ATK

An Application Toolkit for HTK

Version 1.4.1

Steve Young

Email: sjy@eng.cam.ac.uk

Machine Intelligence Laboratory
Cambridge University Engineering Dept
Trumpington Street, Cambridge, CB2 1PZ

Date: 6th July 2004

Contents

1 Overview 4
1.1 The ATK Core 4
1.2 Using ATK in an Application 6
2 Packets 11
2.1 Generic Packet Properties 11
2.1.1 Programming 11
2.2 Empty Packets 12
2.2.1 Programming 12
2.3 String Packets 12
2.3.1 Programming 12
2.4 Command Packets 13
2.4.1 Programming 13
2.5 WaveData Packets 13
2.5.1 Programming 13
2.6 Observation Packets 14
2.6.1 Programming 14
2.7 Phrase Packets 14
2.7.1 Programming 15
3 Buffers 16
3.1 Programming 16
4 Components 17
4.1 Programming 18
5 The Monitor 21
5.1 Programming 22
5.2 Configuration Variables 23
6 System Components 24
6.1 Audio Input (ASource) 24
6.1.1 Configuration Variables 24
6.1.2 Run-Time Commands 25
6.1.3 Programming 25
6.2 Coder (ACode) 26
6.2.1 Configuration Variables 27
6.2.2 Run-Time Commands 28
6.2.3 Programming 29
6.3 Recogniser (ARec) 29
6.3.1 Configuration Variables 31
6.3.2 Confidence Scoring 32
6.3.3 N-Gram Language Models 33
6.3.4 Run-Time Commands 33
6.3.5 Programming 34
7 Resources (ARMan) 35
7.1 Resources and Resource Groups 35
7.1.1 Configuration Variables 35
7.1.2 Programming 36
7.2 Dictionary (ADict) 37
7.2.1 Configuration Variables 37

7.2.2 Programming 37
7.3 Grammar (AGram) 38
7.3.1 Configuration Variables 38
7.3.2 Programming 39
7.4 N-Gram Language Model (ANGram) 40
7.4.1 Configuration Variables 40
7.4.2 Programming 40
7.5 HMMSet (AHmms) 41
7.5.1 Configuration Variables 41
7.5.2 Programming 41
8 Using ATK 43
8.1 Initialisation and Error Reporting 43
8.2 No-Console Mode 43
8.3 ATKLib and Test Programs 44
8.4 Using ATK with Windows MFC 44
8.5 Tuning an ATK Recogniser 47
9 ATK Application Examples 48
9.1 AVite - an ATK-based version of HVite 48
9.2 Simple Spoken Dialogue System (SSDS) 50
Index 53

1 Overview

ATK is an API designed to facilitate building experimental applications for HTK. It consists of a C++ layer sitting on top of the standard HTK libraries. This allows novel recognisers built using customised versions of HTK to be compiled with ATK and then tested in working systems.[1] Like HTK itself, it is portable across the main Unix platforms and Windows (both as a terminal application and an MFC application).[2]

ATK is multi-threaded. It allows multiple user inputs (voice commands, mouse-clicks, gestures, etc) to be combined into a single data stream and it allows multiple data streams and multiple recognisers. Efficiency is a relatively low priority but latency can be reduced to a minimum to enable highly responsive systems to be built.

1.1 The ATK Core

The ATK core is based on three fundamental C++ object classes:

- **Packet**: is a chunk of information. Packets are used for transmitting a variety information between asynchronously executing components. In particular packets are used to convey various forms of user input and output signals (speech, mouse-clicks, etc). In these cases, each packet has a time stamp to define the temporal span to which it relates. The types of data that a packet can carry include text strings, waveform fragments, coded feature vectors, word labels and semantic tags.
- **Buffer**: is a fifo packet queue. Buffers provide the channel for passing packets from one component to another. Buffers can be of fixed size or unlimited size. Components wishing to access a buffer can test to see whether the buffer operation would block before committing to the operation.
- **Component**: is a processing element. Each component is executed within its own individual thread. The three main ATK components are the Audio source component (ASource), the Coder component (ACode) and the Recogniser component (ARec). Components communicate by passing packets via buffers. In addition, components have a command interface which can be used to update control parameters during operation and thereby modify the runtime behaviour of the component. For example, the Audio source component can be made to suspend and then restart sampling when required.

Figure 1 shows a basic system set-up using ATK. The application communicates directly with the Audio component to start and stop "listening", and it communicates directly with the recogniser to control the recognition process, in particular, the application will typically load a new recognition grammar prior to each user utterance.

The recogniser component itself depends on various resource objects (AResource). The three main resource object types are: dictionary (ADict), grammar (AGram) and HMM set (AHmms). A dictionary object defines the way each vocabulary item is pronounced and it is usually initialised by reading from an external file. A grammar object defines a finite state network of allowable word sequences: it can be loaded from a file or created on the fly. Each finite state network can be defined in terms of lower level sub-networks and any word, or sub-network can be given a semantic tag. Word transitions can be given probabilities and additionally, an n-gram language model[3] can be added using an optional fourth resource object type (ANGram). A HMM set contains the HMMs used for the acoustic models and it is always initialised from a file.

HMM sets, dictionaries, grammars and n-gram language models are stored in a resource database managed by a recognition resource manager object (ARMan). Resources can be stored in this

[1] The standard HTK libraries have been extended in places to support the extra functionality needed by ATK. This extra functionality is enabled by the ATK compiler directive. In addition, a new HTK module called HThreads provides basic platform independent thread support.

[2] This does not imply cross platform GUI support beyond that provided by the HTK HGraf library though note that HGraf has been extended to support multiple windows.

[3] ATK supports bigram and trigram language models.

database, edited and deleted. Within the database, resources can be assigned to logical resource groups. When the recogniser is executing, its operation depends on a network compiled from the information stored in a specified resource group. The application can change this resource group as required.

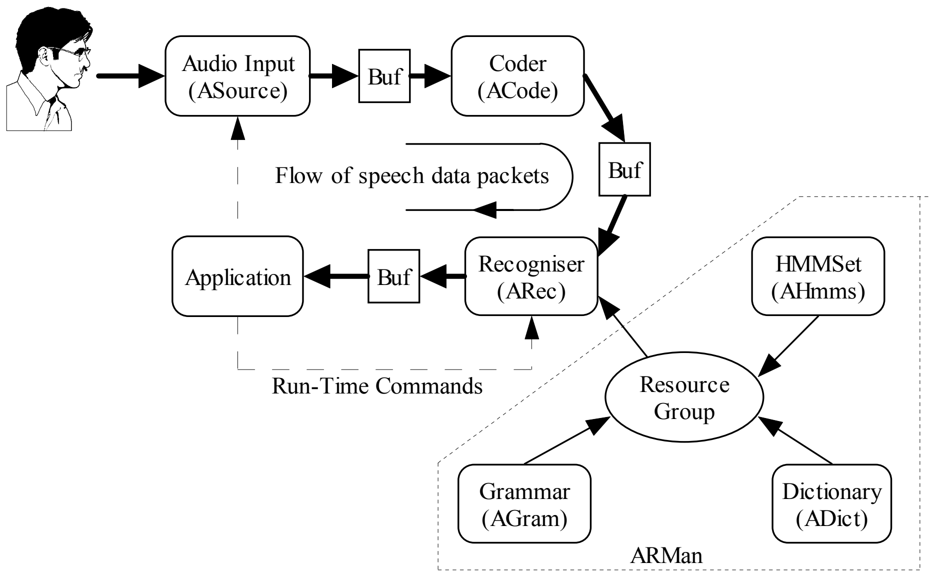


Figure 1 - Basic recognition system

At runtime the recogniser can be configured to run in a variety of modes. These modes control the conditions under which the recogniser starts and stops, whether it restarts automatically or manually, and the method of returning results. With regard to results, the recogniser can return results immediately in the form of its current best guess, as soon as each word becomes fully disambiguated, and in a batch once the end of the utterance has been detected. The recogniser can also return a chart of alternatives[4].

The initial configuration of component objects is determined by a global configuration file which is read once at start-up. This file is also used to configure the HTK libraries. Once the component objects are executing, further configuration changes can only be made via the run-time command interface.

As an example of a more complex configuration, Figure 2 shows a system which uses two recogniser objects and a separate speech/silence detector (the resource groups are omitted for clarity). The use of two recogniser objects, for example, allows one recogniser to operate in word spotting mode, triggering whenever it recognises a key phrase. This is useful for systems where the grammars are loaded from an external source (eg browsing the web by voice), but where there must also be some reliable means of returning to local command mode. This example also shows an extra input source, in this case, packets recording mouse clicks. These latter packets will be passed through the system unchanged until they arrive at the Application itself. This type of architecture would, for example, support a multimodal voice-assisted CAD package in which one could say “Draw a line from here <click mouse> to here <click mouse>”.

Finally it should be noted, that although the examples here focus on the type of interface one might use in a spoken dialogue system, ATK is nevertheless a general purpose interface for building real-time applications which use HTK-derived recognition resources.

[4] This feature is not yet implemented.

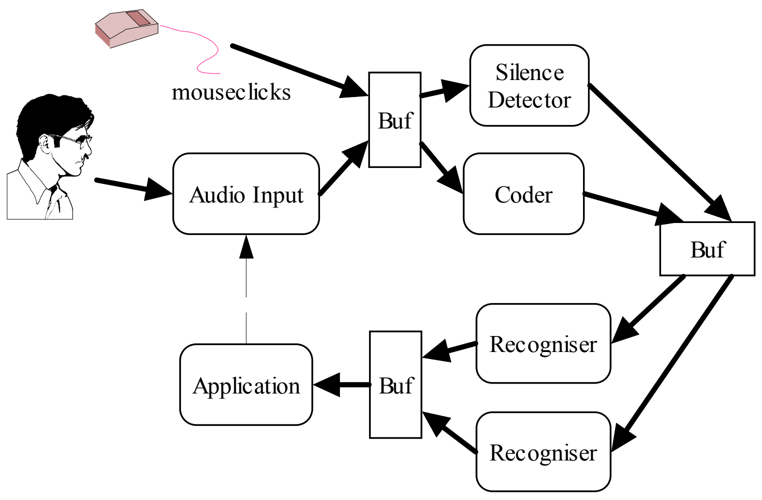


Figure 2 - A more advanced recogniser configuration

1.2 Using ATK in an Application

For many simple applications, interfacing to ATK requires little more than instantiating the required objects and then handling the resulting answer packets from the recogniser. As an example, consider a very simple digit recogniser which might be used in a hands-free dialling program. In this case, the required dictionary and grammar are fixed and can be prepared off-line. The dictionary would contain pronunciations for each digit e.g.

```
ONE          w ah n
TWO           t uw
etc
```

The grammar file would define a simple digit loop as illustrated in Figure 3. In HTK SLF format, this would be represented by a text file containing a list of grammar nodes (the words) and the links needed to loop them together.[5] The following fragment illustrates the basic idea

```
VERSION=1.0
N=16  L=25
I=0    W=!NULL
I=1    W=SIL
I=2    W=!NULL
I=3    W=!NULL
I=4    W=ONE
I=5    W=TWO
...
I=15   W=!NULL
I=7    W=OH
J=0    S=0    E=1
J=1    S=1    E=2
J=2    S=2    E=4
...
J=24  S=14   E=15
```

[5] The HTK SLF format is described in the HTK Book.

The third required resource is a set of HMMs. These would be prepared off-line using HTK, and then stored in any of the supported formats. In this case it will be assumed that the file HMM.list contains the model list and the file HMM.mmf contains the actual models.

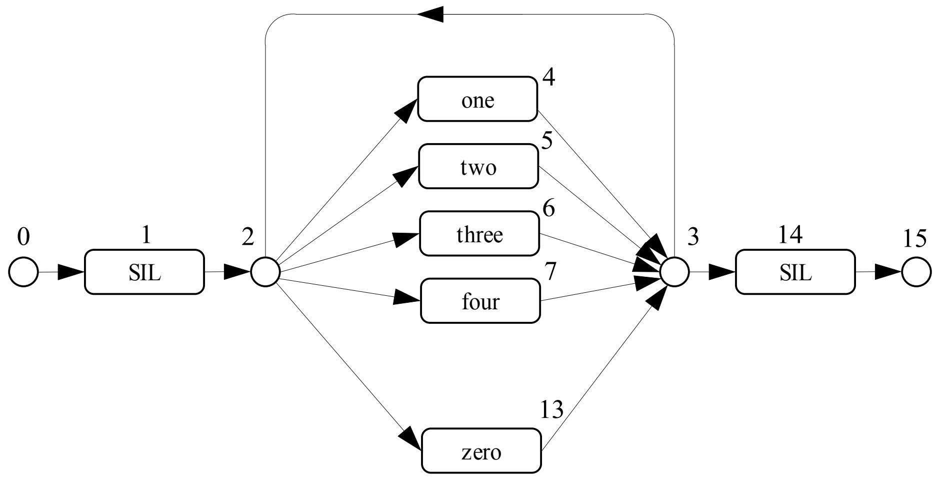


Figure 3 - Simple digit recognition grammar

Each of the three required resources can be defined as entries in a configuration file which is loaded at startup time. Such a file will also typically contain the specification of the coding parameters. The following configuration file is sufficient for this example.

```
# define coding/feature parameters
   SOURCEFORMAT        = HAUDIO
   SOURCERATE          = 625
   TARGETKIND          = MFCC_E_D_N_A
   TARGETRATE          = 100000.0
   USEPOWER            = F
   NUMCHANS            = 24
   CEPLIFTER           = 22
   SILFLOOR            = 50.0
# define needed resources
HMMSET:  HMMLIST       = "HMMS.list"
HMMSET:  MMF0          = "HMMS.mmf"
ADICT:   DICTFILE      = "dialer.dct"
AGRAM:   GRAMFILE      = "dialer.net"
```

Note that the module names where given refer to object instances rather than class names as in HTK.

Having assembled the various resources needed, the required programming is straightforward. The main program must initialise ATK, and the underlying HTK libraries by calling InitHTK as in

```
if (InitHTK(argc,argv,version)<SUCCESS){
  printf("Error: cannot initialise HTK\n"); exit(-1);
}
```

where version is a string giving version information in the standard HTK format[6] eg

```
    char *version = "!HVER!MyApp:   1.1.39 [SJY 31/02/05]";
```

The various objects are then instantiated in the usual way. The architecture will be similar to that shown in Figure 1. First the buffers are defined

```
    ABuffer auChan("auChan");          // waveform data
    ABuffer feChan("feChan");          // features
    ABuffer ansChan("ansChan");        // recognition output
```

and then the source and coder components can be instantiated, connected by the appropriate buffers

```
    ASource ain("AIn",&auChan);        // auChan connects source to coder
    ACode acode("ACode",&auChan,&feChan);   //feChan connects coder to reco
```

Before a recogniser can be instantiated, a resource manager must be created and the various resources stored in it

```
    AHmms hset("HmmSet");              // create HMM set called HmmSet
    ADict dict("ADict");               // create a dictionary called ADict
    AGram gram("AGram");               // create a grammar called AGram
    ARMan rman;                        // create a resource manager
    rman.StoreHMMs(&hset);             // store the resources in rman
    rman.StoreDict(&dict);
    rman.StoreGram(&gram);
```

Note that the actual files used to define the HMM set, dictionary and grammar are defined in the configuration file. The needed resources can now be formed into a resource group and the recogniser instantiated

```
    ResourceGroup *group = rman.NewGroup("dialer");
    group->AddHMMs(&hset);
    group->AddDict(&dict);
    group->AddGram(&gram);
    ARec arec("ARec",&feChan,&ansChan,&rman);        // create recogniser
```

Note that the recogniser has its input connected to the coder output and its output is sent to the buffer called ansChan. The application will read recogniser output via this latter buffer.

The final step is to start up the components

```
    ain.Start();  acode.Start();  arec.Start();
```

By default, the recogniser enters an idle state on initial start-up. It can be started by issuing a start command as in

[6] Note that since an ATK application may not always have a command line interface, the version info normally printed by the -V option can also be obtained by setting the configuration variable PRINTVERSIONINFO=T.

```
arec.SendMessage("start()");
```

Note here that the command "start()" is an ATK run-time command which is interpreted by the receiving ARec component at run-time, it has nothing to do with the C++ source code.

Once the recogniser starts receiving packets from the coder, it will send recognition results back to the application. These are retrieved by extracting packets from the ansChan buffer. For example, the following loop would just print out the packets on the terminal device. Real applications might do something more sophisticated with the recognition results!

```
while(running) {
    APacket p = ansChan.GetPacket();
    p.Show();
}
```

Finally, ATK provides a monitor component which can be used to monitor and control the behaviour of the other components. If used this should be instantiated and started before starting any other components. The code to do this is straightforward

```
AMonitor amon;                  // create a monitor
amon.AddComponent(&ain);        // add the components to monitor
amon.AddComponent(&acode);
amon.AddComponent(&arec);
amon.Start();                   // then start it
```

As the above illustrates, a simple application of using ATK requires very little code to implement. For more realistic tasks, the main extension to the above code will often be to change the grammar prior to recognising each utterance. The simplest way to do this is to load multiple grammars and then switch them. For example, if an application required three grammars G1, G2, and G3, they would be instantiated as objects as in the example above, and each assigned to its own resource group

```
ResourceGroup *group1 = rman.NewGroup("group1");
group->AddHMMs(&hset); group->AddDict(&dict); group->AddGram(&gram1);
ResourceGroup *group2 = rman.NewGroup("group2");
group->AddHMMs(&hset); group->AddDict(&dict); group->AddGram(&gram2);
ResourceGroup *group1 = rman.NewGroup("group3");
group->AddHMMs(&hset); group->AddDict(&dict); group->AddGram(&gram3);
```

Notice that each resource group shares the same HMM set and dictionary but has its own unique grammar. Given the above, the recogniser's grammar can be switched by sending a command to it of the form

```
arec.SendMessage("usegrp(group2)");
```

when the recogniser receives this command, it takes no immediate action. However, next time that the recogniser is reprimed to start a new utterance, it will load the new grammar. This example illustrates the simplest form of grammar modification. ATK allows multiple grammar objects to be stored in a resource group and then combined in parallel. It also allows grammars to be copied, saved/restored, edited and created from scratch.

The above overview is intended to give a flavour of what ATK is, and how it is used. The remainder of this manual describes ATK in some detail. It starts with a description of the core classes on which ATK is built, packets, buffers and components. It then describes the main system components, audio source, coder and recogniser, followed by resources and the resource manager. There is then a section describing various issues relating to HTK such as code differences, usage conventions and resources supplied with ATK. Finally, a few example applications are described, the source code for which is included in the ATK distribution.

2 Packets

2.1 Generic Packet Properties

A packet is a chunk of information passed from component to component. It is polymorphic in the sense that its inner data container can hold a variety of types. Since a packet can have multiple readers, a reference count is also maintained for each packet instance to enable proper memory management. Packets can therefore be passed between asynchronous threads without the need for explicit garbage collection.

2.1.1 Programming

A Packet actually consists of 3 parts: an outer wrapper (APacket), an inner wrapper containing data-independent property information (APacketHeader) and an inner data-specific container (a derived class of APacketData). The outer class APacket is actually just a pointer to the inner wrapper and its methods are forwarded to the inner wrapper. The purpose of this arrangement is to ensure that packets can be passed around the system and properly disposed when no longer needed. Thus, no purpose is served by creating packets with new and referencing them indirectly by pointers. Rather packets should be instantiated as static variables and copied as needed.[7]

The APacket class (conceptually) provides the following data members

StartTime	Start time to which the data in this packet refers	r/w
EndTime	End time of which the data in this packet refers	r/w
Kind	Kind of this packet (Wave, Observation, Phrase, ….)	r
Data	Pointer to inner data packet	r

All packet properties XXX can be read via the methods GetXXX. Properties marked as r/w can also be written via the methods PutXXX.

A packet is created by first creating an inner data container of the required kind eg for storing a string, the following would be needed

```
AStringData *s = new AStringData(“A String”);
```

Then the packet variable itself can be declared giving the data packet as an argument to its constructor.

```
APacket p1(s);
```

Note that as mentioned above, although the inner data container is allocated using new, instantiating the actual packet via new should be avoided.

There are a variety of packet kinds and documentation for each follows in subsequent sections. There is also a special packet kind called AnyPacket. This is used in the context of packet filtering to indicate that any packet kind is allowed (see ABuffer below).

Once created, the packet may be manipulated in the usual way. For example, the start time of p1 could be printed by

```
printf(“Packet start time = %f\n”,p1.GetStartTime());
```

[7] Copying is very cheap since only a pointer is being copied.

When copied, the inner packet (both header and the data container) is shared. Thus, after

```
APacket p2 (p1);
```

there is just one copy of the string data with two packets referencing it. The same applies for assignment. Thus, after

```
p3 = p2;
```

the packets p2 and p3 (and p1) share the same inner packet and data container. This inner data will be disposed when all of p1, p2 and p3 expire.

For debugging purposes, the simplest way to view a packet is via the packet's Show() method.

2.2 Empty Packets

An empty packet contains no data at all. It is used occasionally for signalling purposes. The only operations provided for empty packets are the generic operations described above.

2.2.1 Programming

An empty packet is created by default when the packet declaration has no initialiser. For example, in

```
APacket e1;  APacket e2(e1)
```

Then e1 and e2 refer to a single empty packet. The packet's start and end time can be set as follows

```
e1.SetStartTime(10000); e1.SetEndTime(25000);
```

Note that internally, ATK uses HTK's 100ns time unit, hence 10000 in the above is equivalent to 1ms. Executing the e1.Show() method would result in output of the form

```
Empty Packet[007B1500] refs=2; time=(1.00-2.50) ms
Data: <>
```

The reference in square brackets is the packet's address; the refs value shows that the packet has two references and that it spans the period 1.00 to 2.5 ms. This first line printed by Show() is common to all packet types. The remainder of Show's output depends on the particular data stored, in this case none.

2.3 String Packets

A string packet contains a single string stored in a data container of type AStringData . It is used for general message passing and reporting.

2.3.1 Programming

The simplest and most common way to set up a string packet is to give the required string as an initialiser. When a packet is received, the data it contains can be accessed using the GetData method. Since this method must be coerced to the expected type, it is usually a good idea to check the packet kind first

```
APacket pkt = buffer.GetPacket();  // see Buffers section below
if (pkt.GetKind() == StringPacket){
    AStringData *sd = pkt.GetData();
    string s = sd->data;
    if (s.find("help") != string::npos)  {  // help message received
        .....
} else
    ....
```

2.4 *Command Packets*

Command packets are used to implement the component command interface. The data container of a command packet holds a command name plus an array of command arguments. Each argument is either a string, an integer or a float.

2.4.1 Programming

The inner container of a command packet is of type ACommandData and it is initialised with the command name. Command arguments can then be added using the overloaded AddArg method. For example, a packet containing the command "find(click,5)" would be created by

```
ACommandData *cd = new ACommandData("find");
cd->AddArg("click"); cd->AddArg(5);
APacket cmdPkt(cd);
```

On receipt, of a command packet, methods are provided to get the command name, the number of arguments, the type of the arguments and the arguments themselves. Hence, for example, the above packet received via a buffer might be decoded as follows

```
APacket p = inbuffer.GetPacket();
ACommandData *cd = (ACommandData)p.GetData();
string cmd = cd->GetCommand();
if (cd->NumArgs() != 2) error ...
if (cd->IsString(1) && cd->IsNumber(2)){
    string item = cd->GetString(1);
    int count = cd->GetInt(2);
} else
    error ....
```

In practice, command packets are rarely used by application programmers. They are provided primarily for the implementation of the SendMessage command interface to components.

2.5 *WaveData Packets*

The inner data container of a wave data packet is of type AWaveData and it contains an array of shorts called data for storing 16bit waveform data plus a count wused indicating how many samples in data are actually used. Both of these data members are public and can be accessed directly.

2.5.1 Programming

Given a packet p1, the contained wave data can be accessed as follows:

```
AWaveData * w = (AWaveData *)p1.GetData();
for (int i =1; i < w->wused; i++) printf("%d ",w->data[i]);
```

although an easier way to see the data is to use p1.Show() which might generate as output

```
Wave Packet [007A1910] refs=3; time=(200.00-208.50)ms
Data: 860/1000 samples
        0   85   90   94   93   98    9   55   59   42
       85   95   67   36   38   58   19   28   11   13
       72   98   65  -20  -64  -33  -70  -26  -25  -77
   ...
```

which shows that the packet with ref 007A1910 has 3 references. The time span of this packet is from 200.0 to 208.5ms. The packet contains waveform data, 860 sample slots of the 1000 sample packet are filled and the first few samples are as shown.

2.6 Observation Packets

An observation packet contains a single HTK format observation.

2.6.1 Programming

The inner data container of an observation packet is of type AObsData and it contains a single HTK format observation. No special methods are provided for manipulating an observation since it will normally be created and manipulated entirely by HTK. For example, the recogniser component ARec contains code similar to the following:

```
if (pkt.GetKind() == ObservationPacket){
   AObsData *od = (AObsData *)pkt.GetData();
   ...
   // recognise observation
   ProcessObservation(vri,&(od->data),-1);
   ...
}
```

This code extracts the inner AObsData structure and then passes its observation stored in the data field to the HTK HRec function ProcessObservation. Observation packets are provided primarily for implementation of the coder to recogniser interface, and application level programmers will rarely need to use them.

2.7 Phrase Packets

A phrase packet stores recognition result information generated by the recognition process. It is designed to allow complex data structures such as a chart to be passed from a recogniser back to the application “in pieces”. This provides flexibility and it allows an application to initiate its response before the utterance has finished and the final recognition result computed.

Each phrase packet contains one of the following seven types of information:

Start	indicates the start of the utterance
End	indicates the end of the utterance
OpenTag	the opening bracket of a tagged phrase, an OpenTag packet can also point to a packet indicating an alternative phrase hypothesis starting at approximately the same time
CloseTag	the matching closing tag to the OpenTag
Word	a single word, with optional semantic tag
Null	indicates a null node in the grammar, since null nodes can have semantic tags, null phrase packets should not necessarilty be ignored.
Quit	a special packet sent to indicate that the user has requested a “quit”

In addition to timing information, words and semantic tags, phrase packets can also carry a packet sequence number, the sequence number of its predecessor, the sequence number of an alternative packet, an acoustic score, a language model score, an overall score and a confidence measure in the range 0 to 1.

2.7.1 Programming

The data container of a phrase packet is of type APhraseData. A Phrase packet is initialised by giving the kind, the sequence number and the predecessor. The remaining fields are set by direct assignment as the following illustrates

```
// Create a word phrase packet
APhraseData *pd = new APhraseData(Word_PT,thisseqnum,predseqnum);
pd->ac = ac; pd->lm = lm;     // set recognition scores
pd->score = score;            // set total score
pd->confidence = confidence;  // set confidence level
pd->word = word;              // set identity of word
pd->tag = tag;                // set semantic tag if any
APacket p(pd);                // create the packet
p.SetStartTime(start);        // set times to show temporal span of
p.SetEndTime(end);            // the recognised word
```

Scores, confidence levels and HTK times are floating point numbers; words and tags are strings.

3 Buffers

Buffers provide the interface via which components send and receive packets. They are also used to implement the run-time command interface.

3.1 Programming

The basic methods provided by Buffer objects are:

Method	**Description**
PutPacket(pkt)	Put a packet into the buffer
pkt=GetPacket()	Get next packet from buffer
IsFull(), IsEmpty()	Booleans indicating full/empty state
NumPackets()	Returns number of packets in buffer
pkt=PeekPacket()	Like GetPacket, but leaves the packet in the buffer
PopPacket()	Like GetPacket but extracted packet is just discarded
GetFirstKind()	Return kind of next packet in buffer
SetFilter(PacketKind)	Restrict buffer to only accept packets of a given kind

When a buffer is created it is given a name and an optional size. If a size is given, then calls to PutPacket will block if the buffer is full. Calls to GetPacket will block if the buffer is empty. The integer function NumPackets and the Boolean functions IsFull and IsEmpty can be used to test the state of the buffer before committing to a call which might block. The GetFirstKind and PeekPacket methods allow the next packet in the buffer to be examined before actually removing it from the queue. PopPacket can then be used in place of GetPacket to simply discard the next packet in the buffer. By default a buffer will accept packets of arbitrary data kinds. Where a buffer should accept only one specific type of data packet, SetFilter can be used to enforce this.

The following illustrates the use of buffers. The code

```
ABuffer b(“mybuf”);
b.SetFilter(StringPacket);
```

creates a buffer suitable for passing strings from one task to another. A receiving task might extract string packets by

```
if (!b.IsEmpty()) {
    APacket p = b.GetPacket();
    AStringData sd = (AStringData *)p.GetData();
    string s = sd->data;
}
```

Here the packet contents can be safely cast to string data without checking because the buffer has been initialised to only accept string packets.

Because buffers provide the primary communication channel between asynchronous processes, it is commonly the case that reading packets from a buffer will be just one of several tasks that a receiver process will have to perform. For example, a component process may have several input buffers and it may wish to read the next packet received via any of them. With the methods described above, the only option for such a process is to poll all the buffers continuously until one is found to be non-empty.

However, polling is inefficient and hence ATK allows a process to receive an HTBUFFER event[8] whenever a packet is put into a buffer. For example, suppose that a process has two input buffers. It can request buffer events to be generated by calling the RequestBufferEvents(id) method where the id is an arbitrary integer which is chosen by the receiver process in order to identify which buffer generated the event. For example, the following code creates two buffers and requests events from them

```
const int eva = 1;
const int evb =2;
ABuffer a("abuf"), b("bbuf");
a. RequestBufferEvents(eva);
b. RequestBufferEvents(evb);
```

Now suppose that the receiver process receives an event using

```
e = HGetEvent(0,0);
```

it might handle it as follows

```
switch(e.event){
  case HTBUFFER:
     switch(e.c){
       case eva:  pkt =a.GetPacket();   ..... break;
       case evb:  pkt = b.GetPacket();  ..... break;
     }
  break;
  case ..... handle other events
```

Note however that since the sequencing of events is not always entirely reliable it would be good programming practice to check that a buffer for which there is a pending event really does have a packet in it.

4 Components

Unlike Packets and Buffers which support the direct creation of instances, the AComponent object is an abstract type from which actual components are derived. A further difference is that a component encapsulates an asynchronously executing thread rather than a passive data structure.

Figure 4 shows the structure of a typical component. Each component is normally initialised with a name and references to its input and output buffers.

Once any internal data structures have been initialised by the constructor, the Start method is called. When the calling thread has started up all its tasks, it will then typically wait for each to terminate by calling the Join() method.[9]

Once the component task is executing, it can receive data via its input buffer and output results to its output buffer. In addition, every component task monitors its message buffer and executes any pending commands. The actual commands supported are specific to each component except that all components support commands to suspend and resume execution, and to terminate.

[8] Events can be accessed via the platform independent HTK/HGraf mechanism or directly via platform specific OS event handlers.

[9] Threading in ATK is implemented by the HThreads module which is an ATK specific enhancement to HTK

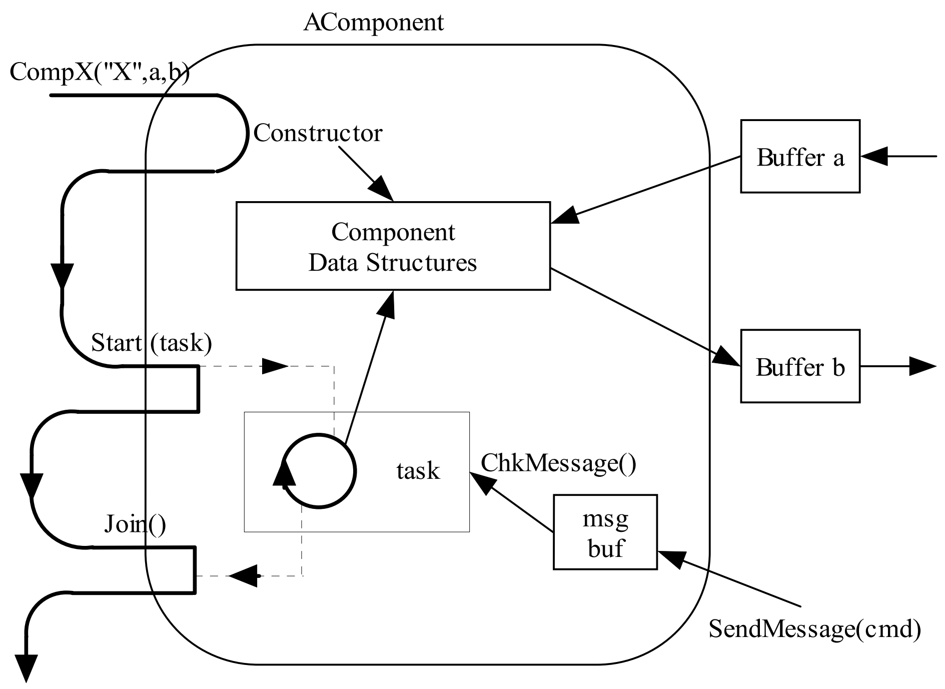


Figure 4 - Structure of a Component Object

4.1 Programming

The Start method of a component class has the following declaration:

```
void Start(HPriority pr, TASKTYPE (TASKMOD *task)(void *));
```

Its arguments specify a priority and a function to execute. Assuming that ATask is a component class derived from AComponent, the task itself should have the form

```
TASKTYPE TASKMOD ATask_Task(void *p)
{
   ATask *thisComp = (ATask *)p;
   ...
}
```

When this task is executed, a pointer to the class instance is passed as argument. Within the task, component members x can then be referenced via thisComp->x. To give the task unrestricted access to its associated class instance, it is usually declared as a friend of the class.

The Start method is usually called by the application's main program. When the latter has nothing further to do, it can call the component's Join method. This will suspend the calling thread until the component task terminates. Thus, in a multi-component system, the structure of main will typically be something like

```
ABuffer ab("ab");      // buffers to link a -> b
ABuffer bc("bc");      // etc
....
```

```
    ABuffer za(“za”);

    // Asynchronous Task Class
    class ATask : public AComponent {
        ATask(const string &name, ABuffer *inb, ABuffer *outb);
        ...
    private:
        friend TASKTYPE TASKMOD ATask_Task(void *p);
        ABuffer *in;
        ABuffer *out;
        ...
    };

    // The task itself, a friend of ATask
    TASKTYPE TASKMOD ATask_Task(void *p)
    {
        ATask *thisComp = (ATask *)p;

    }

    // delare some task instances
    ATask a(“a”, &ab, &za);
    ATask b(“b”, &bc, &ab);
    ....

    // start up the tasks
    a.Start(HPRIO_NORM,ATask_Task);
    b.Start(HPRIO_NORM,ATask_Task);
    ...

    // wait for them to finish
    a.Join();
    b.Join();
    ...
}
```

Tasks communicate in two ways. Firstly, they send data to each other via buffers. Thus, in the example above, each task has an input and an output buffer via which it will send and receive packets.

The second communication method is designed for run-time configuration and control. Each task derived from AComponent has a message buffer via which it can receive command packets containing commands of the form “cmdname(arg1,arg2,...)”. The AComponent class supports this command interface via the following methods

Method	**Description**
SendMessage(msg)	Send msg to task component (called by external task)
ChkMessage(wait)	Check for pending messages (called by internal task)
ExecCommand(msg)	Virtual function defined for each derived class

GetIntArg(int&,lo,hi)	Get next argument as an integer in range lo to hi
GetFltArg(float&,lo,hi)	Get next argument as a float in range lo to hi
GetStrArg(string&)	Get next argument as a string

The external interface is via the single method SendMessage. For example, a task can suspend some other task t by executing

```
t.SendMessage(“suspend()”);
```

All tasks support the standard commands “suspend()”, “resume()” and “terminate()”. All other commands are task specific and are provided by the task using the ExecCommand method. This commonly takes the form of a nested *if* statement as in

```
void ATask::ExecCommand(const string & cmdname)
{
    if (cmdname == "foo")
        FooCmd();
    else if (cmdname == "fum")
        FumCmd();
    else
        printf("Unknown command %s'\n",cmdname.c_str());
}
```

The ExecCommand is called by ChkMessage after first making sure that the command is well-formed and that the command is not one of the standard commands. The argument to ExecCommand is the command name itself. Inside the actual commands, the GetXXXArg routines can be used to retrieve arguments, these return false if an error occurs. For example, if FooCmd expects a single integer argument in the range 0 to 100, it would have the form

```
void ATask::FooCmd()
{
    int i;
    if (GetIntArg(i,0,100)) {
        // implement foo(i)
    }
}
```

In this case, the command does nothing if there is an argument error. An alternative would be to display a message on the Monitor message panel as explained in the next section.

5 The Monitor

The AMonitor class encapsulates a special system task which can be used to monitor all other tasks created via the AComponent class. Once an instance of the AMonitor class has been created, regular component tasks are added to it and then the monitor task is started.

Once started, the monitor task displays a window as illustrated in Figure 5. This contains a panel for each task added via AddComponent(). Each of these panels contains two coloured indicators, a command input box and a message output box.

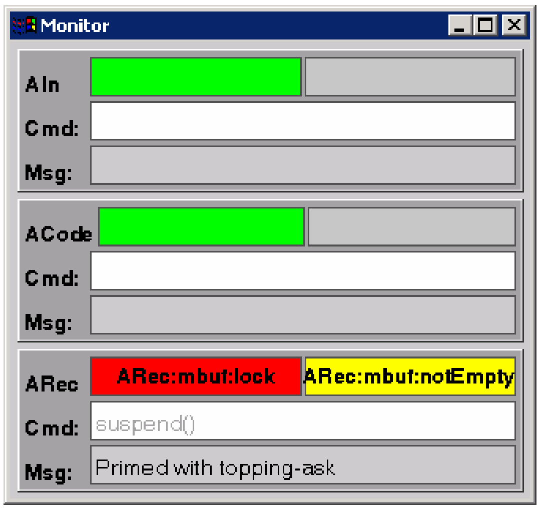


Figure 5 - The Monitor Window

The first (leftmost) indicator shows the running state of the task according to the following convention:

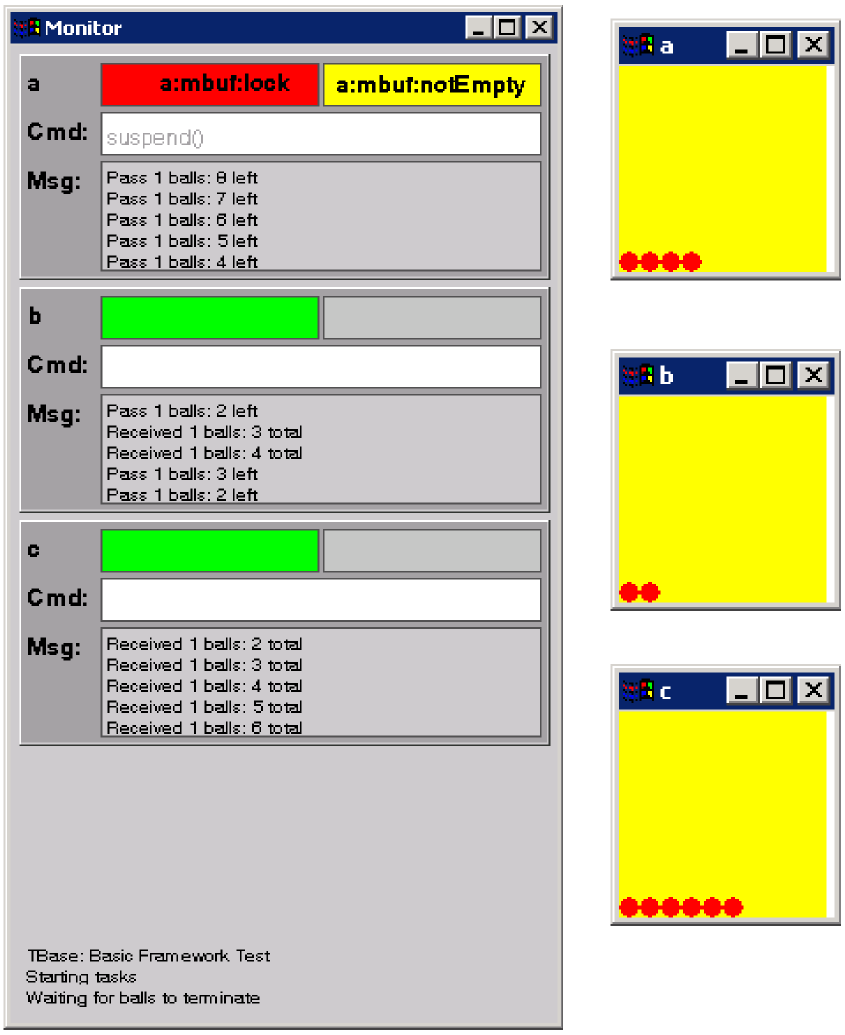
green	task executing
red	task is waiting for a lock whose name is given in the indicator
orange	task is executing inside a critical region
light grey	task is initialised
dark grey	task is terminated

The second indicator is normally light grey, when it goes yellow, it indicates that the task is waiting for a signal. Since a task must be suspended inside a critical region to wait on a signal, the signal indicator will only be yellow when the first indicator is red. As an example, in Figure 5, tasks AIn and ACode are executing but task ARec is in a critical region controlled by lock "ARec:mbuf:lock" waiting for signal "ARec:mbuf:NotEmpty".[10]

When a message is typed into the command box, it is passed to the corresponding task via the SendMessage interface. This facility allows a running system to be controlled at run-time. In Figure 5, task ARec has been suspended by typing in the command "suspend()". When text is being entered into a command box it appears in black, once the command is sent (by pressing the enter key), it is greyed out. The message output panel enables a running task to display a message. This is done by calling the HPostMessage function in the underlying HTK HThreads library.

[10] This actually indicates that ARec is waiting to receive a command from its input message buffer. This situation arises when a task has been suspended and the only action it can then take is to wait for a resume() command.

The monitor task can also be used to provide a set of virtual terminal devices for displaying the output of printf [11]statements. This is done by enlarging the message output panels to provide a display for each component, and adding a further display panel at the bottom for displaying terminal output from the main thread. For example, Figure 6 shows a monitor panel used in the TBase test program included in the ATKLib software directory (see sect ion 8.1). This simple program exercises the core ATK classes. It consists of 3 small windows holding a number of red balls. Each task is connected to another in a circle a->b->c->a. When a window is clicked, it passes a ball to its neighbour via the buffer. Every time a task passes or receives a ball it prints out a single line in its own message box simply by calling printf.



```
a    a:mbuf:lock    a:mbuf:notEmpty
Cmd: suspend()
Msg: Pass 1 balls: 8 left
     Pass 1 balls: 7 left
     Pass 1 balls: 6 left
     Pass 1 balls: 5 left
     Pass 1 balls: 4 left

b
Cmd:
Msg: Pass 1 balls: 2 left
     Received 1 balls: 3 total
     Received 1 balls: 4 total
     Pass 1 balls: 3 left
     Pass 1 balls: 2 left

c
Cmd:
Msg: Received 1 balls: 2 total
     Received 1 balls: 3 total
     Received 1 balls: 4 total
     Received 1 balls: 5 total
     Received 1 balls: 6 total

TBase: Basic Framework Test
Starting tasks
Waiting for balls to terminate
```

Figure 6 - Monitor display showing terminal output

5.1 Programming

Typical code for initialising and invoking the monitor is as follows:

```
AMonitor amon;
amon.AddComponent(&a);
amon.AddComponent(&b);
```

[11] Since ATK is so heavily dependent on HTK which is written in C, ATK itself uses C-like terminal i/o in preference to C++ streams.

```
   ... etc
 amon.Start();
```

The AMonitor class also provides a method called FindComponent(name). This will return a pointer to the component called name. This only works for components which have been added to the monitor via calls to AddComponent. If the component name is not found, FindComponent returns a null pointer. FindComponent is used mostly to locate a component object so that a run-time command can be sent to it.

To use the monitor for terminal output, the HTK subsystem is initialised using NCInitHTK("name of config file") rather than the standard InitHTK(argv,argc) (see sect 8.1). In addition, the number of display lines to be allocated for each component must be given as a second argument to the component constructor. For example, in the TBase program, the tasks are instances of an ABalls class which is itself derived from the AComponent class. The constructor for the ABalls class is as follows where the second argument to AComponent (i.e. 5) allocates 5 display lines to each task (the default is 1):

```
ABalls::ABalls(const string & name, int initBalls, ABuffer *inb, ABuffer *outb)
  : AComponent(name,5) {
    numBalls = initBalls;
    in = inb; out = outb;
    width = 100; height=100; size = 10;
}
```

5.2 Configuration Variables

The monitor display can be sized and positioned by setting the following configuration variables:

Module	**Variable**	**Default**	**Meaning**
AMONITOR	DISPXORIGIN	30	top-left x origin of display
AMONITOR	DISPYORIGIN	20	top-left y origin of display
AMONITOR	DISPWIDTH	460	width of a display panel

6 System Components

All system components are derived from AComponent and system component instances are normally registered with a monitor task. System components rely heavily on HTK libraries for their operation and much of their configuration is determined by a HTK-style configuration file which is read-in on start-up. This configuration file will contain definitions for both HTK and ATK variables. In the component descriptions which follow below, all relevant ATK variables are listed plus the most relevant HTK variables (standard HTK variables described in the HTKBook are indicated by asterisks). In addition to configuration variables, each component description also lists the runtime commands provided by the run-time SendMessage command interface. The commands available for each system component are also listed.

6.1 Audio Input (ASource)

The primary function of the audio input object ASource is to capture input speech. In basic operation, samples from the input audio device are grouped into wave packets and forwarded to the output buffer.

As shown in

Figure 7, the audio input object can optionally display a window showing a start/stop button and a simple volume meter.

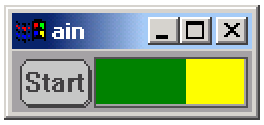


Figure 7 - ASource Volume Meter Control

Pressing the start button will start the source sampling. Pressing again will stop sampling. The name of the button changes between "Start" as in the figure and "Stop" to reflect the current state.

6.1.1 Configuration Variables

An audio object is configured by setting the following variables:

Module	Variable	Default	Meaning
	SOURCERATE	625*	set the sampling rate (in 100ns units)
ASource	DISPSHOW	T	show the volume meter (VM) control
ASource	DISPXORIGIN	30	initial position of VM (top left corner)
ASource	DISPYORIGIN	10	initial position of VM (top left corner)
ASource	DISPHEIGHT	30	initial height of VM
ASource	DISPWIDTH	120	initial width of VM
	SOURCEFORMAT	HAUDIO*	set to a file format (eg HTK) to switch source to a pre-recorded file.
ASource	WAVEFILE	""	define name of source wave file
ASource	WAVELIST	""	list of source wave files
ASource	FLUSHMARGIN	0	duration of zeros inserted on Stop Cmd
ASource	ZEROMEAN	T	zero mean the source waveform

Note that although class names are shown for configuration variable prefixes (e.g. ASource in the above), in practice, the instantiated object name should be used. For example, if the invoked source object is called "AIN", then a directive to set DISPWIDTH in the corresponding configuration file should be

```
AIN:   DISPWIDTH = 200
```

This use of instance names rather than class names (as in HTK) is to allow multiple instances of the same component to be individually configured. It applies to all configurable ATK components. Where a variable does not have a default value, the "default" column gives a recommended or typical setting and it is starred.

Normally, ASource is used to provide a real-time stream of sampled data from a microphone or similar audio source. However, to facilitate testing, ASource can be made to read from a single audio file by setting the configuration variable WAVEFILE. Alternatively, for evaluating performance on a database of test files. the configuration variable WAVELIST can used to identify a file containing a list of waveform files. In either case, the SOURCEFORMAT should be changed to indicate the file format (eg. HTK or NIST).

When ASource is reading from files, it must be sent a start command (see next section) prior to reading each file. Once started, ASource sends a string packet to the output containing the marker "AIN::START(filename)" and then sends the data for that file. When the end of the file is reached, the marker "AIN::STOP" is sent and ASource stops. When reading a list of files, the string marker packet "AIN::ENDOFLIST" is sent once all files have been transmitted to the output.

In either audio or file source mode, a period of silence (i.e. zero sequence) can be inserted by setting FLUSHMARGIN equal the required duration.

6.1.2 Run-Time Commands

In addition to the standard "suspend()", "resume()", and "terminate()" commands, the audio source object supports the following run-time commands:

Method	Description
Start()	Start input sampling (or start reading next input file)
Stop()	Stop input sampling

The effect of executing these commands is identical to pressing the VM start/stop button ie the audio capture is started and stopped, respectively. In addition, when the start command of an ASource component called "AIn" is executed, a string packet is sent to the output containing the marker "AIN::START". Similarly, when a stop command is executed, the marker "AIN::STOP" is sent. As noted above, when reading from a file, the name of the file is appended to the preceding start marker.

6.1.3 Programming

Initialising the audio source requires only that the constructor be given a name and a pointer to the output waveform packet buffer. For example, the following creates an audio source called "ain"

```
// Create Buffer
ABuffer auChan("auChan");

// Create Source
ASource ain("ain",&auChan);
```

Once created, the source is started in the normal way, ie.

```
ain.Start();
```

The sample rate of a source can be found using the method GetSampPeriod. This returns a HTK-style time (ie a floating point number representing time in 100ns units). ASource class has no other public methods.

In addition to the above, ASource provides two specialist constructors. Firstly, the volume indicator used in the stand-alone display window can be incorporated into another (HGraf) window using the constructor

```
ASource("asrc", &aubuf, win,cx0,cy0,cx1,cy1);
```

This will create an ASource object called "asrc" with output buffer aubuf. It will also create and display a volume control in the rectangle defined by cx0,cy0,cx1,cy1. Note that when using this form of the ASource constructor the configuration variable SHOWVM should be set false.

The second specialist constructor has the form

```
ASource("fsrc", &aubuf, "filelist");
```

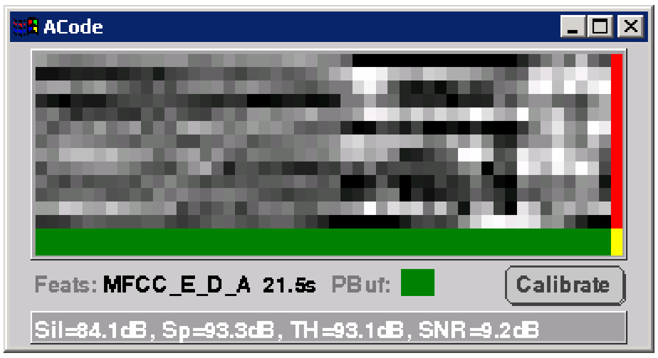
where "filelist" serves the same function as setting the WAVELIST configuration variable. It is provided to allow filelists to be supplied from the command line (see for example the AVite sample application in section 9.1.

6.2 Coder (ACode)

The function of an ACode component is to convert speech waveform data into feature vectors. ACode gathers incoming wave packets and passes them to the HTK module HParm to convert them to feature vectors. Thus, setting up ACode mostly involves setting up the relevant HParm configuration variables. ATK uses a special version of HParm which is functionally similar to the standard HTK module working in buffer mode. However, the implementation of silence detection is different and ATK's HParm also supports real-time cepstral mean normalisation. For the latter, a default cepstral mean vector μ_0 is loaded initially and then a running average cepstral mean is computed using

$$\mu' = \alpha\,(\mu - x) + x$$

where μ' is the updated cepstral mean, x is the input cepstral vector and α is a time constant.



Feats: MFCC_E_D_A 21.5s PBuf: 🟩

Sil=84.1dB, Sp=93.3dB, TH=93.1dB, SNR=9.2dB

Figure 8 - ACode FeatureGram Display

As illustrated by Figure 8, an ACode component can optionally generate a grey scale featuregram display.[12] This display scrolls from right to left during processing. Each vertical stripe represents the amplitude of each component in a single feature vector, indexed from the top downwards. The green/yellow bar at the bottom of the display indicates the speech/silence decision. A vertical green bar indicates the receipt of a start marker from the audio source, and a vertical red bar

[12] Note that the fonts used in ATK component displays may vary from those shown here.

indicates the receipt of a stop marker. Underneath the scrolling feature-gram display, the feature kind and the current time are displayed. Next to the time is shown the status of the HParm parameter buffer: normally this will show as either yellow to indicate that it is calibrating the silence detector or green to indicate that it is running normally. At the bottom of the display, the current silence detector settings are listed. The latter can be recomputed, by pressing the calibrate button.

6.2.1 Configuration Variables

The configuration variables available for setting up ACode are shown in the table below. The ACode specific variables are mostly concerned with specifying the size and location of the feature-gram display. By default all features will be displayed including any delta and delta-delta coefficients, the variable MAXFEATS can be used to restrict the display to the first few features. For example, in a conventional 39 feature MFCC_E_D_A type front-end, setting MAXFEATS to 13 would limit the display to just showing the static features. NUMSTREAMS is used to set the number of feature streams (usually 1).

CALWINDOW is used to set the number of speech frames to include in the speech/silence detector calibration window. This is the window over which a histogram of energy levels is computed and silence identified. SILDISCARD is an energy threshold below which speech frames are ignored. This is useful if the source can generate very low values (eg all zeros) which should not be mistaken for background silence. The actual speech/silence detection threshold is set by SPEECHTHRESH. SPCSEQCOUNT and SILSEQCOUNT determine the windows over which the signal must be designated as speech or silence respectively, in order to change the current speech/silence classification.

If the speech coding includes "_Z", then cepstral mean subtraction must be applied. To enable such a mean to be computed in real time, ACode first loads a default mean vector from the file named by CMNDEFAULT. This file must contain a cepstral mean vector (static coefficients only) of the form

```
<MEAN> 13
6.34 -4.25 2.01 ....
```

that is it consists of the label "<MEAN>" followed by the number of elements in the vector (13 in the example above) followed by the vector itself. In operation, this mean is updated every input frame according to the formula given above where the time constant is set by CMNTCONST. To prevent a mean being estimated from too little data, updating only starts once CMNMINFRAMES have been processed. Finally, when a STOP marker is received via the wave packet input buffer, it is sometimes appropriate (eg when testing on a set of pre-recorded speech files) to reset the cepstral mean vector back to the default and also reset the frame count. CMNRESETONSTOP enables this.

The remaining configuration variables in the table are all described in the HTK book.[13]

Module	**Variable**	**Default**	**Meaning**
	TARGETRATE	10000*	set the sampling rate (in 100ns units)
HPARM	TARGETKIND	MFCC_E_D_A_N*	set the coded feature kind
HPARM	ENORMALISE	FALSE*	dont attempt to normalise energy
HPARM	NUMCHANS	22*	number of filter bank channels
HPARM	NUMCEPS	12*	number of cepstral coefficients
HPARM	CALWINDOW	40*	num frames to calibrate spdet
HPARM	SPEECHTHRESH	10.0*	threshold relative to the estimated silence level above which signal is deemed to be speech

[13] But remember that ATK uses a special version of HParm which has more efficient buffer handling, a different silence detection/calibration strategy and running-average cepstral mean normalisation.

HPARM	SILDISCARD	10.0*	ignore frames with lower energy
HPARM	SPCSEQCOUNT	10*	num frames needed with energy above detector threshold in sequence in order to classify the incoming audio as speech
HPARM	SILSEQCOUNT	100*	num frames needed with energy below detector threshold in sequence in order to classify the incoming audio as silence
HPARM	SPCGLCHCOUNT	0*	when looking for SILSEQCOUNT frames with energy below threshold, max number of frames above threshold which can be ignored
HPARM	SILGLCHCOUNT	2*	when looking for SPCSEQCOUNT frames with energy above threshold, max number of frames below threshold which can be ignored
HPARM	CMNDEFAULT	""	name of file holding the default cepstral mean vector
HPARM	CMNMINFRAMES	10	minimum number of frames required before updating of the running average starts
HPARM	CMNTCONST	0.995	running average time constant
HPARM	CMNRESETONSTOP	T	reset the running cepstral mean to the default value, and set frame counter to 0 every time a STOP packet is received via input wave packet buffer
ACODE	NUMSTREAMS	1	number of streams
ACODE	DISPSHOW	T	enable featuregram display
ACODE	DISPXORIGIN	420	top-left x origin of FG display
ACODE	DISPYORIGIN	80	top-left y origin of FG display
ACODE	DISPHEIGHT	220	FG display height
ACODE	DISPWIDTH	400	FG display width
ACODE	MAXFEATS	1000	Max number of features to display

6.2.2 Run-Time Commands

In addition to the standard “suspend()”, “resume()”, and “terminate()” commands, the coder object supports the following run-time command:

Method	**Description**
calibrate()	Calibrate the speech/silence detector
resetcmn()	Reset the cepstral mean back to its default value

The effect of executing the "calibrate()" command is identical to pressing the FG Calibrate button. The "resetcmn()" command forces the running cepstral mean to be reset back to its default value i.e. it is the same as the coder receiving a STOP packet when CMNRESETONSTOP is set true.

6.2.3 Programming

Initialising a coder object requires only that the constructor be given a name and pointers to the input waveform packet buffer and the output observation packet buffer. All of the detailed configuration is determined from the configuration file.

For example, the following creates a coder object called "ACode"

```
// Create Input/Output Buffers
ABuffer auChan("auChan");
ABuffer feChan("feChan");
// Create a coder object called ACode
ACode acode("ACode",&auChan,&feChan);
```

Once created, the coder is started in the normal way, ie.

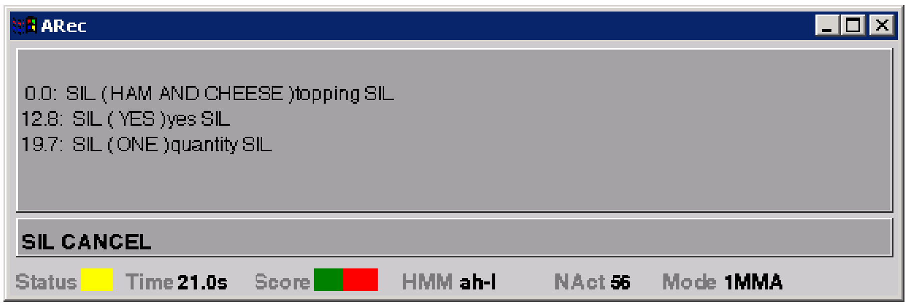
```
acode.Start();
```

The ACode class has no other methods for manipulating an ACode object. However, it does provide a method called GetSpecimen for providing a specimen observation data container (type AObsData). This method is used to enable the compatibility of potential consumers to be checked (see section 7.4)

6.3 Recogniser (ARec)

The ARec component provides similar functionality to the standard HTK Viterbi decoder[14]. It is supplied with a resource group containing the required HMMSet, dictionary, a grammar, and optionally an n-gram language model. It then decodes incoming feature vectors accordingly.

An ARec component can optionally generate a display of the form illustrated in Figure 9.[15] The main part of this display is a scrolling record of each recognised utterance prepended by the time it started. The smaller panel below the scrolling list displays the currently recognised utterance. This is updated in real-time as the recognition proceeds to show the current best hypothesis. Finally, at the bottom of the display are various status indicators: the recogniser status, the current time, a score indication, the identity of the best current HMM, the number of active models and the current operating mode.



ARec

```
0.0: SIL ( HAM AND CHEESE )topping SIL
12.8: SIL ( YES )yes SIL
19.7: SIL ( ONE )quantity SIL
```

SIL CANCEL

Status 🟨 Time 21.0s Score 🟩🟥 HMM ah-l NAct 56 Mode 1MMA

Figure 9 - ARec Recogniser Display

In operation, the recogniser is always in one of five possible states as indicated by the following table and the state diagram shown in Figure 10. The recogniser changes state, dependent on the settings of the operating modes listed in the following table. The ARec display shows the current mode as a sequence of 4 characters: representing the settings for CYCLE(1=oneshot, C=continuous), FLUSH

[14] The standard HTK decoder (ie HVite) is built on two main HTK library modules HNet and HRec. These latter two modules have been significantly extended in ATK to provide full tri-gram language model support and background model-based confidence scoring.

[15] Note that the fonts used in ATK component displays may vary.

(I=immed, M=tomark, S=tospeech), STOP(I=immed, M=tomark, S=tosilence), RESULTS (I=immed, A=asap, E=atend, X=all).

Status	Indicator	Meaning
WAIT	Red	Idle state
PRIME	Orange	Initialised but not running
FLUSH	Yellow	Flushing input packets
RUN	Green	Recognising
ANS	Mauve	Computing a recognition result

Mode	Options	Flag	Meaning
CYCLE	oneshot	0001	return to WAIT state after ANS state
	continuous	0002	return to PRIME state after ANS state
FLUSH	immed	0010	move immediately to RUN state from FLUSH state
	to mark	0020	discard input until START marker found, then move to RUN state
	to speech	0040	discard input until speech frame received, then move to RUN state
STOP	immed	0100	stop recognising and move immediately from RUN state to ANS state
	at mark	0200	stop recognising when STOP marker received and move to ANS state
	at silence	0400	stop when silence observation received and move to ANS state
RESULT	at end	1000	output full results when in ANS state
	immed	2000	output best guess in RUN state, this may be updated later
	asap	4000	output results in RUN state as soon as disambiguated
	all	7000	output as much information as possible

On creation, an ARec object is placed in the WAIT state. When in the WAIT state, the recogniser waits for a "Start()" command to be issued via its command interface. When this Start() command is received, the recogniser moves to the PRIME state in which it loads the recognition resources specified by the current resource group. It then moves immediately to the FLUSH state where it takes packets from its input buffer and discards them until it is ready to start recognising as determined by the flush mode setting. This can either be immediately, when a START marker is received or as soon as the incoming observation packet has a frame marked as speech. Once, in the RUN state, the recogniser recognises incoming packets until either a STOP marker is received, a speech frame is received which is marked as silence or a Stop() command is issued. In the ANS state, the recogniser cleans up the recognition processing and then returns to either the WAIT or PRIME states depending on the setting of the CYCLE mode.

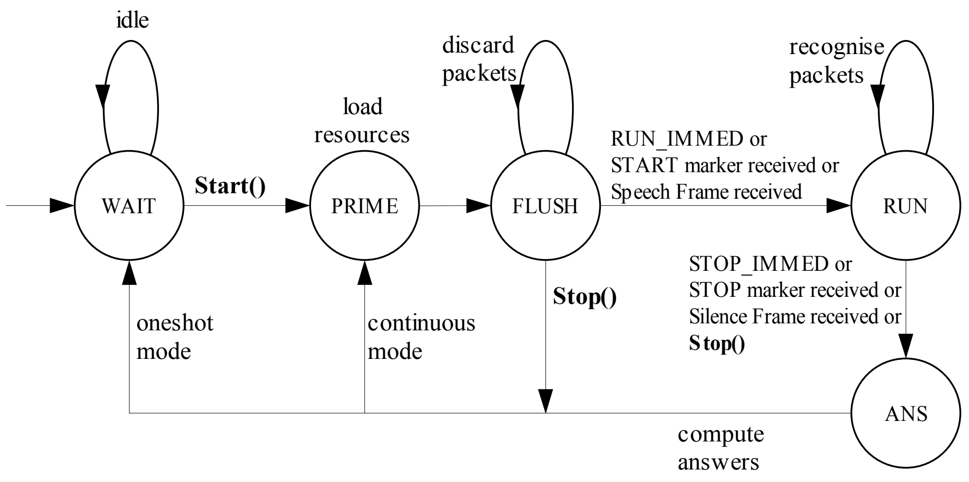

idle
WAIT
Start()
load resources
PRIME
discard packets
FLUSH
RUN_IMMED or START marker received or Speech Frame received
recognise packets
RUN
STOP_IMMED or STOP marker received or Silence Frame received or Stop()
Stop()
oneshot mode
continuous mode
ANS
compute answers

Figure 10 – ARec recogniser state transition diagram

The recogniser sends results to its output buffer in the form of phrase packets . As described earlier in section 2.7, phrase packets can contain start and stop markers, words, null nodes, open tag markers and close tag markers. The start and stop markers are generated automatically and the remainder depend on the recognition grammar (see section 7.3). Thus, a user input of "I want three pizzas please" might be returned by the recogniser output as a sequence of the 10 packets thus:

1	2	3	4	5	6	7	8	9	10
#	I	WANT	<eop>	{	THREE	}qty	PIZZAS	PLEASE	#
start	word	word	null	opentag	word	closetag	word	word	end

where the null node tagged with "<eop>" has been included to mark the end of the arbitrary preamble and the quantity "THREE" has been bracketed and marked with the semantic tag "qty". This output would be returned as 9 distinct phrase packets.

When in the RUN state and the asap RESULT flag is set, the recogniser performs partial traceback every few input frames (determined by the TRBAKFREQ configuration variable – see below). Result packets sent in this mode are guaranteed to be correct. When the recogniser moves to the ANS state, it computes the final recognition hypothesis and returns any remaining packets which have not already been sent. If the asap flag is not set, then all of the phrase packets are sent on completion with the one exception of the start packet which is always sent immediately the recogniser enters RUN mode. Thus, from the applications point of view, the sequence of phrase packets received back from the recogniser will be the same whether or not the asap flag is set. The only difference is that the early parts of the utterance may be returned before the speaker has stopped speaking when the asap flag is set. These early results are obtained at the cost of a small increase in computational load.

When the immed flag is set, then the recogniser returns its current best guess as it progresses through the utterance. In this case, the application may receive multiple phrase packets covering the same segment of the utterance since an early hypothesis which scores well initially is not guaranteed to be part of the final best global hypothesis. In this case, it is upto the application to decide how to interpret the information. In general, the immed mode is only useful for displaying progress information.[16]

6.3.1 Configuration Variables

The ARec configuration variables are listed in the following table. They are mostly concerned with setting the display dimensions and the recognition parameters as described in the documentation for HVite in the HTK Book. Otherwise, the RUNMODE variable is used to specify the initial run mode of

[16] For example, ARec uses the immed mode to drive its display.

the recogniser and the GRPNAME variable can be used to specify a default resource group. This is overriden by the usegrp() command described below in section 6.3.2. The TRBAKFREQ variable sets the frequency with which traceback for the asap and immed modes described above should be computed. Finally, the configuration variables CONFOFFSET, CONFSCALE, CONFBGHMM and CONFMEMSIZE are concerned with computing word confidence measures and these are described in the next section.

Module	**Variable**	**Default**	**Meaning**
AREC	SHOWDISP	T	display recognition processing and status
AREC	DISPXORIGIN	420	top-left X coord of RD Window
AREC	DISPYORIGIN	280	top-left Y coord of RD Window
AREC	DISPHEIGHT	120	height of display window
AREC	DISPWIDTH	400	width of display window
AREC	TRBAKFREQ	1	default traceback freq for RD
AREC	NTOKS	1	number of tokens (>1 for trigram language models and/or multiple hypotheses)
AREC	LMSCALE	1.0	grammar network scale factor
AREC	NGSCALE	0.0	n-gram language model scale factor
AREC	PRSCALE	1.0	pronunciation scale factor
AREC	WORDPEN	0.0	word insertion penalty
AREC	GENBEAM	225.0	general beam width
AREC	WORDBEAM	175.0	word beam width
AREC	NBEAM	225.0	n-best token beam width
AREC	MAXBEAM	0	max number of active nodes (0=disabled)
AREC	RUNMODE	01222	set run mode (atend+cont+tomark+atmark)
AREC	GRPNAME	“”	default resource group name
HNET	ALLOWCXTEXP	T*	Allow phone context expansion
HNET	ALLOWXWRDEXP	F*	Allow cross word triphone expansion
HNET	FORCECXTEXP	F*	Force phone context expansion
HREC	FORCEOUT	F*	force output even when no best path
HREC	CONFSCALE	1.0	confidence measure scale factor
HREC	CONFOFFSET	0.0	confidence measure offset factor
HREC	CONFBGHMM	""	background HMM for confidence measure
HREC	CONFMEMSIZE	2000	# frames to retain for confidence measure

6.3.2 Confidence Scoring

ARec supports a simple method of confidence scoring. Every frame, the acoustic log likelihood (ie acoustic score) of the best matching model state and the best matching background model state is saved. When a word is recognised, these *best-state* and *background state* scores are summed to form a best-possible-acoustic score *(bs)* and a background score *(bg)* over the segment of the waveform for which the word is being hypothesised. A raw confidence score in the range -1 to 1 is then computed as

$$rawconf = \frac{2(a - bg)}{bs - bg} - 1$$

where *a* is the actual acoustic log likelihood of the word. The confidence for that word is then computed as

$$conf = \frac{e^{x}}{e^{x} + e^{-x}}$$

where x is the scaled *rawconf* score

$$x = \alpha(rawconf - \beta)$$

The constant α sets the slope of the confidence curve and β sets the operating point. Their values are set by the configuration parameters CONFSCALE and CONFOFFSET, respectively, with default values of 1.0 and 0.0.

The background model is usually stored in a separate HMM definition whose name is determined by setting the configuration variable CONFBGHMM. Once loaded, this HMM is used to compute the background state probability. Note that the transition matrix of the background HMM is completely ignored in this process. Instead, the probability of the current speech vector is computed for each state of the background HMM, and the maximum log probability used as the background state score. If no background model is loaded, then the average score across all model states is used as a surrogate.

6.3.3 N-Gram Language Models

ATK can support N-gram language models. These must be read from a file (e.g. generated by the HLM toolkit) and stored as an n-gram recognition resource object using the ANGram class. When an instance of this class is included in a recognition resource group, the recogniser will give each new word hypothesis w a language model score (ie a log probability) computed as

$$\log P(link) \times LMScale \ + \ \log P(w \mid hist) \times NGSCale \ + \ WordPen$$

Here *P(link)* represents the network transition probability attached to the link from the previous node to current node and *P(w|hist)* represents the n-gram probability of the new word w given the preceding one or two words (depending on whether a bigram or trigram language model is used). Note that intervening null nodes are ignored by the n-gram model in computing word histories whereas the link probability is not. In general, *LMScale* is set to 0 when an n-gram language model is being used but there is no compulsion to do this.

6.3.4 Run-Time Commands

In addition to the standard “suspend()”, “resume()”, and “terminate()” commands, the ARec recognition object supports the following run-time commands:

Method	**Description**
start()	If the recogniser is in the WAIT state, it moves to the PRIME state. Otherwise the command is ignored
stop()	If the recogniser is in the FLUSH state, it is returned immediately to the WAIT or PRIME state depending on the CYCLE mode. If the recogniser is in the RUN state, it is moved immediately to the ANS state.
setmode(m)	The integer m is regarded as a set of flags with meanings as defined in the mode table above. This command can be issued at any time and it will have immediate effect. Note that if m has a leading 0, it is interpreted as

	an octal number.
usegrp(grpname)	Set the current resource group to grpname. This command can be issued at any time, however, it will have no effect until the next time that the recogniser moves to the PRIME state.

6.3.5 Programming

An ARec recognition component is instantiated with a name and pointers to an input buffer, an output buffer and a resource manager. It has no methods other than the usual Start() method. The following example shows the typical code used to instantiate and start a recogniser component

```
ARec arec("ARec",&feChan,&ansChan,&rman);
arec.Start();
```

The setting up and use of the resource manager is described in the next section.

7 Resources (ARMan)

The basic role and function of the resource manager ARMan was introduced in section 1. It maintains a database of recognition resources. An application can create resources (HMMs, dictionaries and grammars) and store them in the database. Once loaded, dictionary and grammar resources can be saved, edited and restored.

Each stored resource can be made a member of one or more resource groups. At any one time, the recogniser is using a HMM set and a network compiled from one specific resource group. If the resource group is changed or if any member of the group is edited, the network is recompiled and reloaded the next time the recogniser enters the PRIME state (see section 6.3).[17]

Figure 11 illustrates typical use of ARMan. In this example, one HMM set, two dictionaries and three grammars have been stored. These have then been assigned to two resource groups RG1 and RG2 with Gram1 and HMMs belonging to both. Initially, the recogniser component ARec loads a HMM set and a recognition network from resource group RG1. Later, a usegrp(RG2) command is sent to the recogniser. The next time that the recogniser enters the PRIME state, it will compile a new network representing RG2 and load it. If sometime later, the recogniser switches back to RG1, then it will reload the previously computed network. If however, the application has edited, say Gram2, then the recogniser will notice that Gram2 has been updated and it will recompile the network representing RG1 before reloading it.

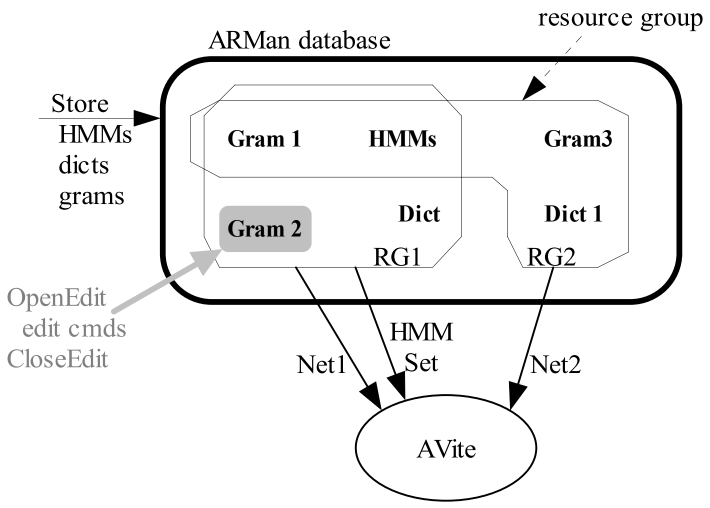


Figure 11 - ARMan Resource Database

7.1 Resources and Resource Groups

AResource is an abstract type from which concrete resources are derived. It provides a name for the resource, a version identifier and a resource lock. It has no methods.

A resource group is represented by the ResourceGroup class. It provides methods for adding resources to the group, and making a HMM set and a network from the group. When a resource group contains more than one dictionary, they will be concatenated together in the order supplied. Any duplicate entries replace existing ones. This allows a small application dictionary to be loaded on top of a larger system dictionary to provide application dependent pronunciations. When a resource group contains more than one grammar object, all of the grammar objects are combined together in parallel.

7.1.1 Configuration Variables

The ARMan configuration variables are listed in the following table.

[17] The HMM set is however loaded just once and cannot be changed.

Module	Variable	Default	Meaning
ARMAN	AUTOSIL	T	enable the automatic addition of a silence model at the start and end of the utterance
ARMAN	TRACE	0	set tracing

When the AUTOSIL variable is true, a silence model is automatically inserted at the start and end of the grammar. This is particularly useful when grammars are joined in parallel since it allows silence before and after an utterance without introducing a set of parallel competing silences.

7.1.2 Programming

A resource group is created by calling the NewGroup method of ARMan. For example,

```
ARMan rman;
ResourceGroup *rg1 = rman.NewGroup("RG1");
```

Once created, resources can be added to the group using the ResourceGroup methods AddXXX. For example, RG1 in the Figure 11 would be populated by

```
rg1->AddHMMs(&HMMs);
rg1->AddDict(&dict);
rg1->AddGram(&gram1); rg1->AddGram(&gram2);
```

where it is assumed that the individual resources had previously been created and added to rman, as in for example

```
ADict dict("dict");
rman.StoreDict(&dict);
```

Once a resource has been stored, then it can be found by name using the FindXXX methods. For example,

```
ADict  * d = rman.FindDict("dict");
```

would return a pointer to the dictionary object created above.

Once a group has been constituted, an ARec recogniser component can use it to create a HMM set and a network by calling the MakeHMMSet and MakeNetwork methods of the group. MakeHMMSet is called just once when the recogniser component starts. The first resource group created within ARMan is designated as the "main" group and this is the group which is used to create HMM sets. MakeNetwork is called every time that the recogniser enters the PRIME state. If there is no configuration variable GRPNAME defined for the recogniser, then by default, the main group will be used for network creation until a usegrp() run-time command is issued. The compiled networks are saved inside the object to avoid unnecessary recompilation.

When MakeNetwork is called, every component and the group itself is locked. Thus, if an application is currently editing a resource used by the currently active resource group, the recogniser itself will block. It is up to the application to minimise this effect by minimising the time it locks resources.

In addition to the NewGroup command, ARMan also provides methods for returning a pointer to the main group (MainGroup) and for finding a group by name (FindGroup).

7.2 Dictionary (ADict)

The ADict class is derived from the abstract Resource class and an instance of the ADict class is used to store a pronunciation dictionary. Logically a pronunciation dictionary can be viewed as a list of word entries where each word entry contains the orthography for the word and a list of pronunciations. Each pronunciation consists of a list of phones, a probability and an output symbol. The latter is optional but if present, recognition output will use the output symbol rather than the word itself.

A dictionary can be created empty and then filled via programmed actions, or more commonly it is loaded from an external file. In either case, a loaded dictionary can be edited by adding/deleting words and changing the pronunciations of existing words.

7.2.1 Configuration Variables

An ADict object supports the following configuration variables, as always, the module name ADICT in the following table should be replaced by the object's instance name

Module	Variable	Default	Meaning
ADICT	DICTFILE	“”	name of external dictionary file to load from
ADICT	TRACE	0	set tracing

7.2.2 Programming

Two constructors are provided for dictionary objects,

```
ADict dict(“name”);
```

This creates a dictionary object called “name”. If there is a DICTFILE configuration variable set for this instance, then the dictionary object will be loaded from the named file. Otherwise, an empty dictionary object will be created. Alternatively, an external load file can be named directly as in

```
ADict dict(“places”, “dicts/placenames”);
```

A dictionary consists of a set of WordEntry objects where each word entry consists of the word and one or more pronunciations. Each pronunciation is represented by a Pronunciation object which consists of a list of phones plus an output symbol and a probability. For example, the following code adds the word “either” to the dictionary dict. The word has two pronunciations which are created first, then a word entry object is created, passing the two pronunciations as arguments to the constructor. Finally, the newly created word entry is actually added to the dictionary by calling the UpdateWord method of the dict object:

```
Pronunciation p1(“either1",”iy dh ax”,0.6);
Pronunciation p2("either2",”ay dh ax”,0.4);
WordEntry we(“either”,p1,p2);
dict.OpenEdit();
dict.UpdateWord(we);
dict.CloseEdit();
```

Notice in the above that, as is the case for all resources, any modifications to a dictionary object must be bracketed by calls to OpenEdit and CloseEdit. These calls are necessary in order to prevent the recogniser asynchronously using the dictionary whilst it is being edited. As with editing any kind of resource object, the time during which a resource object is locked in this way should be minimised.

An existing word entry in the dictionary can be found using FindWord , modified if desired and then the underlying HTK entry updated by calling UpdateWord. A word can be deleted using RemoveWord. For example,

```
dict.OpenEdit();
WordEntry we = dict.FindWord("zebra");
dict.RemoveWord(we);
dict.CloseEdit();
```

If FindWord fails because the word is not in the dictionary, then the corresponding HTK Word type field "w" will be NULL. For example,

```
WordEntry we = dict.FindWord("zebra");
if (we.w ==  NULL) ... // "zebra" not in dictionary
```

For convenience any word can be tested for inclusion in the dictionary directly via the function HasWord. For example the following is identical to the above

```
if (!dict.HasWord("zebra") ... // "zebra" is not in the dictionary
```

The data fields of the principal types (ie Pronunciation and WordEntry) used to represent dictionary entries are public and can be manipulated directly. Apart from constructors and Show, these types have no other methods. It is important to stress that the ATK data structures are just a mirror of the underlying HTK data structures. If any changes are made to a word entry via ATK, then the underlying HTK dictionary entry for that word must be synchronised by calling the dictionary method UpdateWord.

7.3 Grammar (AGram)

A grammar object contains a grammar network defined in HTK SLF format. It consists of a set of nodes connected by arcs (ie links) to form a finite state network. Nodes can hold a word, a call to a sub-network, or they can be empty. The latter are called null nodes. Null nodes can be used to reduce the number of links in a network, they can also be used to mark a point in the network with a semantic tag.

All nodes can have an associated semantic tag. These are preserved by the recogniser and returned with the recognition results. They are particularly useful for tagging phrases and recording the identity of sub-networks which are called from different places for differing semantic purposes. For example, a sub-network of place names could be used in a "to place" and a "from place" context. If the first call is tagged with "toplace" and the second call is tagged with "fromplace" then the resulting recognition results would be of the form "TO (Manchester)toplace FROM (Liverpool)fromplace" avoiding the need for the application to re-parse the output.

7.3.1 Configuration Variables

An AGram object supports the following configuration variables, as always, the module name AGRAM in the following table should be replaced by the instance name

Module	Variable	Default	Meaning
AGRAM	GRAMFILE	""	name of external grammar file to load from
AGRAM	TRACE	0	set tracing

7.3.2 Programming

As in the case of dictionaries, two constructors are provided for grammar objects,

```
AGram gram(“name”);
```

creates a grammar object called “name”. If there is a GRAMFILE configuration variable set for this instance, then the grammar object will be loaded from the named file. Otherwise, an empty grammar object will be created. Alternatively, an external load file can be named directly as in

```
AGram gram(“name”,”grams/digits.net”);
```

An empty grammar file is also created if the second argument is an empty string.

Editing of grammar files follows the same model as for dictionaries, except that the associated types and methods are rather more complex. A grammar consists of a toplevel network plus zero or more sub-networks. In ATK both are represented by the same GramSubN type. Grammar editing consists of finding or creating the appropriate subnet and then modifying its nodes and arcs. For example, suppose the grammar file “digits.net” contained a single level digit grammar in which the ten digits zero to nine are placed in a loop. It is required to modify this grammar by replacing the word “zero” by a subnet tagged by “zero” and containing alternative ways of saying zero ie “zero”, “oh” and “nought”.

The first step is to open gram for editing and create a new sub-network

```
gram.OpenEdit();
GramSubN * zeros = gram.NewSubN("zeros");
```

The required words “zero”, “oh” and “nought” can now be added to this subnetwork along with null nodes for entry and exit using the NewXXXNode methods of the GramSubN class.

```
GramNode * zero = zeros->NewWordNode("zero");
GramNode * oh   = zeros->NewWordNode("oh");
GramNode * nought = zeros->NewWordNode("nought");
GramNode * start = zeros->NewNullNode(0);
GramNode * end = zeros->NewNullNode(1);
```

The subnetwork is completed by adding arcs to link the entry nodes to the words, and the words to the exit node. (

```
zeros->AddLink(start, zero, 0.33);     zeros->AddLink(start, oh, 0.33);
zeros->AddLink(start, nought, 0.33);  zeros->AddLink(zero, end, 1.00);
zeros->AddLink(oh, end, 1.00);         zeros->AddLink(nought, end, 1.00);
```

The next step is to get the main top-level network which is always stored in the main field of the grammar object instance, then find the word “zero” and its predecessor and successor nodes. For the latter, it is assumed that there is only one predecessor and one successor

```
GramSubN *main = gram.main;
GramNode *wzero = main.FindbyWord ("zero");
GramNode *s1 = wzero.pred.first();       // assume only one predecessor
GramNode *e1 = wzero.succ.first();      // and only one successor
```

Finally, the zero word node is deleted and replaced by the subnet tagged with zero

```
main.DeleteNode (wzero);
GramNode *subnet = main.NewCallNode (“zeros”,"zero");
main.AddLink(s1,subnet);   main.AddLink(subnet,e1);
```

Finally, close the grammar object for editing

```
gram.CloseEdit ();
```

The above illustrates the typical style of grammar operations and introduces some of the main methods. In addition to FindbyWord, methods are provided to find a call node by name (FindbyCall), a null node by its numeric id (FindbyNullId) and any node by its semantic tag (FindbyTag).

When a grammar is built from scratch, it is necessary to explicitly identify the start and end nodes. This is done using the SetEnds method of GramSubN. SetEnds can be called with explicit start and end nodes or it can be called with no arguments, in which case the start node is assumed to be the one with no predecessor and the end node is assumed to be the one with no successor. The integrity of a subnetwork can be checked using the IsBroken function which returns zero if the subnetwork is well-formed otherwise it returns a positive error code. Finally, a subnetwork can be normalised by calling the NormaliseProbs function. The application described in section 9.2 provides further examples of manipulating grammars.

7.4 N-Gram Language Model (ANGram)

An N-Gram language model is loaded from a file in standard ARPA format as generated by the HTK/HLM tools. Once loaded, an N-gram language model object cannot be further manipulated.

7.4.1 Configuration Variables

An ANGram object supports the following configuration variables, as always, the module name ANGRAM in the following table should be replaced by the instance name

Module	**Variable**	**Default**	**Meaning**
ANGRAM	NGRAMFILE	“”	name of external n-gram file to load from
HLM	RAWMITFORMAT	F	ignore HTK escaping conventions
HLM	UPPERCASELM	F	convert LM to uppercase on loading
ANGRAM	TRACE	0	set tracing

7.4.2 Programming

As in the case of grammar objects, two constructors are provided for loading n-gram language models,

```
ANGram ngram(“name”);
```

creates a n-gram language model object called “name” and loads an n-gram from the file specified by the NGRAMFILE configuration variable. This configuration variable must be set for this instance otherwise an error will be generated. Alternatively, an external load file can be specified directly as in

```
ANGram ngram(“name”,”mytrigram”);
```

Note that the default value for the n-gram language model scale factor (NGSCALE) is zero and must therefore be set explicitly in the configuration file.

7.5 HMMSet (AHmms)

An instance of the AHmms class is used to store a HTK HMMSet and it is a required resource of every HTK recogniser. Once created, a HMM set is currently a static object although one day, access to HTK’s adaptation facilities will be provided.

7.5.1 Configuration Variables

AHmms is supported by the following configuration variables

Module	**Variable**	**Default**	**Meaning**
AHMMS	HMMLIST	“hmmlist”	name of HMM list file
AHMMS	HMMDIR	“”	name of directory to look for HMMs
AHMMS	HMMEXT	“”	HMM file extension
AHMMS	MMF[0-9]	“”	name of one or more MMF files
HMODEL	ALLOWOTHERHMMS	True	allow mmfs to include hmm definitions not in hmmlist
AHMMS	TRACE	0	trace flag (see AHmms.cpp)

The AHmm configuration variables are the analogues of the corresponding command line arguments used in HTK tools. Where MMF files are specified, they must be assigned to variables MMF0, MMF1, etc in ascending order. The ALLOWOTHERHMMS configuration variable is used to allow or disallow model sets to be loaded for which the HMM list specifies only a subset.

7.5.2 Programming

A HMM set object is created simply by giving it a name. All of the detailed configuration is determined from the configuration file.
For example,

```
AHmms hset("myhmmset");
```

will create a HMM set configured according to the information given in the configuration file. The HMM set object is essentially read-only. Once loaded, its basic properties can be determined by calling one of AHmm’s GetXXX() methods, where XXX is

NumLogHMM	Number of logical HMMs
NumPhyHMM	Number of physical HMMs
NumStates	Total Number of HMM states
NumSharedStates	Total Number of shared states

NumMix	Number of mixture components
NumSharedMix	Number of shared mixture components
NumTransP	Number of distinct transition matrices
ParmKind	Parameter kind (should be same as for Coder)
Kind	HMM set kind (normally SHAREDHS)

Compatibility of the loaded HMM set can be tested using the CheckCompatible method. For example,

```
AObsData *od = acode.GetSpecimen();
if (hset.CheckCompatible(&(od->data))) .... // then ok
```

Here, specimen observation data is extracted from a coder object called acode and this is then checked for compatibility with the HMM set object hset.

There is also an alternative constructor for HMM sets which names the MMF files and the value of TRACE directly via parameters. This is used by AVite in order to allow these values to be set via command line parameters analogous to HVite.

8 Using ATK

This chapter draws together some of the main practical issues in using ATK. Most of the points covered here have been mentioned previously but not in full detail.

8.1 Initialisation and Error Reporting

The HTK libraries which underlie ATK must be initialised before any other processing can take place. As mentioned in section 5.1 there are two ways to do this depending on whether or not the application is run from a terminal window or not.

Traditionally HTK software expects a terminal window, hence the standard initialisation is performed by calling InitHTK, typically using the following incantation in the main program:

```
if (InitHTK(argc, argv,version)<SUCCESS){
        ReportErrors("Main",0); exit(-1);
}
```

With this method of initialisation, standard HTK command-line arguments can be passed to HShell. In particular, the required configuration file must be passed explicitly when using this initialisation, eg a typical invokation might be

```
atkprog -A -D -C config
```

When no terminal window is available, then HTK must be initialised as follows

```
if (NCInitHTK("config",version)<SUCCESS){
        ReportErrors("Main",0); exit(-1);
}
```

where in this case the configuration file name is given explicitly. This form causes all output generated by printf to be printed on the Monitor display panel instead of the terminal. Note also that when using this form, access to HShell's default command line parameter handling is lost.

Error handling throughout ATK/HTK uses the standard HShell function HRError in order to report errors. This error function simply buffers an error message and low level routines typically report an error via HRError and then return a failure code. At some point the higher level function reports an error via HRError and then it raises an exception.

In order to display the chain of error messages resulting from an ATK/HTK exception, the top level of every thread should be protected by an exception handler with catch statements for both ATK_Error and HTK_Error i.e. every thread main should have the form

```
try {
   // main thread code here
}
catch (ATK_Error e){ ReportErrors("ATK",e.i); }
catch (HTK_Error e){ ReportErrors("HTK",e.i); }
```

where the ReportErrors function outputs all pending HRError messages and then exits.

8.2 No-Console Mode

As noted in the last section, when HTK is initialised using the NCInitHTK function, all subsequent output from printf statements is displayed on the monitor panel instead of being sent to the terminal.

The way this works is that printf is renamed as ATKprintf via a global macro and ATKprintf sends all of its output to a buffer associated with the calling thread. The monitor process (if running) notices when these buffers are updated (via a dedicated event) and copies the contents to its display panels. Note that this only causes printf to be diverted in this way, if any code uses say fprintf(stdout, ...) or C++ streams, the output will be lost.

One obvious implication of this is that a monitor component must be invoked in order to see any diagnostic or error output from an ATK-based application running in no-console mode.

Although crude, use of this facility for non-console applications is strongly encouraged since there are many ways for HTK to fail at the initialisation stage, and without a diagnostic message, subsequent debugging of the set-up is virtually impossible.

8.3 Resource Incompatibility Issues

The representation of words in grammar files must match the representation of words in the dictionary exactly. ATK does not do any case translation or fuzzy matching when attempting to find the pronunciation for a word. If an N-Gram language model is also used, the representation of words in the LM must also match exactly.

8.4 ATKLib and Test Programs

This document is not (and never will be) complete. It is always assumed that ATK programmers will inspect the header files for the various ATK classes and occasionally inspect the implementations in order to understand some of the detailed semantics.

As well as all the header and source files, the ATKLib directory in the software distribution also contains a number of test programs which are good examples of simple ATK application programs. All of these programs should be run in the Test directory inside ATKLib which contains the necessary configuration files and resources.

The ATKLib test programs are:

TBase: this is a simple program which uses only the core ATK classes (ie APacket, ABuffer and AComponent). When executed it displays 3 boxes called a, b and c holding red balls, when you click on a box, one ball is moved to the next ball in the sequence. You can also type a command into the monitor command box to create n new balls ("new(n)") or pass n balls ("pass(n)"). Each box is in instance of ABalls which is a class derived from AComponent. The tasks are connected by buffers in a round-robin fashion and each ball is passed as a packet via the buffers. It is configured using the file TBase.cfg

TSource: this program creates a source component and reads its output buffer from main. The wave packets generated are stored and displayed on the terminal as they are received. When the source is stopped, the stored wave data is replayed (using HTK audio output). It is configured using the file TSource.cfg

TCode: this program creates a source component and connects it to a coder component. It is configured using the file TCode.cfg

TRec: this program creates a basic recogniser consisting of source, coder and recogniser. Packets output by the recogniser are simply printed out by the main thread. It is configured using the file TRec.cfg which loads resources stored locally in the Test directory. The grammar is a simple digit dialing task allowing one to say "Dial 123 456" etc.

8.5 Using ATK with Windows MFC

ATK can be used with Microsoft's Foundation Classes (MFC) and since MFC applications have no concept of a terminal, this is obviously a case where NCInitHTK must be used.

The major problem with integrating MFC and ATK is connecting the recogniser's output buffer to the users widget class (typically a CDialog class). The problem arises because MFC is entirely event driven and hence, recognition output can only be read by an event-driven handler. However, unfortunately thread events in MFC can only be handled by CWinThread classes (for the very good reason that MFC has no information as to which window to send the event to). This section offers some advice on how to deal with this issue - it assumes that the reader is familiar with MFC programming.

A strategy for using ATK in MFC applications is as follows. Assuming that a recogniser is created by:

```
arec = new ARec("ARec",feChan,ansChan,rman);
```

output from arec will be in the form of packets sent to ansChan. If the call

```
ansChan->RequestBufferEvents(ANSCHANNEL);
```

is made, then an event defined by

```
#define WM_HTBUFFER 9998
```

will be generated each time that a packet is put into ansChan. A handler for this event should be declared as follows:

```
afx_msg void OnATKBuffer(UINT id);
```

the body would then look something like

```
void OnATKBuffer(UINT id)
{
    // find the actual interface obj and ansChan buffer
    ATKinterface *self = theApp.atkintf;
    ABuffer *bp = self->ansChan;

    // read the pending packets
    for (int i=0; i<bp->NumPackets(); i++){
        APacket  p = bp->GetPacket();
        // process packet p
    }
    if (all_spoken_words_decoded){
        Execute_Spoken_Command();
    }
}
```

In order to get this handler to work properly, it must be in the CWinThread class hierarchy, eg in the main app (see below). The event is linked to the handler by placing the following in the appropriate message map:

```
ON_THREAD_MESSAGE(WM_HTBUFFER, OnATKBuffer)
```

All that remains now is to link this handler to the code in the user derived window class (derived from CWnd and therefore not in the CWinThread hierarchy) where the results can be processed. In MFC, class objects are instantiated dynamically at runtime using the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros. This means that it is not straightforward for one object to locate another. Normally, MFC programmers are not concerned with this since the mapping of the predefined message set handles all of this behind the scenes for them.

Although this is not Microsoft best practice, the simplest way to interface ATK to an MFC application is to follow these guidelines:

a) define the needed ATK variables (eg asource, acode, arec, rman, etc) inside the main application (i.e. directly inside theApp, perhaps wrapped in a class for convenience). For example, define a class something like the following:

```
#define ANSCHANNEL 1
class ATKinterface
{
public:
    ATKinterface();
    ~ATKinterface();
    // Information Channels (plumbing)
    ABuffer *auChan;     // carries audio from source to Coder
    ABuffer *feChan;     // carries feat vecs from Coder to Recogniser
    ABuffer *ansChan;   // carries answers from Rec back to Application
    // Active components (threads)
    ASource *ain;        // audio source
    ACode *acode;      // coder
    ARec *arec;          // viterbi recogniser
    AMonitor *amon;    // system monitor
    // Global resources (loaded from external files defined in config file)
    ARMan *rman;       // resource manager for dict, grammars and HMMSet
    AHmms *hset;        // HMM set is global since it never changes
    ADict *dict;            // ditto dictionary (though it can be edited if desired)
    AGram *ggrm;       // global default grammar
    // Operations
    //  TODO add ops to create context specific grammars,
    // each with its own group name

    void StartRecogniser(char * groupname);
    //  Load the specified group (ie grammar) and send start message to ARec

    // Message handler
    afx_msg void OnATKBuffer(UINT id);
    // this will be called everytime the recogniser puts a packet into ansChan
    // the id should be equal to ANSCHANNEL
};
```

and in the main application class include

```
    ATKinterface *atkinf;
```

and in theApp.InitInstance execute

```
    atkintf = new ATKinterface();
```

The constructor for ATKinterface can then allocate all of the objects and components as required. Since the class instance is in the main App, it can be referred to from any other class in the application by simply declaring theApp as an extern, and calling the required ATKinterface method directly, eg

```
    extern CMyApp theApp;
    theApp.atkintf->StartRecogniser("main");
```

This same direct route can be used by any widget wishing to call any ATKinterface method including for example inserting extra marker packets into the audio stream. Thus, a mouse click handler might encode the click position into a string packet "clickinfo" and call

```
    theApp.atkintf->auChan->PutPacket(clickinfo);
```

The final question then is how to implement Execute_Spoken_Command(); ie, once a spoken command has been decoded, how does the ATKinterface access a method in the appropriate CWnd or

CDialog class (CMyView, say)? There are several possibilities for this, depending on how ATK is being used and what style of multi-modal interactions are being programmed:

a) Use CMyView::OnFirstUpdate() to initialise a pointer to itself called viewp (ie viewp = this) in the ATKinterface. This pointer can then be used to access a method or post a message
b) Use a call-back routine, eg by passing a function pointer as a second argument to StartRecogniser.
c) Encode a pointer to CMyView in the packet inserted into auChan.
d) If the class name eg CMyView is known, use the RUNTIME_CLASS macros.

8.6 Tuning an ATK Recogniser

The real-time performance of an ATK-based recognition system depends on a variety of factors.:

a) type of acoustic model used, especially cross-word sensitivity and number of mixture components. Note that using more mixture components does not necessary incur a speed penalty since the extra precision can allow more aggressive pruning parameters.
b) type of grammar/language model constraints used. As with mixture components, the use of a trigram language model may not carry a significant speed penalty if it substantially reduces the task perplexity and allows more aggressive pruning
c) the number of tokens allowed per state. Multiple tokens are only needed when either a trigram language model is used or multiple hypotheses are required[18].
d) compilation optimisation.

However, as a guide, it should be possible in most cases to obtain real time performance or better on a 1GHz Pentium on tasks of around 1k word vocabulary, trigram language model and 6 mixture word internal acoustic models.[19]

To get the best performance for any given application the following steps are advised:

i. use an optimised version of the code (e.g. on Windows use the release rather than debug version)
ii. set the pruning thresholds to be as low as possible without causing excessive search errors. The default settings are a good place to start but do not rely on these.
iii. if multiple tokens are required, experiment with the number required for best performance. Between 4 and 8 should be adequate to give good trigram language decoding.

[18] Note that although HRec computes multiple hypotheses, ATK does not yet provide a specific mechanism for accessing them.

[19] Note that in order to retain flexibility and compatibility with HTK, ATK deliberately avoids many of the optimisations performed in commercial speech recognisers. In particular, it computes Gaussians exactly and makes no assumptions about the parameter kind. Also, it makes no assumptions about recognition network topology.

9 ATK Application Examples

9.1 AVite - an ATK-based version of HVite

AVite is designed for evaluating speech recognition performance on off-line test sets. It uses the standard ATK ASource, ACode and ARec components to set up a recognition processing pipeline. Its main purpose is to ensure that an ATK-based recognition system is giving similar recognition performance to the equivalent off-line HTK-based system.

AVite implements a subset of the functionality provided by HVite and for most purposes it is plug-compatible. The main limitations of AVite compared to HVite are that the input must be waveform data, only continuous-density HMMs are supported, forced alignments at the model and state level are not supported, adaptation is not supported and the output MLF formatting is limited to the HTK default format (ie start, end, label, score) and the options S (suppress score) and N (normalise scores per frame). In addition, AVite provides an output format flag F which causes the normal log probability scores to be replaced by confidence scores (see 6.3.2).

AVite is invoked from the command-line in an identical fashion to HVite, i.e.

```
AVite [options] VocabFile HMMList DataFiles...
```

where VocabFile is the name of a dictionary, HMMList is a list of HMMs and DataFiles are the speech files to test, the latter are normally listed in a .scp file input via the -S option. The supported options are

-i s	output transcriptions to MLF s
-g s	load n-gram language model from file s
-o s	output label formatting (F=conf scores, S=suppress, N=normalise)
-p f	set inter-word transition penalty to f
-q f	n-gram language model scale factor
-r f	pronunciation scale factor
-s f	set link grammar scale factor to f
-t f	set genereral pruning threshold to f
-u i	set the maximum number of active models to i
-v f	set word end pruning threshold to f
-w s	recognise from network defined in file s (recognition mode)
-A	print command line args
-C cf	load config file from cf
-D	display configuration variables
-H mmf	load hmm macro file mmf (NB at most 2 mmf files can be loaded)
-I mlf	load master label file (alignment mode)
-S f	set script file (.scp) to f
-T N	set tracing to N

For normal operation, a recognition network must be supplied via the –w option. If the –w option is omitted and an mlf is supplied via the –I option, then AVite operates in alignment mode. Each speech file must then have an entry in the mlf. Prior to recognising the file, this entry is used to construct a linear recognition network consisting of the words in sequence. Thus, every file is recognised using its own recognition network and that network forces a word level alignment.

In the absence of an N-gram language model, the results computed by AVite should be very similar to those computed by HVite but they will not be identical for the following reasons:

a) AVite uses a special version of HParm which has no "table mode". When HParm computes delta and delta-delta coefficients over a fixed utterance, it uses a modified formula for the final few frames. Since AVite only works in "buffer mode", it uses an alternative strategy of padding the signal with silence.

b) If the target parameterisation includes "_Z" then cepstral mean normalisation is required. HVite calculates the cepstral mean on a per utterance basis by computing the mean over the whole utterance and then subtracting it from every vector. AVite uses running average cepstral mean normalisation and resets the running-mean back to the default at the start of every input utterance. Some adjustment of the time constant set by CMNTCONST may be necessary to get satisfactory performance.

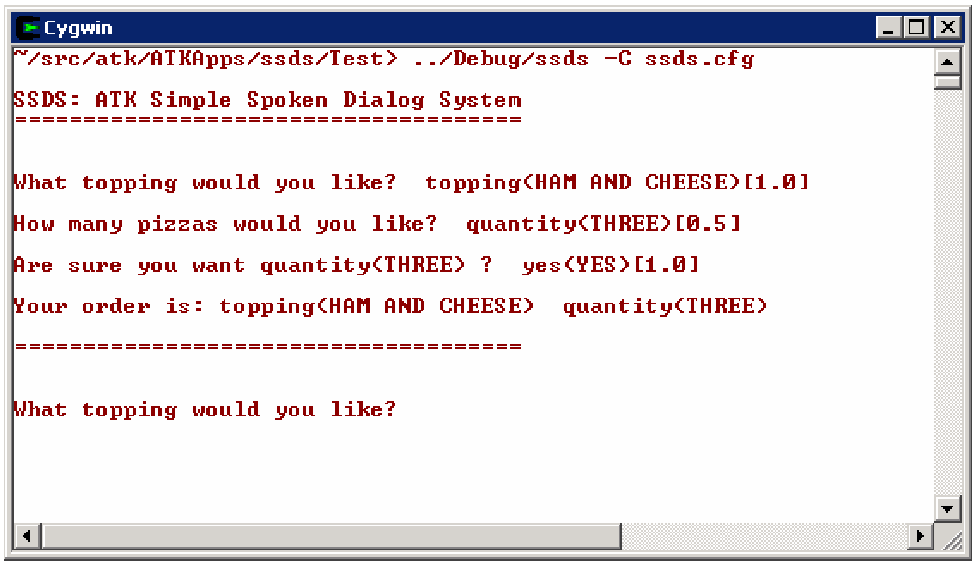
c) If an n-gram language model is used, then there is no direct comparison with HTK since the HTK decoder does not support n-grams. Using a bi-gram language model with a word loop in ATK should give similar results to using a bigram network and no language model in both HTK and ATK.

d) HTK uses a word-pair approximation when computing multiple hypotheses (i.e. when NTOKS > 1). ATK computes multiple hypotheses using a trigram approximation.

9.2 Simple Spoken Dialogue System (SSDS)

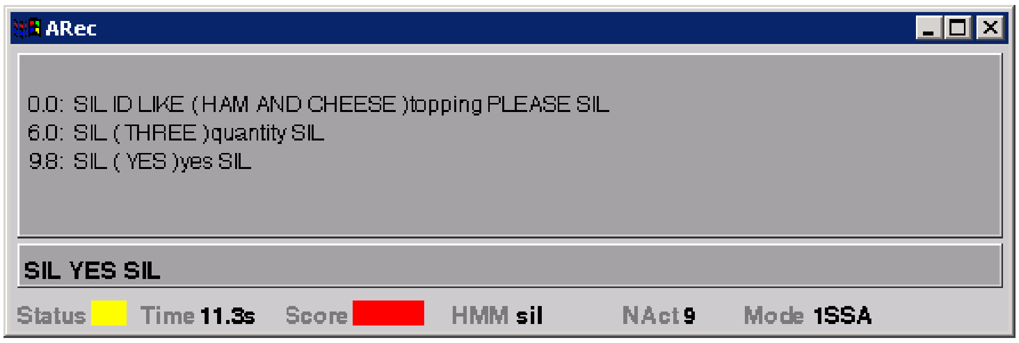
SSDS is a simple demonstration dialogue system in the scenario of ordering a pizza over the telephone. It is provided as an example of how user interations can be programmed in ATK and as an example of grammar manipulation.

SSDS is invoked using the configuration file called ssds.cfg as shown in the screen shot of Figure 12. On starting, SSDS will display the standard ATK display windows. Press the start button on the audio source volume meter and start answering the questions as shown in the screen shot. SSDS's understanding of each response is printed after each question along with the confidence score. If the latter falls below a threshold then a confirmation is requested, otherwise the system proceeds to ask the next question. When both the type of pizza and the quantity are determined, the required order is displayed and the cycle repeats. When confirming, the user can either say "Yes" or "No", or make a correction such as "No, I want two pizzas". Also, at any time the user can say "Cancel" to start over, "Repeat" to request that the question be repeated or "Help" to get help. The screen shot in Figure 12 and the corresponding recogniser display window in Figure 13 show a simple dialogue to order three Ham and Cheese pizzas.



```
~/src/atk/ATKApps/ssds/Test> ../Debug/ssds -C ssds.cfg

SSDS: ATK Simple Spoken Dialog System
=====================================

What topping would you like?  topping(HAM AND CHEESE)[1.0]

How many pizzas would you like?  quantity(THREE)[0.5]

Are sure you want quantity(THREE) ?  yes(YES)[1.0]

Your order is: topping(HAM AND CHEESE)  quantity(THREE)

=====================================

What topping would you like?
```

Figure 12 Terminal window showing execution of the Simple SDS demo program



```
0.0:  SIL ID LIKE ( HAM AND CHEESE )topping PLEASE SIL
6.0:  SIL ( THREE )quantity SIL
9.8:  SIL ( YES )yes SIL

SIL YES SIL

Status      Time 11.3s    Score        HMM sil        NAct 9      Mode 1SSA
```

Figure 13 - Recogniser display corresponding to terminal window snapshot

A key part of this demonstration program is the way that the recognition grammars are constructed. The underlying dialogue representation consists of a sequence of slots. Each slot has a value and a status of {unknown, unconfirmed, grounded, cancelled}. In the program, each slot is an instance of a QA class. The top level control consists of executing the GetSlot method of each QA instance in turn. The GetSlot method uses an Ask method to ask for a value and a Check method to confirm and possibly correct the current slot value. Each QA instance is provided with a prompt, a help string and a grammar when created. The prompt is output when the Ask method is called and the grammar is used to recognise the result. In addition, a global grammar containing the "Help/Cancel/Repeat" commands is placed in parallel. As an example, Figure 14 shows the contents of the global grammar file global.net and Figure 15 shows the slot specific grammar topping.net for determining the required type of pizza. Note that the topping grammar uses standard preamble and postamble subnetworks defined in global.net.

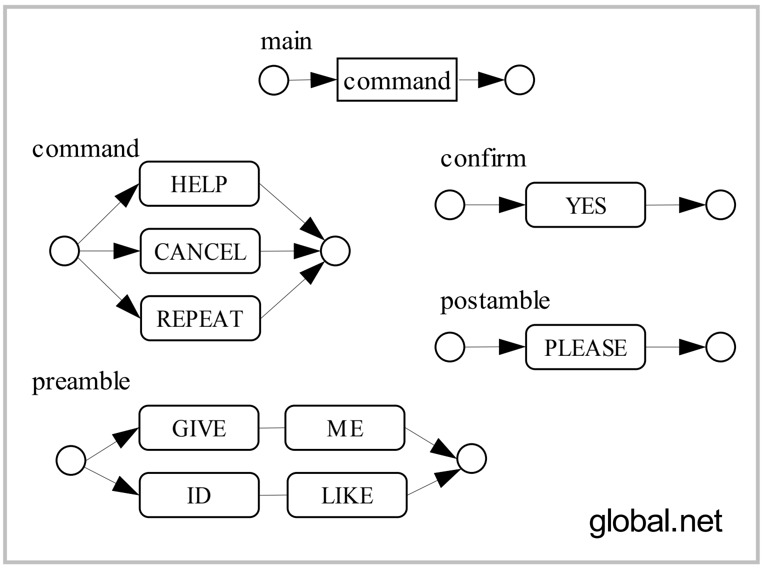


Figure 14 - global.net grammar file

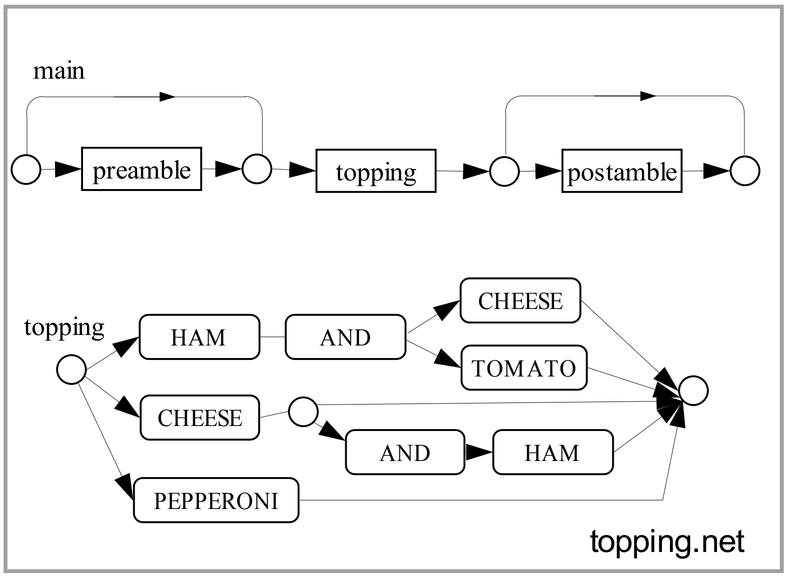


Figure 15 - topping.net grammar file

The grammar for confirmation and correction is synthesised within the program by inserting the word "No" before the slot specific grammar and then placing the confirm grammar in parallel. The result is the grammar network shown in Figure 16. Inspection of the constructor code QA::QA in ssds.cpp will reveal that this synthesis requires just a few lines of code.

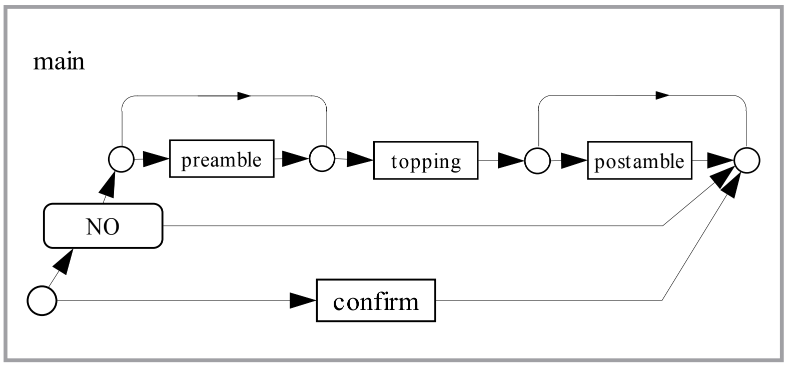


Figure 16 - Synthesised confirmation and correction grammar

Finally, interpretation of the recognition results depends on identifying the semantic tags corresponding to the subnetwork grammars and is straightforward.

Index

ACode, 4, 26
  configuration variables, 27
ACommandData, 13
AComponent, 17, 18, 19, 24
AddArg, 13
AddComponent, 21
AddDict, 36
AddGram, 36
AddHMMs, 36
AddLink, 39
ADict, 4, 37
AGram
  configuration variables, 38, 40
  programming, 39
AHmms, 4, 41
  configuration variables, 41
ALLOWOTHERHMMS, 41
AMonitor, 21
ANGram, 4
  programming, 40
ANS, 30
AnyPacket, 11
AObsData, 14
APacket, 11
APacketData, 11
APacketHeader, 11
APhraseData, 15
ARec, 4, 14, 29, 35
  configuration variables, 31
  programming, 34
  state transition diagram, 30
AResource, 4, 35
ARMan, 4, 35
  configuration variables, 35
  example of use, 35
  programming, 36
  resource database, 35
ASource, 4, 24
AStringData, 12
ATask, 18
ATK compiler directive, 4
ATKinterface, 46
ATKprintf, 44
audio capture, 24
audio input, 24
audio source
  programming, 25
  run-time commands, 25
AUTOSIL, 36
AVite, 48
AWaveData, 13
background HMM, 33
best-state, 32
buffer, 4, 12
  blocking, 16
  input/output, 19
  message, **19**
  name, 16
  size, 16
  waveform packet, 25
calibrate(), 28
CALWINDOW, 27
cepstral mean normalisation, 26
  time constant, 26
CheckCompatible, 42
ChkMessage, 20
class names
  in config files, 25
CloseEdit, 37, 40
CMNDEFAULT, 28
CMNMINFRAMES, 28
CMNRESETONSTOP, 28
CMNTCONST, 28
coder
  programming, 29
  run-time command, 28
command
  run-time, 9
command interface, 16
component, 4, 17
  monitor, 9
CONFBGHMM, 33
confidence, 15
confidence scoring, 32
config file, 24
  example of, 7
configuration variables
  audio source, 24
  coder, 27
  grammar objects, 38, 40

HMM objects, 41
HParm, 27
monitor, 23
CONFOFFSET, 33
CWinThread, 45
CYCLE, 30
DICTFILE, 37
dictionary, 4, 35, 37
case issues, 44
modification of, 38
display
featuregram, 26
DLWIDTH, 23
empty grammar, 39
ENORMALISE, 27
error reporting, 43
exception, 43
ExecCommand, 20
featuregram display, 26
FG Calibrate button, 28
FGHEIGHT, 28
FGWIDTH, 28
FGXORIGIN, 28
FGYORIGIN, 28
FindbyCall, 40
FindbyNullId, 40
FindbyTag, 40
FindbyWord, 39
FindComponent, 23
FindDict, 36
FindGram, 36
FindGroup, 36
FindHMMs, 36
FindWord, 38
FLUSH, 30
FLUSHMARGIN, 24
FORCEOUT, 32
GENBEAM, 32
GetCommand, 13
GetFirstKind, 16
GetFltArg, 20
GetInt, 13
GetIntArg, 20
GetPacket, 16
GetSampPeriod, 26
GetSpecimen, 29
GetStrArg, 20
GetString, 13
GRAMFILE, 38, 39
grammar, 4, 35, 38
digit loop, 6
example networks, 51
examples of manipulation, 50
file editing, 39
operations, 9
GramSubN, 39
GRPNAME, 32, 36
HasWord, 38
HMM set, 4, 35, 41
compatibility of, 42
HMMDIR, 41
HMMEXT, 41
HMMLIST, 41
HMMSet, 41
HPostMessage, 21
HRError, 43
HTK, 4
HGraf, 4
HThreads, 4, 17, 21
libraries, 4, 5, 24
observation, 14
SLF format, 38
HVite. *See* AVite
InitHTK, 23
initialisation, 43
IsBroken, 40
IsEmpty, 16
IsFull, 16
IsNumber, 13
IsString, 13
Join, 17, 18
Kind, 41
LMSCALE, 32
lock, **21**
main program, 18
MainGroup, 36
MakeHMMSet, 36
MakeNetwork, 36
MAXBEAM, 32
MAXFGFEATS, 28
message buffer, 19
MFC, 44
Microsoft MFC, 44
MMF, 41
monitor, 9, 21, 24
display, **21**

indicator, 21
multicomponent system
structure of, 18
NBEAM, 32
NCInitHTK, 23, 43
network, 5, 35
NewCallNode, 40
NewGroup, 36
NewNullNode, 39
NewSubNet, 39
NewWordNode, 39
N-Gram, 4
n-gram language models, 33
NGRAMFILE, 40, 41
NGSCALE, 32
no terminal mode, 43
no-console mode, 22
NormaliseProbs, 40
NTOKS, 32
null nodes, 38
NUMCEPS, 27
NUMCHANS, 27
NumLogHMM, 41
NumMix, 41
NumPackets, 16
NumPhyHMM, 41
NumSharedMix, 41
NumSharedStates, 41
NumStates, 41
NUMSTREAMS, 28
OpenEdit, 37, 39
OpenTag, 14
packet, 4
answer, 6
assignment, 12
command, 13
empty, 12
EndTime, 11
filtering, 11
generic, 11
kind, 11, 12
lifetime, 12
memory management, 11
observation, 14
phrase, 14, 31
programming, 11
properties, 11
reference count, 11
Show, 12
StartTime, 11
string, 12, 16
wave data, 13
ParmKind, 41
PeekPacket, 16
performance testing, 48
performance tuning, 47
phrase
CloseTag, 14
End, 14
Null, 14
OpenTag, 14
Quit, 14
Start, 14
Word, 14
phrase packets
examples of, 31
PopPacket, 16
PRIME, 30, 35
printf
in no-console mode, 22
in no-terminal mode, 44
ProcessObservation, 14
Pronunciation, 37
PRSCALE, 32
PutPacket, 16
RAWMITFORMAT, 40
RDHEIGHT, 32
RDWIDTH, 32
RDXORIGIN, 32
RDYORIGIN, 32
recogniser
asap flag, 31
configuration variables, 31
display, 29
mode, 5, 29
results, 5, 31
run-time commands, 33
state, 29
RemoveNode, 40
RemoveWord, 38
ReportErrors, 43
resetcmn(), 28
resource
database, 4
group, 5
resource group, 35

switching, 9
resource groups, 35
resource lock, 35
resource manager, 35
configuration variables, 35
ResourceGroup, 35
resources, 4
RESULT, 30
resume(), 20
RUN, 30
RUNMODE, 32
runtime command
usegrp(), 32
run-time command
usegrp, 9
run-time commands
recogniser, 33
runtime commands, **20**
run-time commands
audio source, 25
SALIGN, 32
sample period
of audio source, 26
score
acoustic, 15
language model, 15
overall, 15
semantic tag, 4, 38
SendMessage, 20, 21, 24
SetEnds, 40
SetFilter, 16
setmode(), 33
SHOWFG, 28
SHOWRD, 32
SHOWVM, 24
signal
wait for, **21**
SILDISCARD, 28
silence padding, 25
SILGLCHCOUNT, 28
SILSEQCOUNT, 28
simple dialogue system, 50
SLF format, 38
SOURCEFORMAT, 24
SOURCERATE, 24
SPCGLCHCOUNT, 28
SPCSEQCOUNT, 28
speech/silence detection, 27
SPEECHTHRESH, 27
SSDS, 50
Start, 17, 18
start(), 33
start/stop button, 24
statistical language models, 33
STOP, 30
stop(), 33
sub-network, 4, 38
suspend(), 20
TARGETKIND, 27
TARGETRATE, 27
task
running state, 21
signal state, 21
terminal mode, 43
terminal window, 43
terminate(), 20
Test programs
TBase, 22
thread, 4, 11, 17
time
units, 12
TRACE, 36, 37, 38, 40, 41
TRBAKFREQ, 32
tri-grams, 33
Unix, 4
UpdateWord, 38
UpdateWord, 37
UPPERCASELM, 40
usegrp(), 9, 33, 35, 36
using ATK, 43
without a console, 43
version information, 8
virtual terminal, 22
VMHEIGHT, 24
VMWIDTH, 24
VMXORIGIN, 24
VMYORIGIN, 24
volume meter, 24
including in another window, 26
WAIT, 30
WAVEFILE, 24
waveform, 13
WAVELIST, 24
Windows, 4
WORDBEAM, 32
WordEntry, 37

WORDPEN, 32
XORIGIN, 23
YORIGIN, 23
ZEROMEAN, 24