SIS - Application Programmatic Interface
Reference Manual

Version 2.2.2

Institute of Computer Science

Foundation for Research and Technology - Hellas

TABLE OF CONTENTS

1 Introduction

The Application Programmatic Interface (API) designed and implemented in ICS-FORTH offers a complete set of primitive query and update operators, which implement frequently used combinations of primitive operations on the server side for optimization. Stress was laid especially to traversal operations.

In this document we present the set of functions the API consists of, the functionality of the supported queries and at the end a complete example is presented: A TELOS model and description is shown and also the source code of an application which queries according to the model.

API libraries are available in PC versions in C++ and C (Borland 5.01 libraries and dll) and in Java[1]. In this document we present the C version of the SIS programmatic interface. The differences between C interface and the C++ and Java interfaces are presented in "Appendix D - C++ Programmatic Interface" and "Appendix E - Java Programmatic Interface".

2 Querying models

An application which wants to retrieve and/or modify information from a data base created with the TELOS language can use an application programmatic interface offering a collection of primitive access and update functions rather than accessing/updating the data of the base with primitive reads/writes from the files. This would require knowledge of how the objects are stored into the base something that is not required with the application programmatic interface (hereafter called API).

There are two implementations of the API, fully compatible one to the other, meaning that an application which uses the API can be compiled with any of the two libraries which implement the same functions with the same functionality but with a difference described below (see figure 1).

[1] There were also available in UNIX platforms (e.g. Solaris, AIX, etc.) but due to luck of market interest the porting to the platforms above was abandoned.

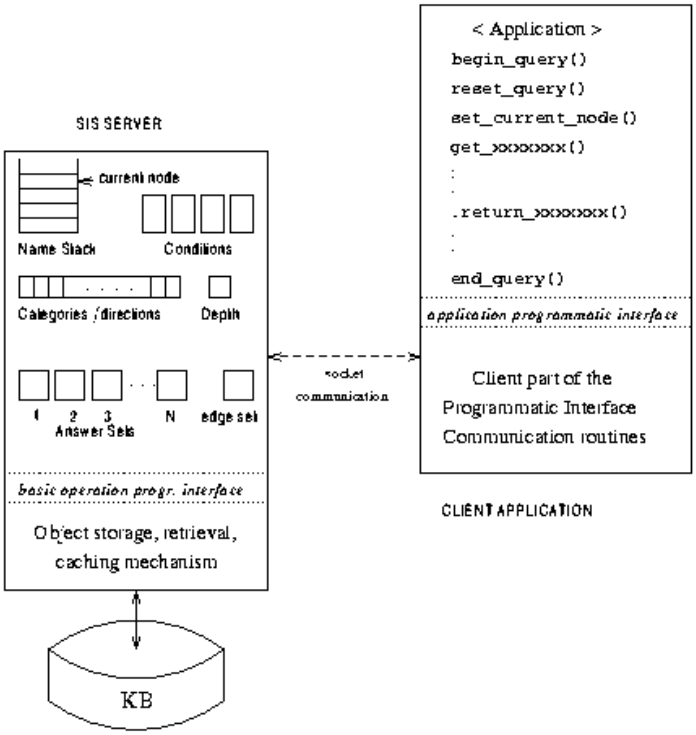

Figure 1: The query model

SIS server is a process running independently and serving a client application. The client-server communication is achieved through sockets. The client application is linked only with the client part of API which simply sends the query /update request to the server where the query /update processing is done.

2.1 Client-server model to access the SIS base

With the first implementation of the API the application is the client of a client-server model of querying and/or modifying the SIS base. This means that a second process (the SIS server) has to run in parallel to the application and this process gives the answers to any question from the application, and modifies the data in the SIS base. The server is responsible for reading and writing the SIS base files.

The main advantage of this model is that the client (or clients) need not run on the same machine the SIS server is running. The communication between the client process and the server process is achieved through Windows sockets.

2.2 Immediate access the SIS base

With the other implementation of the API the application does not communicate with some other process to retrieve the information but the query and update functions themselves access the data from the SIS base. This implementation eliminates the delay on the socket communication but creates a bigger process both in disk demands (executable file) and in run-time memory requirements since the process itself reads the whole SIS base.

3 Functionality of the query and update functions

The answer to most of the queries on an object is a set of objects, that is the query for the instances of an object A is a set of all the objects that are instances of object A, the query for the superclasses of an object B is a set of all the objects that are superclasses of object B etc. So, for most queries, there is an object that the query applies on and an answer set, the result of the query.

In order to avoid long traffic through the sockets, those functions, which should return a set of answers, operate like this: When called, they calculate the answer and locate it into an answer set. The return value is `APIFail`(-1) if the query failed or a positive integer on success. This positive integer is the identifier of the answer set (hereafter will also be called temporary set). Most of these queries take as argument an integer, which specifies the object where the query is going to apply on. If this argument is 0 the query is applied on the current node otherwise it is applied on each object in the answer set with identifier the one passed as argument.

Before calling a query function asking to apply on the current node, an object must be set as current node with the **set_current_node**() or **set_current_node_id**() function.

With the use of temporary set at the server site, the client doesn't get the answer immediately but only when this is asked explicitly with some other function. This gives the possibility for the querying functions to apply on to the objects that exist into a temporary set. The answer is calculated by the server and then located into another temporary set and the identifier of this new set is returned. With this possibility the client doesn't have to get intermediate results, which can reside at the server site for further processing.

4 Name stack

At the beginning of a query session there is no current node. Before calling any querying function, a current node must have been set.

In TELOS language, links are objects themselves and may have their own links, so there is the possibility to set as current node a link object but since the logical names of the attributes of a class are unique for the class but not for the whole SIS base (two different classes may have attributes with the same logical name) there is a name scope problem with the **set_current_node**() function.

For this reason an object stack is maintained. At the beginning of a query the stack is empty. When the stack is empty the call of **set_current_node**() function must have as argument the logical name of a node object (the name scope is the whole SIS base) and then this object is pushed onto the stack (at the top). Any subsequent call of **set_current_node**() function has as name scope the current node, that is the object at the top of stack. So it has to have as argument the logical name of a link of the current node and this link object is pushed onto the stack.

With this mechanism the stack is either empty or contains objects where the first (at the bottom) is a class object and the others are link objects and each of them is attribute of the object located one level below on the stack.

There are functions (described later) for manipulation of this stack and what is important to remember is that when we want to set as a current node another class we have to empty the stack first.

The **set_current_node_id**() function takes as argument an integer and sets as current node the object with system identifier the given integer. The **set_current_node_id**() function resets the name stack and if the new object is link, the name stack contains the start node object and the subsequent link objects till the new current node, as if **set_current_node**() had been used.

5 Programmatic scenario

An application that uses the API to query the SIS database should, in general terms do the following:

1. Create a session to connect with the SIS database with **create_SIS_CS_Session**() function.
2. Establish the connection with the SIS database with **open_connection**() function.
3. Start a query or a transaction session to access or modify the SIS data with **begin_query**() and **begin_transaction**() functions.
4. Set a current node with the **set_current_node**() function.
5. Use a set of query/update functions (described bellow), which retrieve information and collect the answer into temporary sets at the server-site or modify information.
6. Repeat steps 4 and 5 and optionally performs some set operations (described bellow) on the temporary sets.
7. Terminate the query or the transaction session with **end_query**() or **end_transaction**() functions.
8. Terminate connection with the server with the **close_connection**() function.
9. Release the session with **release_SIS_Session**() function.

6 Functions

Here follows the list of functions that constitute the API. In "*Appendix A – C-API function declaration*" there is a list of the definitions of all these functions as well as the definitions of the data types that appear in their argument list. The functions can be categorized according to their functionality.

6.1 Query and transaction sessions

These functions are used to determine the SIS database to connect with, and initialize the API global structures. They also establish the connection with the SIS database and start a query or a transaction session to access or modify the SIS data.

> The functions return `APISucc`(0) on success and `APIFail`(-1) on error (except if stated otherwise).

If the application is linked with the second implementation of API (direct access to the SIS database) some of these functions are not useful but are supported for full compatibility with the first implementation (client-server model).

6.1.1 Creating sessions to interact with a SIS data base

In order to provide real multi-threading to the clients that were using the SIS C-API, we introduced the notion of sessions. The application developer that needs to provide multi-thread access to different servers or the same server (making simultaneous queries or updates) should create multiple sessions to implement this.

The following functions are responsible for selecting the SIS database to establish a communication with. The hostname of the query server is needed along with a port number.

> *NOTE*: *In the following sections all functions described take as first argument the sessionID of the session (access point to the database) that they are quering or updating.*

int create_SIS_CS_Session(*int *sessionID, char *serv_host, int serv_port, char *DBUserName, char *DBUserPassword*)

Creates a session and sets the hostname and the port that this querying session will interact with. The variable *sessionID* will contain new session id. If this function is not called, any other call to some other function of the API will fail. The DBUserName and DBUserPassword will be used in the future to validate the requested connection. Currently they are not used but the should not be `NULL`.

In case the application is linked with the second implementation of API (direct access to the SIS database) the following functions are used. Note that, in this case, only one session should be created.

void get_db_dir(*char **db_dir*)

This utility function reads the environment variable DB_DIR into the string *db_dir*. The space for the *db_dir* string is expected to be already allocated. In case of failure forces the application to exit.

int *init_start_telos*(*char *db_dir, int write_permission)*

Creates an access link to the SIS database. It returns the pointer *start_t* (access link to the SIS database). This link should be used us argument to **create_SIS_SA_Session**() function. *Write_permission* is an optional argument that enables the write access (1) or the read-only access (0) to the SIS database. If it is missing the read-only access is established. In case of failure it returns `NULL`.

int create_SIS_SA_Session(*int *sessionID, int *start_t, char *serv_host, int serv_port, char *DBUserName, char *DBUserPassword*)

Creates a session using the argument *start_t* as an access link to the SIS database. The variable *sessionID* will contain new session id. If this function is not called, any other call to some other function of the API will fail. The hostname and the port (*serv_host, serv_port*) are used to get lock permission from the query server. Note that a query server should be activated in order to provide locking to the database. The DBUserName and DBUserPassword will

be used in the future to validate the requested connection. Currently they are not used but the should not be `NULL`.

int release_SIS_Session(*int sessionID*)

This function should be called to release the session (*sessionID*) created by **create_SIS_CS_Session**() or **create_SIS_SA_Session**().

6.1.2 Connecting to the database

int open_connection*(int sessionID)*

This function is used to open a connection to the SIS server determined by **create_SIS_CS_Session** or **create_SIS_SA_Session** function. With the first implementation described, this function opens the communication socket that establishes the communication with the server. With the second implementation this function just returns successfully. Between an **open_connection**() and a **close_connection**() any number of calls to **begin_query**()/**end_query**() and **begin_transaction**/**end_transaction**() can be performed.

int close_connection *(int sessionID)*

This function is used to close a connection to a server, previously opened with **open_connection**(). With the first implementation described this function closes the communication sockets, that is it terminates the communication with the server. With the second implementation this function just returns successfully.

6.1.3 Performing queries and transactions

int begin_query*(int sessionID)*

Starts a querying session. During this session all API query functions can be called to retrieve information from the SIS base. Possible return values are: `APIFail`(-1), `API_DB_CHANGED`(0), `API_DB_NOT_CHANGED`(1), `API_HASH_TABLES_NEED_EXPANSION`(4), `API_HASH_TABLES_EXPANDING`(8).

int end_query*(int sessionID)*

Ends a querying session.

int begin_transaction*(int sessionID)*

Starts a transaction session. During this session API query and update functions may be called to retrieve and modify information from the SIS base. The system will not permit operations that will leave the database inconsistent. Possible return values are: `APIFail`(-1), `API_DB_CHANGED`(0), `API_DB_NOT_CHANGED`(1), `API_HASH_TABLES_NEED_EXPANSION`(4), `API_HASH_TABLES_EXPANDING`(8).

int end_transaction*(int sessionID)*

Ends a transaction session by committing the changes to the database files. It returns 1 on success, `APIFail`(-1) on failure. In case of failure the operations

that took place between a **begin_transaction**() and an **end_transaction**() have not been committed.

int abort_transaction*(int sessionID)*

Aborts a transaction session. It returns 1 on success, `APIFail`(-1) on failure. In case of failure the operations that took place between a **begin_transaction**() and an **abort_transaction**() have not been committed.

6.1.4 Using the locking mechanism

int get_writelock*(int sessionID)*

Gets writelock on the database. It returns `APIFail` on failure, `API_DB_CHANGED` if the database is changed since last update, `API_DB_NOT_CHANGED` if database has not changed.

int get_readlock*(int sessionID)*

Gets readlock on the database. It returns `APIFail` on failure, `API_DB_CHANGED` if the database is changed since last update, `API_DB_NOT_CHANGED` if database has not changed.

int release_lock*(int sessionID)*

Releases the lock got on the database with functions **get_writelock**() or **get_readlock**(). It returns `APIFail` on failure, `API_DB_CHANGED` if the database is changed since last update, `API_DB_NOT_CHANGED` if database has not changed.

6.2 Set global parameters

There is a set of functions that set or alter values to global parameters that effect or are used by the query functions. Such parameters are the current node where a query function may apply on, the link categories that recursive queries use for traversing link etc.

Most of the functions return `APIFail`(-1) on error and `APISucc`(0) on success, except the ones that set the current node, which return the system identifier of the new current node that has been set.

int reset_query*(int sessionID)*

Reset all variables at server site so that everything is at the state they are when starting a new query session. Pop everything out of the name stack and free all temporary sets.

int set_current_node*(int sessionID, l_name node*)

Set current node the object with logical name *node*. If the name scope is the whole SIS base (default at the beginning of a query session or result of **reset_name_scope**() call) the argument must be the name of a node object otherwise it must be the logical name of a link object pointing from the current node. The function returns the system id of the current node or `APIFail`(-1) on failure.

int set_current_node_id*(int sessionID, int nodeid*)

Set current node the object with system identifier *nodeid*. This object is now at the top of the name stack and if it is a link object the name stack is changed properly (see section 4 where name stack is described).

int reset_name_scope*(int sessionID)*

Set current name scope the whole SIS base. The next **set_current_node**() will refer to a class node.

int pop_name_scope*(int sessionID)*

Go one level below the current name scope. Useful when you have set for example as current node an attribute of some class and want to refer to (set current node) another attribute of the same class.

int set_categories*(int sessionID, categories_set categs*)

Set the categories for any subsequent call of some special recursive query. Whenever this function is called, any previous set of categories is deleted. The data type used for defining the categories is described in section 6.8.1.

int set_depth*(int sessionID, int depth*)

Set the depth the recursive queries are going to traverse links, starting from a node object. If *depth* is a negative integer recursive queries traverse links with no limit to depth.

6.3 Queries

Here follows the set of functions that query the SIS data base and calculate and answer. The first group consists of very simple queries that give the system identifier of an object if we know the logical name and vice versa. The other two groups consist of functions that apply on the current node or on each object in a temporary set, calculate an answer and put it into a new temporary set.

6.3.1 Low level queries

These functions return `APISucc`(0) on success and `APIFail`(-1) on error.

int get_classid*(int sessionID, l_name lname, int *sysid*)

Given the logical name *lname* of a node object, the function returns the system identifier *sysid* of this object.

int get_linkid*(int sessionID, l_name fromcls, l_name label, int *sysid*)

Given the logical name *label* of a link object, the function returns the system identifier *sysid* of this object. Since the logical name of a link object may not be unique in the SIS base, the logical name *fromcls* of the node object the link is pointing from must be given.

int get_loginam*(int sessionID, int sysid, l_name lname*)

Given the system identifier *sysid* of an object, the function returns the logical name *lname* of this object.

6.3.2 Simple queries

The following group of queries applies either on the current node or on each object in a temporary set and store the answer in a temporary set. These queries calculate the answer set with information that exists in the object they apply on or in objects that are connected with immediate links with the applying object except some queries that have to calculate the isA closure of an object.

These functions return `APIFail(-1)` on error or the set identifier of the temporary set where the answer is stored.

int get_classes*(int sessionID, int set_id*)

If *set_id* is 0, get classes of which the current node is instance of. The return value is the descriptor of the answer set that contains the objects (system identifiers) of the classes of the current node.

If *set_id* is a positive integer, apply **get_classes**() on each object in temporary set *set_id*.

int get_all_classes*(int sessionID, int set_id*)

If *set_id* is 0, get classes of the ISA transitive closure of the classes of which the current node is instance of. The answer set contains the system identifiers of these classes.

If *set_id* is a positive integer, apply **get_all_classes**() on each object in temporary set ***set_id***.

int get_Sysclass*(int sessionID, int set_id*)

If *set_id* is 0, get system class the current node is instance of. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_Sysclass**() on each object in temporary set *set_id.*

int get_all_Sysclasses*(int sessionID, int set_id*)

If *set_id* is 0, get all system classes of which the current node is instance of. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_all_Sysclass**() on each object in temporary set *set_id*.

int get_instances*(int sessionID, int set_id*)

If *set_id* is 0, get the instances of the current node. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_instances**() on each object in temporary set *set_id*.

int get_all_instances*(int sessionID, int set_id*)

If *set_id* is 0, get all the instances of the current node calculating the instances of all subclasses. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_all_instances**() on each object in temporary set *set_id*.

int get_class_attr*(int sessionID, int set_id*)

If *set_id* is 0, get the class attributes that are instances of the current category. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_iclass_attr**() on each category in temporary set *set_id*.

int get_all_class_attr*(int sessionID, int set_id*)

If *set_id* is 0, get all the class attributes that are instances of the current category calculating the instances of all subclasses. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_all_class_attr**() on each category in temporary set *set_id*.

int get_superclasses*(int sessionID, int set_id*)

If *set_id* is 0, get the classes that the current code is isA of. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_superclasses**() on each object in temporary set *set_id*.

int get_all_superclasses*(int sessionID, int set_id*)

If *set_id* is 0, get the classes of the isA transitive closure of the classes the current no`de is isA of. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_all_superclasses**() on each object in temporary set *set_id*.

int get_all_Syssuperclasses*(int sessionID, int set_id*)

If *set_id* is 0, get the system classes of which the current node is isA of. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_all_Syssuperclasses**() on each object in temporary set *set_id*.

int get_subclasses*(int sessionID, int set_id*)

If *set_id* is 0, get the classes which are isA of the current node. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_subclasses**() on each object in temporary set *set_id*.

int get_all_subclasses*(int sessionID, int set_id*)

If *set_id* is 0, get the classes of the subclass-transitive closure of the classes that are isA of the current node. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_all_subclasses**() on each object in temporary set *set_id*.

int get_link_from*(int sessionID, int set_id*)

If *set_id* is 0, get the links pointing from the current node. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_link_from**() on each object in temporary set *set_id*.

int get_class_attr_from*(int sessionID, int set_id*)

If *set_id* is 0, get the class attributes pointing from the current node. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_class_attr_from**() on each object in temporary set *set_id*.

int get_inher_link_from*(int sessionID, int set_id*)

If *set_id* is 0, get the links pointing from the current node and the links inherited from all superclasses or current node.

If *set_id* is a positive integer, apply **get_inher_link_from**() on each object in temporary set *set_id*.

int get_inher_link_to*(int sessionID, int set_id*)

If *set_id* is 0, get the links pointing to the current node and the links inherited from all superclasses or current node and pointing to them.

If *set_id* is a positive integer, apply **get_inher_link_to**() on each object in temporary set *set_id*.

int get_inher_class_attr*(int sessionID, int set_id*)

If *set_id* is 0, get the class attributes pointing from the current node and the class attributes inherited from all superclasses or current node.

If *set_id* is a positive integer, apply **get_inher_class_attr**() on each object in temporary set *set_id*.

int get_link_to*(int sessionID, int set_id*)

If *set_id* is 0, get the links pointing to the current node. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_link_to**() on each object in temporary set *set_id*.

int get_category_from*(int sessionID, int set_id*)

If *set_id* is 0, get the categories of the links pointing from the current node. The answer set contains the system identifiers of the classes of the links pointing from the current node.

If *set_id* is a positive integer, apply **get_category_from**() on each object in temporary set *set_id*.

int get_category_to*(int sessionID, int set_id*)

If *set_id* is 0, get the categories of the links pointing to the current node. The answer set contains the system identifiers of the classes of the links pointing to the current node.

If *set_id* is a positive integer, apply **get_category_to**() on each object in temporary set *set_id*.

int get_link_from_by_category*(int sessionID, int set_id, l_name fromcls, l_name categ*)

If *set_id* is 0, get the links pointing from the current node and are instances of the category given. The category is defined by the name of the link and the class of which it is pointing from. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_link_from_by_category**() on each object in temporary set *set_id*.

int get_link_from_by_meta_category*(int sessionID, int set_id, l_name fromcls, l_name categ*)

If *set_id* is 0, get the links pointing from the current node and are instances of some links class that is instance of the category given. The meta-category is defined by the name of the link and the class of which it is pointing from. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_link_from_by_meta_category**() on each object in temporary set *set_id*.

int get_link_to_by_category*(int sessionID, int set_id, l_name fromcls, l_name categ*)

If *set_id* is 0, get the links pointing to the current node and are instances of the category given. The category is defined by the name of the link and the class of which it is pointing from. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_link_to_by_category**() on each object in temporary set *set_id*.

int get_link_to_by_meta_category*(int sessionID, int set_id, l_name fromcls, l_name categ*)

If *set_id* is 0, get the links pointing to the current node and are instances of some links class that is instance of the category given. The meta-category is defined by the name of the link and the class of which it is pointing from. The answer set contains the system identifiers of these links.

If *set_id* is a positive integer, apply **get_link_to_by_meta_category**() on each object in temporary set *set_id*.

int get_category_of_link_from*(int sessionID, int set_id, l_name label*)

If **set_id** is 0, get category of the label given. The answer set contains the system identifiers of the links which the given label is instance of.

If *set_id* is a positive integer, apply **get_category_of_link_from**() on each object in temporary set *set_id*.

int get_to_node*(int sessionID, int set_id*)

If *set_id* is 0, get all objects that are connected to the current node by links pointing to them. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_to_node**() on each object in temporary set *set_id*.

int get_from_node*(int sessionID, int set_id*)

If *set_id* is 0, get all objects that are connected to the current node by links pointing from them. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_from_node**() on each object in temporary set *set_id*.

int get_to_node_by_category*(int sessionID, int set_id, l_name fromcls, l_name categ*)

If *set_id* is 0, get all objects that are connected to the current node by links pointing to them. Links must be instances of the category given. The category is defined by the name of the link and the class of which it is pointing from. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_to_node_by_category**() on each object in temporary set *set_id*.

int get_from_node_by_category*(int sessionID, int set_id, l_name fromcls, l_name categ*)

If *set_id* is 0, get all objects that are connected to the current node by links pointing from them. Links must be instances of the category given. The category is defined by the name of the link and the class of which it is pointing from. The answer set contains the system identifiers of these objects. In the case of class attributes the returned objects are the computed instance set.

If *set_id* is a positive integer, apply **get_from_node_by_category**() on each object in temporary set *set_id*.

int get_to_node_by_meta_category*(int sessionID, int set_id, l_name fromcls, l_name categ*)

If *set_id* is 0, get all objects that are connected to the current node by links pointing to them. Links must be instances of some link class that is instance of

the category given. The meta-category is defined by the name of the link and the class of which it is pointing from. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_to_node_by_meta_category**() on each object in temporary set *set_id*.

int get_from_node_by_meta_category*(int sessionID, int set_id, l_name fromcls, l_name categ*)

If *set_id* is 0, get all objects that are connected to the current node by links pointing from them. Links must be instances of some link class that is instance of the category given. The meta-category is defined by the name of the link and the class of which it is pointing from. The answer set contains the system identifiers of these objects.

If *set_id* is a positive integer, apply **get_from_node_by_meta_category**() on each object in temporary set *set_id*.

int get_from_value*(int sessionID, int set_id*)

Get the objects that the link objects in temporary set *set_id* are pointing from. The answer set contains the system identifiers of these objects.

If *set_id* is 0, apply **get_from_value**() on current node.

int get_to_value*(int sessionID, int set_id*)

Get the objects that the link objects in temporary set *set_id* are pointing to. The answer set contains the system identifiers of these objects, or/and primitive values.

If *set_id* is 0, apply **get_to_value**() on current node.

6.3.3 Logical expressions - Object set filtering

There are cases that you want to select some objects according to some condition. For this reason API offers the following function:

int get_filtered*(int sessionID, int set_id*)

From all objects in the set *set_id* select only those object that satisfy a condition that has been previously defined with the **set_filter_cond**() function, described later. The answer set contains all these selected objects.

If *set_id* is 0, evaluate the condition for the *current node* and if it is true put *current node* in answer set else return an empty set. The function returns `APIFail`(-1) on error or the set identifier of the temporary set where the answer is stored.

The condition used by **get_filtered**() function can be defined with the following function:

int set_filter_cond(*log_exp*)

Define a logical expression that can be true of false for a specific object.

The way to construct logical expressions is described in the following sections.

6.3.3.1 Expressions returning TRUE/FALSE

The argument to the above functions is an API logical expression function that returns true of false. Such functions are the following:

int SUCC*(sessionID)*

Always succeeds (TRUE).

int FAIL*(sessionID)*

Always fails (FALSE).

int AND*(sessionID, log_exp1, log_exp2*)

Logical AND between logical expressions *log_exp1* and *log_exp2*.

int OR*(sessionID, log_exp1, log_exp2*)

Logical OR between logical expressions *log_exp1* and *log_exp2*.

int NOT*(sessionID, log_exp*)

Logical NOT of logical expression *log_exp*.

int BELONGS*(sessionID, sys_id_expr, set_id_exp*)

Returns TRUE if object described by *sys_id_expr* (Section 6.3.3.1) belongs in set described by *set_id_exp* (Section 6.3.3.4) else returns FALSE.

int EQ*(sessionID, int_val_expr1, int_val_exp2*)

Mathematic comparison between integer expressions *int_val_expr1* and *int_val_expr2* (Section 6.3.3.2). If (int_val_expr1 == int_val_expr2) returns TRUE else returns FALSE.

int GT*(sessionID, int_val_expr1, int_val_exp2*)

Mathematic comparison between integer expressions *int_val_expr1* and *int_val_expr2* (Section 6.3.3.2). If (int_val_expr1 > int_val_expr2) returns TRUE else returns FALSE.

int GTE*(sessionID, int_val_expr1, int_val_exp2*)

Mathematic comparison between integer expressions *int_val_expr1* and *int_val_expr2* (Section 6.3.3.2). If (int_val_expr1 >= int_val_expr2) returns TRUE else returns FALSE.

int LT*(sessionID, int_val_expr1, int_val_exp2*)

Mathematic comparison between integer expressions *int_val_expr1* and *int_val_expr2* (Section 6.3.3.2). If (int_val_expr1 < int_val_expr2) returns TRUE else returns FALSE.

int LTE*(sessionID, int_val_expr1, int_val_exp2*)

Mathematic comparison between integer expressions *int_val_expr1* and *int_val_expr2* (Section 6.3.3.2). If (int_val_expr1 <= int_val_expr2) returns `TRUE` else returns `FALSE`.

int MATCH*(sessionID, int set_id, int ptrn_set_id*)

Succeeds if the patterns Telos_String's in the set *ptrn_set_id* match (partially or completely) the names of the objects in set *set_id*. The pattern match conventions are described in section 6.3.7.

int BEFORE*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) ends before the beginning of time interval (tm2) (contained in set, set_id_expr2) returns `TRUE` else returns `FALSE`.

int AFTER*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts after the ending of time interval (tm2) (contained in set, set_id_expr2) returns `TRUE` else returns `FALSE`.

int TIME_EQUAL*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts where time interval (tm2) (contained in set, set_id_expr2) starts and finishes where tm2 finishes, returns `TRUE` else returns `FALSE`.

int MEETS*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) ends where time interval (tm2) (contained in set, set_id_expr2) starts, returns `TRUE` else returns `FALSE`.

int MET_BY*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts where time interval (tm2) (contained in set, set_id_expr2) ends, returns `TRUE` else returns `FALSE`.

int OVERLAPS*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts before the ending of time interval (tm2) (contained in set, set_id_expr2) and ends after the beginning of tm2, returns `TRUE` else returns `FALSE`.

int OVERLAPPED_BY*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts before the ending of time interval (tm2) (contained in set, set_id_expr2) and ends after the beginning of tm2, returns `TRUE` else returns `FALSE`. .

int DURING*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts after the beginning of time interval (tm2) (contained in set, set_id_expr2) and ends before the ending of tm2, returns `TRUE` else returns `FALSE`.

int CONTAINS*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts before the beginning of time interval (tm2) (contained in set, set_id_expr2) and ends after the ending of tm2, returns `TRUE` else returns `FALSE`.

int AFTER*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts after the ending of time interval (tm2) (contained in set, set_id_expr2) returns `TRUE` else returns `FALSE`.

int STARTS*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts where time interval (tm2) (contained in set, set_id_expr2) starts and ends before the ending of tm2, returns `TRUE` else returns `FALSE`.

int STARTED_BY*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts where time interval (tm2) (contained in set, set_id_expr2) starts and ends after the ending of tm2, returns `TRUE` else returns `FALSE`.

int FINISHES*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts before the beginning of time interval (tm2) (contained in set, set_id_expr2) and ends where tm2 ends, returns `TRUE` else returns `FALSE`.

int FINISHED_BY*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) starts after the beginning of time interval (tm2) (contained in set, set_id_expr2) and ends where tm2 ends, returns `TRUE` else returns `FALSE`.

int CBEQ*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *There exist t1∈ tm1, t2∈ tm2: t1 = t2* `TRUE` else returns `FALSE`. This means that there exists at least one element into time interval tm1 that is equal to at least one element that belongs to time interval tm2.

int CBLT*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *There exist t1∈ tm1, t2∈ tm2: t1 < t2* returns `TRUE` else returns `FALSE`. This means that there exists at least one element into time interval tm1 that is less than at least one element that belongs to time interval tm2.

int CBLE*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *There exist t1∈ tm1, t2∈ tm2: t1 <= t2* returns `TRUE` else returns `FALSE`. This means that there exists at least one element into time interval tm1 that is less than or equal to at least one element that belongs to time interval tm2.

int CBGT*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *There exist t1∈ tm1, t2∈ tm2: t1 > t2* returns `TRUE` else returns `FALSE`. This means that there exists at least one element into time interval tm1 that is greater than at least one element that belongs to time interval tm2.

int CBGE*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *There exist t1∈ tm1, t2∈ tm2: t1 >= t2* returns `TRUE` else returns `FALSE`. This means that there

exists at least one element into time interval tm1 that is greater than or equal to at least one element that belongs to time interval tm2.

int MBEQ*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *For-every t1∈ tm1, t2∈ tm2: t1 = t2* returns `TRUE` else returns `FALSE`. This means that every element into time interval tm1 must be equal to every element that belongs to time interval tm2.

int MBLT*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *Forevery t1∈ tm1, t2∈ tm2: t1 < t2* returns `TRUE` else returns `FALSE`. This means that every element into time interval tm1 must be less than every element that belongs to time interval tm2.

int MBLE*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *Forevery t1∈ tm1, t2∈ tm2: t1 <= t2* returns `TRUE` else returns `FALSE`. This means that every element into time interval tm1 must be less than or equal to every element that belongs to time interval tm2.

int MBGT*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *Forevery t1∈ tm1, t2∈ tm2: t1 > t2* returns `TRUE` else returns `FALSE`. This means that every element into time interval tm1 must be greater than every element that belongs to time interval tm2.

int MBGE*(sessionID, set_id_expr1, set_id_expr2*)

Mathematic comparison between the bounds of time intervals contained in sets *set_id_expr1* and *set_id_expr2*. If at least one time interval (tm1) (contained in set, set_id_expr1) fulfills the following condition in comparison to the time interval (tm2) (contained in set, set_id_expr2): *Forevery t1∈ tm1, t2∈ tm2: t1 >= t2* returns `TRUE` else returns `FALSE`. This means that every element into time interval tm1 must be greater than or equal to every element that belongs to time interval tm2.

int SET_EQUAL*(sessionID, set_id_exp1, set_id_exp2*)

Set comparison between sets *set_id_exp1* and *set_id_exp2*. If the sets are equal returns `TRUE` else returns `FALSE`.

int SET_DISJOINT*(sessionID, set_id_exp1, set_id_exp2*)

Set comparison between sets *set_id_exp1* and *set_id_exp2*. If the sets are disjoint returns `TRUE` else returns `FALSE`.

6.3.3.2 Expressions returning an integer value

Functions that are used in logical expressions and return an integer value are the following:

int VAL*(sessionID, int_val*)

The integer value *int_val* passed as argument.

int CARD*(sessionID, set_id_expr*)

The cardinality of set *set_id_exp* (Section 6.3.3.4).

6.3.3.3 Expressions returning a system identifier

Functions that are used in logical expressions and describe an object are the following:

int SYS_ID*(int sessionID, int sysid*)

The object with system identifier *sysid*.

int NODE*(int sessionID, l_name nodename*)

The node object with unique logical name *nodename*.

int LINK*(int sessionID, l_name fromcls, l_name linkname*)

The link object with logical name *linkname* that points from node object with unique logical name *fromcls*.

6.3.3.4 Expressions returning a set identifier

Functions that are used in logical expressions and return a set identifier are the following:

int SET_ID*(sessionID, setid*)

The object with set identifier *setid*. If *setid* is 0 means the object the condition is evaluated for.

int SET_UNION*(sessionID, set_id_expr1, set_id_expr2*)

Set union between set described by *set_id_expr1* and set described by *set_id_expr2*. If the two sets are tuples the union is performed to the tuples (provided they have the same number of columns).

int SET_INTERSECT*(sessionID, set_id_expr1, set_id_expr2*)

Set intersection between set described by *set_id_expr1* and set described by *set_id_expr2*.

int SET_DIFFERENCE*(sessionID, set_id_expr1, set_id_expr2*)

Set difference between set described by *set_id_expr1* and set described by *set_id_expr2*. If the two sets are tuples the difference operation is performed to the tuples.

int SET_COPY*(sessionID, set_id_expr1, set_id_expr2*)

Set copy of set described by *set_id_expr2* to set described by *set_id_expr1*.

int GET_CLASSES*(sessionID, set_id_expr*)

Apply **get_classes**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_ALL_CLASSES*(sessionID, sessionID, set_id_expr*)

Apply **get_all_classes**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_SYSCLASS*(sessionID, set_id_expr*)

Apply **get_sysclass**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_ALL_SYSCLASSES*(sessionID, set_id_expr*)

Apply **get_all_sysclasses**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_INSTANCES*(sessionID, set_id_expr*)

Apply **get_instances**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_ALL_INSTANCES*(sessionID, set_id_expr*)

Apply **get_all_instances**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_CLASS_ATTR*(sessionID, set_id_expr*)

Apply **get_class_attr**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_ALL_CLASS_ATTR*(sessionID, set_id_expr*)

Apply **get_all_class_attr**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_SUPERCLASSES*(sessionID, set_id_expr*)

Apply **get_superclasses**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_ALL_SUPERCLASSES*(sessionID, set_id_expr*)

Apply **get_all_superclasses**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_ALL_SYSSUPERCLASSES*(sessionID, set_id_expr*)

Apply **get_all_Syssuperclasses**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_SUBCLASSES*(sessionID, set_id_expr*)

Apply **get_subclasses**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_ALL_SUBCLASSES*(sessionID, set_id_expr*)

Apply **get_all_subclasses**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_LINK_FROM*(sessionID, set_id_expr*)

Apply **get_link_from**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_CLASS_ATTR_FROM*(sessionID, set_id_expr*)

Apply **get_class_attr_from**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_LINK_TO*(sessionID, set_id_expr*)

Apply **get_link_to**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_FROM_VALUE*(sessionID, set_id_expr*)

Apply **get_from_value**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_TO_VALUE*(sessionID, set_id_expr*)

Apply **get_to_value**() on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_LINK_FROM_BY_CATEGORY*(sessionID, set_id_expr, sys_id_expr*)

Apply **get_link_from_by_category**() with category described by *sys_id_expr* (Section 6.3.3.3), on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

int GET_LINK_TO_BY_CATEGORY*(sessionID, set_id_expr, sys_id_expr*)

Apply **get_link_to_by_category**() with category described by *sys_id_expr* (Section 6.3.3.3), on set *set_id_expr* and return the answer set. If *set_id_expr* is SET_ID(0) then apply query on object the condition is evaluated for.

6.3.3.5 An Example

The following code segment finds all instances of current node that have exactly one link pointing to them:

```
tmp_set = get_instances(sessionID, 0);
set_filter_cond(sessionID,
      EQ(sessionID,
            CARD(sessionID,
                  GET_LINK_TO(sessionID, SET_ID(sessionID,0))),
            VAL(sessionID, 1)
      )
);
ans_set = get_filtered( sessionID, tmp_set);
```

6.3.4 General recursive queries

There are queries whose answer requires traversing of links in the SIS base. API has a query function that starting from an object, traverses links according to some restrictions. The restrictions have to do with the traversed links and/or the visited nodes. Actually, the programmer has to define logical expressions, see next section on how to do so, that are evaluated for each traversed link and visited node: if the expression is true for some link pointing from the currently visiting node and if the expression is true for the to-value of the link then this link is traversed forward. In the same way if the expression is true for some link pointing to the currently visiting node and if the expression is true for the from-value of the link then this link is traversed backwards.

The depth of the traverse can be controlled with the **set_depth**() function. If not explicitly set, the depth is not controlled.

int get_traverse_by_all_links*(int sessionID, int set_id, int isa*)

If *set_id* is 0, starts from the current node and traverses all the links according to the specified conditions. If conditions have not been explicitly set then all links are traversed. The *isa* argument can be any of `UPWARDS`, `DOWNWARDS`, `UP_DOWN` or `NOISA`. If `UPWARDS` is used, then for each node visited on traversing, all superclasses of this node are visited too. If `DOWNWARDS` is used,

then for each node visited on traversing, all subclasses of this node are visited too. In case of `UP_DOWN` both all superclasses and all subclasses for each node are traversed. Nothing of these two happens when `NOISA` is used. The answer set contains the system identifiers of all the traversed links.

If *set_id* is a positive integer, apply **get_traverse_by_all_links**() on each object in temporary set *set_id*.

The function returns `APIFail`(-1) on error or the set identifier of the temporary set where the answer is stored.

6.3.5 Conditions on recursive queries

As mentioned in the previous section the programmer can define four conditions for the general recursive query: A condition evaluated for links pointing from the currently visiting node, a condition evaluated for the to-value of those links, a condition evaluated for links pointing to the currently visiting node and a condition evaluated for the from-value of those links. The API functions for these definitions are respectively the following:

int set_fl_cond*(sessionID, log_exp*)

int set_tv_cond*(sessionID, log_exp*)

int set_tl_cond*(sessionID, log_exp*)

int set_fv_cond*(sessionID, log_exp*)

The argument to these functions is a logical expression constructed as described in the section *Logical expressions - Object set filtering* (section 6.3.3).

6.3.5.1 An Example

The following function calls describe the conditions for the query:

"Traverse all forward links that are of category <PhysicalObj, Parts> but visit only the nodes that have exactly one instance."

```
set_tl_cond( sessionID, FAIL(sessionID));

set_fl_cond( sessionID,
      BELONGS( sessionID,
            LINK( sessionID,"PhysicalObj","Parts"),
            GET_CLASSES( sessionID, SET_ID(sessionID,0))
      )
);
set_tv_cond( sessionID,
      EQ( sessionID,
            VAL( sessionID, 1),
            CARD( sessionID,
                  GET_INSTANCES( sessionID, SET_ID(sessionID,0))
            )
      )
);
```

The following function calls describe the conditions for the query:

"Traverse all forward links that are of category <PhysicalObj, Parts> but visit only the nodes that are pointed to by exactly one link or exist in the answer set 3."

```
set_tl_cond( sessionID,FAIL(sessionID));
set_fl_cond( sessionID,
      BELONGS( sessionID,
            LINK( sessionID,"PhysicalObj","Parts"),
            GET_CLASSES( sessionID, SET_ID( sessionID, 0))
      )
);
set_tv_cond( sessionID,
      OR( sessionID,
            EQ( sessionID,
                  VAL(sessionID,1),
                  CARD(sessionID,
                        GET_LINK_TO(sessionID, SET_ID(sessionID,0))
                  )
            ),
            BELONGS( sessionID,
                  SYS_ID(sessionID,0),
                  SET_ID(sessionID,3)
            )
      )
);
```

6.3.6 Special recursive queries

There are some recursive queries, special cases of the general recursive query, that are frequently used and are explicitly implemented for efficiency. The only condition taking into account in these queries is the category or the meta-category of the traversed links. These categories must have been defined with **set_categories**().

When traversing is done with restricted depth, the nodes which have links that could be traversed but this didn't happed because of the depth, are accumulated into an **edge_set** at the server site and can be given to the client application with the

> These functions return `APIFail`(-1) on error or the set identifier of the temporary set where the answer is stored.

return_edge_node() described below.

int get_traverse_by_category*(int sessionID, int set_id, int isa*)

Before calling this query function some categories must have been defined with the **set_categories**() function. If *set_id* is 0, starts from the current node and traverses all the links whose category is one of those previously defined with the *set_categories*() function. With the **set_depth**() function the depth of traversing can be controlled. The *isa* argument can be any of `UPWARDS`, `DOWNWARDS`, `UP_DOWN` or `NOISA`. If `UPWARDS` is used, then for each node visited on traversing, all superclasses of this node are visited too. If `DOWNWARDS` is used, then for each node visited on traversing, all subclasses of this node are visited too. In case of `UP_DOWN` both all superclasses and all subclasses for each node are traversed. Nothing of these two happens when `NOISA` is used.

The answer set contains the system identifiers of all the traversed links.

If *set_id* is a positive integer, apply **get_traverse_by_category**() on each object in temporary set *set_id*.

int get_traverse_by_meta_category*(int sessionID, int set_id, int isa*)

Before calling this query function some categories must have been defined with the *set_categories* function. If *set_id* is 0, starts from the current node and traverses all the links whose meta-category is one of those previously defined with the *set_categories*() function. With the **set_depth**() function the depth of traversing can be controlled. The *isa* argument can be any of `UPWARDS`, `DOWNWARDS`, `UP_DOWN` or `NOISA`. If `UPWARDS` is used, then for each node visited on traversing, all superclasses of this node are visited too. If `DOWNWARDS` is used, then for each node visited on traversing, all subclasses of this node are visited too. In case of `UP_DOWN` both all superclasses and all subclasses for each node are traversed. Nothing of these two happens when `NOISA` is used.

The answer set contains the system identifiers of all the traversed links.

If *set_id* is a positive integer, apply **get_traverse_by_meta_category**() on each object in temporary set *set_id*.

6.3.7 Pattern Matching queries

There is a query to select from a set of objects those whose the name matches (completely or partially) the logical OR of the Telos_String's patterns. The supported patterns are:

1. **string*
   Matches the names that end by string.
2. *string**
   Matches the names that start by string.
3. **string** or *string*
   Matches the names that contain the substring string.

These functions return `APIFail`(-1) on error or the set identifier of the temporary set where the answer is stored.

The API functions with the above functionality are the following.

int get_matched*(int sessionID, int obj_set_id, int ptrn_set_id*)

From all the objects in the set *obj_set_id* select those objects that match *any* of the string patterns in the set *ptrn_set_id*. The match is case sensitive. The set *ptrn_set_id* can be constructed using the **set_put_prm**() function. The answer set contains all the selected objects. If *obj_set_id* is 0, perform the match on the *current node* and if there is a match put the *current node* in the answer set or return an empty set.

int get_matched_case_insensitive*(int sessionID, int obj_set_id, int ptrn_set_id, int encoding*)

From all the objects in the set *obj_set_id* select those objects that match *any* of the string patterns in the set *ptrn_set_id*. The match is case insensitive. The set *ptrn_set_id* can be constructed using the **set_put_prm**() function. The answer set contains all the selected objects. If *obj_set_id* is 0, perform the match on the *current node* and if there is a match put the *current node* in the answer set

or return an empty set. The *encoding* parameter may be 1 for Latin1, 7 for Latin7. No other encodings are supported yet.

int get_matched_string(*int obj_set_id, cm_value *cmv, int match_type)*

From all the objects in the set *obj_set_id* select those objects that match the string in *cmv* according to the *match type* criterion (`STRING_MATCHED`, `STRING_LESS_EQUAL`, `STRING_LESS_THAN`, `STRING_EQUAL`, `STRING_NOT_EQUAL`). In case of `STRING_MATCHED` it performs the selection using the string as a pattern string as described above. In all other cases the match is performed on the exact string. The answer set contains all the selected objects. If *obj_set_id* is 0, perform the match on the *current node* and if there is a match put the *current node* in the answer set or return an empty set.

6.3.7.1 An Example

The following code segment finds all instances of current node which start by "Ana":

```
cm_value prm_val;
assign_string(&prm_val, "Ana*");
tmp_set = get_instances(sessionID, 0);
ptrn_set = set_get_new(sessionID);
set_put_prm(sessionID, ptrn_set, &prm_val);
ans_set = get_matched(sessionID, tmp_set, ptrn_set);
```

6.4 Set manipulation functions

As mentioned earlier the answers of the query functions are located into temporary sets. There is group of functions for manipulating these sets (operations between sets, freeing sets etc.).

There is a finite number of temporary sets that can be used and so no more useful sets should be released in order to be used again. We handle these sets with their identifier that is a positive integer. The use of unique identifiers for the temporary sets is similar to the use of file descriptors in UNIX operating system.

Most of the functions return `APISucc`(0) on success and `APIFail`(-1) on error, except the **set_get_new**() function which returns a set identifier and the **set_get_card**() function which return the cardinality of a set.

int reset_set*(int sessionID, int set_id*)

Each set has a pointer that moves along the objects of the set and is used when we get these objects one-by-one. This function just sets this pointer to the first item of the set and its call is necessary before first call of a function that reads the next object in a set.

int reset_edge_set*(int sessionID*)

Reset the pointer of the **edge_set** at the first item of the set. (See Section 6.3.6 for definition of the edge_set).

int free_set*(int sessionID, int set_id*)

This function free the temporary set *set_id* so that it can be used later by some other query. Freeing a set is like closing a file descriptor in UNIX.

int free_all_sets*(int sessionID*)

This function frees all the temporary sets.

int set_get_new*(int sessionID*)

This function returns the set identifier of a new empty set.

int set_union*(int sessionID, int set_id1, int set_id2*)

After this operation the first set is the union of the two sets given as arguments. If the two sets are tuples the union is performed to the tuples (provided they have the same number of columns).

int set_intersect*(int sessionID, int set_id1, int set_id2*)

After this operation the first set is the intersection of the two sets given as arguments.

int set_difference*(int sessionID, int set_id1, int set_id2*)

After this operation the first set is the difference of the two sets given as arguments. If the two sets are tuples the difference operation is performed to the tuples.

int set_equal*(int sessionID, int set1, int set2*)

This function returns `TRUE` if set1 = set2.

int set_disjoint*(int sessionID, int set1, int set2*)

This function returns `TRUE` if the intersection of set1, set2 is empty.

int set_copy*(int sessionID, int set1, int set2*)

This function copies set2 to set1.

int set_put*(int sessionID, int set_id*)

This function puts the current node into set *set_id*.

int set_put_prm*(int sessionID, int set_id, cm_value *cmval*)

This function puts the primitive value (integer, string, time etc) *cmval* into set *set_id.*

int set_get_card*(int sessionID, int set_id*)

This function returns the number of object that exist in set *set_id*.

int set_member_of*(int sessionID, int set_id*)

This function checks whether the current node exists in set *set_id*.

int set_position*(int sessionID, int set_id, int pos*)

This function sets the internal server set cursor of set set_id to element pos.

int set_del*(int sessionID, int set_id*)

This function deletes the current node from set *set_id*.

int set_clear*(int sessionID, int set_id*)

This function clears the set *set_id*.

int set_clear_lower*(int sessionID, int set_id*)

This function clears the set *set_id* from position 0 up to current position. It updates the set's cardinality and sets the current position to the beginning of the set. It does not delete the element located in current position.

6.5 Read contents of answer sets

The answers of any of the described queries are located in temporary sets. There is a group of functions to read the information from an answer set. Before first calling any of these functions to read the objects of an answer set A the pointer of this set must be reset at the first item of the set with the **reset_set**() function.

Most of the following functions operate like this: when called, they give the answer for the next object in the set they apply on and return `APISucc`(0). If any error occurred or end of set reached they return `APIFail`(-1).

int return_nodes*(int sessionID, int set_id, l_name cls*)

Return the logical name *cls* of the next object in set *set_id*.

int return_edge_nodes*(int sessionID, int *sysid, l_name node*)

Return the system identifier *sysid* and the logical name *node* of the next object in **edge_set**. (See Section 6.3.6 for definition of the edge_set).

int return_full_nodes*(int sessionID, int set_id, int *sysid, l_name node, l_name Sclass*)

Return the system identifier *sysid*, the logical name *node* and the system class *Sclass* of the next object in set *set_id*.

int return_prm*(int sessionID, int set_id, cm_value *cmv*)

An answer set may contain primitive values (integers, strings etc) and the previous functions would fail to read these items. This function returns the next object in set *set_id* whatever it is. (See section 6.8.1 for definition of cm_value structure).

int return_categ_ids*(int sessionID, int set_id, int *sysid, l_name cls, l_name categ*)

Return the logical name *categ* and the system identifier *sysid*, the logical name *cls* of the from-class of the next object in set *set_id*. Set *set_id* is supposed to contain link objects.

int return_categories*(int sessionID, int set_id, l_name cls, l_name categ*)

Return the logical name *categ* and the logical name *cls* of the from-class of the next object in set *set_id*. Set *set_id* is supposed to contain link objects.

int return_link*(int sessionID, int set_id, l_name cls, l_name label, cm_value *cmv*)

For the next object in set *set_id*, supposes it is a link object, and returns the logical *label* name of the object, the logical name *cls* of the class which it is pointing from and the object it is pointing to *cmv*. Since this object may not be a class but a primitive value (string, integer etc.) it is returned in a structure cm_value which is described in section 6.8 and in *Appendix A – C-API function declaration*.

int return_link_id*(int sessionID, int set_id, l_name cls, int *fcid, int *sysid, cm_value *cmv, int *traversed*)

For the next object in set *set_id*, supposes it is a link object, and returns the logical name *cls* of the class which it is pointing from, the object's system identifier *sysid*, the system identifier *fcid* of the object it is pointing from and the object (or primitive value) *cmv* it is pointing to. The flag *traversed* indicates if the specific link belongs to a category that was previously set with the **set_categories**() function with direction `BACKWARD`.

int return_full_link*(int sessionID, int set_id, l_name cls, l_name label, l_name categ, l_name fromcls, cm_value *cmv, int *unique_category, int *traversed*)

Return the objects of set *set_id*. It is supposed that the objects are link nodes are so it returns the logical name of each object *label*, the logical name of the class where it pointing from *cls*, the object it is pointing to *cmv* which is a structure cm_value described later and also the category of the returned link (from_cls, categ).

Flag *unique_category* indicates if given category is unique (link object may have more than one class) and flag *traversed* indicates if the specific link belongs to a category that was previously set with the **set_categories**() function with direction `BACKWARD`.

int return_full_link_id*(int sessionID, int set_id, l_name cls, int *clsid, l_name label, int *linkid, l_name categ, l_name fromcls, int *categid, cm_value *cmv, int *unique_category*)

Return the objects of set *set_id*. It is supposed that the objects are link nodes are so it returns the logical name of each object *label*, its system identifier's *linkid*, the logical name *cls* of the class where it is pointing from and its system identifier *clsid*, the object it is pointing to *cmv* which is a structure cm_value described later and also the category of the returned link (*from_cls, categ*) and its system identifier *categid*.

Flag *unique_category* indicates if given category is unique (link object may have more than one class).

int return_isA*(int sessionID, int set_id, l_name ob1, l_name ob2*)

Return the logical names *ob1* and *ob2* of a pair of objects A and B, existing in set *set_id* and A isA B.

int return_isA_id*(int sessionID, int set_id, l_name ob1, int *id1, l_name ob2, int *id2*)

Return the logical names and their system identifiers *ob1*, *id1* and ob2, *id2* of a pair of objects A and B, existing in set *set_id* and A isA B.

int return_inst*(int sessionID, int set_id, l_name ob1, l_name ob2*)

Return the logical names *ob1* and *ob2* of a pair of objects A and B, existing in set *set_id* and A is instance of B.

int return_inst_id*(int sessionID, int set_id, l_name ob1, int *id1, l_name ob2, int *id2*);)

Return the logical names and system identifiers ob1*, id1* and *ob2, id2* of a pair of objects A and B, existing in set *set_id* and A is instance of B.

int return_field*(int sessionID, int set_id, cm_value *cmv*)

What we get by sequential calls of this function for each object in set *set_id* is the following: the logical name of the object and for each category previously defined with the **set_categories** function, get the to value of the links whose category was set `FORWARD` or the from value of the links whose category was set `BACKWARD`. The function ignores categories that where set `BOTH_DIR`.

Each item of the information that must be returned by this function is returned one by one by multiple calls of the function. The return value specifies the kind of information that is being returned (namely `END_OF_TUPLE`, `END_OF_SET`, `END_OF_FIELD`, `EMPTY_FIELD`, `MIDDLE_OF_FIELD`, `EMPTY_FIELD_END_OF_TUPLE`).

int return_hidlink*(int sessionID, int set_id, l_name cls, int *cls_id, l_name label, int *sysid, cm_value *cmv1, cm_value *cmv2*)

Returns the from-class name, logical name label, sysid and to-value of next object in set set_id supposing it to be a link object. If the from-object is linked with an inverse link of category previously set with **set_categories**() function with an object A then object A is returned in the position of from-class since this object is supposed to be some kind of member of object A. Recursively the same happens if A has an inverse links of this specific category.

With this mechanism we can get an abstraction of some information hiding the information of links that are instances of a specific category.

int return_xml_description*(int sessionID, int set_id, char **xml_string*)

Return the XML description *xml_string* of the next object in set *set_id*. The function allocates space for buffer *xml_string*. The allocated space should be freed after its use with **free_sis_allocated_space()**. A detailed description of the memery management needed is presented in section 6.8.2. The xml document type definition is presented in "*Appendix F – Describing in XML*".

6.5.1 Parametric projection

API provides a mechanism to project the objects of a set as well as some information for each of them. This information can be retrieved by answering a predefined query for each object of the set. This kind of projection is achieved with the **return_projection**() function and the predefined query is declared with the **set_proj_cond**() function:

int return_projection*(int sessionID, int set_id, cm_value *cmv*)

What we get by sequential calls of this function for each object in set *set_id* is the following : the logical name of the object and the logical name of the objects in the answer set that is retrieved by applying a predefined query to the object. This function returns `END_OF_SET` when the end of set is reached, `APIFail`(-1) in case of error and a positive integer on success which specifies the kind of information that is being returned (namely `END_OF_TUPLE`, `END_OF_SET`, `END_OF_FIELD`, `EMPTY_FIELD`, `MIDDLE_OF_FIELD`, `EMPTY_FIELD_END_OF_TUPLE`).

int set_proj_cond1*(int sessionID, int set_expr*)

int set_proj_cond2*(int sessionID, int set_expr*)

int set_proj_cond3*(int sessionID, int set_expr*)

int set_proj_cond4*(int sessionID, int set_expr*)

Define a query that is constructed according to the functions presented in section 6.3.3.4.

int set_num_of_proj*(int sessionID, int n*)

Used to set which of the set_proj_conds will be used. For example: **set_num_of_proj**(3) means that **set_proj_cond1**(), **set_proj_cond2**() and **set_proj_cond3**() should be used.

6.6 Tuple handling functions

API provides a set of functions for handling tuples. Tuples are sets of columns, created by **set_current_node**() and **get_new_col**(). The columns are indexed starting from 0. The query functions operate on the *working column* of a tuple the same way the operate on a set. The default working column of a tuple is the its column, and it can be changed by **set_set_input**().

The functions presented below return `APIFail`(-1) on error and `APISucc`(0) on success except as stated otherwise.

int get_new_col*(int sessionID, int set_id*)

Creates and appends a new column to the tuple *set_id*.

int tuple_set_input_col*(int sessionID, int set_id,int x*)

Sets the column x as the working column of the tuple *set_id*.

int tuple_set_join_pos*(int sessionID, int set_id, int x, int y*)

Marks the column *x* as the column of the tuple *set_id* to be joined to the column *y* of the candidate tuple to joined with by **tuple_join**() function.

int tuple_reset_join_pos*(int sessionID, int set_id*)

Resets the marked column positions of the tuple *set_id* that where set by **tuple_set_join_pos**() function.

int tuple_join*(int sessionID, int set_id1, int set_id2*)

Forms the join of the tuple *set_id1* that also holds the marked tuple's column position information (set by **tuple_set_join_pos**()) with the tuple *set_id2*.

int tuple_union*(int sessionID, int set_id1, int set_id2*)

Forms the union of the tuple *set_id1* with the tuple *set_id2*.

int tuple_difference*(int sessionID, int set_id1, int set_id2*)

Forms the difference of the tuple *set_id1* with the tuple *set_id2*.

int tuple_no_projection_column*(int sessionID, int set_id, int x*)

Hides the column *x* from tuple *set_id* connecting accordingly the "surrounding" tuples.

int return_tuple*(int sessionID, int set_id, cm_value *cmv*)

Each time this function is called it returns the next item in the given tuple in *cmv*. If the current tuple position is empty *cmv w*ill be undefined and will have the tag `TYPE_EMPTY`. It returns `END_OF_TUPLE` if the end of the tuple was reached or `END_OF_FIELD` if the end of the current field was reached.

int return_relation*(int sessionID, int from_set ,int categ_set, int to_set*)

This function is not implemented yet.

int unary_on_tuple*(int sessionID, int set_id*)

This function is not implemented yet.

6.7 Update Functions

The API provides a set of functions used to make updates in the Database. The operations that can be performed are:

- Addition operations
- Deletion operations
- Modification operations

A difference between the update functions and the rest of the API is that they do not require a current node to work on. Also the concept of the `IDENTIFIER` is used. An `IDENTIFIER` is a structure that can hold either a logical name (char *) or a system identifier (int). The type of the `IDENTIFIER` is given by a tag. Most functions work

with both types. If a function requires only a logical name or only a system identifier it is stated in the function description.

In order to execute updates on a database a transaction session must be initiated. A transaction session can either be initiated directly or from within a query session. When a transaction session is in progress a write lock is applied to the database and only the writing client can access it. A transaction session begins with the **begin_transaction**() function and ends with either the **end_transaction**() function (commits changes) or with the **abort_transaction**() function (does not commit changes). When the transaction session is initiated from within a query session, on termination of the transaction session the query session continues, and thus a read lock exists on the database. In order to release the read lock end the query session by calling **end_query**().

The functions presented below return `APIFail(-1)` on error and `APISucc(0)` on success.

6.7.1 Addition Operations

These operations are used to add objects to the database:

int Add_Node*(int sessionID, IDENTIFIER *node_name, int level*)

Adds a node with the logical name *node_name* at instantiation level *level* (`SIS_API_TOKEN_CLASS,SIS_API_S_CLASS,SIS_API_M1_CLASS,SIS_API_M2_CLASS,SIS_API_M3_CLASS,SIS_API_M4_CLASS`).

int Add_Named_Attribute*(int sessionID, IDENTIFIER *attribute, IDENTIFIER *from, cm_value *to, int iLevel, int catSet*)

Adds a named attribute with the logical name *attribute* pointing from *from* (IDENTIFIER contains the SYSID) to cm_value *to* at instantiation level *iLevel* (`SIS_API_TOKEN_CLASS,SIS_API_S_CLASS,SIS_API_M1_CLASS,SIS_API_M2_CLASS,SIS_API_M3_CLASS,SIS_API_M4_CLASS`) with categories the categories given in the set identifier *catSet*. If *catSet* is -1 then no categories are used. The function fails if any of *from* or *to* does not exist, or any of *iLevel* or *catSet* is invalid

int Add_Unnamed_Attribute*(int sessionID, IDENTIFIER *from, cm_value *to, int catSet*)

Adds an unnamed attribute pointing from *from* to cm_value *to* with categories the categories given in the set identified by *catSe*t. If *catSet* is -1 then no categories are used. The function fails if any of *from* or *to* does not exist, or *catSet* is invalid.

int Add_Instance*(int sessionID, IDENTIFIER *from, IDENTIFIER *to*)

Adds an instance pointing from *from* to *to*. It fails if any of *from* or *to* does not exist.

Add_Instance_Set*(int sessionID, int from_set, to IDENTIFIER *to)*

Adds all instances in set *from_set* pointing to *to*. It fails if an instance with the characteristics given already exists.

int Add_IsA*(int sessionID, IDENTIFIER *from, IDENTIFIER *to*)

Adds an isA pointing from *from* to *to*. It fails if any of *from* or *to* does not exist.

6.7.2 Delete Operations

These operations are used to delete objects from the database:

int Delete_Node*(int sessionID, IDENTIFIER *node_name*)

Deletes a node with the logical name (or sysid) *node_name*. It fails if the node does not exist or has dependencies with other objects.

int Delete_Named_Attribute*(int sessionID, IDENTIFIER *link_name, IDENTIFIER *from*)

Deletes a named attribute with the logical name (or sysid) *link_name* from *from*. The *from* parameter must always specify a SYSID. It fails if the link *link_name* does not exist or has dependencies with other objects.

int Delete_Unnamed_Attribute*(int sessionID, IDENTIFIER *attribute*)

Deletes an unnamed attribute with the given SYSID (given in *attribute*). It fails if a link with the given SYSID *attribute* does not exist or has dependencies with other objects.

int Delete_Instance*(int sessionID, IDENTIFIER *from, IDENTIFIER *to*)

Deletes an instance pointing from *from* to *to.* It fails if an instance with the given *from*, to does not exist or the given link has dependencies with other objects exist.

Delete_Instance_Set*(int sessionID, int from_set, IDENTIFIER *to)*

Deletes all the instances of set *from_set* pointing to *to*. It fails it an instance with the given *from* does not exist or has dependencies with other objects.

int Delete_IsA*(int sessionID, IDENTIFIER *from, IDENTIFIER *to*)

Deletes an isA pointing from *from* to *to.* It fails if an isA with the given *from*, to does not exist or the given isA has dependencies with other objects.

6.7.3 Modification Operations

These operations are used to modify the characteristics of objects in the database:

int Rename_Node*(int sessionID, IDENTIFIER *node, IDENTIFIER *NewNodeName*)

Change the name of a node. The current node is specified by IDENTIFIER *node* and the new logical name for the node by *NewNodeName*. It fails if a node with the given IDENTIFIER *node* does not exist or a node with the given IDENTIFIER *NewNodeName* already exists.

int Rename_Named_Attribute*(int sessionID, IDENTIFIER *name, IDENTIFIER *from, IDENTIFIER *NewName*);

Change the name of a named attribute. The current named attribute is specified by IDENTIFIER *name* (logical name) and the *from* value, or the SYSID in *name*. The new logical name for the named attribute is specified by *NewName*. It fails if a named attribute with the given IDENTIFIER *name* does not exist from IDENTIFIER *from* or a named nttribute with the given IDENTIFIER *NewName* already exists from IDENTIFIER *from*.

int Change_Named_Attribute_To*(int sessionID, IDENTIFIER *Attribute, IDENTIFIER *From, cm_value *To, cm_value *NewTo*);

Change the To value of a named attribute. The named attribute is specified by IDENTIFIER *Attribute* and the from value IDENTIFIER *From* (always SYSID). The current to value is required for checking and it should be specified in cm_value *To*. The new To value for the named attribute is specified by cm_value *NewTo*. It fails if (a) a named attribute with the given IDENTIFIER *name* does not exist, (b) the *NewTo* value does not exist and cannot be created, or (c) a named attribute with the given name A*ttribute* and *From* already exists.

int Change_Unnamed_Attribute_To*(int sessionID, IDENTIFIER *Attribute, cm_value *To, cm_value *NewTo*);

This function is not implemented yet. It returns `APIFail`.

int Change_Instance_To*(int sessionID, IDENTIFIER *from, IDENTIFIER *to, IDENTIFIER *NewTo*);

Change the To value of an instance. The instance is specified by the from value in IDENTIFIER *from* and the To value IDENTIFIER *to*. The new To value for the instance is specified by IDENTIFIER *NewTo*. It fails if *from* is not an instance of *to* or the *New To* value does not exist.

int Change_IsA_To*(int sessionID, IDENTIFIER *from, IDENTIFIER *to, IDENTIFIER *NewTo*)

Change the To value of an isA. The instance is specified by the from value in IDENTIFIER *from* and the To value IDENTIFIER *to*. The new To value for the instance is specified by IDENTIFIER *NewTo*. It fails if there is on isA between the given *from* and *to*, or the *New To* value does not exist.

6.8 Miscalaneous utility structures and functions

The functions of the SIS C application programmatic interface (C-API) described in the previous sections use various structures as arguments. In the following we present the type definitions of these data types:

6.8.1 Utility structures

6.8.1.1 struct IDENTIFIER

An `IDENTIFIER` is a structure that can hold either a logical name (char *) or a system identifier (int). The type of the `IDENTIFIER` is given by a tag. Most functions described in section 6.7 work with both types.

6.8.1.2 struct cm_value

A `cm_value` is a structure to read the primitive values (string, integer, etc.) form an answer set using the functions described in section 6.5 (see **return_prm()**, **return_link()**, etc.).

It is defined as follows:

```
typedef struct cm_value  {
         int type;  /* TYPE_INT, TYPE_STRING, TYPE_FLOAT,
                       TYPE_NODE, TYPE_EMPTY, TYPE_TIME */
         int sysid;
         union  {
           TIME  t;
           char *s;
           int   n;
           float r;
        }value;
     }cm_value;
typedef struct cm_value cm_value;
```

Here follows a list of functions to set the `cm_value` fields.

int assign_node(*cm_value *cmv, char *s, int sysid*)
int assign_string(*cm_value *cmv, char *s*)
void assign_time(*cm_value *cmv, TIME t*)
void assign_int(*cm_value *cmv, int n*)
void assign_float(*cm_value *cmv, float r*)
void assign_empty(*cm_value *cmv*)

> The functions above set the appropriate type and value fields according to their argument passed. Notice that **assign_node()** and **assign_string()** allocate space that should be freed when is no needed any more (see function **free_sis_allocated_space()**).

The system also allocates dynamically, space to return string information to a `cm_value` structure, this space should be freed when not needed.

6.8.1.3 struct category_set

Special recursive queries described in section 6.3.6 use as query condition the category or the meta-category of the traversed links. This condition is set by **set_category()** function that takes as argument a `category_set` array.

The definition of a `category_set` is as follows:

```
struct category_set  {
         name_buffer fcl;
         name_buffer cat;
         int  direction; /* FORWARD, BACKWARD, BOTH_DIR */
       };
typedef struct category_set  categories_set[NUMBER_OF_CATEGORIES];
```

6.8.2 Memory management

There are three cases that the system allocates dynamically space. This space is needed to hold string values that their size cannot be estimated in advance: (a) space to return string information to a `cm_value` structure (function **return_full_link()** allocates space), (b) space to hold the logical name of a node or a string value field of `struct cm_value` (functions **assign_node()** and **assign_string()** allocate space) and (c) space for the XML description of an object (function **return_xml_description()** allocates space).

In all the above cases the allocated space should be freed after its use with **free_sis_allocated_space()**. This function is a simple call to system's function ***free()***. It is created to provide SIS C API dynamic library (DLL) with a consistent way to de-allocate space that was allocated by the DLL, in order to solve the following memory management problem:

An EXE and a DLL each have their own memory heaps. When memory is allocated calling ***malloc()*** inside the DLL, the memory is owned by the DLL and it has to be managed by the DLL. If the memory is allocated in the EXE, it is owned by the EXE and has to be managed accoding. If the EXE tries to free a block that was allocated by the DLL, or vice versa, the result is a corrupted heap.

Thus if you are using the DLL of the SIS C API do not use ***free()***. Use **free_sis_allocated_space()** instead.

Appendix A – C-API function declaration

Here follows a list of the type definitions of data types seen in the argument list of some functions of the SIS C application programmatic interface (C-API):

```
typedef char  l _name[LOGINAM_SIZE];
typedef char name_buffer[INPUT_LOGINAM_SIZE];

struct category_set  {
         name_buffer fcl;
         name_buffer cat;
         int  direction; /* FORWARD, BACKWARD, BOTH_DIR */
       };
typedef struct category_set  categories_set[NUMBER_OF_CATEGORIES];

typedef struct cm_value  {
         int type;  /* TYPE_INT, TYPE_STRING, TYPE_FLOAT,
                       TYPE_NODE, TYPE_EMPTY, TYPE_TIME */
         int sysid;
         union  {
           TIME  t;
           char *s;
           int   n;
           float r;
         }value;
     }cm_value;
typedef struct cm_value cm_value;
```

Here follows a list of functions to set the cm_value fields.

```
char *cm_to_str(cm_value *val);
void  print_val(cm_value *cmv);

int   assign_node(cm_value *cmv, char *s, int sysid);
int   assign_string(cm_value *cmv, char *s);
void  assign_time(cm_value *cmv, TIME t);
void  assign_int(cm_value *cmv, int   n);
void  assign_float(cm_value *cmv, float r);
void  assign_empty(cm_value *cmv);
```

ATTENTION !!! Since the system dynamically allocates space to return string information to a `cm_value` structure, this space should be freed when not needed (see section 6.8.2) on memory management.

In the following pagesof the appendix we present the complete listing of the definition of the functions the API consists of.

```
/*****************************************************************
 *
 *                      Semantic Index System
 *
 *   COPYRIGHT (c) 1992 by Institute of Computer Science,
 *                         Foundation of Research and Technology - Hellas
 *                         POBox 1385, Heraklio Crete, GR-711 10 GREECE
 *
 *
 *                       ALL RIGHTS RESERVED
 *
 *   This software is furnished under license and may be used only in
 *   accordance with the terms of that license and with the inclusion
 *   of the above copright notice.  This software may not be provided
 *   or otherwise made available to, or used by, any other person. No
 *   title to or ownership of the software is hereby transferred.
 *
 *
 *   Module  : c_wrapper.h
 *   Version : 202.12
 *
 *   Purpose : Wraps the C++ API and provides a C API.
 *
 *   Author  : P. Karamaounas
 *   Creation Date :                   Date of last update : 05/26/00
 *
 *   Remarks :
 *
 *
 *****************************************************************/

void  get_db_dir(char **db_dir);
int * init_start_telos(char *db_dir, int wr_permission = 0);

/*
       Open the SIS API
*/
int create_SIS_SA_Session(int *sessionId, int *start_t, char *Server, int port, char
*DBUserName, char *DBUserPassword);           // Stand Allone
int create_SIS_CS_Session(int *sessionId, char *Server, int port, char *DBUserName,
char *DBUserPassword); // Client-Server
int release_SIS_Session(int sessionId);

/*
       Used to open a connection to the SIS Server
*/
int  open_connection(int sessionId);
int  close_connection(int sessionId);
void set_server_info(char *Server, int port);
int  GetSocket(int sessionId);

/*
       Used by tools like the telos parser to get locks
*/
int get_writelock(int sessionId);
int get_readlock(int sessionId);
int release_lock(int sessionId);

/*
       Begin-end a Transaction or Query session
*/
int begin_transaction(int sessionId);
int abort_transaction(int sessionId);
int end_transaction(int sessionId);
int begin_query(int sessionId);
int end_query(int sessionId);

/*
       Get or set error messages from/to the SIS Server
*/
int put_error_message(int sessionId, char *message);
int get_error_message(int sessionId, char *message);
int reset_error_message(int sessionId);

int reset_user_error_message(int sessionId);
int get_user_error_message(int sessionId, char* mess, int *NumOfStrParams, int
*StrSize, int *NumOfIntParams);
int get_user_error_params(int sessionId, char *StrParams, int NumOfStrParams, int
StrSize, int *IntParams, int NumOfIntParams);
int put_user_error_message(int sessionId, char* mess);
```

```
/*
       Functions used to set the cm_value fields.
*/
int assign_node(int sessionId, cm_value *val, char *s, int id);
int assign_string(int sessionId, cm_value *val, char *s);
int assign_int(int sessionId, cm_value *val, int n);
int assign_float(int sessionId, cm_value *val, float r);
int assign_time(int sessionId, cm_value *val, TIME t);
int assign_empty(int sessionId, cm_value *val);
int cm_to_str(int sessionId, cm_value *val, char *str, int str_size);
int string_to_primitive(int sessionId, char *message, char *value, cm_value *cmval);

/*
       Function used to free allocated space for get_xml_description()
      and cm_value string fields.
*/
void free_sis_allocated_space(int sessionId, char* buffer);

/*
       Query Functions
*/
int reset_query(int sessionId);
int reset_name_scope(int sessionId);
int pop_name_scope(int sessionId);
int set_current_node(int sessionId, char *str);
int set_current_node_id(int sessionId, int id);
int set_categories(int sessionId, categories_set);
int set_depth(int sessionId, int depth);
int set_num_of_proj(int sessionId, int n);
int get_category_of_link_from(int sessionId, int set_id, char *label);
int get_traverse_by_all_links(int sessionId, int set_id, int isa);
int get_traverse_by_category(int sessionId, int set_id, int isa);
int get_traverse_by_meta_category(int sessionId, int set_id, int isa);
int get_matched(int sessionId, int obj_set_id, int ptrn_set_id);
int get_matched_case_insensitive(int sessionId, int obj_set_id, int ptrn_set_id, int
encoding);
int get_matched_string(int sessionId, int obj_set, cm_value *cmv, int);
int get_classid(int sessionId, l_name lname, int *sysid);
int get_linkid(int sessionId, l_name fromcls, l_name label, int *sysid);
int get_loginam(int sessionId, int sysid, l_name lname);
int get_filtered(int sessionId, int set_id);
int get_classes(int sessionId, int set_id);
int get_all_classes(int sessionId, int set_id);
int get_Sysclass(int sessionId, int set_id);
int get_all_Sysclasses(int sessionId, int set_id);
int get_instances(int sessionId, int set_id);
int get_all_instances(int sessionId, int set_id);
int get_superclasses(int sessionId, int set_id);
int get_all_superclasses(int sessionId, int set_id);
int get_all_Syssuperclasses(int sessionId, int set_id);
int get_subclasses(int sessionId, int set_id);
int get_all_subclasses(int sessionId, int set_id);
int get_link_from(int sessionId, int set_id);
int get_class_attr_from(int sessionId, int set_id);
int get_inher_class_attr(int sessionId, int set_id);
int get_class_attr(int sessionId, int set_id);
int get_all_class_attr(int sessionId, int set_id);
int get_inher_link_from(int sessionId, int set_id);
int get_inher_link_to(int sessionId, int set_id);
int get_link_to(int sessionId, int set_id);
int get_category_from(int sessionId, int set_id);
int get_category_to(int sessionId, int set_id);
int get_link_from_by_category(int sessionId, int set_id, l_name fromcls, l_name
categ);
int get_link_from_by_meta_category(int sessionId, int set_id, l_name fromcls, l_name
categ);
int get_link_to_by_category(int sessionId, int set_id, l_name fromcls, l_name categ);
int get_link_to_by_meta_category(int sessionId, int set_id, l_name fromcls, l_name
categ);
int get_to_node(int sessionId, int set_id);
int get_from_node(int sessionId, int set_id);
int get_to_node_by_category(int sessionId, int set_id, l_name fromcls, l_name categ);
int get_from_node_by_category(int sessionId, int set_id, l_name fromcls, l_name
categ);
int get_to_node_by_meta_category(int sessionId, int set_id, l_name fromcls, l_name
categ);
int get_from_node_by_meta_category(int sessionId, int set_id, l_name fromcls, l_name
categ);
int get_from_value(int sessionId, int set_id);
int get_to_value(int sessionId, int set_id);
```

```
/*
        Functions used to work with sets
*/
int reset_set(int sessionId, int set_id);
int set_position(int sessionId, int set_id, int pos);
int set_clear(int sessionId, int set_id);
int set_clear_lower(int sessionId, int set_id);
int reset_edge_set(int sessionId);
int free_set(int sessionId, int set_id);
int free_all_sets(int sessionId);
int set_get_new(int sessionId);
int set_get_card(int sessionId, int set_id);
int set_put(int sessionId, int set_id);
int set_put_prm(int sessionId, int set_id, cm_value *cmval);
int set_del(int sessionId, int set_id);
int set_member_of(int sessionId, int set_id);
int set_union(int sessionId, int set_id1, int set_id2);
int set_copy(int sessionId, int set_id1, int set_id2);
int set_intersect(int sessionId, int set_id1, int set_id2);
int set_difference(int sessionId, int set_id1, int set_id2);
int set_disjoint(int sessionId, int set_id1, int set_id2);
int set_equal(int sessionId, int set_id1, int set_id2);

/*
        Functions used to obtain set contents
*/
int return_nodes(int sessionId, int set_id, l_name cls);
int return_edge_nodes(int sessionId, int *sysid, l_name node);
int return_full_nodes(int sessionId, int set_id, int *sysid, l_name node, l_name
Sclass);
int return_prm(int sessionId, int set_id, cm_value *cmv);
int return_categories(int sessionId, int set_id, l_name cls, l_name categ);
int return_categ_ids(int sessionId, int set_id, int *sysid, l_name cls, l_name categ);
int return_link(int sessionId, int set_id, l_name cls, l_name label, cm_value *cmv);
int return_link_id(int sessionId, int set_id, l_name cls,int *fcid, int
*sysid,cm_value *cmv,  int *traversed);
int return_full_link(int sessionId, int set_id, l_name cls, l_name label, l_name
categ, l_name fromcls, cm_value *cmv, int *unique_category, int *traversed);
int return_full_link_id(int sessionId, int set_id, l_name cls, int *clsid, l_name
label, int *linkid, l_name categ, l_name fromcls, int *catid, cm_value *cmv, int
*unique_category);
int return_isA(int sessionId, int set_id, l_name ob1, l_name ob2);
int return_isA_id(int sessionId, int set_id, l_name ob1, int *id1, l_name ob2, int
*id2);
int return_inst(int sessionId, int set_id, l_name ob1, l_name ob2);
int return_inst_id(int sessionId, int set_id, l_name ob1, int *id1, l_name ob2, int
*id2);
int return_field(int sessionId, int set_id, cm_value *cmv);
int return_projection(int sessionId, int set_id, cm_value *cmv);
int return_hidlink(int sessionId, int set_id, l_name cls,int *cls_id, l_name label,int
*sysid, cm_value *cmv1, cm_value *cmv2);
int return_xml_description(int sessionId, int set_id, char** xml_string);

/*
        Functions used to work with projections
*/
int set_tv_cond(int sessionId, int exp);
int set_fv_cond(int sessionId, int exp);
int set_tl_cond(int sessionId, int exp);
int set_fl_cond(int sessionId, int exp);
int set_filter_cond(int sessionId, int exp);

int set_proj_cond1(int sessionId, int exp);
int set_proj_cond2(int sessionId, int exp);
int set_proj_cond3(int sessionId, int exp);
int set_proj_cond4(int sessionId, int exp);

/*
        Functions used to update the SIS Database
*/
int Add_Node(int sessionId, IDENTIFIER * node_name, int level);
int Add_Named_Attribute(int sessionId, IDENTIFIER *attribute,IDENTIFIER *from,
cm_value *to, int iLevel, int catSet);
int Add_Unnamed_Attribute(int sessionId, IDENTIFIER * from, cm_value * to, int
catSet);
int Add_Instance_Set(int sessionId, int from_set, IDENTIFIER * to);
int Add_Instance(int sessionId, IDENTIFIER * from, IDENTIFIER * to);
int Add_IsA(int sessionId, IDENTIFIER * from, IDENTIFIER * to);
int Delete_Node(int sessionId, IDENTIFIER * node_name);
int Delete_Named_Attribute(int sessionId, IDENTIFIER *attribute, IDENTIFIER * from);
int Delete_Unnamed_Attribute(int sessionId, IDENTIFIER * attribute);
```

```
int Delete_Instance_Set(int sessionId, int from_set, IDENTIFIER * to);
int Delete_Instance(int sessionId, IDENTIFIER * from, IDENTIFIER * to);
int Delete_IsA(int sessionId, IDENTIFIER * from, IDENTIFIER * to);
int Rename_Node(int sessionId, IDENTIFIER * node, IDENTIFIER * NewNodeName);
int Rename_Named_Attribute(int sessionId, IDENTIFIER *attribute, IDENTIFIER * from,
IDENTIFIER * NewName);
int Change_Named_Attribute_To(int sessionId, IDENTIFIER *Attribute, IDENTIFIER *From,
cm_value *To, cm_value *NewTo);
int Change_Unnamed_Attribute_To(int sessionId, IDENTIFIER *attribute,  cm_value *To,
cm_value *NewTo);
int Change_Instance_To(int sessionId, IDENTIFIER * from, IDENTIFIER * to, IDENTIFIER *
NewTo);
int Change_IsA_To(int sessionId, IDENTIFIER * from, IDENTIFIER * to, IDENTIFIER *
NewTo);

/*
       Functions used to work on tuples
*/
int return_relation(int sessionId, int Fromset, int CatSet, int ToSet);
int unary_on_tuple(int sessionId, int set_id);
int return_tuple(int sessionId, int set_id, cm_value *Item);
int get_new_col(int sessionId, int set_id);
int tuple_reset_join_pos(int sessionId, int set_id);
int tuple_set_input_col(int sessionId, int set_id,int pos);
int tuple_set_join_pos(int sessionId, int set_id, int x, int y);
int tuple_no_projection_column(int sessionId, int set_id,int x);
int tuple_join(int sessionId, int set_id, int x);
int tuple_union(int sessionId, int set_id, int x);
int tuple_difference(int sessionId, int set_id, int x);

/*
       Operators
*/
int SYS_ID(int sessionId, int obj);
int NODE(int sessionId, l_name nodename);
int LINK(int sessionId, l_name fromcls, l_name linkname);
int STR_VAL(int sessionId, char* string);
int VAL(int sessionId, int int_value);
int CARD(int sessionId, int set_id);
int SUCC(int sessionId);
int FAIL(int sessionId);
int AND(int sessionId, int exp1, int exp2);
int OR(int sessionId, int exp1, int exp2);
int NOT(int sessionId, int exp);
int BELONGS(int sessionId, int obj, int set_id);
int MATCH(int sessionId, int ptrn_set_id, int set_id);
int EQ(int sessionId, int val1, int val2);
int GT(int sessionId, int val1, int val2);
int GTE(int sessionId, int val1, int val2);
int LT(int sessionId, int val1, int val2);
int LTE(int sessionId, int val1, int val2);
int BEFORE(int sessionId, int tm1 , int tm2);
int AFTER(int sessionId, int tm1 , int tm2);
int TIME_EQUAL(int sessionId, int tm1 , int tm2);
int MEETS(int sessionId, int tm1 , int tm2);
int MET_BY(int sessionId, int tm1 , int tm2);
int OVERLAPS(int sessionId, int tm1 , int tm2);
int OVERLAPPED_BY(int sessionId, int tm1 , int tm2);
int DURING(int sessionId, int tm1 , int tm2);
int CONTAINS(int sessionId, int tm1 , int tm2);
int STARTS(int sessionId, int tm1 , int tm2);
int STARTED_BY(int sessionId, int tm1 , int tm2);
int FINISHES(int sessionId, int tm1 , int tm2);
int FINISHED_BY(int sessionId, int tm1 , int tm2);
int CBEQ(int sessionId, int tm1 , int tm2);
int CBLT(int sessionId, int tm1 , int tm2);
int CBLE(int sessionId, int tm1 , int tm2);
int CBGT(int sessionId, int tm1 , int tm2);
int CBGE(int sessionId, int tm1 , int tm2);
int MBEQ(int sessionId, int tm1 , int tm2);
int MBLT(int sessionId, int tm1 , int tm2);
int MBLE(int sessionId, int tm1 , int tm2);
int MBGT(int sessionId, int tm1 , int tm2);
int MBGE(int sessionId, int tm1 , int tm2);
int SET_EQUAL(int sessionId, int set1, int set2);
int SET_DISJOINT(int sessionId, int set1, int set2);
int SET_ID(int sessionId, int set_id);
int SET_UNION(int sessionId, int set1, int set2);
int SET_COPY(int sessionId, int set1, int set2);
int SET_INTERSECT(int sessionId, int set1, int set2);
int SET_DIFFERENCE(int sessionId, int set1, int set2);
```

```
int GET_CLASSES(int sessionId, int val);
int GET_ALL_CLASSES(int sessionId, int val);
int GET_SYSCLASS(int sessionId, int val);
int GET_ALL_SYSCLASSES(int sessionId, int val);
int GET_INSTANCES(int sessionId, int val);
int GET_ALL_INSTANCES(int sessionId, int val);
int GET_SUPERCLASSES(int sessionId, int val);
int GET_ALL_SUPERCLASSES(int sessionId, int val);
int GET_ALL_SYSSUPERCLASSES(int sessionId, int val);
int GET_SUBCLASSES(int sessionId, int val);
int GET_ALL_SUBCLASSES(int sessionId, int val);
int GET_LINK_FROM(int sessionId, int val);
int GET_CLASS_ATTR_FROM(int sessionId, int val);
int GET_CLASS_ATTR(int sessionId, int val);
int GET_ALL_CLASS_ATTR(int sessionId, int val);
int GET_LINK_TO(int sessionId, int val);
int GET_FROM_VALUE(int sessionId, int val);
int GET_TO_VALUE(int sessionId, int val);
int GET_LINK_FROM_BY_CATEGORY(int sessionId, int val, int obj);
int GET_LINK_TO_BY_CATEGORY(int sessionId, int val, int obj);
```

Appendix B - An Example

Let's see a full example

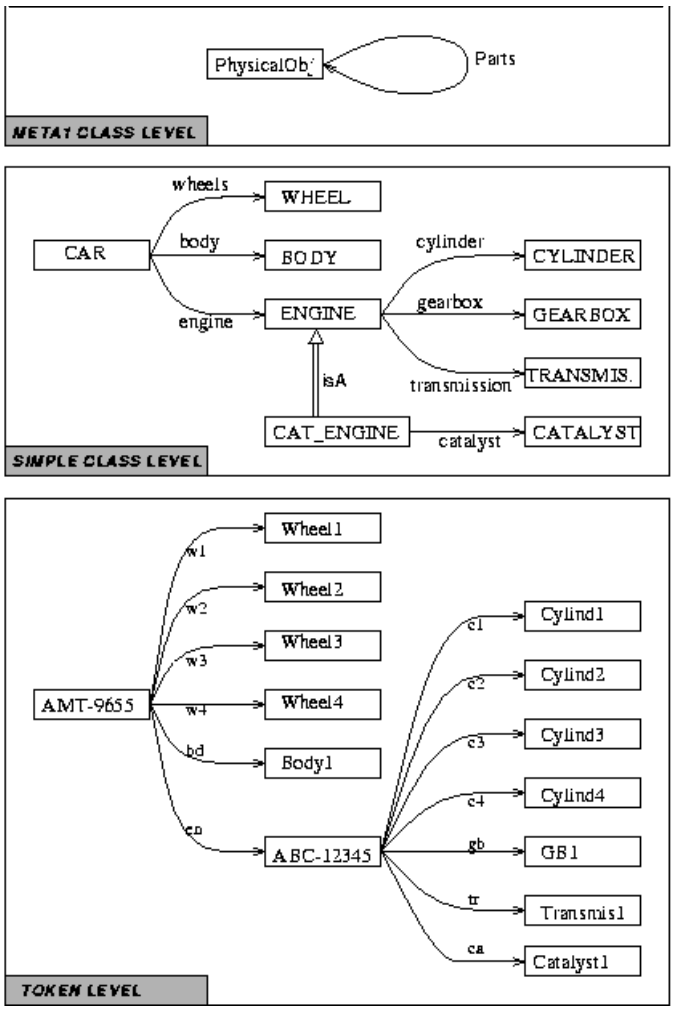

Figure 2 : *A* TELOS *model*

A model from the real world that can be easily described with the TELOS language. The instance relationship is not shown in the picture in order not to be too complicated. The instance relationships that hold are the obvious. CAR, WHEEL, BODY, ENGINE, CAT_ENGINE, CYLINDER, GEARBOX, TRANSMIS and CATALYST are all instances of PlysicalObj and wheels, body, engine, cylinder, gearbox, transmission and catalyst link objects are instances of Parts attribute of PlysicalObj. Similarly there are instance relations between the objects at Token Level and objects at Simple Class Level. Object ABC-12345 is instance of CAT_ENGINE simple class object.

Here follows the TELOS code which describes the semantic net of Figure 2.

```
BEGINTRANSACTION

{--------------- META1 CLASS LEVEL ------------------}

TELL Individual PhysicalObj in M1_Class
with attribute
     Parts : PhysicalObj
end PhysicalObj

{--------------- SIMPLE CLASS LEVEL ------------------}

TELL Individual CAR in S_Class, PhysicalObj
with Parts
       wheels : WHEEL;
       body   : BODY;
       engine : ENGINE
end CAR

TELL Individual WHEEL in S_Class, PhysicalObj
end WHEEL

TELL Individual BODY in S_Class, PhysicalObj
end BODY

TELL Individual ENGINE in S_Class, PhysicalObj
with Parts
     cylinder : CYLINDER;
     gearbox  : GEARBOX;
     transmission : TRANSMISSION
end ENGINE

TELL Individual CYLINDER in S_Class, PhysicalObj
end CYLINDER

TELL Individual GEARBOX in S_Class, PhysicalObj
end GEARBOX

TELL Individual  TRANSMISSION in S_Class, PhysicalObj
end TRANSMISSION

TELL Individual CAT_ENGINE in S_Class, PhysicalObj isA ENGINE
with  Parts
        catalyst : CATALYST
end CAT_ENGINE

TELL Individual CATALYST in S_Class, PhysicalObj
end CATALYST


{--------------- TOKEN CLASS LEVEL ------------------}

TELL Individual (AMT-9655) in Token, CAR
with wheels
     w1 : Wheel1;
     w2 : Wheel2;
     w3 : Wheel3;
     w4 : Wheel4
with body
     bd : Body1
with engine
     en : (ABC-12345)
end (AMT-9655)

TELL Individual Wheel1 in Token, WHEEL
end Wheel1

TELL Individual Wheel2 in Token, WHEEL
end Wheel2

TELL Individual Wheel3 in Token, WHEEL
end Wheel3

TELL Individual Wheel4 in Token, WHEEL
end Wheel4

TELL Individual Body1 in Token, BODY
```

```
end Body1

TELL Individual (ABC-12345) in Token, CAT_ENGINE
with cylinder
     c1 : Cylind1;
     c2 : Cylind2;
     c3 : Cylind3;
     c4 : Cylind4
with gearbox
     gb : GB1
with transmission
     tr : Transmis1
with catalyst
     ca : Catalyst1
end (ABC-12345)


TELL Individual Cylind1 in Token, CYLINDER
end Cylind1

TELL Individual Cylind2 in Token, CYLINDER
end Cylind2

TELL Individual Cylind3 in Token, CYLINDER
end Cylind3

TELL Individual Cylind4 in Token, CYLINDER
end Cylind4

TELL Individual GB1 in Token, GEARBOX
end GB1

TELL Individual Transmis1 in Token, TRANSMISSION
end Transmis1

TELL Individual Catalyst1 in Token, CATALYST
end Catalyst1

ENDTRANSACTION
```

Here follows the source code of an application, which uses the C programmatic interface (SIS C-API) to query the SIS data base which contains the model of **Figure 2**.

The application, does the following :

1. Prints instances of "*PhysicalObj*".
2. Prints instances of "*PhysicalObj*" and their instances too.
3. Prints logical names of all links pointing from object *CAR*.
4. Prints information for all links pointing from object *CAR*.
5. Prints all links of meta-category *Parts* traversed, starting from object *AMT-9655*.

```
/*****************************************************************
 *
 *                     Semantic Index System
 *
 *   COPYRIGHT (c) 1992 by Institute of Computer Science,
 *                         Foundation of Research and Technology - Hellas
 *                         POBox 1385, Heraklio Crete, GR-711 10 GREECE
 *
 *
 *                       ALL RIGHTS RESERVED
 *
 *   This software is furnished under license and may be used only in
 *   accordance with the terms of that license and with the inclusion
 *   of the above copright notice.  This software may not be provided
 *   or otherwise made available to, or used by, any other person. No
 *   title to or ownership of the software is hereby transferred.
 *
 *
 *   Module  :
 *   Version :
 *
 *   Purpose :
 *
 *   Author  :
 *   Creation Date :                    Date of last update :
 *
 *   Remarks :
 *
 *
 *****************************************************************/

#include "stdlib.h"
#include "stdio.h"
#ifdef SIS_WIN32
        #include "conio.h"
#endif

#include "sis_kernel/time.h"
#include "cpp_api/cs_defs.h"
#include "cpp_api/identifier.h"
#include "cpp_api/sis_classes.h"
#include "cpp_api/c_session_wrapper.h"



void print_value(cm_value *val);

 main(int argc, char** argv)
 {
     l_name   cls,
              label;
     categories_set categs;
     int      ret_set1,
              ret_set2;
     cm_value cmv;
     int sessionID;



     if (argc != 3) {
         fprintf(stderr, "Error invoking demo\n");
         fprintf(stderr, "Usage : %s <host> <port> where server resides\n",
                  argv[0]);
        exit(-1);
     }
 #ifdef CLIENT_SERVER
     create_SIS_CS_Session(&sessionID, argv[1], atoi(argv[2]), "", "" );
 #else
     int  *start_t = NULL;
     char *db_dir;
     get_db_dir(&db_dir); /* exits if DB_DIR is not defined ... */
     start_t = init_start_telos(db_dir);
     create_SIS_SA_Session(&sessionID, start_t, argv[1], atoi(argv[2]), "", "");
 #endif

     open_connection(sessionID);
     begin_query(sessionID);

     reset_name_scope(sessionID);
```

```
     if (set_current_node(sessionID, "PhysicalObj") != APIFail)
       if ((ret_set1 = get_instances(sessionID, 0)) != APIFail) {
         reset_set(sessionID, ret_set1);
         printf("\nINSTANCES OF 'PhysicalObj' : \n");
         while (return_nodes(sessionID, ret_set1, label) != APIFail)
            printf("      %s\n", label);
       }

       if ((ret_set2 = get_instances(sessionID, ret_set1)) != -1) {
         set_union(sessionID, ret_set1, ret_set2);
         reset_set(sessionID, ret_set1);
         printf("\nALL INSTANCES OF 'PhysicalObj' : \n");
         while (return_nodes(sessionID, ret_set1, label) != APIFail)
            printf("      %s\n", label);
       }
     free_all_sets(sessionID);
     reset_name_scope(sessionID);
     if (set_current_node(sessionID, "CAR") != APIFail)
       if ((ret_set1 = get_link_from(sessionID, 0)) != APIFail) {
         reset_set(sessionID, ret_set1);
         printf("\nLINKS FROM 'CAR' : \n");
         while (return_nodes(sessionID, ret_set1, label) != APIFail)
            printf("      %s\n", label);

         reset_set(sessionID, ret_set1);
         printf("\nINFO ABOUT LINKS FROM 'CAR' : \n");
         while (return_link(sessionID, ret_set1, cls, label, &cmv) != APIFail){
            printf("   %10s --- %10s ---> ", cls, label);
            print_value(&cmv);
         }
       }

     strcpy(categs[0].fcl, "PhysicalObj");
     strcpy(categs[0].cat, "Parts");
     categs[0].direction = FORWARD;
     categs[1].direction = 0;

     set_categories(sessionID, categs);

     free_all_sets(sessionID);
     reset_name_scope(sessionID);
     if (set_current_node(sessionID, "AMT-9655") != APIFail)
     if ((ret_set1 = get_traverse_by_meta_category(sessionID, 0, NOISA)) != APIFail) {
       reset_set(sessionID, ret_set1);
       printf("\nTRANSITIVE QUERY FROM 'AMT-9655' (meta-category : Parts) : \n");
       while ((return_link(sessionID, ret_set1, cls, label, &cmv)) != APIFail) {
         printf("   %10s --- %10s ---> ", cls, label);
         print_value(&cmv);
       }
     }

     end_query(sessionID);
     close_connection(sessionID);
     release_SIS_Session(sessionID);
     // Wait for a keyboard hit
   printf("Press any key to exit...\n");
   getchar();
 }



void print_value(cm_value *val)
{
     char *tmp;

     if (val == NULL) {
        return;
     }

     switch (val->type) {
        case TYPE_INT    :  printf("Integer       : %d \n",val->value.n);
                            break;
        case TYPE_STRING :  printf("String        : %s \n",val->value.s);
                            break;
        case TYPE_FLOAT  :  printf("Float         : %f \n",val->value.r);
                            break;
        case TYPE_NODE   :  printf("Logical Name  : %s \n",val->value.s);
                            break;
        case TYPE_TIME   :  tmp = (val->value.t).present();
                            printf("Time          : %s \n",tmp);
                            free(tmp);
```

```
                        break;
    }
}
```

On WIN32 systems the include files that should be used to compile this code are:

```
sis_kernel/time.h,
cpp_api/cs_defs.h,
cpp_api/identifier.h,
cpp_api/sis_classes.h,
cpp_api/c_session_wrapper.h.
```

On WIN32 systems the libraries (Borland 5.01 libraries) that should be used to link this code are:

Client – Server:

- lib_c_api_session_cs_2b.lib (The C interface SIS API)
- cpp_api_cs_2b.lib (The C++ interface SIS API)
- lib_sis_kernel_2b.lib (The SIS Kernel)
- lib_time_2b.lib (Time functions library)
- ccomms_2b.lib (Client Communications)
- connection_2b.lib (Used to open a connection to the server)
- libl.lib

Direct Access:

No direct access interface (using sessions) is provided. Instead we provide a set of libraries for backward compatibility that we describe in "Appendix G - Backwards compatibility".

C API on DLL:

- api_dll.dll (The C interface SIS API)

Note: the TIME class is not implemented in dll library. So in order to compile the example, the code in function **print_value()** should be changed:

```
TYPE_TIME   :  /* not implemented yet */
        /*
        tmp = (val->value.t).present();
        printf("Time          : %s \n",tmp);
        free(tmp);
        */
        break;
```

Also notice that operator functions decribed in sections 6.3.3.1, 6.3.3.2, 6.3.3.3 and 6.3.3.4 are renamed (due to dll-library naming concflict) having `OPER_` as prefix (e.g. function `GET_SUPERCLASSES()` was renamed to `OPER_GET_SUPERCLASSES()`).

Appendix C - Changes from previous versions

In the process of upgrading the functionality of the Application Programmatic Interface some functions changed name in order to be more readable or to be in accordance with the API function naming conventions. Some other functions changed the number or the order of their arguments to be in accordance with the API function argument passing conventions.

Changes from version 1.3 to version 1.3.1

The function **MATCH**(*int set_id, int prtn_set_id*) replaced the function **MATCH***(int prtn_set_id, int set_id*). The order of the argument changed.

The function **get_matched**(*int obj_set_id, int prtn_set_id*) replaced the function **get_matched**(*int prtn_set_id, int obj_set_id*). The order of the argument changed.

The function **set_put_prm**(*int set_id, cm_value *cmval*) replaced the function **set_put_pri**(*int set_id, cm_value *cmval*). The name of the function changed.

The function **return_prm**(*int set_id, cm_value *cmval*) replaced the function **return_prs**(*int set_id, cm_value *cmval*). The name of the function changed.

Some new functions have been added to the API: (a) all the functions that handle tuples e.g. **tuple_join**() (described in section 6.6, (b) functions to compare time intervals e.g. **CBEQ**() (described in section 6.3.3.1).

New functions for selecting and establishing the communication with the server have been added: **set_server_info**(), **get_db_dir**(), and **init_start_telos**().

The function **begin_query**(*int* start_t*) replaced the function **begin_query**(). The argument is declared.

Changes from version 1.3.1 to version 2.0

None. The manual version-numbering follows the code version-numbering.

Changes from version 2.0 to version 2.1

New functions for selecting and establishing the communication with the server have been added: **open_connection**() and **close_connection**().

The functionality of the functions **begin_query**() and **end_query**() changed. They are no longer used to establish the communication with the server, but only to start and close the query sessions.

Changes from version 2.1 to version 2.2

Addition of section 6.7 (*Update Functions*).

Change of the return values of the **begin_query**() and **begin_transaction**() API functions.

Addition of DBUserName DBUserPassword parameters to the init_SIS_API_CS and init_SIS_API_SA functions.

The function **get_inher_link_from**() replaced the function **get_inher_link**(). Its functionality changed. The function **get_inher_link_to**() was added.

Changes from version 2.2 to version 2.2.1

In order to provide real multi-threading to the clients that were using the SIS, we introduced the notion of sessions for SIS C and Java programmatic interface.

Creating multiple instances of QClass (JAPI) was not enough to provide multi-thread access to the SIS-Server, since the underlying libraries (dll's) did not support it. Now these libraries support such mechanism via sessions. Thus now creating multiple QClass instances (JAPI) and creating separate sessions for each instance enables the application developer to have real multi-thread access to the SIS-Server.

All C-API functions have changed: they all take as first argument the sessionID (integer). A session is created by the functions **create_SIS_CS_Session()** and **create_SIS_SA_Session(),** which create a session and returns its ID. These functions have replaced the functions **init_SIS_API_CS()** and **init_SIS_API_SA()** accordingly. A session that is no longer needed may be released by **release_SIS_Session(),** which replaced the function **close_SIS_API().**

Also a case insensitive search mechanism was added. We introduced function **get_matched_case_insensitive().**

Changes from version 2.2.1 to version 2.2.2

The xml description generation mechanism was added. We introduced function **return_xml_description().**

We also introduced function **free_sis_allocated_space()** to enable deallocating space that was allocated for the xml description buffer or the `struct cm_value` string elements.

Appendix D - C++ Programmatic Interface

In the following we present the basic differences between the C application programmatic interface (C-API) and the C++ application programmatic Interface (C++ API).

The interface is built on two classes (`sis_api, SIS_Connection`) which provide as public member functions all the functions of described in this document. SIS_Connection class provides the connection and transaction handling (for quering/updating) mechanism: `open_connection(), close_connection(), begin_query(), end_query(), begin_transaction(), end_transaction()`, etc., while `sis_api` class provides the rest of the functions: `set_current_node(), get_classes(), get_instances()`, etc.

Functions such as `create_SIS_CS_Session(), release_SIS_Session()` have no meaning since multi-threading can be achieved by multiple instances of classes sis_api and SIS_Connection. Thus the first argument of the functions of this API (sessionID) is ommited.

Applications build with C++API need **different include files** and **libraries** to be compiled. Below we present an example which is the C++ implementation of the example presented in "*Appendix B - An Example*".

```
/*****************************************************************
 *
 *                     Semantic Index System
 *
 *   COPYRIGHT (c) 1992 by Institute of Computer Science,
 *                         Foundation of Research and Technology - Hellas
 *                         POBox 1385, Heraklio Crete, GR-711 10 GREECE
 *
 *
 *                       ALL RIGHTS RESERVED
 *
 *   This software is furnished under license and may be used only in
 *   accordance with the terms of that license and with the inclusion
 *   of the above copright notice.  This software may not be provided
 *   or otherwise made available to, or used by, any other person. No
 *   title to or ownership of the software is hereby transferred.
 *
 *
 *   Module  :
 *   Version :
 *
 *   Purpose :
 *
 *   Author  :
 *   Creation Date :                   Date of last update :
 *
 *   Remarks :
 *
 *
 *****************************************************************/

#include "stdlib.h"
#include "stdio.h"
#ifdef SIS_WIN32
       #include "conio.h"
#endif

#include "sis_kernel/time.h"
#include "cpp_api/cs_defs.h"
#include "cpp_api/identifier.h"
#include "cpp_api/sis_classes.h"
```

```
#include "sis_kernel/telos_ro.h"
#include "sis_kernel/obj_check.h"
#include "sis_kernel/initial.h"

#include "cpp_api/cs_defs.h"
#ifdef CLIENT_SERVER
       #include "cpp_api/cs_comms.h"
#endif
#include "cpp_api/query_func.h"
#include "cpp_api/set_tuple.h"
#include "cpp_api/q_tmpsets.h"
#include "cpp_api/q_expstack.h"

#include "cpp_api/cs_errcodes.h"
#include "cpp_api/q_ccache.h"

#include "cpp_api/q_class_header.h"
#include "cpp_api/connection.h"


// ------------- Forward declarations -------

void print_value(cm_value *val);


 main(int argc, char** argv)
 {
     l_name   cls,
              label;
     categories_set categs;
     int      ret_set1,
              ret_set2;
     cm_value cmv;

     sis_api *Q1 = NULL;
     SIS_Connection *Connection_class1 = NULL;


     if (argc != 3) {
         fprintf(stderr, "Error invoking demo\n");
         fprintf(stderr, "Usage : %s <host> <port> where server resides\n",
                 argv[0]);
        exit(-1);
     }

     // init_SIS_API_CS(...)
     SOCKET S = 0;
     Q1 = new sis_api(S);
     if(Q1 == NULL) return -1;
     Connection_class1 = new SIS_Connection(Q1, argv[1], atoi(argv[2]), "", "", "");
     if(Connection_class1 == NULL) return -1;

     // open_connection()
     Connection_class1->open_connection();
     S = Connection_class1->GetSocket();
     Q1->SetSocket(S);

     Connection_class1->begin_query();

     Q1->reset_name_scope();
     if (Q1->set_current_node("PhysicalObj") != APIFail)
       if ((ret_set1 = Q1->get_instances(0)) != APIFail) {
         Q1->reset_set(ret_set1);
         printf("\nINSTANCES OF 'PhysicalObj' : \n");
         while (Q1->return_nodes(ret_set1, label) != APIFail)
            printf("      %s\n", label);
       }

       if ((ret_set2 = Q1->get_instances(ret_set1)) != -1) {
         Q1->set_union(ret_set1, ret_set2);
         Q1->reset_set(ret_set1);
         printf("\nALL INSTANCES OF 'PhysicalObj' : \n");
         while (Q1->return_nodes(ret_set1, label) != APIFail)
            printf("      %s\n", label);
       }
     Q1->free_all_sets();
     Q1->reset_name_scope();
     if (Q1->set_current_node("CAR") != APIFail)
       if ((ret_set1 = Q1->get_link_from(0)) != APIFail) {
         Q1->reset_set(ret_set1);
         printf("\nLINKS FROM 'CAR' : \n");
```

```
          while (Q1->return_nodes(ret_set1, label) != APIFail)
             printf("      %s\n", label);

          Q1->reset_set(ret_set1);
          printf("\nINFO ABOUT LINKS FROM 'CAR' : \n");
          while (Q1->return_link(ret_set1, cls, label, &cmv) != APIFail){
             printf("   %10s --- %10s ---> ", cls, label);
             print_value(&cmv);
          }
        }

      strcpy(categs[0].fcl, "PhysicalObj");
      strcpy(categs[0].cat, "Parts");
      categs[0].direction = FORWARD;
      categs[1].direction = 0;

      Q1->set_categories(categs);

      Q1->free_all_sets();
      Q1->reset_name_scope();
      if (Q1->set_current_node("AMT-9655") != APIFail)
      if ((ret_set1 = Q1->get_traverse_by_meta_category(0, NOISA)) != APIFail) {
        Q1->reset_set(ret_set1);
        printf("\nTRANSITIVE QUERY FROM 'AMT-9655' (meta-category : Parts) : \n");
        while ((Q1->return_link(ret_set1, cls, label, &cmv)) != APIFail) {
          printf("   %10s --- %10s ---> ", cls, label);
          print_value(&cmv);

        }
      }

      Connection_class1->end_query();

      // close_connection(...)
      Connection_class1->close_connection();

      // close_SIS_API(...)
      if(Q1 != NULL) delete Q1;
      if(Connection_class1 != NULL) delete Connection_class1;

      // Wait for a keyboard hit
      printf("Press any key to exit...\n");
      getchar();
}

void print_value(cm_value *val)
{
      char *tmp;

      if (val == NULL) {
         return;
      }

      switch (val->type) {
         case TYPE_INT    :  printf("Integer       : %d \n",val->value.n);
                             break;
         case TYPE_STRING :  printf("String        : %s \n",val->value.s);
                             break;
         case TYPE_FLOAT  :  printf("Float         : %f \n",val->value.r);
                             break;
         case TYPE_NODE   :  printf("Logical Name  : %s \n",val->value.s);
                             break;
         case TYPE_TIME   :  tmp = (val->value.t).present();
                             printf("Time          : %s \n",tmp);
                             free(tmp);
                             break;
      }
}
```

On WIN32 systems the include files that should be used to compile this code are:

```
sis_kernel/time.h,
cpp_api/cs_defs.h,
cpp_api/identifier.h,
cpp_api/sis_classes.h,
sis_kernel/telos_ro.h,
sis_kernel/obj_check.h,
sis_kernel/initial.h,
```

`cpp_api/cs_defs.h`,
`cpp_api/cs_comms.h`,
`cpp_api/query_func.h`,
`cpp_api/set_tuple.h`,
`cpp_api/q_tmpsets.h`,
`cpp_api/q_expstack.h`,
`cpp_api/cs_errcodes.h`,
`cpp_api/q_ccache.h`,
`cpp_api/q_class_header.h` (contains the definition of sis_api),
`cpp_api/connection.h` (contains the definition of SIS_Connection).

On WIN32 systems the libraries (Borland 5.01 libraries) that should be used to link this code are:

Client – Server:

- cpp_api_cs_2b.lib (The C++ interface SIS API)
- lib_sis_kernel_2b.lib (The SIS Kernel)
- lib_time_2b.lib (Time functions library)
- ccomms_2b.lib (Client Communications)
- connection_2b.lib (Used to open a connection to the server)
- libl.lib

Direct Access:

- cpp_api_sa_2b.lib (The C++ interface SIS API)
- lib_sis_kernel_big_2b_cs.lib (The SIS Kernel)
- lib_time_2b.lib (Time functions library)
- ccomms_2b.lib (Client Communications)
- connection_2b.lib (Used to open a connection to the server)
- libl.lib

Notice that before running an application that access directly an SIS base the environment variable `DB_DIR` have to be set (`DB_DIR`: locates the directory where the SIS database exists).

C API on DLL:

There is no C++ interface on provided on dll.

Appendix E - Java Programmatic Interface

In the following we present the basic differences between the C application programmatic interface (C-API) and Java application programmatic interface (JAPI).

The interface is built on class `QClass`, which provides as public member functions all the functions of described in this document. The only difference is the functions described in sections 6.3.3.1, 6.3.3.2, 6.3.3.3 and 6.3.3.4 are having `oper` as prefix (e.g. function `GET_SUPERCLASSES()` was renamed to `operGET_SUPERCLASSES())`.

The structures defined for argument passing, such as `cm_value, category_set` etc. are implemented as separate a package and all the classes and dll's are provided in a jar file (called "japi14.jar" for version 1.4 of the Java-API, which is the version presented here). We use these structures as arguments to the functions described in this document. The Java classes that replace them are:

- `Int` replaces `int` (an integer argument, whenever is used 'pass-by-value')
- `IntegerObject` replaces `int*` (an integer argument, whenever is used 'pass-by-reference')
- `String` replaces `char*` (a string argument, whenever is used 'pass-by-value')
- `StringObject` replaces `char*` (a string argument, whenever is used 'pass-by-reference')
- `CMValue` replaces `cm_value`
- `CategorySet` replaces `category_set`
- `Identifier` replaces `IDENTIFIER`
- `Time` replaces `TIME` (not implemented yet)

Also **free_sis_allocated_space()** is not provided in SIS JAPI.

Below we present an example which is the Java implementation of the example presented in "Appendix B - An Example".

```
import java.io.*;

class Car {

public Car() {
    QClass Q = new QClass();
    IntegerObject sis_session = new IntegerObject();

    Q.create_SIS_CS_Session(sis_session, "agnes",1291, "", "");
    Q.open_connection(sis_session.getValue());

    Q.begin_query(sis_session.getValue());
    Q.reset_name_scope(sis_session.getValue());


    StringObject strobject = new StringObject();
    StringObject label = new StringObject();
    StringObject cls = new StringObject();
    CMValue cmv = new CMValue();

    int ret_set1 = -1, ret_set2 = -1;

    Q.reset_name_scope(sis_session.getValue());
    strobject.setValue("PhysicalObj");
    if (Q.set_current_node(sis_session.getValue(), strobject) != Q.APIFail)
        if ((ret_set1 = Q.get_instances(sis_session.getValue(), 0)) != Q.APIFail) {
            Q.reset_set(sis_session.getValue(), ret_set1);
            System.out.println("\nINSTANCES OF 'PhysicalObj' : \n");
            while (Q.return_nodes(sis_session.getValue(),ret_set1, label) !=
Q.APIFail)
```

```
                System.out.println(label);
        }

    if ((ret_set2 = Q.get_instances(sis_session.getValue(),ret_set1)) != Q.APIFail) {
        Q.set_union(sis_session.getValue(), ret_set1, ret_set2);
        Q.reset_set(sis_session.getValue(), ret_set1);
        System.out.println("\nALL INSTANCES OF 'PhysicalObj' : \n");
        while (Q.return_nodes(sis_session.getValue(), ret_set1, label) != Q.APIFail)
            System.out.println("    " +label);
    }

    Q.free_all_sets(sis_session.getValue());
    Q.reset_name_scope(sis_session.getValue());
    strobject.setValue("CAR");
    if (Q.set_current_node(sis_session.getValue(), strobject) != Q.APIFail)
        if ((ret_set1 = Q.get_link_from(sis_session.getValue(), 0)) != Q.APIFail) {
            Q.reset_set(sis_session.getValue(), ret_set1);
            System.out.println("\nLINKS FROM 'CAR' : \n");
            while (Q.return_nodes(sis_session.getValue(), ret_set1, label) !=
Q.APIFail)
                System.out.println("    " + label);

            Q.reset_set(sis_session.getValue(), ret_set1);
            System.out.println("\nINFO ABOUT LINKS FROM 'CAR' : \n");
            while (Q.return_link(sis_session.getValue(), ret_set1, cls, label, cmv) !=
Q.APIFail){
                System.out.print(" " + cls + " --- " + label + " ---> ");
                print_value(cmv);
            }
        }

    Q.free_all_sets(sis_session.getValue());
    Q.reset_name_scope(sis_session.getValue());

    StringObject strfclA = new StringObject("PhysicalObj");
    StringObject strcatA = new StringObject("Parts");
    StringObject strfclB = new StringObject("end");
    StringObject strcatB = new StringObject("end");

    CategorySet[] categs= new CategorySet[2];
    CategorySet csobj1 = new CategorySet(strfclA.toString(), strcatA.toString(),
QClass.FORWARD);
    CategorySet csobj2 = new CategorySet(strfclB.toString(), strcatB.toString(), 0);
    categs[0] = csobj1;
    categs[1] = csobj2;

    Q.set_categories(sis_session.getValue(), categs);

    strobject.setValue("AMT-9655");
    if (Q.set_current_node(sis_session.getValue(), strobject) != Q.APIFail)
        if ((ret_set1 = Q.get_traverse_by_meta_category(sis_session.getValue(), 0,
QClass.NOISA)) != Q.APIFail) {
            Q.reset_set(sis_session.getValue(), ret_set1);
            System.out.println("\nTRANSITIVE QUERY FROM 'AMT-9655' (meta-category :
Parts) : \n");
            while ((Q.return_link(sis_session.getValue(), ret_set1, cls, label, cmv))
!= Q.APIFail) {
                System.out.print(" " + cls + " --- " + label + " ---> ");
                print_value(cmv);
            }
        }

    Q.end_query(sis_session.getValue());
    Q.close_connection(sis_session.getValue());
    Q.release_SIS_Session(sis_session.getValue());
}


static public void print_value(CMValue cmv) {
    int type;

     type =cmv.getType();

     switch (type) {
     case CMValue.TYPE_INT    :System.out.println(cmv.getInt()); break;
     case CMValue.TYPE_STRING :System.out.println(cmv.getString()); break;
     case CMValue.TYPE_FLOAT:System.out.println(cmv.getFloat()); break;
     case CMValue.TYPE_NODE :System.out.println(cmv.getString()); break;
     case CMValue.TYPE_TIME :System.out.println(cmv.getTime()); break;
     case CMValue.TYPE_SYSID:System.out.println(cmv.getSysid()); break;
     }
```

```
}

public static void main(String[] args) {
    new Car();
    System.out.println("press any key to exit....");
    try{
        System.in.read();
    }catch(IOException e){
        System.out.println("Cannot Read!!!");
    }
}
}
```

Appendix F – Describing in XML

Below we present the XML document type (DTD) definition for describing the data, as this is extracted by **return_xml_description()** function.

```
<?xml version='1.0' encoding='UTF-8' ?>

<!--Generated by XML Authority-->

<!ENTITY % sys_lnam "sysid? , log_nam? , from_lnam*">

<!ENTITY % primitive "str_value | int_value | flot_value | tim_value">

<!ELEMENT star (target , from? , to? , sysclass_of? , class_of* , instances? ,
subcl_of* , supercl_of* , attribute_from* , attribute_to*)>

<!ELEMENT target (%sys_lnam;)>

<!ELEMENT class_of (%sys_lnam;)>

<!ELEMENT sysclass_of (%sys_lnam;)>

<!ELEMENT subcl_of (%sys_lnam;)>

<!ELEMENT supercl_of (%sys_lnam;)>

<!ELEMENT instances EMPTY>

<!ELEMENT attribute_from (sysid? , log_nam? , ((to_sysid? , to_lnam?) | %primitive;)?
, inher_from? , categ* , attribute_from*)>

<!ELEMENT attribute_to (sysid? , log_nam? , from_sysid? , from_lnam* , inher_from? ,
categ* , attribute_from*)>

<!ELEMENT from (%sys_lnam;)>

<!ELEMENT to ((sysid? , log_nam?) | %primitive;)>

<!ELEMENT sysid (#PCDATA)>

<!ELEMENT log_nam (#PCDATA)>

<!ELEMENT from_lnam (#PCDATA)>

<!ELEMENT to_sysid (#PCDATA)>

<!ELEMENT to_lnam (#PCDATA)>

<!ELEMENT categ (%sys_lnam;)>

<!ELEMENT inher_from (%sys_lnam;)>

<!ELEMENT str_value (#PCDATA)>

<!ELEMENT int_value (#PCDATA)>

<!ELEMENT tim_value (#PCDATA)>

<!ELEMENT flot_value (#PCDATA)>

<!ELEMENT from_sysid (#PCDATA)>
```

Appendix G - Backwards compatibility

In to keep SIS C programmatic interface backward compatible to the previous version (ver. 2.2) we provide a set of include files and libraries that do not use sessions and thus will not take advantage of the multi-thread capabilities of the SIS server.

The basic change of version 2.2 and 2.2.1 is the introduction of sessions. These libraries inforce the use of the functions **init_SIS_API_CS()**, **init_SIS_API_SA()** and **close_SIS_API()** and all C-API functions should be used without the sessionID as first argument.

On WIN32 systems the include files that should be used to compile this code are:

```
sis_kernel/time.h,
cpp_api/cs_defs.h,
cpp_api/identifier.h,
cpp_api/sis_classes.h,
cpp_api/c_wrapper.h.
```

On WIN32 systems the libraries that should be used to link this code are:
Client – Server:

- lib_c_api_cs_2b.lib (The C interface SIS API)
- cpp_api_cs_2b.lib (The C++ interface SIS API)
- lib_sis_kernel_2b.lib (The SIS Kernel)
- lib_time_2b.lib (Time functions library)
- ccomms_2b.lib (Client Communications)
- connection_2b.lib (Used to open a connection to the server)
- libl.lib

Direct Access:

- lib_c_api_sa_2b.lib (The C interface SIS API)
- cpp_api_sa_2b.lib (The C++ interface SIS API)
- lib_sis_kernel_big_2b_cs.lib (The SIS Kernel)
- lib_time_2b.lib (Time functions library)
- ccomms_2b.lib (Client Communications)
- connection_2b.lib (Used to open a connection to the server)
- libl.lib

Notice that before running an application that access directly an SIS base the environment variable `DB_DIR` have to be set (`DB_DIR`: locates the directory where the SIS database exists).

INDEX