the system. The main problem reported by users was recalling the layout of the internal structures of the windows. The users tended to use their own mnemonic system to remember the arrangement of the windows on the screen. Further, rather

than using the pitch information as a navigation aid, the users instead tended to count the number of tones heard in order to locate the cursor's screen position. Rigas and Alty [136] made use of this factor in their musical diagram reader for the blind (Audiograph) which grouped sequences of pitches into octave groups with pauses between each group. Thus, a series of four-and-a-half octave runs could easily be interpreted as much greater distance across the screen than a series of only one-and-a-half octaves.

Mynatt [122] described *Mercator* a sound-enhanced X-Windows graphical user interface for blind users. Essentially, *Mercator* provides an XMH auditory interface component library to complement the existing visual XMH widget hierarchy. Sound mappings were non-musical, that is, they employed the symbolic mappings of auditory icons. Tests indicated that non-sighted users could successfully navigate around the GUI using the system. Evaluation was done with a small number of subjects (unfortunately, too few to allow thorough statistical analysis). The main limitation that Mynatt identified was the inability of users to identify the auditory cues. The auditory icons were differentiated by applying various filters to a set of sounds. The identification problem was caused by the duration of the auditory icons being too short to be able to discern the filtering characteristics.

Cohen's *ShareMon* [45] and *OutToLunch* [46] (section 2.4.1.5) systems showed how background computer activities could notify users of relevant events (including relative activity of other computer users) by using auditory cues rather than visual messages.

Gaver's *SonicFinder* system [67] employed sound effects in the form of auditory icons (q.v.) to enhance the *Finder* file management program for the Apple Macintosh computer. Operations like copying, selecting or deleting files are each represented by an auditory icon. Gaver used arbitrary mappings such as 'hitting' sounds for object selection operations, scraping sounds for dragging operations and pouring sounds for copying operations. *SonicFinder* is discussed in more detail in section 2.4.1.2. Much of the research dealing with HCI and sound is concerned with *earcons* and *auditory icons*, so we will now briefly discuss those two approaches.

2.4.1.1. Earcons

An earcon is a brief succession of musical pitches structured to transmit specific items of information to the computer user. The term was first suggested by Buxton, Baecker and Arnott in 1985 [41] and was more formally defined by Blattner et al [17]. Brewster [29, 31, 32, 33, 34, 35] has since specified formal earcon design principles and has carried out studies into their usefulness.

The earcon is so-called because it is the aural counterpart to the icon. Earcons have been used in the interface for tasks such as auralisation of data associated with turbulence in fluid flow to enhancing two dimensional maps by associating sound cues with various topographical features [15, 16, 17]. Studies have shown earcons to be effective in communicating information to users [35, 103]. Lucas [103] showed that the accuracy of recognition of the auditory cues was increased when users were made aware of the design principles of the earcons used.

Since its introduction Brewster and colleagues have used the earcon to provide navigation cues in menu hierarchies [31], to make GUI widgets (such as buttons, menus and scrollbars) more useable [30, 48, 98] and to reduce the length of audio messages by using *parallel earcons* [32]. Experiments showed that the time taken to successfully operate such interface components was significantly reduced when the tasks were enhanced by the addition of earcons. Furthermore, a reduction in the mental workload of the subjects (as measured by the NASA Task Load Index [74]) was observed. When earcons were applied to drag and drop activities [30] a significant reduction in time taken and mental workload was similarly observed.

Earcons are simple musical devices that are mapped to computer objects, operations and interactions. The underlying design principle of earcons is that they are constructed from small musical building blocks, or motifs. The motifs are short sequences of musical pitches arranged in such a way as to make them distinct and recognisable. A hierarchy of earcons is then constructed from the motifs to create the audio messages for the particular application.

For example, consider the earcon hierarchy shown in Figure 2.4 (taken from Brewster [29]). This is the family tree of earcons representing various errors. The root motif is a structure comprising a single pitch (middle A) of indeterminate length. The two subclasses of error, (operating system and execution) inherit this structure but modify the timbre used in order to distinguish themselves from each

other. Instances of these subclasses (e.g. overflow and underflow) inherit the timbre from their parent and are distinguished by melodic and rhythmic differences.

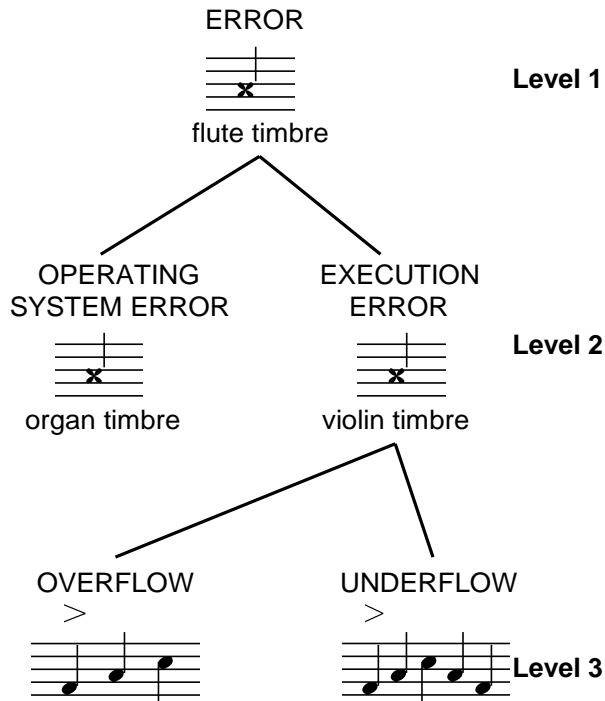


Figure 2.4 An earcon hierarchy

We can see how each level of this hierarchy (from Brewster [29]) inherits characteristics from its parents. The underflow operation inherits the violin timbre from its parent and differs from its sibling in its melodic contour. However, it is not clear what purpose the flute timbre serves at the top of the hierarchy.

Using design principles such as these, earcons have been found to be effective in communicating hierarchical information down to four levels (for example, in telephone-based interfaces [31]).

There is another kind of earcon known as compound earcons. In this approach, various interface and computer objects and operations have an earcon assigned to them and these are combined to represent actions on the objects. For example, if a file was represented by a one-note earcon played using a violin sound, and the delete action by a descending two-note earcon played with an organ sound, then the action *delete file* would be represented by the compound earcon that is achieved by sounding the *delete* and *file* earcons in sequence.

2.4.1.2. Auditory icons

The *auditory icon*, first suggested by Gaver [66, 67, 69] is like the earcon, but whereas the earcon is based around a hierarchical structure of musical melody and

rhythm, the auditory icon simply uses natural, everyday sounds to carry information. Gaver's experience leads him to believe that auditory icons can provide useful information about computer system events within a complex environment [68]. However, he also notes that sounds in the interface can be annoying and that *what "seems cute and clever at first may grow tiresome after a few exposures"* [69].

Where earcons are metaphoric (the mapping between earcon and object is arbitrary and must be learnt) auditory icons are analogic (their representations mimic the information being presented). For example, the auditory icon for dragging an item across the GUI desktop might be a scraping noise.

The most famous application of auditory icons in the interface was Gaver's *SonicFinder* [67]. In this system, Gaver added sounds to the components of the Apple Macintosh's *Finder* GUI. Files were represented by wood-based noises, applications by metallic noises and folders by paper-like sounds. Various modifications of the basic audio parameters were made to represent different features of the interface. For instance, the larger a file, the deeper its sound.

One problem with analogic mappings is that some objects and actions have no real-world analogue. In the *SonicFinder* Gaver used sound of liquid being poured into a container to denote copying a file— the rising pitch heard as the container fills up is supposed to represent the progress of the copying activity. Such a mapping is quite arbitrary as in the real world, pouring of liquids is not a sound associated with making copies of things. As the sounds become more metaphorical and less analogical, there is a danger that their usefulness will wane. Because there is no underlying structure to govern the mapping of sounds to interface components, users must memorise an increasingly large number of discrete sounds. By contrast, the earcon's metaphor is based on a hierarchy. In a hierarchic approach all components share characteristics meaning that fewer discrete mappings need to be learnt. It is probably for this reason more than any other that auditory icons have been left behind by earcons in formal research.

There were two other notable applications of the auditory icon— the *SharedARK* and *ARKola*.

2.4.1.3. SharedARK

Gaver and Smith [69] took auditory icons into large-scale collaborative environments in SharedARK (Shared Alternative Reality Kit). In SharedARK multiple users were able to interact simultaneously with objects in an environment that extended beyond the boundaries of their monitor screens. Auditory icons were introduced to provide confirmation of user actions and information about system state and ongoing processes. The auditory icons also gave navigational information and signalled the presence and activities of other users working in parts of the system that were not visible. The auditory icons were mostly sampled environmental sounds and synthesised musical instrument tones were not included.

Confirmatory sounds included:

- Clicks when buttons were pressed.
- Machine noises to indicate processing of user instructions
- Popping sounds to show when new objects appeared in the system (for instance, as a result of executing a copy command).

System state was communicated by having continuous sounds assigned to various processes. These sounds would fade out over time so as not to be distracting but could be reactivated by the user as required.

Gaver and Smith believed that using environmental sounds that had a semantic relationship with their referents was the best approach for such systems. However, it is not hard to imagine examples of system objects, commands or processes that do not have a good semantic mapping with real-world sounds. At this point the mappings become increasingly metaphorical.

2.4.1.4. ARKola

Gaver, Smith and O'Shea's ARKola project [70] simulated a soft drinks bottling plant using the SharedARK system. The ARKola factory had nine interconnected machines, eight of which were under user control. Five machines provided the raw ingredients to the four processing machines. Each machine had an on/off switch and a control to alter its rate of output. The simulation required users to produce bottled cola without wasting materials by ensuring a correct balance between the rates of the various machines. Occasionally, machines would break down and require repair.

Money was made when bottles were shipped, and spent when supplies were bought. Users were asked to make a profit by the efficient mass production of soft drinks.

Auditory icons were used to enable each machine to indicate its status over time, and a semantic mapping was created between machine and auditory icon. For example, the heating machine made a *'whooshing sound like that of a blowtorch, the bottle dispenser made the sound of clanking bottles, etc.'* The speed and rhythm of the sounds indicated the operation rate of the machine. If a machine exhausted its supplies or broke down then its auditory icon stopped. Material wastage was indicated by other sounds. A splashing sound told users that liquid was being spilt, and lost bottles were represented by the sound of breaking glass.

Gaver, Smith and O'Shea found that the auditory feedback played an important role in the ways in which the users interacted with the system and with each other. The system was quickly learnt by users who had little trouble remembering the meanings of the various sounds. One problem encountered was that multiple streams of concurrent sounds sometimes led to uncertainty when users could not always discriminate between the various sounds. Also, some users failed to notice that a machine had stopped working as they did not notice the absence of the particular sound.

2.4.1.5. Other applications of auditory icons

Other uses of auditory icons include RAVE [65], ShareMon [45] and OutToLunch [46]. The Ravenscroft Audio Video Environment (RAVE) is described by Gaver et al [65]. The motivation behind RAVE was to support collaborative working amongst colleagues dispersed throughout several rooms in a building. Each room had an audio-video node which comprised a camera, monitor, microphone and speakers. All items in the node could be moved and turned on and off at will by the users. Users interacted with RAVE through GUI buttons. For example, pressing the *background* button would select a view from one of the public areas to be displayed on the audio-video node's monitor screen. The *sweep* button would cause a short sampling of each node's camera allowing users to find out who is present in the various offices. A specific location could be viewed by a three-second *glance* to its camera. Offices could also be connected via the *vphone* and *office share* functions which allowed creation of a larger virtual office.

Auditory icons were introduced to allow greater levels of reasonable privacy. For example, when a glance connection was requested, an auditory warning was sounded at the target node's location three seconds prior to the glance being activated. When connections were broken other sounds, such as that of a door closing, were triggered. These auditory icons gave information about system state and user interactions.

Event	Auditory icon
Log in	Knock-knock-knock
Connection reminder	"ahem"
Low %CPU time	Slow walking
Medium low %CPU time	Medium walking
Medium %CPU	Fast walking
High %CPU	Jogging
Very high %CPU	Running
Log out	Door slam

Table 2.2 ShareMon auditory icons I

This table (taken from Cohen [45]) shows each event being signalled by a unique sound. The auditory icons are representational, or analogic; the 'ahem' adds an air of expectancy.

Cohen's ShareMon system [45] was designed to allow monitoring of background file-sharing tasks without having to interrupt primary, or foreground tasks. The system used auditory icons to signal users logging in, logging out and accessing files. Accessing files was expressed as the percentage of CPU time devoted to that file-sharing activity). The complete set of mappings is given as Table 2.2.

Event	Auditory icon
Log in (guest)	Knock-knock-knock door creak
Log in (registered user)	Keys jingling Key-in-lock Door creak
Connection reminder	Chair creak
%CPU time	No sound
File open	Drawer open
File close	Drawer close
Log out	Door slam

Table 2.3 ShareMon auditory icons II

In this version (taken from Cohen [45]) we see multiple sound events to represent some actions. For instance, logging in is represented by a pair of sounds—knocking on a door followed by the sound of a door opening.

A user study showed that these mappings did not work well enough and so a new set of auditory icons was defined (see Table 2.3) which yielded much better results in user testing.

Cohen took the idea of monitoring background activity further with his OutToLunch system [46]. Cohen states: *“The liquid sound of keystrokes and mouse clicks generated by a number of computer users gives co-workers a sense of ‘group awareness’– a feeling that other people are nearby and an impression of how busy they are. When my group moved from a building with open cubicles to one with closed offices, we could no longer hear this ambient sound.”* OutToLunch was an attempt to restore this sense of group awareness through the use of auditory icons and an electronic sign board. Each time a user hit a key or clicked a mouse, a corresponding key click or mouse click sound would be played by the system to the other users. The exact timing of individual key strokes and mouse clicks was not recorded, but the total activity over a thirty-second period.

Because there was not much information conveyed by the sounds, most users soon found the system annoying. Unlike a real shared workspace, OutToLunch did not give directional clues to allow users to determine who was making the key clicks or mouse clicks. In a revised system each person in the group was represented by their own unique musical motif.

2.4.2. Sonification and audification

In recent years interest has been growing in the use of sonification and audification techniques to convey information about data through the use of sound. The range of application areas on which these techniques have been used is very wide and this area continues to be the most popular avenue of auditory display research.

Mezrich et al [117] showed how a combination of audio-visual techniques could be used to represent large data sets (in this case, multivariate time-series data). The authors asserted that when sonifying data, it is better to quantise the corresponding audio frequencies to the nearest semitone-tone as this sounds more acceptable to the ear than unquantised pitches.

Other examples include the sonification of:

- Data sets [15, 18, 19, 142].

- Infrared spectra to assist the blind in identifying chemical compounds [42, 104, 106, 129].
- Stock market data [94].

In addition, several systems have been developed to allow more general sonification of data. They include the Porsonify system [107, 108], the Kyma sonification language [140, 141] and the VAGE auditory display interface system [24]. Madhyastha and Reed's Porsonify system offered the ability to sonify data sets. For example, the system was used to explore multivariate data related to north-American cities. Variables such as population, climate, and housing cost were mapped to different sounds. The resultant sonifications could be used to compare cities. Unfortunately, Madhyastha and Reed offered no formal or empirical evaluation of the system.

Scaletti's Kyma [141] is a visual sound-specification language which can be used to create sonifications for data sets. The system was applied to models of the human arterial system and city air pollution amongst others. Scaletti and Craig claim the system to be successful [142] but offered no formal evaluation to support the claim.

Hayward [77] employed audification techniques to allow seismic data to be listened to. The data from seismic recorders were collected and then scaled up so that the values lay in the audible frequency range. The data were played through an amplification system and it was possible to discern one seismic event from another without having to look at a seismogram plot. Again, no formal evaluation has been carried out.

2.4.3. Assistive technologies

More recently, researchers have begun to pay attention to the needs of disabled computer users. A research area known as *assistive technologies* (now supported since 1994 by its own ACM-sponsored international conference) has grown up to provide a forum for investigation into the application of new audio, visual, and haptic interaction modes. Naturally, auditory display has been most used to assist blind and visually impaired users.

Lunney [104, 106] mapped information about chemical compound spectra to musical chords and found that blind chemists could accurately identify substances from the sound of their spectra.

Much more common is the application of sonification techniques to the user-interfaces of various computer applications, one of the most common being audio-enhanced world-wide-web browsers and related technology [83, 84, 95, 139, 161, 163]. One of the earliest audio-enhanced interfaces was Edwards' 1989 SoundTrack system [58]. Rigas' AudioGraph system [136] used music to communicate information about diagrams to blind users.

2.4.4. Auralisation and program understanding

When we turn from pure data sonification and look towards sonification and auralisation techniques as a complement to existing visualisation models we find that good progress has been made. Brown and Hershberger [37] coupled visual displays of a program during execution with a form of auralisation. They suggested that sound will be a "*powerful technique for communicating information about algorithms*", though some potential difficulties with using sound are presented:

- Sound is difficult to use effectively because of its complex cognitive-perceptual aspects.
- To find the best mapping of data to sound characteristics is hard. Data can be mapped to frequency, amplitude, duration, timbre, stereo panorama, reverberation, attack and decay rates etc.
- When more than one computer is using audio techniques in the same room, isolation of the sounds is difficult, whereas graphical isolation is not.

Brown and Hershberger offered some examples of successful uses of sound, for instance, applying sound to the bubble-sort algorithm. The main use of sound in this work was to reinforce visual displays, convey patterns and signal error conditions and was by no means the main focus of the work. Like most other visualisation systems that employ audio, Brown and Hershberger's work used sound as a counterpart or appendage. However, it should be noticed that they undertook no formal evaluation of the approach.

Mayer-Kress et al [112] applied sonification techniques to chaotic attractor functions which led to musical forms in which the similar but never-the-same regions could be heard in a musical framework that is not unpleasant.

Within the field of auditory-display research, program auralisation as a subject in its own right continues to attract relatively little interest. The tendency remains to use audio as an additional medium of communication in an attempt to reduce clutter on VDUs or to enhance the experience of user interaction. Very little work has been done to investigate whether, in some situations, sound can be the primary channel for communication and visualisation.

Early efforts directed at more pure auralisation have been concerned with specific algorithms, often in the parallel programming domain. Examples include Francioni et al [62, 63] who were interested in using auralisations to help debug distributed-memory parallel programs, and Jackson and Francioni [81, 82] who suggested features of parallel programs that would map well to sound. Those systems that are applied to more general sequential programming problems require a degree of expert knowledge to use, whether in terms of programming skill, musical knowledge, expertise in the use of sound generating hardware, or all three.

The case for using sound to aid debugging has been supported by Jackson and Francioni [81], although they felt that a visual presentation was also needed to provide a context or framework for the audio sound track. It should be noted that these auralisations were of an historic nature, that is, the program state is captured as a set of trace data that are then auralised and animated. This makes audio-enhanced step and trace debugging impossible. They make the argument in defence of auralisation that some types of programming error (such as those that can be spotted through pattern recognition) are more intuitively obvious to our ears than our eyes. Also, they point out that sound, unlike images, can be processed by the brain passively; we can be aware of sounds without needing to listen to them. An implication of this is that a sound related to an exceptional event can be detected without having to pay strict attention to the sounds related to normal behaviour. This is well illustrated by an example from history. On World War II battleships the sound of a bugle was used to transmit messages across the sea as an alternative to the semaphore lamp. The bugle conveys messages which can be understood without needing

to take one's eyes off the job in hand as would be necessary with a semaphore or visual Morse signal

2.4.5. Existing auralisation systems

To-date we have identified the following systems claiming to be program auralisation tools (as opposed to visualisation systems that incorporate some auralisations): InfoSound [148] by Sonnenwald et al, the LogoMedia system by DiGiano and Baecker [53], Jameson's Sonnet system [85, 86], Bock's Auditory Domain Specification Language (ADSL) [22, 23], and the LISTEN Specification Language, or LSL [20, 21, 111]. All these systems convert their auralisations into MIDI data which are sent, via a MIDI port, to a suitable MIDI-equipped sound generator (such as a music synthesiser, or even a sampler).

2.4.5.1. Infosound

InfoSound [148] is an audio-interface tool kit that allows application developers to design and develop audio interfaces. It provides the facility to design musical sequences and everyday sounds, to store designed sounds and to associate sounds with application events. InfoSound combines musical sequences with sound effects. One limitation is that the software developer is expected to compose the musical sequences himself. To the musically untrained (the majority) this militates against its general use.

The system is used by application programs to indicate when an event has occurred during execution and was successfully used to locate errors in a program that was previously deemed to be correct. The authors of InfoSound believe that sound can be a useful feedback mechanism to assist in the debugging process. Users of the system were able to detect rapid, multiple event sequences that are hard to detect visually using text and graphics.

Another drawback with InfoSound is the range of hardware needed to support it. A MIDI-equipped music keyboard connected to an Apple Macintosh computer is needed to compose the music sequences, while the sound storage sub-system needs an IBM AT-compatible computer.

2.4.5.2. LogoMedia

LogoMedia [53], an extension to LOGOmotion (described earlier) allows audio to be associated with program events. The programmer annotates the code with probes to track control- and data-flow. As execution of the program causes variables and machine state to change over time the changes can be mapped to sounds to allow execution to be listened to. Like InfoSound, LogoMedia employs both music and sound effects. The authors assert that comprehending *“the course of execution of a program and how its data changes is essential to understanding why a program does or does not work. Auralisation expands the possible approaches to elucidating program behaviour.”*

The main limitation is that the auralisations have to be defined by the programmer for each expression that is required to be monitored during execution. In other words, after entering an expression, the programmer is prompted as to the desired mapping for that expression. This is unacceptable for our purposes as the novice programmer really only wants to concentrate on the program text. It also has the drawback of creating a more alien programming environment. We believe it is better to retain a familiar interface for program entry and compilation, with the auralisation being carried out automatically at the pre-processor phase.

2.4.5.3. Sonnet

The Sonnet [85, 86] system is specifically aimed at the debugging process. Using a visual programming language (SVPL) the code to be debugged is tagged with auralisation agents that define how specific sections of code will sound. Figure 2.5 shows a component that allows a note to be turned on and off connected to some source code. (The “P” button on the component allows static properties such as pitch and amplitude to be altered.) The component will start a note sounding just before the `while` loop and will turn the note off after the loop has terminated.

Other components allowed the user to specify how many iterations of a loop to play. Program data could also be monitored by components that could be attached to identifiers within the code.

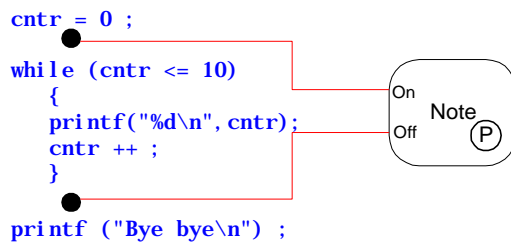


Figure 2.5 A Sonnet VPL component

In this example, taken from Jameson [86], the VPL component is the box on the right. The component has two connections, one to turn on a note (via a MIDI note-on instruction) and one to silence the note (MIDI note-off). In this example the note-on connector is attached to the program just before the loop, and the note-off connector just after the close of the loop. Therefore, this auralisation will cause the note to sound continuously for the duration of the loop. Thus, placement of the connectors defines the auralisation.

The guiding principle is that the programmer, knowing what sounds have been associated with what parts of the program, can listen to the execution looking out for patterns that deviate from the expected path. When a deviation is heard, and assuming the expectation is not in error, the point at which the deviation occurred will indicate where in the program code to look for a bug. A potential drawback with this approach is that unless the programmer explicitly tags a section of code with an auralisation object then that code will generate no sound.

Sonnet is an audio-enhanced debugger. This means that because it interfaces directly with the executing program it is not invasive and does not need to carry out any pre-processing to add the auralisations.

The visual programming language offers great flexibility to the programmer who wants to auralise his program. However, it does require a lot of work if an entire program is to be auralised even very simply.

2.4.5.4. ADSL

Bock's Auditory Domain Specification Language (ADSL) [22, 23] differs from the above three approaches in that it does not require sounds to be associated with specific lines of program code. Instead users define *tracks* using the ADSL meta-language to associate audio cues with program constructs and data. These tracks (see Figure 2.6) are then interpreted by a pre-processor so that the programmer's code has the auralisations added to it at compilation allowing the program to be listened to as it runs. An advantage of this approach is that it is possible to define a general purpose auralisation. That is, by specifying types of program construct to be auralised there is no requirement to tag individual lines of code with auralisation

specifications. When such tagging is required this can be done using the features of the specification language. Like Sonnet, ADSL is non-invasive as the auralisations are added to the source program during a pre-processing phase.

```
Track_name=Loop
{
1 Track=Status('for'):Snd("for_sound");
2 Track=Status('while'):Snd("while_sound");
}
```

Figure 2.6 An ADSL track to monitor for and while loops

This fragment of ADSL (taken from Bock [22]) specifies that *for* and *while* loops are to be signalled by playing the 'for_sound' and 'while_sound' respectively. These two sounds will have been previously defined and could be a MIDI note sequence, an auditory icon or recorded speech. The sounds will be heard when the keywords *for* and *while* are encountered in the program.

ADSL used a mixture of digitised recordings, synthesised speech and MIDI messages. The choice of sounds and mappings is at the discretion of the user, although a common reusable set of auralisations could be created and shared amongst several programmers. Tracks could be refined to allow probing of specific data items or selective auralisation of loop iterations. The system is flexible, allowing reasonably straightforward whole-program auralisation, or more refined probing (such as in Sonnet).

In an experiment [23] thirty post-graduate engineering students from Syracuse University all with some programming experience were required to locate a variety of bugs in three programs using only a pseudo-code representation of the program and the ADSL auditory output. On average, students identified 68% of the bugs in the three programs. It should be pointed out that no control group was used, so it is impossible to determine whether the auralisations assisted in the bug identification.

The question as to whether ADSL could be usefully employed by novices arises. Given that the novice is likely to find writing a simple program in the chosen programming language difficult, it is unreasonable to expect him to learn to use ADSL which is effectively another programming language.

2.4.5.5. Listen

Mathur's *Listen* project [20, 21, 111] follows a similar approach to that used by ADSL. A given program is auralised by writing an auralisation specification in the Listen Specification Language (LSL) meta-language and then a pre-processing phase is used to parse and amend the source program prior to compilation. Again, the original source program is left unchanged.

Auralisations are created by writing *auralisation specifications* or *ASPECs*. An ASPEC specifies the mapping between program-domain events and auditory events. An example ASPEC for an automobile controller is shown in Figure 2.7. This example auralises all calls to the program functions `gear_change`, `oil_check` and `weak_battery`.

```

begin aural spec
specmodule call_auralize
var
  gear_change_pattern, oil_check_pattern, battery_weak_pattern: pattern;
begin call_auralize
  gear_change_pattern: ="F2G2F2G2F2G2C1: qq" + "C1: f";
  oil_check_pattern: ="F6G6: h";
  battery_weak_pattern: ="A2C2A2C2";
  notify all rule = function_call "gear_change" using gear_change_pattern;
  notify all rule = function_call "oil_check" using oil_check_pattern;
  notify all rule = function_call "battery_weak" using battery_weak_pattern;
end call_auralize;
end aural spec.

```

Figure 2.7 An LSL ASPEC for an automobile controller

This LSL auralisation specification (taken from Mathur [111]) defines auralisations for various program events. The ASPEC contains definitions for the auralisations themselves and their usage. For instance, we see that a call to program function `gear_change` causes the auralisation `gear_change_pattern` to be played. This auralisation is defined earlier in the ASPEC as a sequence of MIDI note-on/off instructions.

LSL is said to be a true meta-language which means, in theory, it can be used to define auralisations for programs written in any language. The practice requires an extended version of LSL for each target language.

What is immediately apparent is that writing auralisation specifications using LSL is perhaps almost as difficult as writing the source program. To be able to do this, the programmer must learn the complex syntax of LSL. Additionally, some musical knowledge is needed to know how to specify which pitches are used in the auralisations.

The Listen project has been inactive since 1996 and no formal experimentation or evaluation of the system has been published.

Just as Baecker et al [11] provided a tabular comparison of visualisation systems (Table 2.1), so we present a comparison of program auralisation systems as Table 2.4.

	InfoSound	LogoMedia	Sonnet	ADSL	LISTEN
Domain of use	IC* languages	Logo	Any language	C	Any language
Type of auralisations employed	Music sequences and sound effects	Music pitches and sound effects	Music pitches and modulations	Music pitches and sound effects	Music pitches
Suitable for the novice programmer	No	No	No	No	No
Specialist knowledge required to define auralisations	Music composition abilities	Some music and familiarity with MIDI	Some music and familiarity with MIDI	Some music and familiarity with MIDI	Some music and familiarity with MIDI
Extensible: auralisations can be programmed	Yes	Yes	Yes	Yes	Yes
Monomorphic: host language used in definition of auralisations	No	Yes	No	No	No
Unobtrusive: source code left unchanged by auralisations	?	Yes	Yes	Yes	Yes
Employs pre-processing phase	No	N/A	No	Yes	Yes

Table 2.4 Comparison of program auralisation systems

What all auralisation techniques employ is the ability of sound to assist in looking for further clues. For instance, when we cross a road and hear a car approaching, we interpret the sound to give us a directional fix for the vehicle and then use our eyes to look at the oncoming danger. In the same way auralisation aims to direct us to the region of program code that contains the error. We can then use traditional methodology to identify and correct the bug. Francioni and Rover [64] found that sound allows a user to detect patterns of program behaviour and also to detect anomalies with respect to expected patterns.

2.5. Sound mappings in auditory display

It can be seen that mappings between computer data/events and sound are achieved in a variety of ways. Gaver [66] classified the mappings into three types: symbolic, metaphoric and nomic. Symbolic mappings are arbitrary and rely on some accepted conventions for their meaning. For example, the cross is a symbol for Christianity. Nomic mappings are ones which are derived from the source. A photograph is a nomic mapping of the scene it portrays [66]. Metaphoric mappings make use of some similarity between the object being represented and the representation.

They are not wholly arbitrary, but they do not depend on physical causation in the way that nomic mappings do [66]. For example, the use of tree diagrams in representing genealogies is metaphoric: genealogies are not trees, but they have a similar structure to a tree diagram.

While the difference between nomic and symbolic mappings is clear, symbolic-metaphoric and metaphoric-nomic boundaries are not so easily definable. Kramer [92] thus redefines Gaver's taxonomy as a continuum of analogic (nomic) representations and symbolic mappings.

Using Kramer's definition, work such as Hayward's seismographic audification [77] can be classified as analogic mappings whilst Blattner's earcons [17] are nearer to the symbolic end of the continuum.

2.6. Using music in auralisations

The five auralisation systems surveyed above all make reference to musical pitches and MIDI data in their approaches to auralisation. On closer examination it is seen that 'music' is used loosely as a term to describe output based upon musical pitches. None of the systems uses a formal musical framework or grammar. Even InfoSound only generates musical sequences if the programmer is skilled enough to compose the sequences himself. The systems are simply mapping program data to common frequencies to effect the auralisations without regard to the musicality of the output.

A stark omission from the existing literature is empirical evidence to show that program auralisation is useful. Some preliminary experimentation by Alty indicated that algorithm state could be visualised using musical auralisation techniques [3]. In this experiment reaction to tasks that required the interpretation of musical output was gauged. The first task showed that subjects were fairly accurate (62%) at estimating the difference, measured in semitones, between two musical pitches. The second task required subjects to sketch, on paper, the perceived shape of short musical sequences. The subjects were generally able to pick up the basic shape being presented. Given that the experiment used musical sequences generated by a bubble sort algorithm the results suggest that music can successfully communicate algorithm state.

One of the problems a novice programmer in particular faces is that of understanding his work. Before he can remove an error from a program he must first, unless a fortuitous accident occurs, correctly interpret the behaviour of his program and the messages from the code compiler.

Programming was described by Weinberg as a “*communication between two alien species*” [156]. Conner and Malmin [47] say that we must recognise that a gap in understanding may exist between the communicator and the receiver. For successful communication there must be a common medium between the two in order that the gap may be bridged. Meyer [116] observes that meaning and communication “*cannot be separated from the cultural context in which they arise. Apart from the social situation there can be neither meaning nor communication.*”

In discussing visualisation systems for parallel debuggers, Pancake and Utter [127] note that though a human can apply knowledge of an algorithm when reasoning about program behaviour, a software tool must rely solely on details available from the program text and its run-time behaviour. Debuggers provide information that is constrained to a program’s implementation rather than its underlying problem. Therefore, it is critical that any semantic gap between the concept and the code be minimised [127].

In our case, it is vital that the auralisations used are not so far from the programmer’s frame of reference as to be rendered useless. If music is to be used, it must not rely on forms and intervals that are too unfamiliar or indistinguishable to the average person.

2.7. Summary

Visualisation techniques can be used to improve understanding of software. Over the last decade or so some visualisation researchers have found that aural representations can complement, enhance or even be superior to visual representations alone. This has spawned a new research area known as auditory display in which those involved are examining the different ways in which the auditory channel can be utilised in the process of human-computer interaction.

Different techniques for using sound such as sonification, audification and auralisation have been developed to exploit sound in various HCI applications, be they

audio enhanced software interfaces, sound-controlled data exploration systems, or debuggers that use sound to represent program execution. Early on, such systems tended to be hybrids employing aural and graphical visualisation methods. Today, there is a growing body of research dedicated to the audio channel alone.

Within this latter body, almost no effort has gone into exploring music as a medium for communication. Many systems use sound effects; those that use music tend to do so only in the sense of representing data as musical pitches without any reference to musical forms, structures or syntax. Music is based upon a defined structure, or set of rules. Structuring auralisations according to simple syntactical rules offers the hope of music forming the basis for a bridge of the semantic gap between an incorrectly functioning program and the novice programmer. More recently, efforts have been made to use more formal musical frameworks in auditory display. Leplatre and Brewster [98] have begun investigations into using music to help in navigating around complex hierarchies of information. Hankinson and Edwards [73] have started to lay down a formal theoretical foundation for the use of musical grammars in audio communication applications.