# ConceptBase V6.1 Programmer's Manual

Matthias Jarke, Manfred A. Jeusfeld, Christoph Quix (Eds.)

Contributions to this manual were made by: Lutz Bauer, Rainer Gallersdörfer, Michael Gebhardt, Manfred Jeusfeld, Thomas List, Hans Nissen, Christoph Quix, René Soiron, Martin Staudt and Ralf Stössel.

This manual refers to: ConceptBase V6.1, released 17-Jan-2003

# Contents

# Chapter 1

# Server Interface

This chapter provides basic information necessary for communication with the ConceptBase server. It is possible that the CBserver and the clients 'live' on different machines because communication with the CBserver is realized through a message protocol using inter-process communication (IPC) based on standard Internet sockets. It is even possible (but not recommended :-) ) to use a standard telnet program for the communication with a CBserver. The following chapter describes this protocol as it is necessary to know for a specialized client which wants to request services from the CBserver. Readers who intend only to use one of the programming interfaces for C, C++ or Java may skip this chapter, but it contains some useful basic information.

From a client's point of view the CBserver can be seen as an abstract data type exporting several parameterized operations. These operations comprise methods for storing/retrieving information into/from the KB, methods for establishing and closing the connection to a CBserver and methods for testing the KB. Since the client and CBserver are two different processes a client cannot directly call these methods like procedures but must access them using a message protocol. However, the use of one of the application programming interfaces (API) for C, C++ and Java simplifies the communication and interaction with the CBserver from the viewpoint of an application programmer.

This chapter is organized as follows: Section **??** describes the message protocol which is used to communicate with other processes. Section **??** describes the interface to the CBserver, i.e. the data structures and operations which the ConceptBase kernel offers and the message protocol which makes these operations accessible to other processes.

## 1.1 Message Format

As already mentioned, any client that wants to use the methods of a CBserver has to communicate with CBserver according to a message protocol. So called ipcmessages can be sent via IPC to the port reserved for this CBserver. The CBserver handles such a message and reports back an answer: the ipcanswer.

### 1.1.1 ipcmessage

**ipcmessage ( sender, receiver, method, args ).** where

**sender** is the identifier for the sender of the message,

**receiver** is the identifier for the receiver of the message (usually the CBserver itself, but could be any other client connected to CBserver as well),

**method** is one of the methods exported by the CBserver (or a method known to another client which is adressed by the message),

**args** are the arguments for method.

Note, that it is necessary to "encode" the parameters of an ipcmessage. This means, that the strings must begin and end with `"`. If the string contains the characters `"` or \, they must be escaped with a backslash (\). Please refer to the grammar definition in appendix **??** for full details.

Messages can also be directed to other clients of the CBserver by using a different ID than the server ID as a receiver of message. If messages are sent from client to client, clients have to poll for messages using the method NEXT_MESSAGE. This function has not been tested recently.

A message can be prefixed by the length of the message, which is specified in five bytes. The first byte is always the character 'X', the next bytes are computed by the following formulas (len is the length of the message without this prefix):

1. (len /$256^3$) modulo 256
2. (len /$256^2$) modulo 256
3. (len /256) modulo 256
4. len modulo 256

e.g., the first byte is the highest byte and the last byte is the lowest byte of an unsigned integer. Note, that specifying the length of an IPC-message is optional. IPC-messages without the length information should also be accepted by the server but communication problems might occur in rare circumstances.

### 1.1.2 ipcanswer

**ipcanswer ( sender, completion, return ).** where

**sender** is the identifier of the answering program (usually the CBserver since other programs cannot answer directly but only receive the message and send back another message via the CBserver). This is sent as an encoded string.

**completion** signals success (=ok) or failure (=error) or unability (=not_handled) of handling the message

**return** contains the return value(s) of the handled message. This is sent as an encoded string.

Additionally the CBserver administrates message queues for all connected clients. Whenever a client X sends a message to another client Y (which is not CBserver) the CBserver stores this message into the message queue of the client Y and gives it back to X.
**Remark:** If clients are likely to exchange messages they should periodically poll their message queue.

## 1.2 Methods Exported by the CBserver

The CBserver offers the following methods (list is incomplete):

**general methods:** `TELL`, `UNTELL`, `TELL_MODEL`, `ASK`, `HYPO_ASK`, `NEXT_MESSAGE`, `ENROLL_ME`, `CANCEL_ME`, `GET_MODULE_CONTEXT`

**privileged methods:** `STOP_SERVER`, `REPORT_CLIENTS`

**internal methods:** `LPI_CALL`

Privileged methods affect other clients connected to CBserver as well and should only be executed by an authorized client. That means, that only the *owner* of the ConceptBase server process may execute this methods.

The internal method `LPI_CALL` gives a client the possibility to call internal procedures of the ConceptBase server. This method is mainly useful for ConceptBase developers for debugging and analysing.

In the following description of the methods `return` refers to the respective parameter of `ipcanswer(sender, completion, return)`.

For each error occuring during the execution of a method an error message is stored by the CBserver `receiver` in the message queue of the client `sender`. This error message can be fetched via a call of method `NEXT_MESSAGE`, see below.

### 1.2.1 TELL

**ipcmessage ( sender, receiver, TELL, [ objects ] )**

**objects** encoded string containing object descriptions in Telos represented as frames

**return** `"yes"` in case of success, `"no"` otherwise

The CBserver receiver checks the syntax of `objects` creating a parse tree for each object description, called *SMLfragment*. If no syntax error occurs the SMLfragments are transformed into an internal network representation with specialized rules and constraints. Those facts which are not already retrievable are temporarily added to the KB. A check is then performed to determine whether the updated KB still satisfies the integrity constraints. In the case of satisfaction the new information is made permanent, otherwise it is deleted.

### 1.2.2 UNTELL

**ipcmessage ( sender, receiver, UNTELL, [ objects ] )**

**objects** encoded string containing object descriptions in Telos represented as frames

**return** `"yes"` in case of success, `"no"` otherwise

The `objects` will be untold, i.e. the upper bound of their transaction time interval is set to the time the UNTELL operation takes place. That means from this time on the system does not believe this information anymore. Questions about the current state of the knowledge base yield the same answer as if the objects were never inserted into the system. However questions about earlier states will regard all information (even untold) the transaction time of which contains the time in question (= rollback time). Like in the TELL method, if the UNTELL operation would result in an inconsistent KB state it is rejected by the integrity checker.

### 1.2.3 TELL_MODEL

**ipcmessage ( sender, receiver, TELL_MODEL, [ [ filelist ] ] ).**

**filelist** A list of comma-separeted ipc strings, which contain the full filenames of files to be loaded by ConceptBase server.

This method is similar to the `TELL` method, except that the frames which are told to ConceptBase are loaded from the given files and not passed directly to ConceptBase.

**Remark:** The files to be loaded by ConceptBase must be accessible for the server. This is not always the case, when server and client are running on different machines with different filesystems mounted on. Another problem may occur, due to access protections, because the user running the ConceptBase server is not allowed to read the specified files.

### 1.2.4 ASK

**ipcmessage ( sender, receiver, ASK, [Format, Query, AnswerRep, RollbackTime ] )**

**Format** is either `FRAMES` or `OBJNAMES`, depending of the format of `Query`. If in `Query` only the object name of a query is given (e.g. `AllEmployees`) then the format must be `OBJNAMES`. If the query is specified as frame (e.g. `"QueryClass AllEmployees isA Employee end"`) then the format must be `FRAMES`.

**Query** depending on the `Format` this may be simple object names or frames representing queries. In the later case, the query is temporarily told to the object base and after evaluating deleted from the object base, if it does not already exist in the object base before the transaction.

**AnswerRep** answer format specification, possible values are: `FRAGMENT`, `FRAME`, `LABEL` or an instance of `AnswerFormat`[1]. The syntax of the `FRAGMENT` and `FRAME` formats

[1]See the *ConceptBase User Manual* for details about user-defined answer formats

are explained in the appendix of the *ConceptBase User Manual*. If the answer representation is `LABEL` a comma-separated list of object names is returned.

**RollbackTime** rollback time specification

**return** list of answers in case of success, `"no"` otherwise

The values of the `Format` argument (`FRAMES` and `OBJNAMES`) are ipc message keywords and must not be encoded as the other arguments `Query`, `AnswerRep` and `RollbackTime`.

**Example:**

The following two queries are predefined builtin queries and available after booting the *ConceptBase* server. These queries additionally give good examples for derived expressions by instantiating parameters of generic query classes.

- `exists[x/objname]`
  The answer return is `"yes"` if there is an object named `x`, otherwise `"no"`.
- `get_object[x/objname]`
  The answer is the frame representing the object `x` if there is an object `x`. Otherwise, the answer is `"no"`. Only information that is explicitly stored (i.e. not inherited or deduced) is considered. If you want deduced information, you must specify additional parameters. For example, the answer of the following query is the `Class` object with stored and deduced attributes:

```
get_object[Class/objname,FALSE/dedIn,FALSE/dedIsa,TRUE/dedWith]
```

### 1.2.5 HYPO_ASK

**ipcmessage(sender, receiver, HYPO_ASK,[ ObjList, Format, Query, AnswerRep, RollbackTime])**

**ObjList** string of objects in frame syntax

**Format** see ASK

**Query** see ASK

**AnswerRep** see ASK

**RollbackTime** see ASK

**return** list of answers in case of success, no otherwise

This method allows to process so called 'hypothetical' queries against the KB. The objects in objList are temporarily told. This list may contain query objects which may in turn be referred to by names contained in Query. Then the queries in queryList are evaluated as if the temporary information would belong to the KB. Afterwards the temporary information will be removed.

### 1.2.6 NEXT_MESSAGE

**ipcmessage ( sender, receiver, NEXT_MESSAGE, [type] )**

**type** identifier describing the type of the message (e.g. `ERROR_REPORT`). This argument may not be encoded as other string, but may be *empty*.

**return** contains the next message for the client if its message queue contains at least one message, `empty_queue` if no message exists

Client sender requests a message from the CBserver receiver stored in its message queue. Usually, this method is called after the CBserver returns error for a previous method. The client program must then get all error messages until it gets `"empty_queue"` as answer.

### 1.2.7 STOP_SERVER

**ipcmessage ( sender, receiver, STOP_SERVER, [password] )**

**password** password allowing a client to stop a CBserver (may be *empty*)

**return** `"yes"` in case of success, `"no"` otherwise

The CBserver receiver is terminated if the password is correct and the user running the client is also the owner of the CBserver to be stopped. To the requesting client `STOP_SERVER` has the same effect as `CANCEL_ME`. It is recommended to terminate the CBserver by using the respective menu choice from the "Server Menu" of ConceptBase Workbench if you want to stop the CBserver process.

### 1.2.8 REPORT_CLIENTS

**ipcmessage ( sender, receiver, REPORT_CLIENTS, [ ] )**

**return** list of all clients currently connected to CBserver receiver

CBserver receiver reports back the identifier, toolclass and owner name of all currently connected clients including itself.

### 1.2.9 ENROLL_ME

**ipcmessage ( sender, receiver, ENROLL_ME, [toolclass,username] )**

**toolclass** 'class' the client belongs to

**username** name of the user running the client

**return** identifier assigned to the client by CBserver

`sender` and `receiver` have value `""` since they are not known. The sending client will be registered as a new client of the CBserver with its own identifier and message queue. This message must be sent to the CBserver before any other message can be sent, since all other messages require valid identifiers to be assigned to sender and receiver. If the user specified by `username` is stored as an instance of the class `CB_User` of the CBserver, then the value of the attribute `homeModule` of that user is taken as the initial module context of the client. Otherwise, the default module context `System` is assigned to the client. In a variant of ENROLL_ME, one can specify a third parameter `module` which will set the module context explicitely.

### 1.2.10 CANCEL_ME

**ipcmessage ( sender, receiver, CANCEL_ME, [ ] )**

**return** `"yes"` in case of successful disconnection, `"no"` otherwise

Client sender will be disconnected from CBserver. This means that from now on the sender is no longer known to the CBserver (no further messages can be sent) and its message queue is deleted. After successfully canceling the connection, the ipc sockets to the server must be closed by the client program.

### 1.2.11 GET_MODULE_CONTEXT

**ipcmessage ( sender, receiver, GET_MODULE_CONTEXT, [] )**

**return** name of the module currently assigned to client `sender`

This service allows clients to interrogate the CBserver about the currently active module context in which they operate.

### 1.2.12 LPI_CALL

**ipcmessage ( sender, receiver, LPI_CALL, [call] )**

**call** an internal routine

**return** `"yes"` if call succeeded, `"no"` otherwise

This is for debugging and testing purposes only.

# Chapter 2

# Programming Interface for a C Client: libCB

This chapter describes the programming interface for ConceptBase. The programming interface consists of a number of data structures and C functions which are defined in the header file `CBinterface.h`. Make sure that this header file is included in each of source files that use functions of libCB. The data structures are explained in section **??**. The C library `libCB` contains all functions described in section **??**.

The libraries can be found in the following directories:

**Solaris/SPARC:** The directory `$CB_HOME/sun4/lib` contains the libraries for static linking of your application.

**Solaris/PC:** The directory `$CB_HOME/i86pc/lib` contains the libraries for static linking of your application.

**Linux:** The directory `$CB_HOME/linux/lib` contains the libraries for static linking of your application.

**Windows:** The directory `$CB_HOME/windows/lib` contains the dynamic libraries for dynamic linking of your application.

There are currently no plans to build dynamic libraries for the Unix-based platforms.

The directories `$CB_HOME/examples/Clients/LogClient` and `$CB_HOME/examples/Clients/C_Client` contain example programs, which uses the programming interface `libCB` to communicate with the ConceptBase server. The `LogClient` program is explained in appendix **??**. Information on you how to compile and link your source code with the ConceptBase libraries can be found either in the directories of the example clients or in section **??**.

## 2.1 Data Structures

This section describes the data structures used by the API, in particular structures which are passed to and returned by the interface procedures.

The following C-types are defined in the file `CBinterface.h` which is located in the directory `$CB_HOME/include`.

### 2.1.1 Completion

```
typedef enum {CB_OK=0, CB_ERROR, CB_NOT_HANDLED,
              CB_TIMEOUT, CB_CONN_BROKEN} Completion;
```

The different return values have to be interpreted as follows:

**CB_OK** the message has been handled successfully.

**CB_ERROR** an error occurred during the execution of the message; the ConceptBase server stores some error reports for you on your message queue which may be read calling `em = get_errormessages()` (see below).

**CB_NOTIFICATION** indicates that the message is a notification message. Notification messages are sent by the server if the client has requested notification on updates on certain views.

**CB_NOT_HANDLED** the server was not able to manage your message at all. This may be due to an invalid format of input parameters (e.g. wrong Telos syntax) or missing parameters.

**CB_TIMEOUT** the message has been sent successfully to the server, but there has been no answer from the server after a specific amount of time (depends on the type of message sent). This may be due to the number of clients which are active or due to the kind of message you sent (some queries may last longer than others). The client is responsible for the correct handling of answers returned after CB_TIMEOUT occured.

**CB_CONN_BROKEN** the sending of the last message failed (the connection to the server is no longer accessible). Again, the client is responsible to handle this return value (e.g. stopping the client).

### 2.1.2 Answer

```
struct answer { char *sender;
                 Completion completion;
                 char *return_data;
               };
typedef struct answer Answer;
```

The `Answer` structure is returned by most library functions. The first field `sender` contains the name of the sender as it is maintained by the ConceptBase server. The second one specifies the status of the message processing (see section **??**) while the third one contains return values of the message called.

### 2.1.3 Server

```
struct server { char *serverName;
                 char *client;
                 int connected_to_CB_server;
                 SOCKET socket;
               };
typedef struct server Server;
```

This structure is allocated and filled by the `connect_CB_server()` call and used as an anchor by all the other routines to get the right server. The field `connected_to_CB_server` should usually be true, as it indicates that the client is connected to the server (or not). The `socket` field represents the socket which is used for the communication with the CB server and should be used only internally.

### 2.1.4 Clients

```
struct clients { char *client;
                 char *toolclass;
                 char *username;
                 struct clients *next;
                  };
typedef struct clients Clients;
```

This structure represents a simply linked list of clients. A pointer to this structure is returned as result of the `report_clients` call.

### 2.1.5 Error_Messsages

```
struct errormessages { char *errormessage;
                       struct errormessages *next;
                     };
typedef struct errormessages Error_Messages;
```

List of error messages given by the ConceptBase server every time a communication event can not be processed correctly. This list may be obtained calling `get_errormessages()`.

## 2.2 Functions

### 2.2.1 connect_CB_server

```
int connect_CB_server(int      portnr,
                      char    *hostname,
                      char    *clientname,
                      char    *username,
                      Server **server)
```

***Description:***

Sets up a connection to a given ConceptBase server. This routine has to be called once before calling one of the following routines.

***Input parameters:***

**portnumber** number of the port of the server (this port number is unique per server as may be defined at the server's start up time).

**hostname** name of the machine on which you started the server

**clientname** name of the client to be connected (e.g. *TelosEditor*)

**username** name of the user who started the client

**server** pointerpointer to a struct server; on a succesfull connection the structure will be allocated and filled

***Result:***

**0** Connection established

**-1** There is no such server (probably wrong portnumber and/or host)

$>$**0** a completion value (see section **??**)

### 2.2.2 disconnect_CB_server

```
int disconnect_CB_server(Server *server)
```

***Description:***

Closes a previous connection to a ConceptBase server. This procedure has to be called every time a client is stopped (but usually the CBserver is not affected by clients that crash or do not disconnect correctly).

***Input parameters:***

**sever** pointer to the structure discribing the current ConceptBase server

***Result:***

**0** Connection correctly terminated

**-1** error, not connected

$>$**0** a Completion value

### 2.2.3 tellCB

```
Answer* tellCB(Server *server, char *objects)
```

***Description:***

Inserts a set of objects into the ConceptBase server. This function has been renamed from previous releases as `tell` is a operating system function on some systems.

***Input parameters:***

**server** pointer to the structure discribing the actual server

**objects** pointer to a list of objects, which should be inserted into the knowledge base. This should be a normal NULL-terminated C-string.

***Result:***

An answer struct where `return_data` is either `yes` or `no` and where the completion value indicates the result of the operation:

**CB_OK** operation sucessfull

**CB_ERROR** There was an error while inserting, get the errormessages by calling `get_errormessages()`

**other** see the description in section **??**

### 2.2.4 untell

```
Answer* untell(Server *server, char *objects)
```

***Description:***

Removes a list of objects from the knowledge base. Note the specific semantics of the untell method as described in chapter **??** of this Manual.

***Input parameters:***

**server** pointer to the structure discribing the actual server

**objects** pointer to a list of objects, which should be deleted. This should be a normal NULL-terminated C-string.

***Result:***

An answer struct where `return_data` is either `yes` or `no` and where the completion value indicates the result of the operation:

**CB_OK** operation sucessfull

**CB_ERROR** There was an error while removing, get the errormessages calling `get_errormessages()`

**other** see the description in section **??**

### 2.2.5 tell_model

```
Answer* tell_model(Server* server, char** models);
```

***Description:***

Tells the given files to the server. Note that the server must be able to find these files in its file system.

***Input parameters:***

**server** pointer to the structure discribing the actual server

**objects** pointer to a NULL-terminated array of C-strings, containing the file names which should be loaded by the server.

***Result:***

An answer struct where `return_data` is either `yes` or `no` and where the completion value indicates the result of the operation:

**CB_OK** operation sucessfull

**CB_ERROR** There was an error while removing, get the errormessages calling `get_errormessages()`

**other** see the description in section **??**

### 2.2.6 get_errormessages

```
Error_Messages *get_errormessages(Server *server)
```

***Description:***

Gets the errormessages corresponding to the last error. This procedure has to be called every time CB_ERROR has been returned by a given procedure. Otherwise, further messages may be disturbed by the error messages which are returned first by the server.

***Input parameters:***

**server** pointer to the structure discribing the actual server

***Result:***

list of errormessages (see section **??**)

### 2.2.7 ask

```
Answer* ask(Server* pServer,
            char* szQuery,
            char* szAskFormat,
            char* szAnsFormat,
            char* szRBTime);
```

***Description:***

Sends the query in the specified format (`szAskFormat`) to the server and returns the result of the server, which will be represented in the format given in `szAnsFormat`. The rollback time (`szRBTime`) is usually `Now`.

***Input parameters:***

**pServer** a pointer to a server structure

**szQuery** the query

**szAskFormat** the format of the query (FRAMES or OBJNAMES)

**szAnsFormat** the format of the answer (e.g. FRAME, LABEL,...)

**szRBTime** rollback time (e.g. Now)

***Result:***

an answer struct:

**sender** the tool that has provided the answer, usually the ID of the server

**completion** Completion value indicating the success of the method, e.g. `CB_OK`, `CB_ERROR`

**return_data** the result of the query in the specified format, or the string `"nil"` if there are no results or if there was an error during query processing

### 2.2.8 ask_frames

```
Answer* ask_frames(Server *pSserver,
                 char *szQuery,
                 char* szAnsFormat,
                 char *szRBTime)
```

***Description:***

As `ask`, but `szAskFormat` is fixed to be `FRAMES`, i.e. queries have to be given as frames.

### 2.2.9 ask_objnames

```
Answer* ask_objnames(Server *pSserver,
                   char *szQuery,
                   char* szAnsFormat,
                   char *cbfoGet,
                   char *szRBTime)
```

***Description:***

As `ask`, but `szAskFormat` is fixed to be `OBJNAMES`, i.e. queries have to be given as object names (or derive expressions).

### 2.2.10 hypo_ask

```
Answer* hypo_ask(Server* pServer,
          char* szFrames,
          char* szQuery,
          char* szAskFormat,
          char* szAnsFormat,
          char* szRBTime);
```

***Description:***

As `ask`, but first tells the frames given in szFrames to the server, then performs the query and finally deletes the told frames from the object base.

### 2.2.11 report_clients

```
Clients* report_clients(Server *server)
```

***Description:***

Returnes a list of all clients connected to the server.

***Input parameters:***

**server** pointer to the structure describing the actual server

***Result:***

list of clients or NULL on error

### 2.2.12 get_servermessage

```
Answer* get_servermessage(Server* server, char* type);
```

***Description:***

Gets a message from the server for the client. This function is called by `get_errormessages()`.

***Input parameters:***

**server** a pointer to a server structure

**type** type of the message to be retrieved (e.g. ERROR_REPORT)

***Result:***

an answer object with the message or EMPTY_QUEUE in return_data

.

### 2.2.13 get_notification

```
Answer* get_notification(Server* server, int timeout);
```

***Description:***

Looks for a notification message. Notification messages are sent by the server if the client has requested notification on updates on certain views. The method will wait for a message from the server for the specified time.

***Input parameters:***

**server** a pointer to a server structure

**timeout** time to wait for a message

***Result:***

an answer object with completion `CB_NOTIFICATION` when a message was received, otherwise a completion value, usually `CB_TIMEOUT`.

### 2.2.14 stopServer

```
Answer* stopServer(Server* server, char* password);
```

***Description:***

Stops the server. Note, that only the user who has started the server may stop it.

***Input parameters:***

**server** a pointer to a server structure

**password** a password (not used, may be empty)

***Result:***

the result of the method

### 2.2.15 LPICall

```
Answer* LPICall(Server* server, char* lpicall);
```

***Description:***

Performs a LPI-Call at the server. With LPI (Logic Programming Interface) one can call ProLog predicates defined in an LPI-Module.

***Input parameters:***

**server** a pointer to a server structure

**lpicall** the predicate to be called

***Result:***

the result of the method

### 2.2.16 free*

```
void freeAnswer(Answer* ans);
void freeServer(Server* srv);
void freeClients(Clients* c);
void freeErrorMessages(Error_Messages* err);
```

***Description:***

These functions free the allocated memory by the corresponding structures. Note that memory of all results which are returned by the library methods have to be freed by the caller.

### 2.2.17 send_message

```
Answer send_message(Server *server,
                    char   *method,
                    char   *data)
```

***Description:***

This procedure is the most general one and used by most functions mentioned before. It sends a message of type `method` to the (already connected) CBserver `server`. `data` is a string containing data expected by the method `method`[1]. For normal usage of the client library, this function is not necessary. The more specific functions (e.g. `tellCB, untell, ...`) are more useful.

***Input parameters:***

**server** pointer to the structure describing the actual server

**method** string which defines the type of the message (e.g. `TELL`)

**data** the arguments for the given message type (e.g. `["Class Employee with ... end"]`)

***Result:***

an `Answer` structure containing `sender`, `completion` and `return_data`

[1]See chapter **??** and appendix **??** of this manual for a complete description of the available methods and their expected data

### 2.2.18 CBdecodeString

`char* CBdecodeString(const char* s);`

***Description:***

Decode a string. ConceptBase encodes all strings with ”” and
. To get the plain string, use this function.

***Input parameters:***

The string to decode.

***Result:***

The decoded string, it is a duplicate of the input if the input string is not encoded. The memory allocated by the result has to be freed by the caller.

### 2.2.19 CBencodeString

`char* CBencodeString(const char* s);`

***Description:***

Encode a String. ConceptBase encodes all strings with ”” and
. Use this function if you want to use Strings in Telos frames.

***Input parameters:***

The string to encode.

***Result:***

The encoded string. The memory allocated by the result has to be freed by the caller.

### 2.2.20 CBgetEncodeLength

`unsigned CBgetEncodedLength(const char* s);`

***Description:***

Return the length of an encoded string. This function is called by CBencodeString to allocate the memory of the encoded string.

### 2.2.21 CBgetLabels

`char** CBgetLabels(const char* labelList);`

***Description:***

Parse a comma-separated list of labels. ConceptBase returns sometimes comma-separated list of labels (e.g., for the answer format LABEL). This function makes an array of strings out of one plain string. This is a lazy function that will fail to produce a correct result if the object names contain commata (e.g., `"This, is, a, Telos, object, name, with, commata."`).

***Input parameters:***

A string with comma-separated-list.

***Result:***

A NULL-terminated array of strings.

## 2.3 Compiling and Linking

If you want to *compile* your source that uses libCB, you have basically to make sure two things:

- the header files of ConceptBase are found, and
- the correct system header files are included in `CBinterface.h`

The first item is usually achieved by adding a parameter -I with the include-directory of your ConceptBase installation to the list of compiler options. For the second point, you have to define the symbol LINUX, WIN32 or SOLARIS (usually done with the -D option of the compiler), depending on the operating system of your client application.

We have used the following compiler flags (with gcc 3.2 on the UNIX-based systems, and MS Visual C++ 6.0 on Windows):

**Solaris** `-I$(CB_HOME)/include -DSOLARIS`

**Linux** `-I$(CB_HOME)/include -DLINUX`

**Windows** `-nologo -MT -W3 -GX -O2 -I$(CB_HOME)/include -D "WIN32" -D "NDEBUG" -D "_CONSOLE" -D "_MBCS" -Fo".\\" /Fd".\\" -c`

If you want to *link* your application, you have to make sure that libraries are found by the system (-L option of gcc) and that the library libCB is indeed linked to your application (-l option). We have used the following linker options:

**Solaris** `-L$(CB_HOME)/sun4/lib -lCB -lnsl -lsocket`

**Linux** `-L$(CB_HOME)/linux/lib -lCB`

**Windows** `kernel32.lib user32.lib wsock32.lib $(CB_HOME)/windows/lib/libCB.lib -nologo -subsystem:console -incremental:no -machine:I386`

# Chapter 3

# Programming Interface for a C++ Client: libCBview

The libCBview provides a C++ encapsulation of libCB. It provides only an object-oriented API for ConceptBase and does not provide any additional methods in contrast to libCB.

Compilation and linking has to be done in the same way as for libCB. Note that you have to link both libraries libCB and libCBview if you want to use the C++ classes.

The documentation in the following sections has been generated with DOC++ (`http://docpp.sourceforge.net`).

## 3.1 CBclient

**class CBclient**

*A client class for ConceptBase*

### 3.1.1 CBclient

**CBclient ()**

**Description:**

Constructs an ”empty” client which is not connected

### 3.1.2 CBclient

**CBclient (char* host, int port, char* tool=(char*)NULL, char* user=(char*)NULL)**

**Description:**

Constructs a new CBclient object and connect to the specified host

### 3.1.3 ˜CBclient

**virtual ˜CBclient ()**

**Description:**

Disconnects from the CBserver and deallocates the memory

### 3.1.4 tell

**CBanswer* tell (char* )**

**Parameters:**

| | |
|---|---|
| `char` | *frames the frames |

**Returns:**

| | |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Tells frames to the server

### 3.1.5 untell

**CBanswer* untell (char* )**

**Parameters:**

| | |
|---|---|
| `char` | *frames the frames |

**Returns:**

| | |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Untells frames to the server

### 3.1.6 tellModel

**CBanswer* tellModel (char**)**

**Parameters:**

| | |
|---|---|
| `char**` | files an array of filenames |

**Returns:**

| | |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Tells files containing frames to the server

### 3.1.7 ask

**CBanswer* ask (char* query, char* format=”OBJNAMES”, char* answerrep=”FRAME”, char* rollbacktime=”Now”)**

**Parameters:**

| | |
|---|---|
| `char` | *query the query |
| `char*` | format the format of the query (FRAMES or OBJNAMES) |
| `char*` | answerrep the format of the answer (FRAME) |
| `char*` | rollbacktime Rollback Time (e.g. ”Now”) |

**Returns:**

| | |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Sends a query to the ConceptBase server

### 3.1.8 hypoAsk

**CBanswer* hypoAsk (char* frames, char* query, char* format="OBJNAMES", char* answerrep="FRAME", char* rollbacktime="Now")**

**Parameters:**

| | |
|---|---|
| `char` | *frames frames to be told |
| `char` | *query the query |
| `char*` | format the format of the query (FRAMES or OBJNAMES) |
| `char*` | answerrep the format of the answer (FRAME) |
| `char*` | rollbacktime Rollback Time (e.g. "Now") |

**Returns:**

| | |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Sends frames and a query to the ConceptBase server. The frames are told temporarely, the query is evaluated, and the temporarely objects are removed.

### 3.1.9 askObjNames

**CBanswer* askObjNames (char* query, char* answerrep="FRAME", char* rollbacktime="Now")**

**Parameters:**

| | |
|---|---|
| `char` | *query the query |
| `char*` | answerrep the format of the answer (FRAME) |
| `char*` | rollbacktime Rollback Time (e.g. "Now") |

**Returns:**

| | |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Sends a query to the ConceptBase server. Same as ask but with fixed query format (OBJNAMES).

### 3.1.10 askFrames

**CBanswer* askFrames (char* query, char* answerrep="FRAME", char* rollbacktime="Now")**

**Parameters:**

| | |
|---|---|
| `char` | *query the query |
| `char*` | answerrep the format of the answer (FRAME) |
| `char*` | rollbacktime Rollback Time (e.g. "Now") |

**Returns:**

| | |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Sends a query to the ConceptBase server. Same as ask but with fixed query format (FRAMES).

### 3.1.11 enrollMe

**int enrollMe (char* host, int port, char* user=NULL, char* tool=NULL)**

**Parameters:**

|  |  |
|---|---|
| `host` | hostname of the machine where the server runs |
| `port` | port number of server |
| `*user` | the name of the tool |
| `*tool` | the name of the user |

**Description:**

Connects to a ConceptBase Server Return the return value of connect_CB_server (see CBinterfaceh): -1: if socket to specified can not be openend 0: ok other: a completion value (see CBinterfaceh)

### 3.1.12 cancelMe

**int cancelMe ()**

**Description:**

Disconnects from a ConceptBase Server

Return the return value of disconnect_CB_server (see CBinterface.h): -1: error, not connected 0: ok other: a completion value (see CBinterface.h)

### 3.1.13 stopServer

**CBanswer* stopServer (char* password=NULL)**

**Returns:**

|  |  |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Stops the ConceptBase server. Note that a server may be stopped only by the user who has started it.

### 3.1.14 reportClients

**Clients* reportClients ()**

**Description:**

Return a list of clients connected to the CB server. The result will be a list of Client objects as defined in libCB.

### 3.1.15 nextMessage

**CBanswer* nextMessage (char* method=””)**

**Parameters:**

|  |  |
|---|---|
| `char*` | method the type of the message to be retrieved |

**Returns:**

|  |  |
|---|---|
| `a` | CBanswer object containing the result and the completion |

**Description:**

Gets a message from the server

### 3.1.16 getErrorMessages

**CBerror* getErrorMessages ()**

**Returns:**

a string containing all error messages

**Description:**

Gets the error messages from the server

### 3.1.17 LPICall

**CBanswer* LPICall (char* )**

**Description:**

Perform a LPI call on the server. A LPI call is a call of Prolog-predicate of the CBserver. This is mostly used for debugging.

### 3.1.18 connected

**inline int connected ()**

**Description:**

Check whether this client is connected

### 3.1.19 operator int

**inline operator int ()**

**Description:**

The operator int checks also if the client is connected.

### 3.1.20 getServerName

**char* getServerName ()**

**Description:**

Return the name of the server

### 3.1.21 getClientName

**char* getClientName ()**

**Description:**

Return the name of the client

## 3.2 CBanswer

**class CBanswer**

*C++ Wrapper for Answer struct of libCB*

### 3.2.1 CBanswer

**CBanswer (Answer* ans)**

**Parameters:**

`ans` pointer to the Answer struct

**Description:**

Constructs a CBanswer object from a Answer struct

### 3.2.2 ˜CBanswer

**˜CBanswer ()**

**Description:**

Deallocate the memory of the object

### 3.2.3 getCompletion

**Completion getCompletion ()**

**Description:**

Get the completion value of the answer

### 3.2.4 getResult

**char* getResult ()**

**Description:**

Get the result string of the answer

### 3.2.5 getRespondingTool

**char* getRespondingTool ()**

**Description:**

Get the ID of the responding tool of the answer. This usually the CBserver

## 3.3 CBerror

**class CBerror**

*C++ Wrapper of Error_Messages struct in libCB.*

### 3.3.1 CBerror

**CBerror (Error_Messages* e)**

**Parameters:**

`e` pointer to the Error_Messages

**Description:**

Construct a CBerror object from a list of Error_Messages

### 3.3.2 ˜CBerror

**˜CBerror ()**

**Description:**

Deallocate the memory of a CBerror object

### 3.3.3 getErrorMessage

**char* getErrorMessage ()**

**Description:**

Get the error message of this object. This will return only the first error message of the list.

### 3.3.4 getAllErrorMessages

**char* getAllErrorMessages ()**

**Description:**

This method will return all error messages of the list. The method will allocate a new string, thus the resulting string has to be freed by the calller.

### 3.3.5 getNextError

**CBerror* getNextError ()**

**Description:**

Get the next error message in the list

# Chapter 4

# Processing of Telos Frames: libtelos

This chapter explains the library `libtelos`, which contains the Telos parser. The Telos parser is able to parse the answers in FRAME or LABEL format from ConceptBase.

To call the Telos parser, you must link your program with the library `libtelos.a/libtelos.dll` which can be found in the directory `$CB_HOME/<arch>/lib` where `<arch>` is either sun4, i86pc, linux, or windows.

In your source files, you must include the header files `fragment.h`, `te_access.h`, `te_callparser.h`, `te_cursor.h`, and/or `te_smlutil.h`. All header files are located in the directory `$CB_HOME/include`.

The following sections explain several functions to call the parser and to handle the data structures. In principle, there are three different ways to parse and to access Telos frames:

- Using the functions and data structures defined in `fragment.h`, `te_callparser.h`, and `te_smlutil.h`: The Telos parser is invoked directly and the contents of the Telos frames is retrieved by navigating over a list of fragments (a fragment is a data structure for a Telos frame). See section **??** for details.
- Using the functions and data structures defined in `te_access.h`: Telos frames are represented in vectorized structure. One can use functions to create, destroy or apply to filters to the structure. See section **??** for details.
- Using the functions and data structures defined in `te_cursor.h`: Iterating over a set of Telos frames is done by using a cursor. See section **??** for details.

The documentation in the following sections has been generated with DOC++ (`http://docpp.sourceforge.net`).

## 4.1 fragment.h and te_callparser.h

### 4.1.1 Typedef: BindingList

**typedef struct bindingList BindingList**

**Description:**

A binding list represents the list of parameters in a derive expression

### 4.1.2 Typedef: ObjectIdentifier

**typedef struct objectIdentifier ObjectIdentifier**

**Description:**

An object identifier represents a Telos object name. It may be a simple object name, a derive expression, or a select expression.

### 4.1.3 Typedef: te_ClassList

**typedef struct classlist te_ClassList**

**Description:**

A class list is a list of object identifiers

### 4.1.4 Typedef: AttrClassList

**typedef struct attrclasslist AttrClassList**

**Description:**

An AttrClassList is a list of attribute categories. Attribute categories or simple labels.

### 4.1.5 Typedef: SpecObjId

**typedef struct specObjId SpecObjId**

**Description:**

Used only internally for extended syntax

### 4.1.6 Typedef: SelectExpB

**typedef struct selectexpb SelectExpB**

**Description:**

Used only internally for extended syntax

### 4.1.7 Typedef: Restriction

**typedef struct restriction Restriction**

**Description:**

Used only internally for extended syntax

### 4.1.8 Typedef: ObjectSet

**typedef struct objectset ObjectSet**

**Description:**

Used only internally for extended syntax

### 4.1.9 Typedef: PropertyList

**typedef struct propertylist PropertyList**

**Description:**

A property list is a list of attributes. Attributes have a label and a value. The member objectSet is used only in an extended syntax.

### 4.1.10 Typedef: AttrDeclList

**typedef struct attrdecllist AttrDeclList**

**Description:**

An AttrDeclList is a list of attribute declarations. It represents everthing between ”with” and ”end” in a Telos frame. One attribute declaration has a list of attribute categories and a list of properties (attribute definitions).

### 4.1.11 Typedef: te_SMLfragmentList

**typedef struct smlfragmentList te_SMLfragmentList**

**Description:**

A SMLfragmentList is a list of Telos frames. Each Telos frame has an object identifier (id). It may have in addition an inOmega class, a list of in-Classes, a list of isA-Classes, and an attribute declaration. Except id, all members may be NULL.

### 4.1.12 Typedef: FrameParseOutput

**typedef struct frameParseoutput FrameParseOutput**

**Description:**

FrameParseOutput is the structure returned by the function te_frame_parser. It contains either a list of fragments or information about the parse error.

| | |
|---|---|
| `te_SMLfragmentList* smlfrag` | *the list of fragments* |
| `int error` | *0 if ok, 1 if parse error, 2 if input is null* |
| `char* errortoken` | *If there was an parse error, this should indicate the token that caused the error.* |
| `int errorline` | *If there was an parse error, this should be the line number of the error.* |

### 4.1.13 Typedef: ClassListParseOutput

**typedef struct classlistParseoutput ClassListParseOutput**

**Description:**

ClassListParseOutput is the structure returned by the function te_classlist_parser. It contains either a list of classes or information about the parse error.

| | |
|---|---|
| `te_ClassList* classlist` | *A list of classes (object names)* |
| `int error` | *Non-zero if an error occured* |
| `char* errortoken` | *If there was an parse error, this should indicate the token that caused the error.* |
| `int errorline` | *If there was an parse error, this should be the line number of the error.* |

### 4.1.14 te_frame_parser

**FrameParseOutput* te_frame_parser (char* indata)**

**Parameters:**

| | |
|---|---|
| `indata` | a string containing the input frames |

**Returns:**

| | |
|---|---|
| `a` | pointer to a FrameParseOut structure |

**Description:**

Calls the Telos Parser to parse frames.

### 4.1.15 te_classlist_parser

**ClassListParseOutput* te_classlist_parser (char* indata)**

**Parameters:**

| | |
|---|---|
| `indata` | a string containing the object names |

**Returns:**

| | |
|---|---|
| `a` | pointer to a ClassListParseOut structure |

**Description:**

Calls the Telos Parser to parse a list of object names.

### 4.1.16 FragmentToString

**char* FragmentToString (te_SMLfragmentList *cursor**

**Description:**

Unparse a fragment list into a string

### 4.1.17 DestroySMLfrag

**void DestroySMLfrag (te_SMLfragmentList* fragment)**

**Description:**

Destroy a fragment list

### 4.1.18 Destroy_ClassList

**void Destroy_ClassList (te_ClassList* clist)**

**Description:**

Destroy a class list

## 4.2 te_access.h

### 4.2.1 Structure: te_AttrDecl

**struct te_AttrDecl**

**Description:**

This structure represents an attribute declaration. An attribute declaration is a list of attribute categories with a list of properties (label and values) that belong to these attribute categories.

| | |
|---|---|
| `char** aszCategory` | *contains the list of category labels* |
| `char** aszLabel` | *contains the list of property labels corresponding to* |
| `char** aszValue` | *the list of property values* |

### 4.2.2 Typedef: TAttrDecl

**typedef struct te_AttrDecl TAttrDecl**

### 4.2.3 Typedef: PAttrDecl

**typedef TAttrDecl* PAttrDecl**

**Description:**

The pointer for TAttrDecl

### 4.2.4 Typedef: VTelos

**typedef struct te_VectorizedTelosframe VTelos**

**Description:**

The type for te_VectorizedTelosframe

### 4.2.5 Typedef: PVTelos

**typedef VTelos* PVTelos**

**Description:**

A pointer to VTelos

### 4.2.6 Typedef: AVTelos

**typedef PVTelos* AVTelos**

**Description:**

An array of VTelos pointers

### 4.2.7 Structure: te_TelosReport

**struct te_TelosReport**

**Description:**

A te_TelosReport is a projection on certain attributes of a frame

| | |
|---|---|
| `char** aszLabel` | *List of labels in the report* |
| `char** aszValue` | *List of values in the report* |

### 4.2.8 Typedef: TReport

**typedef struct te_TelosReport TReport**

### 4.2.9 Typedef: PReport

**typedef TReport* PReport**

### 4.2.10 vt_createByFragment

**AVTelos vt_createByFragment (te_SMLfragmentList* fl)**

**Parameters:**

| | |
|---|---|
| `fl` | contains the fragment list in the way produced by the parser |

**Returns:**

| | |
|---|---|
| `NULL` | if the argument is NULL too, else the pointer to the vector tree structure |

**Description:**

Maps the given fragmentlist fl into a vector of frames.

### 4.2.11 vt_create

**AVTelos vt_create (char* szTelos)**

**Parameters:**

| | |
|---|---|
| `szTelos` | should be a string of correct Telos |

**Returns:**

| | |
|---|---|
| `NULL` | if the szTelos fails the parsing process else it contains the AVTelos with all its componends |

**Description:**

Maps the given Telos text szTelos into a vector of frames.

### 4.2.12 vt_destroy

**void vt_destroy (AVTelos avtFrames)**

**Parameters:**

| | |
|---|---|
| `avtFrames` | points to the vector tree structure |

**Description:**

Disposes the given vector tree.

### 4.2.13 rep_create

**PReport rep_create (PVTelos pvtFrame, char** aszCategories)**

**Parameters:**

| | |
|---|---|
| `pvtFrame` | points to a single Frame, which must exists@pararm aszCategories should be a NULL terminated vector of the categories to filter as a conjunction. |

**Returns:**

| | |
|---|---|
| `In` | each case a report will be created, even if the result is empty. |

**Description:**

Filters all attributes to those properties which belong to all given categories at the same time. Note: If there is a category wrong typed, it has the effect that result will always be an empty vector with NULL at index 0.

### 4.2.14 rep_destroy

**void rep_destroy (PReport prepReport)**

**Parameters:**

| | |
|---|---|
| `prepReport` | points to the report which should be disposed |

**Description:**

Disposes the given report structure. Should be called to free the result of rep_create.

### 4.2.15 getValueOfLabel

**char* getValueOfLabel (PVTelos pvtFrame, char* szLabel)**

**Parameters:**

| | |
|---|---|
| `pvtFrame` | points to a single frame, which should be analyzed |
| `szLabel` | contains the keyword, which should be searched in the labels |

**Returns:**

| | |
|---|---|
| `the` | string containing the value according to the given labelor NULL if no appropriate value was found |

**Description:**

A simple service routine, which support the access on frames.

### 4.2.16 getCategories

**char** getCategories (PVTelos pvtFrame)**

**Parameters:**

| | |
|---|---|
| `pvtFrame` | points to a single frame, which should be analyzed |

**Returns:**

| | |
|---|---|
| `array` | of strings with the categories |

**Description:**

Lists the categories as flat list, each elemant appears only once. The categories will be ordered by their appearance.

### 4.2.17 destroyASZ

**void destroyASZ (char** asz)**

**Parameters:**

| | |
|---|---|
| `asz` | the array of strings to be disposed |

**Description:**

Disposes an asz (array of strings) structure.

## 4.3 te_cursor.h

### 4.3.1 Structure: te_framecursor

**struct te_framecursor**

**Description:**

A cursor for a Telos frame

| | |
|---|---|
| `te_SMLfragmentList* flAll` | *Parsed telos frames in a fragment list* |
| `te_SMLfragmentList* flCur` | *fragment list cursor* |
| `te_ClassList* clCurOmega` | *cursor for inOmega* |
| `te_ClassList* clCurIn` | *cursor for in* |
| `te_ClassList* clCurIsA` | *cursor for isA* |
| `AttrDeclList* alCur` | *cursor for attribute declarations* |
| `AttrClassList* clCurCategory` | *cursor for attribute categories (within one attribute declaration)* |
| `PropertyList* plCur` | *cursor for property list (within one attribute declaration)* |
| `AttrDeclList* alChecked` | *internal filter: tests if alCur was checked* |

### 4.3.2 Typedef: TFrameCursor

**typedef struct te_framecursor TFrameCursor**

### 4.3.3 Typedef: PFrameCursor

**typedef TFrameCursor* PFrameCursor**

### 4.3.4 te_createCursor

**PFrameCursor te_createCursor ( te_SMLfragmentList* fl)**

**Description:**

Creates and initializes a cursor structure for the given smlfragmentlist. and returns a pointer to it.

### 4.3.5 te_destroyCursor

**void te_destroyCursor (PFrameCursor pfc)**

**Description:**

Deallocates a cursor structure

### 4.3.6 te_resetFrame

**void te_resetFrame (PFrameCursor pfc)**

**Description:**

Sets the frame cursor to the first frame and resets all sub cursors

### 4.3.7 te_nextFrame

**int te_nextFrame (PFrameCursor pfc)**

**Returns:**

`true` (non-zero) if there is a next element

**Description:**

Sets the frame cursor to the next frame in the list and resets the Omega, IsA, In, AttrDecl, Category and Property cursors.

### 4.3.8 te_retOID

**char* te_retOID (PFrameCursor pfc)**

**Description:**

Returns the OID of a frame as plain string, even if it is a select expression. Memory for this string has to be deallocated by the caller.

### 4.3.9 te_resetOmega

**void te_resetOmega (PFrameCursor pfc)**

**Description:**

Resets the Omega cursor

### 4.3.10 te_nextOmega

**int te_nextOmega (PFrameCursor pfc)**

**Returns:**

| | |
|---|---|
| `true` | (non-zero) if there is a next element |

**Description:**

Sets the omega cursor to the next element.

### 4.3.11 te_retOmega

**char* te_retOmega (PFrameCursor pfc)**

**Description:**

Returns the omega object as string

### 4.3.12 te_resetIsA

**void te_resetIsA (PFrameCursor pfc)**

**Description:**

Resets the Isa cursor

### 4.3.13 te_nextIsA

**int te_nextIsA (PFrameCursor pfc)**

**Returns:**

| | |
|---|---|
| `true` | (non-zero) if there is a next element |

**Description:**

Sets the IsA cursor to the next element.

### 4.3.14 te_retIsA

**char* te_retIsA (PFrameCursor pfc)**

**Description:**

Returns the IsA object as string

### 4.3.15 te_resetIn

**void te_resetIn (PFrameCursor pfc)**

**Description:**

Resets the In cursor

### 4.3.16 te_nextIn

**int te_nextIn (PFrameCursor pfc)**

**Returns:**

| | |
|---|---|
| `true` | (non-zero) if there is a next element |

**Description:**

Sets the In cursor to the next element.

### 4.3.17 te_retIn

**char* te_retIn (PFrameCursor pfc)**

**Description:**

Returns the In object as string

### 4.3.18 te_resetAttrDecl

**void te_resetAttrDecl (PFrameCursor pfc)**

**Description:**

Sets the attr decl block cursor to the first attr decl block in the current frame and resets the sub-cursors Category and Property

### 4.3.19 te_nextAttrDecl

**int te_nextAttrDecl (PFrameCursor pfc)**

**Returns:**

| | |
|---|---|
| `true` | (non-zero) if there is a next element |

**Description:**

Sets the Property cursor to the next Property class in the attr decl block frame and resets the Category and Property cursors.

### 4.3.20 te_resetCategory

**void te_resetCategory (PFrameCursor pfc)**

**Description:**

Resets the category cursor

### 4.3.21 te_nextCategory

**int te_nextCategory (PFrameCursor pfc)**

**Returns:**

`true` (non-zero) if there is a next element

**Description:**

Sets the category cursor to the next element.

### 4.3.22 te_retCategory

**char* te_retCategory (PFrameCursor pfc)**

**Description:**

Returns the category as string

### 4.3.23 te_resetProperty

**void te_resetProperty (PFrameCursor pfc)**

**Description:**

Resets the property cursor

### 4.3.24 te_nextProperty

**int te_nextProperty (PFrameCursor pfc)**

**Returns:**

`true` (non-zero) if there is a next element

**Description:**

Sets the category cursor to the next element.

### 4.3.25 te_filterPropertyByCategory

**int te_filterPropertyByCategory (PFrameCursor pfc, char* category)**

**Description:**

Lists all properties that are of the type ”category”. The usage of this function is similar to the function te_filterPropertyByCategories. The only diffrence is the simplicity of the second parameter for the case that you only need to filter with one category.

### 4.3.26 te_filterPropertyByCategories

**int te_filterPropertyByCategories (PFrameCursor pfc, char* categories[])**

**Description:**

Lists all properties that matches all types of categories If the categories are empty then any category matches. In difference to the nextXXX functions, these function should be called before the first access via te_retLabel or te_retValue, because it must search the first valid AttrDecl. This means at the beginning you should call: ”te_resetAttrDecl( ... );” AND ”te_filterPropertyByCategories( ... );”

### 4.3.27 te_filterPropertyByLabel

**char* te_filterPropertyByLabel (PFrameCursor pfc, char* )**

**Description:**

Lists the value of the property with label of the current frame. This results only one value which is a new created string. Note that the caller has to dispose the return value !

### 4.3.28 te_retLabel

**char* te_retLabel (PFrameCursor pfc)**

**Description:**

Returns the current label of the current property in the current decl block or NULL

### 4.3.29 te_retValue

**char* te_retValue (PFrameCursor pfc)**

**Description:**

Returns the current value of the current property in the current decl block or NULL

# Chapter 5

# Programming Interface for a Java Client

The Java Application Programming Interface (Java API) consists of a package for the communication with the ConceptBase server, and the Telos Parser which uses the Java Generic Library 3.1.0 of ObjectSpace Inc (`http://www.objectspace.com`). The Telos Parser was generated with the tool JavaCC of SunTest. All classes relevant to ConceptBase are in the packages under i5.cb.

This chapter gives only an overview on how to use the Java API of ConceptBase. Detailed documentation of the classes and their methods can be found in the API documentation generated by javadoc. This should be included in the package with programmers information, otherwise contact the ConceptBase Team (cb@i5.informatik.rwth-aachen.de).

The Java API for ConceptBase consists of three main packages:

**i5.cb.api** contains classes that handle the communication with a ConceptBase, e.g. create a connection, send messages, retrieve answers,

**i5.cb.telos.frame** contains a Telos parser to parse Telos frames and classes to represent the structure of Telos frames in Java, and

**i5.cb.telos.object** provides a one-to-one representation of the Telos objects of the ConceptBase server in a Java client. Methods provide facilities to retrieve all instances, subclasses, attributes, etc. of an object.

The preferred method for the interaction with ConceptBase is the usage of the package i5.cb.telos.object. The classes of the other packages can be used, too, but then more programming in your client application is required.

Some examples for the communication with ConceptBase can be found in the directory
`$CB_HOME/examples/Clients/JavaClient`.

## 5.1 Communication with ConceptBase: i5.cb.api

The main class of the package `i5.cb.api` is the class `CBclient`. The connection with a ConceptBase server can be established during the construction of an object of this class or with the method `enrollMe`.

This class has methods like `tell`, `untell`, `ask` etc. to perform the usual operations on the ConceptBase server. They return in most cases an object of the class CBanswer, which represents the answer delivered by the ConceptBase server. The methods and the structures are similar to the methods and structures defined in the C and C++ API.

Furthermore, several get...-methods allow to retrieve status information of the client object and some set-methods change some parameters of the client, e.g. the timeout value or the current module.

The class CButil contains some static methods for decoding and encoding of strings, so that they are accepted by ConceptBase.

The class CBterm is used only internally, it parses Prolog-like terms.

Nearly every method throws an exception if some unexpected error has occured during the operation. All exceptions are derived from the class i5.cb.CBException. The exceptions of the class CBIOException are thrown if, for example, the communication between client and server is broken or a timeout has occured. CBUtilExceptions are thrown if a string cannot be decoded or encoded.

## 5.2 Parsing Telos Frames: i5.cb.telos.frame

The package i5.cb.telos.frame provides all classes and methods that are necessary to parse (and unparse) Telos frames or list of Telos object names, and to represent frames and objects as Java objects.

The parsing of Telos frames or a list of object names requires two steps. First you have to construct a TelosParser object:

```
TelosParser tpParser=new
   TelosParser(new StringBufferInputStream(sFrame));
```

The constructor of TelosParser requires an InputStream object as parameter, therefore it is necessary to construct a StringBufferInputStream out of a String object.

In the second step, you have to call a method of TelosParser to start the parsing. Possible methods are:

**telosFrames** to parse a set of Telos frames,

**telosFrame** to parse one Telos frame, and

**objectNames** to parse a list of Telos object names.

The methods return TelosFrame(s) or ObjectName objects, that can be accessed with several methods. For details, see the API documentation or the examples provided in `$CB_HOME/examples/Clients/JavaClient`.

It is also possible to construct a TelosFrame object step by step, without parsing a string. This is shown in the method `test2` of ExampleParser.java. The TelosFrame class has a method toString which converts the TelosFrame into a string, which can be given as an argument to the tell method of CBclient.

## 5.3 ObjectBaseInterface: i5.cb.telos.object

This package provides methods and classes to represent Telos objects and sets of them as Java objects. There are two possible ways of using this package:

- *without a connection to a ConceptBase server:* Telos objects are created directly in the Java program by using the static methods `getIndividual`, `getSpecialization`, `getInstantiation` and `getAttribute`. Objects may be added to `ITelosObjectSets` that have been created by the `TelosObjectSetFactory`. Within an `ITelosObjectSet`, one can search for instances, subclasses, attributes, etc. of a Telos object.
- *with a connection to a ConceptBase server:* An instance of the class `ObjectBaseInterface` has to be created (using a CBclient object). Then, this object can be used to retrieve an object from the ConceptBase server (`getIndividual`), to list all instances of an object (`getAllInstancesOf`), to retrieve all attributes of an object (`getAttributesOf`), etc. The class `ObjectBaseInterface` is an implementation of `ITelosObjectSet`. Also, insertion and deletion of objects in the CB-server via the methods `add` and `remove` is possible. However, it is usually easier to construct a Telos frame and use the method `tell` of `CBclient` than constructing a set of Telos objects.

Note that the relationships of a Telos object to other Telos objects are specific to a Telos object set, e.g. X might be an instance of Y in one set and not in another set. Therefore, methods such as `getAllInstancesOf` are methods of the ITelosObjectSet and not of TelosObject.

The file ExampleOBI.java in `$CB_HOME/examples/Clients/JavaClient` contains uses the `ObjectBaseInterface` to test various operations.

# Appendix A

# Example C Client

This chapter explains the usage of the C programming interface with an example program called `LogClient`. This program is able to read the `OB.log` file created by ConceptBase server and performs the operations stored in this file.

The full source code of this program is in the directory `$CB_HOME/examples/Clients/LogClient`. The file `LogClient.c` contains beside the main program some little functions to read the log file. This source file must be compiled and linked together with a version of the ConceptBase library `libCB.a`. The file `MakeLogClient` is a makefile, which executes the necessary commands to compile and link the file with `gcc` on a Unix-platform. The following paragraphs explain only the important parts of the main program.

Before any functions of libCB can be used, one must include the header file `CBinterface.h`.

```
#include <CBinterface.h>

int main(int argc,char* argv[]) {

  int PortNr;
  char* HostName;
  char* UserName;
  Answer *ans;
  Server *gserver;
  char* command;
  char* arg;

  char   *ClientName  = "LogClient";
```

The variables `PortNr`, `HostName`, `UserName` and `ClientName` are initialized with the command line arguments and passed to the `connect_CB_server` function below. `ans` stores the answer of an operation with the ConceptBase server. `gserver` is a pointer to a `Server` structure which is filled by the connect-function. `command` is the command which has been read from the log file and `arg` is the argument for this command.

```
  /* Reading and checking command line arguments */
  /* ... */

  /* Connect to CBserver */
  connect_CB_server(PortNr, HostName, ClientName, UserName, &gserver);
  if (!gserver) {
      fprintf(stderr,"Connection failed!\n");
      return 1;
  }
```

The function connect_CB_server opens an IPC socket to the specified ConceptBase server and performs an ENROLL_ME method as described in chapter **??**. If the connection can be successfully established, the gserver variable points to the connected server. Otherwise, gserver will be NULL.

Now, the program begins to read the log file. As long as there are commands in the log file, the variable command points to a string containing the actual method and arg contains the arguments of this method.

Depending on the value of command, the program executes the corresponding function to pass the method with its arguments arg to the ConceptBase server. Possible values for command are e.g. tellCB, untell, ask_frames, ...

```
    /* Read commands from logfile until end of file */
    while(readLogCommand(fp,&command,&arg)) {

        /* Ask user, if the command should be executed */
        /* ... */

        /* Tell */
        if (!strcmp(command,"tell")) {
            printf("Telling: %s \n\n",arg);
            ans=tellCB( gserver, arg );
        }

        /* Untell */
        if (!strcmp(command,"untell")) {
            printf("Untelling: %s \n\n",arg);
            ans=untell( gserver, arg );
        }

        /* Tell Model */
        if (!strcmp(command,"tell_model")) {
            printf("Loading models: %s \n\n",arg);
            files=commaList2charArray(arg);
            ans=tell_model( gserver, files );
            for(i=0;i<MAX_FILES;i++) {
                if (files[i])
                    free(files[i]);
                free(files);
            }
        }
```

Note, that the tell and untell functions take a simple string containing frames as argument, whereas the function tell_model takes a list of filenames as argument. The frames are loaded from these files by ConceptBase and the told to the knowledge base.

Tell and untell operations return a pointer to an Answer object. For tell and untell, it is sufficient to check the completion value of the answer. The return_data can be ignored for these methods.

The following ask functions return also an Answer object. The answer of the query is stored in the field return_data, the completion is CB_OK, if the query could be evaluated. Otherwise the completion will CB_ERROR or CB_TIMEOUT.

```
        /* Ask objnames */
        if (!strcmp(command,"ask_objnames")) {
            printf("Ask (OBJNAMES): %s \n\n",arg);
            ans=ask_objnames( gserver, arg, "LABEL","Now" );
            printf("Answer: %s\n\n", ans->return_data);
        }
        /* Ask frames */
        if (!strcmp(command,"ask_frames")) {
            printf("Ask (OBJNAMES): %s \n\n",arg);
            ans=ask_frames( gserver, arg, "LABEL","Now" );
            printf("Answer: %s\n\n", ans->return_data);
        }

        /* Check completion */
        if (ans && ans->completion) {
            fprintf(stderr,
                    ">>> Server reports error on method: %s(%s)\n\n",
                    command,arg);
        }
```

When an error occured, i.e. `completion` is not zero (another value than `CB_OK`), than a error message is printed on the console. Perhaps, it is also useful to get the all error messages from ConceptBase server, but this is not done here[1].

```
    }
    /* Close connection to CBserver */
    disconnect_CB_server(gserver);

    return 0;
}
```

If the while-loop is finished the connection to the ConceptBase server can be closed with the function `disconnect_CB_server` and the program is finished.

[1]But that should be done in a *good* client program.

# Appendix B

# Syntax Specifications

## B.1 Syntax Specification for IPC messages

```
<ipcmessage>          ->  ipcmessage(<sender>,<receiver>,<method_and_args>).

<sender>              ->  IPCSTRING

<receiver>            ->  IPCSTRING

<method_and_args>     ->  <tell>
                      |   <untell>
                      |   <ask>
                      |   <hypoask>
                      |   <tellmodel>
                      |   <enrollme>
                      |   <cancelme>
                      |   <nextmessage>
                      |   <stopserver>
                      |   <reportclients>
                      |   <lpicall>

<tell>                ->  TELL , [ <telosframes> <modulearg> ]

<tellmodel>           ->  TELL_MODEL , [ <filelist> <modulearg> ]

<filelist>            ->  [ <ipcstringlist>  ]

<untell>              ->  UNTELL , [ <telosframes> <modulearg> ]

<ask>                 ->  ASK , [ <askargs> <modulearg> ]

<askargs>             ->  <query>  , <answerrep>  , <rollbacktime>

<query>               ->  FRAMES , <telosframes>
                      |   OBJNAMES , <objnames>

<objnames>            ->  IPCSTRING

<answerrep>           ->  IPCSTRING
```

```
<rollbacktime>        ->  IPCSTRING

<hypoask>             ->  HYPO_ASK , [ <telosframes>  , <askargs> <modulearg> ]

<enrollme>            ->  ENROLL_ME , [ <toolclass>  , <username> <modulearg> ]

<modulearg>-->',' IPCID
 | "empty"

<toolclass>           ->  IPCSTRING

<username>            ->  IPCSTRING

<cancelme>            ->  CANCEL_ME , [ ]

<nextmessage>         ->  NEXT_MESSAGE , [ <method>  ]

<method>              ->  "empty"
                      |   IPCID

<stopserver>          ->  STOP_SERVER , [ <method>  ]

<reportclients>       ->  REPORT_CLIENTS , [ ]

<lpicall>             ->  LPI_CALL , [ IPCSTRING ]

<telosframes>         ->  IPCSTRING

<ipcstringlist>       ->  IPCSTRING
                      |   <ipcstringlist>  , IPCSTRING

IPCSTRING             ->  everything enclosed in " except " and \,
                          which must be escape with \

IPCID                 ->  [a-zA-Z]+[a-zA-Z0-9_]*
```

## B.2 Syntax Specification for IPC answers

```
<ipcanswer>           ->  ipcanswer(<sender>,<completion>,<result>).

<sender>              ->  IPCSTRING

<completion>          ->  ok
                      |   error
                      |   not_handled

<result>              ->  IPCSTRING
```