SOUR

**SYSTENA**

# INTELLIGENT QUERY SYSTEM

Functional Specification
&
Architecture

Version: 2
Revision: 1

COPYRIGHT © 1993, SYSTENA

## Table of Contents

Part I

1 Context ........................................ 2
    1.1 Document Layout ........................................ 2
2 IQS in SOUR context ........................................ 3
3 IQS main goals ........................................ 4

Part II

4 IQS operation modes ........................................ 5
    4.1 Assisted Mode ........................................ 5
        4.1.1 Syntactic Help ........................................ 6
        4.1.2 Semantic Help ........................................ 7
        4.1.2.1 Redundancy ........................................ 8
    4.2 Batch Mode ........................................ 9
5 Interface Query Language ........................................ 10
    5.1 IQL grammar(s) ........................................ 13
6 The History ........................................ 14

Part III

7 IQS data structures design ........................................ 18
8 IQS architecture ........................................ 20

Part IV

9 Final Remarks ........................................ 22
    9.1 Interaction with Conceptualizer ........................................ 22
    9.2 Interaction with Comparator & Modifier ........................................ 22
    9.3 Adaptable UI ........................................ 22
References ........................................ 23
Appendix A - IQL grammar ........................................ 24
Appendix B - IQS API ........................................ 30
    B.1 Data Types ........................................ 30
    B.2 Functions ........................................ 31

# Part I

# 1 Context

This document presents the Functional Specifications of the Intelligent Query System (IQS) of the SOUR system, following the informal requirements established in a former document [Systena & Sss 1993].

Its main purpose is to describe the main design issues of the IQS and to descibe the way of communication between IQS and the other SOUR modules.

IQS is a front-end component of SOUR. Thus, its specification also includes a description of the user-interface, as prototyped in Visual Basic Professional Version 3.0.

This overall description will also refer to some technical aspects related to IQS integration with the subsequent *C* layer and with other modules, such as the Conceptualizer and the Result Manager, and the underlying modules of the SOURLIB software bus (such as CTS [CTS-1.4 1993] and in particular ERA [ERA-3.5a 1994]) to which IQS has been plugged at final implementation level.

## 1.1 Document Layout

This document is structured as follows.

It starts by establishing out IQS's role in SOUR´s overall architecture.

Next, IQS's main objectives are put forward (this will omit many technical details which will be described later on the text).

That will lead the reader to IQS's *modus operandi*, which reflects much of the tool´s design philosophy.

A Query Language that serves IQS's needs will be presented, as well as the *Minimal Efficient Query* concept. The set of templates, *i.e*, the set of query phrases considered to be general enough to cover any IQS search will then be introduced.

After this, some technical details will be referred in order to justify the data structures choice that serve both convenience and efficiency goals, during search, retrieval and manipulation operations over repository objects.

Finally, IQS architecture is presented, showing relations between internal parts as well as the external relations with other SOUR modules.

The *Interface Query Grammar* is presented in Appendix A.

In Appendix B the first contact with IQS API takes place: the prototype definitions of a minimal set of functions that provide for the basic constructs upon which the complete module architecture is based, will be presented.

## 2 IQS in SOUR context

The Intelligent Query System is the SOUR module exclusively concerned with search and retrieval of information saved in the repository during the Conceptualization phase.

Therefore, IQS is one of the end-components of the SOUR global architecture, as shown in the diagram bellow.

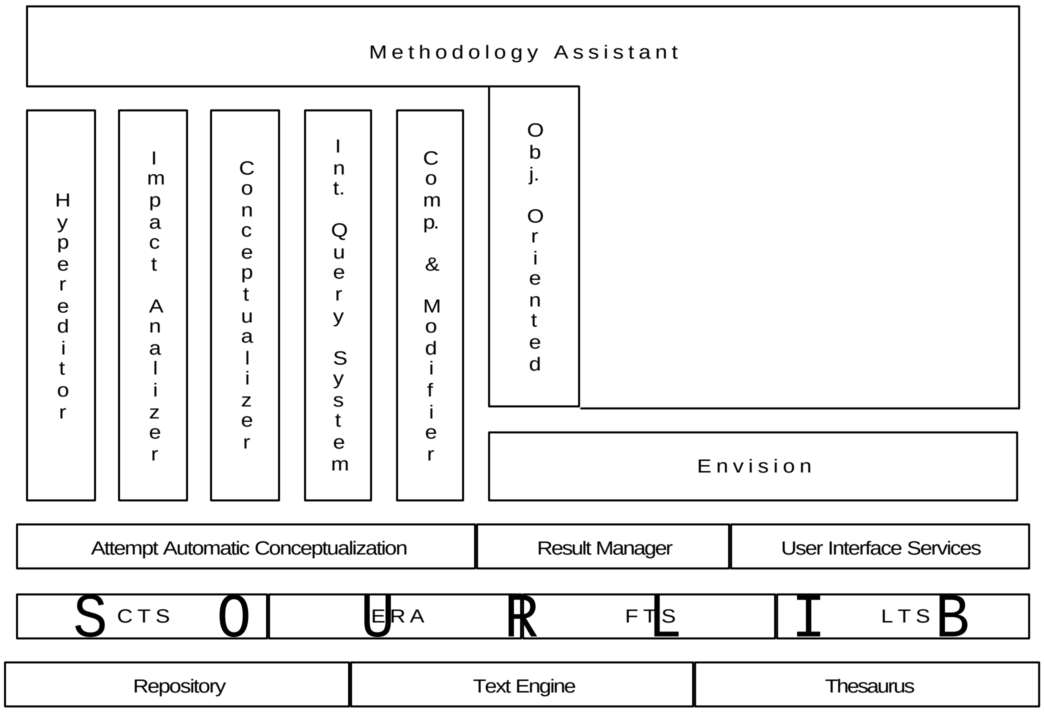

Figure 1 - IQS integration in SOUR global architecture

IQS is the browsing tool of the SOUR environment with the capability of accepting queries in order to retrieve reusable software components.

It offers the possibility of construction of query phrases, which are ultimately converted in repository search functions.

The queries made by using IQS are based on the conceptual schema of the SOUR repository presented in [CON-2.0 1993].

So, IQS will be able to browse and recover reusable software components, which were previously classified by the Conceptualizer and inserted into the repository, based on a linguistic standard.

By taking a minimal (but precise) grammatical description of an intended set of objects and by invoking a set of functionalities provided by SOURLIB (mainly by ERA[1]) as well as modules at the same abstract level (as Conceptualizer), the IQS set of functions will provide for query answers which are sets of objects.

[1]cf. [ERA-3.5a 1994] and regarding [ERA-1.2 1993].

## 3 IQS main goals

The IQS subsystem of SOUR has to accomplish the following main goals:

- to provide for the search of objects previously inserted into SOUR's repository by Conceptualizer, and to interact (as the main SOUR browsing tool) with the other components of the SOUR layout;

- to mirror the repository's logical schema for the query construction, *i.e*, to allow search by attributes (generic ones or class defined), facets, links, characteristic relations, aggregation degree, etc.;

- to allow search based on both strict or fuzzy criteria (in latter case, a certain degree of semantic similarity with respect to a certain object should be maintained during searching and filtering); *fuzzy* search will be implemented on top of CTS functionality;

- to be able to interact with other SOUR tools by feeding them with the querying results, in an easy access format; visualization of query results via the Result Manager is a typical example of this kind of cooperation;

- to avoid, as much as possible syntactical as well as semantic errors during query construction and resolution; such a constant assistance, guiding the end-user all over the query process is, perhaps, IQS's major added value, being materialized in an **Assisted Mode** of operation; being SOUR a set of tools oriented towards software reuse, it seems logical to offer such a degree of user friendliness;

- to allow also an operation mode in which convenience is deferred in favor of efficiency, by removing some aid features and interaction from the interface and providing batch resolutions. A **Batch Mode** way of querying is offered;

- to provide for a good integration of the various operation modes (Assisted and Batch), making it easy to swapp between them and reusing, at will, the set of calculated results; this integration is done on behalf of a History data structure that stores query phrases (and respective results) produced during the actual IQS session.

- to implement a minimal level of adaptability to the users particular needs, by making it possible to build, maintain, save, load and join query Histories, and by ensuring, at every moment, the coherence of the Histories internal data;

# Part II

# 4 IQS operation modes

From the main goals IQS is purposed to achieve, one can detect the convenience of at least two fundamental operation modes:

- an **Assisted Mode**, mainly concerned with validation procedures and therefore always trying to give a non-empty solution set to the query;
- a **Batch Mode,** best suited to evaluate large sets of queries without any interference from users;

The existence of these two operation modes makes possible the use of IQS by differently skilled users. This capability encompasses the goal of adaptability.

A novice end-user will naturally tend to start by using IQS in Assisted Mode. After becoming sensitive to the grammar behind the interaction scheme, it will be understood that Batch Mode can provide for less restrictive ways of querying.

IQS will provide a pseudo-natural language based way of querying. The formalism supporting the description of such a language will be a grammar, which will mainly allow to:

- check if the submitted query phrase is syntactically well formed, that is, if it belongs to the possible set of valid query phrases;
- to support design, implementation and operation of the interface layer ; the visual layer will then be implicitly under the control of the grammar behind the query description language.

This grammar can be used to automatically generate a parser. Thus, the parser is the code implementation of the grammar deterministic automata.

## 4.1 Assisted Mode

**Context Dependency** is the key feature of the Assisted Mode, allowing a permanent syntactic and semantic help while a query is under construction.

In practical terms, it implies that at any moment users are only allowed to proceed if the possible query result is a non-empty set of solutions.

This means that IQS doesn't have to deal, in Assisted Mode, with problems of grammar incompleteness, since the activation state of a query is well defined and only occurs when a complete grammar situation is possible.

So, there always must be a complete knowledge of all the possible future valid states, knowing the actual one.

This is possible because as IQS accepts query phrases, it must follow a grammar that controls the query execution and also deterministically sets the next state of the user interface.

In other terms, by knowing:

- the query construction language;

- the attributes and respective values that still remain to inspect,

IQS implicitly knows the set of tokens that can be added to the query synthesized so far and still can produce valid results.

In fact, it will lead the user through one of those possible valid paths, filtering, whenever possible, the set of objects remaining, these objects being the most specialized query solution so far obtained.

Note that object filtering is scattered in time. Query evaluation is a process that alternates with grammar checking on query tokens (which, in Assisted Mode are generated upon visual events).

Thus, one cannot say that at a certain moment all the query is evaluated at once. There is not a *unique* moment for query evaluation. The query solution refinement process is done as the query is being built.

### 4.1.1 Syntactic Help

Being user interface embedded, the underlying grammar will have the ability to prevent users from having bad syntactically formed queries. Both interface menus and buttons assure, at every time, that the next token added to the query being synthesized, is a valid one (one of the look-aheads of the present grammar rule).

The actual internal state of the parser for that grammar will be reflected in the *enable-disable* state and in the contents of the various objects being part of the visual interface (such as menus, buttons, list panes, etc.), and so, shall severely restrict interface choices to a set of syntactically valid ones.

In Assisted Mode there is a permanent interaction between the interface layer and the parser layer, the former being controlled by the latter.

Therefore, at parser level, the visual state is well known and pre-defined.

The user interface layer delivers to the parser a interactively synthesized query, that is, a query that was synthesized at the cost of mouse selections and other events that resulted in the addition to the query of a token (or a set of tokens).

Basically, every interactive event leads to a call to the parser, but only a small portion of these events will make the parser to filter the present temporary solution set.

There are some circumstances under which repeatedly adding a token to the query being submitted to the parser does not make the parser to filter the present set of AOIDs. For instance, by repeatedly clicking a button that presents a list of available AOG attributes being visible, and by adding the corresponding token(s) to the query it will have the same effect as a single click!

Even in the case that no repeated tokens are introduced in the query (one after another), there could be no filtering of objects. For instance, in the previous example, filtering does not happen until a pair *(*AOG *attribute selected,* AOG *attribute value selected)* is provided.

Thus, the parser, has two fundamental tasks:

- to set the next state of the user interface layer (syntactic control), even in the case of redundant tokens being added to the synthesized query;

- to filter the objects selected so far, whenever that is possible (semantic selection);

Assuming that, in the case of a successful finished query, we would like to preserve the synthesized text of that query, one can ask if is it worth to preserve all the query text, once only a small portion of the query received by the parser is relevant in terms of objects filtering.

Why not keep only the non-redundant part of the query, which by it would be enough to obtain exactly the same results if submitted to the parser?

That's why although the parser receives a query from the interface layer, it also synthesizes its own. Every time the parser does a filtering, the relevant tokens are added to an internally synthesized query, that is, two queries coexist during the same Assisted Mode session:

- the one provided by the visual layer, possibly with redundancies;

- the one synthesized so far by the parser; if desired, this query can be later reused for providing exactly the same results as its counterpart, but in an efficient way;

We shall call this latter form a *Minimal Efficient Query*.

Later, when presenting Batch Mode and the History as an integrator between the Assisted Mode and the Batch Mode, the role of these efficient forms will become quite clear.

For now, it is mainly important to retain that as a result of a successful query resolution via Assisted Mode, its Minimal Efficient Query version will be kept somewhere, possibly for a future reuse.

### 4.1.2 Semantic Help

In Assisted Mode, IQS will never synthesize queries with syntactical errors.

Another kind of possible errors are the semantic errors.

Semantic errors, all alone, imply empty solutions, despite the fact that in syntactical terms everything may be correct.

Therefore, a good semantic help is essential as a complement to a syntactical assistance.

From the IQS viewpoint, a semantic error occurs whenever a syntactically well-formed query produces an empty set of solutions.

The reasons for this kind of failure could be, among others:

**1.** there isn't really nothing in the repository that fits the provided description;

**2.** a bad path to the intended objects was specified;

**3.** the objects are actually there, even the path is correct, but some of the description provided for the objects does not suit them;

In a fully implemented Semantic Assisted Mode, users would expect to choose paths or descriptions from the user interface, matching at least one of the objects filtered so far. About such an assistance degree, one could say that "what you choose is what you get".

It would be impossible to choose a value of a certain attribute from a list pane if there were no objects that had that value for that attribute. In fact, that attribute would have been prevented from having access to the interface layer if there wasn't at least one object with a well-defined value for that attribute.

The enable and disable control of every visual feature will, then, obey to a simple but strict rule:

*IF (the next visual state supports this visual feature AND the objects filtered so far match at least one of the items of that feature)*
*THEN enable and refresh that feature;*
*ELSE disable it.*

For instance, if the next visual state supports a button that when pressed makes a list pane with names of attributes available, then, only if at least one of the objects filtered so far has a non empty value for at least one of those attributes, should the button be enable.

Following this scheme, the parser level prevents users, by anticipation, from taking a wrong path by means of a strong validation of every available interface option in the present state.

#### 4.1.2.1 Redundancy

Semantic help also reflects the way redundancy his handled.

The process of avoiding redundancy at the semantic level consists of not repeating any choice made before.

That is achieved by removing off the list-panes presented in the visual layer those items selected in a previously successful sub-query.

When a certain list-pane becomes empty, the parser level, which maintains that list-pane (and every low level data structure) and controls the state of the visual layer, simply disables that part of the interface, ensuring that every visual path accessing that visual object will be blocked.

There are, however two situations where only one choice can be made, even if the list-panes have more than one element:

- Class choice: should a class below the USR class be chosen, no more class choices will be allowed;

- Software Life Cycle choice: this reflects the fact that a set of objects may not belong to more than one Software Life Cycle;

In any case, the two situations above also reflect semantic conditions to be obeyed.

## 4.2 Batch Mode

A fully implemented Assisted Mode prevents users little familiarized with the repository information scheme from producing syntactical and semantic errors during an IQS session.

However, such a degree of assistance may become a restriction to those users who do have a comprehensive knowledge of both the repository schema and the query construction language, and would like to make queries in a more flexible way, by using that query construction language.

This mode allow users to submit to IQS a batch of queries and expect IQS to solve them or to interrupt the solving process whenever a syntactical or semantic error occurs. In the latter case, the user would have to correct the batch text in order to avoid the error and submit again the batch of queries to be solved.

Therefore, Batch Mode also encompasses the adaptability goal, which guides not only IQS, but also all the SOUR environment. The possibility that the end-user has to switch between Assisted and Batch Mode provides a high degree of flexibility and adaptability to different users needs.

In this way, the Batch Mode will best fit those users who already have managed to understand the adopted linguistic query standard.

There are two main issues related to Batch Mode and intimately connected with Assisted Mode:

- the language that supports the query description, known as the *Interface Query Language* (IQL);
- the place where the set of solutions of the solved queries are kept, known as the *History*;

## 5 Interface Query Language

At this point it should be clear that a query description language is a main design issue of the IQS SOUR sub-system.

The pseudo-natural language supporting the Batch Mode should be compatible, with the one behind the Assisted Mode parser. In fact, both languages should be the same, if possible.

The reason justifying this need is as follows: if the language used by the Assisted Mode parser to synthesize queries and the language on which Batch Mode queries are based on are both the same, then it will be possible to the Batch Mode to re-solve a query visually synthesized in Assisted Mode, as while as a communication mechanism is provided between the two operation modes. As we shall see, the History provides such a mechanism.

A common language and a History are, thus, the main features upon which a good integration between the two operation modes will be achieved. Besides, such an integration will make possible to share other features of both modes, providing for a clean implementation and easy maintenance.

Having the Batch Mode and the Assisted Mode parser to share a common language has an obvious, but important, consequence: the set of queries one can submit to the Batch Mode is equal to the set of synthesizable queries by the Assisted Mode parser!

However, during Syntactic Help discussion we have stated that those synthesized queries are Minimal Efficient forms. Thus, these are the forms the Batch Mode queries must obey. Redundant forms are simply not allowed in Batch Mode.

Remember that redundancy was introduced in Assisted Mode as a consequence of the behaviour of the visual layer: sometimes it is possible to use a visual feature, repeatedly, without any object filtering having place. That cannot not happen in Batch Mode (in fact, those visual features aren't accessible in Batch Mode but their use could be simulated by introducing the right tokens in the right places when writing the query text).

The complete set of non-redundant synthesizable queries in Assisted Mode, and thus, the set of valid possible queries one can submit to a Batch Mode resolution is known as the *Templates* (or *Minimal Efficient Forms*) -- see Figure 2.

The Interface Query Language (or IQL) is the language on which the Templates are based on. Thus, Templates are the syntactically valid combinations of the various tokens of the IQL.

There are two groups of Templates:

- the so called *Kernel* templates (see Figure 2.1);

- the reuse based (non-*Kernel)* templates (see Figure 2.2).

```
NUMBER TEMPLATE1 GET ALL CLASS="class" <AttributeDescription1>*

NUMBER TEMPLATE2 GET ALL CLASS="class" <AttributeDescription2>*

NUMBER TEMPLATE3 GET ALL CLASS="class" <AttributeDescription1>*
                               [ AND IS COMPOUND
                               <CompoundDescription>* ]

NUMBER TEMPLATE4 GET ALL CLASS="class" <AttributeDescription1>*
                               [ AND BELONG TO COMPOUND
                               <CompoundDescription>* ]
```

Figure 2.1 - IQL *Kernel* Templates

```
NUMBER TEMPLATE5 <Query> [ LINKED BY "relation" [ WITH <Query> ] ]

NUMBER TEMPLATE6 <Query> [ LINKED TO <Query> [ BY "relation" ] ]

NUMBER TEMPLATE7 <Query> [ OR <Query> ]

NUMBER TEMPLATE8 <Query> [ RESTRICTED TO <AttributeDescription1>+ ]
```

Figure 2.2 - IQL non-*Kernel* Templates

*Kernel* templates are those templates that don't reuse the set of results from previous solved templates. In that sense, they may be called *atomic* or *kernel*.

On the other hand, a second group of templates is exclusively or partially based on that reuse. Note that both *Kernel* and non-*Kernel* templates can be reused (what really matters is their associated set of objects).

However, this reuse suffers from certain restrictions:

- one template cannot make a reference to himself, *i.e*, references must not be recursive;

- references cannot be made to templates not yet solved.

In fact, the first restriction turns out to be a particular case of the second, but it's important enough, by itself, to be pointed out separately.

Concerning the second restriction, the nature of a reference to a solved query is intimately connected with the History structure.

Figure 2.3 shows some shared definitions of *Kernel* and non-*Kernel* Templates.

The meaning of each *Template* is as follows:

- *Template1*: allows for the search of objects of a certain class, by describing their generic or class attributes and facets; in the case of a facet, besides a name and a value, a conceptual distance must also be provided;

- *Template2*: extends *Template1* by allowing the specification of one or more Software Life Cycle Phases or of a Software Life Cycle (SLC); having filtered by a certain Software Life Cycle implies that no more Software Life Cycle should be specified because one object can only have one Software Life Cycle associated; also, in this case, choosing later a Phase cannot be done because no additional filtering would occur; this is due to the fact that the ERA layout does not provide for associating an object to a Phase; in particular, if a Phase is chosen even before a SLC, this implies refining the present query solution by the SLCs containing that Phase;

- *Template3*: same as *Template1* but allowing to chose the objects being compounds, in which case the refinement by one or more characteristic relations becomes possible; searching by a certain characteristic relation, implicitly implies, in turn, filtering again, because only clusters have characteristic relations associated and so clusters are selected from the set of compounds;

- *Template4:* extends *Template1* by making possible to characterize the compound objects containing the ones selected so far; from these primitive objects, only the ones being members of the compounds matching the provided compounds description (possibly involving the specification of one or more characteristic relations) are kept;

- *Template5*: allows for the filtering of objects from a previously solved query, kept in the History, by considering them as sources of a certain relation, whatever are the sinks; optionally, a set of sinks can be specified, again by referencing a set of objects associated with an History query;

- *Template6*: similar to *Template5* but a set of hypothetical sinks is considered first, and the refinement by a relation eventually follows;

- *Template7*: allows for the OR*ing* of the objects associated with two History queries;

- *Template8*: similar to *Template1*, except that the initial set of objects to be characterized is not retrieved from a specified class, but from a History query; this allows for later refinement of a previously solved query, kept in the History;

```
     <AttributeDescription1> = AND ( <GenDescp> | <FacDescp> | <AttDescp> )
     <GenDescp> = GENNAME="genname" AND GENVALUE="genvalue"
     <FacDescp>    =    FACNAME="facname"    AND    FACVALUE="facvalue"    AND
CONCEPTDIST>=<Dist>
     <AttDescp> = ATTNAME="attname" AND ATTVALUE="attvalue"

     <AttributeDescription2> = ( <AttributeDescription1>* (AND <PhaDescp>)* )*
                               [ AND <SlcDescp> <AttributeDescription1>* ]
     <SlcDescp> = SLCNAME="slcname"
     <PhaDescp> = PHANAME="phaname"

     <CompoundDescprition> = <AttributeDescription1>* (AND <CrlDescp>)*
                             <AttributeDescription1>*
     <CrlDescp> = CRLNAME="crlname"

     <Dist> = NUMBER%
     <Query> = #NUMBER

     NUMBER = [0-9]+
```

Figure 2.3 - *Kernel* and non-*Kernel* Templates common definitions

## 5.1 IQL grammar(s)

The previous discussion pointed out the advantages of having a shared query description language used by both Batch and Assisted modes.

However, the compatibility between both languages is obtained at the cost of eliminating the redundancy (and possibly some incompleteness[2]) introduced by the Visual Layer during the query phrase construction, in Assisted Mode. Remember that, in Assisted Mode, the parser has to synthesize a *Minimal Efficient Form* from the received query.

It is then clear that, in Assisted Mode, to be able to recognize the query phrases visually synthesized, the parser in charge must follow a grammar with slightly different rules than the parser taking care of Batch Mode. Those rules are the ones allowing for recognition of the redundancy and incompleteness. These situations are not acceptable in Batch Mode, thus resulting in a simpler grammar for that operation mode.

In lexical terms nothing changes between the two grammars, *i.e.*, the set of valid tokens (the vocabulary) remains intact. Only the rules to combine them are slightly more flexible in Assisted Mode, due to the impact of the Visual Layer (keep in mind, however, that the Assisted Mode parser stills synthesize internally a *Minimal Efficient Form* to provide for compatibility with Batch Mode).

[2]In this context, incompletness means that, althought not redundant, some of the tokens being part of the query phrase won't imply any filtering; thus, there's something missing (incompletness) in the query phrase to make filtering ocur. See also section **4.1.1 Sintatic Help**.

However, the differences between the two operation mode grammars are not purely syntactical ones. The semantic actions executed when the same token or set of tokens is recognized may vary from mode to mode. These differences lie not only in the implementation of the semantic action. They may even occur at the placement of the semantic actions among the various tokens in the rules.

When building the grammar for each operation mode, the placement of the calls to the semantic actions in the rules definition, reflects much of the behaviour philosophy of the interface layer.

In Assisted Mode grammar, for instance, all semantic actions are terminal[3], that is, the parser is not allowed to recognize a set of tokens, execute the respective semantic action, and next try the matching process again. In Assisted Mode, almost every visual event implies adding a token to the query phrase under construction, and every time that happens, the parser is called in order to recognize the complete query phrase and execute one terminal semantic action: the one resulting from having the new token(s) added to be the last one(s) of the production rule. The reason for this kind of behaviour is convenience in implementation terms: to expand the grammar in a *stair-fashion* (see Appendix A) and to call the parser every time a new token enters the present Visual Layer query phrase is much easier than keeping the parser in background, always trying to recognize something valid.

This strict alternation between adding a token(s) to the query phrase, let the parser recognize the new token(s), execute the respective semantic actions and immediately return the control to the Visual Layer, does not happen in Batch Mode. In Batch Mode, query phrases are supposed to be complete (and not under construction) when delivered to the parser, and thus, semantic actions can be placed anywhere among the rules of the grammar. This is a reason why a Batch Mode grammar is simpler than an Aide Mode one.

Having two grammars (each one for a specific operation mode) one could naturally conclude for the need of two parsers.

The number of parsers actually needed is one, serving both operation modes, and thus following a unique grammar resulting from the union of both operation mode's grammars.

The reason lying behind this is *convenience*: being automatically generated and due to many similarities of their grammars, both parsers share many common data structures and functionalities; this situation would rise many conflicts when trying to gather both parsers into one single code module; the advantages of automatic code generation would soon be diluted by the effort spent in the mix process.

So, the solution is to join the grammars by moving up their roots (the first rule of the grammar) to a common level (see Figure 3.1 and Figure 3.2) and thus allowing only one parser, serving both grammars, to be generated.

A complete formal description of the common grammar is included in the Appendix A.

# 6 The History

The History is the main integrator mechanism between the two IQS operation modes.

[3]They are the rightmost symbol in a rule description.

In abstract terms, the History is a data structure that implements a set of pairs of the format (*query text, set of objects solving the query*) -- see Figure 4.

```
/* root of the Batch Mode grammar; semantic actions omitted*/
batchIqs → batchIqs batchTemplate1
        | ...
        | batchIqs batchTemplate8
        | batchTemplate1
        | ...
        | batchTemplate8
        | ε

/* remaining Batch Mode grammar rules omitted */

/* root of the Assisted Mode grammar; semantic actions omitted*/
assistIqs → assistTemplate1
         | ...
         | assistTemplate8

/* remaining Assisted Mode grammar rules omitted */
```

Figure 3.1 - Before joining Batch and Assisted Mode grammars

```
/* common Batch and Assisted mode grammar root */
Iqs → batchIqs
   | assistIqs

/* same as contents of Figure 3.1 */
```

Figure 3.2 - After joining Batch and Assisted Mode grammars

$$\text{History} \equiv \{ \ (t_1, \{o_{11}, \ldots, o_{1n}\}), \ldots, (t_i, \{o_{i1}, \ldots, o_{im}\}) \ \}$$

where $t_i \equiv$ text of the ith query

and $o_{im} \equiv$ mth object solving the ith query

Figure 4 - Structure of the History

During an Assisted Mode IQS session, a History is built based on the Minimal Efficient Queries of all the successful solved queries.

It is important not to lose that History: think of one user wanting to proceed with the session later, preserving the work done during the previous session.

This capability introduces a curious problem: it's nice to save the History for future reuse, but what happens if meanwhile the repository contents changes in such a way that the set of objects that claim to solve each query in the History are no longer the valid solutions?

Thus, it is of no use to save the complete History: only the text of the queries is of interest because there is no guarantee at all that the same solutions to a query apply between IQS sessions.

Every time a History is loaded, it must be re-solved, query by query.

The Batch Mode it's ideal to perform this task: one can load the text part of the History and let the Batch Mode take care of solving it. After that, the History reflects the true actual state of the repository and, if wanted, a swapping to the Assisted Mode can be performed.

Swapping between Assisted Mode and Batch Mode should now be a trivial task:

- a user in Assisted Mode, no matter the origin of the present History, should be able to change to Batch Mode and, once there, to edit the History (modify it by hand) or to load another History, replacing the actual or joining them;

- a user in Batch Mode, no matter the origin of the present History, should be able to change to Assisted Mode and, once there, to access the History in a totally transparent way;

Thus, the History serves two main purposes:

- total integration between Batch Mode and Assisted Mode: the History allows coherent swapping between Assisted Mode and Batch Mode, because the data structure implementing which stores the History is shared between the two operation modes.

- in conjunction with Batch mode, implements a minimum level of adaptability by allowing the retrieval of work done in previous IQS sessions.

A final note concerns the way how queries are referenced once kept in a History.

The easiest way (and the one assumed) is to use the arrival order to the History: the $n^{th}$ query is the $n^{th}$ query arrived to History and thus any non-Kernel queries referencing others must assume that rule. That's why valid references are only those behind the arrival order of the query presently being solved.

The non-Kernel templates (see Figure 2.2) use a clear notation to make those references:

```
<Query> = #NUMBER

NUMBER = [0-9]+
```

Figure 5 - Syntax of History references

Note that during query resolution, a non-*Kernel* query referencing a History query doesn't care about his text component. In evaluating a query what really matters are the objects associated; they are the ones to be reused, not the query text.

Also, there is an important issue related to what happens when joining histories. Consider the example of Figure 6.

One could ask what happens when during Batch Mode an attempt is made to solve the imported History. After all, references made in the loaded History could be correct in the context of <u>that</u> History, but certainly aren't in the global context of <u>both</u> Histories (the actual one and the loaded ). For instance, the query #2 of the loaded History reuses queries #0 and #1 of <u>that</u> History, not of the present one.

```
/* History for the present IQS session*/

#0 TEMPLATE1 GET ALL CLASS = "X"
#1 TEMPLATE5 #0 LINKED BY "linkname"

/* Loaded History of an old IQS session */

#0 TEMPLATE1 GET ALL CLASS = "Y"
#1 TEMPLATE1 GET ALL CLASS = "Z"
#2 TEMPLATE7 #0 OR #1
```

Figure 6 - Joining the queries of two Histories

A possible solution to these problems is not to re-solve the actual History (provided nothing changed in the repository) and let the Batch Mode handle only the loaded one.

If a query from the loaded History is successfully solved, then it is added to the actual History, all references being converted from local to global ones.

During resolution of the loaded History any time the parser recognizes the beginning of a query, it checks to see if his ordering number is correct in the local context of the loaded History. If so, it continues the solving process. If the query is non-*Kernel* there will be local references to other queries of the loaded History. If those references are valid, *i.e*, they point to previously solved queries of the loaded History, then they are converted to global references (remember that local solved queries enter the global History) in order to access their set of associated objects.

At the end of this process, if everything went fine, then the loaded History has seen all his local references converted into global ones!

For the previous example, the final History, resulting from the union of both the actual and the loaded ones, should be:

```
/* History for the present IQS session*/

#0 TEMPLATE1 GET ALL CLASS = "X"
#1 TEMPLATE5 #0 LINKED BY "linkname"
#2 TEMPLATE1 GET ALL CLASS = "Y"
```

```
#3 TEMPLATE1 GET ALL CLASS = "Z"
#4 TEMPLATE7 #2 OR #3
```

Figure 7 - Result of Histories union

# Part III

# 7 IQS data structures design

The choice of the best suited data structures to support the most relevant design features of IQS is a critical issue.

A good policy is to integrate all those relevant data structures in a record. That should allow an easy access and management control.

Which data to put into the record depends upon which data is relevant or considered enough to define the state of the user-interface. Note that the record does not integrate any low-level parser automatic generated control data structures, *i.e*, those data structures exclusively concerned with token recognition. Those are details that were left on the behalf of automatic generated parser code. Of course one cannot left the record only with visual layer related items. There must be a place for the History, the set of objects presently solving the query so far synthesized, the text of the query itself, etc.

An abstract representation of a record used for keeping the most relevant information concerning the present user-interface and queries resolution state, could be:

```
HighLevelParserState ≡ {
QS  ≡ { objects presently solving the query },
QSC ≡ { objects being compounds },
QSV ≡ { generic attributes values or class attributes values
        or facets values },

Query   ≡ "text of the synthesized query until present moment",
History ≡ history for the actual iqs session - see Figure 3,

VBState ≡ { flags controlling Enable/Disable state of Visual Basic
            interface features },

AOGAttribs ≡ { available AOG attributes to choose from },
FACAttribs ≡ { available facets to chose from },
CLAAttribs ≡ { available class attributes to choose from },
SLCNames   ≡ { available software life cycles to choose from },
PHANames   ≡ { available software life cycle phases to choose from },
CRLNames   ≡ { available characteristic relations to choose from },
LNKNames   ≡ { available links to choose from },

BatchOn     ≡ flag sensing batch mode activation,

RM  ≡  { copy of QS exclusively to be used by Result Manager },

NextLocalHIndex ≡ next valid history index in a local context
}
```

Figure 8 - High level IQS parser state

IQS is an hybrid tool, made at the cost of two different technologies: Visual Basic Professional 3.0 taking care of the user-interface and a C DLL module accessing the repository database in order to solve queries.

The control of the IQS interface reflects the integration of these two technologies. That's why the `VBState` field does not take care totally of the control of the Visual Layer. Instead, many details are left to the Visual Layer (Visual Basic layer handling the user interface). As far as the `VBState` field is concerned, just the deterministic behaviour of the user interface is considered.

One thing is to decide what information to preserve. Another one is to choose physical layouts implementing the abstract descriptions. That's when convenience and efficiency may conflict between each other.

For the sake of preserving this discussion far from implementation details, for now, one needs only to know that the main design decision concerning the implementation of the previous High Level Parser State was to use Dynamic Arrays to implement sets.

In IQS context, a Dynamic Array is as follows:

```
Dynamic Array  ≡ { index, { information items } }
```

Figure 9 - Structure of a Dynamic Array

A Dynamic Array is nothing more than a structure with two fields:

**1**. an index reflecting the actual number of objects contained in the array;

**2**. the actual array containing the set of objects.

The index component has some basic properties:

• index ≤ -1 implies an empty array, that is, the array field is discarded;

• index + 1 = number of objects in the array, as long as index ≥ 0.

This simple structure allows a quick and clean implementation of a sub-module of basic set operations such as *CopySet*, *SetDifference*, *SetIntersection*, *SetUnion*, *MakeSet*, etc., which are enough to shield IQS API main functions from many low-level details concerning manipulation and filtering of objects during query resolution.

# 8 IQS architecture

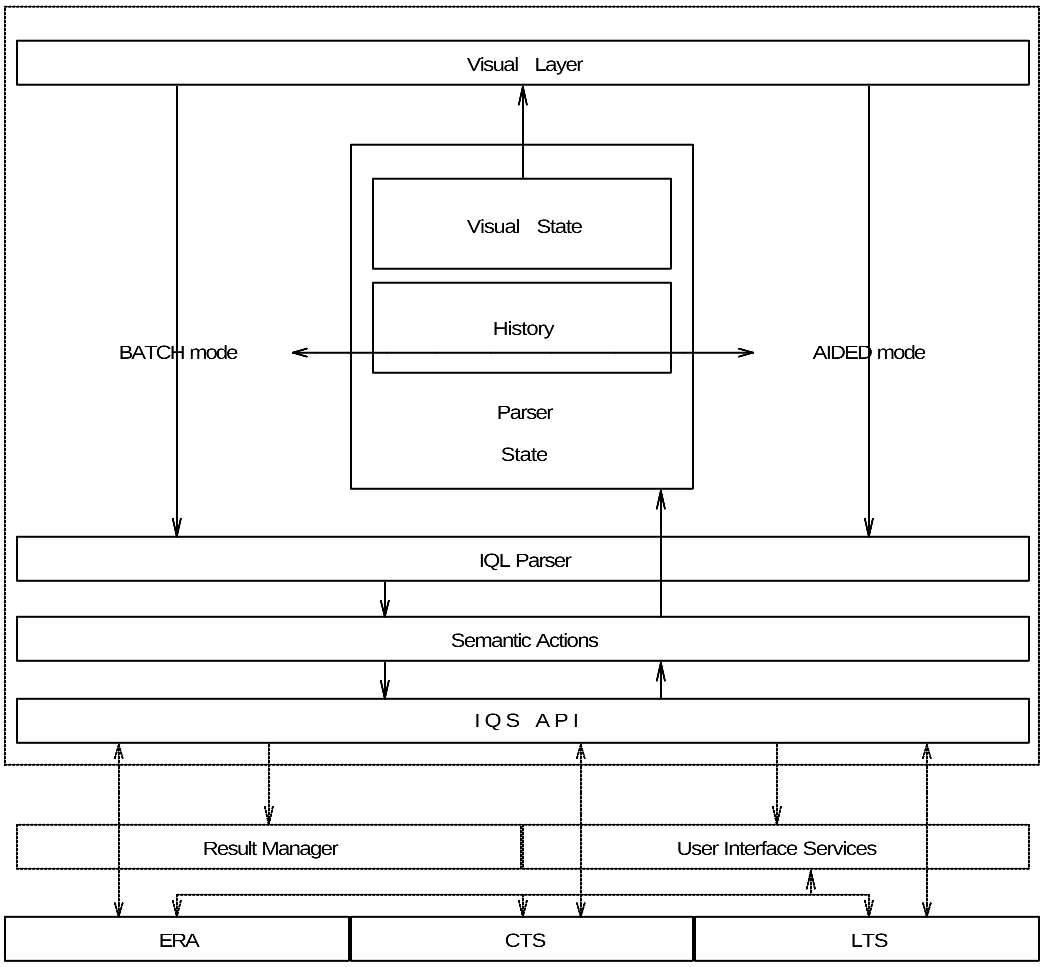

Figure 10 - Intelligent Query System Architecture

Figure 10 depicts the main internal parts of the IQS subsystem as well as the relations involved among them.

On the basis of the contents of Figure 10 it is possible to quickly describe the overall behaviour of IQS.

On Assisted Mode, the Visual Layer (Visual Basic Professional 3.0 implemented) reacts to interface events (mouse clicks, selections, etc.) by building a query phrase and then delivering it to the IQL parser. On Batch Mode, a "job" or "batch" of queries is submitted to the same IQL parser. That's why Figure 10 presents two ways of calling the Parser Layer, each one following a specific behaviour[4].

Most of the times, the semantic actions specified to be executed whenever a successful syntactical recognition of the query phrase(s) occurs, will be based on IQS API services (in turn, this services are implemented on top of other SOUR modules and SOURLIB).

[4]see also section **5.1 IQL grammar(s).**

As a result of those semantic actions, the present Parser State will be modified. The Parser State keeps information about the query presently being synthesized (text and objects associated - the temporary solution), the History, the next state of the Visual Layer (Visual State) depending on the success or insuccess of the semantic action, and other internal management structures[5].

Ultimately, the Visual Layer reflects the changes in the Parser State by retrieving some critical interface information (the Visual State). This dependency of the state of the Visual Layer on a low-level control mainly occurs in Assisted Mode.

Every time a swap occurs between Assisted Mode and Batch Mode, the History data structure should remain intact, if one desires to preserve and reuse results from previous successful queries.

The IQL Parser is the automatic generated parser implementing the common grammar for both Assisted and Batch Modes[6].

Bellow IQS API, only the modules providing services to IQS are considered.

[5]for an high level abstract description of the parser state recall section **7 IQS data structure design.**

[6]refer to section **5.1 IQL grammar(s)** for more details about this issue.

# Part IV

# 9 Final Remarks

This document presented INESC's proposal for SYSTENA about the functional specification of the IQS subsystem of the SOUR software system.

This part of the document briefly skims through a few topics which may be of interest concerning the discussion and upgrading of the solution found in order to implement IQS.

## 9.1 Interaction with Conceptualizer

IQS can be used as a tool plugged with Conceptualizer in order to make possible to *reconceptualize* or *delete* a set of objects, given by a call to an IQS query. If it's desired to apply the same action to objects with a common characteristic (*e.g.* an attribute, a facet term, *etc.*), a call to IQS provides a natural and linguistic based way of doing that.

## 9.2 Interaction with Comparator & Modifier

As stated in [CM-1.4 1993] logical AOs may be used as "search templates" providing IQS with a set of objects which can be latter refined by a future query. Comparator & Modifier can therefore be plugged to IQS in order to make possible to have queries using logical AOs for AO-repository browsing.

## 9.3 Adaptable UI

There are several issues to be considered here, although the overall idea is simply to instruct the machine on how to provide "good hints" for incomplete information:

- A semi-Assisted Mode could be suggested, providing the end-user the capability of switching on and off the semantic on-line assistance.
- A History can be saved in a file, making possible to the user to load a previously made batch of queries.
- Proximity driven list panes --- this could be applicable to facet values by presenting the CTS-closure of terms ordered by the proximity measure.
- user-adaptable terminology --- the idea is to endow each user-group with a local thesaurus mapping the group's own terminology to the standard one, cf. [Larsen &Yager 1992]. Such a thesaurus could actually be built implicitly by a "learning process" which would record pairs *t,t'* where *t* is the first term input by the user and *t'* is his/her final choice after an eventual CTS-browse.

## References

CM-1.4 1993

*Comparator & Modifier*. Functional Specification & Architecture. Version: 1; Revision: 4; Workpackage WP2B of Collaboration Offer by INESC.

CON-2.0 1993

*Conceptualizer*. Specifiche dei Requisiti - Architecttura. Version: 2; Revision: 0; Author: R. Brunialti, November 1993.

CTS-1.4 1993

*Concept/Thesaurus Subsystem*. Specifiche dei Requisiti - Architettura - Note di implementazione. Version: 1; Revision: 4; Authors: Brunialti R., Marano D. April 1993.

ERA-1.2 1993

*Easy Repository Access*. Specifiche dei Requisiti - Architettura. Version: 1; Revision: 2; Author: Brunialti R., April 1993.

ERA-3.5a 1994

*Easy Repository Access*. API Reference. Version: 3; Revision: 5a; Author: R. Brunialti, June 1994.

Larsen &Yager 1992

Larsen &Yager 1992 Larsen H.L., Yager R.R. *The Use of Fuzzy Relational Thesauri for Classificatory Problem in Information Retrieval and Expert Systems*. IEEE Trans. Syst.,Man, Cybern, SMC-23(1):31-41.

SYSTENA &SSS 1993

*Requirement Specifications for Conceptualizer, IQS, Comparator and Modifier*. Annex of Collaboration Offer by INESC, April 1993.

## Appendix A - IQL grammar

The following is a BNF based description of the Interface Query Language grammar. This formal description omits semantic actions details. Instead, the notation `{...}` is used in order to show where a call to a semantic action would take place if the parsing process succeeded.

Remember that this grammar result from joining an Assisted Mode suited grammar and a Batch Mode one[7]. For each *Template* the rules relative to each mode are presented.

Notice also the *stair-fashion* assumed by the rules concerning Assisted Mode, resulting from the alternation between Visual Layer and Parser Layer control of the IQS.

```
/* HYBRID GRAMMAR FOR BATCH MODE AND ASSISTED MODE IQL */

/* common Batch and Assisted mode grammar root */
Iqs  ::    batchIqs
     |     assistIqs

/* batch mode branch */
batchIqs::   batchIqs batchTemplate1 {...}
        |    batchIqs batchTemplate2 {...}
        |    batchIqs batchTemplate3 {...}
        |    batchIqs batchTemplate4 {...}
        |    batchIqs batchTemplate5 {...}
        |    batchIqs batchTemplate6 {...}
        |    batchIqs batchTemplate7 {...}
        |    batchIqs batchTemplate8 {...}
        |    batchTemplate1 {...}
        |    batchTemplate2 {...}
        |    batchTemplate3 {...}
        |    batchTemplate4 {...}
        |    batchTemplate5 {...}
        |    batchTemplate6 {...}
        |    batchTemplate7 {...}
        |    batchTemplate8 {...}

/* assisted mode branch */
assistIqs::   assistTemplate1
         |    assistTemplate2
         |    assistTemplate3
         |    assistTemplate4
         |    assistTemplate5
         |    assistTemplate6
         |    assistTemplate7
         |    assistTemplate8
         |    CHECK {...}
         |    ABORT {...}

/* some assisted and batch mode common rules */
ListOfIDENT ::  IDENT {...}
           |   ListOfIDENT IDENT {...}
/*-----------------Template1 BATCH mode rule------------------*/
```

[7]refer to section **5.1 IQL grammar(s)**.

```
batchTemplate1    ::   NUMBER {...} TEMPLATE1 {...} GETALLCLASS =
                       "ListOfIDENT" {...} batchTemplate11

batchTemplate11   ::   ε
                  |    AND batchAttrDesc

batchAttrDesc   ::   batchGenDesc batchTemplate11
                |    batchFacDesc batchTemplate11
                |    batchAttDesc batchTemplate11

batchGenDesc    ::    GENNAME {...} = "ListOfIDENT" {...} AND
                      GENVALUE = "ListOfIDENT" {...}

batchFacDesc    ::    FACNAME {...} = "ListOfIDENT" {...} AND
                      FACVALUE = "ListOfIDENT" AND
                      CONDIST >= NUMBER {...}

batchAttDesc    ::    ATTNAME {...} = "ListOfIDENT" {...} AND
                      ATTVALUE = "ListOfIDENT" {...}

/*-----------------Template1 ASSISTED mode rule------------------*/
assistTemplate1   ::   TEMPLATE1 {...}
                  |    TEMPLATE1 GETALLCLASS = "ListOfIDENT" {...}
                  |    TEMPLATE1 GETALLCLASS = "ListOfIDENT" AND
                       AttrDesc

AttrDesc    ::   AttrName
            |    AttrNameAnd AttrDesc
            |    AttrNameEqId
            |    AttrNameEqIdAnd AttrDesc
            |    AttrNameEqIdAndAttrValueEqId
            |    AttrNameEqIdAndAttrValueEqIdAnd AttrDesc

AttrName    ::   GENNAME {...}
            |    FACNAME {...}
            |    ATTNAME {...}

AttrNameAnd ::   GENNAME AND
            |    FACNAME AND
            |    ATTNAME AND

AttrNameEqId::   GENNAME = "ListOfIDENT" {...}
            |    FACNAME = "ListOfIDENT" {...}
            |    ATTNAME = "ListOfIDENT" {...}

AttrNameEqIdAnd::   GENNAME = "ListOfIDENT" AND
               |    FACNAME = "ListOfIDENT" AND
               |    ATTNAME = "ListOfIDENT" AND

AttrNameEqIdAndAttrValueEqId::   GENNAME = "ListOfIDENT" AND GENVALUE
                                 = "ListOfIDENT" {...}
                            |    FACNAME = "ListOfIDENT" AND FACVALUE
                                 = "ListOfIDENT" AND CONDIST
                                 >= NUMBER {...}
                            |    ATTNAME = "ListOfIDENT" AND ATTVALUE
                                 = "ListOfIDENT" {...}
AttrNameEqIdAndAttrValueEqIdAnd::   GENNAME = "ListOfIDENT" AND
```

```
                                          GENVALUE = "ListOfIDENT" AND
                                    |     FACNAME = "ListOfIDENT" AND
                                          FACVALUE = "ListOfIDENT" AND
                                          CONDIST >= NUMBER AND
                                    |     ATTNAME = "ListOfIDENT" AND
                                          ATTVALUE = "ListOfIDENT" AND

/*-----------------Template2 BATCH mode rule------------------*/

batchTemplate2     ::    NUMBER {...} TEMPLATE2 {...} GETALLCLASS =
                         "ListOfIDENT" {...} batchTemplate22

batchTemplate22    ::   ε
                   |    AND batcht2AttrDesc

batcht2AttrDesc    ::   batchGenDesc batchTemplate22
                   |    batchFacDesc batchTemplate22
                   |    batchAttDesc batchTemplate22
                   |    batchSlcDesc batchTemplate11
                   |    batchPhaDesc batchTemplate22

batchSlcDesc       ::    SLCNAME {...} = "ListOfIDENT" {...}

batchPhaDesc       ::    PHANAME {...} = "ListOfIDENT" {...}

/*-----------------Template2 ASSISTED mode rule------------------*/

assistTemplate2    ::   TEMPLATE2 {...}
                   |    TEMPLATE2 GETALLCLASS = "ListOfIDENT" {...}
                   |    TEMPLATE2 GETALLCLASS = "ListOfIDENT" AND
                        t2AttrDesc

t2AttrDesc   ::   AttrName
             |    AttrNameAnd t2AttrDesc
             |    AttrNameEqId
             |    AttrNameEqIdAnd t2AttrDesc
             |    AttrNameEqIdAndAttrValueEqId
             |    AttrNameEqIdAndAttrValueEqIdAnd t2AttrDesc
             |    SlcDesc
             |    PhaDesc

SlcDesc      ::   SLCNAME {...}
             |    SLCNAME AND t2AttrDesc
             |    SLCNAME = "ListOfIDENT" {...}
             |    SLCNAME = "ListOfIDENT" AND t2AttrDesc

PhaDesc      ::   PHANAME {...}
             |    PHANAME AND t2AttrDesc
             |    PHANAME = "ListOfIDENT" {...}
             |    PHANAME = "ListOfIDENT" AND t2AttrDesc

/*-----------------Template3 BATCH mode rule------------------*/

batchTemplate3     ::    NUMBER {...} TEMPLATE3 {...} GETALLCLASS =
                         "ListOfIDENT" {...} batchTemplate33
```

```
batchTemplate33   ::   ε
                  |    AND batchAttrDesc batchTemplate333
                  |    AND ISCOMPOUND {...} batchTemplate3333

batchTemplate333  ::   ε
                  |    AND ISCOMPOUND {...} batchTemplate3333

batchTemplate3333 ::   ε
                  |    AND batchCAttrDesc

batchCAttrDesc    ::   batchGenDesc batchTemplate3333
                  |    batchFacDesc batchTemplate3333
                  |    batchAttDesc batchTemplate3333
                  |    batchCrlDesc batchTemplate3333

batchCrlDesc    ::    CRLNAME {...} = "ListOfIDENT" {...}

/*-----------------Template3 ASSISTED mode rule------------------*/

assistTemplate3   ::   TEMPLATE3 {...}
                  |    TEMPLATE3 GETALLCLASS = "ListOfIDENT" {...}
                  |    TEMPLATE3 GETALLCLASS = "ListOfIDENT" AND
                       AttrDesc
                  |    TEMPLATE3 GETALLCLASS = "ListOfIDENT" AND
                       AttrDescBeforeC ISCOMPOUND {...}
                  |    TEMPLATE3 GETALLCLASS = "ListOfIDENT" AND
                       AttrDescBeforeC ISCOMPOUND AND CAttrDesc
                  |    TEMPLATE3 GETALLCLASS = "ListOfIDENT"
                       AND ISCOMPOUND {...}
                  |    TEMPLATE3 GETALLCLASS = "ListOfIDENT"
                       AND ISCOMPOUND AND CAttrDesc

AttrDescBeforeC   : AttrNameAnd
                  | AttrNameAnd AttrDescBeforeC
                  | AttrNameEqIdAnd
                  | AttrNameEqIdAnd AttrDescBeforeC
                  | AttrNameEqIdAndAttrValueEqIdAnd
                  | AttrNameEqIdAndAttrValueEqIdAnd AttrDescBeforeC

CAttrDesc   ::   AttrName
            |    AttrNameAnd CAttrDesc
            |    AttrNameEqId
            |    AttrNameEqIdAnd CAttrDesc
            |    AttrNameEqIdAndAttrValueEqId
            |    AttrNameEqIdAndAttrValueEqIdAnd CAttrDesc
            |    CRLNAME {...}
            |    CRLNAME AND CAttrDesc
            |    CRLNAME = "ListOfIDENT" {...}
            |    CRLNAME = "ListOfIDENT" AND CAttrDesc

/*-----------------Template4 BATCH mode rule------------------*/

batchTemplate4    ::   NUMBER {...} TEMPLATE4 {...} GETALLCLASS =
                       "ListOfIDENT" {...} batchTemplate44
```

```
batchTemplate44   ::   ε
                  |    AND batchAttrDesc batchTemplate444
                  |    AND BELONGTOCOMPOUND {...} batchTemplate3333

batchTemplate444  ::   ε
                  |    AND BELONGTOCOMPOUND {...} batchTemplate3333

/*-----------------Template4 ASSISTED mode rule------------------*/

assistTemplate4   ::   TEMPLATE4 {...}
                  |    TEMPLATE4 GETALLCLASS = "ListOfIDENT" {...}
                  |    TEMPLATE4 GETALLCLASS = "ListOfIDENT" AND
                       AttrDesc
                  |    TEMPLATE4 GETALLCLASS = "ListOfIDENT" AND
                       AttrDescBeforeC BELONGTOCOMPOUND {...}
                  |    TEMPLATE4 GETALLCLASS = "ListOfIDENT" AND
                       AttrDescBeforeC BELONGTOCOMPOUND AND CAttrDesc
                  |    TEMPLATE4 GETALLCLASS = "ListOfIDENT"
                       AND BELONGTOCOMPOUND {...}
                  |    TEMPLATE4 GETALLCLASS = "ListOfIDENT"
                       AND BELONGTOCOMPOUND AND CAttrDesc

/*-----------------Template5 BATCH mode rule------------------*/

batchTemplate5    ::   NUMBER {...} TEMPLATE5 {...} NUMBER {...}
                       batchTemplate55

batchTemplate55   ::   ε
                  |    LINKEDBY "ListOfIDENT" {...} batchTemplate555

batchTemplate555  ::   ε
                  |    WITH NUMBER {...}

/*-----------------Template5 ASSISTED mode rule------------------*/

assistTemplate5   ::   TEMPLATE5 {...}
                  |    TEMPLATE5 NUMBER {...}
                  |    TEMPLATE5 NUMBER LINKEDBY "ListOfIDENT" {...}
                  |    TEMPLATE5 NUMBER LINKEDBY "ListOfIDENT" WITH
                       NUMBER {...}

/*-----------------Template6 BATCH mode rule------------------*/

batchTemplate6    ::   NUMBER {...} TEMPLATE6 {...} NUMBER {...}
                       batchTemplate66

batchTemplate66   ::   ε
                  |    LINKEDTO NUMBER {...} batchTemplate666

batchTemplate666  ::   ε
                  |    BY "ListOfIDENT" {...}

/*-----------------Template6 ASSISTED mode rule------------------*/

assistTemplate6   ::   TEMPLATE6 {...}
                  |    TEMPLATE6 NUMBER {...}
                  |    TEMPLATE6 NUMBER LINKEDTO NUMBER {...}
                  |    TEMPLATE6 NUMBER LINKEDTO NUMBER BY
                       "ListOfIDENT" {...}

/*-----------------Template7 BATCH mode rule------------------*/

batchTemplate7    ::   NUMBER {...} TEMPLATE7 {...} NUMBER {...}
                       batchTemplate77
batchTemplate77   ::   ε
                  |    OR NUMBER {...}

/*-----------------Template7 ASSISTED mode rule------------------*/

assistTemplate7   ::   TEMPLATE7 {...}
                  |    TEMPLATE7 NUMBER {...}
                  |    TEMPLATE7 NUMBER OR NUMBER {...}

/*-----------------Template8 BATCH mode rule------------------*/

batchTemplate8    ::  NUMBER {...} TEMPLATE8 {...} NUMBER {...}
                      batchTemplate88

batchTemplate88   ::  ε
                  |   RESTRICTEDTO batchAttrDesc

/*-----------------Template8 ASSISTED mode rule------------------*/

assistTemplate8   ::   TEMPLATE8 {...}
                  |    TEMPLATE8 NUMBER {...}
                  |    TEMPLATE8 NUMBER RESTRICTEDTO AttrDesc
```

# Appendix B - IQS API

This appendix introduces some design issues which are implementation related. A more detailed description is provided in the Technical Reference document.

## B.1 Data Types

The C data types implementing the Dynamic Array definition as stated in section **7** (**IQS data structure design**) are presented in Figure 11. These C data types are very important to the Parser State implementation because they are the ones upon which the most important information fields of the Parser State structure are implemented.

```
typedef ObjID unsigned long int;
typedef AOID unsigned long int;

// a set of ObjIDs
typedef struct {
    long int index;
    ObjID FAR *objids;
} ObjID_list;

// a set of AOIDs
typedef struct {
    long int index;
    AOID FAR *aoids;
} AOID_list;


// a set of names (of attributes, facets, links, etc)
typedef struct {
    long int index;
    char FAR *names;
} Name_list;

// a set of sets of names (of attributes, facets, links, etc)
typedef struct {
    long int index;
    Name_list FAR *list;
} Name_list_list;

// a set of values (of attributes, facets, links, etc);
// main diference between Name_list is the distance field
typedef struct {
    long int index;
    int distance;
    char FAR *info;
} Array_list;

// a resolved query is a pair (querytext, queryaoids) with
// queryaoids being a set of type AOID_list
typedef struct {
    char FAR * querytext;
    AOID_list queryaoids;
} ResolvedQuery;
```

```
// an History is a set of resolved queries
typedef struct {
    long int index;
    ResolvedQuery FAR * queries;
} Query_list;
```

Figure 11 - IQS C data types implementing the Dynamic Array concept.

## B.2 Functions

The functions making the basic IQS API are the ones in charge with retrieving objects from the repository during query solving. They are strongly based in the use of functionalities provided by other modules[8].

---

```
int iqsGetHierarchyAoids(AOID_list FAR *aoid_list,
                               char FAR *class)
```

Given a `class` name, this function puts in `aoid_list` all the AOIDs of the class sub-hierarchy starting at `class`. Based on `eraGetObj` calls for each class bellow the one provided, `iqsGetHierarchyAoids` will remove, from `aoid_list`, the *system-object* TUTTO (used by Comparator-Modifier as a upper-bound to close the lattice - see [CM-1.4 1993]), if found during the search.

Return values:

- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no objects found for the selected class hierarchy;
- `IQS_SUCCESS` - operation successful.

Secondary effects:

- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list`

---

```
int iqsGetAOGAttribs (AOID_list FAR *aoid_list,
                      Name_list FAR *aog_attrs)
```

[8]see section **8 IQS architecture**.

`iqsGetAOGAttribs` will search the generic attributes for whom the objects in `aoid_list` define a value, that is, for each object in `aoid_list`, the "`AOG`" class is inspected via `eraGetObject` in order to check if each generic attribute has a well-defined non-empty value. As soon as a value has been found for all the generic attributes, the search is stopped (this could happen at the very first object of `aoid_list` if this object defines a non-empty value for all of the generic attributes). The defined generic attributes (except "`AOID`") are returned via the *in/out* parameter `aog_attrs`.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no generic attributes defined (except "`AOID`");
- `IQS_SUCCESS` - operation successful.

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aog_attrs`.

---

```
int iqsGetAOGValues(AOID_list FAR *aoid_list,
                    Array_list FAR *attr_values)
```

For each object in `aoid_list`, `iqsGetAOGValues` inspects the "`AOG`" class via `eraGetObject`, checking for the value the generic attribute passed in `attr_values->info` assumes. The goal is to make `attr_values` the set of those values.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - generic attribute `attr_values->info` undefined;
- `IQS_SUCCESS` - operation successful.

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `attr_values`.

---

```
int iqsGetFacets(AOID_list FAR *aoid_list,
                 Name_list FAR *facets)
```

`iqsGetFacets` will search the facets for whom the objects in `aoid_list` have a value defined, that is, for each object in `aoid_list`, the '`FACETS`" class is inspected via `eraGetObject` in order to check if each facet has a well-defined non-empty value. As soon as a value has been found for all the facets, the search is stopped (this could happen at the very first object of `aoid_list` if this object defines a non-empty value for all of the facets). The defined facets are returned via the *in/out* parameter `facets`.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no facets defined;
- IQS_SUCCESS - operation successful.

Secondary effects:
- IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans facets.

---

```
    int iqsGetFacetsValues(AOID_list FAR *aoid_list,
                                   Array_list FAR
*facet_values)
```

For each object in aoid_list, iqsGetFacetsValues inspects the "FACETS" class via eraGetObject, checking for the value the facet in facet_values->info assumes. The goal is to make facet_values the set of those values.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - facet facet_values->info undefined;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans facet_values.

---

```
    int iqsGetClassAttributes(AOID_list FAR *aoid_list,
                          Name_list_list FAR
*class_atts_list)
```

iqsGetClassAttributes will search the class attributes for whom the objects in aoid_list define a value, that is, for each object in aoid_list, the "AOG" class is inspected via eraGetObject in order to retrieve the value of the "CLASS" generic attribute; the class whose name is given by that value is then inspected, once again using eraGetObject, and all its attributes, having a well-defined non-empty value, are retrieved into a set of names; this set is object specific and so this task must always be done for every object of aoid_list. Since a set of class attributes is eventually needed for each object, the *in/out* parameter, `class_atts_list`, is a set of set of names.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no class attributes defined;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans class_atts_list.

---

```
    int iqsGetAttribsValues(AOID_list FAR *aoid_list,
                            Array_list FAR
*attr_values)
```

For each object in aoid_list, iqsGetAttribsValues inspects the "AOG" class via eraGetObject, checking for the value of the "CLASS" generic attribute; the class whose name is given by that value is then inspected, once again using eraGetObject, in order to retrieve the value of the class attribute originally contained in attr_values->info. The goal is to make attr_values the set of the values obtained that way.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - class attribute attr_values->info undefined;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans attr_values.

---

```
    int iqsGetSLCs(AOID_list FAR *aoid_list,
                   Name_list FAR *slcs)
```

For each object in aoid_list, iqsGetSLCs inspects the "PRJ" class via eraGetObject, checking for a well-defined non-empty value of the "SLC" (Software Life Cycle) attribute. At the end, slcs will contain the Software Life Cycles retrieved that way.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;

- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no software life cycles defined;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
• IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans slcs.

---

```
int iqsGetPHAs(AOID_list FAR *aoid_list,
               Name_list FAR *phas,
               Name_list FAR *slcs)
```

Firstly, iqsGetSLCs is called in order to get into slcs the Software Life Cycles of the aoid_list objects. After that, iqsGetPHAsBySLC will check, for each Software Life Cycle, his specific Software Life Cycle Phases. At the end, phas will contain the Software Life Cycles Phases retrieved that way.

Return values:
• IQS_PARAMERR - bad parameters; operation aborted;
• IQS_NOMEMORY - not enough memory; operation aborted;
• IQS_ERROR - internal or unknown error; operation aborted;
• IQS_NOTFOUND - no software life cycles or no phases defined;
• IQS_SUCCESS - operation successful;

Secondary effects:
• IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans phas and slcs.

---

```
int iqsGetPHAsBySLC(Name_list FAR *phas_list,
                    char FAR *slc)
```

Given a Software Life Cycle slc, conGetSLCPHA is invoked in order to retrieve all the Software Life Cycle Phases of that Software Life Cycle into phas_list.

Return values:

• IQS_PARAMERR - bad parameters; operation aborted;
• IQS_NOMEMORY - not enough memory; operation aborted;
• IQS_ERROR - internal or unknown error; operation aborted;
• IQS_NOTFOUND - no phases found for the software life cycle slc;
• IQS_SUCCESS - operation successful;

Secondary effects:
• IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans phas_list;

```
int iqsGetAoidsBySLC(AOID_list FAR *aoid_list,
                     char FAR *slc)
```

For each object in `aoid_list`, `iqsGetAoidsBySLC` inspects the "`PRJ`" class via `eraGetObject`, checking for the value of the "`SLC`" (Software Life Cycle) attribute. At the end, `aoid_list` will keep only the objects for whom the value of the "`SLC`" attribute equals the `slc` parameter.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOAOIDSSLCS` - no objects found with any software life cycle;
- `IQS_NOAOIDSSLC` - no objects found with `slc`;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOAOIDSSLCS`, `IQS_NOAOIDSSLC`: cleans `aoid_list`.

---

```
int iqsGetSLCsByPHA(Name_list FAR *slcs_list,
                    char FAR *pha)
```

For each Software Life Cycle in `slcs_list`, calls `iqsGetPHAsBySLC` retrieving all its Phases. Then, it checks if `pha` is among those Phases. At the end, `slcs_list` will keep only those Software Life Cycle *containing* `pha`.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no software life cycles found with `pha`;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `slcs_list;`

---

```
int iqsGetCompounds(AOID_list FAR *aoid_list)
```

For each object in aoid_list, iqsGetCompounds calls conGetMbr once, verifying if it returns A_SUCCESS, in wich case the object is assumed to be a compound object. At the end, aoid_list will keep only those objects which passed the previous test, that is, those objects being compounds.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no compounds found;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans aoid_list.

---

```
int iqsGetCaractRel(AOID_list FAR *aoid_list,
                    Name_list FAR *crls)
```

For each object in aoid_list, iqsGetCaractRel calls conGetMbrLnk in order to retrieve a set of ObjIDs, each one standing for a Characteristic Relation. conGetLnk will then allow for each one of those ObjIDs to be maped into a string: the name of the Characteristic Relation. In the end, crls will contain set of Characteristic Relation names retrieved as described.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no characteristic relations found;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans crls.

---

```
    int iqsGetClustersByCaractRel(
                                AOID_list FAR
*aoid_list,
                                char FAR *crl)
```

For each object in aoid_list, iqsGetClustersByCaractRel calls conGetMbrLnk in order to retrieve a set of ObjIDs, each one standing for a

Characteristic Relation. `conGetLnk` will then allow for each one of those `ObjIDs` to be mapped into the name of the respective Characteristic Relation. If the parameter `crl` matches at least one of these Characteristic Relations, then the object currently under survey is considered to be a Cluster (being `crl` one of his Characteristic Relation). In the end, `aoid_list` will keep only the objects being Clusters.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no clusters found with any Characteristic Relation;
- `IQS_NOVALUES` - no clusters found with the Characteristic Relation `crl`;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`, `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`, `IQS_NOVALUES`: cleans `aoid_list`.

---

```
int iqsGetClaoByMember(AOID_list FAR *aoid_list)
```

The objects that aggregate the ones in `aoid_list`, are retrieved and placed there. `conGetMbr` is the low-level functionality on which `iqsGetClaoByMember` mainly relies for that purpose.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or uknown error; operation aborted;
- `IQS_NOTFOUND` - no compounds found;
- `IQS_SUCCESS` - operation successful.

Secondary effects:
- `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list`.

---

```
int iqsGetMemberByClao(AOID_list FAR *aoid_list)
```

For each object in `aoid_list`, `iqsGetMemberByClao` calls `conGetMbr`, retrieving all his members. At the end, `aoid_list` will be the set of all the objects contained by the ones initialy there.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;

- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no compounds found;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list`.

```
int iqsGetSources(AOID_list FAR *aoid_list)
```

For each object in `aoid_list`, `iqsGetSources` calls `conGetLnk`, in order to check if the current object is source of some link. At the end, `aoid_list` will keep only the source objects.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no sources found;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list`.

```
    int iqsGetSourcesAndLinks(AOID_list FAR *aoid_list
                              Name_list_list FAR
*links_set_list)
```

For each object in `aoid_list`, `iqsGetSourcesAndLinks` calls `conGetLnk`, in order to check if the current object is source of some link. If so, the set of all the outgoing links from that object is retrieved. At the end, `aoid_list` will keep only the source objects and `links_set_list` will contain the respective sets of outgoing links.

Return values:
- `IQS_PARAMERR` - bad parameters; operation aborted;
- `IQS_NOMEMORY` - not enough memory; operation aborted;
- `IQS_ERROR` - internal or unknown error; operation aborted;
- `IQS_NOTFOUND` - no sources found;
- `IQS_SUCCESS` - operation successful;

Secondary effects:
- `IQS_PARAMERR`: cleans `links_set_list`;
- `IQS_NOMEMORY`, `IQS_ERROR`, `IQS_NOTFOUND`: cleans `aoid_list` and `links_set_list.`

```
    int iqsGetSourcesByLinkAndSinks(
                                AOID_list FAR
*aoid_list,
                                char FAR *link,
                                AOID_list FAR *sinks)
```

For each object in aoid_list, iqsGetSources calls conGetLnk, in order to check if the current object is source of link to at least one sink in sinks. At the end, aoid_list will keep only the objects founded to be sources in this way.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no sources found;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans aoid_list and sinks.

```
    int iqsGetSourcesAndLinksBySinks(
                      AOID_list FAR *aoid_list,
                      Name_list_list FAR
*links_set_list,
                      AOID_list FAR *sinks)
```

For each object in aoid_list, iqsGetSources calls conGetLnk, in order to check if the current object is source of some link to some sink in sinks. If so, the set of all the outgoing links from that object to all the sinks is retrieved. At the end, aoid_list will keep only the source objects and links_set_list will contain the respective sets of outgoing links to the at least one of the sinks.

Return values:
- IQS_PARAMERR - bad parameters; operation aborted;
- IQS_NOMEMORY - not enough memory; operation aborted;
- IQS_ERROR - internal or unknown error; operation aborted;
- IQS_NOTFOUND - no sources found;
- IQS_SUCCESS - operation successful;

Secondary effects:
- IQS_PARAMERR, IQS_NOMEMORY, IQS_ERROR, IQS_NOTFOUND: cleans aoid_list and sinks.