UNIVERSIDADE DO MINHO

Escola de Engenharia

Relatório de Estágio

da

Licenciatura em Engenharia de Sistemas e Informática

Projecto **SOUR**
(**SO**ftware **U**se and **R**euse)

- Implementação do Intelligent Query System -

José Carlos Rufino Amaro
Departamento de Informática da Universidade do Minho

Supervisores

Engº Fernando Mário Martins
Prof. José Nuno Oliveira

Braga
Setembro de 1994

Agradecimentos

Agradeço ao Engº Fernando Mário Martins pela confiança depositada em mim ao convidar-me a participar num projecto da envergadura do SOUR. Uma palavra de apreço também para o Dr. José Nuno Oliveira, com quem foi sempre gratificante discutir as questões do SOUR.

A equipe que encontrei no terreno, constituida pelo António Nestor, Luis Neves, e Filipe Marques, foi inestimável na ajuda que me proporcionou nos primeiros (sempre difíceis) contactos com o SOUR, na assistência prestada durante o resto do trabalho, e ainda durante a elaboração do relatório.

Agradeço também o apoio prestado por Rui Mendes e Paulo Romualdo.

And last, but not least, agradeço aos meus pais, António e Ana. Sem o seu encorajamento constante, não teria a minha licenciatura chegado a bom porto.

Braga, 27 de Setembro de 1994

José Carlos Rufino Amaro

Índice

Parte I

1 Objectivo

O objectivo deste estágio era o de codificar uma DLL, para o WINDOWS 3.1, na linguagem de programação C, a fim de implementar o sub-sistema *Intelligent Query System* (IQS) do projecto SOUR. Tal implementação deveria ser baseada na documentação produzida durante a concepção e *design* do IQS bem como num protótipo dessa ferramenta, desenvolvido em Camila.

1.1 Local do Estágio

O Estágio foi levado a cabo nas instalações do Departamento Informático da Universidade do Minho, em Gualtar, nomeadamente no Laboratório de Tecnologia da Programação, onde a equipe do SOUR desenvolve maioritariamente a sua actividade.

2 Estrutura do Relatório

A estrutura do resto do relatório reflecte o facto de que uma parte razoável constitui documentação oficial do SOUR, nomeadamente a *Functional Specification & Architecture* [IQSFSA94] e a *Technical Reference* [IQSTR94] do *Intelligent Query System* (IQS). Disposições contratuais obrigaram ainda a que fosse o Inglês a língua com base na qual foram redigidos esse documentos, que serão apresentados em anexo.

Alguma dessa documentação (em concreto [IQSFSA94]) vem no seguimento do trabalho realizado pelo Engº António Nestor Ribeiro durante o seu estágio de Licenciatura e que consistiu, grosso modo, na concepção da arquitectura do IQS.

O presente relatório aborda o processo de implementação de dita arquitectura, tomando como ponto de partida a Especificação e Protótipo em Camila (implementando funcionalidades mínimas), e a descrição de uma API básica para o IQS.

Após uma ligeira introdução que pretende situar o SOUR no contexto das ferramentas CASE, continua-se, descrevendo de uma forma breve a arquitectura do SOUR como um todo, a fim de tornar claro o papel do subsistema IQS.

A seguir, o IQS é apresentado, ainda sem entrar em detalhes de implementação. Esta descrição do IQS baseia-se em grande parte na informação que a *Functional Specification & Architecture* encerra, abordando aquilo que se espera do IQS bem como os aspectos de desenho mais importantes.

Descrevem-se então os passos fundamentais do processo que conduziu à implementação da **iqs.dll**, considerando a investigação feita, alguns compromissos assumidos, decisões tomadas, etc. Não pretende ser, de forma alguma, uma descrição exaustiva, mas sim algo que traga à superfície aquilo que foi mais relevante durante a implementação. Como [IQSTR94] oferece descrições mais detalhadas sobre estas matérias, os esclarecimentos dos pormenores de implementação mais importantes serão inevitavelmente redireccionados para esse documento, em anexo.

Ainda antes dos anexos, serão apresentadas algumas direcções em que trabalho futuro poderá ser realizado a fim de enriquecer a funcionalidade do IQS.

Os anexos são exclusivamente constituidos pela *Functional Specification & Architecture* e pela *Technical Reference* do IQS. A *Functional Specification & Architecture* cobre os aspectos de *design* e arquitectura do IQS e a T*echnical Reference* debruça-se sobre diversos aspectos de implementação e manutenção. Ambos são documentos oficiais do SOUR.

2.1 Convenções

Termos de origem anglo-saxónica integrados em texto em português serão escritos em *itálico*.

Transcrições obedecerão ao formato "`Transcrição`".

Código fonte usará o tipo `código fonte`.

Conceitos importantes e nomes de ficheiros serão escritos em **bold**.

3 Introdução

Da brochura de apresentação do SOUR, [EU89], extraiu-se o seguinte comentário:

"`When sugar first reached Europe, demand soon exceeded supply by such a margin that it could be sold for its own weight in silver. Today, a similar shortfall characterises the market for computer software.`"

Estas palavras evidenciam um problema com que actualmente se debatem as organizações para as quais o uso de meios informáticos, correndo *Software* adequado às suas necessidades, se tornou vital.

A integração da Informática no quotidiano das organizações, trazendo consigo o tratamento automático (e eventualmente em tempo real), de grandes volumes de informação, tornou-se não apenas um factor competitivo e de diferenciação, como até um factor de sobrevivência.

Naturalmente, ambos os meios intra e extra-organizacionais sofrem mutações, e essa evolução implica um esforço de adaptação constante. Uma boa parte desse esforço

recai sobre os Departamentos Informáticos, sujeitos a pressões constantes. É também sintomático o aumento do volume da carteira de encomendas a *Software Houses* e Empresas de Consultadoria.

No meio deste panorama, ferramentas que acelerem o ciclo de desenvolvimento de Sistemas de Informação, são sempre bem-vindas. As ferramentas surgidas no âmbito daquilo que comummente se designa por CASE (*Computer Aided Software Engineering*), têm obtido um grau de satisfação variável junto do utilizadores.

A Implementação é uma das fases que, com maior ou menor grau de importância atribuida, podemos encontrar de forma geral em quase todas as metodologias.

Compreensivelmente, é a este nível que as oportunidades de reutilização, nomeadamente de componentes de *Software*, se afiguram mais evidentes. A prática seguida vem, aliás, corroborar este facto: segundo [EU89], pensa-se que actualmente a taxa de reutilização ronde os 25%.

Com efeito, reutilizar o estudo (ou partes) de uma organização torna-se mais complexo e difícil, não só porque não há duas organizações iguais, como também devido ao factor de subjectividade que qualquer Análise abstracta (e consequente modelo) introduz. Contudo, como veremos, o SOUR também pretende dar o seu contributo nesta área, não se limitando a oferecer os seus préstimos apenas no que diz respeito à Implementação.

Relativamente à reutilização durante a fase de Implementação, e pensando por exemplo em termos de *Software* (repare-se que em termos de soluções de *Hardware*, a reutilização é regra geral menos problemática, porque um determinado parque informático pode ser a base material de diversas soluções, até para problemas diferentes ...), a granularidade dessa reutilização é variável: desde um simples pedaço de código interior a uma função, passando pela própria função, bibliotecas ou módulos, até uma aplicação inteira.

Naturalmente, esta classificação não é (e nem pretende ser) formal, exaustiva e completamente enumerativa. Pretende-se apenas sugerir que a reutilização, mesmo nos moldes em que tem vindo a ser efectuada é tarefa complexa e altamente dependente da visão que os intervenientes num projecto de Software têm sobre o(s) objecto(s) a reutilizar.

O SOUR, surge assim como uma ferramenta da vasta família CASE, fundamentalmente ligado à gestão e reutilização dos mais diversos objectos associados (e associáveis) a qualquer projecto de Sistemas de Informação.

Parte II

4 O SOUR

4.1 Introdução

O projecto SOUR (acrónimo para *SOftware Use And Reuse*) nasce, oficialmente, em 1989, duma iniciativa conjunta de algumas empresas do consórcio Olivetti (Systena SPA, Syntax Sistemi Software e OIS Ricerca) e do INESC. Desenvolvendo uma representativa actividade no campo dos Sistemas de Informação, estas organizações adquiriram uma sensibilidade especial relativamente aos benefícios da Reutilização.

Integrado num programa mais vasto (o programa EUREKA, com base em fundos comunitários), o SOUR, oficialmente designado por EU 379, propõe-se, nas palavras do Dr. Francesco Fusco, *project's lead partner*, atingir os seguintes objectivos informais [EU89]:

- "...to extend re-use techniques to the whole development process.", *i.e.*, para além da Implementação, levar a reutilização à Especificação, Análise e *Design* de Sistemas de Informação;

- "...transforming re-use from a handicraft activity to an industrial procedure...", *i.e.*, elevar o grau e eficiência da Reutilização, procurando convertê-la numa actividade mais racional e mecanizada;

- "The construction of tools to facilitate the capture, manipulation and retrieval of this [know-how of the individual programmers and system analysts who work for them]...", ou seja, o desenvolvimento de ferramentas capazes de gerir o conhecimento que só a experiência traz e ao qual faz falta um tratamento sistemático.

Em traços gerais, podemos então dizer que o SOUR há-de ser capaz de levar a cabo tarefas de:

- Classificação;

- Pesquisa;

- Comparação;

- Visualização;

por forma a diminuir o esforço de Reutilização e ao mesmo tempo assumindo-se como um mecanismo de unificação de visões distintas sobre o repositório de um projecto de Sistemas de Informação, visões essas resultantes da aplicação de Metodologias diferentes.

4.2 Arquitectura do SOUR

A Figura 1 visualiza, de uma forma esquemática, a arquitectura geral do SOUR [SPS93].

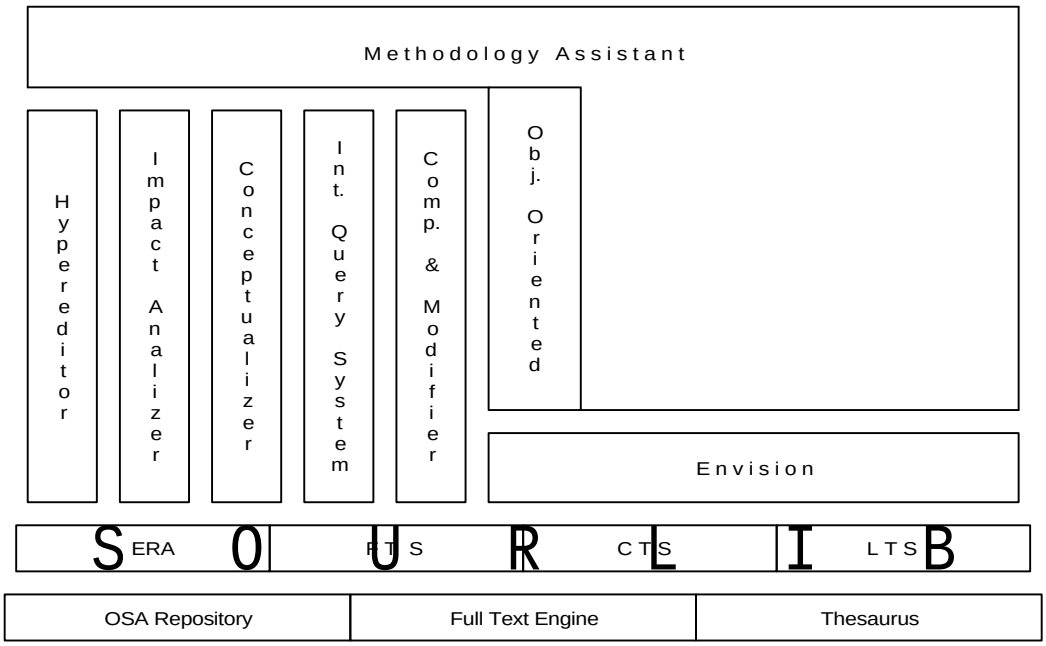

Figura 1 - arquitectura geral do SOUR.

Tal arquitectura distribui-se essencialmente por 3 camadas:

- Gestores Físicos (*Physical Managers*);
- SOURLIB, um conjunto de APIs que fornecem a *interface* com os Gestores Físicos;
- Nível Aplicativo, constituido pelas ferramentas desenvolvidas e que permitem ao utilizador aceder, ainda que indirectamente (via SOURLIB), aos serviços oferecidos pelos Gestores Físicos.

Segue-se uma breve descrição dos componentes mais representativos dessas camadas.

4.2.1 Gestores Físicos

4.2.1.1 OSA Repository (OSA/R)

O *OSA Repository* é parte integrante da *OSA Architecture* da OLIVETTI (é, por assim dizer, o seu *corporate repository product*) e é também o repositório que o SOUR usa para guardar a informação.

Entre outros aspectos, o OSA/R caracteriza-se por [ANR93]:

- combinar o *standard* oferecido pelo modelo Entidades-Relações (ou modelo ER) com a versatilidade da classificação por objectos (objectos, classe e herança são conceitos oferecidos de base no OSA/R);
- tornar fácil a expansão do Modelo de Informação, num dado momento embebido no repositório, através da introdução de novos objectos e relações;
- permitir o uso de diferentes ferramentas CASE, sempre que estas disponibilizem a sua própria API de acesso ao repositório;

Para além destas, o OSA/R goza das habituais propriedades que os repositórios devem oferecer, nomeadamente: persistência (a informação é conservada durante um período de tempo arbitrário após ter sido depositada no repositório), partilha (acesso simultâneo por vários processos), suporte a transacções[1];

Em resumo, o OSA/R assume-se como um repositório capaz de suportar diferentes esquemas lógicos montados por cima do seu esquema físico, o que se revela fundamental para o SOUR dado que pretende ser uma ferramenta independente da Metodologia usada. Os esquemas lógicos serão construídos sobre o repositório físico com base nas funções que são oferecidas pela *interface* do repositório.

4.2.1.2 Full Text Engine (FTE)

O *Full Text Engine* foi concebido com o objectivo de enriquecer as capacidades de pesquisa do IQS. Entretanto, na actual versão do SOUR, ainda não se procedeu à sua integração dessa ferramenta.

Entre outras potencialidades, o FTE deveria ser capaz de proporcionar [SPS93]:

- operadores relacionais entre termos de uma pesquisa;
- operadores de adjacência;
- consulta a dicionários.

As funcionalidades do FTE seriam fornecidas através da API de um *Full Text Server* (FTS), parte integrante da SOURLIB.

[1]uma *transacção* implica a execução de um conjunto de acções indissociáveis; se pelo menos uma dessas acções não é executada, então a *transacção* não é considerada efectuada.

4.2.1.3 Thesaurus

Apesar de se situar ao nível dos Gestores Físicos, o *Thesaurus* por si só não implementa funcionalidades de gestão. Antes, é uma estrutura física que conserva um conjunto de termos, sucessivamente acumulados pelo SOUR. O *Lexicon Thesaurus Subsystem* (LTS)[2] é o componente da SOURLIB que confere estrutura ao *Thesaurus* (já que monta em cima dele um esquema lógico próprio), e fornece serviços de acesso ao seu nível físico.

4.2.2 Sourlib

Sendo um ferramenta CASE, e de acordo com o que foi dito sobre o OSA/R, o SOUR, deverá providenciar a sua própria API para aceder ao repositório (bem como aos outros componentes da camada dos Gestores Físicos). No contexto SOUR, existem não uma mas várias APIs. Esse conjunto de bibliotecas constitui a SOURLIB.

A SOURLIB, como *software layer* de interposição entre o estrato físico e os módulos aplicativos do SOUR que a ele pretendam aceder, deverá fornecer serviços capazes de [ANR93]:

- esconder os detalhes de implementação da *physical layer*, protegendo os utilizadores de eventuais mudanças na tecnologia de implementação (fornece pois um mecanismo de encapsulamento);

- proteger a própria estrutura física do Repositório, bem como o FTE e o *Thesaurus* de accessos não *standard*, providenciando uma espécie de protocolo de acesso a esse níveis, a usar de uma forma transparente e universal pelas camadas superiores. Tal será conseguido fornecendo funções que implementam em termos de chamadas aos níveis físicos a estrutura lógica dos mesmos.

Isto está de acordo com o que se disse sobre a possibilidade que o OSA/R oferece de suportar diferentes modelos lógicos de informação sempre e quando uma API de acesso, própria, seja desenvolvida.

A SOURLIB encontra-se dividida em bibliotecas homogéneas que a seguir se passam a descrever.

4.2.2.1 Easy Repository Access (ERA)

O ERA é o sub-sistema da SOURLIB que chama a si a responsabilidade de implementar um esquema lógico de representação e classificação dos objectos guardados no repositório, independentemente da estrutura física interna adoptada por este.

O nível OSA/R, é desta forma encapsulado, o que lhe permite ser acedido pelos Níveis Aplicativos, de uma forma totalmente transparente. As

[2]para mais pormenores sobre o LTS consultar **4.2.2.4 Lexicon Thesaurus Subsystem (LTS)**.

funcionalidades da ERA API assumem-se pois como a única forma garantidamente validada de acesso à informação depositada no repositório.

Os esquemas de classificação do ERA são *object-oriented*. Consequentemente, o ERA considera todas as entidades que lhe são submetidas como sendo objectos abstractos (*Abstract Objects* ou AOs), resultantes de um processo de Conceptualização[3].

Para o ERA, os objectos podem ser [CON93]:

- objectos físicos (*Physical AOs* ou PAOs): ficheiros de um *filesystem* (*Filesystem PAOs* ou FPAOs) ou objectos embebidos em estruturas físicas (*Embebbed PAOs* ou EPAOs) como é o caso de diagramas Envision, tabelas EXCEL, etc;

- objectos lógicos (*Logical AOs* ou LAOs): qualquer objecto sem existência física.

Os objectos lógicos podem ainda ser compostos (*Composite LAOs* ou CLAOs), ou seja, agregar pelo menos dois objectos lógicos. O critério de agregação é variável, o que introduz vários tipos de objectos compostos (ver Figura 2):

- os *containers*: limitam-se a estabelecer uma relação simples de inclusão com os seus membros, ignorando possíveis relações que estes possam ter entre si;

- os clusters: agregam objectos que para além de terem em comum a propriedade de pertencerem a um mesmo CLAO, têm entre si relações características (ou *links*).

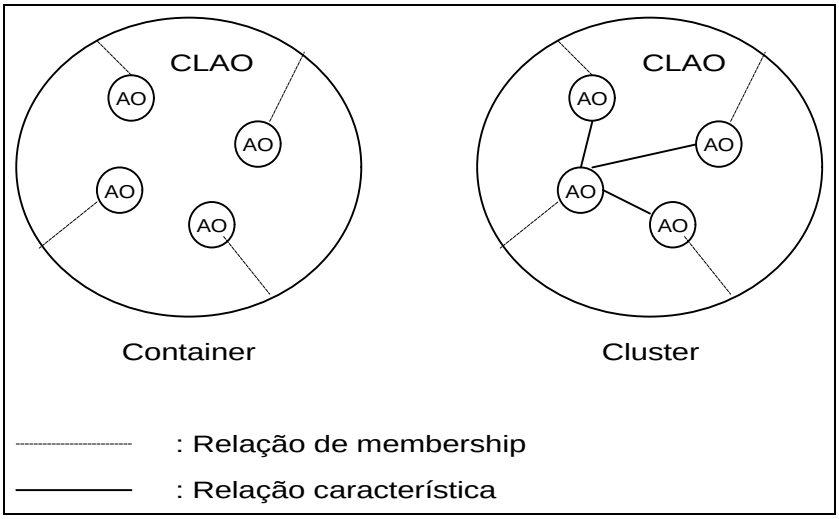

Figura 2 - *Composite Logical Objects (CLAOs)*

[3]consultar a secção **4.2.3.3 Conceptualizer** para mais detalhes sobre o processo de Conceptualização.

Para além do caso particular dos *clusters*, o MAO (*Model of Abstract Objects)* assumido pelo ERA admite ainda a existência de relações (ou *links*) mais gerais, entre quaisquer dois AOs. Tais *links* são, de *per si*, AOs com direito a uma classe própria (a classe AOLINK a referir oportunamente).

A representação de um AO pelo ERA é baseada na repartição da informação relevante desse AO ao longo de uma hierarquia de classes, consideradas suficientes para o descrever. A informação sobre cada AO, distribuida por essa hierarquia, serve essencialmente para [CON93]:

- aumentar a capacidade discriminante durante operações de consulta ao repositório;
- permitir a própria gestão do AO por parte do sistema.

A hierarquia implementada pelo ERA está representada na Figura 3.

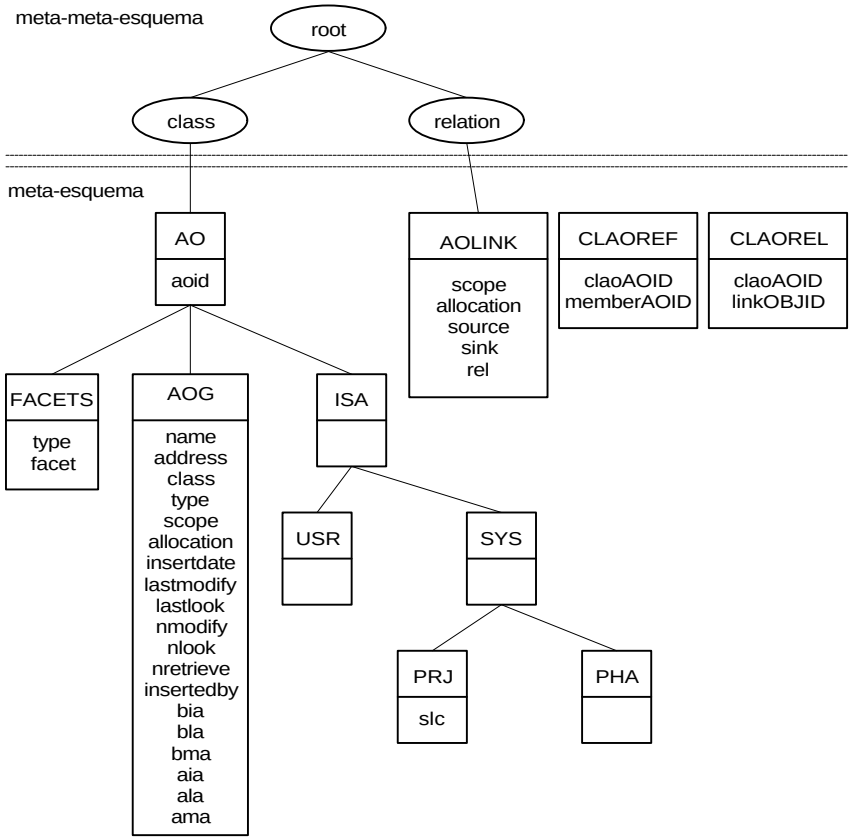

Figura 3 - Hierarquia de descrição de um AO

Assim, o OSA/R oferece de base os conceitos primitivos de classe e relação, sob a forma das classes *class* e *relation* (implementados de forma distinta, tal como o meta-meta-esquema sugere).

Na prática, os utentes do ERA, não lidam directamente com essas duas classes, antes assumindo o meta-esquema directamente derivado do meta-meta-

esquema: a classe *class* deriva numa hierarquia que vai comportar toda a informação atributiva (classe AOG) e facetada [PD87] (classe FACETS) dos AOs e a classe *relation* deriva uma única classe, AOLINK, suficiente para descrever relações gerais (*links*) entre os objectos.

A classe AO, que encabeça a hieraquia derivada de *class*, não disponibiliza desde logo toda a informação sobre um AO (objecto abstracto), antes definindo apenas o seu *Abstract Object Identifier* (AOID), identificador único e universal de qualquer AO (objecto abstracto) inserido no sistema.

Como todas as classes abaixo de AO (classe) herdam o AOID, este acaba por ser o equivalente a uma chave no Modelo Relacional da Informação, permitindo o acesso às classes onde se encontra o resto da informação sobre o objecto abstracto. Portanto, a informação sobre um AO encontra-se particionada pelas várias classes abaixo da classe AO e o AOID é o sustentáculo dessa hierarquia.

A classe CLAOREF foi criada exclusivamente para descrever *Containers* e a classe CLAOREL para descrever *Clusters*.

Nem todas as classes do meta-esquema são instanciáveis (dizem-se por isso classes abstractas ou meta-classes), servindo apenas como *templates* para a definição de outras classes, que herdarão os atributos da classe "mãe". A classe AO é um exemplo típico de uma meta-classe.

4.2.2.2 Full Text Subsytem (FTS)

O FTS é a biblioteca encarregue de fornecer aos Níveis Aplicativos as funcionalidades do FTE. O FTS apresenta-se a esse níveis sob a forma de uma API.

4.2.2.3 Concept Thesaurus Subsystem (CTS)

O CTS é o componente da SOURLIB que toma a seu cargo a gestão de conceitos [ANR93]. Foi criado com o intuito de identificar e quantificar relações semânticas entre os termos de um *Thesaurus*, com base em conceitos associados a esses termos.

O CTS compreende dois tipos de entidades:

- termos *kernel* (ou termos atómicos) do *Thesaurus*: representam conceitos não refináveis; só termos *kernel* do LTS[4] podem ser termos *kernel* do CTS;
- conceitos criados pelo próprio CTS.

Um grafo conceptual directo, pesado e acíclico é um modelo apropriado para descrever as relações entre os conceitos. Os termos *kernel* são as folhas

[4]a propósito do LTS, consultar a seccção **4.2.2.4 Lexicon Thesaurus Subsystem (LTS)**.

desse grafo, pois são conceitos não especializáveis. Os outros termos, são conceitos cada vez mais generalizantes à medida que nos afastamos das folhas. Os arcos, ligando diferentes níveis de conceitos, têm um peso no intervalo]0,1[, *i.e.*, exprimem uma medida percentual de proximidade ou semelhança semântica entre os conceitos que ligam. Assim, a fim de medir a distância conceptual entre dois conceitos, o critério usado baseia-se necessariamente no peso dos arcos do menor caminho entre eles.

O conceito de *fuzziness* que o SOUR disponibiliza em algumas das suas ferramentas, baseia-se nas funcionalidades deste componente da SOURLIB. Lofti Zadeh, fundador da lógica *fuzzy*, descreve-a da seguinte maneira [LZ84]:

"`a kind of logic using graded or qualified statements rather than ones that are strictly true or false`".

4.2.2.4 Lexical Thesaurus Subsystem (LTS)

O LTS é responsável pela aquisição e gestão de conhecimento léxico acumulado pelo SOUR [ANR93], fornecendo as bases para tornar o SOUR independente da linguagem do utilizador e introduzindo assim um certo grau de adaptabilidade.

Apesar de aceder directamente ao *Thesaurus* (suportando inclusivamente vários), o LTS tem uma visão particular sobre o seu conteúdo, que reflecte uma divisão lógica nos seguintes domínios (ou *views*):

- *kernels*;
- *roots*: sinónimos dos *kernels*;
- termos: flexões gramaticais das *roots*.

Uma *root* sem sinónimos é considerada em simultâneo um *kernel* (e portanto *kernels* são casos particulares de *roots*).

Como o CTS só usa termos *kernel* do LTS, então o LTS deve procurar reduzir qualquer termo não-*kernel* ao *kernel* respectivo e passá-lo, se isso for requisitado, ao CTS.

Concluindo, o LTS suporta [SPS93]:

- gestão de termos;
- relações de sinonímia;
- gestão de flexões gramaticais;
- normalização (pre-processamento para classificação);

4.2.3 Nível Aplicativo

Descrevem-se aqui alguns dos módulos baseados nos serviços da SOURLIB.

4.2.3.1 Hypereditor & Hyperbrowser

O HYPEREDITOR propõe-se [SPS93]:

- organizar objectos e texto de uma forma eficiente, gerindo fragmentos relacionados, em vez de estruturas rígidas.
- contribuir com a sua quota parte de informação no processo de Análise.

O HYPERBROWSER é o sistema de apresentação dos documentos criados pelo HYPEREDITOR. A navegação (ou *browsing*) faz-se com base nos links criados pelo HYPEREDITOR entre os diversos componentes (considerados suficientemente representativos para terem individualidade própria) dos documentos.

4.2.3.2 Impact Analyzer

O IMPACT ANALYZER pretende [SPS93]:

- facilitar a percepção do peso que uma mudança no modelo lógico da informação tem nas outras áreas do projecto do Sistema de Software.
- estimar o custo e tempo dispendidos em tais mudanças.

A análise de tais efeitos comporta duas fases distintas:

1. a contrução de uma base de conhecimento específica à Metodologia em questão, envolvendo todos os objectos e relações significativos da Metodologia;
2. o uso de regras de dependência implícitas na base de conhecimento a fim de avaliar o impacto das mudanças.

4.2.3.3 Conceptualizer (CON)

O CONCEPTUALIZER é o sub-sistema responsável pela entrada de informação no repositório, pois é a ferramenta que fornece o *front-end* que permite identificar e descrever os componentes de Software que mais tarde se pretendem. eventualmente reutilizar. A informação relativa aos objectos do mundo exterior, produzida com base numa determinada Metodologia de

Desenvolvimento de Sistemas de Informação, é assimilada e catalogada de acordo com o Modelo dos Objectos Abstractos, fornecido pelo ERA[5]. O nível ERA encarregar-se-á de mapear essa hierarquia nas estruturas físicas do repositório.

Os objectivos gerais do CONCEPTUALIZER são então [SPS93]:

- compreender a natureza do objecto;
- associar acções por defeito a um objecto, a efectuar quando é inserido (pre-processamento) no repositório, reconceptualizado, *i.e.*, modificado (re-processamento) ou visualizado;
- aceitar a criação de *links* entre o objecto a conceptualizar e outros pré-existentes;
- classificar componentes de software, quer por atributos quer por facetas [PD87].

A implementação do CONCEPTUALIZER baseia-se exclusivamente em chamadas a serviços disponibilizados pelas APIs das bibliotecas da SOURLIB, e compreende duas fases fundamentais:

- fase da *identificação*, durante a qual a natureza do objecto, segundo uma descrição fundamentalmente atributiva, é determinada; compreende aspectos como o *pathname*, o formato -ascii, binário, etc-, o editor ou *browser* por defeito, a fase do ciclo de vida do Projecto de Software (que segue uma determinada metodologia) a que pertence, as acções por defeito de pre-processamento ou re-conceptualização, etc;
- fase da *descrição*; consiste basicamente na descrição de relações semânticas, via *links*, com outros objectos, bem como na classificação multifacetada do objecto.

4.2.3.4 Intelligent Query System (IQS)

Se o CONCEPTUALIZER tem a seu cargo o *front-end* de entrada de informação no repositório, é ao IQS que cabe a tarefa de providenciar as funcionalidades e o ambiente adequado à pesquisa e recuperação dessa informação.

O IQS é vital para o SOUR na medida em que se assume como a ferramenta de *browsing* do repositório, capaz de responder a solicitações de pesquisa dos objectos adequados a reutilizar, mediante o fornecimento de descrições com uma semântica mínima.

[5]rever **4.2.2.1 Easy Repository Access (ERA)**.

Sendo o IQS o tema deste relatório, os mais variados aspectos de *Design* e Implementação são discutidos em capítulos próprios (**5** e **6**), pelo que se remete para eles uma análise mais aprofundada desta ferramenta.

4.2.3.5 Comparator & Modifier (CM)

O COMPARATOR chama a si a tarefa de comparar um par qualquer de AOs. O resultado dessa comparação é também um AO, que pode ser sujeito a um processo normal de conceptualização, entrando pois no repositório. No estádio actual de implementação do SOUR, o CONCEPTUALIZER é encarregue dessa conceptualização, mas o MODIFIER deverá também ser capaz de implementar essa funcionalidade.

A comparação entre AOs é feita com base nos seus atributos e na sua descrição multifacetada. Uma comparação com base em facetas permite a adopção de critérios *fuzzy* [PD87], sendo esta a característica que distingue o COMPARATOR de outra ferramentas similares do universo das Bases de Dados.

O MODIFIER, para além de providenciar pela Conceptualização dos AOs que o COMPARATOR produz, tem a seu cargo providenciar pela eliminação de redundâncias no esquema de classificação embebido no repositório, bem como manter a coerência das relações entre os seus objectos sempre que se produz uma entrada nova que possa modifcar a semântica dessas relações.

5 Concepção e Design do IQS

5.1 Introdução

O *Intelligent Query System* (IQS) é a ferramenta do Nível Aplicativo do SOUR que toma a seu cargo a pesquisa do repositório a fim de recuperar objectos que, eventualmente, se pretendam reutilizar.

O interesse de uma ferramenta do género do IQS no contexto do SOUR é por demais evidente se pensarmos que o SOUR , para além de perseguir objectivos de catalogação genérica e gestão dos mais variados objectos associados a um projecto de Sistemas de Informação, pretende também colher o fruto dessa classificação sistemática sob a forma de altas taxas de reutilização.

A posição do IQS na Arquitectura Global do SOUR é evidenciada na Figura 4. As funcionalidades que o IQS porá à disposição quer de utentes do SOUR, quer até de outras ferramentas do Nível Aplicativo, serão implementadas com base nos serviços disponibilizados pelas APIs da SOURLIB.

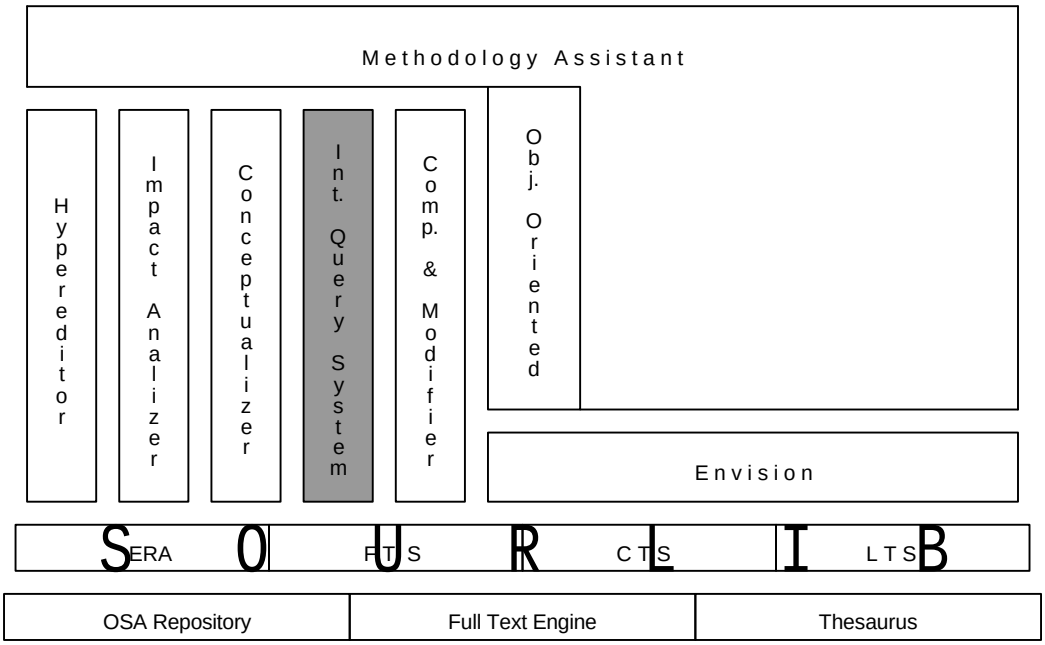

Figura 5 - Um lugar para o IQS na Arquitectura Global do SOUR.

5.2 Objectivos gerais para o IQS[6]

O IQS pretende ser um Sub-Sistema de Query inteligente, o que significa que para além de cumprir com os requisitos gerais de qualquer sistema vulgar de interrogação a um repositório, deverá proporcionar facilidades acrescidas que justifiquem o adjectivo aplicado.

[6]alternativamente, consultar a secção **3 IQS main goals** de [IQSFSA94], em anexo.

Assim, adicionalmente à capacidade de recuperar objectos com base numa descrição que reflicta o esquema lógico do repositório e de trocar informação com outras ferramentas do Nível Aplicativo, o IQS propõe-se:

- assegurar um bom nível de assistência sintáctica e semântica durante o processo de interrogação; uma *interface* permanentemente validado que guie o utilizador durante a construção e resolução de uma *query* é a essência de um dos modos de operação para o IQS: o **Modo Assistido**; outro modo a disponibilizar é o **Modo Batch**, que renuncia a um controlo tão apertado da sintaxe e da semântica da *query*, sendo bastante menos interactivo e por isso adequado à resolução de *queries* em *batch*;

- oferecer adaptabilidade aos utilizadores, permitindo-lhes por exemplo reutilizar os resultados de interrogações ao repositório, quer ainda durante a mesma sessão IQS onde se levaram a cabo essas pesquisas, quer recuperando, em sessões posteriores o trabalho desenvolvido anteriormente; um **Historial** de *queries* conjuntamente com dois modos possíveis de interrogar o repositório (Modo Batch e Modo Assistido) dão pois o seu contributo à prossecução deste objectivo;

- tirar partido da *fuzziness* que o CTS implementa, permitindo a execução de *queries* com um certo nível de incerteza no que toca à descrição dos objectos pretendidos;

- disponibilizar um bom grau de independência da linguagem que o utilizador usa na descrição dos objectos que pretende recuperar; tal facilidade, a ser concretizada, será montada em cima do LTS.

5.3 IQL - uma linguagem de query para o IQS

Uma vez que o IQS deve aceitar *queries*, que não são mais do que descrições dos objectos a recuperar, coloca-se imediatamente a questão da linguagem a usar nessas descrições.

Ora, o Modelo dos Objectos Abstractos (MAO) que o ERA usa para implementar um esquema lógico de classificação montado em cima do repositório, proporciona-nos toda a informação necessária para determinar o leque de *queries* possíveis.

Considerando apenas as *queries* mais representativas[7], podemos sugerir uma gramática de base para as descrever. As *Templates* (ver Figura 6) são uma descrição BNF possível, em termos léxico/sintácticos, daquilo que podem ser as diferentes categorias fundamentais de *queries* bem como as variantes (sub-*queries*) para cada categoria.

[7]no sentido de que cobrem minimamente toda a hierarquia de classes do ERA.

```
// Interface Query Language Kernel Templates
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>* ]

// Interface Query Language non-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>+ ]

// common definitions
<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 6 - As *Templates*

As oito *Templates* propostas subdividem-se, de forma natural, em 2 subconjuntos fundamentais, conforme façam ou não uso dos resultados de *queries* previamente resolvidas:

- *templates* atómicas (ou *kernel)*: independentes de outras *queries*;
- *templates* não-atómicas (ou não-*kernel*): total ou parcialmente dependentes de outras *queries*;

Uma vez aceites as *Templates* como base razoável para uma *Interface Query Language* (IQL), a questão que agora se coloca é determinar até que ponto tal IQL satisfaz as necessidades específicas dos dois modos de operação propostos para o IQS: o Modo Batch e o Modo Assistido. A secção que se segue responde a essas e outras questões.

5.4 Dois modos de operação

5.4.1 O Modo Assitido

O Modo Assistido do IQS pretende impor-se pela permanente assistência sintáctica e semântica durante o processo de interrogação, com base numa forte componente interactiva.

No contexto do Modo Assistido, a **assistência sintáctica** consiste em assegurar, em qualquer instante, que a frase de *query* que está a ser interactivamente construída, não sofre de "deficiências" léxico-sintácticas, ou seja, que só as palavras que a IQL reconhece como válidas, fazem parte da *query* e que elas se combinam de forma correcta para formar tal *query*. Essas palavras (ou *tokens* da IQL) são colocadas à disposição do utilizador a nível da *Interface* do Modo Assistido e sendo assim, o utilizador fica limitado a escolher *tokens* lexicamente correctos, pois só esses lhe são disponibilizados. O momento em que esse *tokens* são disponibilizados implementa o controle sintáctico da construção da frase de *query*.

O Modo Assistido pretende também providenciar **assistência semântica**, *i.e.*, procurar garantir que uma *query* não tenha como resposta um conjunto vazio de soluções. Ora, isto só poderá ser conseguido se o Modo Assistido evitar que o utilizador escolha *tokens*, que apesar de lexicamente correctos e sintacticamente aceitáveis no contexto actual da construção da *query*, não tenham equivalente num conjunto de objectos no repositório. Mais uma vez, há que evitar, antecipadamente, que o utilizador escolha aquilo que não deve.

Concluimos pois que a assistência sintáctica e semântica que o Modo Assistido do IQS promete, só poderá ser implementada com base numa validação rígida do conteúdo e comportamento da sua *interface*. Quando o IQS se encontra no Modo Assistido, todo o evento a nível da *interface*[8], considerado relevante, é convertido num *token*[9] da IQL, equivalente a esse evento. A frase de *query* vai sendo construida por adição sucessiva desses *tokens* e sempre que um novo *token* (ou sequência de *tokens*) é acrescentado à frase de *query*, é preciso determinar o próximo estado da *interface* que ainda garante assistência sintáctica e semântica. É então preciso efectuar o

[8] *click* do rato, selecção de uma linha numa *list-pane*, etc.

[9] ou numa sequência de *tokens*, semânticamente indivisível, *i.e.*, os *tokens* dessa sequência não fazem sentido quando isolados uns dos outros e portanto comportam-se semânticamente como um único agregado.

reconhecimento de toda a frase de *query*, embora só os *tokens* terminais constituam a diferença entre esta frase e a que a precedeu. Do reconhecimento bem sucedido da *query* resulta o eventual[10] refinamento da solução temporária[11] e a transição da *interface* para um novo estado.

Talvez seja oportuno colocar agora a seguinte questão: haverá realmente necessidade de traduzir eventos de *interface* em *tokens* de uma linguagem para depois ter que efectuar o reconhecimento das frases dessa linguagem e ainda ter que fazer com que as acções semânticas resultantes desse reconhecimento se traduzam nas modificações apropriadas da *interface*? Por que não capturar os eventos da *interface* e traduzi-los de imediato nos seus efeitos, sem recorrer a representações intermédias?

Ora, o problema aqui não se resume apenas à representação de informação mas sim essencialmente de fornecer um mecanismo que controle deterministicamente o seu fluxo e que seja simultaneamente capaz de manter a coerência do processo de interrogação.

Do exposto, emerge a ideia de que o comportamento geral do Modo Assistido é governado pela **dependência contextual**, o que acaba por não ser de todo estranho, já que pretendemos construir interactivamente frases de *query* e essas frases são estruturalmente descritas por uma gramática. Como a essa gramática se associa o correspondente autómato determinístico, então podemos modelar o comportamento da *interface* com base nesse autómato, desde que se providencie um mecanismo de comunicação entre os dois (*interface* e autómato). Contudo, esse mecanismo de comunicação depende directamente das particularidades da variante da IQL usada pelo Modo Assistido.

5.4.1.2 Propriedadades da IQL de Modo Assistido

Até agora, poderá ter subsistido a ideia de que uma só IQL seria suficiente para "interrogar o repositório", quer no Modo Batch, quer no Modo Assistido. Como veremos, a IQL fornecida pelas *Templates* é suficiente para cobrir as necessidades do Modo Batch. Para o Modo Assistido, porém, a IQL deve contemplar algumas situações adicionais que resultam directamente das particularidades do comportamento da sua *interface*. Isto sugere que a IQL se deve desdobrar em duas variantes, adaptadas às peculiaridades de cada modo de operação, mas mantendo uma compatibilidade básica que permita um nível satisfatório de integração entre os dois modos.

A IQL a usar no Modo Assistido deve contemplar situações de:

- **redundância**: uma sequência de *tokens* iguais[12] numa frase de *query* deve traduzir-se no mesmo efeito que resultaria da ocorrência de apenas um desses *tokens*; eliminar a redundância de uma frase de *query*, para

[10] como veremos, nem sempre um ou mais *tokens* contribuem para o refinamento da solução.

[11] pois a solução só se considera definitiva quando explicitamente é dada por terminada a síntese interactiva da frase de *query*.

[12] o que facilmente se consegue por exemplo "*clickando*" repetidamente no mesmo botão.

além de a manter compatível com a IQL do Modo Batch, é uma forma de optimizar um eventual recálculo;

- **incomplitude**: pode haver eventos de *interface* que não se traduzem na especialização da solução actual da *query*.

A IQL de Modo Assistido deve além disso usufruir de um nível mínimo de **compatibilidade** com a IQL de Modo Batch a fim de que uma *query* possa, se pretendido, ser recalculada sem ser necessário sintetizá-la de novo interactivamente. Além disso, como veremos, a única forma de recuperar *queries* calculadas numa sessão IQS passada é através do Modo Batch.

Finalmente, a estrutura da gramática para a IQL de Modo Assistido há-de reflectir o facto de que a resolução da *query* apesar de progressiva, alterna com a sua síntese via *interface*, *i.e.*, a frase de query tem de ser novamente reconhecida sempre que como resultado de um evento na *interface* se lhe acrescenta um novo *token* e esse reconhecimento implica eventualmente[13] refinar a solução actual. Donde, esperamos uma gramática altamente particionada, com produções para todas as sub-*queries* possíveis de cada *Template*, ou seja, uma **gramática em escada**[14].

5.4.2 O Modo Batch

O Modo Batch é a alternativa ao Modo Assistido para aqueles utilizadores que já conhecem razoavelmente o modelo lógico do repositório que o ERA implementa e pretendem resolver de uma só vez um "lote" de *queries*, num ambiente menos amigável, é certo, mas que permite também por isso uma resolução mais rápida.

Contudo, uma outra função não menos importante deste modo de operação é a de permitir recuperar **Historiais** de *queries* de sessões anteriores do IQS e recalculá-los, com base no estado actual do repositório. O Modo Batch deve inclusivamente possibilitar a junção entre um Historial importado e o Historial da presente sessão IQS.

Recorde-se que a possibilidade de operar em dois modos distintos e de recuperar o estado de uma sessão IQS passada, são elementos fundamentais de adaptabilidade, um do objectivos do IQS e que se extende ao SOUR em geral.

5.4.2.1 Propriedades da IQL de Modo Batch

A variante *batch* da IQL afigura-se bastante mais simples que a sua congénere *assistida*, pois não precisa de lidar com a alternância patente no Modo Assistido entre a *interface* e o reconhecedor da linguagem de *query*. Logo, a sua gramática não é particionada pois quando o "lote" de *queries* é submetido ao reconhecedor, espera-se que as *queries* respeitem uma das

[13] nem sempre, por causa da incomplitude.

[14] ver **2.4.1 Batch Mode IQL sub-grammar issues** em [IQSTR94], para esclarecimentos mais detalhados sobre esta matéria

formas previstas para cada *Template*. Estas formas não contemplam redundância e/ou incomplitude e por isso chamam-se Formas Mínimas Eficientes, que a serem reconhecidas implicam a sua resolução imediata por inteiro.

5.5 O Historial

Um **Historial** não é mais do que um conjunto de *queries* resolvidas com sucesso. Assim, a definição da Figura 7 é uma descrição conceptualmente aceitável.

Historial ≡ { (t_1, { o_{11}, ..., o_{1m} }), ..., (t_i, { o_{i1}, ..., o_{in} }) }

onde t_i ≡ texto da i'ésima *query*

e o_{in} ≡ n'ésimo objecto que responde à i'ésima *query*

Figura 7 - Estrutura do Historial

De notar que a reutilização dos objectos-solução de uma *query* do Historial só pode ser efectuado desde que o estado do repositório não tenha mudado e com ele a resposta à *query*. É por essa razão que qualquer Historial importado deve ser recalculado[15] a fim de assegurar que reflecte o estado actual do repositório. Sendo assim, ao gravar um Historial não adianta guardar os objectos-solução de uma *query*; só o seu texto é preservado.

O Historial permite ainda uma comutação suave entre os dois modos de operação do IQS, pois o trabalho desenvolvido em qualquer dos modos pode ser conservado quando se transita de um modo para outro. O Historial é pois o elemento unificador entre os dois modos, durante uma sessão IQS, providenciando pela sua total integração.

5.6 Arquitectura do IQS

A arquitectura geral do IQS é aquela que se mostra na Figura 8. Nela se torna evidente a dualidade Modo Batch/Modo Assistido bem como a separação nítida entre a camada de *Interface* e a camada encarregue do reconhecimento de *queries* e consulta ao repositório.

Com base na Figura 8, é possível descrever sumariamente o funcionamento do IQS. Em Modo Assistido, a camada de *Interface* encarrega-se de capturar qualquer evento de interesse para o IQS e eventualmente converte-o num ou mais *tokens* representativos desse evento; de seguida adiciona tal *token(s)* à frase de *query* actualmente em síntese e envia-a ao reconhecedor da variante assistida da IQL. Em Modo Batch, um "lote" completo de *queries* é enviado ao reconhecedor da variante

[15] o que, recorde-se, é feito em Modo Batch.

batch da IQL. O esquema apresenta apenas um reconhecedor porque, como veremos ao discutir os pormenores de implementação, é possível unificar as gramáticas para ambos os modos e partilhar muito do código que implementa as acções semânticas. Em resposta a um reconhecimento bem sucedido, determinadas acções semânticas são executadas, tendentes a recuperar objectos do repositório e determinar qual o próximo estado da *Interface* que ainda garante assistência sintáctica e semântica. As acções semânticas baseiam-se numa IQS API, montada em cima das funcionalidades de outros módulos, entre os quais algumas bibliotecas da SOURLIB. O Estado do Reconhecedor, para além de informação interna relativa ao seu autómato determinístico, alberga a solução temporária da *query*, o Estado da *Interface* (conteúdo e *flags enable-disable* de vários objectos visuais) e o Historial. Finalmente, a camada de *Interface* recupera o controle do fluxo do programa, reflectindo o Estado da *Interface*. Saliente-se ainda o Historial, presença activa sempre que se comuta de modo de operação.

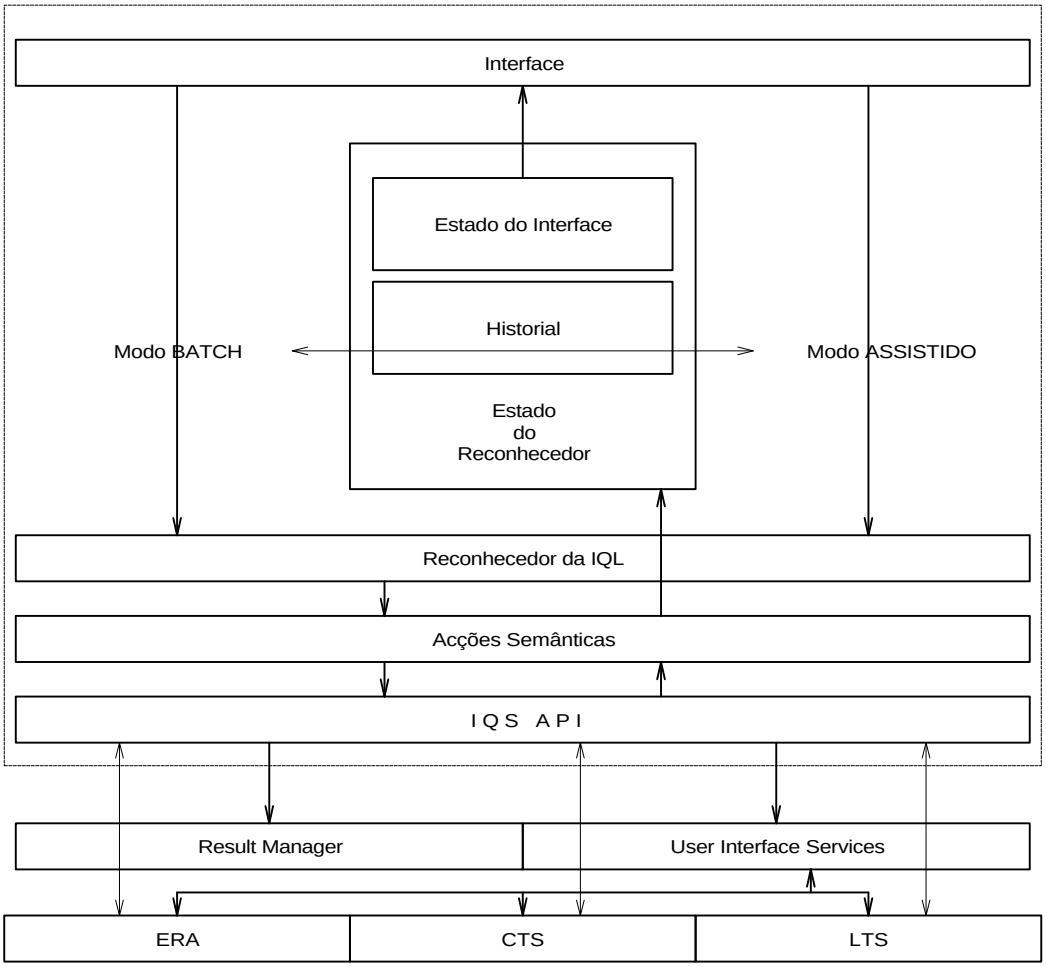

Figura 8 - Arquitectura do *Intelligent Query System*

Parte III

6 Implementação do IQS (iqs.dll)

A implementação da **iqs.dll**, objectivo último do presente estágio, vem no seguimento do trabalho descrito em [ANR93], o qual, para além da Concepção do IQS, apresentava já a declaração C de uma IQS API[16] básica, bem como um estudo das estruturas de dados mais adequadas ao processo de interrogação. Encontrava-se também disponível um protótipo em Camila simulando as funcionalidades pretendidas para o IQS.

Assim, as *Templates*[17] e uma API fundamental, prototipada em Camila, seriam o ponto de partida para a implementação em C da IQS API. As próximas secções descrevem as diversas fases dessa implementação ao longo do tempo.

6.1 Estudo da Arquitectura do SOUR

Antes de pensar em quaisquer pormenores de implementação, havia que compreender minimamente o SOUR: *o que era? o que pretendia? o que trazia de novo? qual o papel de cada elemento da sua arquitectura?*

Os capítulos **3** e **4**, reflectem, ainda que de forma necessariamente breve, a preocupação que houve em encontrar resposta a estas e outras perguntas. O que é importante aqui referir é que antes de "mergulhar" no IQS, foi necessário adquirir uma perspectiva global do SOUR a fim de melhor entender os objectivos da sua ferramenta de *Query*, cuja implementação era, afinal, o objectivo do estágio.

6.2 Estudo das DLLs

Sabendo que se pretendia codificar em C uma DLL que implementasse o sub-sistema IQS, então, antes de iniciar a tradução Camila → C da IQS API já disponível, foi necessário um estudo prévio [CP92] sobre as regras e filosofia próprias da programação de DLLs, a serem respeitadas durante a codificação da **iqs.dll**[18].

Uma vez que o Microsoft Visual C++ 1.5 simplifica consideravelmente a codificação de DLLs ao gerar de forma automática todos os ficheiros intermédios necessários, esses pormenores serão aqui omitidos.

---

[16] naturalmente, e reflectindo pormenores de implementação, algumas dessas funções veriam o seu protótipo modificado...

[17] recorde-se o capítulo **5.3 IQL - uma linguagem de query para o IQS**.

[18] como veremos oportunamente (capítulo **6.7 Reconhecedor para a IQL**) este estudo, embora de valor e oportunidades inegáveis, não foi de todo aproveitado devido a pormenores ligados ao código C encarregue do reconhecimento das frase de *query*.

6.2.1 DLLs no contexto Windows

As DLLs (*Dynamic Link Libraries* ou Bibliotecas de Ligação Dinâmica) são um dos mais importantes elementos estruturais do Windows. A maior parte dos ficheiros em disco, associados ao Windows, são módulos do Windows ou DLLs: **kernel.exe**, **user.exe**, **gdi.exe**, **keyboard.drv**, **system.drv** e **sound.drv,** os controladores de vídeo e impressora e outros *device drivers* são todos DLLs. Os próprios ficheiros de fontes (**.fon)** são DLLs, embora apenas de recursos, não contendo código ou dados.

Um programa Windows é um ficheiro executável que, genericamente, cria uma ou mais janelas e recebe informações do utilizador através de um mecanismo de mensagens. Normalmente, as DLLs não são directamente executáveis. Além disso, também não recebem mensagens. São antes ficheiros separados que contêm funções que podem ser chamadas por programas e eventualmente por outras DLLs, a fim de executar determinadas tarefas.

Uma biblioteca de ligações dinâmicas só é trazida à "vida" quando alguém chama uma das funções da biblioteca. As DLLs com extensão **.dll** são carregadas automaticamente pelo Windows (o Windows procura a DLL na directoria corrente, ao longo do *PATH*, na directoria do Windows e por fim na sua subdirectoria **system**).

6.2.2 Ligação estática *versus* ligação dinâmica

Uma ligação estática (ou *linkagem* estática) ocorre quando criamos um executável (**.exe**) através da *linkagem* de módulos objecto (**.obj**) e de bibliotecas estáticas (**.lib**). Sendo assim, o executável compreende em si as próprias bibliotecas e sempre que outros executáveis gerados da mesma forma e usando a(s) mesma(s) biblioteca(s) se encontrem simultaneamente em memoria, haverá naturalmente desperdício desse recurso (que, lógicamente, se estende ao espaço que esses executáveis ocupam em disco).

Ligação dinâmica (ou *linkagem* dinâmica) refere-se ao processo que o Windows usa para ligar a chamada a uma função (chamada essa que ocorre no contexto de um determinado módulo) ao seu código na biblioteca respectiva. Uma ligação dinâmica ocorre durante a execução do próprio programa e recorre a uma única cópia da DLL mantida em memória e partilhada por todos os executáveis que invoquem as funcionalidades dessa biblioteca. Deste modo, as DLLs possuem vantagens evidentes sobre a *linkagem* estática, pois:

- permitem poupar recursos (a *linkagem* em *run-time* é largamente compensada pela poupança de memória e disco);

- mudanças no código da biblioteca não implicam voltar a *linkar* a biblioteca com todos os programas que a usam, desde que os pontos de acesso à biblioteca se mantenham inalterados.

6.2.3 Exportação de funções

A fim de poderem ser invocadas do exterior, as funções de uma DLL necessitam de ser exportadas. Em Microsoft Visual C++ 1.5, isso é conseguido colocando as palavras reservadas `WINAPI __export` imediatamente a seguir à declaração do tipo do valor de retorno da função, quer aquando da sua prototipagem, quer aquando da sua implementação.

Por exemplo:

```
int WINAPI __export dummy_fun ( char dummy_char);
```

é o protótipo de uma função `dummy_fun`, exportável de uma determinada DLL.

6.2.4 Resolução de endereços

Quando o Windows carrega um programa na memória para ser executado, vai ser necessário resolver as chamadas que o programa faz às funções importadas. Se por sua vez o módulo da biblioteca que contém essas funções ainda não foi carregado para a memória, o Windows carrega pelo menos o segmento de Dados e um segmento de Código e invoca uma pequena rotina de inicialização no módulo da biblioteca. O Windows também cria rotinas de *reload* para as funções exportadas na biblioteca. As chamadas efectuadas no programa a funções importadas de uma dada DLL podem então ser resolvidas inserindo os endereços das rotinas de recarga no segmento de Código do programa.

6.2.5 Stack de uma DLL

Cada programa Windows tem a sua *stack*. Logo, o Windows precisa de mudar de *stack* sempre que comuta de programa. A presença de uma *stack* num programa identifica-o como um processo distinto, com um determinado estado[19] característico e com a capacidade de receber mensagens.

Quando um programa Windows chama uma função de uma DLL é como se o processo fosse o mesmo (diz-se que não há mudança de *task*). Para o Windows, o programa que faz a chamada à DLL ainda está em processamento, mesmo durante a execução do código da biblioteca.

Sendo assim, conclui-se que uma DLL usa necessariamente a *stack* do invocador.

[19] dado pelo conjunto de registos, a fila de mensagens pendentes, etc.

6.2.6 A questão DS ≠ SS

Os microprocessadores da família *ix86* têm quatro registos considerados suficientes para referenciar os segmentos[20] em uso por um programa: o registo CS para o código (*Code Segment*), DS para os dados (*Data Segment*), SS para a *stack* (*Stack Segment*) e um auxiliar ES (*Extra Segment*). O IP (*Instruction Pointer*) referencia sempre um endereço no *Code Segment*. O SP (*Stack Pointer*) aponta para o topo da *stack*, algures no *Stack Segment*.

Os outros registos da arquitectura *ix86* podem endereçar qualquer posição num dos quatro segmentos. Os apontadores de 16 *bits* designam-se *near* e consistem apenas no deslocamento (*offset*) relativo ao início de um determinado segmento[21].Os apontadores de 32 *bits* chamam-se *far* e contêm quer o endereço inicial do segmento, quer o *offset* dentro do segmento.

Ora, os compiladores para a linguagem C armazenam no *Data Segment* todas as variáveis definidas como externas, globais ou estáticas, usando apontadores *near*, relativos ao registo DS, para as endereçar.

Contudo, todos os parâmetros das funções e todas as variáveis locais (desde que não estáticas) são armazenadas no *Stack Segment*, e mais uma vez, recorre-se a apontadores *near*, desta feita relativos ao registo SS, para lhes aceder.

Sendo assim, quando num programa C se usam apontadores *near*, estes podem referenciar uma posição quer no *Data Segment* quer na *stack*, pois o Compilador não consegue determinar se os apontadores *near* são *offsets* em relação a DS ou SS! Por este motivo, os programas em C são geralmente construídos para usar o mesmo segmento para os Dados e para a *stack*.

Por exemplo, seja a seguinte linha de código C[22]:

```
tamstr=strlen(string);
```

Se a variável `string` for declarada como:

```
char *string="ola mundo!";
```

então ficará armazenada em memória estática, ou seja, no *Data Segment*, e o endereço de `string`, a receber por `strlen`, será um apontador *near*, relativo a DS. Mas se a declaração de `string` fosse dentro de uma função:

```
char string [100];
```

então `string` seria alocada na *stack* e a função `strlen`, não sabendo a origem do apontador *near* que recebe (neste caso relativo a SS), tem que assumir DS = SS !

[20] zonas de memória de 64k de extensão e endereço inicial múltiplo de 16.

[21] o endereço inicial desse segmento é conhecido (mantido eventualmente noutro registo), mas não é guardado juntamente com o *offset*, uma vez que o apontador é apenas de 16 *bits*.

[22] assumindo um modelo de compilação que não gera por *default* todos os apontdores como *far*.

Em resumo, o C para a família 8086 assume, tacitamente, que DS = SS, e que portanto todas as referências *near* (estáticas ou não) são, na prática, relativas a DS.

Esta suposição não é válida para as DLLs pois apesar de possuírem *Data Segment* próprio, usam o *Stack Segment* do seu invocador, ou seja DS ≠ SS! Para o exemplo anterior da chamada a `strlen`, no caso de uma DLL `string` é alocada pelo Compilador C na sua *stack* mas `strlen` acede, por defeito, ao *Data Segment*, neste caso da biblioteca, em vez de aceder ao *Stack Segment*, onde verdadeiramente reside `string`.

A fim de evitar estes problemas com as DLLs, devem-se observar as seguintes regras básicas:

- declarar as variáveis locais como estáticas[23], passando-as dessa forma para o Data Segment da DLL; ainda para o exemplo anterior:

```
static char string [100];
```

- usar apontadores *far* para referenciar variáveis locais e parâmetros das funções, ambos alocadas por defeito no *Stack Segment* da DLL; por exemplo:

```
char string [100];
char far dummyptr;
int tamstr;

dummyptr=string;
tamstr=strlen(dummyptr);
```

Uma forma mais radical de contornar o problema é recorrer a um modelo `Large` de compilação, que por defeito usa apenas apontadores do tipo *far*. O preço a pagar, porém, é a duplicação da memória usada para armazenar e manipular os apontadores, agora de 32 *bits*, bem como um natural decréscimo na *performance* das funções.

6.3 Estudo do IQS

O passo seguinte foi então estudar o IQS [ANR93], bem como o protótipo em Camila. Havia também já disponível um estudo prévio do comportamento pretendido a nível da *interface*, compreedendo as relações de algumas funções da IQS API primitiva com os objectos visuais encarregues de apresentar o resultado das chamadas a essa funções.

O capítulo **5** condensa os aspectos mais importantes do *Design* efectuado em [ANR93], mas apresenta também ideias que só amadureceram durante implementação, nomeadamente a estrutura particionada da gramática para a variante assitida da IQL, o Historial e até a própria arquitectura e comportamento do IQS.

[23] o que porém pode consumir rapidamente o espaço disponível no *Data Segment*.

O documento *Functional Specification & Architecture* [IQSFSA94], em anexo, cobre de forma detalhada os mais variados aspectos de Design do IQS, quer com base em [ANR93] quer ainda reflectindo *feedback*[24] de implementação.

6.4 Estruturas de dados típicas do IQS

A escolha das estruturas de dados[25] mais adequadas à manutenção e manipulação da informação recuperada do repositório foi a preocupação seguinte. Com efeito, antes de proceder à codificação da IQS API, era necessário definir os tipos de dados com base nos quais se iriam instanciar as variáveis responsáveis por albergar os conjuntos de objectos recuperados do repositório durante o processo de interrogação.

Precisamente, aqui a palavra chave é *conjunto*. Haveria que definir um tipo de dados em C capaz de suportar a definição abstracta de *conjunto*, de forma conveniente e, se possível, eficiente. Ou seja, para além do acesso aos elementos do conjunto ser fácil, deveria também ser o mais rápido possível. Contudo, é bem sabido que a conveniência e a eficiência, quando perseguidas em simultâneo, raramente se atingem...

Uma solução de compromisso encontrada foi o uso de *Arrays* Dinâmicos. Nestes, o acesso é directo[26] (conveniência) e por consequência bastante rápido (eficiência).

Uma possível definição de *Arrays* Dinâmicos em C é:

```
typedef struct {
   long index;  /* index >=0 implica que o array tenha index+1 objectos       */
                /* index <=-1 equivale a ter o array vazio                     */
   void *array; /* apontador generico para o bloco de memoria contendo o array*/

} Array_Dinamico;
```

O campo *index* é usado para guardar o "estado" do conjunto (se há elementos -e quantos- ou se está vazio) e o conjunto propriamente dito é armazenado no bloco de memória apontado pelo campo *array*. Assim, o apontador *array* é desprezado[27] ou não, conforme *index* indique um conjunto vazio ou não.

Obviamente, nem tudo são vantagens com esta implementação de conjuntos. Uma desvantagem que se pode tornar limitativa da *performance* ganha pelo acesso directo é o facto de que cada vez que for preciso introduzir (ou retirar) um elemento do conjunto, o campo *array* precisar de ser realocado. Em qualquer implementação à

[24] é natural que assim seja já que é sempre difícil durante a fase de *Design* perspectivar limitações encontradas durante a Implementação; qualquer projecto de Software deve pois estar preparado para enfrentar as influências recíprocas entre estas duas fases de desenvolvimento.

[25] recomenda-se a consulta dos capítulos **7 IQS data structures design** de [IQSFSA94] e do capítulo **4 IQS main data structutres** de [IQSTR94].

[26] por indexação directa, em tudo semelhante a um *array* estático.

[27] e portanto considerado livre de qualquer associação com um bloco de memória dinâmica.

base de listas ligadas ou árvores binárias[28], seria apenas necessário alocar ou desalocar o espaço relativo ao elemento em questão, permanecendo o resto de estrutura do conjunto praticamente intacta[29]. Contudo, os algoritmos de manipulação de listas são mais complexos, sendo as listas estruturas de dados mais vulneráveis a erros de gestão.

Consequentemente, durante a implementação da IQS API, procurou-se, sempre que possível, limitar o número de realocações dos conjuntos, quer associando-lhes à partida um bloco de memória de tamanho considerado suficiente para satisfazer todos os pedidos durante a operação em causa, quer pedindo um bloco múltiplo dos elementos do conjunto, caso a realocação seja efectivamente necessária.

A definição em C do tipo `Array_Dinamico` deve ser entendida como uma definição de base, *i.e.*, não pretende ser a única definição para todos os conjuntos a usar pela IQS API. Na verdade, ela deve ser extendida sempre que necessário, a fim de contemplar aspectos particulares de alguns conjuntos (como é o caso de conjuntos de conjuntos, ou conjuntos que para além do *index* precisam de mais um campo para serem caracterizados, etc). O capítulo **4.1.1 Dynamic Arrays implementing Sets** de [IQSTR94], em anexo, mostra a este propósito um extracto do *header file* **iqs.h** com as declarações de todos os tipo de conjuntos usados pela IQS API, bem como esclarece estes e outros pormenores sobre os *Arrays* Dinâmicos como forma de implementação de conjuntos.

6.4.1 Uma Set API básica

A conveniência já de si proporcionada pelos *Arrays* Dinâmicos implementando conjuntos, foi extendida às operações básicas a executar sobre eles. A fim de tornar a codificação da IQS API mais fácil e legível[30] bem como tirar todo o partido possível dos tipos C implementando conjuntos, foi decidido criar um pequeno "módulo" de funções representativas de algumas operações básicas sobre conjuntos. Assim, sempre que fosse necessário manipular conjuntos de objectos recolhidos do repositário, estas funções ofereceriam o encapsulamento necessário (e conveniente).

O capítulo **4.1.2 A basic Set API** de [IQSTR94], em anexo, descreve todos os pormenores relevantes ligados ao desenvolvimento desta mini-API, bem como um *quick reference* sobre as funções disponíveis.

6.4.2 O Estado do IQS

A fim de guardar o estado de um processo de interrogação ao repositório, quer sob o ponto de vista dos objectos recuperados até ao momento, quer relativamente à informação de controlo e conteúdo dos objectos visuais da camada de *Interface*[31], era necessário organizar de alguma forma essa informação. Assim, reflectindo um decisão de *Design* expressa já

[28]que mais não são que um esquema mais elaborado (e porventura mais eficiente) de listas.
[29]a menos das necessárias actualizações dos *links* entre as células da lista.
[30]o que é importante em termo de manuntenção.
[31]o que era fundamental pelo menos no Modo Assistido.

em [ANR93] e refinada em **7 ISQ data structures design** de [IQSFSA94], foi decidido definir uma estrutura cujos campos, quando instanciados, constituiriam a informação considerada relevante em termos da definição do estado actual do processo de *querying*.

O tipo de dados em C dessa estrutura foi designado por `ParserState` e a única instância (global) desse tipo, designada de `iqsState`. O capítulo **4.2 The ParserState data type** em [IQSTR94] apresenta os detalhes de implementação do tipo `ParserState`, nomeadamente qual a informação que se entendeu aí preservar[32].

Nota importante:

Nesta fase da implementação, ainda não estava completamente definido o mecanismo a adoptar para o reconhecimento léxico/sintáctico das frases de *query* e por isso, à partida, não se consideraram no tipo `ParserState` quaisquer campos directamente relacionados com este aspecto. Era apenas suficiente considerar que de *alguma* forma as frases de interrogação seriam reconhecidas e as funções da IQS API encarregar-se-iam de recuperar a informação e depositá-la nos campos correctos de `iqsState`, actualizando também o estado da *Interface*, que o Visual Basic deveria consultar.

Como mais tarde se veio a verificar, esta decisão, embora resultante do desconhecimento de alguns aspectos críticos de implementação, na altura indefinidos, não se revelou errada dado que o uso de geradores automáticos (Lex & Yacc) de código C reconhecedor das frases de *query*, remeteu para esse código a responsabilidade[33] de gerir as estruturas de dados representativas do estado do processo de reconhecimento.

6.5 Implementação da IQS API

Tendo pois disponível uma API básica de conjuntos e um local bem definido onde guardar a informação sucessivamente recuperada, procedeu-se à codificação em C da IQS API sugerida por [ANR93]. Recorde-se que esta API foi concebida com o intuito de responder às necessidades das *Templates* em termos de recuperação de objectos. Além disso, uma API conforme às *Templates* é directamente reutilizável pelo Modo Assistido já que este, basicamente, é uma extensão[34] do Modo *Batch*, o qual, recorde-se, reconhece apenas frases de *query* directamente derivadas das *Templates*.

Relativamente à IQS API primitiva, apenas constituem novidade as funções:

- `int iqsGetClustersWithCaractRel(AOID_list FAR*, Name_list_list FAR*, char FAR *);`

- `int iqsGetAoidsBySLC(AOID_list FAR *, char FAR *);`

[32]em particular, chama-se a atenção para a secção **4.2.1** de [IQSTR94] que descreve os detalhes de implementação mais importantes do Historial.

[33]a este propósito, a introdução do capítulo **4.2** de [IQSTR94] apresenta comentários adicionais.

[34]contemplando pormenores específicos à *Interface* (consultar **2.4.1** de [IQSTR94]).

- `int iqsGetPHAsBySLC(Name_list FAR *, char FAR *);`

- `int iqsGetSLCsByPHA(Name_list FAR *, char FAR *);`

A descrição das funções da IQS API é fornecida em **5 The IQS API** de [IQSTR94], em anexo, e um *cross-reference* completo pode também ser observado no anexo [IQSTR94], capítulo **8 IQS module cross reference manual**. Por agora, é suficiente dizer que o código implementando essas funções se baseia maioritariamente em chamadas à ERA API e CON API. Essas chamadas seguem a filosofia *start-next* com parâmetros de entrada-saída, característica da maioria das invocações às funções dessas APIs.

Em termos de implementação, a designação IQS API emprega-se com um significado mais lato que aquele que lhe temos vindo a atribuir. Na verdade, o ficheiro **iqs.c** congrega não apenas as funções encarregues de recuperar objectos do repositório, mas também todo um conjunto de outras funções auxiliares, cuja necessidade se fez sentir durante a implementação. Neste sentido, doravante qualquer referência à IQS API deve levar em conta que ela abarca:

- a IQS API propriamente dita;

- a já mencionada SETAPI;

- o conjunto das funções exportáveis, a invocar pelo Visual Basic a fim de trocar informação com a camada C e recuperar o estado da sua *Interface*.

- um conjunto de funções auxiliares, envolvendo algumas redefinições de funções conhecidas.

Quer as funções invocadas pelo Visual Basic, quer as funções ditas auxiliares serão objecto de discussão em secções próprias (**6.8** e **6.5.1** respectivamente).

6.5.1 Funções auxiliares

As funções auxiliares (por fornecerem serviços ao resto das funções da IQS API) que houve necessidade de implementar, foram as seguintes:

- ```
  int iqsFrealloc (void FAR ** memptr, size_t memsize,
                   int ptrtype);
  ```

  Esta função baseia-se numa chamada à função

  ```
  void FAR * _frealloc (void FAR * memblock, size_t size)
  ```

  prototipada em `malloc.h`, modificando ligeiramente o seu comportamento. Assim, em vez de retornar um `void FAR *`, correspondente ao endereço do bloco realocado (e possivelmente movido), a função retorna um código de sucesso/insucesso, usando um parâmetro de entrada/saída (`void FAR ** memptr`) para atribuir o novo endereço ao bloco que se pretende realocar. A razão deste procedimento
  ```

é o facto de que a função `_frealloc` retorna `NULL` quando a realocação não é bem sucedida e por isso qualquer invocação do tipo

```
dummy_ptr=_frealloc(dummy_ptr, DUMMYSIZE)[35];
```

corre o risco de atribuir o valor `NULL` a `dummy_ptr`. No caso em que `dummy_ptr` apontava para um bloco de memória previamente à tentativa de realocação, então se a realocação falhar podemos perder a referência para esse bloco, a não ser que antes de invocar `_frealloc` se fizesse uma cópia de `dummy_ptr`. Foi precisamente com a intenção de "encapsular" esses pormenores que se optou pela utilização de `iqsFrealloc` em vez de `_frealloc`;

- `char FAR * iqsStrtok (char FAR * str);`

A função `iqsStrtok` pretende oferecer basicamente a mesma funcionalidade que a função

```
char FAR * _fstrtok (char FAR * string1,
                     const char FAR * string2)
```

prototipada em `string.h`, ou seja, percorrer `string1`, devolvendo os *tokens* separados pelos caracteres delimitadores que `string2` contém, mas para o caso particular do delimitador ser apenas o caracter `','` (vírgula) e mantendo `string1` intacta[36]. Além disso, `iqsStrtok` deverá devolver o *token* (`char Far *)NULL` sempre que encontra dois separadores (vírgulas) seguidos, ao contrário de `_fstrtok` que ignora esses casos. A necessidade de implementar `iqsStrtok` prende-se com o facto de ser relativamente frequente a inspecção de *strings* que consistem em listas de *tokens* separados por vírgulas. Esses *tokens* são geralmente nomes de atributos ou os correspondentes valores, recuperados do repositório invocando funções da ERA API. Essas funções usam em geral uma instância de uma estrutura do tipo `eraClassEntry`, que serve de parâmetro de entrada/saída da função. A definição de tal estrutura é, recorde-se:

```
typedef struct
 {
  ObjID id;
  char ClassName[LEN_CLASS];
  int AttrNumber;
  eraAttrItem AttrList;
  char Parent[LEN_CLASS];
 }eraClassEntry;
```

em que `eraAttrItem` é definida como:

[35]com `dummy_ptr` declarado como apontador tipo `FAR *`.

[36]note-se que `_fstrtok` insere um `'\0'` após cada *token* que encontra em `string1` (o `'\0'` sobrepõe-se ao caracter delimitador encontrado...).

```
typedef struct
 {
  char AttrName[LEN_ATTRNAME_LIST];
  char AttrType[LEN_ATTRTYPE_LIST];
  char AttrLen[LEN_ATTRLEN_LIST];
  char AttrValue[LEN_ATTRVALUE_LIST];
 }eraAttrItem;
```

Após o regresso de uma chamada à ERA API, os campos `AttrName` e `AttrValue` contêm nomes e valores de atributos, respectivamente, separados por `','` (vírgula). Em geral, apenas um desses *tokens*, tem interesse e portanto é necessário percorrer aqueles campos, até recuperar o *token* pretendido.

- `int iqsCheckWordInList(char FAR * word, char FAR * list, long int FAR * index, int mode);`

Esta função recorre a chamadas a `iqsStrtok` a fim de pesquisar `list` em busca de um determinado *token* a devolver em `word`[37]. Para além de ser invocada por `iqsGetAttrValue`, é largamente utilizada em muitas operações sobre conjuntos de *tokens*[38].

- `int iqsGetAttrValue (char FAR *names, char FAR *name, char FAR *values, char FAR *value);`

Recorrendo a `iqsCheckWordInList`, esta função implementa projecções de uma lista de *tokens*, dada por `names`, noutra lista de *tokens*, dada por `values`, tendo o caracter `','` (vírgula) como separador dos *tokens*. A projecção consiste em percorrer `names` em busca de `name`, registar a posição deste *token* e retornar em `value` o *token* que está na mesma posição na lista `values`. O uso desta função para projectar `AttrName` sobre `AttrValue` é bastante vulgar nas funções da IQS API que fazem chamadas à ERA API.

A fim de obter uma descrição mais pormenorizada das funções auxiliares, deve-se consultar a secção **5.1 IQS Auxiliary API**, em [IQSTR94].

6.6 Reconhecedor para a IQL

Uma vez implementadas as funcionalidades da IQS API necessárias à resolução de frases de *query*, quer a partir do Modo *Batch*, quer a partir do Modo Assitido, era necessário pensar em formas de reconhecer essas interrogações, escritas em qualquer das duas variantes da IQL (*Batch* ou *Assistida)*.

[37] entre outras funcionalidades que podem ser consultadas na secção **5.1 An Auxiliary IQS API**, em [IQSTR94].

[38] para esses conjuntos o bloco de memória seria uma *string* de *tokens* separadas por `','`.

Como **4 IQS Operation Modes**, em [IQSFSA94], refere, a melhor forma de descrever ambas as IQLs é através de uma gramática. Tal descrição pode ser usada para gerar automáticamente código reconhecedor, recorrendo por exemplo a ferramentas como o par **Lex & Yacc** [LY92]. O capítulo **5.1 IQL grammars**, em [IQSFSA94], alerta ainda para a conveniência em unificar[39] as gramáticas relativas às variantes *Batch* e Assistida da IQL.

Para além de referir também estes aspectos, o capítulo **2 IQL Parsing**, de [IQSTR94], descreve todos os pormenores relevantes relacionadas com o uso do **Lex** e do **Yacc** a fim de gerar o *lexer* e o *parser*, respectivamente, para a IQL unificada. Nomeadamente, a **estrutura em degrau**[40] da variante Assistida é posta em evidência em **2.4.1 Aided Mode IQL sub-grammar issues** de [IQSTR94], como forma de contornar as dificuldades de implementação do *parsing* em *background*, a nível da DLL em WINDOWS 3.1.

O processo de importação do código gerado para a DLL em construção (**iqs.dll**), não seria de todo pacífico, dado que para além de ter sido necessário redireccionar o *input* do *lexer* [GF93], outros detalhes tiveram de ser atendidos[41]. Em particular, o facto de ser necessário usar o modelo *Large* de compilação[42] foi algo não previsto inicialmente, e que à primeira vista tornou "inglório" o esforço empreendido em codificar a parte da IQS API até então construída segundo as convenções das DLLs[43]. Esse esforço não foi em vão, porém, porque se mais tarde fosse entendido usar outro reconhecedor, eventualmente compatível com a filosofia das DLLs, o código escrito provavelmente não necessitaria de ser modificado. Para o caso presente, contudo, o modelo *Large* (implicando o uso exclusivo de apontadores *far*, *i.e.*, de 32 *bits*) era a única garantia de que após ter sido compilado com sucesso, o código importado não violaria o princípio DS ≠ SS (*i.e.*, *Data Segment* ≠ *Stack Segment*), que governa toda a programação de DLLs.

6.7 Acções Semânticas

A ordem em que surge este capítulo não deve ser entendida como indicativa do facto de que as actividades aqui descritas foram levadas a cabo somente depois da questão do *parsing* ter sido resolvida. Na realidade, algumas dessas tarefas foram executadas em paralelo com o que foi descrito em **6.6**.

A descrição gramatical contida no ficheiro **iqs.y**, a fornecer ao Yacc, incluía necessariamente chamadas às funções implementando as acções semânticas a executar sempre que um reconhecimento fosse bem sucedido. Inicialmente, essas funções reduziam-se praticamente ao seu cabeçalho[44], pois o objectivo não era ainda executá-las, mas sim verificar apenas a validade da descrição gramatical do ficheiro **iqs.y**. Esta

[39] unificação essa entendida aqui como elevar, se possível, a *root* de ambas as gramáticas a uma *root* comum.

[40] cada degrau correspondendo a uma expansão válida da *query* sintetizada na *Interface*.

[41] a este propósito, ver **3 Importing the generated code to a Windows DLL**, [IQSTR94].

[42] **3.6 The Large model of compilation**, [IQSTR94].

[43] recordar **6.2.6 A questão DS ≠ SS**.

[44] a este propósito, recordar a utilidade da directiva ao *C pre-processor*, `#define __DOS__` (**3.5 The #define __DOS__ directive**, [IQSTR94]).

fase consistiu pois no teste da gramática[45] e na Concepção das acções semânticas (nomeadamente, na elaboração dos seu protótipos).

Após se ter estabilizado o processo de importação do código gerado para a DLL, procedeu-se então à codificação das acções semânticas relativas a cada *Template*. Esta implementação contemplou em primeiro lugar as produções do ramo Assistido da gramática unificada, por se considerar que era esse modo de operação o mais crítico em termos de viabilidade e implementação.

A implementação das acções semânticas para as produções da variante *Batch* não ofereceu obstáculos de maior uma vez que essas acções eram basicamente as mesmas do Modo Assistido. Grosso modo, seria apenas necessário isolar do Modo *Batch* as linhas de código que no Modo Assistido servem para determinar o estado da *Interface*.

Finalmente, de referir que à parte algumas funções auxiliares[46], as acções semânticas fizeram uso intensivo, como era natural, das funcionalidades disponibilizadas pela IQS API[47], contribuindo inclusivamente para o refinamento de algumas dessas funções e implementação de outras novas[48].

A descrição da funcionalidade de cada acção semântica (e funções auxiliares) encontra-se no capítulo **6 The IQS Semantic Actions API (actions.c)** de [IQSTR94].

6.8 Comunicação com o Visual Basic

O SOUR é, por assim dizer, uma ferramenta híbrida, porque utiliza o melhor de dois mundos. Assim, a interacção entre os Níveis Aplicativos e os seus utilizadores foi implementada recorrendo às facilidades que o Visual Basic 3.0 oferece em termos de *design* e controlo do *Interface* em WINDOWS 3.1. Por seu turno, os serviços que o utilizador espera das ferramentas com que interactua, são requeridos, pela camada de *Interface*, a níveis inferiores, implementados em C por razões de eficiência. O IQS não escapa a esta regra, o que aliás se torna evidente observando a sua arquitectura[49].

Sendo assim, coloca-se imediatamente a questão da comunicação entre a camada de *Interface*, implementada em Visual Basic 3.0 e as funcionalidades disponibilizadas pela IQS API. Como esta API assume a forma de uma DLL, então só as funções exportáveis[50] é que poderão ser invocadas pela camada de *Interface*. Essas funções definem um subconjunto da IQS API, que se encarrega fundamentalmente de:

- fazer o *init*, o *reset* e o *clear* da variável de estado[51] do IQS (`iqsState`) antes da execução da 1ª *Template*, antes da execução da 2ª e próximas, e antes de abandonar o IQS, respectivamente; no Modo Batch, porém, o *reset* não é invocado a partir do Visual Basic, mas sim na camada C, após a resolução de uma *query* e antes da próxima, uma vez que esta camada recebe um *batch* de

[45] ainda em ambiente Unix e "alimentando" o *lexer* normalmente, *i.e.*, à base de ficheiros.
[46] apresentadas em **6.1 The IQS Semantic Actions Auxiliary API**, [IQSTR94].
[47] à excepção do subconjunto exportável, destinado apenas a ser invocado pelo Visual Basic.
[48] ou seja, o *feedback* de implementação novamente em acção...
[49] recordar **5.6 Arquitectura do IQS**.
[50] recordar **6.2.3 Exportação de funções**.
[51] consultar **6.4.2 O Estado do IQS**.

queries a resolver e só devolve o controlo da aplicação ao *Interface* após a resolução da última *query*;

- operações várias sobre o Historial, nomeadamente: *init*, *clear*[52], *get*, *load* e *save*;
- recuperação dos campos da variável de estado do IQS, `iqsState`, que controlam o *enable/disable* (e eventualmente o conteúdo) de vários objectos da *Interface*;
- formatar convenientemente os resultados de uma *query* a fim de serem submetidos ao *Result Manager*;
- invocar o reconhecedor da IQL com base numa frase de *query* sintetizada em Modo Assistido ou fornecendo um "lote" de *queries* a partir do Modo *Batch*.

Uma descrição detalhada dos campos da variável de estado `iqsState`, relacionados com o Visual Basic, é fornecida na secção **4.2 The ParserState data type**, em [IQSTR94]. As funções exportáveis são dissecadas em **7 The IQS Visual Basic related API**, também em [IQSTR94].

7 Trabalho Futuro

Apresentam-se de seguida alguns tópicos cuja implementação enriqueceria a funcionalidade do IQS, mas que, por limitações de tempo, não foi possível levar avante:

- um Modo Semi-Assistido, semelhante ao Modo Assistido, a menos da assistência semântica permanente;
- limitar ainda mais as hipóteses de resposta vazia a uma *query* em Modo Assistido, pese embora o *overhead* que isso deixa antever[53];
- tornar a gestão de memória mais robusta e eficiente, procurando implementações do tipo abstracto Conjuntos, alternativas aos *Arrays* Dinâmicos;
- desenvolver uma IQL mais flexível, porventura com combinando as *Templates* actuais e introduzindo, se necessário, outras novas.

A secção **9 Final Remarks**, em [IQSFSA94], apresenta também alguns aspectos, sob um ponto de vista de *Design*, a ter em conta em futuras versões do IQS.

[52] embora *init* e *clear* do Historial também aconteça no *init* e *clear* da variável de estado, é por vezes necessário efectuar estas operações fora do contexto da variável de estado.

[53] a fim de compreender o nível de assistência alcançado na presente implementação, consultar a secção **6.1.1 Semantical Assistance implementation constraints**, [IQSTR94].

8 Conclusões

O IQS é um exemplo típico dos problemas que se enfrentam em qualquer implementação de uma *interface* Homem-Máquina. O desejo de providenciar uma interacção suave e intuitiva, encontra frequentemente limitações de implementação e de eficiência, para além de ser difícil (senão impossível) agradar a todos os potenciais utilizadores do produto final.

Qualquer aplicação que sirva de *front-end* a um processo de *querying*, confronta-se, pelo menos, com duas questões essenciais, cuja resolução determina o seu sucesso:

- o desenvolvimento de uma *interface* que torne a construção de *queries* fácil, intuitiva e flexível;

- a interacção com o repositório da informação, a fim de recuperar os objectos que respondem às *queries*.

Este projecto de estágio versou fundamentalmente aspectos da *interface*. O termo *interface* deve aqui ser entendido num sentido mais lato, envolvendo não só os detalhes de interacção entre o utilizador e a aplicação (detalhes esse do foro da programação em Visual Basic) mas também a construção de uma *software layer* de acesso à SOURLIB, uma vez que esta fornece o encapsulamento e as funcionalidades necessárias para aceder ao repositório. Especificamente, a implementação dessa *software layer* sob a forma de uma DLL para o WINDOWS 3.1 constituiu o cerne do estágio.

A concepção do IQS previa, à partida, pelo menos dois modos de operação, dirigidos a utilizadores com diferentes níveis de conhecimento da estrutura lógica do repositório. Tal permitiu introduzir alguma adaptabilidade à ferramenta. Em particular, o Modo Assistido, providenciando assistência sintáctica e semântica permanente, é talvez a componente que mais se destaca no IQS, pois traduziu-se na necessidade de implementar um mecanismo de tradução dos eventos da *interface* nas respectivas chamadas à SOURLIB bem como no controlo determinístico do próximo estado desse *interface*. Esse controlo foi entregue a um autómato definido pela gramática reconhecedora das frases de *query*, cujo código seria gerado automaticamente via *Lex* e *Yacc*. Aliás, o *Lex* e o *Yacc* mostraram-se ambos ferramentas imprescindíveis pela economia de tempo e trabalho que proporcionaram.

Sob o ponto de vista da implementação, a dualidade de operação do IQS traduziu-se ainda num esforço de compatibilização do código C relativo a ambos os modos, a fim de reutilizar, o mais possível, as chamadas à SOURLIB. A integração e reutilização de código foi assim uma preocupação constante no desenvolvimento do IQS, razão principal da modularidade conseguida nas suas *sources* C.

A implementação do IQS foi ainda um campo de ensaio no que toca à integração de código C, proveniente de geradores automáticos, com código desenvolvido. Com efeito, uma parte não desprezável do estágio foi dedicada à investigação dos diversos aspectos a ter em conta na importação de código gerado pelo *Lex* e *Yacc*, para uma DLL em WINDOWS 3.1, uma vez que esse código não é directamente portável.

Concluindo, o estágio permitiu adquirir um conhecimento geral do projecto SOUR, e em particular do seu subsistema de interrogação ao repositório, o IQS, cuja implementação introduziu alguns aspectos inovadores, nomeadamente o controlo da sua *interface* com base numa gramática, bem assim como a importação e integração de código gerado pelo *Lex* e *Yacc* numa DLL em WINDOWS 3.1.

Referências

[EU89] Eureka Project EU 379 - SOUR Project Profile, 1989.

[SPS93] Sour Presentation Session, INESC, 1993.

[ANR93] Relatório de Estágio de LESI - Análise e Concepção de um *Intelligent Query System* para um sistema de CASE, António Nestor Ribeiro, 1993.

[CON93] Conceptualiser - Specifiche dei Requisiti e Architettura, Version 2, Revision 0 - Systena, 1993

[PD87] Classifying Software for Reusability - Rubén Prieto-Diaz & Peter Freeman, IEEE Software, 4(1):6-16, 1987

[LZ84] "Making Computers Think Like People", *IEEE Spectrum*, Vol.21, Nº 8, pp. 26-32 - Lofti A. Zadeh, 1984

[CP92] "Programando para Windows 3", Charles Petzold - Makron Books, 1992

[IQSFSA94] "Intelligent Query System - Functional Specification & Architecture", INESC/Systena, 1994

[IQSTR94] "Intelligent Query System - Technical Reference", INESC/Systena, 1994

[LY92] "lex & yacc", John Levine, Tony Mason & Doug Brown - O'Reilly & Associates Inc., 1992

[GF93] "Como redireccionar o *input* de um reconhecedor baseado em *lex* e *yacc* para uma zona de memória", Geraldina Fernandes - Universidade do Minho,1993