SIS - Query Interpreter : An Interactive Program to Use PQI functions

Version 2.0

Costas Dadouris, Martin Dörr, Stavroula Kizlaridou, Nikos Prekas

Institute of Computer Science

Foundation for Research and Technology - Hellas

About This Manual

The Programmatic Query Interface as described in "SIS - Programmatic Query Interface Reference Manual", is a set of functions that can be used in programs to express queries to a SIS base. *qi* is a program that uses PQI to access a SIS base and gives a user the capability to call any of the PQI functions interactively. In this way the user can express queries to the SIS base and test query scenarios which can then be used in programs. *Qi* 's input can be redirected so query macros can be written in files and then be redirected to *qi* for execution. Apart from that, *qi* can be used inside programs as an interpreter for batch execution of queries. It is used as a configuration language for *gain* Graphical Analysis Interface.

When someone starts *qi*, has to define an object existing in the SIS base as *current* and then call any of the query commands to apply on this object. The answer to a query command is stored in the current set of objects. When someone calls a query command, he can specify this query to apply on every object existing in the current set which was created by previous calls to query commands. The answer to a query can be viewed by projecting the contents of the set using one of the projection commands.

PQI functions are renamed to four-character commands and with these names they can by called within *qi* as well as some other help commands.

Table Of Contents

1. How to run *qi*

In directory `bin` the following executables exist: `qserver`, `qi.client` and `qi`.

If you want to run *qi* as a client process according to the client-server model of PQI you should do the following steps:

1. Set environment variable `DB_DIR` to represent the path where the SIS base is located.
2. Run `qserver` *[socket_port]*. The default socket port is 1201.
3. Run `qi.client` *<hostname>* *<socket_port>*.

If you want to run *qi* as an independent process which accesses the SIS base directly, you should do the following steps:

1. Set environment variable `DB_DIR` to represent the path where the SIS base is located.
2. Run `qi`.

If everything is OK you should get the following on your screen:

```
***************************************************************
*                          Q  I                               *
*                                                             *
*  An interactive query interface for the Telos language.     *
*                                                             *
*  Queries apply on a current node that has to be set at      *
*  the beginning of a query session and return the answer      *
*  to the current set. Any other query apply on the           *
*  contents of the current set. Access to other sets can      *
*  be authorized with stor and apon function calls. Sets      *
*  are distinguished with logical names defined by the user.   *
*                                                             *
*  Since the number of the temporary sets that can be used    *
*  is fixed (about 50), you should use the free commands      *
*  that clear a set (or all sets) for reuse.                  *
*                                                             *
*  Type '?' for a list of commands.                           *
*  Type 'help' or 'HELP' for help on a certain command.       *
*                                                             *
***************************************************************

(qi) _
```

You are now in *qi*'s interactive mode.

2. Using *qi*

In the next sections you can find all the *qi* commands, what they stand for and a sort description for each of them. For more information about a command that represends a specific PQI function you can refer to the "SIS - Programmatic Query Interface Reference Manual".

When you start *qi* you have to set an object existing in the SIS base as *current* with the `scn` (Section 2.2) or `scni` (Section 2.2) commands. You can then call any of the query commands. The answer to a query command is stored in the current set of objects. When you call a query command you can specify this query to apply on every object existing in the current set which was created by previous calls to query commands.

You can see the answer to a query, by projecting the contents of the current set, using a projection command from the group of commands, that are presented in the following sections.

2.1. Miscellaneous commands

Use the following commands to end or reset a query session or ask for help on some commands.

`q` (**Q**uit)
Exit from qi.

`rq` (**R**eset **Q**uery)
Resets all global parameters as if the query session has just started. Resets name scope and categories, frees all temporary sets, resets all traverse and filter conditions and disables depth control in traverse queries.

`?` Get a list of groups of commands. You select the number of a group and then you get a list of all the commands of this group.

`help` Similar to the `?` command but after selecting a group, you can ask for some sort help on a certain command simply typing it.

`HELP` Same as help.

2.2. Commands to set global parameters

With the following commands you can set some global parameters that affect the query processing and the behaviour of some query commands.

`rns` (**R**eset **N**ame **S**cope)

Empty the name stack. The next attempt to set a current node will expect to be given the name of a node in the semantic network.

See help on `scn` below.

`pns` (**P**op **N**ame **S**cope)

Pop the name stack.

See help on `scn` below.

`scs` (**S**et **C**urrent **S**et)

Set as *current* the stored set that you are prompted to give.

The contents that the current set had before the execution of the command are lost and now the current set contains the elements of the given set. The name of the set that you are prompted to give must have been declared in the scope. After the execution of the command the set that you have been prompted to give is removed from the scope.

See help on `scn` below.

`scn` (**S**et **C**urrent **N**ode)

You are prompted to enter the logical name of an object which object is then tried to be set as *current* so that next calls of query commands can apply on it.

When first setting a current node you have to enter the name of a node in the semantic network but when a current node has already been set, an attempt to set a current node will expect you to give the name of an attribute of the current node. For this reason a name stack is used and the first item in the stack is a node object and each next item is a link pointing from the previous object. The top item of the stack is the current node and there are commands for manipulating the name stack. The name stack has to be empty before setting as *current* an object that is a node in the semantic network.

See help on `pns` and `rns` below.

`sncn` (**S**et **N**ew **C**urrent **N**ode)

Reset name stack (see help on `rns` above) and prompt to enter the logical name of an object which object is then tried to be set as *current* so that next calls of query commands will apply on it. The object has to be a node in the semantic network.

`scni` (**S**et **C**urrent **N**ode by system **I**dentifier)

You are prompted to enter an integer (the system identifier of an object) and this object then is tried to be set as *current* so that next calls of query commands will apply on.

The name stack is changed properly even if the new current node is a link object.

`sc` (**S**et **C**ategories)

Define a group of categories (link classes) that can be used later by other commands such as `rf` (Section 2.9) , `tc` (Section 2.5) and `tmc` (Section 2.5).

Each category is a link object, so not only the name of the link must be given but also the name of the object it is pointing from. For each category, a direction can be defined. The use of the defined direction is explained in the description of each of the above commands that uses the categories defined with the `sc` command.

`stvc` (**S**et **T**o-**V**alue **C**ondition)

You are prompted to define a logical expression that must be true for the to-value of links traversed forward by `tal` (Section 2.5) command.

Commands used to construct logical expressions are presented in section 2.6.

`sfvc` (**S**et **F**rom-**V**alue **C**ondition)

You are prompted to define a logical expression that must be true for the from-value of links traversed backwards by `tal` (Section 2.5) command.

Commands used to construct logical expressions are presented in section 2.6.

`stlc` (**S**et **T**o-**L**ink **C**ondition)

You are prompted to define a logical expression that must be true for the link traversed backwards by `tal` (Section 2.5) command.

Commands used to construct logical expressions are presented in section 2.6.

`sflc` (**S**et **F**rom-**L**ink **C**ondition)

You are prompted to define a logical expression that must be true for the link traversed forward by `tal` (Section 2.5) command.

Commands used to construct logical expressions are presented in section 2.6.

`sfc` (**S**et **F**ilter **C**ondition)

You are prompted to define a logical expression that is used by `gf` (Section 2.4) command.

Commands used to construct logical expressions are presented in section 2.6.

`spc1` (**S**et **P**rojection **C**ondition 1)

You are prompted to define an expression that defines a query to be executed for each object projected with `rp` (Section 2.9) command.

Commands used to construct logical expressions are presented in section 2.6.

`spc2` (**S**et **P**rojection **C**ondition 2)

You are prompted to define a second expression that defines a query to be executed for each object projected with `rp` (Section 2.9) command.

Commands used to construct logical expressions are presented in section 2.6.

`spc3` (**S**et **P**rojection **C**ondition 3)

You are prompted to define a third expression that defines a query to be executed for each object projected with `rp` (Section 2.9) command.

Commands used to construct logical expressions are presented in section 2.6.

`spc4` (**S**et **P**rojection **C**ondition 4)

You are prompted to define a fourth expression that defines a query to be executed for each object projected with `rp` (Section 2.9) command.

Commands used to construct logical expressions are presented in section 2.6.

`snp` (**S**et **N**umber of **P**rojections)

You are prompted to define the number of projections that you are going to use.

`sdep` (**S**et **D**epth)

Set the depth where transitive queries will stop while traversing. Depth is used in the same way as in graph traversing. If **Depth** < 0 there will be depth control on transitive queries.

2.3. Commands for low level queries

With the following commands you can retrieve low-level information about specific objects in the SIS base.

`gln` (**G**et **L**ogical **N**ame)

You are prompted to enter the system identifier of an object (an integer) and get the logical name of this object.

`gcid` (**G**et **C**lass system **ID**entifier)

You are prompted to enter the logical name of an object which is a node in the semantic network, and get its system identifier.

`glid` (**G**et **L**ink system **ID**entifier)

With this command you can get the system identifier of a link object. Since the logical name of a link object is not unique in the SIS base you have to enter the logical name of the object from which the link is pointing from as well as the logical name of the link itself.

2.4. Commands for simple queries

The following query commands have the same functionality. Each of these apply on the current set. As current set consider the set in which the result of the last query has been stored.

`gf` (**G**et **F**iltered)

Get a set that contains the objects that satisfy the logical expression that was previously set with `sfc` (Section 2.2) is true.

`gm`	(**G**et **M**matched) You are prompted to enter the logical name of a set that contains matching-patterns (Telos_String's). The current set will be searched for any matches. Pattern types are i) left matching (word*), ii) right matching (*word), and iii) any sub-string matching (word) The result is stored in the current set.
`gc`	(**G**et **C**lasses) Get a set that contains the classes that the current object is instance of.
`gac`	(**G**et **A**ll **C**lasses) Get a set that contains all classes that the current object is instance of and also the classes that all superclasses of the current object are instance of.
`gSc`	(**G**et **S**ystem **C**lass) Get a set that contains the system class that the current object is instance of.
`gaSc`	(**G**et **A**ll **S**ystem **C**lasses) Get a set that contains all the system classes that the current object is instance of.
`gi`	(**G**et **I**nstances) Get a set that contains the instances of the current object.
`gai`	(**G**et **A**ll **I**nstances) Get a set that contains the instances of the current object and the instances of all subclasses of the current object too.
`gI`	(**G**et Class Attr**I**butes Of) Get a set that contains the class attributes that are instances of the current category.
`gaI`	(**G**et **A**ll Class Attr**I**butes Of) Get a set that contains the class attributes that are instances of the current category and the instances of all subclasses of the current category too.

`gsc` (**G**et **S**uper**C**lasses)

Get a set that contains the superclasses of the current object.

`gasc` (**G**et **A**ll **S**uper**C**lasses)

Get a set that contains all superclasses (isA closure) of the current object: superclasses of its superclasses etc.

`gSsc` (**G**et all **S**ystem **S**uper**C**lasses)

Get a set that contains all system superclasses of the current object.

`gsbc` (**G**et **SuB**Classes)

Get a set that contains the classes that are isA of the current object.

`gasb` (**G**et **A**ll **SuB**classes)

Get a set that contains all subclasses (inverse isA closure) of the current object: subclasses of its subclasses etc.

`glf` (**G**et **L**ink **F**rom)

Get a set that contains all links that are pointing from the current object.

`gLf` (**G**et c**L**ass Attributes **F**rom)

Get a set that contains all class attributes that are pointing from the current object.

`glfc` (**G**et **L**ink **F**rom by **C**ategory)

Get a set that contains all links pointing from the current node and are instances of the category you are prompted to define. The category is defined not only by the name of the link but also by the name of the object the link is pointing from.

`glfm` (**Get Link From** by **M**etacategory)

Get a set that contains all links pointing from the current object and are of the meta category you are prompted to define. The category is defined not only by the name of the link but also by the name of the object the link is pointing from.

`gilf` (**Get Inherited Link From**)

Get a set that contains all links that are pointing from the current object and the links inherited from all its superclasses.

`giLf` (**Get Inherited** c**L**ass Attributes **From**)

Get a set that contains all class attributes that are pointing from the current object and the class attributes inherited from all its superclasses.

`glt` (**Get Link To**)

Get a set that contains all links that are pointing to the current object.

`gltc` (**Get Link To** by **C**ategory)

Get a set that contains all links pointing to the current object and are instances of the category you are prompted to define. The category is defined not only by the name of the link but also by the name of the object the link is pointing from.

`gltm` (**Get Link To** by **M**etacategory)

Get a set that contains all links pointing to the current object and are of meta category you are prompted to define. The category is defined not only by the name of the link but also by the name of the object the link is pointing from.

`gcf` (**Get Category From**)

Get a set that contains all classes of the links that are pointing from the current object.

`gct` (**Get Category To**)

Get a set that contains all classes of the links that are pointing to the current object.

`gtn` (**G**et **T**o-**N**ode)

Get a set that contains all objects that are pointed by links that are pointing from the current object.

`gtnc` (**G**et **T**o-**N**ode by **C**ategory)

Get a set that contains all objects that are pointed to by links that are pointing from the current object and are instances of the category you are prompted to define. The category is defined not only by the name of the link but also by the name of the object the link is pointing from.

`gtnm` (**G**et **T**o-**N**ode by **M**etacategory)

Get a set that contains all objects that are pointed to by links that are pointing from the current object and are instances of the meta category you are prompted to define. The category is defined not only by the name of the link but also by the name of the object the link is pointing from.

`gfn` (**G**et **F**rom-**N**ode)

Get a set that contains all objects that have links pointing to the current object.

`gfnc` (**G**et **F**rom-**N**ode by **C**ategory)

Get a set that contains all objects that have links pointing to the current object and are instances of the category you are prompted to define. The category is defined not only by the name of the link but also by the name of the object the link is pointing from. In the case of class attributes the returned objects are the computed instance set.

`gfnm` (**G**et **F**rom-**N**ode by **M**etacategory)

Get a set that contains all objects that have links pointing to the current object and are instances of the meta category you are prompted to define. The category is defined not only by the name of the link but also by the name of the object the link is pointing from.

`gfv` (**G**et **F**rom-**V**alue)

Get a set that contains the nodes that the current object points from, supposing the current object is a link.

`gtv` (**G**et **T**o-**V**alue)

Get a set that contains the value (object or primitive value) that the current object points to, supposing the current object is a link.

2.5. Commands for transitive queries

This section describes the commands that can be used to express traverse queries. These queries require some extra parameters for the traverse.

`tal` (General **T**raverse **A**ll **L**inks)

Get a set that contains all the links traversed, starting from the current node and traversing all links according to conditions previously set with : `stvc`, `sfvc` `stlc` and `sflc` (Section 2.2) commands.

Also, you are asked if you want during the traverse, for each node visited, to visit also all its superclasses, all its subclasses, both superclasses or none of these.

Depth control can be achieved with the use of `sdep` (Section 2.2) command.

Cycle detection is performed.

`tc` (**T**raverse by **C**ategory)

Get a set that contains all the links traversed, starting from the current object and traversing all links which are instances of the categories previously defined with the `sc` (Section 2.2) command.

For each category, if the direction was set to FORWARD only the links pointing from each node are checked, if the direction was set to BACKWARD only the links pointing to each node are checked and if direction was set to BOTH DIR all links are checked.

Depth control can be achieved with the use of `sdep` (Section 2.2) command.

Cycle detection is performed.

`tmc` (Traverse by MetaCategory)

Get a set that contains all the links traversed, starting from the current object and traversing all links which are instances of some instance of the categories previously defined with the `sc` (Section 2.2) command. That is traversing links that are of some Metacategory defined with the `sc` command.

For each category, if the direction was set to FORWARD only the links pointing from the node are checked, if the direction was set to BACKWARD only the links pointing to the node are checked and if direction was set to BOTH DIR all links are checked.

Depth control can be achieved with the use of `sdep` (Section 2.2) command.

Cycle detection is performed.

2.6. Commands to construct logical expressions

This section describes the commands that can be used to construct logical expressions.

The commands that require a logical expression prompt you to enter some of the commands that are presented in this section. While constructing the logical expression, you are prompted to enter commands of a specific group of commands which are: LOGICAL, OBJECT, SET_EXPR and INT_VAL. If you type "?" you get a list of the commands you can use.

The logical expressions are constructed in prefix notation so you have to enter the function name first and then its arguments.

2.6.1. LOGICAL group of commands

When you are prompted to enter a LOGICAL command you can use any of the following commands that return a logical value.

`succ` Equivalent to TRUE in logical expressions. Always succeeds.

`fail` Equivalent to FALSE in logical expressions. Always fails.

`and` Logical AND between the next two logical expression that you will be prompted to enter.

`or` Logical OR between the next two logical expression that you will be prompted to enter.

`not`	Logical negation of the next logical expression that you will be prompted to enter.
`blng`	(BeLoNGs) True if the object that you will by prompted to describe (OBJECT group of commands) exists in the set of objects that you will be also prompted to describe (SET_EXPR group of commands).
`mtch`	(MaTCH) True if there is an object name in the set of objects that you will be prompted to describe (SET_EXPR group of commands) matches (partialy) any pattern in the set of Telos_Strings that you will also be prompted to describe (SET_EXPR group of commands).
`eq`	(EQual) You will be prompted to enter two INT_VAL commands. True if the first of them is equal to the second one.
`gt`	(Greater Than) You will be prompted to enter two INT_VAL commands. True if the first of them is greater than the second one.
`gte`	(Greater Than or Equal) You will be prompted to enter two INT_VAL commands. True if the first of them is greater than or equal to the second one.
`lt`	(Less Than) You will be prompted to enter two INT_VAL commands. True if the first of them is less than the second one.
`lte`	(Less Than or Equal) You will be prompted to enter two INT_VAL commands. True if the first of them is less than or equal to the second one.

`bfr` (BeFoRe)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). True if at least one of the time values pointed by the links of the first set ends before the time value included in the second set starts

`aftr` (AFTeR)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). This operand is the inverse operand of bfr. True if at least one of the time values pointed by the links of the first set starts after the time value included in the second set finishes.

`tmeq` (TiMe EQual)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). True at least one of the time values pointed by the links of the first set starts where the time value (tm2) included in the second set starts and finishes where tm2 finishes.

`mts` (MeeTS)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). True at least one of the time values pointed by the links of the first set ends where the time value included in the second set starts `mtb` (MeT By)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). This operand is the inverse operand of mts. True if at least one of the time values pointed by the links of the first set starts where the time value included in the second set ends.

`ovl` (OVerLaps)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). True if at least one of the time values pointed by the links of the first set starts before the ending of the time value included in the second set (tm2) and ends after the begining of tm2.

`ovlb` (OVerLaPped By)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). This operand is the inverse operand of ovl. True if at least one of the time values pointed by the links of the first set starts before the ending of the time value included in the second set (tm2) and ends after the begining of tm2.

`drng` (DuRiNG)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). True if at least one of the time values pointed by the links of the first set starts after the begining of the time value included in the second set (tm2) and ends before the ending of tm2.

`ctns` (ConTaiNS)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). This operand is the inverse operand of drng. True if at least one of the time values pointed by the links of the first set starts before the begining of the time value included in the second set (tm2) and ends after the ending of tm2.

`srts` (StaRTS)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)). True if at least one of the time values pointed by the links of the first set starts where the time value included in the second set (tm2) starts and ends before the ending of tm2.

`srtb` (StaRTed By)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) This operand is the inverse operand of srts. True if at least one of the time values pointed by the links of the first set starts where the time value included in the second set (tm2) starts and ends before the ending of tm2.

`fshs` (FiniSHeS)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set starts after the begining of the time value included in the second set (tm2) and ends where tm2 ends.

`fshb` (FiniSHed By)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) This operand is the inverse operand of fshs. True if at least one of the time values pointed by the links of the first set starts before the begining of the time value included in the second set (tm2) and ends where tm2 ends.

`cbeq` (Can Be EQual)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *There exist* $t1 \in tm1, t2 \in tm2: t1 = t2$ This means that there exists at least one element into time interval (value) tm1 that is equal to at least one element that belongs to time interval tm2. Lets remind here that Telos adopts an interval-based time model.

`cblt` (Can Be Less Than)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *There exist* $t1 \in tm1, t2 \in tm2: t1 < t2$ This means that there exists at least one element into time interval (value) tm1 that is less than at least one element that belongs to time interval tm2. Lets remind here that Telos adopts an interval-based time model.

`cble` (Can Be Less or Equal)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *There exist t1* $\in$ *tm1, t2* $\in$ *tm2: t1 <= t2* This means that there exists at least one element into time interval (value) tm1 that is less than or equal to at least one element that belongs to time interval tm2. Lets remind here that Telos adopts an interval-based time model.

`cbgt` (Can Be Greater Than)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *There exist t1* $\in$ *tm1, t2* $\in$ *tm2: t1 > t2* This means that there exists at least one element into time interval (value) tm1 that is greater than at least one element that belongs to time interval tm2. Lets remind here that Telos adopts an interval-based time model.

`cbge` (Can Be Greater or Equal)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *There exist t1* $\in$ *tm1, t2* $\in$ *tm2: t1 >= t2* This means that there exists at least one element into time interval (value) tm1 that is greater than or equal to at least one element that belongs to time interval tm2. Lets remind here that Telos adopts an interval-based time model.

`mbeq` (Must Be EQual)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *Forevery* $t1 \in tm1$, $t2 \in tm2$: $t1 = t2$ This means that every element into time interval (time value) tm1 must be equal to every element that belongs to time interval (time value) tm2. Lets remind here that Telos adopts an interval-based time model.

`mblt` (Must Be Less Than)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *Forevery* $t1 \in tm1$, $t2 \in tm2$: $t1 < t2$ This means that every element into time interval (time value) tm1 must be less that every element that belongs to time interval (time value) tm2. Lets remind here that Telos adopts an interval-based time model.

`mble` (Must Be Less or Equal)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *Forevery* $t1 \in tm1$, $t2 \in tm2$: $t1 <= t2$ This means that every element into time interval (time value) tm1 must be less than or equal to every element that belongs to time interval (time value) tm2. Lets remind here that Telos adopts an interval-based time model.

`mbgt` (Must Be Greater Than)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *Forevery* $t1 \in tm1$, $t2 \in tm2$: $t1 > t2$ This means that every element into time interval (time value) tm1 must be greater that every element that belongs to time interval (time value) tm2. Lets remind here that Telos adopts an interval-based time model.

`mbge` (Must Be Greater or Equal)

You will be prompted to enter two SET_EXPR commands. The first one must be a set that contains links that points to time primitives (their to value is time value). The second one must be a set that contains a specific time value, which is used as the condition for the comparison (this set is created using sppr (Section 2.7)) True if at least one of the time values pointed by the links of the first set (lets say tm1) fullfills the following condition in comparison to the time value included in the second set (tm2) : *Forevery* $t1 \in tm1$, $t2 \in tm2$: $t1 >= t2$ This means that for every element into time interval (time value) tm1 must be greater than or equal to every element that belongs to time interval (time value) tm2. Lets remind here that Telos adopts an interval-based time model.

2.6.2. INT_VAL group of commands

When you are prompted to enter an INT_VAL command you can use any of the following commands that represent an integer value.

`val` With this command you can represent a specific integer that you will be prompted to enter.

`card` With this command you can represent an integer which is the cardinality of the set that you will be prompted to enter (SET_EXPR group of commands).

2.6.3. OBJECT group of commands

When you are prompted to enter an OBJECT command you can use any of the following commands that describe an object.

`sys` With this command you can represent an object by its system identifier which you are prompted to enter. If you enter system identifier 0 means the object that the logical expressions is evaluated for.

`node` With this command you can represent an object which is a node in the semantic network in the SIS base, by its logical name which you are prompted to enter.

`link` With this command you can represent an object which is a link in the semantic network in the SIS base, with the combination of the logical name of the object it is pointing from and its logical name which you are prompted to enter.

2.6.4. SET_EXPR group of commands

When you are prompted to enter a SET_EXPR command you can use any of the following commands that are describing a set of objects.

If you want query commands to apply on the object that the logical expression is evaluated for, you should use `set` command of this group with argument "0".

`set` With this command you can describe a set by its unique identifier which you are prompted to enter. If you enter system identifier 0 represents the object that the logical expressions is evaluated for.

`su` The set union of the set you are prompted to enter and the current set.

`si` The set intersection of the set you are prompted to enter and the current set.

`sd` The set difference of the set you are prompted to enter and the current set.

`gc` The answer set of `gc` (Section 2.4) command applied on the current set.

`gac` The answer set of `gac` (Section 2.4) command applied on the current set.

`gSc`	The answer set of `gSc` (Section 2.4) command applied on the current set.
`gaSc`	The answer set of `gaSc` (Section 2.4) command applied on the current set.
`gi`	The answer set of `gi` (Section 2.4) command applied on the current set.
`gai`	The answer set of `gai` (Section 2.4) command applied on the current set.
`gI`	The answer set of `gI` (Section 2.4) command applied on the current set.
`gaI`	The answer set of `gaI` (Section 2.4) command applied on the current set.
`gsc`	The answer set of `gsc` (Section 2.4) command applied on the current set.
`gasc`	The answer set of `gasc` (Section 2.4) command applied on the current set.
`gSsc`	The answer set of `gSsc` (Section 2.4) command applied on the current set.
`gsbc`	The answer set of `gsbc` (Section 2.4) command applied on the current set.
`gasb`	The answer set of `gasb` (Section 2.4) command applied on the current set.
`glf`	The answer set of `glf` (Section 2.4) command applied on the current set.
`gLf`	The answer set of `gLf` (Section 2.4) command applied on the current set.
`glt`	The answer set of `glt` (Section 2.4) command applied on the current set.
`gfv`	The answer set of `gfv` (Section 2.4) command applied on the current set.

`gtv`	The answer set of `gtv` (Section 2.4) command applied on the current set.
`glfc`	The answer set of `glfc` (Section 2.4) command applied on the current set.
`gltc`	The answer set of `gltc` (Section 2.4) command applied on the current set.

2.7. Commands to manipulate sets of objects

This section describes the commands that can be used to manipulate sets of objects.

`fs` (**F**ree **S**et)

Free the set with identifier the one you are prompted to enter, for later reuse.

`sgn` (**S**et **G**et **N**ew)

Get a new empty temporary set which will be the current set.

`sgc` (**S**et **G**et **C**ardinality)

Get the number of the objects contained in the current set.

`su` (**S**et **U**nion)

You are prompted to give the logical name of a set, then a set union operation is performed between the current set and the given one. The result is stored in the current set.

`si` (**S**et **I**ntersection)

You are prompted to give the logical name of a set, then a set intersection operation is performed between the current set and the given one. The result is stored in the current set.

`sd` (**S**et **D**ifference)

You are prompted to give the logical name of a set, then a set difference operation is performed between the current set and the given one. The result is stored in the current set.

`sdis` (Set **DIS**isjoint)

You are prompted to give the logical name of a set, then a set disjoint operation is performed between the current set and the given one. The result is stored in the current set.

`seq` (Set **E**qual)

You are prompted to give the logical name of a set, then a set equal operation is performed between the current set and the given one. The result is stored in the current set.

`sput` (Set **PUT**)

Put the current node in the current set.

`sppr` (Set **PUT PRimitive**)

You are prompted to specify the type of the primitive you want to put into the set (Telos_Integer, Telos_Real, Telos_Time, Telos_String) and then enter the value of the specified primitive. In order to use this command you have to allocate a new set in a previous step (by calling sgn (Section 2.7)).

`sdel` (Set **DEL**ete)

Delete the current node from the current set.

`smo` (Set **M**ember **O**f)

Check if current node exists in the current set.

2.8. Branching functions

Branching (changing the *current set*) in the sequence of execution of query functions can be achieved with the following functions:

`stor` (**STORe**)

You are prompted to give the logical name of a set, then you store the contents of the current set to this set. This temporary set can be used later in set operations or for branching in the execution of a sequence of query function calls. A virtual copy is created, which becomes real after the change of the current set.

When a temporary set is stored with this function, it becomes write protected, that is it can not be used in functions that update the contents of a set.

`apon` (**AP**ply **ON** Set/Object)

Set as current set the temporary set previously stored with a given logical name. If the logical name is ‘‘cn’’ then next call to a query function will apply on current node.

`crsc` (**CR**eate **SC**ope)

A scope mechanism is supported for the logical names of the stored temporary sets. The logical names of the sets are unique in a scope which is created with this `crsc()` function. Scopes can be nested.

`rtrn` (**ReTuRN** Scope)

End a scope, previously created with the `crsc()` function (above). After return, only the current set is known to the external scope. All other temporary sets stored in the scope are deleted.

`info` (**INFO**rmation)

Print scope information

2.9. Commands to project objects

This section describes the commands that can be used to project the objects that exist in a specific set. After calling any of these commands you are prompted to enter the identifier of the set whose objects you want to project. Then you are asked if you want output on screen (enter "`s`") or redirected in a file (enter "`f`"). If you select "`f`" you are prompted to enter the name of the file where the output is going to be redirected.

The difference among the following commands is in the amount of information presented for each object in the set.

`rn` (**R**eturn **N**odes)

Print out a list of the logical names of the objects contained in the current set.

`rfn` (**R**eturn **F**ull **N**odes)

Print out a list of the system identifier, the logical name and the system class of the objects contained in the current set.

`ren` (**R**eturn **E**dge **N**odes)

Print out a list of the logical names of the objects, where the traverse query stopped because of the depth control, but the traversing should continue from these objects.

`rprs` (**R**eturn **PRS** values)

Print out a list of the objects contained in the current set. If the object is a node or a class the logical name of it, is printed, if it is a primitive value (integer, real or string) the actual value of it is printed. Usually sets contain only objects but some commands, like `gtv` (Section 2.4) may create a set that contains primitive values as well. In this case the `rn` (Same section) and `rfn` (Same section) will return only the objects contained in the set.

`rc` (**R**eturn **C**ategories)

Supposing that objects contained in the current set are links then print out the logical name of the class the object is pointing from and the logical name of the object.

`rci` (**R**eturn **C**ategories **I**dentifier)

Supposing that objects contained in the current set are links then print out the logical name of the class the object is pointing from, the system identifier and the logical name of the object.

`rl` (**R**eturn **L**inks)

Supposing that objects contained in the current set are links then print out the logical name of the class the object is pointing from, the logical name of the object and the object/value the object is pointing to.

`rli` (**R**eturn **L**ink **I**dentifiers)

Supposing that objects contained in the current set are links then print out the logical name of the object and also the system identifier of the object it is pointing from, the system identifier of the link object and the system identifier of the object or primitive value it is pointing to. If the link object is pointing to a primitive value, system identifier 0 is printed.

`rfl` (**R**eturn **F**ull **L**inks)

Supposing that objects contained in the current set are links then for each of them print out :

- A flag "F" or "B" denoting that the link was traversed forward of backward by a traverse query. If link object was not retrieved by a traverse query "F" is printed.
- The logical name of the class the object is pointing from.
- The logical name of the link object.
- In parenthesis the category of the link object.
- A flag "UNIQUE" or "NOT UNIQUE" denoting that the category is or not unique for this link object, since link object may be instance of more than one category.
- The object or primitive value the link object is pointing to.

`risa` (**R**eturn **ISA** related objects)

Return pairs of objects A and B, existing in the current set and A isA B.

`rins` (**R**eturn **INS**tance pairs)

Return pairs of objects A and B, existing in the current set and A is instance of B.

`rf` (**R**eturn **F**ields)

Print a list of tuples where the first field is the logical name of an object existing in the current set and next fields vary according to categories previously set with `sc` (Section 2.2) command. For each category, there is a field in the tuple, and the value of the field is the to-value of the link of this category if the category was set FORWARD, or the from-value of the link of this category, if the category was set BACKWARD.

For each object in the current set, its links are checked to be instances of each category or of some superclass or subclass of this category.

`rhl` (**R**eturn **H**idden **L**inks)

Suppose that objects contained in the current set are links then print out the logical name of the class the object is pointing from, the logical name of the object and the object/value the object is pointing to.

If the from-object is linked with an inverse link of category previously set with `sc` (Section 2.2) command 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.

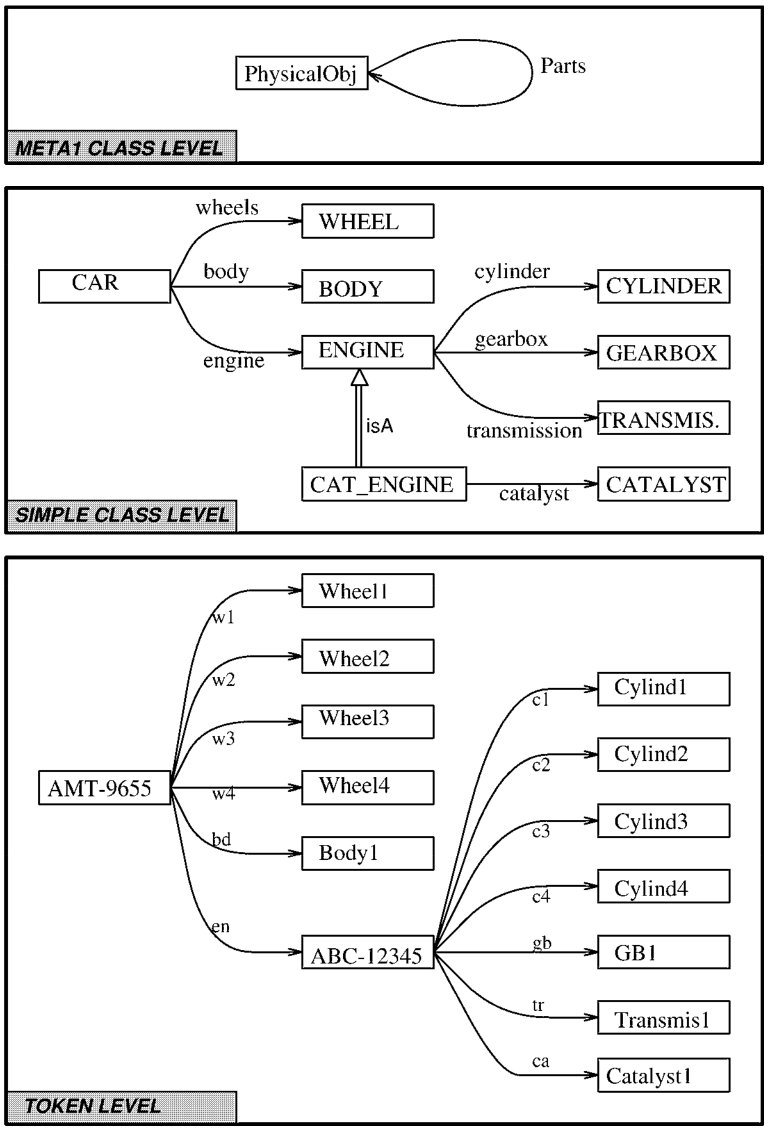
`rp` (**R**eturn **P**rojection)

Print a list of tuples where first field is the logical name of an object existing in the current set and next fields contain the logical names of objects in the answer set that is retrieved by applying zero to four predefined queries to this object.

The number of predefined queries can be set with the `snp` (Section 2.2) command. The predefined queries can be previously defined with the `spc` (Section 2.2) command.

3. Appendix A - An example

Let's see an example of the use of *qi* for querying a SIS base that contains the description of the following model:



A SIS 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.

In the following examples, highlighted is what we type.

We can find instances of class *PhysicalObj* by:

```
(qi) scn

    SET CURRENT NODE
    Enter logical name of node : PhysicalObj

    CURRENT NODE IS NOW: PhysicalObj (SYSID 33)

(qi) gi

    GET INSTANCES


(qi) rn

    RETURN NODES

    SET No  1 CONTAINS :
-------------------------------
  No |  Object name
-------------------------------
   1 | CAR
   2 | WHEEL
   3 | BODY
   4 | ENGINE
   5 | CYLINDER
   6 | GEARBOX
   7 | TRANSMIS
   8 | CAT_ENGINE
   9 | CATALYST
-------------------------------
```

And then all instances of these objects by:

```
(qi) gi

    GET INSTANCES
```

```
(qi) rn

    RETURN NODES

    SET No  2 CONTAINS :
-------------------------------
  No |  Object name
-------------------------------
   1 | AMT-9655
   2 | Wheel1
   3 | Wheel2
   4 | Wheel3
   5 | Wheel4
   6 | Body1
   7 | ABC-12345
   8 | Cylind1
   9 | Cylind2
  10 | Cylind3
  11 | Cylind4
  12 | GB1
  13 | Transmis1
  14 | Catalyst1
-------------------------------
```

After this, we store the results from the previous query and then we ask for all wheels of car *AMT-9655* by:

```
(qi) stor

    Enter logical name of set  : 2


(q ) sncn

    SET CURRENT NODE
    Enter logical name of node : AMT-9655

    CURRENT NODE IS NOW: AMT-9655 (SYSID 51)


(qi) glfc

    GET LINK FROM BY CATEGORY

    Enter Class and category          : CAR wheels


(qi) rl

    RETURN LINK
    SET No  1 CONTAINS :
----------------------------------------------------------------------------
  No |                Class |      Label Name |(Type ) To value
----------------------------------------------------------------------------
   1 |             AMT-9655 |              w1 |(NODE ) Wheel1 (id:53)
   2 |             AMT-9655 |              w2 |(NODE ) Wheel2 (id:55)
   3 |             AMT-9655 |              w3 |(NODE ) Wheel3 (id:57)
   4 |             AMT-9655 |              w4 |(NODE ) Wheel4 (id:59)
----------------------------------------------------------------------------
```

Set 2 contains the instances of instances of metaclass *PhysicalObj* and we can filter this set to get only those object that have more that 5 attributes (there are 5 link objects pointing from them) by:

```
(qi) sfc

    SET FILTER CONDITION
    (LOGICAL  ) : gt
        (INT_VAL  ) : card
            (SET EXPR) : glf
                (SET EXPR) : set
                    (SET NAME ) : 0
        (INT_VAL  ) : val
            (INTEGER  ) : 5

(qi) apon

    Enter logical name of set  : 2

(qi) gf

    GET FILTERED
```

We can see the objects selected after filtering by:

```
(qi) rn

    RETURN NODES

    SET No  2 CONTAINS :
-------------------------------
  No |  Object name
-------------------------------
   1 | AMT-9655
   2 | ABC-12345
-------------------------------
```

But we can ask to see not only the name of these objects but also the names of their attributes.

```
(qi) spc1

    SET PROJECTION CONDITION
    (SET EXPR) : gtv
        (SET EXPR) : glf
            (SET EXPR) : set
                (INTEGER  ) : 0

(qi) rp

    RETURN PROJECTION

    SET No  2 CONTAINS :
----------------------------------------------------------------------------
| No |       NODE        |   node,...,node | node,...,node | ... | node,...,node>
----------------------------------------------------------------------------
|  1 | AMT-9655 (id:51)  | Wheel1(id:53), Wheel2(id:55), Wheel3(id:57) ,  |  |  |
|    |                   | Wheel4(id:59), Body1(id:61), ABC-12345(id:63)  |  |  |
|  2 | ABC-12345(id:63)  | Cylind1(id:65), Cylind2(id:67), Cylind3(id:69),|  |  |
|    |                   | Cylind4(id:71), GB1(id:73), Transmis1(id:75),  |  |  |
|    |                   | Catalyst1(id:77)                               |  |  |
----------------------------------------------------------------------------
```

Finally, we can ask for all parts of a car by traversing all links of category <PhysicalObj, Parts>, starting from node CAR.

```
(qi) sncn

    SET NEW CURRENT NODE
    Enter logical name or sysid of node : CAR

    CURRENT NODE IS NOW: CAR (SYSID 35)


(qi) sc

    SET CATEGORIES
    Enter category[0]    (From_class  Category)
    ('end end' to stop) : PhysicalObj Parts
    Enter traverse direction 1-FW  2-BW  3-Both : 1
    Enter category[1]    (From_class  Category)
    ('end end' to stop) : end end


(qi) tc

    TRAVERSE BY CATEGORY
    For each node, visit also
    (1)superclasses(2)subclasses(3)both(0)none : 0
```

We could express the same query using the conditionally traverse query:

```
(qi) sflc

    SET FROM LINK CONDITION
    (LOGICAL  ) : blng
        (OBJECT   ) : link
            (LINK NAME) : PhysicalObj Parts
        (SET EXPR) : gc
            (SET EXPR) : set
                (SET NAME ) : 0

(qi) stvc

    SET TO VALUE CONDITION
    (LOGICAL  ) : succ

(qi) stlc

    SET TO LINK CONDITION
    (LOGICAL  ) : fail

(qi) tal

    GENERAL TRAVERSE BY ALL LINKS
    For each node, visit also
    (1)superclasses(2)subclasses(3)both(0)none : 0
```

The answer to the last two queries is the same and contains the following link objects:

```
(qi) rl

    RETURN LINK

    SET No  1 CONTAINS :
----------------------------------------------------------------------------
  No |                Class |      Label Name |(Type ) To value
----------------------------------------------------------------------------
   1 |                  CAR |         wheels |(NODE ) WHEEL (id:37)
   2 |                  CAR |           body |(NODE ) BODY (id:39)
   3 |                  CAR |         engine |(NODE ) ENGINE (id:41)
   4 |               ENGINE |       cylinder |(NODE ) CYLINDER (id:43)
   5 |               ENGINE |        gearbox |(NODE ) GEARBOX (id:45)
   6 |               ENGINE |   transmission |(NODE ) TRANSMISSION (id:47)
----------------------------------------------------------------------------
```

4. Appendix B - User defined queries in the SIB Static Analyser

As mentioned above, *qi* commands can be invoked inside a program, that is non-interactively. Here we shall present an example of non-interactive use of *qi*: the queries written for the SIB Static Analyser (see also: SIB Static Analyser Graphical Analysis Intrface - User's Manual). The queries offered by the SIS Graphical Analysis Interface can be defined by the user and stored in the SIS base. These queries consist of *qi* commands that are executed at run-time in batch mode.

In order to write queries, the user has to include the following segments of TELOS code in the model he uses.

First, there must be a *Model* object with a category *uiMenus*:

```
TELL Individual Model in M1_Class with
    attribute
        ...
        uiMenus : MenuDescription
end
```

MenuDescription contains textual query submenus (*queryMenu*), graphical query submenus (*viewMenu*) and retrieval menus (*retrievalMenu*):

```
TELL Individual MenuDescription in S_Class with
    attribute
        queryMenu: SubMenu;
        viewMenu : SubMenu;
        retrievalMenu : RetrievalMenu
end
```

SubMenu contains other submenus (thus, it is recursively defined) and queries that will be executed by the user (*commands*):

```
TELL Individual SubMenu in S_Class with
    attribute
        subMenu : SubMenu;
        commands: QueryMacro
end SubMenu
```

The recursive definition of the *SubMenu* class allows to the query menus of the SIS Graphical Analysis Interface to have submenus at any depth.

RetrievalMenu is a subclass of *SubMenu* with one more category:

```
TELL Individual RetrievalMenu in S_Class isA SubMenu with
    attribute
        return_conditions: QueryMacro
end
```

This additional *QueryMacro* allows presenting results in formatted mode.

QueryMacro objects are the actual queries:

```
TELL Individual QueryMacro in S_Class with
    attribute
        code         : Telos_String;
        inputType    : Telos_String;
        outputType   : Telos_String;
        iterator     : Telos_String;
        outputHeader : Telos_String
end QueryMacro
```

Category *code* represents the actual qi commands that will be executed, *inputType* and *outputType* are used for type checking, *iterator* is the return function used to retrieve parts of the answer set that are of interest to the user and *outputHeader* concerns the presentation of the results on the screen. All of these will be explained in detail later.

All identifiers presented here are keywords used by the system to construct the query menus at startup. The entry point is the logical name of the instance of *Model*, given on the command line of the SIS Graphical Analysis Interface. Consequently, this code has to be written exactly in the way presented here. If some mistake occurs (e.g. a link does not exist or does not have the correct name), part of the query menus (or all of them) will not appear on screen when the user interface is created and the user will be appropriately warned.

Specifically, the keywords necessary for creation of the menus are the following: **Model**, **uiMenus**, **MenuDescription**, **queryMenu**, **viewMenu**, **SubMenu**, **subMenu**, **commands**, **QueryMacro**, **code**,**inputType**, **outputType**, **iterator**, **outputHeader**.

4.1. Writing Queries - The CooL example

Queries presented under the *Views* and *Queries* menus of the SIS Graphical Analysis Interface are stored in the SIS base. The example presented here is based on the model for the CooL language; for each model used the user can define a different set of queries.
CooL (our model) has the following menu description:

```
{*************************************************************
**      The CooL Model                                       *
**************************************************************}

TELL IndividualClass CooL in S_Class, ImplementationModel with
     ...
    uiMenus
        : CooLMenus
end CooL
```

CooL is also the command line argument for the SIS Graphical Analysis Interface. The *CooLMenus* object contains the description of the menus.

```
TELL Individual CooLMenus in Token, MenuDescription with
    queryMenu
        Queries : CooLTextQuery
    viewMenu
        Views   : CooLGraphQuery
    retrievalMenu
        (Faceted_Retrieval) : CooLRetrievalQuery
end CooLMenus
```

The names of the links under the categories *queryMenu* and *viewMenu* (in this case the labels *Queries* and *Views*) are displayed on screen as button names of the respective menus; the user thus has the possibility of configuring the appearance of the menus. The name of the link under the category *retrievalMenu* (in this case the label *Faceted_Retrieval*)appears on the retrieval card of the SIS Graphical Analysis Interface.

The *CooLTextQuery* object describes textual queries and the *CooLGraphQuery* object describes the graphical queries. Let us now examine the contents of these objects:

```
TELL Individual CooLTextQuery in Token, SubMenu with
    subMenu
        (Classification_Queries) : ClassiQuery;
        (File_Queries)           : CooLFileQuery;
        (Variable_Queries)       : CooLVariableQuery;
        (Procedure_Queries)      : CooLProcedureQuery;
        (Method_Queries)         : CooLMethodQuery;
        (Refer_Queries)          : CooLReferQuery;
        (Class_Queries)          : CooLClassQuery
end
```

CooLTextQuery consists of some submenus while *CooLGraphQuery* below has no sub-menu and consists of some commands (mixing the two categories in one object is also possible).

```
TELL Individual CooLGraphQuery in Token, SubMenu with
    commands
        (Subtype_Tree)       : CooLSubtypeTree;
        (Supertype_Tree)     : CooLSupertypeTree;
        (Call_Tree)          : CooLCallTree;
        (Called_By_Tree)     : CooLCallByTree;
        (Reference_Tree)     : CooLreferTree;
        (Referenced_By_Tree) : CooLreferByTree
end CooLGraphQuery
```

The labels of *subMenu* and *commands* links are used as labels of the menu buttons in the user interface (only '_' characters are omitted). In this way, the user can set the names of the options in the menu.

Each individual query is a *QueryMacro* object:

```
{------------------------------------------------------------
    Display all the files that are referenced by the given
    file directly or indirectly. The query target must be
    of kind CooLFile.
-------------------------------------------------------------}
TELL Individual CooLreferTree in Token, QueryMacro with
    code
        (1) : "sc CooLFile references 1 end end";
        (2) : "tc 0"
    inputType
        : "CooLFileReference"
    iterator
        : "flin"
end CooLreferTree
```

The *qi* commands are read from the SIS base by the SIS Graphical Analysis Interface and stored internally. When the query is selected by the user, these commands are passed to the *qi* for execution. Here are some remarks about this code segment:

- The order of execution of the *qi* commands is marked by their respective integer label. The first command should have a label "1" and no two labels can be identical.
- The *inputType* field shows that an object of type *CooLFileReference* must be Query Target so that the query can be executed correctly. The SIS Graphical Analysis Interface stores this type internally for every query and performs a check on the current Query Target each time the query menus are "pulled down". All queries having an *inputType* different than the classes where Query Target belongs become inactive and the user cannot execute them. If the *inputType* of the query matches one of the Query Target's classes, two actions are performed when the query is selected: a *scn* command is executed first with the current Query Target as parameter and then all *qi* commands stored in the *QueryMacro* object are executed. Thus, the *inputType* field implicitly states that the query needs an external target found in the Query Target area of the SIS Graphical Analysis Interface. In case this target does not have to belong to a specified type, the inputType field should simply be: *"ALL_TYPES"*. If the query does not demand an external target at all, the *inputType* category should not be instantiated.

Another remark concerns the *iterator* category:

- Strings instantiated under the category *iterator* are used by the SIS Graphical Analysis Interface as function identifiers in order to retrieve the contents of the answer set which is created as answer to the query. It is an iterator that selects parts of the answer set, in order to present them to the user.

  Graphical queries like the one presented can use one of the following iterators:
  isa (corresponding to a return_isa command),
  Isa (corresponding to return_inverse_isa),
  flin (corresponding to return_full_link),
  flIs (corresponding to return_full_link_inverse_isa),
  hidl (corresponding to return_hidden_links),
  inst (corresponding to return_instances),
  Inst (corresponding to return_inverse_instances)
  and *sIn* (corresponding to return_inverse_isa_inverse_instances).

Textual queries can use one of the three following qi commands as iterators:
rn (return_nodes),
rf (return_fields) and
rp (return_projection).

With the two first remarks it becomes clear that each query can be associated with a particular class of objects and also with a specific return function.

The submenus of the textual queries look similar:

```
{**************************************************************
    These are the queries concerning the objects that
    are of kind CooLVariable. None of them requires
    a query target.
**************************************************************}
TELL Individual CooLVariableQuery in Token, SubMenu with
    commands
        (List_All_Variables) : CooLListVar
end CooLVariableQuery
```

A query from this submenu is the following:

```
{-------------------------------------------------------------
    Display all objects of kind CooLVariable (namely
    parameters and global variables - local variables are
    not part of the analysis).
-------------------------------------------------------------}
TELL Individual CooLListVar in Token, QueryMacro with
    code
        (1) : "scn CooLVariable";
        (2) : "gai";
        (3) : "sc CooLObject type 1 CooLFile variables 2 end end"
    outputHeader
        : "Variable Type File"
end CooLListVar
```

There are some more features in this query that need explanation:

- The *outputHeader* is a string that contains labels to be displayed above the text window of the SIS Graphical Analysis Interface when the text output is in formatted state. The presence of the *outputHeader* link indicates to the SIS Graphical Analysis Interface that this query can have formatted output, that is, additional information for each object in the final answer set (for each variable, its type and the file it is defined in will be displayed). The part of the query that is responsible for retrieving that information is the last command (*sc..., Set Categories*), where the user indicates the categories that must be set in order to get the relevant information. A command like this should be the last one given in the *QueryMacro* object, so that additional processing can be performed after execution of the query.
- All queries are expected to leave their final answer in current set of the *qi*. This is where the SIS Graphical Analysis Interface expects to find the answers to the queries that are requested by the user. In most cases it is trivial, since all qi functions leave their results in the current set.

To simplify the writing of queries in the database (in the form presented above) some additional processing is performed before executing the *qi* commands. We internally reset the name scope: thus, the user does not have to explicitly invoke the `rns` command in the *QueryMacro* object. Apart from that, each query is executed in an independent scope, so that set names can be considered unique for each *QueryMacro*.

After executing a query such as the one presented here, the SIS Graphical Analysis Interface displays the query answer set to the user.

The *CooLRetrievalQuery* object describes the retrieval queries. Let us now examine the contents of this object:

```
TELL Individual (CooLRetrievalQuery) in Token, RetrievalMenu with
     commands
        (1_Abstraction) : CooLAbstractionQuery;
        (2_Operation)   : CooLOperationQuery;
        (3_OperatesOn)  : CooLOperatesOnQuery
end (CooLRetrievalQuery)
```

The labels of the links instantiated under *commands*, must begin with a number and a '_' character. This number determines the order of appearance of the respective entry in the retrieval card of the SIS Graphical Analysis Interface. The rest of the label appears on the left of the respective field in the retrieval card.

Every time the user wants to perform a retrieval query, each sub-query instantiated under the *commands* category (that is, each *QueryMacro*) is performed on the query targets that the user has provided in the respective field. The results of each execution are then combined using the logical operators that the user has provided (logical AND, logical OR). Finally, all these partial results are combined (using the logical AND operator) to form the final query answer. Every *QueryMacro* corresponds to a field of the retrieval card, that is, the same *QueryMacro* code is executed for every query target entered in the same field of the retrieval card.

Because each of these sub-queries might be executed more than once, the condition functions of the *qi* should be written in a separate object, so that they can be called only once at the end of the execution.

Each individual query is a *QueryMacro* object :

```
TELL Individual CooLAbstractionQuery in Token, QueryMacro with
    code
        (1) : "gai";
        (2) : "stor 1"
        (3) : "apon cn"
        (4) : "gfnc Facet synonym";
        (5) : "stor 2"
        (6) : "gai";
        (7) : "scs 1"
        (8) : "su 2"
        (9) : "stor 1"
        (10) : "gfnc CooLObjectType supertype";
        (11) : "su 1";
        (12) : "fs 1";
        (13) : "fs 2";
   inputType
        : "ALL_TYPES"
end
```

Another useful remark about queries in general is the following:

- If a *qi* command is too long to fit in one string of category *code*, it can be divided in to parts, that is, in to subsequent strings. A special character '@' is used as the last of the string in order to denote that the current string is continued into the next one. Thus, strings 12 through 17 in the previous example are in fact one single *qi* command. This technique can be used in order to keep the query code readable.

All previous remarks about textual and graphical queries hold for retrieval queries as well.

5. Appendix C - Changes from previous versions

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

Changes from version 1.3 to version 1.3.1

-- In section *2.6.1 LOGICAL group of commands* temporal expressions were added (e.g. bfr, aftr etc.).

-- The server `qserver` and the client `qi.client` are now called with an additional argument: the socket port number.