TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Auditory extension of a GUI based user interface for visually disabled persons

Citation for published version (APA):
van der Knijff, R. M. (1995). *Auditory extension of a GUI based user interface for visually disabled persons*. (IPO rapport ; Vol. 1060). Instituut voor Perceptie Onderzoek (IPO).

Document status and date:
Published: 11/07/1995

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 18. May. 2025

Institute for Perception Research
PO Box 513, 5600 MB Eindhoven

11.07.1995

Rapport no. 1060

Auditory extension of a GUI based user interface for visually disabled persons

R.M. van der Knijff

FACULTEIT DER ELEKTROTECHNIEK
TECHNISCHE UNIVERSITEIT
EINDHOVEN
VAKGROEP MEDISCHE ELEKTROTECHNIEK

AUDITORY EXTENSION OF A GUI BASED USER INTERFACE FOR VISUALLY DISABLED PERSONS

by R.M. van der Knijff

Rapport van het stagewerk
uitgevoerd van december 1994 tot juni 1995
onder leiding van ir. L.H.D. Poll

DE FACULTEIT DER ELEKTROTECHNIEK VAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN AANVAARDT GEEN VERANTWOORDELIJKHEID VOOR DE INHOUD VAN STAGE- EN AFSTUDEERVERSLAGEN

Abstract

To make computer systems, based on Graphical User Interfaces (GUI's), accessible to visually disabled people an alternative I/O device is under construction, the *SoundTablet*. For this, the Current Screen Model (CSM) represents the screen content of a GUI based computer system. The following method is developed for the transformation of GUI objects into audible ones: A standard set of audio related attributes, derived from GUI object characteristics, is added to each CSM object. Sound information of these attributes consists of non-speech, prerecorded speech and synthetic speech, which can be made audible, transparent, on 3 different resources. Keeping each audio attribute up to date and making the right attributes audible at the right moment does accomplish the desired transformation. An extra pop-up menu is introduced for the handling of GUI aspects which are suspected to cause a great deal of trouble for visual disabled users.

Contents

1 **Introduction** 3

2 **Software engineering** 4
- 2.1 Object data dictionary 5
- 2.2 Object information model 5
- 2.3 Object behaviour model 6
- 2.4 Object process model 6

3 **Environment analysis** 7
- 3.1 Current screen module 8
- 3.2 Digitizer 8
- 3.3 Gravis Ultra Sound (GUS) 8
- 3.4 K2000 synthesizer 8
- 3.5 Text to speech system 8
  - 3.5.1 Environment analysis 9
  - 3.5.2 Object data dictionary 9
  - 3.5.3 Object information model 9
  - 3.5.4 Object behaviour model 9
  - 3.5.5 Object process model 10
- 3.6 Real-Time Kernel (RTK) 11
- 3.7 Development platform 11

4 **Requirements** 12
- 4.1 Physical properties 12
- 4.2 Identity properties 12
- 4.3 Transforming from visual into auditory 13

5 **Concept** 14

6 **Audio Object** 16
- 6.1 Object data dictionary 17
- 6.2 Object information model 17
- 6.3 Object behaviour model 18
- 6.4 Object process model 19
  - 6.4.1 Constructors 19
  - 6.4.2 Initialization 21
  - 6.4.3 Sound control 21
  - 6.4.4 AudioList functions 23

6.5 User manual 24
6.5.1 Declaration 24
6.5.2 Object instance update 25
6.5.3 Sound control 25

7 CSM extension **27**
7.1 Object data dictionary 27
7.2 Object information model 27
7.3 Object behaviour model 36
7.3.1 Point at 36
7.3.2 Events 37
7.3.3 Guidance 37
7.3.4 Right mouse button 38
7.3.5 Pop-up menu 38
7.4 Object process model 38

8 Conclusions **40**

A Object process model **42**
A.1 Windows 42
A.2 Client area 42
A.3 Button 43
A.4 Check box 43
A.5 Close button 44
A.6 Cursor 44
A.7 Dialog-box 44
A.8 Dial 44
A.9 Dia-client 45
A.10 Edit-contr 45
A.11 Edit-line 45
A.12 Graph 46
A.13 Group-box 46
A.14 Icon 46
A.15 List box 47
A.16 Menu item 47
A.17 Menu list 47
A.18 Pointer 48
A.19 Pull-Down edit 48
A.20 Radio button 48
A.21 Slider 49
A.22 Scroll bar 49
A.23 Text line 50

Chapter 1

Introduction

Visually disabled people (VDP's) are due to their handicap generally designated to jobs which allow them to work at a desk. A lot of these jobs require the use of a computer which should be accessible through a non-visual user interface. Existing non-visual user interfaces, like refreshable braille lines and synthetic speech systems, are developed for character based user interfaces. It is expected that all character based user interfaces will be replaced by graphical user interfaces (GUI's) in the near future.

The need for non-visual user interfaces based on GUI's has created a research field not only for social reasons but also from a financial point of view: The perspective of intensive educated VDP's becoming unemployed would cause commotion and costs money. Therefore the European Community funded projects via the TIDE [1] program.

From February 1992 until the end of 1993 the Institute for Perception Research (IPO) participated in a TIDE project in which a user interface has been implemented based on an existing access device which allows VDP's to work with the kind of applications they were already using [1, Waterham,Beumer,Klimbie,Poll]. Because this interface does not bring the advantages of a GUI in reach of VDP's, an alternative I/O device is selected that not only allows the user to access GUI information, but also to perform GUI specific operations like selecting, activating and dragging. With the current screen model (CSM), developed for the TIDE project representing the actual GUI screen contents, and the alternative I/O device, a new nonvisual user interface will be developed.

This report describes the extension of the existing CSM with visual to audible translation facilities, according to the object oriented methodology as proposed in [2, Poll] and summarized in chapter 2. In chapter 3 the experimental setup is described together with all realated resources. On the basis of the requirements presented in chapter 4 a concept is introduced in chapter 5. This concept is elaborated in chapters 6 and 7. Finally chapter 8 gives an evaluation together with some conclusions.

[1] Technology Initiative for Disabled and Elderly

Chapter 2

Software engineering

Software development for this project is being done with an 'Object Oriented Methodology' (OOM) described in [2, Poll]. The main principles of this method will be given in this chapter.

Software engineering methods comprise the development phase from problem statement to the actual implementation and use of software. Several phases in the software development process can be identified, each resulting in separate software documents. The first document describes the environment with which the software has to perform (hardware, software, user behaviour etc.). In the second document the requirements are given which will result in a conceptual model as third document, in which a readable specification is given. This concept is worked out in a formal specification document (discussed in the following paragraphs) from which an implementation can be built. Implementation, testing, maintenance and user-manual documents may be necessary, depending on the application.

The basic software quality demands can be summarized as follows:

1. *Correctness*: The ability of software products to exactly perform their tasks, as defined by the requirements and specification.
2. *Robustness*: The ability of software systems to function even in abnormal conditions.
3. *Extendibility*: The ease with which software products may be adapted to changes of specifications.
4. *Compatibility*: The ease with which software products may be combined with other products.
5. *Re-usability* The ability of software products to be reused, in whole or in part, for new applications.

Object oriented programming (OOP) is a type of programming that provides a way of creating modular software by establishing partitioned modules, for both data and methods on data, that can be used as templates for spawning copies of such modules on demand.

The basic principles of OOP are the following:

Identity Every object has an identity. A concrete example is a file, a conceptual example is a scheduler as member of an operating system.

Classification Objects belong to a certain class that is an abstraction of the object itself.

Polymorphism The ability to have several entities of one object definition.

Inheritance Classes are able to inherit aspects of other classes.

The core of all software documents is formed by the formal specification, semantic strongly dependant on the software development methodology. Following sections will summarize the OOM formal specification principles.

2.1 Object data dictionary

In this document a description is given of the objects that can be identified without going into details. The classic approach of decomposition of a problem into smaller problems has to be applied in order to write this document.

2.2 Object information model

In this document a description is given of object attributes consisting of several data structures and links to other objects. The domains of the attributes are not given yet but the attributes are only listed together with eventual links to other objects. Figure 2.1 indicates how the object to object relations (parent child relations) are presented graphically. The squares represent the objects from which the one on top

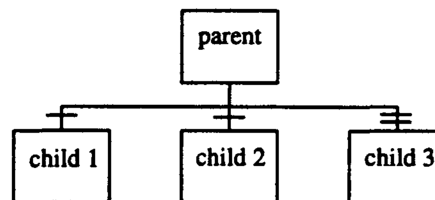

Figure 2.1: Graphical presentation of parent child relations

of the tree is called 'parent' and the ones below 'children'. The first text line in each square indicates the object type as defined in the previous document. The short horizontal lines above an object indicate whether the parent owns one or more objects of this type: One line indicates one object, two lines indicate more than one object.

Object correlations are depicted in diagrams such as 2.2 and reflect mutual object relations. This might be the result of direct communication but could also be the result of one child-object that causes another child-object of the same parent to change via the parent object.

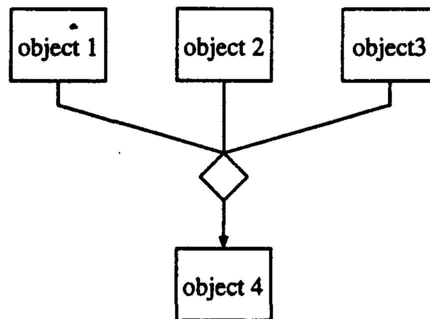

Figure 2.2: Graphical presentation of object correlations

2.3 Object behaviour model

This document resembles the state transitions diagrams that are often used by other software engineering methods. All states and the related transitions determine the object behaviour.

2.4 Object process model

In this document an undetailed description is given of the set of functions responsible for all the object aspects specified in the preceding documents, including local variables. These functions may belong to specific objects but can also stand by themselves. A pseudo programming language is used in the TIDE project to describe the functional flow, but also flow charts can be used.

Chapter 3

Environment analysis

The interface system under construction, depicted in figure 3.1, is build up around a 486-based PC on which all the interface software is running. GUI information

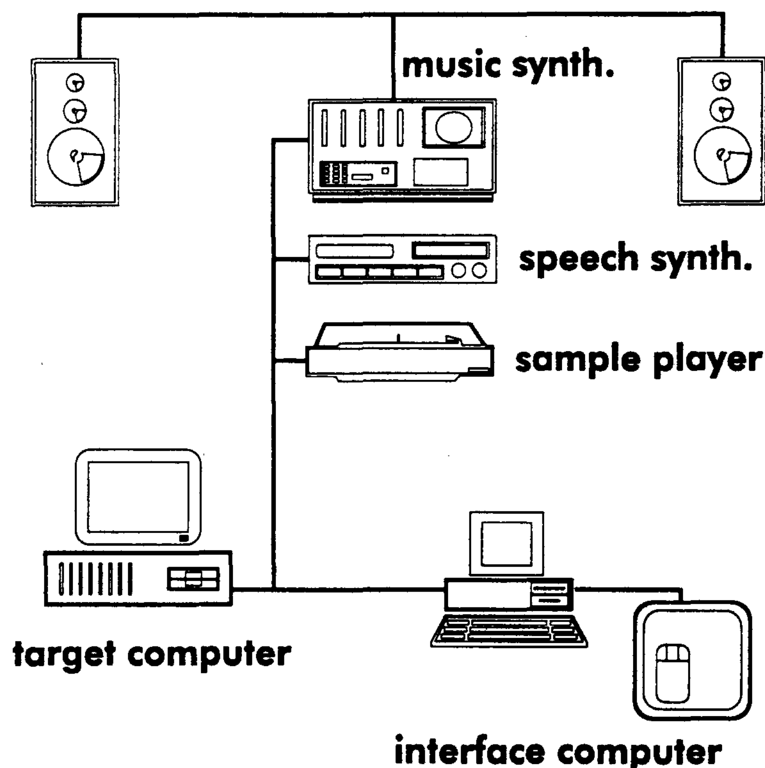

Figure 3.1: Non visual interface system setup

is extracted from the target computer (the 'office application computer') with a video acquisition board. OCR (Optical Character Recognition) software is used to construct and update a model (CSM) representing the current screen contents. For experiments the CSM can be fed by a software simulator which makes target computer, video acquisition hardware and OCR software superfluous.

From the user point of view an alternative I/O device, called *SoundTablet*, forms the heart of the interface. This device consists of a square with small edges on which

a mouse-like device, with absolute coordinates, can me moved around, and also of a normal computer keyboard. While moving, a soundscape corresponding to the GUI information content is made audible. For this purpose a sample player, a music synthesizer and a speech synthesizer are used. Besides this 'display' function the SoundTablet substitutes all GUI mouse actions and translates them, together with keyboard information, into proper target computer signals.

The following sections of this chapter give information about the different subsystems, especially about the software modules which are used by the audio extension software.

3.1 Current screen module

The CSM keeps track of the current screen contents and restores the relations between separately recognised objects. The CSM software uses an GUI object library, a screen construct/update module and an information parsing module as described in [9, Gerrits, Poll, Waterham].

3.2 Digitizer

A digitizer, normally used for accessing CAD applications, forms the heart of the soundtablet. The housing of the original cordless puck has been replaced by the housing of a three-button mouse. This mouse can be moved within a rectangle on the digitizer limited with 4 standing edges. The digitizer software module is described in [7, Poll].

3.3 Gravis Ultra Sound (GUS)

The GUS is a PC board with AD/DA capabilities and a MIDI interface. In this project it is used as a prerecorded speech player and as a MIDI transmitter. The software modules are described in [5, Poll] and [6, Poll].

3.4 K2000 synthesizer

The Kurzweil K2000 music synthesizer is used to generate all non-speech audio under control of the GUS MIDI interface. Because only general MIDI commands are used no dedicated software is needed for the K2000.

3.5 Text to speech system

Generation of unpredictable speech is done with a stand-alone text-to-speech system also known under the name *Typestem* ([4, Deliege]). The system is able to convert ASCII text, transmitted from a RS232 port, into synthetic speech. The software module developed for RS232 control of the speech system will be discussed in the following subsections.

Table 3.1: Jumper settings text-to-speech system

No.	function	present
1	text-to-speech mode	Y
2	application mode	N
3	terminal 19200 baud	N
4	host 19200 baud	N
5	reset at time out	N
6	normal operation	N

3.5.1 Environment analysis

The text-to-speech system has to be configured into text-to-speech/application mode by placing the jumpers according to table 3.1. The speech system and the interface computer must be connected via a cross-coupled RS232 cable with connections given in figure 3.2. The drawn shorts on the PC side are needed for proper functioning of the PC-BIOS and/or real-time kernel (introduced in 3.6) routines.

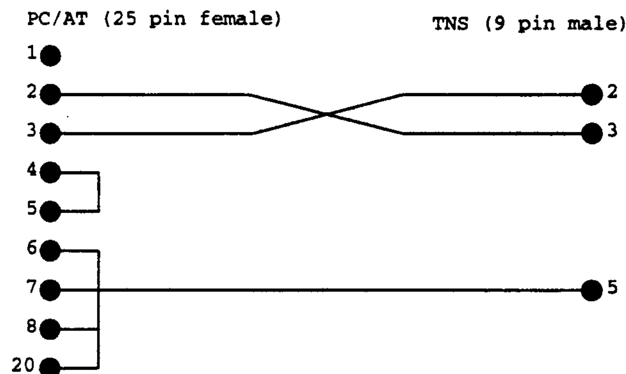

Figure 3.2: Serial cable connections

3.5.2 Object data dictionary

There is only one object: The *SpeechSynth* object which is used to pass text from the host computer to the speech synthesizer for playing.

3.5.3 Object information model

The SpeechSynth object contains only one attribute: *Status*, with which the actual status of the synthetic speech system can be determined.

3.5.4 Object behaviour model

Figure 3.3 depicts the state transition diagram of the object SpeechSynth.

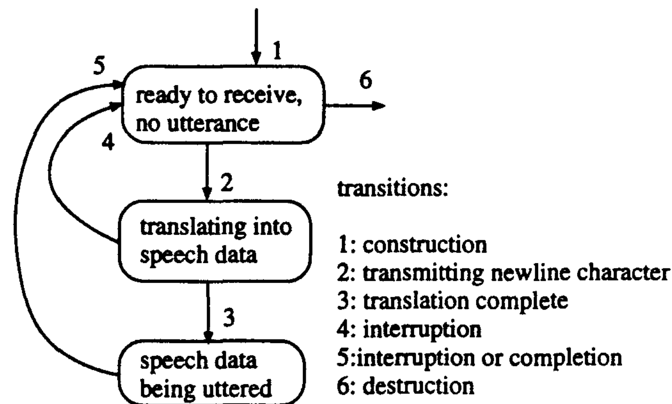

Figure 3.3: State transition diagram of the SynthSpeech object

3.5.5 Object process model

Status	→	{SS_IDLE,SS_BUFTRAN,SS_BUFPLAY}

Contructor:

```
SpeechSynth()
{
   initialise COM port;
   response=getCOMstatus();
   if (response==COM_DATA_READY)
   {
      initialise synthetic speech board;
      Status=SS_IDLE;
   }
   else
      exit(error);
}
```

Speak: Loads a text string into the speech board and starts the translation.

text	→	pointer to a text string

```
Speak(char *text)
{
   transmit text string;
   transmit end of line character;
   Status=SS_BUFTRAN;
   wait for prompt
   Status=SS_BUFPLAY;
}
```

Break: Resets the synthetic speech board.

```
Break()
{
   transmit reset character;
   Status=SS_IDLE;
}
```

SpeechSynthStopped: Returns TRUE if no speech is being uttered.

```
Boolean SpeechSynthStopped()
{
   if (Status==SS_IDLE)
```

```
      return TRUE;
   if (Status==SS_BUFPLAY)
   {
      transmit 1 space;
      transmit end of line character;
      wait for prompt;
      Status=SS_IDLE;
      return TRUE;
   }
   return FALSE;
}
```

3.6 Real-Time Kernel (RTK)

The capabilities of MS-DOS and the BIOS kernel are not sufficient for this project where priority tasks and real-time performance are needed. Therefore a real-time kernel is used ([8, RTK]). The object code of this kernel is only available in the *huge* memory model. This implies the use of the huge memory model for all software modules. Another implication of multi-tasking is the possible introduction of critical sections. This phenomenon will be discussed later on in this report.

3.7 Development platform

All software will be developed with *Borland C++* under MS-DOS extended with the discussed real time kernel.

Chapter 4

Requirements

Aim of this (sub)project is the extension of the existing screen model with features which can 'audiolize' the modelled GUI objects and the dynamic model behaviour, without, from a user point of view, noticeable delay, and with the same 'hear and feel' as the original GUI's 'look and feel' . A naive approach would be the realisation of a one-to-one mapping from the screen content onto a soundscape. This is not possible because the parallel processing capabilities of human's visual system are much more powerful than the ones of human's hearing system. This fact implies a parallel to serial conversion and the need for data reduction. It is also expected that some GUI specific aspects have to be replaced by more VDP friendly ones. For example locating the close button of a window could be replaced by a menu item as part of an extra pop-up menu.

The information content of a GUI can be divided into two groups: Physical properties and identity properties.

4.1 Physical properties

Physical properties of GUI objects embody spatial location, size, and mutual object relations. For example the mutual relation between the GUI objects *Pointer* and *Window*: A window can only be activated when the pointer is located inside the window area.These properties are continuously perceptible via the computer display of GUI based systems.

4.2 Identity properties

In current GUI's at least 25 different objects can be identified, most of them with several states. For example the GUI object *Button* which can be identified as an rectangular graph with a label on it. If there are several buttons on the screen one of them might be default, which can be seen by the thick border of the button. Identity properties are perceptible via the computer display from the moment they occur.

4.3 Transforming from visual into auditory

To fulfil the mentioned parallel to serial conversion and data reduction, several GUI objects have to be packed together into larger objects and additional objects have to be introduced increasing the efficiency of interaction without decreasing accuracy. From an object oriented point of view this implies the exchange of object attributes between parent objects and child objects. A constraint to this process is the fact that the *hear and feel* of the new user interface has to resemble the *look and feel* of the original GUI. This makes it easier for the VDP user to consult on line manuals and allows him to discuss GUI specific aspects with other sighted GUI users.

Another constraint is the efficiency of the transformation. From a user point of view working with a GUI based system has the advantage of controlling all applications with the same interface. This advantage has to be an invariant condition for the transformation. Delay between the actual screen content and the transformation result should not be noticeable by the VDP user because that will cause fuzzy object behaviour (e.g. hearing a window that has already disappeared).

A lot of details concerning the optimal transformation are still research topics and/or in an experimental stage. For this reason the implementation of the transformation should be flexible, extendable, and understandable.

Chapter 5

Concept

The CSM uses 25 objects with which the screen content can be modelled. An exact definition of the visual representation of these objects with their attributes is given in [10, ipo].To model the auditory/tactile presentation of the GUI objects an extra group of attributes is added to the GUI library. Within this group the following types of sound can be distinguished on the basis of their functionality:

Space occupation A non-speech impact sound is used to indicate that an object is entered by the pointer. While the pointer is present in the object area, a steady-state non-speech sound is used. When the pointer is positioned within an object without hitting the borders (by lifting the mouse), or when a new object appears underneath the pointer, no impact sound is played and the steady state sound is started immediately.

Guidance Non-speech sound is used to indicate where the nearest object, relative to the current pointer position, is located.

Appearing Non-speech is used to indicate the appearing of new GUI objects (e.g. appearing of a dialog box).

Disappearing Non-speech audio is used to indicate the disappearing of existing GUI objects (e.g. disappearing of a list box belonging to a pull down edit).

These sounds indicate the physical aspects of GUI objects. Other sounds are used for identification purposes:

Identity Prerecorded speech is used to present the identity of the 25 GUI objects. This information is also contained in the steady-state non-speech sounds from the *Space occupation* type.

Information content Synthetic speech is used to present the information content of an object (e.g. window title).

State Prerecorded speech is used to indicate the state of an object (e.g. default,not active). This information is also contained in the steady state non-speech sounds from the *Space occupation* type.

State changing Both prerecorded speech and non-speech audio is used for state changes (e.g. a check box becoming selected and default).

In addition to the current screen content a pop-up menu can be added to the CSM. With this menu some GUI aspects are handled which are suspected to give problems for visual disabled users. The pop-up menu is activated when the middle mouse button is held down in some already active GUI objects. After activation the pop-up menu behaves as a GUI vertical menu with the difference that it remains attached to the actual pointer position. Whether or not a pop-up menu is available, and which items are present in the menu, is described in the OIM (object information model) tables (7.2).

The main idea is the following: Create and update audio attributes for each GUI object present in the CSM. Activating the right attributes at the right moment, initiated by CSM information and user actions, transforms the visual GUI in an audible interface. This implies 3 sub tasks:

- Creation of an object to handle audio information.
- Inserting and updating audio information into each CSM object.
- Activating the right audio information at the right moment.

The first sub task is discussed in the next chapter, the other two in the chapter following the next.

Chapter 6

Audio Object

The *Audio* object contains information which can be made audible with the incorporated methods on 3 different resources: A music synthesizer, a sample player and a speech synthesizer. The information content of one audio object instance may consist of audio information for multiple resources.

Before any audio object can be used an initialization function has to be executed. This function creates a new task parallel to the current one with a higher priority. This task ensures that at any time only one speech voice is being uttered. When an Audio object wants to make his information content audible, all speech information is placed in a FIFO queue. When the parallel task detects a non empty queue, the front queue element is made audible only when no other speech is audible at that moment. Non-speech information might be audible concurrent with other non-speech and/or speech information.

The usage of parallel executing tasks operating on the same data (FIFO queue) introduces so called 'critical sections': Sequences of elementary actions on shared data which are only provable correct if only one task is executing such a section. To illustrate the phenomena consider the following situation: The information content of an Audio object is placed in the FIFO queue (task 1) containing only one element which is made audible at the same time (task 2). The result of these actions depend on the exact execution order of machine code instructions which is managed by the real time kernel routines and not known by the tasks. When task 1 reads the queue tail pointer before task 2 has updated this pointer and task 1 updates the reference pointers based on information before the updating of task 2, a faulty queue will be created.

The real time kernel delivers a mechanism to prevent parallel execution of critical sections: *Protection semaphores*. Critical sections are surrounded by *RTK-Wait(psemaphore)* and *RTKSignal(psemaphore)* instructions. Task execution can only continue after an *RTKWait(psemaphore)* instruction when no other task is executing a critical section with the same protection semaphore.

Another type of semaphore is used to avoid so called 'busy form of waiting': A program construction which consumes a lot of processor cycles waiting for the occurrence of some event. Detecting a non empty queue can be done by continu-

ously polling the queue head pointer, but with the *Counting semaphores* of the real time kernel a more elegant solution is possible. The queue handling task begins with a *RTKWait(csemaphore)* instruction which suspends the task giving other tasks more processing time. Each time when an element is added to the FIFO queue a *RTKSignal(csemaphore)* instruction is given which creates 1 pass condition for the *RTKWait(csemaphore)* instruction. Because the number of times a *RTKWait(Csemaphore)*instruction can be passed equals the number of *RTKSignal(csemaphore)* instructions, busy form of waiting can be avoided.

Busy form of waiting can not completely be banished in the speech queue task. A polling function is used to detect if speech, coming from either the GUS board or the synthetic speech board, is audible. After each polling cycle the task is suspended for a while in order to give tasks with a lower priority processing time.

6.1 Object data dictionary

Four types of Audio objects can be distinguished:

NonSpeech Contains information suitable for a music synthesizer.

PrerecSpeech Contains information suitable for the Gravis Ultra Sound (GUS) board.

SynthSpeech Contains information suitable for the synthetic speech board.

AudioList Contains a list build up with the former 3 object types.

6.2 Object information model

Audio objects contain the following global attributes (same attribute in each instance).

nsport Reference to a *MIDI_Port* object ([6, Poll]).

psport Reference to a *SoundBoard* object ([5, Poll]).

ssport Reference to a *SpeechSynth* object (3.5).

sq Reference to a FIFO queue with all ready to play speech audio. At most one speech item must be audible at any time, eventually together with one or more non-speech audio items.

sem_mbox Counting *Semaphore* used by the RTK, to avoid busy form of waiting.

sem_queue Critical section protection *Semaphore* used by the RTK.

Audio objects contain the following local attributes:

identifier Pointer to the information content of the audio object. In the case of *NonSpeech* and *PrerecSpeech* identifier is pointer to a filename, in the case of a *SynthSpeech* object identifier is a pointer to a text string.

status Status information about the object instance.

AudioList objects might contain multiple attributes, one for each list element as depicted in figure 6.1.

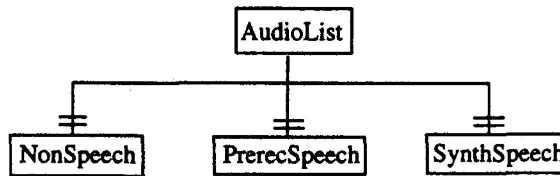

Figure 6.1: AudioList object relations

6.3 Object behaviour model

The figures 6.2, 6.3, 6.4 and 6.5 depict the objects behaviour of respectively the NonSpeech, PrerecSpeech, SynthSpeech and AudioList object behaviour.

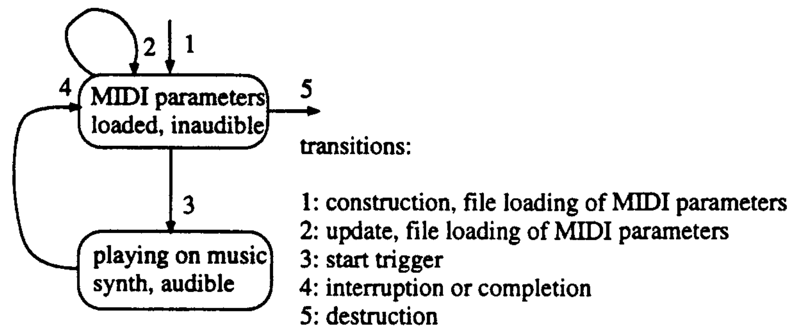

Figure 6.2: State transition diagram of the Nonspeech object

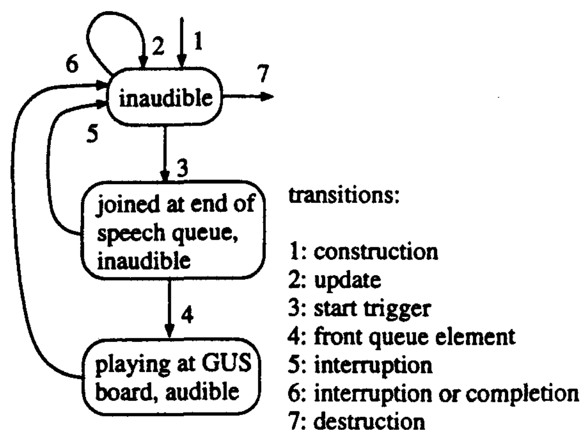

Figure 6.3: State transition diagram of the PrerecSpeech object

The AudioList object behaviour is depicted in figure 6.5. The state following transition 3 implies parallel execution of the previous 3 state transition diagrams.

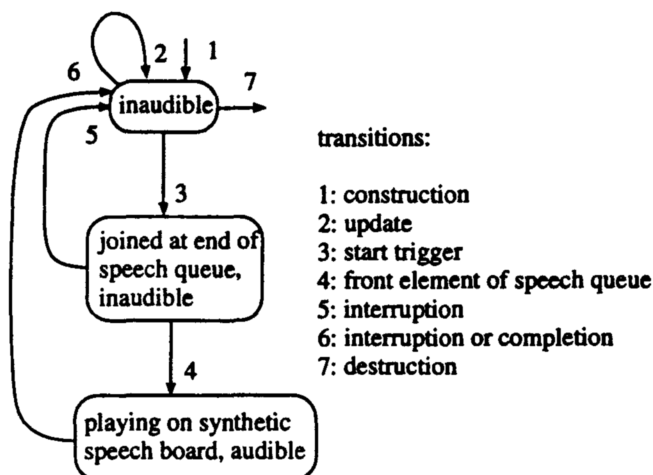

Figure 6.4: State transition diagram of the SynthSpeech object

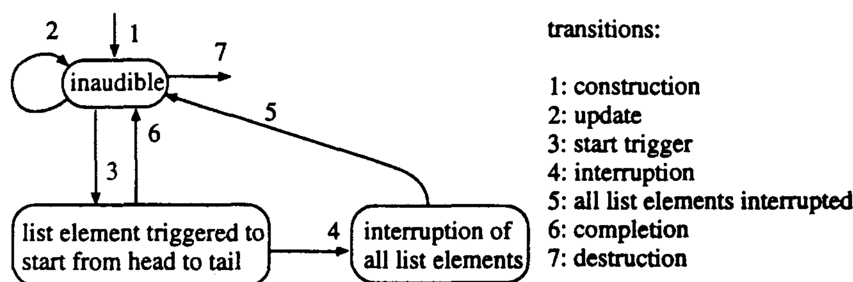

Figure 6.5: State transition diagram of the AudioList object

6.4 Object process model

This section describes the variables and functions that are used within the Audio module.

status	→	{IDLE,QUEUE,PLAY}	;object sound status
	→	∪ {1..16}	;
identifier	→	char pointer	;to a text string
nsport	→	MIDI_Port	;object to control non-speech
psport	→	SoundBoard	;object to control prerecorded speech
ssport	→	SpeechSynth	;object to control synthetic speech
sq	→	Queue	;FIFO speech list
sem_mbox	→	Semaphore	;prevents busy form of waiting
sem_queue	→	Semaphore	;FIFO speech list protect semaphore

6.4.1 Constructors

NonSpeech: Besides the above mentioned attributes, which are available in all Audio objects, NonSpeech objects have the following attributes to hold the MIDI-sequences needed for playing the audicons and earcons as described in [12, Joep]:

channel	→	{1..16} ∪ {-1} ;MIDI channel number
prg	→	{0..127} ∪ {-1};MIDI program change number
ctrl1	→	{0..127} ∪ {-1};MIDI controller number
ctrl1_val	→	{0..127} ∪ {-1};MIDI controller data belonging to ctrl1
ctrl2	→	{0..127} ∪ {-1};MIDI controller number
ctrl2_val	→	{0..127} ∪ {-1};MIDI controller data belonging to ctrl2
key1	→	{0..127} ∪ {-1};MIDI note number
vel1	→	{0..127} ∪ {-1};velocity value belonging to key1
key2	→	{0..127} ∪ {-1};MIDI note number
vel2	→	{0..127} ∪ {-1};velocity value belonging to key2
hold_time	→	{1..} ∪ {-1} ;hold time

All these attributes are read from a file, with name pointed at by the identifier attribute, at construction/update time. In these files all values of the above mentioned attributes are listed, in the same order, separated by commas. Undefined attributes must be set to a "−1" value.

In the next pseudo code, parameters surrounded by [ ] are optional.

```
NonSpeech([char *filename])
{
   if (filename!=NULL)
   {
      identifier=AllocMem(filename);
      OpenFile(identifier);
      read all MIDI attributes from the file;
      CloseFile(identifier);
   }
   else
      identifier=NULL;
   status=IDLE;
}
```

PrerecSpeech:

```
PrerecSpeech([char *filename])
{
   if (filename!=NULL)
      identifier=AllocMem(filename);
   else
      identifier=NULL;
   status=IDLE;
}
```

SynthSpeech:

```
SynthSpeech([char *string])
{
   if (string!=NULL)
      identifier=AllocMem(string);
   else
      identifier=NULL;
   status=IDLE;
}
```

AudioList:

```
AudioList()
{
   head=NULL;
   tail=NULL;
}
```

6.4.2 Initialization

init: This function must be executed before any Audio object is used.

```
void init()
{
   make this program a real time kernel task;
   initialize semaphores;
   create "TaskSpeech" task with higher priority than this one;
}
```

TaskSpeech: Endless task managing the speech being uttered.

```
void TaskSpeech()
{
   Audio *lastactive=NULL;
   do (as long as the parent task is alive)
   {
      suspend until the FIFO queue becomes filled;
      wait until no speech is uttered;
      if (lastactive)
         lastactive.status=IDLE;
      entering critical section;
         start the "uttering" of front queue element;
         lastactive=front queue element;
         remove the front queue element;
      leaving critical section;
   }
}
```

6.4.3 Sound control

The sound information of audio objects can be controlled by 3 functions:

start() Makes the sound information audible as soon as possible. For non-speech this means immediately, for speech this means after the the sound information of all objects started before this one has been uttered.

play() Makes the sound information audible immediately.

stop() Interrupts audible information or removes the object from the speech queue.

6.4.3.1 start()

NonSpeech:

```
void  start()
{
   if (object contains sound information)
      play();
}
```

PrerecSpeech:

```
void  start()
{
   if (object contains sound information)
   {
      entering critical section;
         Put object in FIFO speech queue;
         status=QUEUE;
      leaving critical section;
      signal TaskSpeech() task;
   }
}
```

SynthSpeech:

```
void  start()
{
   if (object contains sound information)
   {
      entering critical section;
         Put object in FIFO speech queue;
         status=QUEUE;
      leaving critical section;
      signal TaskSpeech() task;
   }
}
```

6.4.3.2 play()

NonSpeech:

```
void  play()
{
   transmit the activate information encoded in the MIDI attributes;
   status=PLAY;
}
```

PrerecSpeech()

```
void play()
{
   status=psport.PlayFile(identifier);
}
```

SynthSpeech:

```
void play()
{
   ssport.Speak(identifier);
   status=PLAY;
}
```

6.4.3.3 stop()

NonSpeech:

```
void stop()
{
   transmit deactivate information encoded in the MIDI attributes;
   status=IDLE;
}
```

PrerecSpeech:

```
void stop()
{
   if (status==QUEUE)
   {
      entering critical section;
         remove object from FIFO queue;
         status=IDLE;
      leaving critical section
   }
   else if (status!=IDLE)
   {
      psport.StopPlay(status,NOW);
      status=IDLE;
   }

}
```

SynthSpeech:

```
void stop()
{
   if (status==QUEUE)
   {
      entering critical section;
         remove object from FIFO queue;
         status=IDLE;
      leaving critical section
   }
   else if (status!=IDLE)
   {
      ssport.Break();
      status=IDLE;
   }

}
```

6.4.4 AudioList functions

The 3 discussed sound control functions are also available for the AudioList type audio object, and are not more than sequences of functions for all objects in a list. The AudioList type object has some extra functions for list manipulation purposes:

insert: Add audio object at the end of the linked list or after the one pointed at by an optional parameter.

```
Audio *insert(char *identifier, \{NS,PS,SS\} type, [Audio *afterobject])
{
   if (type==NS)
      construct new NonSpeech(identifier) object;
   else if (type==PS)
      construct new PrerecSpeech(identifier) object;
   else
      construct new SynthSpeech(identifier) object;
   if (afterobject)
         insert constructed object after afterobject in the list;
      else
         add constructed object at the end of the list;
   return address of constructed object;
}
```

search: Return pointer to an audio object with given identifier or NULL if not found.

```
Audio *search(char *seed)
{
   if (an audio object with identifier seed exists)
      return address of found object;
   else
      return NULL;
}
```

remove: Remove an audio object.

```
{SYS_SUCCES,SYS_FAIL} remove(char *seed)
{
   Audio *removeobject=search(seed);
   if (seed!=NULL)
   {
      remove removeobject from the list;
      return SYS_SUCCES;
   }
   else return SYS_FAIL;
}
```

flush: Remove all audio objects from the list.

```
flush()
{
while (AudioList contains audio objects)
    remove an audio object;
}
```

replace: Replace an audio object with another one.

```
{SYS_SUCCES,SYS_FAIL} replace(char *seed, char *newident, {NS,PS,SS}type)
{
   Audio *removeobject=search(seed);
   if (seed!=NULL)
   {
      insert(newindent,type,removeobject);
      remove removeobject from the list;
      return SYS_SUCCES;
   }
   else return SYS_FAIL;
}
```

6.5 User manual

This section gives information about the use of the in C++ developed audio module (text between [ ] is optional).

In all files were the audio objects are used the header file *audio.h* must be included. Before any audio object can be used the initialization function *Audio::init()* must be executed. Four types of audio object instances can be declared.

6.5.1 Declaration

`NonSpeech instance_name[(char *filename)]`
With *filename* an optional pointer to a file name with the following, single line, format: `int,int,int,int,int,int,int,int,int,int,int`. Undefined fields have to be set to -1. The directory from which the files are loaded is defined by `NONSPEECH_PATH` in the *audio.h* header file. This audio type is able to control a music synthesizer via the GUS MIDI interface.

`PrerecSpeech instance_name[(char *filename)]`
With *filename* an optional pointer to a sample file name without extension. The directory from which the files are loaded is defined by `PREREC_PATH` in the *audio.h* header file. This audio type is able to control the GUS sample player.

`SynthSpeech instance_name[(char *string)]`
With *string* an optional pointer to a text string. This audio type is able to control the stand-alone text-to-speech system.

`AudioList`: This audio type can contain a list of the former 3 audio types and is able to control all there resources.

6.5.2 Object instance update

The 3 single object types have only one update function:
`void set(char *ident)`
Changes the (optional) declaration argument to the argument pointed at by ident.

The AudioList object type has the following update functions:

`ListElement *insert(char *ident, int type, [char *afterobj])`
With *ident* the discussed declaration argument, *type* one of {NS,PS,SS} depending on the single object type, and *afterobject* an optional pointer to an identifier already in the list. This function constructs an new object instance of a single audio object type, and adds it to the linked list. If no third argument is given, the constructed instance is placed at the end of the list, else after the object with an identifier equal to *afterobject*. Inserting at the head can be done with help of the predefined **HEAD** sentinel as third argument. A pointer to the constructed instance is returned. *ListElement* is used to refer to elements of 3 possible types. All declaration arguments within one AudioList object instance should be unique.

`ListElement search(char *seed)`
This function searches the list and returns the address of the object with the same declaration argument as *seed* is pointing at. If no matching is found a NULL pointer is returned.

`int replace(char *seed, char *ident, int type)`
This function replaces the object instance with the same declaration argument as *seed* is pointing at, by a new, *type* type of object instance with the same declaration argument as *ident* is pointing at. If no matching is found `SYS_FAIL` is returned, SYS_SUCCES otherwise.

`int remove(char *seed)`
This function removes the object instance with the same declaration argument as *seed* is pointing at. If no matching is found `SYS_FAIL` is returned, `SYS_SUCCES` otherwise.

`void flush()`
This function removes all object instances from the list.

`void empty()`
This function initializes the list to an empty one, without removing object instances. Calling this function is only legal in case of an already empty list, or when all object instances are part of another AudioList instance.

6.5.3 Sound control

Playing sound information on the 3 resources is done with the following functions which are available for all 4 object types:
`void start()`

The sound information encoded in the object attributes is transmitted to the right resources and made audible.

`void stop()`
All sound information encoded in the object attributes and audible via the different resources is made inaudible, and the resources are released.

The order of sound control of an AudioList object is from begin to end. Speech information is played sequentially concurrent with non-speech information.

Chapter 7

CSM extension

This chapter describes the inserting, updating and audibility of audio information in each CSM GUI object, with the use of the developed *Audio* object types. The idea is the following: Each GUI object in the CSM gets an instance of the same set of audio attributes based on the classification discussed in chapter 5. The audio information of these attributes is kept up-to-date as long as the CSM object is alive. Making the sound information of these attributes audible at the right moment results in a visual to audio transformation. Because all CSM GUI objects have the same audio attributes (possible empty), this algorithm is not very complex and yields real time performance.

7.1 Object data dictionary

No extra objects are introduced at the CSM level. Each CSM GUI object gets an extra class of attributes as discussed in the next section

7.2 Object information model

The categorisation made so far can be used to model the following attribute class:

entrance Indicates how sound is used on entrance of an object.

above Indicates how non-speech sound is used when the pointer is above an object.

right mouse button A list of the words that are spoken when the right mouse button is held down.

pop-up menu A list of menu items that are present in the pop-up menu which can be made audible by holding down the right mouse button. When an item is selected with the left mouse button an user action is initiated.

appearance Indicates whether or not sound is used to indicate the appearance of an object.

disappearance Indicates whether or not sound is used to indicate the disappearance of an object.

dragging Indicates whether or not sound is used to indicate the dragging of an object.

state change Indicates how the state changes are presented.

In the following OIM tables for each GUI object the auditory/tactile attributes are defined. The symbol $\natural$ indicates non-speech sound.Text between " " indicates prerecorded speech.Variables are given between ⟨ ⟩ and indicate prerecorded or synthetic speech. the symbol + is used as concatenation operator. Variables that are not always present are surrounded by [ ]. Variables of which two or more values can occur simultaneous, have the maximum amount printed in superscript. Text between { } enumerates the possible values a variable can have. An *X* indicates that the attribute has no effect at all for the object it belongs to.

Object name: **Button**	
entrance	$\natural$ +"button" +⟨information⟩+⟨state⟩
above	$\natural$ (⟨state⟩)
right mouse button	⟨information⟩+"button" +⟨state⟩
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	⟨state⟩+"button" +⟨information⟩
⟨state⟩	{"default","not selectable"}
⟨information⟩	textual content of the child-object **text_line**

Object name: **Check_box**	
entrance	$\natural$ +"check box" +⟨information⟩+⟨state⟩^{2}
above	$\natural$ (⟨state⟩)
right mouse button	⟨information⟩+"check box" +⟨state⟩^{2}
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	⟨state⟩+"check box" ⟨information⟩
⟨state⟩	{"default","not selectable","marked","unmarked"}
⟨information⟩	textual content of the child-object **text_line**

Object name: **Client_area**	
entrance	X
above	X
right mouse button	⟨information⟩
pop-up menu	⟨information⟩
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	{"not active"}
⟨information⟩	parent **Window** object attribute of the same name

Object name: **Close_but** [1]	
entrance	X
above	X
right mouse button	X
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	X

Object name: **Cursor**	
entrance	X
above	X
right mouse button	X
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	X

Object name: **Dialog_box**	
entrance	♮ +"dialog box" +⟨information⟩+⟨state⟩
above	♮ (⟨state⟩)
right mouse button	⟨information⟩+"dialog box" +⟨state⟩
pop-up menu	{["close dialog box"]
appearance	♮
disappearance	♮
dragging	♮
state change	⟨state⟩+"dialog box" ⟨information⟩
⟨state⟩	{"not active"}
⟨information⟩	textual content of the child-object **text_line**

Object name: **Dial** [2]	
entrance	♮ +"dial" +⟨information⟩+⟨state⟩
above	♮ (⟨state⟩)
right mouse button	⟨information⟩+"dial" +⟨state⟩
pop-up menu	{["close dial"]
appearance	♮
disappearance	♮
dragging	♮
state change	⟨state⟩+"dial" +⟨information⟩
⟨state⟩	{"not active"}
⟨information⟩	value content of the child-object **values**

Object name: **Dia_client**	
entrance	X
above	X
right mouse button	⟨information⟩
pop-up menu	⟨information⟩
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	{"not active"}
⟨information⟩	parent **Dialog_box** object attribute of the same name

Object name: **Edit_contr**	
entrance	$\natural$ +”edit control” +⟨information⟩
above	$\natural$
right mouse button	⟨information⟩+”edit control”
pop-up menu	pop-up menu content of parent object
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	textual content of the dialog box title of which this object is a child

Object name: **Edit_line**	
entrance	$\natural$ +”edit line” +⟨information⟩
above	X
right mouse button	⟨information⟩+”edit line”
pop-up menu	pop-up menu content of parent object
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	textual content of the child object **text_line**

Object name: **Graphic**	
entrance	$\natural$ +”graphic”
above	$\natural$
right mouse button	”graphic”
pop-up menu	X
appearance	X
disappearance	X
dragging	$\natural$
state change	X
⟨state⟩	X
⟨information⟩	X

Object name: **Group_box**	
entrance	X
above	♮
right mouse button	⟨information⟩+"group box"
pop-up menu	pop-up menu content of parent object **dialog_box**
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	X

Object name: **Icon**	
entrance	♮ +"icon" +⟨information⟩+⟨state⟩
above	♮
right mouse button	⟨information⟩+"icon" +⟨state⟩
pop-up menu	X
appearance	X
disappearance	X
dragging	♮
state change	⟨state⟩+"icon" +⟨information⟩
⟨state⟩	{"default"}
⟨information⟩	textual content of the child object **text_line**

Object name: **List_box**	
entrance	♮ +"list box" +⟨information⟩+⟨state⟩
above	♮
right mouse button	⟨information⟩+"list box" +⟨state⟩
pop-up menu	X
appearance	♮ (only if parent object is **PD_edit**)
disappearance	♮ (only if parent object is **PD_edit**)
dragging	X
state change	X
⟨state⟩	{"not active"}
⟨information⟩	textual content of parent object's **text_line**

Object name: **Menu_item**	
entrance	"menu item" +⟨information⟩+⟨state⟩
above	♮
right mouse button	⟨information⟩+"menu item" +⟨state⟩
pop-up menu	pop-up menu content of parent object
appearance	X
disappearance	X
dragging	X
state change	⟨state⟩+"menu item" +⟨information⟩
⟨state⟩	{"default","not selectable"}
⟨information⟩	textual content of the child object **text_line**

Object name: **Menu_list**	
entrance	♮ +"menu list" +⟨state⟩
above	♮ (⟨state⟩)
right mouse button	"menu list" +⟨state⟩
pop-up menu	X
appearance	♮ (⟨information⟩)
disappearance	♮ (⟨information⟩)
dragging	X
state change	X
⟨state⟩	{"horizontal","vertical"}
⟨information⟩	textual content of child object **menu_item** which is default.

Object name: **PD_edit**	
entrance	♮ +"pull down edit" +⟨information⟩+⟨state⟩
above	♮
right mouse button	⟨information⟩+"pull down edit" +⟨state⟩
pop-up menu	{["pull up"]}
appearance	X
disappearance	X
dragging	X
state change	⟨state⟩+"pull down edit" +⟨information⟩
⟨state⟩	{"default","not selectable"}
⟨information⟩	textual content of child object **edit_line**

Object name: **Pointer**	
entrance	X
above	X
right mouse button	X
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	♮ (⟨state⟩)
⟨state⟩	{"busy"}
⟨information⟩	X

Object name: **Radio_but**	
entrance	♮ +"radio button" +⟨information⟩+⟨state⟩[2]
above	♮ (⟨state⟩)
right mouse button	⟨information⟩+"radio button" +⟨state⟩[2]
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	⟨state⟩+"radio button" +⟨information⟩
⟨state⟩	{"default","not selectable","on","off"}
⟨information⟩	textual content of the child object **text_line**

Object name: **Resize_corn**	
entrance	X
above	X
right mouse button	X
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	X

Object name: **Scroll_bar**	
entrance	X
above	X
right mouse button	X
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	X

Object name: **Slider**	
entrance	X
above	X
right mouse button	X
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	X

Object name: **Text_line**	
entrance	$\natural$ +"text line" +⟨information⟩
above	X
right mouse button	⟨information⟩+"text line"
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	textual content of this object

Object name: **Window**	
entrance	♮ +"window" +⟨information⟩+⟨state⟩
above	♮ (⟨state⟩)
right mouse button	⟨information⟩+"window" +⟨state⟩
pop-up menu	{["close window"],["restore window"] ,["minimize window"], ["maximize window"]}
appearance	♮
disappearance	♮
dragging	♮
state change	"active window" +⟨information⟩(only if window becomes active)
⟨state⟩	{"not active"}
⟨information⟩	textual content of the child object **text_line**

Object name: **Win_border**	
entrance	X
above	X
right mouse button	X
pop-up menu	X
appearance	X
disappearance	X
dragging	X
state change	X
⟨state⟩	X
⟨information⟩	X

7.3 Object behaviour model

Audio related behaviour of CSM GUI objects can be partitioned into the following classes:

point at Related to the object which is visually under the pointer.

events Not directly related to the pointer location.

guidance Mutual relation between objects and the pointer location.

right mouse button Related to the right mouse button.

pop-up menu Extra CSM object.

7.3.1 Point at

In figure 7.1 the state transition diagram is given of this behaviour class. To each state transition in figure 7.1 belongs the following sound control action:

1. no action
2. make **entrance** attribute audible; make **above** attribute audible
3. make **entrance** and **above** attributes inaudible

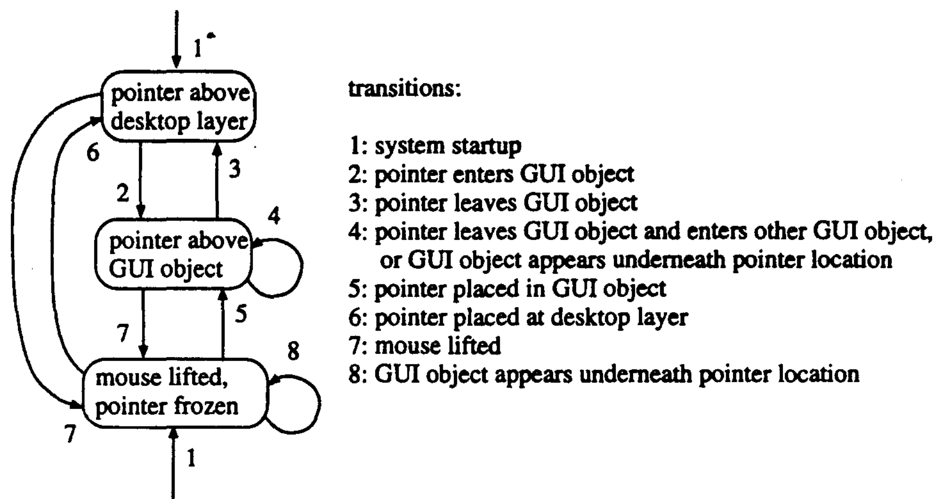

Figure 7.1: state transition diagram **point at** behaviour

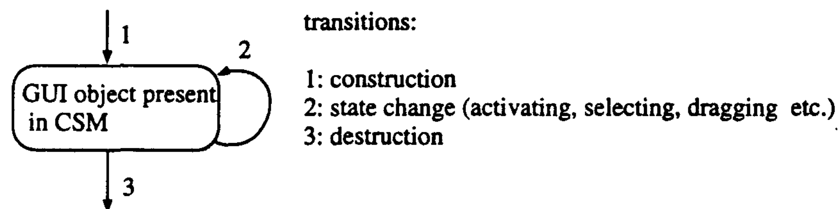

Figure 7.2: state transition diagram **events** behaviour

4. make **entrance** and **above** attributes of leaving object inaudible;make **entrance** and **above** attributes of entering object audible, or make **appear** and **above** attributes of appearing object audible.
5. make **above** attribute audible
6. no action
7. no action
8. make **appear** and **above** attributes audible

7.3.2 Events

In figure 7.2 the state transition diagram of this behaviour class is given. To each state transition in figure 7.2 belongs the following sound control action:

1. make **appearance** attribute audible
2. make **state change** attribute audible
3. make **disappearance** attribute audible

7.3.3 Guidance

A model for guidance is still under research.

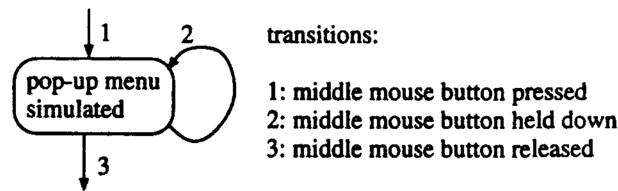

Figure 7.3: state transition diagram **pop-up menu** behaviour

7.3.4 Right mouse button

This one is trivial: When the right mouse button is pressed the **right mouse button** attribute is made audible; at release of the button the **right mouse button** attribute is made inaudible.

7.3.5 Pop-up menu

After pressing the middle mouse button a *Menu_list* object is simulated located under, and attached to, the pointer position. Instead of simulation, a real GUI *Menu_list* object could be added to the CSM, but then the menu items would be uttered by the synthetic speech synthesizer with an inferior sound quality compared to the GUS sample player. The menu items are coded in the **pop-up menu** attribute of the CSM GUI When a menu item from the pop-up menu is selected, the corresponding GUI command is executed. The pop-up menu is removed after the release of the middle mouse button. In figure 7.3 the state transition diagram of this object is given.

GUI objects modelled by the pop-up menu are not audible presented by the CSM. The resulting 'physical gaps' in the screen representation have to be filled with stretched neighbour GUI objects. This problem is not trivial and needs further investigation.

7.4 Object process model

The following audio related attributes and child objects are used by the GUI objects:

Aud_Entrance	→	Linked list of Audio_objects
Aud_Above	→	NonSpeech Audio_object
Aud_Right_mouse_button	→	Linked list of Audio_objects
Aud_Pop-up_menu	→	Linked list of Text_line objects
Aud_Appear	→	NonSpeech Audio_object
Aud_Disappear	→	NonSpeech Audio_object
Aud_Drag	→	NonSpeech Audio_object
Aud_State_change	→	Linked list of Audio_objects

The initialisation and updating of audio attributes is added to the existing CSM in 3 modules:

1. The GUI object construction module (section 5.3 of [9, Gerrits,Poll,Waterham]).

2. The GUI object update module (section 5.4 of [9, Gerrits,Poll,Waterham]).
3. The user action module (chapter 4 of [10, Gerrits,Poll,Waterham]).

The pseudo code for the addition of audio related attributes in the construction module is given in appendix A. Because coding of the initialisation and updating turned out to be a very straightforward translation of the object information tables, further generation of pseudo code is omitted.

Chapter 8

Conclusions

The implemented audio module has been tested in a stand alone program and seems to work properly. The extended CSM has been tested with simulation input and functioned as suspected. Further testing, especially with dynamic model behaviour, is necessary in order to improve software quality.

The coding of audio information into CSM GUI objects guarantees consistency and avoids time consuming case distinction algorithms. The use of a general object for audio handling improves flexibility, extendability and understandability for future software engineering tasks. Disadvantages of this approach are the abundant consumption of computer memory resources and the intensive disk access. Both disadvantages can be overcome by the use of some *sound cache* system. Intensive disk access can also be reduced by the use of a large disk cache on the drive on which the prerecorded sound files and the MIDI parameter files are located.

The sound quality of the speech synthesizer is not optimal, especially when more than one language is used. The use of two different resources for speech is expected to cause confusion and should maybe therefore be replaced by one high quality speech synthesizer.

Bibliography

[1] R.P. Waterham, J.J. Beumer, P.B. Klimbie, L.H.D. Poll, *Results of the VISA-comp evaluation.*, IPO report no.958.

[2] L.H.D. Poll, *Software Quality Doc*, internal IPO document.

[3] L.H.D. Poll, *Digitizer software engineering documents*, software engineering document.

[4] R.J.H. Deliege *User manual stand-alone text-to-speech system*, IPO handleiding no. 100.

[5] L.H.D. Poll, *Gravis Ultrasound soundboard*, software engineering document.

[6] L.H.D. Poll, *MIDI software engineering documents*, software engineering document.

[7] L.H.D. Poll, *Digitizer*, software engineering document.

[8] **On Time** informatik gmbh, *RTKernel: Real-Time Multitasking Kernel for C*, user's manual version 3.0.

[9] A.H.J. Gerrits, L.H.D. Poll, R.P. Waterham, *VISA-Comp current screen module (software engineering document)*, IPO Report no. 915.

[10] A.H.J. Gerrits, L.H.D. Poll, R.P. Waterham, *VISA-Comp object library (software engineering document)*, IPO Report no. 916.

[11] L.H.D. Poll, *Non-visual access to Graphical User Interfaces (a user interface concept)*,internal IPO document.

[12] J. Fruman, *Sound design for an auditory reproduction of a Graphical User Interface*, IPO Report no. 1053.

Appendix A

Object process model

This section gives pseudo code for the audio attribute initialisation within the existing GUI object construction module.

A.1 Windows

```
Window constructor()
{
     Aud_Entrance=make_linked_list(\nonspeech,"window",Text_line);
     Aud_Above=\nonspeech(type=active);
     Aud_Right_mouse_button=make_linked_list(Text_line,"window");
     Aud_Pop-up_menu=make_linked_list("close window")
     if (maximize button available)
          append_item(Aud_Pop-up_menu,"maximize window");
     if (minimize button available)
          append_item(Aud_Pop-up_menu,"minimize window");
     Aud_Appear=\nonspeech;
     Aud_Disappear=\nonspeech;
     Aud_Drag=\nonspeech;
     Aud_State_change=make_linked_list("active window",Text_line);
     // Window becomes always active at construction
     PlaySound(Aud_Appear,Aud_State_change);
}
```

A.2 Client area

```
Client_area constructor()
{
     if (horizontal menu list available within client area)
          construct menu_line;
     for (all icons present within the client area)
          construct icon;
     for (all child windows present within the client area)
          construct window;
     for (all pull-down edits present within the client area)
          construct pull-down edit;
     for (all edit controls present within the client area)
          construct edit control;
     for (all listboxes present within the client area)
          construct listbox;
     for (all buttons present within the client area)
          construct button;
     for (all text present within the client area)
          construct textline;
     for (all graphics present within the client area)
          construct Graphic;
     Aud_Entrance=NULL;
     Aud_Above=NULL;
```

```
    Aud_Right_mouse_button=parent_window->Aud_Right_mouse_button;
    Aud_Pop-up_menu=parent_window->Aud_Pop-up_menu;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
}
```

A.3 Button

```
Button constructor()
{
    Aud_Entrance=make_linked_list(\nonspeech,"button",Text_line);
    Aud_Above=\nonspeech(button_type);
    Aud_Right_mouse_button=make_linked_list(Text_line,"button");
    Aud_Pop-up_menu=NULL;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    if (button_type=BUTTON_DEFAULT)
    {
        Aud_State_change=make_linked_list("default","button",Text_line);
        append_item(Aud_Entrance,"default");
        append_item(Aud_Right_mouse_button,"default");
        PlaySound(Aud_State_change);
    }
    if (text_line not selectable)
    {
        Aud_State_change=make_linked_list("not selectable","button",Text_line);
        append_item(Aud_Entrance,"not selectable");
        append_item(Aud_Right_mouse_button,"not selectable");
        PlaySound(Aud_State_change);
    }
    if (there is a Graph on the right of the check_box)
        construct graphic;
}
```

A.4 Check box

```
Check_Box constructor()
{
    Aud_Entrance=make_linked_list(\nonspeech,"check box",Text_line);
    Aud_Above=\nonspeech(checkbox_type);
    Aud_Right_mouse_button=make_linked_list(Text_line,"check box");
    Aud_Pop-up_menu=NULL;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=make_linked_list();
    if (checkbox_type=CHECK_BOX_ON)
    {
      append_item(Aud_State_change,"marked");
      append_item(Aud_Right_mouse_button,"marked");
    }
    if (checkbox_type=CHECK_BOX_OFF)
    {
      append_item(Aud_State_change,"unmarked");
      append_item(Aud_Right_mouse_button,"unmarked");
    }
    if (checkbox_type=DEFAULT)
    {
      append_item(Aud_State_change,"default");
      append_item(Aud_Right_mouse_button,"default");
    }
    if (text line not selectable)
    {
```

```
        append_item(Aud_State_change,"not selectable");
        append_item(Aud_Right_mouse_button,"not selectable");
    }
    PlaySound(Aud_State_change);
    if (there is a graphic on the right of the check_box)
        construct graphic;
}
```

A.5 Close button

```
Close_But constructor()
{
    if (graphic is present)
        construct graphic;
    if (a vertical menu_list is present)
        construct vertical menu_list;
    // audio behaviour modelled as part of parent object (window,dialog_box)
    Aud_Entrance=NULL;
    Aud_Above=parent->Aud_Above;
    Aud_Right_mouse_button=parent->Aud_Right_mouse_button;
    Aud_Pop-up_menu=parent->Aud_Right_mouse_button;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
}
```

A.6 Cursor

```
Cursor constructor()
{
    Aud_Entrance=NULL;
    Aud_Above=NULL;
    Aud_Right_mouse_button=NULL;
    Aud_Pop-up_menu=NULL;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
}
```

A.7 Dialog-box

```
Dialog_box constructor()
{
    Aud_Entrance=make_linked_list(\nonspeech,"dialog box",Text_line);
    Aud_Above=\nonspeech(type=active);
    Aud_Right_mouse_button=make_linked_list(Text_line,"dialog box");
    if (close_button available)
    {
        construct close_but();
        Aud_Pop-up_menu=make_linked_list("close dialog box");
    }
    else Aud_Pop-up_menu=NULL;
    Aud_Appear=\nonspeech;
    Aud_Disappear=\nonspeech;
    Aud_Drag=\nonspeech;
    Aud_State_change=NULL;
    // Dialog box becomes always active at construction
    PlaySound(Aud_Appear);
}
```

A.8 Dial

Not yet implemented !

A.9 Dia-client

```
Dia_client constructor()
{
     for (all pull-down edits present within the client area)
          construct pull-down edit;
     for (all edit controls present within the client area)
          construct edit control;
     for (all listboxes present within the client area)
          construct listbox;
     for (all groupboxes present within the client area)
          construct groupbox;
     for (all radio buttons not grouped by a groupbox present within the client area)
          construct radio button;
     for (all checkboxes present within the client area)
          construct checkbox;
     for (all buttons present within the client area)
          construct button;
     for (all text present within the client area)
          construct textline;
     for (all graphics present within the client area)
          construct graphic;
     Aud_Entrance=NULL;
     Aud_Above=NULL;
     Aud_Right_mouse_button=parent_dialog_box->Aud_Right_mouse_button;
     Aud_Pop-up_menu=parent_dialog_box->Aud_Pop-up_menu;
     Aud_Appear=NULL;
     Aud_Disappear=NULL;
     Aud_Drag=NULL;
     Aud_State_change=NULL;
}
```

A.10 Edit-contr

```
Edit_contr constructor()
{
     if (horizontal scrollbar next to the edit_contr is present)
     {
          construct horizontal scrollbar;
     }
     if (vertical scrollbar next to the edit_contr is present)
          construct vertical scrollbar;
     for (all edit_lines present within the edit_contr)
          construct edit_line;
     if (cursor present within the edit_contr)
          construct cursor;
     for (all graphics present within the edit_contr)
          construct graphic;
     Aud_Entrance=make_linked_list(\nonspeech,"edit control",parent_dialog_box->Text_line);
     Aud_Above=\nonspeech;
     Aud_Right_mouse_button=make_linked_list(parent_dialog_box->Text_line,"edit
     control");
     Aud_Pop-up_menu=parent_dialog_box->Aud_Pop-up_menu;
     Aud_Appear=NULL;
     Aud_Disappear=NULL;
     Aud_Drag=NULL;
     Aud_State_change=NULL;
}
```

A.11 Edit-line

```
Edit_line constructor()
{
     construct Text_line;
     Aud_Entrance=make_linked_list(\nonspeech,"edit line",Text_line);
     Aud_Above=NULL;
     Aud_Right_mouse_button=make_linked_list(Text_line,"edit control");
```

```
    Aud_Pop-up_menu=parent_dialog_box->Aud_Pop-up_menu;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
    if (text_line is highlighted)
    {
        append_item(Aud_Entrance,"selected");
        append_item(Aud_Right_mouse_button,"selected");
    }
}
```

A.12 Graph

```
Graph constructor()
{
    Aud_Entrance=\nonspeech;
    Aud_Above=\nonspeech;
    Aud_Right_mouse_button=make_linked_list("graph");
    Aud_Pop-up_menu=NULL;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=\nonspeech;
    Aud_State_change=NULL;
}
```

A.13 Group-box

```
Group_box constructor()
{
    for (all radio buttons present in group_box)
        construct radio button;
    for (all checkboxes present in group_box)
        construct checkbox;
    Aud_Entrance=NULL;
    Aud_Above=\nonspeech;
    Aud_Right_mouse_button=make_linked_list(Text_line,"group box");
    Aud_Pop-up_menu=parent_dialog_box->Aud_Pop-up_menu;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
}
```

A.14 Icon

```
Icon constructor()
{
    construct Text_line;
    construct Graphic;
    Aud_Entrance=make_linked_list(\nonspeech,"icon",Text_line);
    Aud_Above=\nonspeech(type);
    Aud_Right_mouse_button=make_linked_list(Text_line,"icon");
    Aud_Pop-up_menu=NULL;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=\nonspeech;
    if (Icon is default)
    {
        append_item(Aud_Entrance,"default");
        append_item(Aud_Right_mouse_button,"default");
        Aud_State_change=make_linked_list("default","icon");
        PlaySound(Aud_State_change);
    }
    else Aud_State_change=NULL;
}
```

A.15 List box

```
List_box constructor()
{
     if (horizontal scroll_bar present)
          construct horizontal scrollbar;
     if (vertical scroll_bar present)
          construct vertical scrollbar;
     for (all icons present in list_box area
          construct icon;
     for (all text_line present in list_box area)
          construct text_line;
     for (all graphics present in list_box area)
          construct graphic;
     Aud_Entrance=make_linked_list(\nonspeech,"list box",parent->Text_line);
     Aud_Above=\nonspeech(type=active);
     Aud_Right_mouse_button=make_linked_list(parent->Text_line,"list box");
     Aud_Pop-up_menu=parent->Aud_Pop-up_menu
     Aud_Appear=NULL;
     Aud_Disappear=NULL;
     if (parent object is PD_edit)
     {
          Aud_Appear=\nonspeech;
          Aud_Disappear=\nonspeech;
          PlaySound(Aud_Appear)
     }
     Aud_Drag=NULL;
     Aud_State_change=NULL;
}
```

A.16 Menu item

```
Menu_item constructor()
{
     if (text is present) construct Text_line;
     if (graphic is present) construct graphic;
     Aud_Entrance=make_linked_list("menu item",Text_line);
     Aud_Above=\nonspeech(menu_item type);
     Aud_Right_mouse_button=make_linked_list(Text_line,"menu item");
     Aud_Pop-up_menu=NULL;
     Aud_Appear=NULL;
     Aud_Disappear=NULL;
     Aud_State_change=NULL;
     if (menu_item type is default)
     {
          append_item(Aud_Entrance,"default");
          append_item(Aud_Right_mouse_button,"default");
          append_item(Aud_State_change,"default","menu item",Text_line);
          PlaySound(Aud_State_change);
     }
     if (menu_item type is not selectable)
     {
          append_item(Aud_Entrance,"not selectable");
          append_item(Aud_Right_mouse_button,"not selectable");
     }
     Aud_Drag=NULL;
     if (vertical submenu list available on right or left of this menu_item)
        construct Menu_list;

}
```

A.17 Menu list

```
Menu_list constructor()
{

     Aud_Entrance=make_linked_list(\nonspeech,"menu list");
```

```
    Aud_Right_mouse_button=make_linked_list("menu list");
    Aud_Pop-up_menu=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
    if (menu list is oriented vertically)
    {
        append_item(Aud_Entrance,"vertical");
        append_item(Aud_Right_mouse_button,"vertical");
        Aud_Above=\nonspeech(type=vertical);
        Aud_Appear=\nonspeech(type=vertical);
        Aud_Disappear=\nonspeech(type=vertical);
    }
    else // menu list is oriented horizontally
    {
        append_item(Aud_Entrance,"horizontal");
        append_item(Aud_Right_mouse_button,"horizontal");
        Aud_Above=\nonspeech(type=horizontal);
        Aud_Appear=\nonspeech(type=horizontal);
        Aud_Disappear=\nonspeech(type=horizontal);
    }
    PlaySound(Aud_Appear);
    for (all menu_items available)
      construct menu_items;
}
```

A.18 Pointer

```
Pointer constructor()
{
    if (pointer state is BUSY)
        PlaySound(\nonspeech);
}
```

A.19 Pull-Down edit

```
PD_edit constructor()
{
    construct Edit_contr;
    if (listbox present)
    {
        construct listbox;
        Aud_Pop-up_menu=make_linked_list("pull up");
    }
    else Aud_Pop-up_menu=NULL;
    Aud_Entrance=make_linked_list(\nonspeech,"pull down edit",edit_control->text);
    Aud_Right_mouse_button=make_linked_list(edit_control->text,"pull down edit");
    Aud_State_change=NULL;
    if (PD_buttontype is default)
    {
        append_item(Aud_Entrance,"default");
        append_item(Aud_Right_mouse_button,"default");
        append_item(Aud_State_change,"default","pull down edit",edit_control->text);
        PlaySound(Aud_State_change);
    }
    Aud_Above=\nonspeech(type=PD_buttontype);
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    }
}
```

A.20 Radio button

```
Radio_but constructor()
{
```

```
    Aud_Entrance=make_linked_list(\nonspeech,"radio button",text_line);
    Aud_Right_mouse_button=make_linked_list(text_line,"radio button");
    Aud_State_change=NULL;
    if (radiobut_type is radio_button_on)
    {
        append_item(Aud_Entrance,"on");
        append_item(Aud_Right_mouse_button,"on");
        Aud_Above=\nonspeech(state=on);
    }
    else
    {
        append_item(Aud_Entrance,"off");
        append_item(Aud_Right_mouse_button,"off");
        Aud_Above=\nonspeech(state=off);
    }
    if (text is not selectable)
    {
        append_item(Aud_Entrance,"not selectable")
        append_item(Aud_Right_mouse_button,"not selectable");
        Aud_Above=\nonspeech(state=not_selectable);
    }
    if (text is default)
    {
        append_item(Aud_Entrance,"default")
        append_item(Aud_Right_mouse_button,"default");
        Aud_Above=\nonspeech(state=default);
        append_item(Aud_State_change,"default",radio button,text_line);
        PlaySound(Aud_State_change);
    }
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
}
```

A.21 Slider

```
Slider constructor()
{
// audio behaviour not modelled but projected at parent object !
    Aud_Entrance=parent->Aud_Entrance;
    Aud_Above=parent->Aud_Above;
    Aud_Right_mouse_button=parent->Aud_Right_mouse_button;
    Aud_Pop-up_menu=parent->Aud_Pop-up_menu;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
}
```

A.22 Scroll bar

```
Scroll bar constructor()
{
// audio behaviour not modelled but projected at parent object !
    Aud_Entrance=parent->Aud_Entrance;
    Aud_Above=parent->Aud_Above;
    Aud_Right_mouse_button=parent->Aud_Right_mouse_button;
    Aud_Pop-up_menu=parent->Aud_Pop-up_menu;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
}
```

A.23 Text line

```
Text_line constructor()
{
    Aud_Entrance=NULL;
    Aud_Above=NULL;
    Aud_Right_mouse_button=NULL;
    Aud_Pop-up_menu=NULL;
    Aud_Appear=NULL;
    Aud_Disappear=NULL;
    Aud_Drag=NULL;
    Aud_State_change=NULL;
    if (parent object is client area or list box)
    {
        Aud_Entrance=make_linked_list(\nonspeech,"text line",content);
        Aud_ Right_mouse_button=make_linked_list(content,"text line");
    }
}
```