TR-IT-0201

Application interfaces for the Chatr synthesiser

Pao Chen (George) Hwang & Nick Campbell

1996.12

ABSTRACT
In this report, we present three application interfaces for the ATR Chatr synthesiser implemented by George Hwang during his student internship from the University of British Columbia. The first is an all-purpose interface, PTerm, using stdin/stdout so that existing software like TDMT can be easily interfaced. The second describes three applications of dynamic time-warping to match Chatr input/output with real-world audio, and the third is a multi-lingual text processor that allows mixed text (from different languages and character sets) to be synthesised using a single speaker.

Application Interfaces for Chatr

Pao-Chen (George) Hwang
Nick Campbell
ATR ITL Dept. II

December 27, 1996

Contents

1 Introduction

This report is based on my work at ATR-ITL as a student intern from the University of British Columbia between June 1996 and December 1996, under the supervision of Nick Campbell. It includes 3 major sections: a) development of PTerm and its application, b) prosody and CHATR and c) an auto-language detector and parser.

A method of interfacing the speech syntheiser CHATR with other programs has been developed during my stay at ATR. The program PTERM, using stdin/stdout to connect existing speech tools at ATR, can interface CHATR with existing software such as TDMT or the JANUS speech recognizer.

The ATR ITL speech synthesizer, CHATR, is still under development. The importance of prosody in correct information transferring is well known. This paper illustrates the importance of prosody to intended meaning, and presents experimental results from one of Nick Campbell's experiments which show that prosody is essential for a concise and detailed information transfer in the proposed ATR-ITL spoken language translation system. Also it proposes several interesting applications for CHATR using dynamic time warping technique.

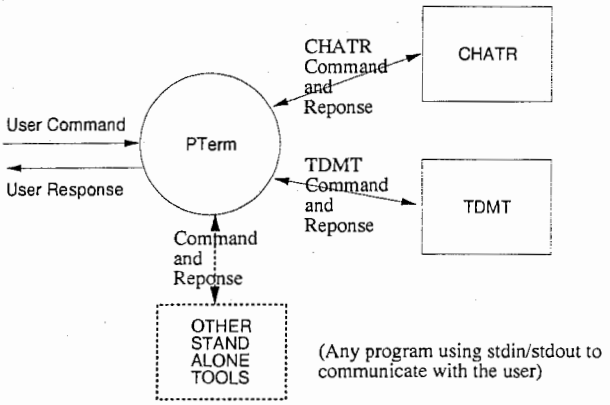
Currently, the user must specify the input language type for CHATR to choose the correct language speaker. However, the machine-readable input text already contains language information in its coding system; therefore, by use this coding information, CHATR can detect the input language and choose a suitable speaker or database. This improvement gives CHATR the ability to process multi-lingual text input. The detected language can then be processed separately for CHATR to synthesize multi-language output using a single language database.

If there are any remaining questions concerning details of the implementation, the first author can be reached by email at phwang@unixg.ubc.ca, and will give as much help as possible.

2 Psuedo Terminal (PTerm)

Currently, there are many speech tools being developed at ATR. Tools such as TDMT and interfaces to JANUS. PTerm is meant to be used to integrate the tools. One common property of these tools is that they take input from stdin and write their results to stdout.

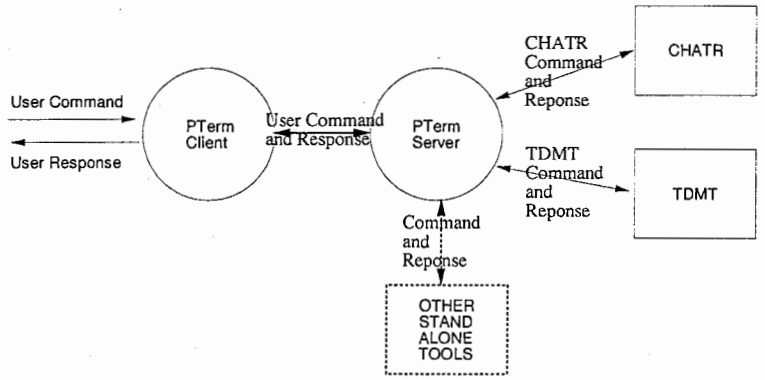
PTerm stands for Pseudo Terminal. It spawns a UNIX child process to run one of the stand-alone tools such as CHATR or TDMT. These child processes stdin and stdout are connected to the PTerm; thus, PTerm can communicate with the tools WITHOUT requiring modifyication of any of the these tools.
Following is the basic initial block diagram.



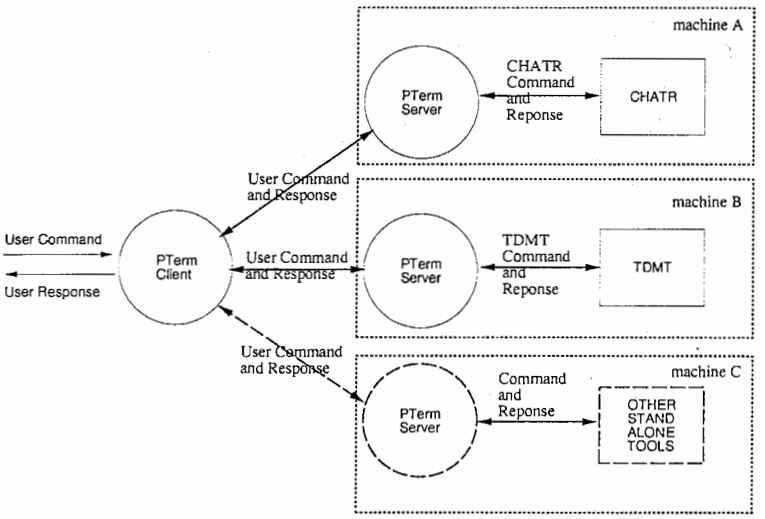
2.1 PTerm Structure

Soon after implementation, it was realized that time to start up CHATR and TDMT was much longer than its execution time of any single actual task. Therefore, a modification of PTerm was made so that it was sub-divided into a PTerm Server and a PTerm Client using a Server and Client model. Currently, both the PTerm Server and PTerm Client use the same binary executable. Upon execution, it checks for the argument flag to determine the mode of PTerm.
The PTerm Server runs the actual stand-alone tools while the PTerm Client takes users' requests and sends the requests to PTerm Server. Also, the PTerm Server may take requests from the user directly. This architecture makes PTerm much faster than before because the overhead of a PTerm Client is very small. This also, on the other hand, makes PTerm more difficult to implement and trace.

Following is the block diagram showing the relationship between a PTerm Server and a PTerm Client.



Furthermore, as a bonus from the client-server model implementation, the PTerm Client may connect to a PTerm Server on different physical machines to share loads among the machines on the network, because PTerm uses sockets for communication between the Server and the Client.



PTerm Server

The PTerm server simply loops and waits until requests arrive from clients; however, it is blocked if there are no requests from the Client. Received requests will then be preprocessed and redirected to each appropriate child. A child process can be a speech synthesis program such as CHATR, a language translation program such as TDMT, or any program that uses stdin and stdout as input or output. PTerm itself does not do intensive processing. It does minor text formatting where necessary for the child processes. The purpose of PTerm is to redirecting tasks to the children, not to do intensive calculation. This makes PTerm an ideal tool to glue two programs together if the output formatting of one program does not match the input formatting of the other program. All of the 'real' work is done by the child processes. PTerm only acts as an agent for passing data between the children. This means that PTerm must be simple and flexible so that it can integrate with other systems easily. At the moment, any program, which uses stdin as input and stdout as output, can be integrated with PTerm.

PTerm Client

The PTerm client loops itself and waits for input from the user; however it is blocked if there are no requests from the user. Received requests, without being modified, are redirected to the PTerm Server.

2.2 PTerm Commands

This section describes the CURRENT implemented commands in PTerm. New commands can be implemented into PTerm for other purposes.

Sending a Request to CHATR

(Assuming CHATR is running on the PTerm Server)
(chatr)[CHATR request]
eg.
(chatr)(SayText "hello")
PTerm can process the input string. It typically only parses the header (chatr) out and treats the rest as a CHATR command and sends it to CHATR directly. This can be useful as a quick test to see if CHATR is still running on the background. Note: PTerm does not check for the validity of the input command for CHATR.

Sending Request to TDMT

(Assuming TDMT is running at the PTerm Server)
(tdmt)[request]
eg.
(tdmt)(transfer::translate "hello")
PTerm does process the input string. It parses the header (tdmt) out and treats the rest as a TDMT command and sends it to TDMT directly. This can be used as a quick test to see if TDMT is still running on the background. Note: PTerm does not check for the validity of the input command for TDMT.

Translating Sentences

To use TDMT to translate one language to another language and sending the translated string to CHATR, PTerm has provided following commands.
(e-g): translate from English to German
(e-j): translate from English to Japanese
(e-k): translate from English to Korean
(j-g): translate from Japanese to German
(j-k): translate from Japanese to Korean
(j-e): translate from Japanese to English

Currently, ATR's TDMT only provides translation in the following modes: E-J, J-E, J-G, J-K, K-J. To translate from English to German, PTerm first translates the string to Japanese using TDMT E-J and translates the string again to German using TDMT J-G. This means, firstly, to do E-G operation, TDMT E-J and TDMT J-G must be running on the PTerm Server. Secondly, if either E-J or J-G translation fails, the translation becomes incomplete.

Command format: (e-j) "[text string]"
for example: (e-g) "good morning"
or (j-k)"名前"

IMPORTANT NOTE
If one tries to grep a Japanese Text from the shell screen using the mouse, he will notice that the format of the text would be changed! Make sure to only use the valid language coding! One may use 'od -ax' to check the formats. Detailed language coding information will be explained in the "Automated Language Detection and Romaji Conversion" section.

TDMT image-dump binaries in ../tdmt/bin only take *INTERNAL* coding. Therefore, Japanese, Korean and German inputs must be in the *INTERNAL* coding. It is possible for TDMT to take different coding methods; however, it was not done in PTerm and one will need to do a new image-dump of TDMT.
Also, these TDMT executables are statically created; it is not the newest version of TDMT available at ATR. Therefore, to obtain a newer version of TDMT, new image-dumps are needed.

CSTAR Related Commands

During the integration of CHATR with the CSTAR-II demonstration system. PTerm also takes the following as the input commands:

```
#chatr JR DIVS "[text_string]"
#chatr JR KANJI "[text_string]"
#chatr JR ROMAJI "[text_string]"
#chatr JP DIVS "[text_string]"
#chatr JP KANJI "[text_string]"
#chatr JP ROMAJI "[text_string]"
#chatr TARGET_SENT
-----------------------
JR : Japanese Recognized Sentence
JP : Japanese Parsed Sentence
DIVS  : ATR Lattice's phoneme-based string. (This may be different from
the phoneme set used in CHATR)
KANJI : KANJI Format (in EUC format)
ROMAJI : Romaji of the Kanji
```

Quitting from PTerm

Do not use CTRL-C to exit from PTerm, it will not close the open sockets correctly and users will have type 'ps -x' and kill all the children generated by PTerm manually! To quit from PTerm correctly type :
(quit)
and this will close all the sockets, files, and kill the child processes it generated.

However, if the PTerm client or server exits because of a critical error (such as allocated memory failure) the user will have to kill the children manually.

Trouble Shooting

CHATR is not outputting any speech

1. Check if CHATR is running on the PTerm Server, if it is, then on the PTerm Server type "(chatr)(SayText "hello")". This will request CHATR to output speech to the speaker. If speech is not produced, look at the AUDIO_COMMAND_LINE inside the configuration file for the PTerm server. In a UNIX shell with the same user account and machine name, start a CHATR; enter the line as same as AUDIO_COMMAND_LINE; make certain that the correct speaker is set; then, enter "(SayText "hello")".

If speech is not produced, then the error may be in CHATR, not PTerm, find the cause of the problem in CHATR and try again.

2. If speech is produced on the PTerm Server but not in the PTerm Client, then check the AUDIO_COMMAND_LINE inside the configuration file for PTerm client, because during the runtime, this AUDIO_COMMAND_LINE, in PTerm Client's cfg file, is send to PTerm so that if PTerm Client is running on a different host, CHATR will send the audio to the PTerm Client's host,instead of PTerm Server's host. Note: to do this, PTerm Client must use a utility like Datlink's 'naplay' to play a wavefile across the network, because the actual wavefile is never transferred by PTerm. Programs which allow playback on local machines such as DEC Alpha's audioplay should not be used if PTerm Client is running on a host with a different byte-order from the PTerm Server's host.

2.3 Message flow between The PTerm Server and Client

This section describes the message flow between a PTerm server and its client by giving an example.
1. Start a PTerm Server in one shell
2. Start a PTerm Client in another shell
3. type (chatr)(SayText "hello") in the PTerm client
4. Look at the PTerm Client's shell, scroll the window back and search for a message similar to this:

IN THE CLIENT'S WINDOW:

```
Pmem id : 0
-------------------
(PRead) Keyboard:
-------------------
string length : 25
EXECUTE TYPE: 0
size of : 25
allocated : 501
type: 0  request: 0
string : (chatr)(SayText "hello")

*****************************
Pmem id : 0
-------------------
(PWrite): PMEM_TYPE
-------------------
string length : 112
EXECUTE TYPE: 0
size of : 112
allocated : 112
type: 0  request: 0
string : (Audio Command "/usr/local/datlink/bin/naplay -f raw -e Linear
-o mono -u as65 -s $SR $FILE")
(SayText "hello")

*****************************
```

(PRead) Keyboard: contains the string which is read from the keyboard.
(PWrite): PMEM_TYPE: contains the string which is written to the PTerm Server.
if the message is for CHATR, then the "type" will be 0. The type of the message is corresponding to the child's enum number defined in ptermstruc.h.
After the PTerm Client writes to the PTerm Server, the PTerm Server will receive the message and thus following messages should be displayed in PTerm Server's shell.

IN THE SERVER'S WINDOW:

```
Pmem id : 0
-------------------
(PRead) SOCKET:
-------------------
string length : 112
EXECUTE TYPE: 0
size of : 112
```

```
allocated : 112
type: 0  request: 0
string : (Audio Command "/usr/local/datlink/bin/naplay -f raw -e Linear
-o mono -u as65 -s $SR $FILE")
(SayText "hello")

*****************************
Pmem id : 0
-------------------
(PWrite): PMEM_TYPE
-------------------
string length : 112
EXECUTE TYPE: 0
size of : 112
allocated : 112
type: 0  request: 0
string : (Audio Command "/usr/local/datlink/bin/naplay -f raw -e Linear
-o mono -u as65 -s $SR $FILE")
(SayText "hello")

*****************************
```

(PRead) SOCKET: contains the string which the PTerm Server read from the socket. This should be the same as the one the PTerm Client wrote. The size of allocated may be different; however, the size of the actual memory used must be the same.
(PWrite): PMEM_TYPE: contains the strings which the PTerm Server wrote to the CHILD. Also, check the type of the string. It should be 0 if it is for CHATR.

IN THE SERVER'S WINDOW:

```
Pmem id : 0
-------------------
(PRead) CHILD:
-------------------
string length : 122
EXECUTE TYPE: 0
size of : 122
allocated : 123
type: 0  request: 0
string : (Audio Command "/usr/local/datlink/bin/naplay -f raw -e Linear
-o mono -u as65 -s $SR $FILE")
ok
chatr>
*****************************
Pmem id : 0
-------------------
```

```
(PWrite): PMEM_REPLY_TYPE
------------------
string length : 122
EXECUTE TYPE: 0
size of : 122
allocated : 123
type: 0  request: 0
string : (Audio Command "/usr/local/datlink/bin/naplay -f raw -e Linear
-o mono -u as65 -s $SR $FILE")
ok
chatr>
*****************************
------------------
(PRead) CHILD:
------------------
string length : 118
EXECUTE TYPE: 0
size of : 118
allocated : 123
type: 0  request: 0
string : (SayText "hello")
Waiting for output to finish...Done
Transferred 4724 samples
#<Utt 4cf988>
chatr>
*****************************
Pmem id : 0
------------------
(PWrite): PMEM_REPLY_TYPE
------------------
string length : 118
EXECUTE TYPE: 0
size of : 118
allocated : 123
type: 0  request: 0
string : (SayText "hello")
Waiting for output to finish...Done
Transferred 4724 samples
#<Utt 4cf988>
chatr>
*****************************
```

(PRead) CHILD: contains the string which PTerm read from CHATR. Note that the requesting string appears on the reply string; it is because CHATR puts the request string back with the reply. It is correct.
(PWrite): PMEM_REPLY_TYPE: contains the string which PTerm returns to the PTerm Client. Note that type of the string is still 0 for CHATR

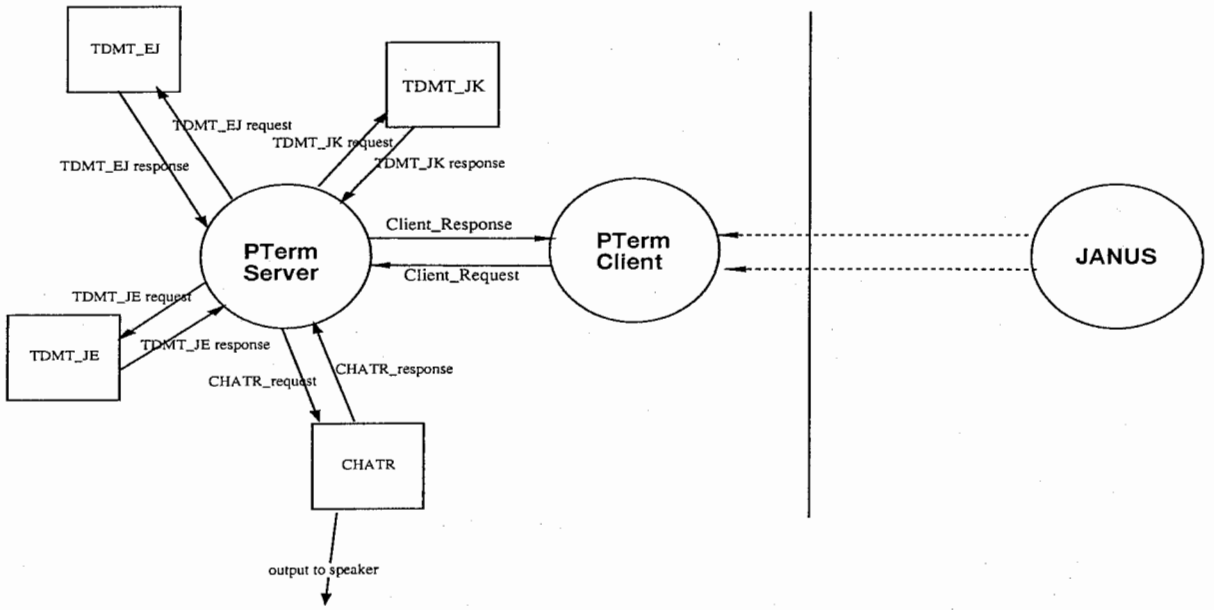
IN THE CLIENT'S WINDOW:

```
Pmem id : 0
-------------------
(PRead) SOCKET:
-------------------
string length : 122
EXECUTE TYPE: 0
size of : 122
allocated : 122
type: 0  request: 0
string : (Audio Command "/usr/local/datlink/bin/naplay -f raw -e Linear
-o mono -u as65 -s $SR $FILE")
ok
chatr>
*****************************

#controller CHATR : READY
Pmem id : 0
-------------------
(PRead) SOCKET:
-------------------
string length : 118
EXECUTE TYPE: 0
size of : 118
allocated : 118
type: 0  request: 0
string : (SayText "hello")
Waiting for output to finish...Done
Transferred 4724 samples
#<Utt 4cf988>
chatr>
*****************************
```

Finally, the PTerm Client receives a reply from CHATR vis PTerm Server.

2.4 Integrating JANUS system using PTerm

In this system, PTerm waits for the semaphore ready-file from JANUS, speech recognizer developed at CMU. Then, PTerm formats the recognized string for TDMT, the text translator developed at ATR. Finally both recognized and translated strings are synthesized with CHATR.



to start JANUS:
1. log in as 'tschultz' (or use su - tschultz)
2. go to '/homes/tschultz/janus-demo'
3. start by entering 'run_janus_demo'

2.5 Interfacing CHATR in the CSTAR-II Demonstration using PTerm

In this demo system, PTerm acts as an interface agent between the controller and CHATR. PTerm formats the string from the ATR recognizer and the controller for CHATR, then, passes the commands to CHATR. This simplifies the workload of the controller, so the system can be easily integrated in a short time.

2.6 PTerm Modules (Programmer's Guide)

This section describes PTerm's modules in detail.

PMEM (pmem.c)

One disadvantage of programming in C is that memory allocation is not tracked by the program. For example, one may allocate a memory block of 100 bytes but write over a boundary without noticing it until the program crashes or ends up in an unknown state. Unfortunately, this kind of error is very difficult to track and it is much easier to write and use a set of functions calls to allocate and to deallocate the memory, and at the same time to keep track of the size of allocated memory blocks. The PMEM module provides many methods to do

string-manipulation and it also keeps track of the memory allocated automatically. It will automatically reallocate memory if the allocated memory block is too small to use.

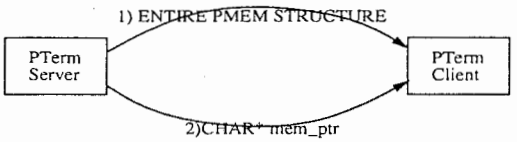
The detailed descriptions of each method described in pmem.c

Another advantage of using this module is that it also keeps track of the byte ordering of the memory blocks. For example, if the PTerm client is running on a SunOS while the PTerm server is running on a OSF1 machine, pmem will able to detect this difference and swap the byte order correctly. Currently, SunOS and DEC Alpha has been tested. PTerm has also been compiled on HP; however, it has not been well-tested when it connects to a different OS system.

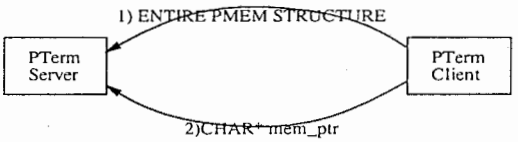
Two more important notes associate with PMEM structure. First, if a new item is inserted to the Pmem structure, one should make sure that PRead() in the psock.c is modified accordingly. In PRead(), it swaps the byte ordering depending on the machine type of PTerm Server and PTerm client, if the new inserted item in the PTerm structure is not byte swapped correctly, it will cause undetermined errors.

Secondly, notice that inside the Pmem structure, There is a pointer that is not used, this was done to compensate the difference of the pointer size in each different OS. In OSF1, a char pointer is 8 bytes while in SunOS, a char pointer is 4 bytes; therefore to align the bytes, we need the extra pointer in SunOS to pad the structure. Whenever the Pmem structure is changed, check the size of the Pmem structure by displaying the size of the structure ON ALL SUPPORTED machine types. Make sure that the size of the Pmem structure is the SAME on all support machine types. If they are different, one must align the structure boundaries and compile the code again.

The size of the PMEM structure is important because the entire PMEM structure is written to the socket for faster operation. In the long run, this creates fewer problems because one does not need to keep track the 'order' which the structure elements are written to the socket. After the structure is written to the socket, mem_ptr is written to the socket. Therefore, if the size of the PMEM structure does not agree on the different operating system, a write/read from the socket will corrupt the data in the socket pipe.



Write from Server to Client



Write from Client to Server

PBUFFER (pbuffer.c)

This module keeps a circular buffer of size defined by:

```
#define MAX_BUF_ITEM 100 (in pbufferstruc.h)
```

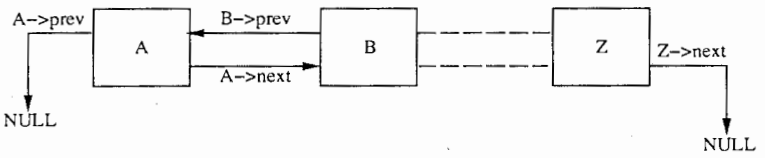
Therefore, to change the size of this buffer, you will need to recompile PTerm. This module handles the internal buffering of the PTerm. To send a message/request to a PTerm server or to send a reply back to a PTerm client, simply add the request/reply to the pbuffer and the message will be passed to the destination. Of course, the message's destination must be correctly set BEFORE putting it to the buffer.

PSOCK (psock.c)

This module is designed to handle all the socket I/O, keyboard inputs, or stdin inputs.
The two main functions in this module are PWrite and PRead. This module is intended to be as transparent to the programmer as possible. However, to able to detect 'end of command' from the child processes, prompts of the children are used. This is the only part in PTerm that is hard-coded.

PTOKEN (ptoken.c)

This module is a LIST implementation and it is much easier to explain this structure graphically; the, following is a graphical description of the LIST.



Each of the rectangular blocks is called a token which keeps track of its previous and following token in the list. Apart from keeping track of the link relationship, it also keeps a PMEM structure for saving information. This is implemented on top of PMEM so that memory allocation problem can be also transparent for this module.

PMISC (pmisc.c)

This module include several useful function calls.
1. filelog(): This function is called to log a a PMEM structure to a file. It checks for the PMEM type, and writes to the pre-designated log file for that PMEM type. The file pointer to this log file is declared globally and the file must be open before execution of PTerm.

PCHATR (pchatr.c)

This module contains functions to process the output from CHATR.

PTDMT (ptdmt.c)

This module contains functions to process the output from TDMT.

PJANUS (pjanus.c)

This module contains functions to process the output from JANUS. JANUS is not well integrated into PTerm because it is too large to run as a child process. Also, it is too complicated to be run as one child, because JANUS, itself spawns children and communicates with these children using its own protocols. Therefore, this module just checks the semaphore-ready files from JANUS. If the semaphore file is detected, it then processes the JANUS outputs accordingly. One thing that needs to be mentioned is that PTerm is blocked at the SELECT statement in function PTman() of pterm.c. This will ensure that PTerm does not waste system resources by looping itself repeatedly. However, this also means that, if there is no input from socket, keyboard OR stdin, PTerm will be blocked indefinitely even if the JANUS's semaphore file is created. To solve this problem, PTerm will have to timeout itself from the SELECT statement. The timeout period is set with the local variable 'timeout.tv_sec' in the function. If this timeout period is long, it will be more 'efficient' but the response time to JANUS's output will be longer. If this timeout period is short, PTerm will use too much system resource.

to start JANUS:
1. log in as 'tschultz'
2. go to '/homes/tschultz/janus-demo'
3. start by entering 'run_janus_demo'

2.7 Start up sequence in PTerm (Programmer's Guide)

Connections between Server and the Client are done synchronously described as follows:
1. PTerm Server starts to execute and it is running.
2. PTerm Client starts to execute, it checks the configuration file and figures out which host the PTerm Server is running.
3. PTerm Client connects to the PTerm Server using port 4369 (0x1111). This 0x1111 port was picked to avoid byte ordering problem between SunOS and OSF1
4. PTerm Server accepts PTerm Client's request and socket-level connection established
5. PTerm Client sends a signature to the PTerm Server. The signature includes OS information which the client is running on, kind of requests which client will be using this socket connection for, such as TDMT or CHATR requests.

6. PTerm Server reads the signature and records them locally.
7. PTerm Server ACKs the signature back to the client with it own information.
8. PTerm Client records the signature file locally as well.
9. Connection established.

If error occurs in any of the above steps, the connection will fail and an error will be reported.

Similarly, if the PTerm Client is closing a connection. It will send a message to the PTerm Server requesting closing the connection BEFORE the connection is really closed. This is used to prevent unprocessed messages being queued up on the PTerm Server's buffer module while PTerm Client closes the socket connection by itself. PTerm Server will experience a write error, if it attempts to write to a closed socket.

2.8 Inserting a New Child Type (Programmer's Guide)

It is possible to add a new child type into PTerm. Efforts were tried to make PTerm easy to integrate with other stand-alone programs such as CHATR or TDMT; however, due to the increasing complexity of PTerm, it is not as easy as it was expected to be. The following steps will be explained here when inserting a new child type.

1. Modify ptermstruc.h, to add a new enum variable to the list. Also increase definition of NUM_MODULES by one.
2. Modify ReadSetUpCfg() in ptermconfig.c, so that the new enum variable is processed correctly.
3. Modify Create_Slave_Worker() in pmisc.c, so that the new enum variable is processed correctly.
4. Modify filelog() in the pmisc.c to take the new enum variable. Please do not forget to declare a new global variable for the file pointer in pterm.h and fopen the logfile in pterm.c.
5. Modify ProcessExecCommandProc() in pcommandproc.c to take the new enum variable.

After this, recompile the code and PTerm will able to run the child and connect the child's stdin/stdout properly.

2.9 Modify pmisc.c

Upon execution, PTerm reads the configuration file given, by default, in ptermsrv.cfg (if server) or ptermcli.cfg (if client). If a new child type is added, new informations about the child will be read from the configuration file, thus, the function which reads the configuration file need to be modified. Such information is read by the function:

2.10 Special Routine Handling

Occasionally, one needs to obtain information from the children before continues to process the data. However, all messages are sent to the children are simply handled by the PBUFFER

module and programmers are not able to determine when the reply messages will be returned from the children. It will be inefficient if PTerm is blocked for the reply message (other request may be pending on the queue). To solve this problem, Pterm's PMEM structure takes another variable called execute_type. By default, it is set to zero; however, if reply messages from the children is required further processing, special enum values may be inserted to the field. As the reply message returns, the the variable is checked and appropriate function call is made from ProcessExecCommandProc(). The ProcessExecCommandProc() itself is called from pterm.c

2.11 Current Unsolved Problems in PTerm

There is a yet unsolved problem in PTerm's socket module. If the PTerm client crashes, PTerm Server will not be able to detect such error and it will leave the socket open until PTerm Server exits the program. It will be ideal if PTerm Server can checks for all socket connections once a while and closes the sockets which PTerm Client has crashed.

Known problems when connecting to the children

PTerm's earlier versions used 'tty' and 'pty' to connect the children's stdin/stdout. However, due to limited number of tty's and pty's available on most UNIX OS, PTerm was modified to use the Unix function 'socketpair()' to connect with the children's stdin/stdout in its later versions. This method works fine with TDMT, but it does not work with CHATR. The reason is not yet clear. Therefore, currently, CHATR uses tty and pty while TDMT uses socketpair(). In either case, the basic module architecture remain unchanged. When creating connecting the children to PTerm in pterm.c, there are two methods to create the connections, tty/pty method and socketpair method. One should try to use the socketpair method, if data became missing on the socketpair connection, then try the tty/pty method.

2. PTerm is not yet able to exit the code completely; it seems that it can in certain conditions get into an infinite loop and fail to exit the PTerm. We can use CTRL-C to exit PTerm but need to remember to kill the left over processes.

2.12 Glossary

PTerm Server: The PTerm which spawns and runs the stand-alone programs such as CHATR, and TDMT as children.

PTerm Client: The PTerm which DOES NOT spawn any children. It handles requests by redirecting the request to the server and waiting for the server's reply. It treats the server's reply AS IF it is the direct reply from one of the children.

Child: Child processes such as CHATR, and TDMT spawned by the PTerm Server.

- connection between server and the child processors (socketpair and stdout) unknown

- hard coded part of the PRead. - where to modify if new module is wish to be added. -

weakness of PTerm. - it became much more complicated than I first intended. - function return is not unified

3 Prosody and CHATR

In this section, importance of prosody information in a speech-speech translation system will be discussed.

3.1 Problem with a Text-Only Translation System

Conventionally, a speech-to-speech translation system contains three major components: a speech recognizer, a language translator and a speech synthesizer.

Current speech translation system

The speech recognizer processes the input speech and outputs a text of what has been recognized. Then the language translator takes the recognized string and compares to the example sentences which it knows how to translate. At this point, information could have been already corrupted, due to the loss of prosody.
Prosody contains important information. The same text said with with different prosody can have quite a different meaning. For example, "Could I have your phone number?" translates quite differently if focus is placed on the different words within the sentence. The meaning of the following sentence changes with respect of the focus on the highlighted words.

Could **I** have your phone number too?

Could I have **your** phone number too?

Could I have your **phone** number too?

Could I have your **phone number** too?

Could I have your phone number **too** ?

Accordingly, the translation can be appropriately different if the language translator can detect such information during the language translation phase. This prosodic information is in some cases vital for a concise, and correct translation.

For the same reason, the speech synthesizer also requires the prosodic information to deliver the meaning of a sentence correctly. One may ask if human can detect prosody changes in a synthesized speech. The answer was confirmed by one of the experiments done by Nick

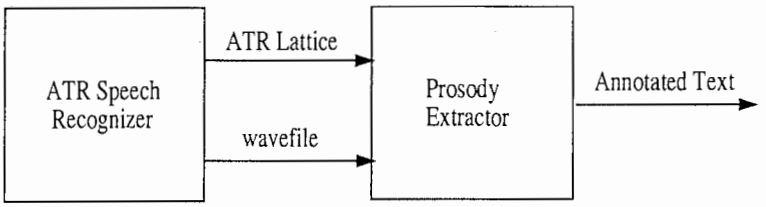
Campbell. He prepared 10 example Japanese sentences; each of which was produced with 3 different focal prominences on the different words; each sentence was synthesized/spoken 3 times to generate 90 test sentences. It was found that 84 percent of these sentences spoken by a Tokyo Japanese were correctly interpreted by native speakers of the language, while 74 percent of the sentences spoken by the ATR speech synthesizer, CHATR, were correctly detected. The small (only 10 percent) difference in the experimental results shows that human listeners CAN detect a switch in focal prominence even in a synthesized utterance. Thus prosody information processing is important for both correct language translation and information delivering. Without this information, correct information transfer between languages is not possible.

In the final version of the ATR speech-to-speech translation system, both TDMT and CHATR will require the prosody information for correct information transfer. However neither of them takes enough advantage of this useful information at the moment.

3.2 Prosody Extraction

Having shown the importance of prosody in forming a meaningful language transfer, we can see that prosody extraction becomes, naturally, the next issue. To be able to deliver accurate and meaningful synthesized speech, we must first understand what the parameters or the characteristics of meaningful prosodic variation are. Therefore, if we can model prosody and extract it from a given speech, we will begin to understand what functional prosody is.

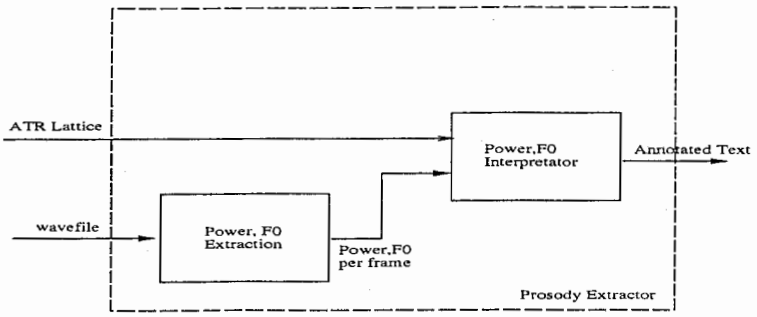
The following is the proposed block diagram for the prosody extraction.



Prosody Extraction Overview

The recognized wavefile and the ATR lattice information are used to extract prosody. This has two advantages; it abstracts the prosody extraction from the speech recognizer and it maintains all the prosody information in the wavefiles. However, this will increase the workload on the computers. Output of the prosody extractor will be the annotated text containing focus, duration and intonation of the spoken speech. This annotated text can be passed to CHATR for better speech synthesis or to TDMT for more concise language translation.

Following is the block diagram of prosody extractor.

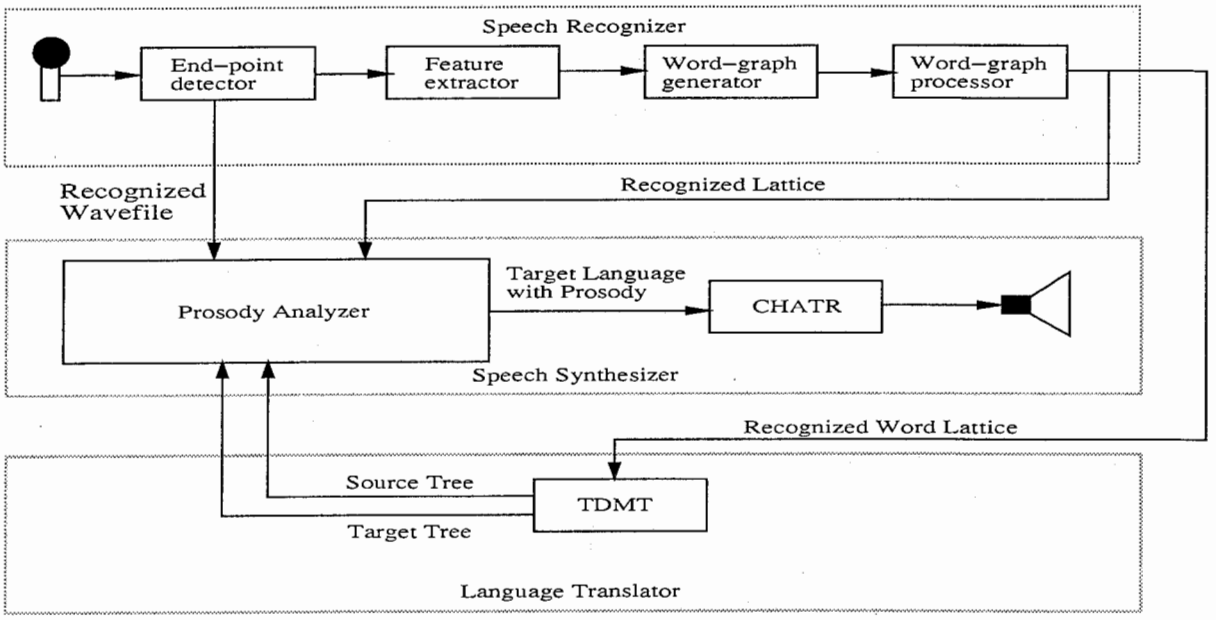


The wavefile's f0, power and other parameters to model the prosody are calculated. Details of these parameters are described in Nick Campbell's and Wen Ding's published papers. Since the ATR lattice gives timing information for each word that it recognizes, we may put a special symbol on the word for which the focus is detected, or equivalently on the translated word for which the intonation should be raised.

3.3 Prosody Transfer

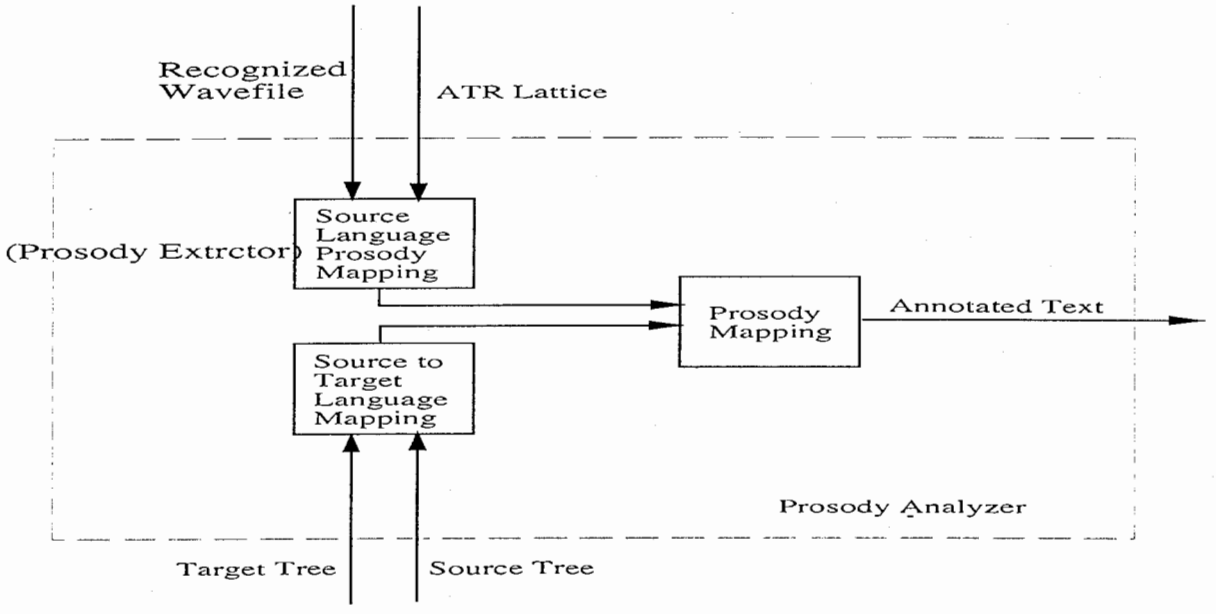
Currently, TDMT does not take annotated text as its input, instead, it only takes the plain text containing no prosodic information and translates the text using the examples in its database. In the previous example "Could I have your phone number?" shown before, it contains five possible focal prominences! It is not possible for TDMT to 'guess' the intended meaning without using prosody. Therefore we can conclude that until TDMT changes it current input method, CHATR will not able to synthesize correct translated speech. To compensate for this, we attempt a workaround by transfering the prosody using TDMT's source and target trees.

The following is our proposed prosody transfer system.



Information flow for prosody transfer

At the center of the diagram is the prosody analyzer which is shown in detail below:



prosody transfer mapping

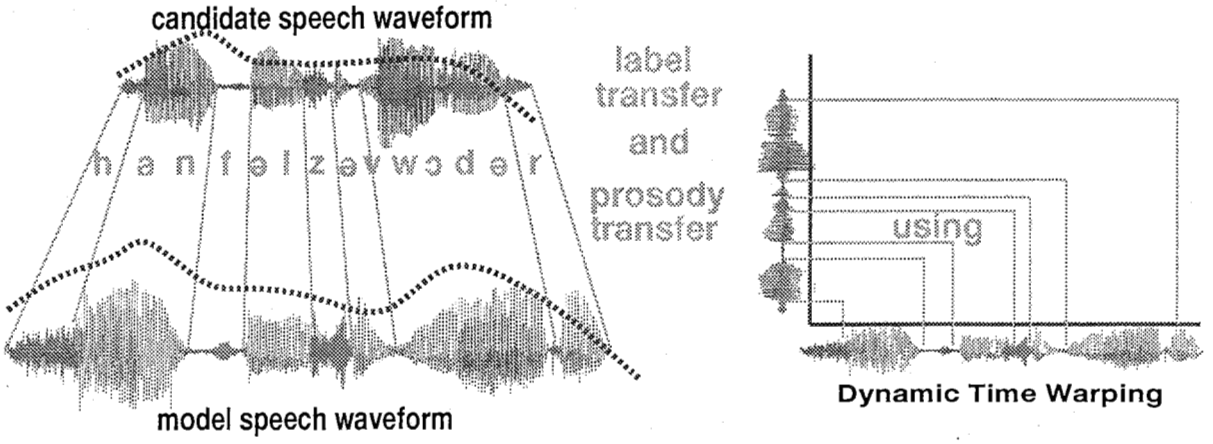
It basically contains a prosody extractor which extracts prosody information from the source speech, as described earlier. Because TDMT can give a source and a target tree structure of the translated string, then if we know the word which contains the focus in the source tree, we can insert a similar focus on the word in the target tree. Following is an example of the TDMT source and target tree generations.

```
Input: "Could I have your phone number, too?"
                          ^^^^^^^^^^^^

=> TOP [(?X too) --- AS]
   |--?X [(could ?X <PRON-V> ?Y) --- MS]
      |--?X [i]
      |
      |--?Y [(?X <V-DET> ?Y) --- PN]
         |--?X [(have)]
         |
         |--?Y [(your ?X) --- DN]
            |--?X [(phone number)]

=> TOP [0.000 ((副詞 また) {読点_ 区切} !X)]
   |--!X [0.000 ({_ 副詞節} !Y (接続助詞 ても)
                    (形容詞 よい) (終助詞 か))]
      |--!Y [0.000 (!Y (格助詞 を) !X)]
         |--!Y [0.000 ((接頭辞 御) !X)]
         |  |--!X [STRING ((普通名詞 電話番号))]
         |
         |--!X [STRING ((本動詞 伺う))]

  (副詞 "また"){読点_ 区切}
  ({_ 副詞節}
   (接頭辞 "御")
   (普通名詞 "電話番号")
   (格助詞 "を")
   (本動詞 "伺う")
   (接続助詞 "ても")
   (形容詞 "よい")
   (終助詞 "か")))

=> また、御電話番号を伺ってもよいですか。
          ^^^^^^^
```

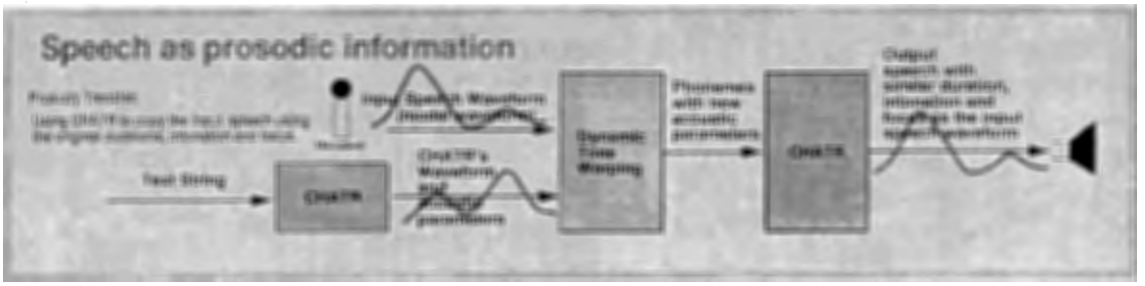
In the above example, if the focus is detected in the word,"telephone number", we will able to insert a focus on the "電話番号" in the target sentence. However, due to the fundamental differences among languages, if the focus is detected in the word 'I', which is often omitted in an English-Japanese translation, the word cannot be found in the target tree and therefore the attempt for prosody transfer will fail. We are not able to do prosody transfer on any given cases due to this differences among languages; however, in many cases, prosody transfer can be done successfully. Another very important factor in prosody transfer is the speech recognizer. Often, a speech recognizer puts out a parsed or regenerated sentence, a sentence which differs from the sentence which the recognizer recognizes. In this case, the timing information that the recognizer gives will be not align correctly with the actual speech wave; therefore, the prosody extraction will fail and so will the prosody transfer.

3.4 Application of CHATR using Prosody and DTW

In this section, some interesting applications of CHATR and DTW will be discussed. DTW stands for dynamic time warping. It is a technique to map two similar wavefiles from one to the other and gives the distance between the two wavefiles. A diagram which illustrates DTW is given below.

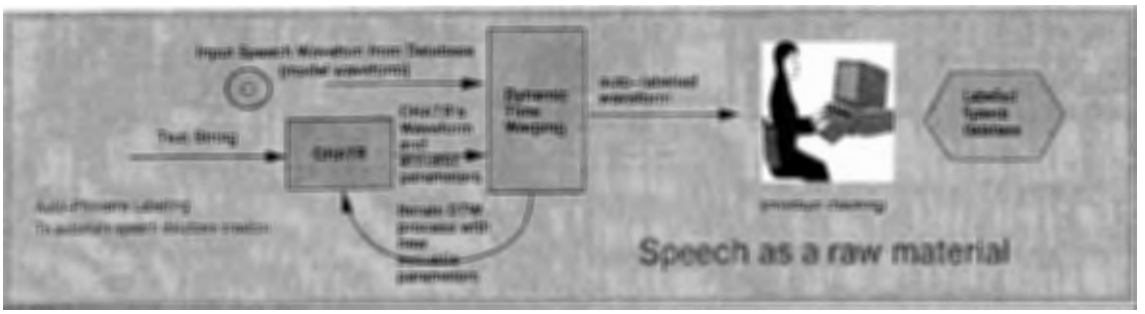


A Voice Driven Speech Synthesizer



Traditionally, a speech synthesizer has been designed to be only text-driven. However, with DTW, we can envision a voice-and-text driven speech synthesizer. In the above diagram, a text string is first input to the synthesizer and the synthesizer outputs a synthesized speech. This synthesized speech frequently has unnatural intonation. At this time, if a model speech wave with correct intonation and focus is presented to the system we can correct it. Such model wave is, then, compared to the synthesized wave; we will be able to adjust the intonation of the synthesized speech and thus increase the quality of the synthesized speech. The potential for such an application becomes understandable when we consider that Chatr is now capable of producing highly natural-sounding voice-quality in a variety of different, and sometimes very well-known voices.

Automatic Database Labelling

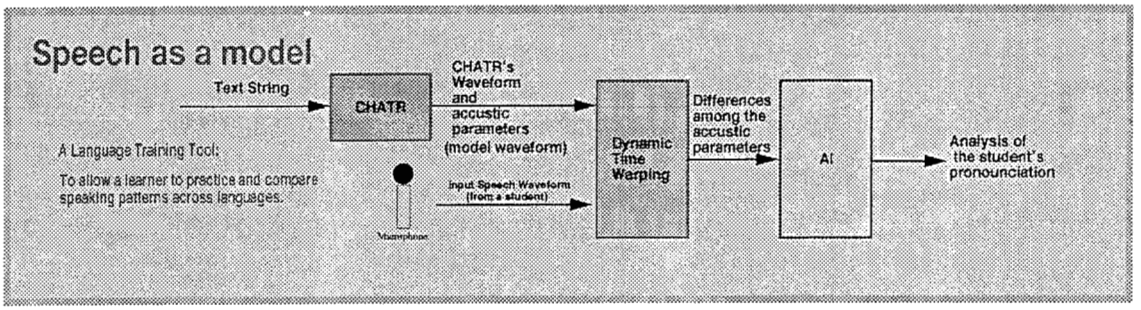


As mentioned before, DTW is a technique to map two similar waveforms. The more similar the waveforms are, the shorter the distance between these two waveforms. Thus the mapping becomes more accurate.

Currently, tremendous efforts are spent to label a speech database. The unlabeled speech database contains numbers of wavefiles and the text of these waves. A labeller has to label the wavefiles phoneme by phoneme. This is a time consuming process. For CHATR, the larger its database is, the better synthesis it can generate. Therefore, the ideal goal is to label the database automatically by machines.

If the text of the speech in the unlabeled database is used to generate a synthesized speech using CHATR. Such speech is then compared to the unlabeled wavefile in the database using DTW. A distance and a map will be obtained from the DTW process. We can, then, use CHATR to resynthesize the same speech using a more detailed information such as 'f0' and 'power' obtained from the DTW mapping. The idea is to get the CHATR to synthesize a close-enough speech for more accurate mapping between the two speeches. Even if the map is not entirely correct, it has already eased the labeller's work, because the labeller only has to checks the accuracy of the map and modify the inappropriate mappings.

A Language Learning Tool



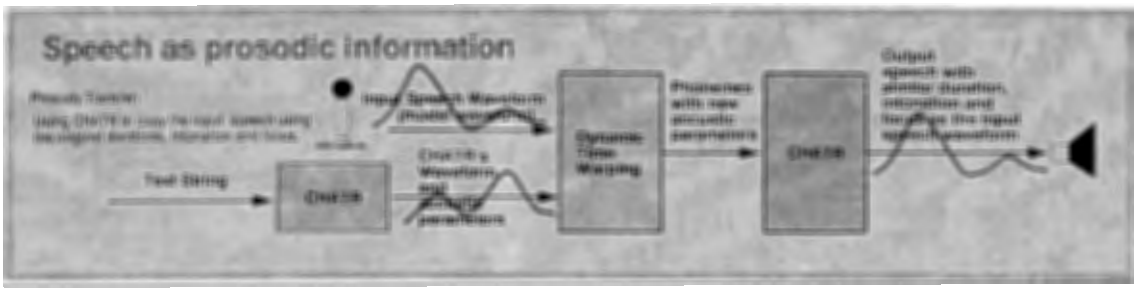
So far, we have only considered cases which CHATR to learn prosody from a human. Conversely, human could learn prosody from CHATR, especially in a foreign language, if CHATR

can output better synthesized speech. At the moment, although CHATR's Japanese is not as good as native Japanese speakers, CHATR's Japanese is surely a lot better than many non-native Japanese speakers.

In this application, a student's speech is recorded and compared to CHATR's synthesized speech using DTW. The comparison can be analysized; feedbacks about student's focus, intonation are reported to the student. This enable a student to improve his pronunciation and speed up the language learning process by himself!

3.5 Automatic Database Labelling

This section describes the implementation of the automatic database labelling. Time Warping (DTW) technique is used to map the two similar speech waves. Sequence of Process is described as:
1. Send the text string to CHATR and obtain the wave generated by CHATR and the duration of the phonemes.
2. Processes DTW on the original wave and CHATR wave.
3. Obtain the f0 and power from the original wave and use CHATR to re-synthesize the speech.



PTerm could be used to connect CHATR and therefore obtain the wave and phoneme-segment information from CHATR. However, as an interim solution, the process was written in shell scripts instead. This script can be found at "/home/as65/xphwang/src/auto_label".

3.6 Modules/Script/Program Description

==========

get_text.awk:
This awk script will convert a text string in a file into a format which can be directly sent to CHATR server. This includes, setting a speaker, setting the concat_method and requesting CHATR server to save the wavefile, XSegs, and unitlabel to the file system.

Usage/Example:
(in the directory of auto_label)

cat ./example/sample.txt — gawk -f get_text.awk

Note:
1) CHATR server saves the wavefile, XSegs ..etc to ITS directory.
To obtain these files, you MUST copy the files to your own space!

==========

get_segment.awk:
This awk script will convert a XSegs files into a format which can be directly sent to CHATR server. This includes setting a speaker, setting the concat method and requesting CHATR server to wavefile, XSegs, and unitlabels to the file system.

Usage/Example:
(in the directory of auto_label)
cat ./example/sample.xlb — gawk -f get_segment.awk

Note:
1) CHATR server saves the wavefile, XSegs ..etc to ITS directory.
To obtain these files, you MUST copy the files to your own space!

==========

unit2xlb.awk:
Because output of CHATR's XSegs labels is NOT corresponding to the actual wavefile concatenated by CHATR. To compliment with this, I have written a script with convert CHATR's UnitLabels to a Xlb formatted file.

Usage/Example:
(in the directory of auto_label)
cat ./example/sample.unit — gawk -f unit2xlb.awk

NOTE:
1) This should be removed from the final script once the bug has been fixed.

2) Currently, the output of CHATR wavefile STILL does not match with the information given by the UnitLabel file. (difference about 2ms) This may cause error when we are doing DTW mapping.

==========

3.7 How To Run DTW

% get_label [name]

Input File:
a model wavefile and a textfile corresponding to that wavefile. naming of these two files are corresponding to the argument of the get_label script.

wavefile: [name].wav
textfile: [name].txt

eg. % get_label record
the model wavefile would be named as 'record.wav'
and
the textfile would be named as 'record.txt'

Output Files:

result.unit: file for Ding-san to do prosody modification.
result.unit.read : semaphore file for notifying that the result.unit is ready
ncDTWchatr.d : CHATR generated wavefile in ESPS format
ncDTWchatr.xlb : CHATR generated XSegs file
ncDTWchatr.unit : CHATR generated UnitLabel file
ncDTWchatr.map : DTW mapping table between the two wavefiles
ncDTWchatr.cep : Cepstrum file of ncDTWchatr.d
ncDTWchatr.d.noheader : CHATR generated wavefile without header
ncDTWchatr.f0 : output file of get_f0 on ncDTWchatr.d
ncDTWchatr.mfc : output file of HCopy on nhkDTWchatr.d

nhkDTWchatr.d : original wavefile in ESPS format
nhkDTWchatr.xlb : XSegs file generated with DTW mapping info
for the original wavefile
nhkDTWchatr.noheader : original wavefile WITHOUT header
nhkDTWchatr.noheader.epd : original wavefile after EPD
nhkDTWchatr.f0 : output of get_f0 on nhkDTWchatr.d
nhkDTWchatr.mfc : output of HCopy on nhkDTWchatr.d

3.8 DTW Module : dtw_g.c

This program takes *.mfc input format files. The input files will be different in length. The shorter wave has been with padded zeros at beginning and at the end of the wave to match the length of the longer wave. This was done because, the original recorded wave usually has

silent at the beginning and at the end of the speech. CHATR, however, does not put out silence/pause This makes CHATR's speech wave shorter than the original most of the time.

3.9 Final Notes

This program is built up out of many scripts and small programs. It is better to go through them one by one and figure out the sequence of the logic.
Firstly, the script gets a wave generated by CHATR which usually contains a NIST-header. In order to do DTW, the NIST-header should be removed from the script. Also, because dtw_g takes *.mfc as input, we need to convert the raw waves (both CHATR generated, and the original) to the HCODE mel cepstrum format using HCopy as below:

```
/usr/local/HTK/HTK_V2.02/bin.sun4/HCopy -C hcode.config {input file} {output file}
(remember to add HTK_V2.02 to your shell path and hcode.config is
copied to the current working directory)
```

Because silence is not generated at the start of CHATR's speech wave, and pauses in Chatr are still not well predicted, the DTW mapping accuracy seems to be indirectly proportional to the number of the silences in the original speech.

4 Automated Language Detection and Romaji Conversion

The final section of this report describes a program which will convert given encodings of language inputs after detecting the language type. Upon detection, it converts the input text to romaji-like phoneme specification.
The source code and the executables are in
"/home/xphwang/as65/src/conv_ romaji".

conv_romaji can take the following input encoding format and converts them to speaker-specific phonemes for chatr:

Japanese EUC Coding
Japanese JIS Coding
Japanese Internal Coding
Korean EUC Coding
Korean KIS Coding (Equivalent to Japanese JIS Coding)
Korean Internal Coding
German Internal Coding
German Umlaut Coding (basically, it is Internal Coding with out the internal code header
German GIS Coding

Note:
Since the kanji conversion currently uses KDD's kan2rom script. The program 'kan2rom' must be in the search path!

4.1 How to Use

conv_romaji can take input from standard in or from a file. In both cases, the converted romaji and print it using standard out (stdout).

Program Flags

-debug : this will print output message verbosely.
-kakasi : this will use kakasi as the Kanji-romaji converter instead of kan2rom. It is *much faster* than kan2rom
-nokey : if this is on, then beginning of each language segment will not have a language indicator such as /E/ for English, /K/ for Korean, /G/ for German and /J/ for Japanese.
-phoneme : if this is on, it will print out phonemes instead of romaji. (not fully implemented yet. it partially works for Japanese right now)
-f {file}: it takes the argument following this switch as the input file. If this flag is not used in execution time, stdin is default.

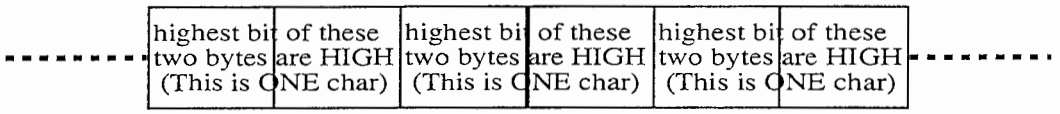
-E : (if -phoneme is also selected), it gives the phonemes for a CHATR English Speaker using 'beep.ch'.
-J : (if -phoneme is also selected), it gives the phonemes for a CHATR Japanese Speaker using 'nuuph.ch'.
-K : (if -phoneme is also selected), it gives the phonemes for a CHATR Korean Speaker using 'korean_hcode.ch'
-G : (if -phoneme is also selected), it gives the phonemes for a CHATR German Speaker.

4.2 Language Codings

This section explains the different codings that the program supports. One may use 'od -ax' to see the coding of the input string/file.

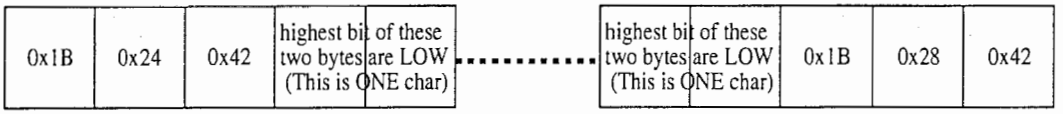
Japanese EUC Coding

This is a two-byte coding method. Basically each pair of bytes will represent a kanji, hiragana or katagana character. The highest bit of the two bytes will be set to HIGH. This coding confuses the program because there is no indication of either this is a Japanese EUC Coding or Korean EUC Coding. However, because Korean character set is much smaller than the Japanese character set, if an EUC coding is detected, the program tries to convert to the Korean Romaji first. If it fails to convert to Korean Romaji, then, it assume that it is Japanese EUC Coding. Therefore, it is possible for detecting a Japanese EUC Coding as a Korean EUC Coding.



Japanese JIS Coding

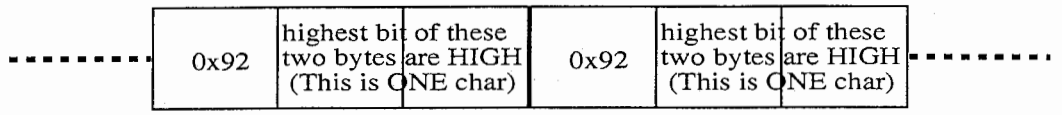
This is a two-byte coding method with a header and tailer. Similar to Japanese EUC, each pair of bytes will represent a kanji, hiragana or a katagana character. However, the highest bit of the two bytes will be set to LOW.



Japanese Internal Coding

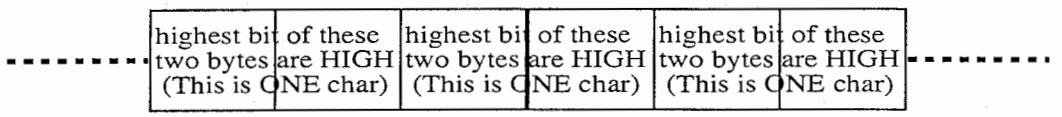
This is a three-byte coding method. Each three bytes will represent a single kanji, hiragana or a katagana character. First byte is an *INTERNAL* header. One may check this

header and detect the language type, because this header is different from Korean or German *INTERNAL* codings.



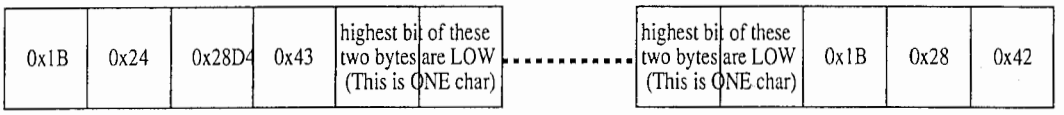
Korean EUC Coding

This is a two-byte coding method. Similar to Japanese EUC Coding, each two bytes will represent a Korean character. The highest bit of the two bytes are set to HIGH. This coding confuses program because there is no indication of either this is a Japanese EUC Coding or Korean EUC Coding. However, because Korean character set is much smaller than the Japanese character set, if an EUC coding is detected, the program tries to convert to the Korean Romaji first. If it fails to convert to Korean Romaji, then, it assume that it is Japanese EUC Coding. Therefore, it is possible for detecting a Japanese EUC Coding as a Korean EUC Coding.



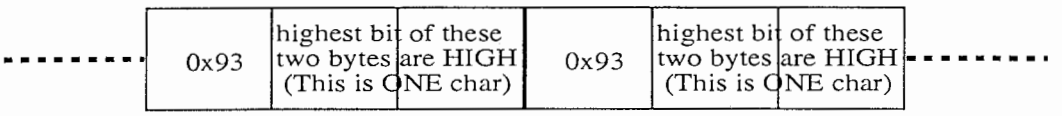
Korean KIS Coding(Equivalent to Japanese JIS Coding)

This is a two-byte coding method with a header and tailer. One can use this header to determine the language type, because the header is different from the Japanese JIS header. Similar to Japanese JIS Coding, the highest bit of the two bytes are set to LOW.



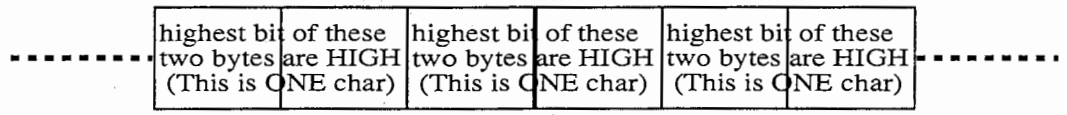
Korean Internal Coding

This is a three-byte coding method, similar to Japanese Internal Coding. Each three byte represent a Korean character. First byte is an *INTERNAL* header. One may check this header and detect the language type, because this header is different from Korean and German *INTERNAL* Codings.



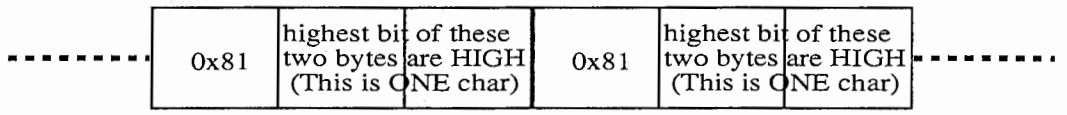
German Umlaut Coding

This is a one byte coding method used for the special Umlaut characters in German.



German Internal Coding

This is a two byte coding method used for the special Umlaut characters in German. The first byte of this coding method is an *INTERNAL* header. One can use this header to determine the language type, because the header is different from the Japanese and Korean *INTERNAL* header.



German GIS Coding

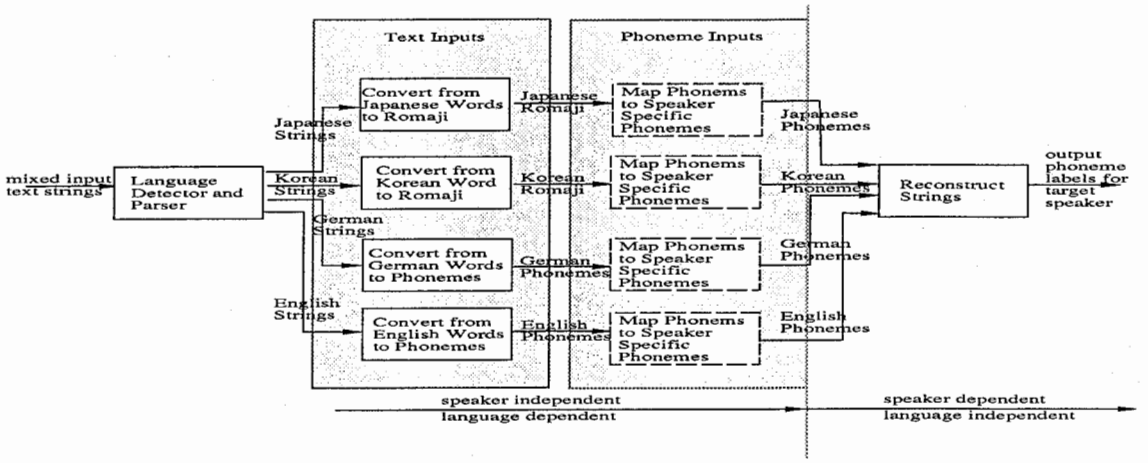
This is a two-byte coding method with a header and tailer. Similar to German Umlaut Coding, each two bytes will represent an umlaut. However, the highest bit of the two bytes will be set to LOW.

4.3 Compile conv_romaji

This program uses PMEM structure and PTOKEN structure from PTerm to handle the memory allocation problem. Instead of copying the file physically, links to the PTerm directory was made, these links should be present during compilation. If not, establish the link and compile again.

4.4 Program Structure (Programmer's Guide)

Basically, the program can be sub-divided into 4 parts: Language Detection and Parsing, Romaji Conversion, Phoneme Mapping and finally Reconstruction as shown below:



Language Detection and Parsing

This module takes mixed-language inputs and parse them into 4 different LISTS. Each LIST represents one language type and sequences of the string orders are also recorded for later reconstruction. There are special cases where the program will fail:

1. A German sentences without an Umlaut character in the sentence will be detected as a English sentence.
2. An English sentence, followed by a German sentence contains an Umlaut, will be detected as a German sentence.
3. A Japanese EUC-coded sentence uses only the character set in Korean EUC code will be detected as Korean.
4. Korean EUC-coded sentence followed by a Japanese EUC-coded/JIS-coded/Internal-coded sentence will be detected as Japanese.

Romaji Conversion

This module converts the inputs to romaji. Currently, converting from Japanese Kanji to Romaji can be done using KDD's KAN2ROM or KAKASI. KAKASI is much faster while KAN2ROM gives better intonation indicator in Romaji. Using KAKASI is much faster because of two reasons. First of all, KAKASI does not predict intonation of the given Japanese Kanji. Secondly, during Kanji conversion, for better performance, all tokens in the LIST are written to a file. Then, a single system called is made using KAKASI to start the conversion. After that, the Romaji is read by the program and the process continues. When KAN2ROM is used, KAN2ROM removes all the carriage returns. Because of this, it is not possible to read the Romaji back to the system and keep the order of the sentences as before. If this can be modified from KAN2ROM, Kanji conversion using KAN2ROM will be faster.

Korean and German are converted inside the program itself. No external programs are used.

Phoneme Mapping

This module takes the Romaji inputs and map the Romaji into phonemes of certain speaker. At first, simple 1-1 phoneme mapping would be used; however during the implementation and test, the quality of this mapping turned out to be very bad and therefore, better mapping scheme should be implemented in replace of the current one.

More detailed explanation and pointers will be given in later sections.

Reconstruction

This modules reconstruct the original text string order from the 4 LISTS. Since the order of the strings were recorded when the original text was parsed, reconstruction can be done quite easily.

4.5 Problems with Phoneme Mapping

The goal of this program is to produce a speaker specific phoneme strings which is independent of input language string. For example, if a user inputs a string in English and speaker_fmp559 is selected. This program will produce a phoneme set for e.g., speaker_fmp559 to produce a speech in English. At first, this program will use CHATR's Lexicon-look-up function to find the phonemes of the input English text string. This is quite simple. After that, each English phonemes will be mapped using a mapping table for the target speaker (speaker_fmp559 in this case). This method is simple; however, the synthesis output using such mapping can at times be so bad that it is not distinguishable. This is because CHATR does not only look up the database with that phoneme, but also tries to find the better matching neighbour phonemes. Because of the differences among languages, the phonemes that CHATR selects may be correct in one language but incorrect for another one. Several solutions have been suggested for this problem, but each will need some research.

4.6 Future Work

1. Fix KAN2ROM so that KAN2ROM does not remove the carriage returns when converting the input Japanese Kanji to Romaji. Once this is completed, change the program so that KAN2ROM is called only once during the Kanji conversion.

2. A better phoneme mapping mechanism is needed. Currently, the phoneme mapping across different languages gives unsatisfactory speech synthesis; although it is recognizable if the text of the speech is also provided. Perhaps the quickest solution is to relabel each database using the HMM models of the closest speaker of each target language. Another solution would be to use features rather than phone labels in the selection process.

3. For German and English, a dictionary is used to find the phonemes for a word. If the word

is not found in the dictionary, we will not be able to process the word. In CHATR, however, there is a module which predicts phonemes of any given word. Since CHATR's library files is already linked to the program, one should able to use such functions directly to predict the phonemes.

4. Finally, this program is meant to be integrated with CHATR so that, in the future, CHATR may take multi-language input without the user needing to specify the input language type. This means that the current CHATR input method will need to be upgraded because it is not possible to enter Japanese, Korean or German Umlauts using current CHATR's command-line interface.

5 Conclusion

A program, PTerm, usng stdin/stdout to connect software such as CHATR and TDMT was implemented. PTerm can also be used as an interfacing filter for CHATR. This makes future integration of the spoken dialogue translation system easier since input to CHATR can be transparent with respect to the entire system and can be adapted without requiring any modification to the CHATR code itself.

The importance of prosody for concise and detailed information transfer is well known. Handling prosody is not yet an easy task, but it is essential to demonstrate this feature in the proposed ATR-ITL spoken language translation system. Both the language translator and the speech synthesizer can thereby take advantage of prosody information to improve their current performance. Several interesting applications for CHATR using dynamic time warping techniques have also been proposed in this paper.

Finally, with the newly implemented auto-language detector and parser, CHATR is now able to process multi-lingual text input and to synthesize multi-languge output using a single-language single-speaker database. This program takes advantage of the machine-readable input text coding systems that contain language-identifying information.

While none of these implementations is yet claimed to be complete, we trust that they will make a significant contribution to Chatr's ease-of-use.