ConceptBase V3.1 Programmers Manual

Stefan Eherer, Rainer Gallersdörfer
Informatik V (Informationssysteme)
RWTH Aachen
Ahornstr. 55
5100 Aachen, Germany
email: [eherer|gallersd]@informatik.rwth-aachen.de

This manual is intended as a description for developers of *ConceptBase V3.1* client programms which will be implemented in PROLOG or C. It is necessary to have an installed and valid *ConceptBase V3.1* license to use the provided librarys and test the clients that shall be realized. Furtheron you should have a *ConceptBase V3.1 User Manual* which describes the useful server methods in its chapter 4, because the *Programmers Manual* only extends the *User Manual*.

Contents

Appendices

abcd

1 Programming Interface for a PROLOG Client

For the communication between a PROLOG client and a CBserver we offer a programming interface based on the BIMprolog IPC package. With this interface one can immediately gain access to a CBserver since the message sending and receiving operations are already implemented. Please refer to the source code of the ExampleIpcClient (appendix 1) which illustrates how the below described parameters and procedures can be imported and applied.

The programming interface consists of the two modules `IpcParameters.pro` and `IpcChannelAccess.pro` which must be consulted by a client together with the module `BimIpc.pro` from $CB_HOME/CB_Exe/KSV_Exe directory.

`IpcParameters.pro` contains several parameters in the form of PROLOG facts specifying the interaction between a client and a CBserver :

`init_timeout(_t1),service_timeout(_t2),service_retry(_num)`

`_t1:` time in seconds your client uses to initalize its input port before a call to the CBserver takes place.

`_t2:` time in seconds your client waits for an answer from the CBserver.

`_num:` specifies how often your client tries to get an answer to a message from the CBserver.

Normally it is not necessary to change the value of these parameters.

`thisToolId(_tid), whatCBserverId(_sid), server_id(_pid)`

`_pid:` is the physical identifier of the CBserver your client is connected to. This parameter is used as first argument in IpcChannel_call/3 (see below) and in addition it signals if your client is currently connected to a CBserver (_pid <>unknown).

`_sid:` is the identifier of the CBserver to which your client is connected.

`_tid:` is the identifier which was assigned to your client by the CBserver after establishing a connection.

These parameters are updated automatically and can be imported whenever they are needed for sending new messages. Their initial status `unknown` should not be changed.

`whatIpcServer(_host,_port)`

`_host:` hostname and

`_port:` port number assigned to the CBserver to which a connection is to be established.

You can either define both parameters as default values or update them dynamically.

thisToolClass(_tc)

`_tc`: is the 'class' to which your client belongs.

This parameter can be used as argument in messages with method ENROLL_ME. The default value is an example from using `IpcParameters.pro` for the client ExampleIpcClient.

user_name(_name)

`_name`: contains the name of the user running the client.

This parameter can be used as argument in messages with method ENROLL_ME. The *ConceptBase* usage environment automatically assigns the login name of the user during startup.

The module `IpcChannelAccess.pro` implements sending and receiving messages and therefore provides the following three procedures:

IpcChannel_call (_servid, _ipcmessage, _ipcanswer)

`_ipcmessage`,
`_ipcanswer`: have formats as described in the ConceptBase V3.1 User Manual
`_servid`: is the identifier stored in `server_id/1` in `IpcParameters`

Calling this procedure results in sending `_ipcmessage` to `_servid` and waiting for `_ipcanswer`.

Some additional remarks regarding the behaviour of the `IpcChannel_call`:

-- Whenever a message with method ENROLL_ME is sent to the CBserver an existing connection `(server_id(_sid), _sid <> unknown)` to another server is cancelled. This means that a message with method CANCEL_ME is sent to this 'old' CBserver.

In addition the parameters `server_id`, `thisToolId`, `whatCBserverId` in `IpcParameters.pro` are updated automatically.

-- Whenever a message with method CANCEL_ME is sent to the CBserver, besides a disconnection to this CBserver, the parameters `server_id`, `thisToolId`, `whatCBserverId` in `IpcParameters.pro` will be updated with unknown.

connectCBserver (_host , _port)

This procedure establishes a connection to a CBserver by sending an `ipcmessage` with method ENROLL_ME to the CBserver running on machine `_host` with port number `_port`. The parameters for ENROLL_ME are taken from the facts `thisToolClass` and `user_name`. The parameter `whatIpcServer/2` in `IpcParameters` is updated according to the specification of `_host` and `_port`.

Example: Connecting from ExampleIpcClient to a CBserver running on a host named 'picasso' accessable via communication port 4001.

```
host: @ andorfer.
port: @ 4001 .

Connected to service (andorfer4001)
ipcanswer(CBserver#4703.0,ok,ExampleIpcClient#4703.1)
```

The corresponding message sent to CBserver would look like:

```
ipcmessage(unknown,_83,ENROLL_ME,[ExampleIpcClient,
example_user])
```

It should be noted that any host mentioned in `/etc/hosts` which supports the Internet and IPC protocols can be specified as the host of the server. This is also true for very remote hosts. We have experimented with a client in Germany and a server in Canada without any problems.

disconnectCBserver

This procedure closes a connection to a CBserver as follows:

-- The parameter `whatIpcServer/2` in `IpcParameters` is updated with `unknown`.

-- A message with method CANCEL_ME is sent to the CBserver connected to your client.

You must make sure that a termination of your client is always preceeded by a disconnection to the CBserver for reasons of program integrity.

To start a client `BimIpc.pro` must be consulted from directory $CB_HOME/CB_Exe/KSV_Exe by your client process.

Calling `ExampleIpcClient` would look like:

```
% cd $CB_HOME/CB_Exe/Examples/CLIENT-SERVER
% $BIM_PROLOG_DIR/bin/BIMprolog -Pq+ -Ps+  ExampleIpcClient
```

2 Programming Interface for a C Client

2.1 Communication between Client and Server

This chapter describes the data structures used by the API, in particular structures which are passed to and returned by the interface procedures. The C-library `libCB.a` contains all procedures described in section 2.2.

2.1.1 Data Structures

The following C-types are defined in the file `CB_interface.h`.

2.1.1.1 Completion

```
typedef enum {CB_OK, CB_ERROR, 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()` and pushed to the standard *ErrorWindow* calling `erw_SendStructError(em, 0)`(see below).

`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).

2.1.1.2 Answer

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

This format is used only by the general methods `send_message()` and `get_servermessage()`. 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 while the third one contains return values of the message called.

2.1.1.3 Server

```
typedef struct { char *server;
                 char *client;
                 int connected_to_CB_server;
                 Connection conn;
               } 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. This structure may be used to pass messages to any client connected to the *ConceptBase* server. To achieve this, the `Server->server` field have to be filled with the internal name of this client (the name given by the `connect_CB_server` routine). The `connection` field contains pointers to the communication channels between client and server and is used by the general `send_message` procedure only.

2.1.1.4 Telos_Objects

```
typedef struct telos_objects { char *telos_object;
                               struct telos_objects *next;
                             } Telos_Objects;
```

Simply a linked list to specify more than one object for the procedures `tell()` and `untell()`.

2.1.1.5 Clients

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

Result of the `report_clients` call.

2.1.1.6 Error_Messages

```
typedef struct errormessages { char *errormessage;
                               struct errormessages *next;
                             } 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 `em = get_errormessages()` and sent to the standard *ErrorWindow* calling `erw_SendStructError(em, 0)`.

2.1.1.7 Returnstrings

```
char *returnstrings[];
```

Array of constant strings which are used as results of the 'ask'-type routines to characterize specific error cases (will be changed to `Completion` type in the next version).

2.1.2 Procedures

2.1.2.1 General message

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

This procedure is the most general one. It sends a message of type `method` to the (already connected) *ConceptBase* server `server`. `data` is a string containing data expected by the method `method` (see chapter 4 of the *ConceptBase User Manual* for a complete description of the available methods and their expected data).

2.1.2.2 Procedures managing the connection to a ConceptBase server

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

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:`	Other error (not further specified)

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

Closes a previous connection to a *ConceptBase* server. This procedure has to be called every time a client is stopped. Otherwise the server's internal client tables will run out of space.

Input parameter:

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

Result:

`0`: Connection correctly terminated
`<>0`: Something went wrong

2.1.2.3 Procedures updating the content of the knowledge base

```
Completion
tell (Server *server, Telos_Objects *objects)
```

Inserts a set of objects into the *ConceptBase* server.

Input parameters:

server: pointer to the structure discribing the actual server
objects: pointer to a list of objects, which shall be inserted into the knowledge base. Note, that the strings have to be provided as a list of ASCII numbers to avoid parsing problems of the *ConceptBase* server.

Result:

`CB_OK`: operation sucessfull
`CB_ERROR`: There was an error while inserting, get the errormessages calling `em = get_errormessages()` and send them to the standard *ErrorWindow* calling `erw_SendStructError(em, 0)`
other: see the description in section 2.1

```
Completion
untell (Server *server, Telos_Objects *objects)
```

Removes a list of objects from the knowledge base. Note the specific semantics of the `untell` method as described in chapter 4 of the *ConceptBase User Manual*.

Input parameters:

server: pointer to the structure discribing the actual server
objects: pointer to a list of objects, which shall be untelled. Note, that the strings have to be provided as a list of ASCII numbers to avoid parsing problems of the *ConceptBase* server.

Result:

`CB_OK`: operation sucessfull
`CB_ERROR`: There was an error while removing, get the errormessages calling `em = get_errormessages()` and send them to the standard *ErrorWindow* by calling `erw_SendStructError(em, 0)`
other: see the description in section 2.1

2.1.2.4 Error handling procedures

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

Gets the errormessages corresponding to the last error. This procedure have 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 parameter:

server:	pointer to the structure discribing the actual server

Result:

list of errormessages

```
boolean
erw_SendStructError(Error_Messages *error_messages,
                    int             queuenumber )
```

Input parameter:

error_messages:	list of error messages returned by the server
queuenumber:	destination of the error messages 0: standard ErrorWindow (used for error messages) 1: standard AnswerDisplay (used to display answers of queries)

Result:

`TRUE`, if the sending was successful, `FALSE` otherwise

2.1.2.5 Query procedures

```
char *
ask (* Server *server, char *query)
```

Sends a query to the server and returns the answer of the server (not interpreted by the ask procedure). Note, that this procedure may receive a timeout which must be handled by the client.

Input parameters:

server:	pointer to the structure discribing the actual server
query:	the ASCII-representation of the query

Result:

string containing the answer of the server
If the first character of the answer is \240 (i.e. ASCII-code 160), the answer is a short error-description.

```
char *
hypo_ask (Server *server, char *query)
```

Asks the server for an answer of a query object with a temporarily told hypothesis (see chapter 4 of the *ConceptBase User Manual* for further details).

Input parameters:

server:	the server you want to ask
query:	string containing the appropriate data (see chapter 4 of the *ConceptBase User Manual* for further details)

Result:

string containing the answer of the server
If the first character of the answer is \240 (i.e. ASCII-code 160), the answer is a short error-description.

2.1.2.6 Privileged procedures

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

Returnes a list of all clients connected to the server. This procedure may be used to obtain the internal name of a client to which a message should be sent directly.

Input parameters:

server:	pointer to the structure describing the actual server

Result:

list of clients or `NULL` on error

2.2 Obtaining User Input

The interaction window is used to obtain user input (e.g. the portnumber and host name specifying the server the client wants to be connected to) in a clean way. To use these procedures within your tools you have to link your code against `libIntWin`.

2.2.1 Datatypes

The following datatypes are defined in the header file `iw_interact.h`:

```
struct interaction { char *prompt;
                     char *def;
                     boolean (*consistency_check)();
                     char *argument;
                    };
```

Used to define the properties of a line of the interaction window (see below). The fields have to be read as follows:

`prompt`: string, that is displayed in front of the input line; indicates what the user should type in (must not be `NULL`)

`def`: string which is presented to the user as a default value (may be `NULL`).

`consistency_check`: pointer to a procedure which provides a kind of consistency check of the user's input string (see the description of available procedures below to get a feeling what this mechanism is doing).

`argument`: a pointer which is passed to the consistency check as parameter (is not interpreted by the interaction window).

2.2.2 Interaction window procedure

The following procedure is the interface to the interaction window.

```
char **
interactionwindow(struct interaction **interactions)
```

This procedure creates a window, which is part of the calling process. In order to not disturb the interaction window a kind of semaphor guarantees that there is only one interaction window active at a given time.

Input parameter:

`interactions`: pointer to a `NULL`-terminated array of `struct interaction`.

Result:

pointer to an `NULL`-terminated array of strings which contains the inputs of the user; usually as much input strings are returned as prompts are given to the interaction window; `NULL` if the interaction window is terminated by pressing the quit button.

Example:

Lets say one wants to obtain a Telos object which must exist in the current knowledge base. First, the array of `struct interaction`s have to be declared:

```
struct interaction choice = {"object name", NULL,
                             ExistsObject, NULL},
                   choices[] = {choice, NULL};
```

The interaction window called with this structure will come up with a single line prompting the user for an object name. After typing in an arbitrary string and confirming (either pressing the confirm button of the window or pressing the return key twice) this string is passed to the `ExistsObject` consistency check procedure. This procedure asks the *ConceptBase* server whether the given object exists within the knowledge base. If so, the procedure returns `TRUE` and the string is returned to the procedure which has started the interaction window. If not, the procedure returns `FALSE` and the consistency check displays an appropriate error message within the lower area of the interaction window. Now, the user has to type in another object identifier which is then checked in the same way. Pressing the quit button of the interaction window terminates it and a `NULL` pointer is returned.

2.2.3 Consistency check procedures

The header file `iw_conchecks.h` declares a series of consistency check procedures which may be used by the programmer. In general a consistency check has to have two parameters: (1) a string containing the actual user input of the corresponding line in the interaction window and (2) a pointer to an argument that depends on the consistency check at hand. Note, that the consistency check itself has to interpret this pointer and the programmer is responsible that the given pointer is valid.

Note also, that the consistency checks need the following variables defined externally:

`Server *server` (description of an already connected *ConceptBase* server)
`char *host, *portnumber, *username` (to set up such a connection if there is none)

All of the following procedures call the procedure

```
void
Display_Errormessage(char * message)
```

to display an appropriate error message within the lower part of the interaction window. If a consistency check is called outside an interaction window the error messages are passed to the standard message interface of the Andrew Toolkit. Note, that the string passed to `Display_Errormessage` must be terminated by '`\n`'.

Currently, the following consistency checks are provided:

```
boolean
IntegerInRange(char *input, IntegerRange *range)

struct IntegerRange { int min;
                      int max;
                    };
```

Checks wether the input string is a number and is in the range specified by `range`, i.e. the following must hold:

```
range->min <= input <= range->max.
```

```
boolean
StringInSet(char *input, char **stringarray)
```

Checks whether the input string is one of the strings contained in the `NULL`-terminated array of strings `stringarray`.

```
boolean
ExistsObject(char *input, char *dummy)
```

Checks whether `input` specifies an object contained in the current knowledge base. Note, that this procedure uses the external variable `server`. The second parameter is not used and should therefore be `NULL`.

```
boolean
ExistsObjectOrString(char *input, char *string)
```

Checks first whether input equals `string`. If not, checks whether `input` specifies an object contained in the current knowledge base. Note, that this procedure uses the external variable `server`.

```
boolean
ExistsAndNotSExpr(char *input, char *dummy)
```

Checks whether the input string is **not** a valid select expression (see the *ConceptBase User Manual* for a specification of valid select expressions) and **is** contained in the current knowledge base. Note, that this procedure uses the external variable `server`. The second parameter is not used and should therefore be `NULL`.

```
boolean
IsInstanceOf(char *input, char **params)
```

Checks whether `input` is an object contained in the knowledge base and has an attribute link to destination `params[1]` with label `params[0]` (see the *ConceptBase User Manual* for a definition of the terms `attribute link`, `destination`, and `label`). Note, that this procedure uses the external variable `server`.

```
boolean
co_noObject(char *input, char *dummy)
```

Checks whether `input` is not contained in the current knowledge base. Note, that this procedure uses the external variable `server`.

2.3 The Standard *GraphBrowser*'s Interface

In this chapter we assume that you are familiar with the core concepts of the Andrew Toolkit. In particular, that you know what classes and insets are, also the terms dataobject and view. If you do not know these terms, refer to the Andrew Toolkit's documentation, namely the file `../andrew/doc/atk/progenv.doc`, and the examples located in `../andrew/examples`.
The Andrew classes *cbGraph* and *cbGraphView* together form an inset. The dataobject `cbGraph` stores information about objects and attributes between objects contained in the *ConceptBase* server, e.g. the internal ID of the object within the *ConceptBase* server or the graphical type of the object. The view `cbGraphView` provides methods for displaying and manipulating (moving nodes and links, selecting nodes and links etc.) a `cbGraph`.

In the following we describe these classes in more detail (subsection 2.3.1 and 2.3.2) and give you some hints how to extend the standard *Graphbrowser* of the *ConceptBase* client system by your own menu procedures and object descriptions (subsection 2.3.3).

2.3.1 cbGraph

2.3.1.1 Description

The class `cbGraph` stores the part of the semantic network to be displayed and provides methods for manipulating it. In particular, methods for inserting and deleting objects, accessing the fields stored in the instances of the class, setting some of these fields, and computing the neighbours of an object. Note, that you have to access the data fields only by the methods provided, otherwise you may run into trouble if the structure of the class changes.

In the following we first describe the meanings of the data fields of the class and then the methods provided.

2.3.1.2 Data

```
char *name
```
The label of the node or link as displayed in the *Graphbrowser*.

```
char *CB_ID
```
The internal ID of the node or link within the knowledge base server. Always use this field when passing the object to the *ConceptBase* server. This is because the label of object may not be unique. This differentiation is made because there may be situations where the unique ID is not wanted to be displayed as label of the object.

```
enum Shape           { rectangle, oval, circle, icon, link }
enum Pattern         { none, black, gray, dashed_link, double_link }
struct GraphicalType { enum Shape shape
                       enum Pattern pattern }
struct GraphicalType *graphicalType
```
The description of how to display the node or link. These fields are set by the insert method according to the specification given in the initialization file of the *Graphbrowser* (see the *ConceptBase User Manual* for a description of this file).

```
struct rectangle rect
```

Definition of the location of the object. You have to specify this by hand if you insert a single object by calling the insert method. If you use the `gb_parser` routine to insert more than one object at a time the locations of the objects are computed according to the table given in the *ConceptBase User Manual*. See subsection 2.3.3.2.1 for a description of the `gb_parser` routine.

```
struct cbGraph *source
struct cbGraph *target
```

Pointers to the source and target object of the current object. In case that the current object is a node these pointers are the same as the current object (imagine the node is a circular link with source and target are equal). In case that the current object is a link `source` and `target` point to the source and target object of the directed link (note that links are attributes in Telos and therefore directed).

```
struct cbGraph *next
```

Pointer to the next object of the `cbGraph`. By now the graph is stored as a simple linked list but this may change in the future.

```
boolean drawn
```

Currently not used, set to false during object initialization.

2.3.1.3 Methods

2.3.1.3.1 Classprocedures

```
cbGraph_InitializeObject(struct cbGraph *self) returns boolean
cbGraph_FinalizeObject(struct cbGraph *self)
```

Initialization and finalization methods as required by the Andrew Class System. The former sets default values for every data field of the new object (all pointers to `NULL`, `drawn` to `FALSE`, `rect` to `{0,0,0,0}`), the latter frees all allocated memory.

2.3.1.3.2 Methods

```
cbGraph_Insert(struct cbGraph *self, char *id, char *source,
               char *label, char *target, char *graphicalType,
               struct rectangle rect)returns struct cbGraph *
```

This method inserts a new object into the graph. Currently, the new object is appended as last object. `self` is a pointer to the first object of the linked list, it may be obtained by calling `cbGraphView_GetDataObject()`. `id` denotes the unique identifier of the object as given by the *ConceptBase* server. `label` denotes the label of the object as it will appear on the screen. `source` and `target` are the unique identifiers of the source and target object. In case that the object should be a node, `id`, `source`, and `target` must be the same string; this is the criterion the method uses to decide whether an object is a node or a link. `graphicalType` is the notion of the type as given in the initialization file (see the *ConceptBase User Manual* for a description of the initialization file). Usually, this is the `id` of a Telos class the object is an instance of. `rect` is a structure containing the coordinates of the location of the object. See the Andrew Toolkit documentation on rectangles for further information on filling the rectangle structure. In case that the object is a node the method passes the rectangle to the drawing routine of the Andrew Toolkit's class view. In case that the object is a link the `top` and `left` fields are interpreted as the starting point of the link whereas the `width` and `height` fields are interpreted as the end point of the link which gets the arrow. The procedure returns the new created object. If the object already exists in the graph, it is left unchanged.

```
cbGraph_Delete(struct cbGraph *self, char *object)
                returns struct cbGraph *
```

Deletes the object with `CB_ID` equals `object` from the graph pointed to by self. The remaining graph is returned. Note, that if you delete the first object of the linked list, the pointer you probably have passed to the `cbGraphView` which displays the graph is now undefined. So you have to be aware of passing the new pointer to your `cbGraphView`.

```
cbGraph_Neighbours(struct cbGraph *self, struct cbGraph *object)
                    returns struct list *
```

Computes the neighbours of `object`. Neighbours are all links which are linked to `object`. This method is used by the routines which are moving the objects around the screen. It is not recommended to use this method.

```
cbGraph_SetCB_ID(struct cbGraph *self, char *cb_id)
cbGraph_SetName(struct cbGraph *self, char *name)
```

Set the `id` or `name` resp. to `cb_id` or `name`.

2.3.1.3.3 Macromethods

```
cbGraph_GetNext(struct cbGraph *self)      (self->next)
cbGraph_GetName(struct cbGraph *self)      (self->name)
cbGraph_GetRect(struct cbGraph *self)      (self->rect)
cbGraph_GetSource(struct cbGraph *self)    (self->source)
cbGraph_GetTarget(struct cbGraph *self)    (self->target)
cbGraph_GetGraphicalType(struct cbGraph *self)  (self->graphicalType)
```

All these methods are used to access the data fields of a `cbGraph` object. It is highly recommended to only access the data fields via these methods.

2.3.2 cbGraphView

2.3.2.1 Description

The class `cbGraphView` maintains some data used for displaying the graph. In addition, it provides some methods which actually draw the graph, scroll the window, and so on. Note, that, usually, it should not be necessary for you to call one of these methods directly, they will be called by the *Graphbrowser* for you. Instead, you should call `cbGraphView_WantUpdate` (if you changed the underlying graph) or `cbGraphView_WantInputFocus` (if you changed the `mask` field).
According to subsection 2.3.1 we will now describe in detail the data fields and methods the class provides.

2.3.2.2 Data

```
struct menuview *menuview
```

A pointer to the list of menus the class maintains. This list will be set up once during start up of the *Graphbrowser*. You can extend this list via the initialization file (see the *ConceptBase User Manual* for further details). Usually, you should not touch this field.

```
struct cbGraph *currentObject
```

A pointer to the current selected object. This pointer may be used in procedures you provide for your menu items.

```
long x, oldx
long y, oldy
```

These are used for maintaining the status of the scrollbars. You should not touch these fields not to disturb the scrolling procedures.

```
long mask
```

Passed to the menuview, each time an object is selected and `cbGraphView_WantInputFocus` is called. You may find it useful for you to set this mask within your menu procedures to filter out existing or introduce new menu items. See the Andrew Toolkit documentation on `menulist` for further details on menu masks.

2.3.2.3 Methods

2.3.2.3.1 Classprocedures

```
cbGraphView_InitializeClass(struct cbGraphView *self) returns boolean
cbGraphView_InitializeObject(struct cbGraphView *self,
                             struct cbGraphView *cbgv)
                             returns boolean
```

Initialization and finalization methods as required by the Andrew Class System. The former binds some default menus for erasing nodes and links and calling other tools of *ConceptBase*' client system as is recommended by the Andrew Toolkit Programming Guide. The latter sets default values for every data field of the new object (`currentObject` to `NULL`, all longs to zero, and duplicates the list of menus bound in the `InitializeClass` method).

2.3.2.3.2 Overrides

```
cbGraphView_FullUpdate(struct cbGraphView *self,
                       enum view_UpdateType type,
                       long left,long top,long width,long right)
cbGraphView_Update(struct cbGraphView *self,)
```

Draw the actual graph according to the part of the whole drawing area which is currently visible. These methods are called by `cbGraphView_WantUpdate` or if window manager events occur (like resizing the window or deiconifying the window). You should not call these procedures directly.

```
cbGraphView_Hit(struct cbGraphView *self,
                enum view_MouseAction action,
                long x,long y,long numberOfClicks)
                returns struct view *
```

Is called every time you press, release, or drag the mouse. By default select the object nearest to the mouse pointer and highlights it. Also the `currentObject` field is set to this object. Again, you should not call this method directly.

```
cbGraphView_GetInterface(struct cbGraphView *self, int type)
                         returns char *
```

The interface provided by the class for the scollbar. This method is used only internally.

```
cbGraphView_ReceiveInputFocus()
```

Passes the mask field to the menuview and updates it.

2.3.3 Extending the Graphbrowser

In order to extend the *Graphbrowser*'s functionality by procedures of your own you have to perform the following steps:

1. Write your own Andrew class containing your procedures and
2. Modify the `.graphbrowserinit` file in one of your own directories.

In the following we will describe in more detail how to perform the first step (subsection 1). The second step is described in the *ConceptBase User Manual* (subsection 3.2.5). We give some hints how to use our procedures, and describe how to start the extended *GraphBrowser*.

2.3.3.1 Writing your own class

In this section we give you an algorithm how to write your own class and how to make a dynamic object out of it which can be loaded dynamically into the standard *Graphbrowser*.

1. <u>Class header file</u>
Firstly, you have to create a class header file containing the initialization procedures of the class.

Example: `myclass.ch`

```
#define myclass_VERSION 1

class myclass {
    classprocedures:
        InitializeClass() returns boolean;
        InitializeObject(struct myclass *self) returns boolean;
        FinalizeObject();
};
```

2. <u>Corresponding C file</u>
You then have to create a file containing the C code of the class methods and your own procedures. The latter two methods (`InitializeObject` and `FinalizeObject`) may be no-ops, but the `InitializeClass` method have to obtain a pointer to the `cbGraphView`'s classinfo structure and to enter all your own procedures into `cbGraphView`'s procedure table.

Example: `myclass.c`

```
#include "myclass.eh"
#include "cbgraphv.ih"
#include "cbGraph.ih"
#include "proctbl.ih"

/* example procedure, does nothing */
static void
myclass_proc(struct cbGraphView *self, long rock)
{
}
```

```
boolean myclass__InitializeObject(classID,self)
struct classheader *classID;
struct myclass *self;
{
    return TRUE;  /* always required */
}

void myclass__FinalizeObject(ClassID,self)
struct classheader *ClassID;
struct myclass *self;
{
}

boolean myclass__InitializeClass(ClassID)
struct classheader *ClassID;
{
    struct classinfo *cbGvtype = class_Load("cbGraphView");

    proctable_DefineProc("myclass-proc", myclass_proc, cbGvtype,
                         NULL, "currently does nothing");

    return TRUE;
}
```

See the Andrew documentation of `proctable` for a more detailed description of `proctable_DefineProc`. Note, that the heads of your procedures have to be of the same arity of our example procedure `myclass_proc`. This is, because the only way to call your procedure is to bind it to a menu item and then select this item.

3. Compiling your class

a) Precompile the header file by typing the following

```
$ANDREWDIR/bin/class/ -B -I$ANDREWDIR/include
-I$ANDREWDIR/include/atk myclass.ch
```

b) Compile your C file

```
cc -I$ANDREWDIR/include -I$ANDREWDIR/include/atk
-c myclass.c
```

c) Create a dynamic object

```
$ANDREWDIR/bin/makedo -d $(ANDREWDIR)/lib
-b $(ANDREWDIR)/bin -o myclass.do myclass.o
$ANDREWDIR/bin/doindex myclass.do
```

2.3.3.2 Communicating with the ConceptBase server

For communicating with the *ConceptBase* server within your procedures refer to chapter 2 of this volume. In particular the ask and tell procedures may be used by your routines. Within the standard *Graphbrowser* a global variable `server` is provided which can be passed as first parameter to the communication routines. This global variable will be set at start up time, so that there will be always a connection to the knowledge base server.

In the following we will describe how to get the right answer format containing all information needed for displaying the objects and some procedures which we provide for your convenience.

2.3.3.2.1 How to ask the server

To get all informations necessary as input for the `Insert` method you have to use the `get_links` query of the server. Look at the following example for the syntax (see also the description of the programming interface in chapter 2 of this volume and in the *ConceptBase User Manual*).

Example: Getting all generalizations of a given class `myclass`.

```
rd = ask(server, "get_links([going_into/dir,
                             myclass/source,
                             '*isa'/label,
                             1/dist]),
                             FORMAT:X11GraphBrowserEDGE")
```

`rd` will contain a list of all pairs of objects satisfying the query. Its format is as follows:

```
[edge(source1, label1, destination1),
 edge(source2, label2, destination2),
 ...]
```

where each $\texttt{source}_\texttt{i}$, $\texttt{label}_\texttt{i}$, $\texttt{destination}_\texttt{i}$ is of the following format:

`CB_Object(oid, s, l ,d, bt, tt, gt)` with

`oid`: identifier of the object (intended to be assigned to `CB_ID`)
`s`: source component of the proposition with id `oid`
`l`: label component of the proposition with id `oid`
`d`: destination component of the proposition with id `oid`
`bt`: belief time of the proposition with id `oid`
`tt`: transaction time of the proposition with id `oid`
`gt`: graphical type of the object as given in the init file or proposition if there is no match

There are two ways to insert such an edge list into an existing `cbGraph`. You may write your own routine which filters out the relevant data and pass it to the `Insert` method. Note, that you first have to insert the objects and then the link relating them. This is to guarantee that the source and target pointers of the `cbGraph` structure are well defined. Or you can use the `gb_parser` routine of the standerd *Graphbrowser*.

```
void gb_parser(struct cbGraphView *self,
               char *rd,
               int type)
```

where `rd` is the return string of the `ask` routine and `type` is an integer indicating where to draw the new nodes and what direction the links should have. Since this routine is designed for the standard *Graphbrowser* only the types described in the *ConceptBase User Manual* are supported. It is assumed that there is a unique start object (`self->currentObject`) and only objects directly linked to it are retrieved at a time. Currently, there is no routine to display whole subtrees. The following types are supported by now:

0: draw objects according to the table given in the *ConceptBase User Manual*
1: draw objects to the right, arrow at the right end. This type is used when expanding the attributes of a given object. Therefore, specialization and instantiation links are filtered out by this

2.3.3.2.2 Running the extended *Graphbrowser*

To run the new extended *Graphbrowser*, you have to perform the following steps:

1. Add the path of your own dynamic object to your `CLASSPATH` environment variable

2. Copy your own init file either in your current directory, your home directory, or into the directory where the *X11 Usage Environment* is installed (usually `$CB_HOME/CB_Exe/XI_Exe`), because these are the directories the *Graphbrowser* will look for it.

3. Start the *Graphbrowser* either from the *Toolbar* or directly from the shell:
   ```
   graphbrowser portnumber hostname username object
   ```
   where `portnumber` and `hostname` describe the *ConceptBase* server you want to connect to, `username` is your login name and `object` is the initial object you want to browse.

Appendix 1: Example Prolog Client

```
{
*
* File:        %M%
* Version:     %I%
* Creation:    30-Aug-1989, Manfred Jeusfeld (UPA)
* Last Change: %G%, Hans W. Nissen (RWTH)
* Release:     %R%
*
-----------------------------------------------------------------------------
*
* This module is an example for an interactive program that connects
* itself to the ConceptBase server as a client.
*
* Predicates playing an important role for the communication between client
* and server are marked by  '** begin **' and  '** end **'  .
*
}

:- module(ExampleIpcClient).

:- import IpcChannel_call/3 from IpcChannelAccess.
:- import connectCBserver/2 from IpcChannelAccess.
:- import disconnectCBserver/0 from IpcChannelAccess.

:- import thisToolClass/1 from IpcParameters.
:- import thisToolId/1 from IpcParameters.
:- import whatCBserverId/1 from IpcParameters.
:- import server_id/1 from IpcParameters.
:- import whatIpcServer/2 from IpcParameters.
:- import init_timeout/1 from IpcParameters.
:- import service_timeout/1 from IpcParameters.
:- import service_retry/1 from IpcParameters.
:- import user_name/1 from IpcParameters.
:- import client_report/1 from IpcParameters.

{  ... initializing the connection }

{  ... first try it with the parameters provided by IpcParameters.pro }

init_connection :-
  whatIpcServer(_host,_port),
  connectCBserver(_host,_port),
  !.

init_connection :-
  manual_connect.
```

```
{ ... then try it with your own input parameters and }
{  ... if the connection attempt failed try it again  ! }

 manual_connect :-
  nl,
{ ** begin ** }
  repeat,
  write('host: '), read(_host),
  write('port: '), read(_port),
  connectCBserver(_host,_port),
{ ** end ** }
  !.


{  ... disconnecting from ConceptBase server }

close_connection :-
{ ** begin ** }
  disconnectCBserver.
{ ** end ** }

{ ... looping, i.e. accepting requests from the terminal, sending them to }
{ the ConceptBase server and displaying the answer                        }

loop :-
  nl, write('Method: '), read(_method),
  write('Arguments ([a1,..,an]): '), read(_arglist),
  nl, handleIt(_method, _arglist),
  !,
  loop.


handleIt(exit,_) :- stop.

{ ... connecting to another server: }

handleIt(connect,_) :-
  close_connection,
  manual_connect ,
  !.


{  ... displaying parameters currently stored in module IpcParameters.pro:
}

handleIt(param,_) :-
  GetAndDisplayParam,
  !.
```

```
handleIt(_m, _a) :-
{ ** begin ** }
  server_id(_sid),
  thisToolId(_tid),
  whatCBserverId(_CBsid),
  IpcChannel_call(_sid, ipcmessage(_tid,_CBsid,_m,_a),
                        ipcanswer(_s,_c,_r)),
{ ** end ** }
  write('Responding tool: '),write(_s),nl,
  write('Completion:      '),write(_c),nl,
  write('Return value:    '),write(_r),nl,
  !.


GetAndDisplayParam :-
  whatIpcServer(_host,_port),
  server_id(_sid),
  thisToolId(_tid),
  whatCBserverId(_CBsid),
  thisToolClass(_ToolClass),
  init_timeout(_t1),
  service_timeout(_t2),
  service_retry(_num),
  user_name(_user),
  client_report(_on_off),

  write('host: '),write(_host),nl,
  write('port: '),write(_port),nl,
  write('serverID: '),write(_sid),nl,
  write('ToolID: '),write(_tid),nl,
  write('whatCBserverId: '),write(_CBsid),nl,
  write('thisToolClass: '),write(_ToolClass),nl,
  write('init_timeout: '),write(_t1),nl,
  write('service_timeout: '),write(_t2),nl,
  write('service_retry: '),write(_num),nl,
  write('user_name: '),write(_user),nl,
  write('client_report: '),write(_on_off),nl.


consult_IpcModules :-
  consult('$CB_HOME/Examples/CLIENT-SERVER/IpcParameters'),
  consult('$CB_HOME/IpcChannelAccess'),
  consult('$CB_HOME/BimIpc').


go :-
  consult_IpcModules,
  init_connection,
  loop,
  close_connection.

?-  go.
```