TR-IT-0268

A Speech Recognition Database Library

Thomas Wahl

1998 年 7 月 31 日

This Technical Report describes a C++ database object that supports experimental work in speech recognition. Difficulties that can make those experiments inconvenient include:

- the number of utterances is often very high,
- the waveform and feature data files can be very large, and
- utterances from different sources may have different formats.

The database is designed to be an interface that allows experimenting with many utterances in a standardized fashion without having to care about vast amounts of data that are present in form of the samples and (after preprocessing) the features.

Section 1 gives an introduction to the purpose of the database, including how to make it ready for use.

Section 2 explains its functionality in detail.

Section 3 contains a description of how to use scripting languages along with the database library.

Section 4 provides examples for using the library both within C++ and within scripting languages.

Section 5 gives some information on extensibility and maintainance, including a data structure description.

Section 6 finally is intended as a quick reference manual for important things that may get lost in a big paper, but may be needed quickly when working with the database library.

目次

1 Introduction

1.1 What is it for?

Speech recognition usually starts with huge amounts of data that represent utterances, which are intended as training or test material for a recognizer. An *utterance* contains some information on the circumstances when it was uttered (*text data*), like name of the speaker, his/her age, the technical facilities used to record it, and perhaps one or several transcriptions, i.e. some form of text representation of the spoken utterance.

A second component of an utterance are the actual speech data, called the *waveform*, which can usually be understood as a (high-dimensional) vector of an elementary data type. Each vector component stands for the data of some time in the course of the spoken utterance. Waveform data are often also referred to as *samples*, so in this manual.

The objective of the preprocessing stage of speech recognition is to perform various kinds of transformations and computations on the samples to make them a suitable input for a recognizer module. This process is known as *feature extraction*, the result of which, accordingly, is a *feature* containing recognition-relevant data (depending on the current task) of the original samples. The features are the third basic component of an utterance.

Due to the special circumstances, experiments in speech recognition usually require the capability to cope with large amounts of data, as not only the number of utterances in a test or training set may be very high, but also the samples and features associated with an utterance may consume considerable memory. The library presented here helps to overcome these problems. It is equipped with data structures and a memory management that is designed to handle large amounts of data efficiently both in terms of time and space. Thus, its main purpose can be described as "organizing utterance manipulation procedures", especially (but not only) for the preprocessing phase of speech recognition.

A usual way to use the database looks as follows:

Create a database object by filling it with utterances. The utterances – at this time – only consist of text data. These data usually come along with the training or test set that is used, but they may also be made up by the user. First, a *description file* must be constructed that contains these data in a format that is readable by the database. This format is extremely general; for details, refer to section 2.2.

Secondly, create an empty database and then submit the name of the created description file to the database, so it will read in the data and make (internal) utterances out of it.

(Optional) Select the utterances you actually want to work with. You might not want to use all of the utterances contained in the database, but only those that have properties like `Sex==female`, or you might want to combine certain utterances from several databases created previously. The library offers basic database operations to carry out those utterance selection steps, as exemplified in the abstract on the first page.

If you have done this, there is a function to produce a list of the ID's of all remaining utterances. This is useful, as many functions in the library require the specification of an utterance ID so as to perform some operation only on that utterance. In other words, this list acts as an interface between the user and the database: it enables you to refer to specific utterances.

You may now want to save the data so far contained in the database on disk, such that some time later they can be loaded, and work may continue.

Run experiments. You can now "play" with the utterances. The text data read in during creation of the database usually contain a filename that tells the database where to get the samples from ("waveform file", often a CD rom file, perhaps compressed). On user demand, these samples are read in from the file (and automatically uncompressed, if necessary). A copy is saved in main memory, another copy returned to the user. Hence, when you have run some manipulations on the samples and decide to throw it away and try again, the database will provide you with another copy.

At some time, the database object will be closed (end of program), and all main memory data are lost. Normally, next time you would have to read the sample data again from the original (CD rom) file, which

may be slow, and you may not even have the original file handy at that time. Therefore, it is possible to save the samples (uncompressed) into a file. Once the they are requested again, the database will automatically read them from this location.

You may now perform preprocessing or other operations with the samples of an utterance. For that purpose, a function package written by Mike Schuster (called `mat2D`) can be used easily with the database, but you may use your own functions. The result of the preprocessing are feature vectors, which – like the samples – can be saved in a file, such that they are available when the database is loaded at a later time.

Summarizing the operation of loading samples, there are three options for the library:

1. if the samples are currently in main memory, return a copy of contents of the appropriate address.
2. otherwise, if the samples are in the data file created by a save operation, read them from that file into main memory and go to step 1.
3. otherwise read the samples from the waveform file, possibly uncompress them, store them in main memory and go to step 1.

The same holds true for the features except that only steps 1 and 2 are available options.

Of course, main memory will usually be too small to read in samples and construct features of *all* utterances. Therefore, the database has a sample buffer and a feature buffer (the size of which the user can specify in bytes). If samples or features of an utterance are requested, it is first checked whether the buffer size will suffice to add the data demanded to the buffer. If so, they are simply loaded. Otherwise, the respective buffer is emptied, and starting from the utterance in question, consecutive samples or features are loaded into the buffer until it is full again. Thus, when the user browses through all utterances of the database in the order in which they are stored (and contained in the utterance ID list mentioned above) and requests the samples or features, they are not *all* being loaded into main memory at the same time (which would sooner or later overextend it). On the other hand, they are not loaded, worked with and thrown away one by one either, which would be too slow. System resources permitting, you may set the buffer size to a very high value, such that the reloading procedure won't happen very often.

Note that "loading/saving" a database just means loading/saving the text data and perhaps some filenames. Sample or feature data are not loaded or saved simultaneously, but have to be loaded or saved explicitly be the user (using special commands provided by the database). The reason is that storing all text data *and* all samples and features in one huge file (which potentially might comprise terabytes) often will not be possible. The text data on the one hand and the binary data on the other are always kept in separate files. Apart from the size-related reason, that should make handling these files outside the database easier.

When experimenting with utterances, it is often desired to just try out potential preprocessing steps and wait the results before continuing with other operations. Doing that exclusively with a compiler language would require the user to write a (rather small) program for every such step, compile it, and run it to try out. A much more convenient way is offered by *scripting languages*, which make it possible to execute functions from a C or C++ library in a command-line manner, i.e. as an interpreter language would do it. For this purpose, the library must be attached to the scripting language, such that single commands can be passed on to the library.

In the case of C++, this can be realized using the SWIG package, which comes up as an interface between C++ and a scripting language. For the present library, experiments using PYTHON as scripting language were run. Instructions and examples to do that are contained in the `swig` subdirectory and are explained in section 3.

The library was written in C++, as the structure of the problem specification suggested a hierarchical definition of the data types involved (see 1.3). It was tested using the g++ compiler versions 2.7.2.2 and 2.8.1 on different platforms (including a DEC architecture, personal computers running Linux and HP machines with the HP-UX operating system).

For large-scale tests, the Wall Street Journal waveform file collection on CD rom served as testing material.

1.2 Installing and using the library

After obtaining the file `database.tgz`, it is unpacked using

```
tar xzf database.tgz
```

(`tar` here stands for the appropriate GNU tar command on your system.)

To install, just follow the instructions in the created `README` file. It also contains a description of the package's file structure.

This file also tells how to compile the library. Beside the actual database functions, the library also contains the code of a string utilities tool and the matrix manipulation library `mat2D` mentioned earlier.

Linking the database to other code is accomplished by passing

```
-ldatabase
```

to the linker when making an executable.

If you want to make a shared library, you have to check how to do it on your system. Some support the option `-shared` for g++, others use the `ld` command directly, employing option `-b`. In the first case, the Makefile can create the shared libraries for you; consult the `README` file.

1.3 Basic structure of a database

The basic type hierarchy (all types are `classes` or `structs`, see section 5.1) within a database object is as follows:

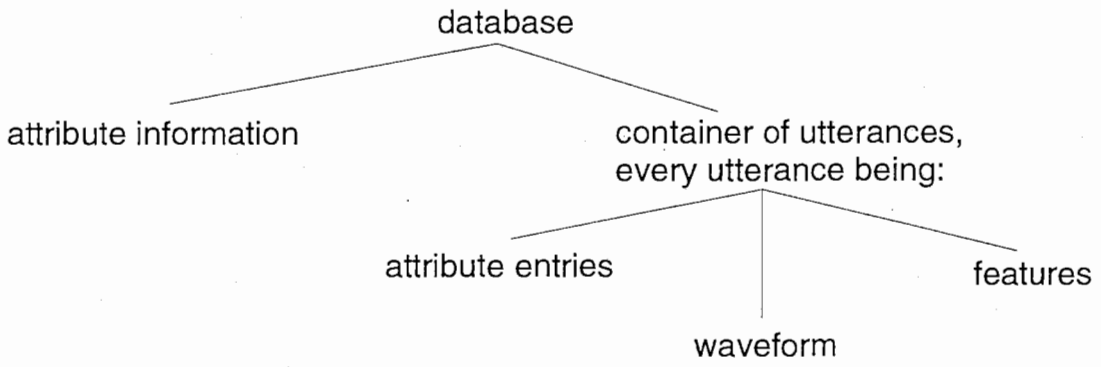

Figure 1: Structure of a database object

The basic data structure is a database. This is the class that the user can make objects of and that provides all the methods to manipulate such an object. All other classes are used only internally as sub-structures of the database class.

Most importantly, a database is a container of utterances (right side of the figure). Each utterance in turn has, as mentioned above, three (logical) components:

- the text data, containing basic information on the circumstances of the utterance. These are stored in *attribute entries* of the shape `(Speaker,John)`. Here, `Speaker` is an attribute, `John` is the corresponding entry.
- the waveform (mainly containing the samples of the utterance)
- the feature.

The attribute information (left side of the figure) contains all attributes and entries that occur in the database and is not of much importance to the user.

Further, a database instance contains a `name`, which is the default filename for load and save operations, and a `waveform_path`, which is used as a common path for waveform files, see section 2.6.1 for details. These members are in this documentation referred to as *parameters*.

1.4 Error handling

The files `include/error.hh` and `source/error.cc` furnish information on error handling. Throughout the library functions, excessive checking of user input and displaying appropriate warnings or error messages was implemented to facilitate the analysis of errors. This includes parse errors when reading the contents of files, but also filename mistakes, missing read or write permission, etc.

There are three types of library messages to the user. They can be understood as "error messages of increasing severeness":

messages will be issued, if there is just a notification to the user about something that might hold up the database for some time or that doesn't effect the user's work but seemed worth telling.

warnings will occur, if some user input does not seem reasonable, but will not lead into processing problems. Therefore, warnings usually don't interrupt the program.

Sometimes, however, it is hard to decide whether an interrupt or some form of continuation is the proper way to respond. This is the case, for example, when a continuation would imply to erase lots of the database contents. In such instances, a warning along with a request what to do is issued, such that the user has to decide on-line. This may cause undesired behavior if the output of some database function is redirected into a file. Therefore, warnings with requests are used sparingly throughout the package. – "Interrupt" in this case means `return` to the calling program.

errors occur whenever some user input cannot be interpreted in a way that allows a continuation of the program. In this case, the C++ typical error handling is launched, i.e. an exception of type `database_error` is thrown. The user now has three options:

1. `return` from the database function safely to the calling program (may be `main` or a scripting language),
2. `throw` the exception again. In this case, the calling program will have to catch the error; otherwise, the behavior is undefined; or
3. `exit` the program. This normally also kills the calling program, so this is only the last resort.

There is no `exit`, `abort` or a similar call that promptly kills the process created by the program unless the user wishes so in case of an error message. – All kinds of messages are always sent to `cerr`.

Changing the error handling is easily done just by altering the two files mentioned above.

2 Functions

The following part of the manual describes all available functions in detail (refer to table 1 on page 5 for a complete listing). All the methods mentioned are members of the `database` class and therefore declared in the file `include/database.hh`. The prefix `database::` is omitted in the documentation. – Functions that belong to other classes/structs used within the database source code are not supposed to be called by the user and are not explained.

2.1 Creation, assignment, and clearing

A database object can be constructed as follows:

Section	Page	Declaration	
2.1	4		database()
		database&	operator=(const database& source)
		database&	assign_from(const database& source)
		string	set_filename(const string& new_filename="")
		void	clear_data()
		void	clear()
2.2	6	void	add_from_string(string& description)
		void	add_from_file(const string& filename)
		vector<string>	get_utterance_list()
		vector<string>	get_utterance_list_if(const string& condition)
2.3.1	8	string	get_string_entry(const string& utterance_ID, const string& attribute_name)
		float	get_number_entry(const string& utterance_ID, const string& attribute_name)
		void	set_attribute_entry(const string& utterance_ID, const string& pair)
		void	set_attribute_entry_of_all(const string& pair)
2.3.2	9	void	add_from_database(const database& source)
		void	add_from_database_if(const database& source, const string& condition)
2.3.3	10	void	remove_utterance(const string& utterance_ID)
		void	remove_if(const string& condition)
		void	remove_if_not(const string& condition)
		void	remove_if_present(const string& attribute_name)
		void	remove_if_absent(const string& attribute_name)
.2.4.1	10	void	show_parameters()
		void	show_attributes()
		void	show_utterances(int in_a_row=0)
		void	show(int in_a_row=0)
2.4.2	11	long	show_main_memory_usage()
		long	main_memory_usage()
		int	get_number_of_attributes()
		int	get_number_of_utterances()
2.5	12	void	save(const string& filename="")
		void	load(const string& filename="")
2.6.1	13	_1Dchar*	get_char_sample(const string& utterance_ID)
		void	save_char_sample(const string& utterance_ID, _1Dchar* new_sample=0)
		void	unsave_char_sample(const string& utterance_ID)
		_1Dshort*	get_short_sample(const string& utterance_ID)
		void	save_short_sample(const string& utterance_ID, _1Dshort* new_sample=0)
		void	unsave_short_sample(const string& utterance_ID)
2.6.2	15	_2Dfloat*	get_feature(const string& utterance_ID)
		void	save_feature(const string& utterance_ID, _2Dfloat* new_feature=0)
		void	unsave_feature(const string& utterance_ID)
		vector<string>	garbage_collection()

Table 1: Complete list of functions

`database()`

constructs an empty database (i.e. a database with no utterances), sets its filename to `./noname.dtb` and its waveform path to the empty string.

`database& operator=(const database& source)`

`database& assign_from(const database& source)`

are identical in their behavior and make the target database a copy of `source` (all previous data in the target are destroyed). "copy" means that also the parameters (filename, waveform path) of source and target will have the same values afterwards. – The reason for introducing the second function is that in some environments, overloaded operators like `operator=` cause trouble and cannot be used[1]

Beside these copy assignment functions, a copy constructor is also defined which creates a new database as a copy of another.

`string set_filename(const string& new_filename="")`

If `new_filename` is "" or not specified, the function returns the filename currently associated with the database object.

Otherwise, the database filename is set to `new_filename` and returned. This may or may not contain an extension; see sections 2.5 and 2.6 for the precise meaning of this filename.

`void clear_data()`

clears everything in the database but its name, i.e. it clears all utterances and, hence, the attribute information. In particular, this function removes all samples and features from main memory. The waveform path is reset to "".

`void clear()`

resets the name of the database to `./noname.dtb` and calls `clear_data()` afterwards. The database is now in the same state as it would be right after definition using the parameterless constructor.

2.2 Filling the database

The following two methods provide the means to fill an empty database (but they can also be used to add attribute entries to a non-empty database):

`void add_from_string(string& description)`

`void add_from_file(const string& filename)`

They read in utterance text data, make utterances (without samples or features at this time) out of them and write them into the database. The description string (first function) and the description file (second) have exactly the same format requirements. The only important difference between the functions is that the description string is destroyed in the first function (due to parsing reasons), whereas the contents of the description file is not affected in the second. – The format is as follows:

1. The input must be line-oriented, with every line standing for one utterance. Within one utterance, a newline character is not allowed, unless the attribute or entry name it appears in is surrounded by "".

[1] To be precise, this problem occurs when the library functions were connected to PYTHON using SWIG as an interface between C++ and PYTHON.

2. Every utterance line consists of attribute entry pairs, which are separated by a comma.
3. Every attribute entry pair has the form

   `<attribute name> = <entry>`

   with arbitrary spacing between the three fragments. Using == here instead of = will lead to a parse error.

There are two types of attribute: those containing string values and those containing numerical values. They are distinguished by the attribute name using the following convention: if the initial character is small or a digit, the attribute is number-valued, and string-valued otherwise. So, for example,

`age` and `2nd-employment-year`

are number attribute names, whereas

`Speaker`, `DATE` and `_place_of_recording`

are interpreted as string attributes. The distinction between the two attribute types determines the way of evaluating its entries: in an expression like `destination = Paris`, `destination` would be assumed a number attribute and `Paris` would be reduced to zero as its numerical value[2]

Some attributes, called *special attributes*, are reserved for internal use in the database. They are related to properties that every utterance has and therefore are always present. A complete list of these names is given in the reference section 6 at the end of this manual. Except the fact that their meaning cannot be changed by the user, there are no restrictions in the usage of these attributes, i.e. their values can (and sometimes must) be set, be read, etc. When the user wants to refer to the values of special attributes, it is essential to use their particular names so that the parser can recognize them.

One of these special attributes, named `Relative_path`, is of particular importance. It contains the relative path (in the easiest case, just a filename) of the (original) waveform file, from which the utterance's samples are read (for details, refer to section 2.6.1). This value is believed to be unique among the utterances, and is therefore taken as the ID of an utterance. In other words, the entry of this attribute is the one for the user to refer to a specific utterance in the database, for example, when calling utterance-specific functions. Without this ID, an utterance can only be accessed via loops that go over *all* utterances, but never individually. Therefore, specification of this ID in the description string or file is considered to be essential, which is why a warning is issued if it was not supplied.

Apart from these conventions, any names are allowed for attributes and entries and may even contain special characters including =, `\n`, spaces, etc., but then have to be enclosed in "". If the " character itself is supposed to occur an attribute or an entry (for example, in a transcription), it has to be canceled using \:

```
Transcription = "Yesterday, \"That is what the world is waiting for\",
a British trader said"
```

Summarizing, the string (or the file, respectively) takes the following shape:

```
<attr11> = <entry11> , <attr12> = <entry12> , <attr13> = <entry13> [ , ... ]
<attr21> = <entry21> , <attr22> = <entry22> [ , ... ]
<attr31> = <entry31> , <attr32> = <entry32> , <attr33> = <entry33> [ , ... ]
```

[2]The reason for this strict kind of attribute type assignment is that in expressions like `WWW-address = 141.35.2.80`, the entry must be prevented from being evaluated as a number. Recognizing the type of an attribute only from the entry value would require writing a non-trivial parser.

Whenever an attribute occurs a second time within one line (i.e. within one utterance), the value of the entry is simply overridden. For example, of the following two description lines,

```
Day = Friday , day = 13
Day = Friday , Day = 13
```

the first one is perfectly OK, whereas the second not only loses the `Friday` entry, but the `13` is stored as a string and cannot be used in numerical comparisons, like `Day < 13` for dates before the 13th or such.

The order of the attribute entry pairs within one line is equally unimportant as the order of the utterances (they will be sorted anyway). Further, the file may contain empty lines between utterances.

When applied to a non-empty database, only utterances that are not yet in the database are added, so no duplicates are produced (duplicates are, of course, recognized by having the same ID).

`vector<string> get_utterance_list()`

Returns the ID's (see above) of all utterances in a vector of (C++) strings. When you want to call an utterance-specific function with all utterances consecutively, you just browse over that vector and call the function with the current utterance ID.

`vector<string> get_utterance_list_if(const string& condition)`

Returns the ID's of all utterances that satisfy the `condition`. The condition string must be specified in a fixed format, which is explained in section 2.3.2 on page 9, along with the `add_from_database_if` function.

2.3 Standard database operations

2.3.1 Attribute entry manipulation

The following functions all refer to a specific utterance via its ID and read or write attribute entries in it:

`string get_string_entry(const string& utterance_ID,const string& attribute_name)`

`float get_number_entry(const string& utterance_ID,const string& attribute_name)`

These functions return specific values of entries belonging to the utterance pointed to via ID and to the attribute referred to via its name. This name must refer to a string attribute, in case of the first function, or a number attribute, in case of the second. Otherwise, an error occurs. This will also happen if the utterance ID or the attribute don't exist in the database or if the attribute doesn't occur in the utterance.

`void set_attribute_entry(const string& utterance_ID,const string& pair)`

This function sets attribute entries according to the information provided in the `pair`. The format of it very much follows the conventions in description files or strings, as described in section 2.2. That is, it must look like

```
<attribute name> = <entry>
```

Please refer to all remarks made about the exact writing style on page 7.

The function first searches the utterance specified by the ID in the database. If found, it searches through the attribute entries of it. If the attribute specified in `pair` already exists in the utterance, its value is changed according to the new entry. If not, the attribute entry pair is added to it.

`void set_attribute_entry_of_all(const string& pair)`

Does the same as the previous function, but here, in all utterances the same attribute entry pair is set/added. This is useful only for utterance-independent data, for instance

```
set_attribute_entry_of_all("Microphone_type = Sennheiser")
```

An error is reported if the user tries to set an attribute in any utterance that already exists, but with different type: you cannot change string into number attributes or vice versa using this function.

2.3.2 Adding utterances from one database to another

The following two functions allow to merge databases by adding utterances from a source database to the object. Note that these functions only make sense if the `waveform_path` variable in both databases is identical. Otherwise, there will be utterances in the target object whose waveform files cannot be found on disk (see section 2.6.1 on page 13 for details on how this path is derived). – Therefore, if the waveform paths are set in both databases and differ, a warning is issued, and a `warning_with_request` whether to continue or not is made (see section 1.4 on page 4).

`void add_from_database(const database& source)`

adds all utterances from `source` to the target, if not in yet. This function represents the classical merge routine for databases. The attribute information of the target database must be updated, which might take some time. Therefore, the number of processed source utterances is displayed during the merge.

`void add_from_database_if(const database& source,const string& condition)`

adds the utterances from `source` to the target, if not yet in and if the condition specified in the second argument is satisfied. This condition string must take the following form:

```
<attribute name>   <compare operator>   <entry>
```

As usual when submitting attributes or entries, pay attention to attribute types and special characters that might be contained in the string. See section 2.2 for details. Between the three components, spaces are ignored.

As compare symbol, the following binary operators are recognized:

```
==      !=      <      <=      >      >=
```

but only the first two are allowed in expressions involving string attributes. A single = causes a parse error when used in conditions.

The notion of fulfillment of a condition becomes a bit vague when attributes occur in some utterances, but not in all. How these cases are handled in this database implementation, is shown as follows:

Let U be an utterance and C = `<attr> <comp> <entry>` a condition string.

U satisfies C $\iff$ U contains `<attr>` *and* C evaluates to `TRUE`
U does not satisfy C $\iff$ U does not contain `<attr>` *or* C evaluates to `FALSE`

This way it is ensured that "fulfillment" is the logical negation of "non-fulfillment". For example, in the statement

```
d1.add_from_database_if(d2,"age<24")
```

all utterances of `d2` that contain the attribute `age` with value less than 24 are added to `d1`. Utterances in `d2` that don't have an `age` attribute are ignored.

2.3.3 Removing utterances

The following functions remove certain utterances from a database, either by specifying it explicitly via ID, or logically via a condition.

`void remove_utterance(const string& utterance_ID)`

removes the specified utterance from the database, or issues an error if not found.

`void remove_if(const string& condition)`

`void remove_if_not(const string& condition)`

removes all utterances in the database that satisfy (don't satisfy, in case of the second function) the specified condition. The condition string must follow the conventions given in section 2.3.2 above. It is important to understand how these functions affect utterances that don't contain the attribute mentioned in the condition. It follows logically from the definition of "fulfillment" given in the previous section. Look at these examples:

`d1.remove_if(d2,"age<24")` and `d1.remove_if_not(d2,"age>=24")`

These two calls don't perform the same action, although it might seem so: Be U an utterance that doesn't contain the `age` attribute. By definition, U does not fulfill the condition `"age<24"`, as a condition is not satisfied if the attribute is not there. So it is not removed in the first call. For the same reason, U does not satisfy the condition `"age>=24"`. Hence, U *will* be removed by the second call.

`void remove_if_present(const string& attribute_name)`

`void remove_if_absent(const string& attribute_name)`

removes all utterances that contain (don't contain, in the case of the second function) the attribute, regardless of its entry. If all utterances contain the attribute mentioned, then the first function call would mean to clear the entire database. As a precaution, a `warning_with_request` is issued. The second method, in this case, doesn't have anything to do, but a message is displayed as to inform the user.

2.4 Show routines

There is a variety of functions to display information on the contents or the memory usage of the current database object. Target of the output is always `cout`.

2.4.1 Showing database contents

`void show_parameters()`

shows name and waveform path, as well as the number of distinct attributes and entries, and the number of utterances that the object currently has.

`void show_attributes()`

for every attribute that currently occurs in the database, shows its name, its ID, its type (string or number) and the *occurrence frequency*, which specifies in how many utterances the attribute appears.

`void show_utterances(int in_a_row=0)`

for every utterance that currently occurs in the database, shows:

1. the list of its attribute entries. This comprises, for every attribute entry, the attribute name, its assigned ID, and its entry value. If the entry is of type string, then it has an ID as well, which is then displayed behind the entry.
2. its waveform parameters. This includes all the attributes characterizing the waveform (see section 2.6.1 for details), in particular, the relative waveform path, which is the utterance's ID. Further, information on the location of the utterance's waveform samples is provided, namely whether they are in main memory or not, and whether they had been saved to a specific data file. (If both answers are `NO`, it means that the waveform is only available via its original file [e.g., on CD rom].)
3. information on its feature vectors, namely whether they are in main memory or not, and whether they had been saved to a specific data file. (If both answers are `NO`, it means that no features are currently available for this utterances.)

A database object normally has a huge number of utterances, often several thousand. In this case, listing all of them in a row is not desired. Therefore, you may specify a value in the `in_a_row` argument that makes the printout of utterances wait after this number of utterances have been printed. You may then hit the `<RETURN>` key to continue the printing or hit `<q>` plus `<RETURN>` to interrupt the output and leave the `show` function. – Default value of this argument is `0`, which means, showing all utterances without pausing.

```
void show(int in_a_row=0)
```

shows the parameters, attributes and utterances of the database object (in this order) by calling the functions mentioned above consecutively. The `in_a_row` parameter is passed to the `show_utterances` function.

2.4.2 Showing memory needs

The following two functions give information on the main memory requirement of the database instance.

```
long show_main_memory_usage()
```

displays the main memory that is currently used by the database parameters, by the container of all attributes and the container of all string entries (attribute information), and finally and most importantly, by all utterances. The latter includes memory consumed by samples or features of an utterance, if any. The current number of attributes, string entries and utterances is also shown. At the end, the total main memory usage of the database object at this time is shown and returned by the function.

```
long main_memory_usage()
```

also returns, but doesn't display the current total main memory usage of the database object.

Note that the binary data, i.e. the samples and features, are not saved in the same file as the text data of the database. Therefore, the value returned by the functions above can only be taken as an approximation of the save file size (i.e. on disk), if the database currently contains no samples and no features.

```
int get_number_of_attributes()
```

```
int get_number_of_utterances()
```

return the number of attributes that occur in the database (not regarding in how many utterances it does), and the number of all utterances.

2.5 Save & load

As mentioned before, these functions load and save only the text data of a database, i.e. everything but samples and features. For those, special instructions must be employed. The idea is that the binary data in a database can make the save file really huge, which is why you may want to split them among several files.

`void save(const string& filename="")`

saves the text data of a database object.

`void load(const string& filename="")`

loads the text data of a database object. If applied to a non-empty database object, the object is cleared before without warning.

As saving and loading a large databases may take some time, the percentage of currently processed utterances is displayed while accessing the target or source file.

For both functions, the filename where to save/load the database to/from, is obtained in the following manner:

- If `filename` is "", the value of the database `name` parameter is used as filename.
- Otherwise, the functions first set the `name` parameter of the database to the filename passed as argument, and take this argument as filename for loading or saving, respectively.

In other words, the two instruction sequences

(1) `d.load(name_string)` and
(2) `d.set_filename(name_string) ; d.load()`

have the same effect.

2.6 Waveform and feature handling

Waveforms (consisting of samples) and features represent the raw and preprocessed speech data of an utterance, respectively. Their handling has been detached from the one of the other data, for two reasons:

(a) They form the major part of the utterance, in the sense that on these data, the feature extraction routines are supposed to be run. Therefore, they must be passed to preprocessing libraries that offer the functions for the extraction process.

(b) They are binary and form the largest part of the memory consumed by the utterance. Therefore, they often cannot be kept with the text data in one file for simple resource reasons.

The proposed way to do the preprocessing is using the `mat2D` package compose by *Mike Schuster*. For this reason, the samples and the features of an utterance are stored in data types taken from this library. The definitions of those data types are contained in the database library, so that they are always available.

The internal way of getting the binary data of an utterance had been described earlier in section 1.1 on page 2. In brief, the data are always loaded from the "cheapest" (in terms of access time) medium they are currently on. The user does not have to care about this. Therefore, there is just one `get` function for each of those data types.

Secondly, there is a respective function to save the data. Saving here means to store them on a file on your private disk. As for the sample date, this way you can get rid of the original waveform data (for example, on CD rom), which might not be convenient to access, and which might be slower than disk (in the case of CD rom).

2.6.1 Waveform handling

On page 7, special attributes were mentioned which have a fixed meaning in every utterance. In the current version, almost all of them happen to be related to the waveform of an utterance, namely (remember that capitalized names refer to string attributes):

`Relative_path` The original sample files are accessible via a filename, which must be stored with the respective utterance. These filenames, however, often share a more or less long part at the beginning, for example, `/cdrom/wsj0/si_tr_s`.[3] To save memory, this common path should be stored only once, and this happens in the `waveform_path` variable, as mentioned before. The rest of the path, which characterizes each single waveform, should be the contents of the `Relative_path` attribute. For its uniqueness, this path (filename) is used as utterance ID and therefore should be stated at the time of the creation of the utterance, as was explained in section 2.2.

`Format` Original format of the waveform when stored on CD rom or such. Currently, the following formats are recognized:

```
NIST            RAW_CHAR            RAW_SHORT
```

So if, for example, your waveform test set comes in raw 8 bit (= `sizeof(char)`) data, you should specify this by having the statement

```
Format = RAW_CHAR
```

in the description file when creating the utterance.

`sampling_frequency` Frequency of the recorded samples in Hz.

`number_of_samples` Total number of samples in the waveform file.

`bytes_per_sample` Size of one individual sample. Currently, only `sizeof(char)` and `sizeof(short)` are accepted. Accordingly, there are char samples and short samples for utterances.

`Order_of_bytes` Only relevant in case of short samples. In C and C++, a short has at least two bytes. The order of storing those bytes (for one particular sample) can be reversed on different architectures and is referred to as *low byte first* or *high byte first*, resp. This string variable indicates this by the values `0` or `1`, resp.

`Char_sample_file` Name of a file to which the char samples of the parent utterance are to be stored. This variable defaults to `-`, which means: Take the database `name` variable without filename extension and append `.wvf`[4] to it to get the filename. Note that as long as this variable has its default value, all char samples of those utterances will be stored into the same file.

When databases are merged, the source database's `name` variable is lost, which is way all their default data filenames have to be changed from `-` to the explicit name of the source database. This requires more storage. If the user decides to rename those files to the name of the new (target) database plus extension `.wvf`, he can explicitly assign the value `-` to those filenames, indicating that they now have the default name, saving (a lot of) text data storage.[5]

`Short_sample_file` Same for short samples.

The specification of some of these attributes is important for the `get_..._sample` functions explained below.

In the database, char and short samples are stored in vectors of type `_1Dchar*` and `_1Dshort*`, resp. The type definitions are taken from the `mat2D` matrix library.

[3] This is the common path for all samples of the Wall Street Journal test and training data on CD rom.

[4] This is the value of the `_waveform::extension` static variable.

[5] All this weakly assumes that `-` is never used as a real filename.

```
_1Dchar* get_char_sample(const string& utterance_ID)

_1Dshort* get_short_sample(const string& utterance_ID)
```

As mentioned earlier, these functions try to load their respective samples for the utterance specified from the fastest medium that they are on:

1. If the sample is in main memory, make a copy of it and return the address of the copy.
2. Otherwise, check if the samples had been saved to a data file before. If so, load them into main memory and go to step 1.
3. Otherwise, check the waveform file the name of which is given by `database::waveform_path` + `relative_path`. If this file contains samples of the correct type, i.e. as specified in the function name, load them into main memory and go to step 1. Otherwise, return a warning indicating that getting the samples failed.

It is crucial to note that the database never returns a pointer to an object that it administers itself, but only a copy of it. This has two reasons:

- ▷ Allocating and freeing memory resources of internal data structures can thus not be affected adversely by the user (the database always knows what to free and when)
- ▷ (Mainly for experiments) When the results of some processing step with the samples was not as desired, then it is easy to re-try by requesting the data again from the database.

On the other hand, this way of implementing it also makes it essential that the user takes care himself of the pointers returned by the database, in other words, he/she is responsible for freeing those. You must remember the addresses returned by the database and free them after use, otherwise, memory will eventually overflow.

In the third step of the above enumeration, the database may need some help by the user to load the samples from the original waveform file:

1. The `Format` variable must be set before data can be loaded from an original waveform file, because it influences the way of loading:
2. (a) If the format is `NIST`, nothing else (except for the two path variables above, of course) needs to be specified. The `NIST` format is highly standardized, and the database can read all the information from a header that precedes each sample file. This header also contains information on whether the samples are stored in compressed form. If so, the database will try to uncompress them using the command given by the `_waveform::shorten_decode_command` static variable and a temporary file given by `_waveform::tmp_filename`, and load them afterwards. – `NIST` samples can be char samples or short samples (which is automatically detected).
   (b) If the format is `RAW_CHAR`, the attribute `number_of_samples` must be set.
   (c) If the format is `RAW_SHORT`, the attributes `number_of_samples` and `order_of_bytes` must be set.

As has been mentioned previously in section 1.1 on page 2, whenever samples have to be loaded (no matter whether from the original waveform file or from the database data file), is taken care that the sample buffer does not overflow. If necessary, the buffer is freed, and concurrent samples (including the one that has been requested) are loaded until the buffer is full.

`void save_char_sample(const string& utterance_ID,_1Dchar* new_sample=0)`

`void save_short_sample(const string& utterance_ID,_1Dchar* new_sample=0)`

If `new_sample` is specified and not `0`, the contents of this pointer is copied into the database's respective main memory position for the sample. If the user here delivers an address that does not point to a sample of the respective type, the program might crash with a memory fault immediately or later when this contents is to be read.

In any case, the contents of the database's respective pointer for char or short samples is written to the data file. This file's name is the contents of the `Char_sample_file` or `Short_sample_file` attribute, as illustrated above. If there are currently no samples in main memory, a message is displayed.

This way of loading and saving samples assures that the samples in main memory and in the data file are always the same, unless the data file samples have not yet been loaded into main memory at the beginning.

`void unsave_char_sample(const string& utterance_ID)`

`void unsave_short_sample(const string& utterance_ID)`

For their respective samples,

- Mark the samples as unsaved, i.e. the database assumes that it cannot read the data from a user data file next time. Hence, if the data were saved before, those savings are now obsolete and will be removed upon the next *garbage collection* (cf. section 2.6.3).
- Remove the data from main memory.

Again, this way there will be no inconsistency between the data in main memory and the data saved by the user to a file.

The purpose of these functions is to get rid of waveform file data, if they are not needed any more for some reason, or, even worse, if the previously saved file data are somehow destroyed or lost. In this case, the database will be in an inconsistent state until you have "informed it of the loss" by claiming the samples unsaved using the above functions.

2.6.2 Feature handling

The features of an utterance are much easier to handle, as there are only two possible sources for them (main memory or a database data file), and there is only one type of features: `_2Dfloat*`, which is again taken from the `mat2D` matrix library. This type describes a two-dimensional vector (matrix) of float entries. Further, there is exactly one special attribute associated with features:

`Feature_file` Name of a data file to store the features to (and read them from). The same comments apply as given above for the `Char_sample_file`, except that the default extension here is `.ftr`[6]

`_2Dfloat* get_feature(const string& utterance_ID)`

tries to load features for the given utterance from a data file into main memory, if not already there, and returns a copy of the data in a new pointer which must be freed by the user. If features are neither in memory nor on a file, a message is issued and `0` returned. – All comments from the corresponding waveform section above apply accordingly.

[6]This is the value of the `_feature::extension` static variable.

```
void save_feature(const string& utterance_ID,_2Dfloat* new_feature=0)
```

"saves" the feature to a data file. The meaning of "save" is analogous to the previous section about saving samples.

```
void unsave_feature(const string& utterance_ID)
```

"unsaves" the feature from data file and removes them from main memory. See the corresponding sample function for details.

2.6.3 Cleaning up data files

```
vector<string> garbage_collection()
```

As experimentation proceeds, there may be many times when the user saved the samples or features for one specific utterance. Such a save operation always writes its data to the end of the data file and makes a note about the position in the file. Previous instances of these samples or features are not overwritten or removed. This is because such an update would require to rearrange the entire data file in order to close gaps, etc, which probably would hold the save routine up at an unacceptable degree.

The negative effect of this is that the file size may grow steadily even if the user only saves data of one single utterances, and those files contain a lot of garbage. Therefore, the above routine can be used to clean those files. It browses through all data files and reduces them to the currently samples or features. A list of all the files containing the valid binary data is returned. All other files that may have been created by the database earlier are entirely "garbage" and may be removed by the user. – Note that the database does not know about those files and will do nothing about them. The user has to remove them himself.

The garbage collection function is public and can be called by the user at any time, but it is not called automatically due to its potentially time-consuming behavior. It is his/her responsibility to make sure not to waste too much memory by allowing the garbage in files to become too large.

3 Scripting Languages

As mentioned before, one goal while writing this database library was to make it useful especially for experimentations. For practical work, it is not very convenient to write a C or C++ program "around" function calls, translate it, hopefully have no errors in it, and run it. If the experimental result is not as desired, you will have to change your `main` program, re-translate, and so forth.

Of real benefit would be a system that combines the run-time efficiency of compiler languages with the trial-and-error way of experimentations in case of interpreters. This is possible with C or C++ libraries using *scripting languages*. The idea is as follows:

Assume we have a C or C++ library. Probably, there is one or are several (header) files that represent the interface to the user, i.e. all functions from the library that can be called. Put them into one interface file.

1. For every function in the interface file, write a little C function around it, such that, when compiled, it becomes an executable, reading its arguments from a command line (similar to a `main` function in C). This process is called *wrapping* the C functions.

2. Compile the wrapped functions, and finally

3. link them together to make a new library, which is readable by your target scripting language.

The first step can be done by hand, but as it is highly automatic, a package called SWIG exists, which writes wrapper functions for each C or C++ function. The second and third step can usually be accomplished by a C or C++ compiler.

For the present database library, the above steps were tried out using SWIG as the wrapper generator and PYTHON as the target scripting language. In this case, you should now tell it where to find the PYTHON library created in step 3. Then you may either call PYTHON and create objects on-line, work on them and do real experiments, or you run a script file that contains the PYTHON instructions line by line.

The `swig` subdirectory in the library package contains the interface file `swig_database.hh` mentioned above, more precise instructions how to process it, a makefile to actually do it, and many examples to see how it works. The `database.i` file is another input file for SWIG (beside the interface file), which tells SWIG the name of that interface file, and gives it various instructions about argument passing.

As it took quite a while to make the database work from PYTHON, some comments on problems that occurred seem advisable. The SWIG wrapper generator was originally written for C, and later enhanced to allow for C++ features. This process, however, seems neither completed nor absolutely flawless yet. All problems that could not be solved seem to be attributable to SWIG rather than PYTHON. Those include:

- SWIG currently does not yet support all C++ features, in particular, operator overloading. This is the reason for many inconveniences, like different functions for conditional and unconditional operations, as in the case of the `add_from_database` and `add_from_database_if` routines. Also, the assignment operator = can not be used between database objects, which gives rise for the odd `assign_from` operation.

- In contrast to the SWIG manual, private declarations in the interface file are not ignored, as they should be. Various errors occur when SWIG parses the private section of this file. As a consequence, it is not possible to use the original `database.hh` header file also as interface file (which would be much more convenient), but rather a separate file has to be written that contains only the public declarations of the database class.

- Constant default values for string arguments (as "", in particular) in functions are not correctly handled, which is why global variables had to be introduced for such default values.

- In accordance with other SWIG-and-PYTHON users, the passing of C++ exceptions from a library (via SWIG) to PYTHON does not work properly (they are ignored in PYTHON). The inelegant consequence is that currently, each single database library function has to catch errors itself instead of passing them on to the calling program (which causes no trouble in the case of a C++ `main` function).

4 Examples

The following section contains example scripts that are essentially identical to those in the `example` and `swig` subdirectories that come with the library package. So it is easy to try them out rather to read them in this manual. – The two subsections each contain a C++ and a PYTHON example, which are identical in their effect, so you can compare the different way of stating instructions in either language.

4.1 Basic example

The following example (cf. `example1.cc`) shows a very elementary progression of how to use the library:

```
(1)  #include "database.hh"
(2)
(3)  main() {
(4)    database d;                                          // Make empty database
(5)    d.add_from_file("description/descriptions.txt");     // Fill from descr. file
(6)    d.show(1);                                           // Show database
(7)                                                         // Do something
(8)    d.save("database/example1.dtb"); }                   // Save database to file
```

Note that only the `database.hh` header file needs to be included (line 1). All other structures exist only logically for the user.

You start by creating an empty database (line 4). At the beginning, the only possible source to fill the database from is a description file (line 5). The argument 1 in line 6 means, output one utterance in a row, after each of which a confirmation by the user is requested.

The featureless line 7 may include extracting some utterances of the ones now in the database, but mainly this is the place to work on the samples and features and possibly save them to a file. This always affects only the binary data for one specific utterance. Therefore, in line 8, you probably want to save the text data of the utterances. This saves exactly what had been read from the description file (unless some text data have been changed meanwhile), but in the database-internal format, of course.

A PYTHON script file (cf. `example1.py`) performing the same actions looks like this:

```
(1)  #!/usr/local/bin/python1.5
(2)
(3)  from database import *
(4)
(5)  d=database()                                          # Make empty database
(6)  d.add_from_file("../example/description/descriptions.txt")   # Fill from descr. file
(7)  d.show(1)                                             # Show database
(8)                                                        # Do something
(9)  d.save("../example/database/save_example.dtb")        # Save database to file
```

Here, line 1 tells the calling environment (shell) that the file should be executed by PYTHON. Line 3 makes PYTHON read in all the database functions that had been compiled into the specific database PYTHON library before. The rest is basically the same as the C++ code above. Note, however, that the correct way to create an object (line 5) differs a bit from the way in C++ (line 4 in the previous listing).

4.2 Advanced example

The following script (cf. `example2.cc`) briefly demonstrates the usage of basic database operations and the way errors are handled:

```
(01)  #include <iostream.h>
(02)  #include "database.hh"
(03)
(04)  main() {
(05)
(06)  try {
(07)    database d1;                                       // Make empty database
(08)    d1.load("database/load_example.dtb");              // Load a database
(09)    d1.add_from_file("description/descriptions.txt");  // Add (raw) utterances
(10)    d1.set_attribute_entry_of_all("Record_date = 04/13/98");  // Add a new attribute
(11)    d1.show_main_memory_usage();                       // Show memory usage
(12)
(13)    database d2;                                       // Make new database
(14)    d2.add_from_database_if(d1,"age <= 34");           // Extract utterances
(15)    d2.remove_if("Format == NIST");                    // Remove utterances
(16)    d2.remove_if_absent("Origin"); }                   // Remove utterances
(17)
(18)  catch (const database_error& error) {                // Error handling
(19)    cout << "Message was :      " << error.message  << endl;
(20)    cout << "Throwing function: " << error.function << endl;
(21)    cout << "Return value:      " << error.code     << endl;
(22)    return error.code; }}
```

In this example, a database d1 is created, utterances are loaded into it and added from a description file. Then a new attribute with an utterance-independent value is added to all utterances. Finally, the main memory consumed by the utterances so far is displayed (lines 7 through 11).

A second database, d2, is created and filled with all utterances from d1 that contain the age attribute and for which the value of this attribute is at most 34. (Note that utterances from d1 without the age attribute will not appear in d2.) Now, all utterances featuring the NIST format are removed as well as those lacking the Origin attribute (lines 13 through 16).

This ends the actual code. The file access in line 8, for example, might fail due to various reasons. The error handling for such cases is introduced in line 6 with the try statement. Whenever an error occurs in the following lines, an exception of type database_error will be thrown and caught by the catch statement in line 18. An instance of the database_error struct contains a message, the function where the error occurred, and an error code. Those values are displayed in lines 19 through 21, and the error code is returned to the shell. Execution is not continued behind the position where the error occurred.

The PYTHON equivalent (cf. example2.py) is:

```
(01)  #!/usr/local/bin/python1.5

(02)  from database import *

(03)  d1=database()                                                  # Make empty database
(04)  d1.load("../example/database/load_example.dtb")                # Load a database
(05)  d1.add_from_file("../example/description/descriptions.txt")    # Add (raw) utterances
(06)  d1.set_attribute_entry_of_all("Record_date = 04/13/98")        # Add a new attribute
(07)  d1.show_main_memory_usage()                                    # Show memory usage

(08)  d2=database()                                                  # Make new database
(09)  d2.add_from_database_if(d1,"age <= 34")                        # Extract utterances
(10)  d2.remove_if("Format == NIST")                                 # Remove utterances
(11)  d2.remove_if_absent("Origin")                                  # Remove utterances
```

As mentioned above in section 3, the passing of exceptions from a C++ library on to PYTHON did not work, so error handling within this script doesn't make sense – the database catches all errors itself. Apart from that, the result is the same as in the C++ code above.

5 Maintainance

This section describes issues that become relevant when the database library is being enhanced by a programmer. Hopefully, the comments given here help to read and understand the code in the source files.

5.1 Data structures

All newly developed data structures are C++ classes or structs, whose name in general starts with an underscore to mark them as types rather than variables. The exceptions to this rule are the main class database, the exception structure database_error (see section 1.4), and some very small structs of the name pattern cmp_..., which are needed as compare objects for binary search routines.

Their precise declarations can be found in the header files in the include subdirectory. The basic data structure hierarchy is as shown in figure 1 on page 3. At the top of that hierarchy, we have the type database. This is the class that the user can make objects of and that provides all the methods to manipulate such an object. All other classes are used only internally as sub-structures of the database class.

A database instance contains a name, which is the default filename for load and save operations, and a waveform_path, which is used as a common path for waveform files, see section 2.6.1 for details. Further, there are two major components:

Attribute information: The struct `_attr_information` holds all attributes (struct `_attribute`) and entries (struct `_entry`) of the database, that occur in some utterance. Their purpose is to attach an integer ID to an attribute or entry name, such that in an actual utterance, only the ID of an attribute or entry has to be stored, rather than a string representing its name.[7] This reduces memory requirements considerably. If the user wishes to extract all utterances with the property `Speaker==John`, then the ID's for attribute `Speaker` and entry `John` are retrieved from the attribute information. After that, browsing through all utterances only involves integer comparison rather than string comparison, making the loop faster. – The "special" attributes, mentioned on page 7, don't get an ID assigned, as their names are fixed and they occur in every utterance. For a listing, see the reference section 6, or have a look at the type definition `_special_type` in the `attribute.hh` header file.

Utterances: These are stored in a container of elements of type `_utterance`. An instance of this struct consists of three components:

1. Attribute entries: They are maintained in a container of elements of type `_attribute_entry`. The latter has three components, which are the ID of the attribute, the type of the corresponding entry (string or number), and a union containing either the entry ID, if string, or its float value, if number.
2. Waveform: The structure `_waveform`, as was described in section 2.6.1.
3. Features: The structure `_feature`, as was essentially described in section 2.6.2.

Remember that the data types for the samples of a waveform and the feature vectors come from the `mat2D` package.

Every attribute and every entry are associated with a counter of all the utterances that contain that attribute or that entry. Such, when browsing through the container of all utterances to search for a specific attribute entry, the search can be truncated after the number of utterances containing that attribute/entry has been reached.

Utterances, attributes, entries, and attribute entries are stored in "containers", which are currently C++ vectors. Those always occupy a coherent portion of main memory, so binary search can be applied. For this reason, the four containers are kept sorted. The criterion is the ID of the respective data type; in case of the attribute entries, it is the ID of the attribute (as every attribute occurs at most once in every utterance). – This way, the memory savings using ID's for strings are considerable, whereas the time consumption by converting the ID's in strings and vice versa is low.

5.2 Programming issues

5.2.1 Statics, globals, defaults

The following (sub-)structures contain the following user-relevant static variables (defaults in parentheses):

`string database::tmp_dir ("/usr/tmp")` to temporarily save files while doing *garbage collection* (see section 2.6.3)

`char _utterance::delimiter ('@')` to mark utterance boundaries in a save file

`string _waveform::shorten_decode_command` to be executed when an original waveform file is still compressed

`string _waveform::tmp_filename (tmpnam(0))` to temporarily save the uncompressed waveform file to

`string _waveform::extension ("wvf")` to append to the database name to make a sample data file name, if not specified by the user (cf. `Char_sample_file` variable in section 2.6.1)

`long _waveform::buffer_size ((long)5E06)` size of the buffer in bytes to be filled with sample data (section 2.6.1)

[7] This does, of course, not apply to entries having number values themselves, such as the age of the speaker. ID's are only attached to string entries.

`string _feature::extension ("ftr")` to append to the database name to make a feature data file name, if not specified by the user (cf. `Feature_file` variable in section 2.6.2)

`long _feature::buffer_size ((long)5E06)` size of the buffer in bytes to be filled with feature data (section 2.6.2)

The C function call `tmpnam(0)` supplies a temporary filename on the current system. – All these static variables are constants. They can only be set by the user before compiling the library. Statics that are not user-relevant are not listed.

Defaults of non-static member attributes are:

```
string database::name      =  "noname"
string database::extension =  "dtb"
```

All other class members default to "-" (strings) or -1 (integral types). Note that strings should not have "" as default, as this value cannot be saved to a file and re-read. It is important to know these defaults when values of not-yet-assigned attributes in the database are requested, for example using the `get_..._entry` functions.

Finally, there are two constant global variables of minor importance: `__NO_VALUE__` is used in `show` routines only, and is printed for the value of attributes that have no meaningful value assigned yet (printing the default "-" or -1 might be inexpressive). It defaults to `"(none)"`. The other global variable is `__EMPTY_STRING__` and has the value "". The reason for introducing this seemingly unnecessary variable was given in section 3 on page 17.

5.2.2 Writing style in the source files

While writing the code for the database package, excessive use of the C `assert` macro was made to check whether program invariants hold true. Almost all of these statements can be removed from the code to make it slightly more efficient, but they are highly recommended in programming sessions when enhancing the code by new features. If the code contains a bug, being informed about it by a violation of such a statement often tells you immediately what the cause is, very much unlike a segmentation fault, which is the alternative.

There are two special types of `assert` statements, which must not be removed:

- `assert(false)` is used in `switch` statements to claim that the corresponding case does not happen, i.e. the execution will not reach this point. Those occurrences don't waste any execution time as long as the program works correctly.

- Some of the boolean expressions in the argument of `assert` have a side-effect, i.e. they perform some action and, at the same time, check a related program invariance. The action itself is the main purpose of these statements. If they are removed, the result will normally be disastrous. A typical example is

  ```
  assert(fgetc(in)=='\n')
  ```

  which moves the file pointer associated with `in` forward one character and also checks whether the symbol skipped was a newline, as expected.

  To prevent these lines from accidentally being removed, they are marked by a

  ```
  // essential side effect!
  ```

  comment.

Comments are mainly provided in the header files, where they specify the purpose of every function. In the actual source files, comments are given only for non-obvious or non-portable parts of the code.

5.2.3 System-dependencies

Although mostly avoided successfully, some parts in the program rely on the architectural environment, and the user's help is needed to adapt some variables to local circumstances:

1. If original waveform files contain compressed data, they need to be uncompressed before they can be read into main memory, and the user has to supply the proper name of the uncompress command in the `_waveform::shorten_decode_command` variable mentioned above. The execution of this command happens via a `system` call. Please refer to the `waveform.cc` source file to check if the arguments to this command are passed correctly (flags may be required, etc.).

2. The `garbage_collection` routine needs to move files. The C function `rename` can be used and is system-independent, but it has the drawback that is does not move files between different file systems. This is, however, expected to happen quite often in the context of speech recognition, as large data files may reside on other machines than the executable programs. Therefore, a system call is used here to rename files. Currently, the UNIX `mv` command is employed, which must be adapted to other environments, especially to WINDOWS operating systems.

3. The `database::tmp_dir` variable is expected to provide a directory for temporarily placing files. It defaults to `"/usr/tmp"`, which may not even exist on your system or might be too small for moving large files to it. The reason for doing the temporary file access on this directory is that on UNIX systems, it is guaranteed to be on a disk of the current running machine, on which the program was started. Hence, file access is fast. This is not necessarily the case if the current directory (`"."`) is used.

All these occurrences of system-dependent parts of the code are marked by a

```
// System-dependency!!
```

comment.

5.2.4 Compiler issues

The database library was so far mainly used on LINUX platforms using the g++ compilers 2.7.2.2 and 2.8.1. At least on the local machines, the older version was not totally reliable, some functions (as in the C++ vector class) were simply missing, others causes program crashes. This could be avoided by avoiding those functions themselves.

The newer release, 2.8.1., seems to have gotten rid of these flaws, but the following syntactical changes are required to compile the library under the new version:

- The name of the flag to allow C++ exception handling was changed from

  `handle-exceptions` into `exceptions`.

  This must be set accordingly in the makefiles of the `src` and `swig` directories.

- The name of the function to remove elements from a string in C++ was changed from

  `remove` into `erase`.

  In order to avoid having to change the code at many places, a macro had to be introduced that must be set to the "current" name of the remove function. This macro is defined using the `-D` option in the compiler call, so it is found in the `src/Makefile`.

6 Quick Reference

In the following, some important details are enumerated, which have been discussed already in the text, and are here summarized only. Numbers in parentheses at the end of a line refer to page numbers for more information.

Parameters: denotation for the `name` and `waveform_path` attributes of a database object. (4)

Attribute names: if number-valued, initial must be small or digit (`'a'..'z','0'..'9'`), others are string attributes. (7)

Special attributes: occur in every utterances and don't get an ID assigned. Currently:

- `Format`
- `sampling_frequency`
- `number_of_samples`
- `bytes_per_sample`
- `Order_of_bytes`
- `Relative_path`
- `Char_sample_file`
- `Short_sample_file`
- `Feature_file`

The agreement on the initial letter of an attribute name to identify its type holds for special attributes also. (7)

Utterance ID: taken to be the relative path; used for sorting and indexing. (7)

Format of description file: line oriented, every line being one utterance, every utterance being a list of "`attribute = value`" pairs, separated by comma. Spacing between tokens and newlines between utterances arbitrary. If special characters appear within attribute or entry names, enclosing double quotes are required. " itself must be canceled by \. (7)

Saving time: if names of attributes and entries are kept short, run time will be reduced for functions that require parsing, for example, those taking condition strings. Memory needs are barely reduced, as ID's are stored instead of strings.

Satisfaction of conditions: A condition "`attribute <OP> value`" is satisfied by an utterance if the attribute exists and the condition is satisfied. – Conversely, it is not satisfied if the attribute does not exist *or*, it exists but the condition is violated. (9)

Static variables: Perhaps need to be adapted depending on local architecture. Check section 5.2.1 on page 20 for a complete listing.

参考文献

[1] Bjarne Stroustrup: THE C++ PROGRAMMING LANGUAGE. *Addison-Wesley, Massachusetts, 1997.*

[2] Dirk Louis: C UND C++: PROGRAMMIERUNG UND REFERENZ. *Markt und Technik, Buch- und Software-Verlag, Haar bei München, 1996.*

[3] HP C/HP-UX REFERENCE MANUAL. *Hewlett-Packard Company, 1992.*

[4] Dave Beazley: SWIG. SIMPLIFIED WRAPPER AND INTERFACE GENERATOR. *Online-manual at* `http://www.cs.utah.edu/~beazley/SWIG/swig.html`.

[5] Mark Lutz: PROGRAMMING PYTHON. *O'Reilly & Associates, Inc., October 1996.*

[6] Helmut Kopka: LaTeX. EINE EINFÜHRUNG. *Addison-Wesley, Bonn, München, Paris, 1992.*