HyperPro: Sistema de Programas e Documentação em um Ambiente de Programação Baseado no Paradigma de Estilo Literário

Relatório Final: POC2

Orientando: Flávia Peligrinelli Ribeiro
Orientadora: Profa. Mariza A. S. Bigonha

LLP03/99

Sumário

1 Introdução

O projeto de pesquisa proposto é um subprojeto de um projeto maior que engloba a construção da segunda versão da ferramenta `HyperPro`, denominada `HyperPro Básico`. Este projeto faz parte do programa de cooperação internacional entre o *Institut National de Recherche en Informatique e Automatique* (INRIA) e o Departamento de Ciência da Computação da UFMG. Um dos objetivos do projeto `HyperPro` é auxiliar o projetista de grandes programas Prolog a melhorarem a qualidade de seus programas por meio de comentários coerentes baseados em uma metodologia consistente. Para atingir esse objetivo é necessário criar mecanismos que facilitem a escrita e depuração de programas e textos no ambiente de programação em lógica `HyperPro`. A nossa proposta de trabalho foi implementar para o `HyperPro Básico` o sistema de projeções.

Uma projeção mostra, em uma vista separada, partes selecionadas de um documento. Em um ambiente integrado de programação e documentação é importante que o usuário tenha uma forma de testar seu programa separadamente. Por isso, dentre as funcionalidades propostas pela equipe do `HyperPro Básico`, as projeções desempenham um papel importante.

O `HyperPro` [2, 3, 4, 5, 6] é um ambiente experimental idealizado para desenvolver programação em lógica baseado no paradigma do estilo literário [14, 16, 17]. Seu objetivo é documentar programas CLP [20] (*Constraint Logic Programming*) oferecendo ao usuário a possibilidade de editar, em um ambiente homogêneo e integrado, diferentes versões e programas, comentários e informações de verificação formal, assim como a possibilidade de executar, depurar e testar os programas. Os programas são vistos como documentos executáveis. Um aspecto importante desse sistema é que, com poucas alterações, o `HyperPro` pode ser adaptado para outras linguagens. O sistema utiliza-se da ferramenta `Thot` [7, 8].

O `Thot` é um sistema desenvolvido para produzir documentos estruturados. É um poderoso editor WYSIWYG (*what you see is what you get*) que possui entre outras, facilidades de hypertexto. Ele permite que o usuário crie, modifique e consulte interativamente os documentos, além de permitir a produção de documentos homogêneos. O usuário pode, então, se concentrar na organização e no conteúdo dos seus documentos.

Com o intuito de prover as facilidades descritas acima foram desenvolvidos no protótipo do `HyperPro` padrões de estruturas de documentos para programas de acordo com a metodologia de programação em lógica. Esses padrões utilizam-se das linguagens do `Thot`. São elas:

- linguagem S [10]: responsável pelas estruturas genéricas que formam a base do documento.
- Linguagem P [10]: responsável pela apresentação do documento.
- Linguagem T [10]: responsável pela exportação ou tradução de documentos da forma canônica do `Thot` para outros formalismos, por exemplo: HTML [22], LaTeX[21], etc.
- API [11]: responsável pelos esquemas de interface.
- Linguagem A [12]: responsável pela geração de aplicações.

Este relatório está organizado da seguinte forma: os Capítulos de 1 a 2 apresentam um breve histórico do projeto, os objetivos e motivações. No Capítulo 3, é apresentado o estado da arte em relação aos sistemas existentes para elaboração de documentação, em particular, focalizando sistemas que utilizam programação literária [15, 14, 18, 16, 17]. A seção 3.2

apresenta uma visão geral do sistema `Thot`, desenvolvido por Vincent Quint, Hélène Richy, Cécile Roisin e Irène Vatton [7].

O Capítulo 4 descreve os principais componentes do sistema HyperPro, o protótipo desenvolvido. A Seção 4.4 apresenta o HyperPro Básico. Nessa seção é mostrada a metodologia empregada no desenvolvimento das projeções.

O Capítulo 5 descreve o principal objetivo de nosso projeto que foi a implementação de quatro projeções para o `HyperPro Básico`.

O Capítulo 6 apresenta as conclusões deste trabalho, as dificuldades encontradas e os trabalhos futuros.

O Apêndice A mostra a árvore de estrutura do HyperPro Básico.

Os Apêndices B, C, D e E apresentam os códigos fontes para as projeções manual, recursiva, de versões e automática e suas caixas de diálogo, respectivamente.

2 Motivações

O `Thot` e o `HyperPro` fazem uso e um paradigma muito importante para a compreensão de programas e textos denominada programação literária. O paradigma da programação literária estabelece que a documentação e o código de um programa estejam em um único documento. As ferramentas implementadas para usar esse paradigma permitem que o usuário digite as partes de um programa em qualquer ordem e extraia a documentação e o código do mesmo arquivo quer seja para gerar um artigo ou para executar um programa e obter um resultado.

O paradigma da programação literária estabelece que a documentação e o código de um programa estejam em um único documento. As ferramentas implementadas para usar esse paradigma permitem que o usuário digite as partes de um programa em qualquer ordem e extraia a documentação e o código do mesmo arquivo quer seja para gerar um artigo ou para executar um programa e obter um resultado.

Apesar da linguagem Prolog [13] já ter atingido sua maturidade, ainda não existe um ambiente que facilite a implementação e a documentação de seus programas. Neste sentido, a programação literária surge como uma alternativa para o desenvolvimento de uma metodologia de documentação de programas no paradigma lógico.

3 Revisão da Literatura

Para entender o sistema HyperPro e ter condição de lhe introduzir novas facilidades tornando-o mais robusto dividimos nossa tarefa no projeto em três fases. A primeira fase consistiu de uma revisão sobre temas básicos: como programação literária, os sistemas existentes, WEB [14], NOWEB [18, 17]. A partir de então, os estudos se concentraram no sistema `Thot`[7], ferramenta usada para programar o HyperPro. Foi estudada as linguagens S, P, T, A e a API.

A segunda fase consistiu no estudo do protótipo HyperPro, sua arquitetura e seu código fonte.

A terceira fase está relacionada com o desenvolvimento das facilidades propriamente ditas para o HyperPro Básico, ou seja, a especificação e implementação das projeções.

Além dessas fases, durante todo projeto, estão sendo produzidos os manuais do sistema e do usuário.

3.1 Sistemas de Programação Literária

Ao invés de considerar o papel do programador como sendo o de dar ordens ao computador, passaremos a nos concentrar em explicar para as pessoas o que os programas de computadores fazem. Um programa deve ser escrito não só para as máquinas lerem mas também para os seres humanos entenderem. Essa é a principal idéia de programação literária.

As ferramentas de programação literária permitem arranjar as partes de um programa em qualquer ordem e extrair a documentação e o código do mesmo arquivo quer seja para gerar um artigo ou para executar um programa e obter um resultado.

A programação literária foi introduzida em 1983 por Donald Knuth por meio de uma ferramenta chamada WEB [14]. A partir daí, falar sobre programação literária significava falar sobre WEB. Este sistema permite que o código fonte coexista com o texto descritivo em um único documento e que as partes de um programa sejam rearranjadas em qualquer ordem auxiliando na explicação das funções dos programas.

Em meados dos anos 80, o método de programação literária se difunde e vários programas são escritos. WEB é adapatado para outras linguagens como C, Ada, Modula2, Fortran, etc [19]. Porém, com certo tempo, de uso muitos programadores ficaram insatisfeitos devido a complexidade do WEB. Essa complexidade acabou por criar uma barreira entre o programador e esse estilo de programação. Dessa forma, a programação literária se tornou útil somente para aqueles que podiam construir suas próprias ferramentas para simplificar o WEB.

Surge então NOWEB [17, 18] que provê as utilidades do WEB mas menos complexo. O NOWEB foi desenvolvido no ambiente UNIX e é portável a outras plataformas desde que essas possam simular *pipelines* e suportar ANSI C e AWK ou Icon.

WEB

O WEB [14] é a combinação de duas outras linguagens: a linguagem de documento formatado Tex e uma linguagem de programação Pascal.

Para usar o WEB o usuário deve preparar um arquivo com a extensão .WEB e submetê-lo a um programa do sistema chamado WEAVE. O programa WEAVE pega um arquivo .WEB, remove os módulos de comentários Pascal e Tex unindo-os em um único documento estruturado com a extensão .TEX. Este documento descreve claramente o programa e as facilidades para sua manutenção. Como o programa e a sua documentação são ambos gerados a partir do mesmo fonte, eles são consistentes. O programa Tex processa o arquivo .TEX e gera uma versão do .WEB responsável pelos detalhes tipográficos tais como *layout* de página e uso de identação. O usuário também pode submeter o mesmo arquivo .WEB a um outro programa do sistema chamado TANGLE. O programa TANGLE pega um arquivo WEB e extrai a parte de código do documento produzindo um arquivo com a extensão .PAS. Este arquivo contem o código em Pascal do programa.

Uma das desvantagens desse sistema é que, dada sua complexidade, fica difícil usar o WEB pois é necessário muito esforço para se adaptar à ferramenta. Além disso, o WEB trabalha mal com o LaTeX.

NOWEB

O NOWEB [17, 18] foi desenvolvido a partir da idéia de escrever trechos de código em qualquer ordem com uma documentação inter-relacionada. Sua simplicidade advém de uma sintaxe simples. Ele possui entre outras facilidades: estrutura de arquivos, *chunks* de código e documentação e características de índices e referências cruzadas.

Estrutura de arquivos: um arquivo NOWEB é uma seqüência de *chunks*. Uma *chunk* pode conter código, possui um nome, ou documentação, que não possui nome. As *chunks* podem aparecer em qualquer ordem.

O NOWEB usa suas ferramentas NOTANGLE e NOWEAVE para extrair código e documentação respectivamente. Quando o NOTANGLE é executado em um arquivo NOWEB ele produz um programa no padrão de saída. Quando o NOWEAVE é executado em um arquivo NOWEB ele o lê e produz, em uma saída padrão, um fonte TEX, a documentação.

As *Chunks* de código: possuem o código do programa fonte e as referências para outras *chunks*. A definição de *chunk* é como a definição de macro. O NOTANGLE extrai o programa expandindo de uma *chunk*. A saída NOTANGLE é legível, pois preserva os espaços em branco e a identação.

***Chunk* de documentação:** possui um texto que é ignorado pelo NOTANGLE e copiado para a saída padrão pelo NOWEAVE, exceto pelos códigos citados: um código pode ser citado na *chunk* de documentação. O NOWEAVE pode trabalhar com o LATEX. Também pode usar HTML, linguagem de hipertexto para Mosaic e WWW.

Características de índice e referências cruzadas: as referências cruzadas de *chunks* e os identificadores fazem dos grandes programas mais fáceis de serem entendidos. O NOWEB usa número de páginas para referências cruzadas. Como o WEB, o NOWEB escreve as referências cruzadas da *chunk* em uma fonte de rodapé, abaixo de cada *chunk* de código. O NOWEB também inclui informação de referência cruzada para identificadores.

As ferramentas NOWEB são implementadas como *pipelines*[23]. Cada *pipeline* começa com um arquivo fonte NOWEB. As *pipelines* são responsáveis pela extensibilidade do NOWEB, o que permite aos usuários criarem novas características de programação literária sem precisar escrever suas próprias ferramentas. Além da representação *pipeline*, o NOWEB possui, entre outras facilidades, comandos de indexação e referências cruzadas, filtros e *backends* padrões.

1. A Representação de *PIPELINES*

   As palavras-chave estruturais representam diretamente a sintaxe do fonte NOWEB. Elas devem aparecer em uma ordem particular que reflete a estrutura do fonte. Elas representam as *chunks*. Ex: @begin, @end

   As palavras-chave *tagging* podem ser inseridas essencialmente em qualquer lugar e com algumas exceções não são geradas por *markup*. Elas não carregam código ou documentação, somente informação sobre esses. Ex: @file, carrega o nome do arquivo fonte do qual a próxima linha vem.

As palavras-chave *wrapper* marcam o início e o fim de um arquivo. Elas são usadas pelo NOWEAVE e não pelo NOTANGLE e são inseridas diretamente pelo *shell*.

2. Indexação e Referência Cruzada

   Os comandos de indexação e referências cruzadas usam rótulos (*labels*), *ident* e *tags*.

   O passo final da referência cruzada envolve a geração de *tags* e associação de um *tag* a um rótulo.

   A função mais básica da referência cruzada é associar rótulos e apontadores com elementos do documento que é feito com as palavras-chave @xref ref e @ xref label.

   A referência cruzada entre *chunks* introduz uma idéia de âncora, que é um rótulo que se refere a um ponto de interesse que é identificado no início de uma *chunk* de código.

3. Filtros Padrões

   Os filtros podem ser usados para adicionar características significantes. As características de referência cruzada e indexação do NOWEAVE usam dois filtros: um que encontra o uso dos indicadores definidos e outro que insere a informação de referência cruzada.

   Na maioria dos casos, os programadores devem marcar as definições de identificadores a mão, usando @ def, mas em alguns casos, um terceiro filtro de linguagem dependente pode ser usado para marcar as definições de indicadores, gerando um índice completamente automático.

4. *Backend*s Padrão

   - nt: implementa o NOTANGLE. Ele extrai o programa definido pela *chunk* de código e escreve o programa em saída padrão.
   - mnt: pode extrair diversas *chunks* de código de um documento em uma passada.
   - tohtml : emite HTML.
   - totex: implementa ambos *backends* TEXe LATEX.
   - unmarkup: inverso do markup - um documento já em *pipeline* é convertido para a forma do fonte NOWEB.

   O NOWEB desenvolve código e documentação em um único documento. O usuário pode escrever as *chunks* de código em qualquer ordem. São geradas automaticamente referência cruzada e índice.

   Como o WEB, o NOWEB possui todas essas características de uma forma mais simplificada.

Outras Ferramentas

A maioria das ferramentas de programação literária são dependentes de linguagem e complexas. As ferramentas mais novas, como o NOWEB, NUWEB [16] e FUNNELWEB [16] são independentes de linguagem.

O NOWEB e o NUWEB apesar de muito similares, possuem algumas diferenças de sintaxe. O FUNNELWEB é uma ferramenta complexa que inclui seus próprios *typesettings* de linguagem e comandos *shell*.

O projeto inicial do NUWEB foi baseado no NOWEB. O NUWEB usa *pipeline* e é um programa C. Sua estrutura o torna portátil pois somente é necessário um compilador C. Também é mais rápido porque não há partes interpretadas e a sobrecarga da criação de *pipelines* é eliminada, mas ele é difícil de ser extendido.

Comparação Entre WEB e NOWEB

As ferramentas WEB podem ser usadas para diferentes linguagens de programação, mas estas devem ser lexicamente similares a liguagem C. NOWEB não depende de nenhuma linguagem de programação, o que o torna mais simples, porém menos poderoso. O WEB tem certas características de dependência de linguagem tais como: *prettyprinting*, *typesetting* comentários usando TEX, expansão de macros, avaliações de expressões constantes e conversão de *strings* literais para índices em uma *string pool*.

O NOTANGLE conserva os espaços em branco e mantém a identação na expansão das *chunks*.

O programa WEAVE do WEB fornece um número a cada *chunk* que é usado pelo seu índice e pelas informações de referência. O NOWEB usa o LATEXpara emitir o índice e as informações de referência para informar o número de página.

O WEB trabalha mal com o LATEX. O NOWEB trabalha com ambos, TEXe LATEX. Ambos, WEAVE e NOWEAVE dependem do formato do texto de duas maneiras: a fonte do programa e o suporte de macro.

O mais importante é a verossemelhança dessas ferramentas estudadas: uma simples entrada produz um programa compilável e um documento publicável. Ambos satisfazem essas expectativas.

Como conclusão do estudo dessas ferramentas, temos que: o WEB nos fornece uma visão clara da programação literária, ele faz de tudo. O objetivo do NOWEB é compor ferramentas simples que manipulam arquivos no formato NOWEB. O NOWEB foi desenvolvido debaixo do ambiente UNIX e usa as suas ferramentas como o AWK e o BOURNE SHELL, mas a única característica que ele realmente precisa é o *pipeline*. A indexação e as referências cruzadas fazem do NOWEB menos simples do que ele poderia ser.

3.2 Thot

O `Thot`[7] é um sistema desenvolvido para produzir documentos estruturados. É um poderoso editor WYSIWYG (*What You See Is What You Get*) que possui entre outras, facilidades de hipertexto. Ele permite que o usuário crie, modifique e consulte interativamente os documentos. Os modelos permitem a produção de documentos homogêneos. O `Thot` possui outras operações tais como numeração, referências cruzadas e índices. Todos os serviços que o `Thot` provê são baseados no sistema interno de representação de documentos.

O modelo de documento do `Thot` permite que o usuário trabalhe com certas entidades que ele tem em mente quando faz um documento. E são essencialmente essas entidades lógicas, por exemplo, parágrafos, seções, capítulos, notas, títulos e referências cruzadas que fornecem ao documento sua estrutura lógica. O modelo `Thot` é baseado no aspecto lógico dos documentos. As diferenças entre documentos não são apenas as entidades que neles aparecem, mas também as relações entre essas entidades e as formas como elas estão ligadas.

O sistema `Thot` usa um meta-modelo que permite a descrição de numerosos modelos onde cada um descreve uma classe de documentos que, por sua vez, possuem estruturas muito

parecidas. Cada documento possui uma estrutura específica que organiza suas partes. A estrutura genérica define as formas pelas quais uma estrutura específica pode ser construída, ou seja, descreve a organização lógica do documento.

Há uma correspondência de um para um entre uma classe e uma estrutura genérica: todos os documentos de uma classe são construídos de acordo com a mesma estrutura genérica. Uma classe é um conjunto de documentos cuja estrutura específica é formada de acordo com a mesma estrutura genérica. Portanto, uma classe é caracterizada por sua estrutura genérica.

As definições de meta-linguagem e de classes de documentos também podem ser aplicadas a objetos. Objetos possuem o mesmo nível de representação lógica dos documentos, contudo, com algumas vantagens. Em particular, é possível definir a apresentação separada dos objetos e atá-los a classes. Então, como documentos, os objetos do mesmo tipo possuem uma apresentação uniforme e a apresentação de cada objeto em uma dada classe pode ser mudada simplesmente substituindo apresentação genérica de classe. Outra vantagem de usar esse modelo de documento é que o sistema é transparente ao usuário, permitindo que ele se concentre no aspecto lógico do documento e dos objetos.

Para programar no `Thot` existe disponível no sistema quatro linguagens: S, P, T e A, responsáveis respectivamente pelo `Esquema de Estrutura`, `Esquema de Apresentação`, pelo `Esquema de Tradução` e pelo `Esquema de Aplicação` do documento. Estas linguagens serão apresentadas nas próximas seções.

A LINGUAGEM S

Estruturas genéricas formam a base do modelo de documento do `Thot`. Cada estrutura genérica, que define as classes de documentos e objetos, é especificada por um "programa" escrito em uma linguagem S [10], e chamado `Esquema de Estrutura`. A gramática de S, assim como as gramáticas das linguagens P e T [10], são descritas usando a meta-linguagem M, derivada de Bakus-Naur Form (BNF).

Um `Esquema de Estrutura` define a estrutura genérica lógica de uma classe de documentos ou objetos independente das operações que podem ser feitas no documento.

Cada `Esquema de Estrutura` começa com a palavra-chave STRUCTURE e termina com END. A palavra-chave STRUCTURE pode ser seguida pela palavra-chave EXTENSION, no caso do esquema definir uma extensão, seguido do nome da estrutura que o esquema define e de um ponto e vírgula.

Em um esquema completo, a definição do nome da estrutura é seguido pelas declarações de *default*, atribuições globais, parâmetros, regras de estrutura, elementos associados, unidades, elementos de esqueleto e exceções.

As seções ATTR, STRUCT, ASSOC E UNITS definem novos atributos, e fazem com que novos elementos, novos elementos associados e novas unidades adicionem suas definições ao esquema principal.

As noções de atributos, construtores e elementos estruturados são usados na definição da estrutura lógica e genérica de documentos e objetos.

Um construtor está no nível da meta-estrutura, portanto, ele não descreve a existência de relações entre estruturas dadas, mas define como os elementos são passados para a linguagem de montagem para construir uma estrutura conforme o modelo.

```
{------------------------------------------------------------------------ }
{ Document model for Logic programming environment HyperPro            }
{ Pierre Deransart / Ali Ed-Dbali / Khalid El Qorchi                     }
{ October 1st, 1997.                                                    }
{------------------------------------------------------------------------ }

STRUCTURE HyperPro;

DEFPRES HyperProP;

STRUCT

   HyperPro (ATTR First_page_number = Integer;
  First_section_number = Integer) =
BEGIN
    Document_title  = Lines;
  ? Document_date   = Lines;
  ? Authors         = LIST OF (Author);
  ? Affiliations    = LIST OF (Affiliation = Lines);
  ? Key_words    = Lines;
       Sections_seq;
    ? Bibliography   = LIST OF (Citation_biblio = RefBib);
  ? Annexes         = LIST OF (Annexe);
       END;

   Author (ATTR Author_type = Principal_author,Secondary_author) =
     Content + (Affiliation_ref)
     - (Rel_def_ref, Current_pred, Figure_ref, Formula_ref,
Section_ref, Annexe_ref, Titled_group_ref);
   ...

ASSOC
   Note = Paragraphs_seq;

UNITS
   ...
   Titled_group_ref = REFERENCE (Titled_group);
   Other_thing = NATURE;
...
END
```

Exemplo 1

Quatro construtores são usados para produzir um documento: *aggregate*, *list*, *choice*, e suas extensões *Unit* e *Schema*, *reference*, e sua variação *inclusion*. Eles introduzem grande flexibilidade às estruturas genéricas. O construtor *choice* pode representar diversos elementos diferentes. O construtor *list* permite a adição de vários elementos do mesmo tipo. Os esquemas de extensão definem atributos, elementos, elementos associados, unidades, mas eles só podem ser usados junto do `Esquema de Estrutura` que eles completam. Por outro lado, os `Esquemas de Estrutura` podem sempre ser usados sem

as extensões quando estas não estiverem disponíveis.

Os elementos cuja posição na estrutura do documento não é fixa são chamados elementos associados. Eles são definidos como estruturas em que o conteúdo pode ser organizado logicamente pelos construtores de elementos primitivos ou construídos. Pode acontecer que os elementos associados estejam totalmente desconectados do documento, como por exemplo um comentário.

Elementos associados introduzem um novo uso para o construtor *reference*. Ele serve não somente para criar elos entre os elementos da estrutura principal do documento, mas também liga os elementos associados a estrutura principal.

O modelo é suficientemente flexível para levar em consideração todas as fases da vida útil de um documento. Por exemplo, se uma estrutura genérica específica deve conter um título, uma citação, uma introdução e pelo menos dois capítulos e uma conclusão, isso significa que o documento deverá conter todos esses elementos. Quando o autor começa a escrever, nenhum desses elmentos está presente. O editor `Thot` usa esse modelo e ainda tolera documentos que não estão exatamente iguais a essa estrutura genérica. O Exemplo 1 mostra um trecho do `Esquema de Estrutura` que define o protótipo do HyperPro usando a linguagem S.

A LINGUAGEM P

Devido ao modelo adotado pelo `Thot`, a apresentação do documento está separada da sua estrutura e conteúdo. O conceito de apresentação engloba o chamado *layout* de página, composição e modelo do documento. O esquema de apresentação define o conjunto de operações que exibe o documento na tela ou o imprime. A apresentação do documento é definida genericamente por uma linguagem chamada P.

O elo entre a estrutura e a apresentação é claro: a organização lógica do documento é usada para compor a apresentação, já que a proposta da apresentação é mostrar a organização do documento. O `Thot` utiliza uma aproximação de dois níveis, onde a apresentação é descrita, primeiramente em termos abstratos, sem levar em consideração cada estrutura em particular, e então, a apresentação é realizada sem o achatamento da estrutura dada.

A descrição da apresentação também define a apresentação genérica, já que ela descreve a aparência da classe de documentos ou objetos.

Para preservar a homogeneidade entre os documentos, a apresentação é descrita como um simples conjunto de ferramentas simples que dá suporte ao *layout* de grandes documentos tão bem quanto a composição de objetos, como figuras gráficas ou fórmulas matemáticas.

Para assegurar a homogeneidade das ferramentas para documentos tanto quanto para os objetos que eles contenham, toda apresentação no `Thot` é baseada na noção de caixa, como em TEX. Uma caixa está associada a uma *string* de caractere, linha do texto, página, parágrafo, título, fórmula matemática ou tabela de células.

```
{-------------------------------------------------------------------------}
{ Document model for Logic programming environment                        }
{ Pierre Deransart / Ali Ed-Dbali / Khalid El Qorchi                       }
{ October 1st, 1997.                                                      }
{-------------------------------------------------------------------------}

PRESENTATION HyperPro;

VIEWS
  Text_view, Table_of_contents, Program_view, Comment_view, Assertion_view, Type_view;
...
DEFAULT
   BEGIN
      HorizRef : Enclosed . HRef;
      ...
      PageBreak : Yes;
   END;
BOXES
   Box_Odd_Page_number:
BEGIN
   Background : White;
   Foreground : Black;
   Fillpattern : nopattern;
   Content : VarPageNumber;
   VertPos : Top = Previous PAGE_BREAK . Bottom + 0.3 cm;
   HorizPos : Right = Previous PAGE_BREAK . Right;
   Height : 2 cm;
   Size : 11 pt;
   Font : times;
   Style : Roman;
END;
   ...
RULES
   HyperPro:
BEGIN
   Justify : No;
   Size : 12 pt;
          ...
   Paragraphs_seq:
BEGIN
          Vertpos : Top = Previous . Bottom + 15 pt;
   Justify : Yes;
   IN Program_view Visibility : 0;
   IN Comment_view Visibility : 0;
   IN Assertion_view Visibility : 0;
   IN Type_view Visibility : 0;
END;
    ...
END
```

Exemplo 2

Uma apresentação genérica define os valores dos parâmetros de apresentação, ou uma forma para calcular esses valores para a estrutura genérica. A definição dos parâmetros de apresentação é feita com a linguagem P. Um programa feito nessa linguagem, é chamado `Esquema de Apresentação`. Ele usa a mesma meta-linguagem que foi usada com a linguagem S para produzir o `Esquema Estrutural`.

O `Esquema de Apresentação` começa com a palavra-chave PRESENTATION e termina com END. A palavra-chave PRESENTATION é seguida pelo nome da estrutura genérica na qual a apresentação será aplicada. Este deve ser o mesmo nome usado no `Esquema de Estrutura`.

Para evitar ter que especificar para cada tipo de elemento definido no `Esquema de Estrutura`, valores para cada um dos numerosos parâmetros de apresentação o `Esquema de Apresentação` permite a definição de um conjunto de regras de *default*. Essas regras aplicam-se:

1. a todas as caixas de elementos definidos no `Esquema de Estrutura`;
2. nas caixas de apresentação e *layout* de página definidos nos `Esquemas de Apresentação`.

Uma regra de apresentação define parâmetros de apresentação ou funções de apresentação. As funções de apresentação são:

- criação de uma caixa de apresentação;
- estilo de quebra de linha e quebra de página;
- copiar de outra caixa.

Um parâmetro de apresentação pode ser definido pela referência ao mesmo parâmetro de outra caixa em uma árvore de caixas. A caixa de referência pode ser aquele elemento imediatamente acima da estrutura (*Enclosing*), dois níveis acima (*GrandFather*), imediatamente abaixo (*Enclosed*) ou imediatamente antes (*Previous*).

Para cada caixa e cada perspectiva, todo parâmetro de apresentação é definido somente uma vez ou explicitamente, ou pelas regras de *default*. Em contraste, as funções de apresentação não são obrigatórias e podem aparecer muitas vezes para o mesmo elemento. O Exemplo 2 define a `Estrutura de Apresentação` para o sistema HyperPro utilizando a linguagem P do `Thot`.

A LINGUAGEM T

Devido ao seu modelo de documento, o `Thot` pode produzir documentos de forma abstrata em alto nível. Essa forma, chamada de forma canônica é específica do `Thot`. Ela se adapta bem às manipulações do editor, mas não se adapta necessariamente a outras operações que possam ser aplicadas aos documentos. Por esse motivo, o editor `Thot` oferece a escolha de salvar documentos na forma canônica ou em um formato definido pelo usuário. Neste caso, o documento `Thot` é transformado pelo programa de tradução. Essa facilidade pode também ser usada para exportar documentos do `Thot` para outros sistemas usando outros formalismos.

A tradução pode ser usada para exportar documentos para formatadores básicos como TEX, LATEX, Scribe e troff. Também pode ser usado para traduzir documentos para SMGL e HTML.

Para cada documento ou classe do objetos, um conjunto de regras de tradução pode ser definido, especificando como a forma canônica deve ser transformada. Estas regras de tradução estão agrupadas nos `Esquemas de Tradução`. A mesma estrutura lógica pode ter diferentes `Esquemas de Tradução` cada um definindo regras para um formalismo diferente. Os esquemas de tradução são genéricos.

Os `Esquemas de Tradução` são escritos em uma linguagem chamada T. A gramática da linguagem T é especificada usando meta-linguagem e os esquemas de tradução são escritos usando as mesmas convenções dos `Esquemas de Estrutura`, linguagem S, e `Esquemas de Apresentação`, linguagem P.

Um `Esquema de Tradução` começa pela palavra-chave TRANSLATION, seguida do nome da estrutura genérica para qual está sendo definida e um ponto e vírgula, e termina pela palavra-chave END. O nome da estrutura genérica deve ser o mesmo da `Esquema de Estrutura`.

A tradução dos elementos que compõe um documento é feita na ordem induzida pela estrutura de árvore, exceto quando a regra Get [10] é usada para mudar a ordem de tradução. Para cada elemento, o tradutor primeiro aplica as regras específicas para o tipo do elemento antes da tradução do seu conteúdo. Se várias regras são aplicadas a esse elemento, o tradutor as aplica na ordem em que elas aparecem no esquema de tradução.

```
{-------------------------------------------------------------}
{ Translation schema to generate the HTML version from an     }
{ HyperPro document.                                          }
{ Mariza A. S. Bigonha 06/04/98 -      DCC/ICEx/UFMG          }
{-------------------------------------------------------------}

TRANSLATION HyperPro ;
RULES
   HyperPro :
        BEGIN
        Use ParagraphH for Paragraphe;
        ...
        END;
   Document_title:
        BEGIN
        Create '\12<h1 ALIGN=CENTER>';
        Create '</h1>\12' After;
        Get Document_date After;
        Get Authors After;
        END;
   Document_date:
        BEGIN
          Create '<h4 ALIGN=RIGHT> ' ;
          Create '</h4>\12' After;
        END;
   ...
END
```

Exemplo 3

Essa ordem pode ser mudada com as opções de *Attributes* e *Presentations* da regra Create [10].

O Exemplo 3 mostra um trecho de programa usando um `Esquema de Tradução` escrito na linguagem T do `Thot` para gerar uma parte do programa de um documento HyperPro.

A Linguagem A

No sistema `Thot`, a geração da aplicação baseia-se em `esquemas de aplicação` que são escritos em uma linguagem chamada A [12] (*Generation Application Language*). Esta linguagem é usada para a definição da interface gráfica e para a criação de menus os quais podem ser associados às funções padrões do sistema ou a novas funções específicas.

Uma aplicação é formada por comandos e ações. Os comandos são executados ao se escolher um item de menu e as ações são executadas quando os eventos aos quais elas estão associadas ocorrem. Ambos estão especificados nas seções DEFAULT, ELEMENTS e ATTRIBUTES. Os comandos de edição do `Thot` geram dois tipos de eventos cujos nomes diferem pela extensão:

- `.Pre`: quando o comando padrão é chamado pelo usuário antes de ser processado pelo editor e retorna um booleano;
- `.Post`: quando a ação é executada depois que o editor executou o comando padrão e não possuem valor de retorno.

Os eventos foram agrupados de acordo com os objetos aos quais eles são transmitidos: atributos, elementos, regras de aplicação específicas, documentos, visão e aplicação.

Um `esquema de aplicação` se relaciona a um `esquema de estrutura` e possui o mesmo nome deste com a extensão `.a`, ou então, é o esquema principal com o nome `EDITOR.a`, onde estão definidos os menus e comandos específicos associados.

Um `esquema de aplicação` começa com a palavra-chave APPLICATION seguida da palavra EDITOR ou do nome do `esquema de estrutura` relacionado e termina com a palavra-chave END.

Um `esquema de aplicação` pode ser composto de uma ou mais seções abaixo explicadas:

- DEFAULT: possui associação de eventos e ações e em geral se aplica a todos os tipos de elementos e a todos os atributos definidos no `esquema de estrutura` correspondente.
- ELEMENTS: contém ações que são chamadas por um elemento.
- ATTRIBUTES: define ações que são chamadas por um dado atributo.

Pelo menos uma dessas seções deve estar presente nos `esquemas de aplicação` associados.

```
APPLICATION EDITOR;

USES
{ HyperPro schema }
HyperPro,
        ...
DEFAULT
  BEGIN
   Init.Post -> HP_Event_InitHyperPro;
   ViewOpen.Post -> HP_Event_AfterOpenView;
   ...
  END;
MENUS
   Main Window:
BEGIN
File button:BNew ->  TtcCreateDocument;
        ...
END;
   Document Windows:
BEGIN
...
Present.Views toggle:TSynchro -> HP_Menu_LocChangesynchronize;
...
   HyperPro Windows:
BEGIN
...
Tools.Indexes button:BTextIndex -> TtcIndex;
Tools.Indexes button:BPredCrossRefIndex -> HP_Menu_PredCrossRefIndex;
Tools.Indexes button:BVersionIndex -> HP_Menu_VersionIndex;
Tools separator;
Tools.Indexes button:BAllIndexes -> HP_Menu_AllIndexes;
Tools.Version button:BNameVersion -> HP_Menu_NameVersion;
Tools.Version button:BClearVersion -> HP_Menu_ClearVersion;
Tools.Projection button:BManual -> HP_Menu_ManualProjection;
Tools.Projection button:BRegExpression -> HP_Menu_RegExpressionProjection;
Tools.Projection button:BVersion -> HP_Menu_VersionProjection;
Tools.Projection button:BRecursive -> HP_Menu_RecursiveProjection;
Tools.Projection button:BIndexBased -> HP_Menu_IndexBasedProjection;
Tools.Test button:BVersion -> HP_Menu_VersionTest;
Tools.Test button:BClauses -> HP_Menu_ClausesTest;
Tools.Test button:BRelation -> HP_Menu_RelationTest;
Tools.SyntaxVerif button:BVersion -> HP_Menu_VersionSyntaxVerif;
Tools.SyntaxVerif button:BClauses -> HP_Menu_ClausesSyntaxVerif;
Tools.SyntaxVerif button:BRelation -> HP_Menu_RelationSyntaxVerif;
Help_ button:BHyperProInfo ->  HP_Menu_HyperProHelp;
Help_ button:BThotInfo ->  HP_Menu_ThotHelp;
END;
END
```

Exemplo 4

O `esquema de aplicação` principal, `EDITOR.a`, possui ainda duas outras seções que não aparecem nos outros esquemas:

- USES: essa seção possui os nomes dos outros `esquemas de aplicação` bem como a lista com todos os módulos necessários à aplicação . Essa seção é opcional.
- MENUS: última seção do `EDITOR.a` que define os menus da barra de menus. Para cada item do menu, o `esquema de aplicação` associa um comando específico exceto para os menus padrão.

O conjunto de ferramentas do `Thot` permite a declaração de menus em cascata de um nível e ainda suporta a opção de vários idiomas, *multilingual dialogue*. Além dessas facilidades, o kit de ferramentas permite que o usuário execute comandos via teclado.

O exemplo abaixo mostra um trecho do `esquema de aplicação EDITOR.a` do protótipo do `HyperPro Básico`.

A API

O conjunto de feramentas do `Thot` é um conjunto de funções C que lidam com documentos estruturados no ambiente UNIX/ X Windows. Geralmente uma aplicação usa essas funções para, entre outras tarefas, criar documentos novos, modificar documentos existentes, extrair informações de documentos e mostrar partes de documentos.

O conjunto de ferramentas do `Thot` possui cerca de 200 funções agrupadas em grupos onde cada um enfatiza um aspecto diferente do documento, todas baseadas no modelo `Thot` de documentos. Os grupos são:

- Grupo de Aplicação: possui funções de gerência da API.
- Grupo de Interface: permite que a aplicação extenda ou modifique o `Thot`
- Grupo de Mensagens: permite a aplicação gerenciar mensagens e caixas de diálogo.
- Grupo de Diálogo: com estas funções a aplicação é capaz de gerenciar menus e formulários.
- Grupo de Documentos: essas funções gerenciam os esquemas e todas as ações relacionadas ao documento em si.
- Grupo Árvore: essas funções lidam com operações relacionadas a árvore de estrutura que representa a organização lógica do documento.
- Grupo de Conteúdo: possui funções que manipulam as folhas da árvore de estrutura.
- Grupo de Atributos: contém funções que lidam com os atributos dos elementos.
- Grupo de Referência: funções que manipulam as refências do documento, *links* de hypertexto.
- Grupo de Linguagem: contém funções que gerenciam o idioma usado em um texto.
- Grupo de Apresentação: lida com a apresentação específica do elemento.
- Grupo de Vistas: possui funções que manipulam as várias vistas do documento.
- Grupo de Seleção: funções que manipulam a seleção de elementos.

Essas funções são acessadas por meio da API(*Application Programming Interface*[11]). O conjunto de ferramentas do `Thot` é composto de duas bibliotecas: a biblioteca do *kernel* do `Thot` e a biblioteca do editor `Thot`. A primeira permite que a aplicação lide, de forma automática, com a estrutura lógica e o conteúdo do documento. A segunda contém todas as facilidades incluídas na primeira e ainda acrescenta funções que mostram o aspecto gráfico do documento.

4 HyperPro

O `HyperPro` [3, 5, 6] é um ambiente experimental que foi projetado para desenvolver programação lógica baseado em programação literária, especificadamente na ferramenta GrifThot. Ele oferece uma maneira de lidar com edição de texto e programação CLP (*Constraint Logic Programming*[20]). A edição de texto é feita usando o sistema `Thot`, que possui facilidades de hypertexto.

O objetivo do HyperPro é documentar programas CLP oferecendo ao usuário a possibilidade de editar, em um ambiente homogêneo e integrado, diferentes programas e versões de programas, comentários, informações de verificação formal, assim como a possibilidade de executar, depurar e testar os programas. Para isso, foi desenvolvida uma estrutura genérica de documentos para programas lógicos de acordo com a metodologia de programação lógica. Os programas são vistos como documentos executáveis. Com poucas alterações, o protótipo pode ser alterado para outras linguagens.

As funções básicas do HyperPro atualmente disponíveis são:

1. diferentes visões do documento;
2. exportações para Latex, ASCII e HTML;
3. índices manuais;
4. tabela de conteúdo.

4.1 Estrutura do Documento HyperPro

Um documento HyperPro é basicamente um documento `Thot`. Ele possui um título, pelo menos uma seção, uma tabela de conteúdo e um índice de referência cruzada.

Em um documento HyperPro, um parágrafo também pode ser uma definição de relação. Uma definição de relação é definida por um título e uma lista de pelo menos uma definição de predicado. O título é um indicador de predicado, nome do predicado e sua aridade, ou um nome.

Uma definição de predicado é composta de três itens: comentários informais, que são seqüências de parágrafos; asserções, que são seqüências de linhas de texto opcionais e um conjunto de cláusulas, que podem ser cláusulas Prolog ou `clp(FD)`.

4.2 Arquitetura do Protótipo

A Figura 1 ilustra como o HyperPro funciona. Ele utiliza-se de um editor de texto, o `Thot`, e sua API. As API's (*Application Program Interface*) do `Thot` permitem desenvolver aplicações específicas que potencialmente podem atuar na edição de um documento. O kit de ferramentas

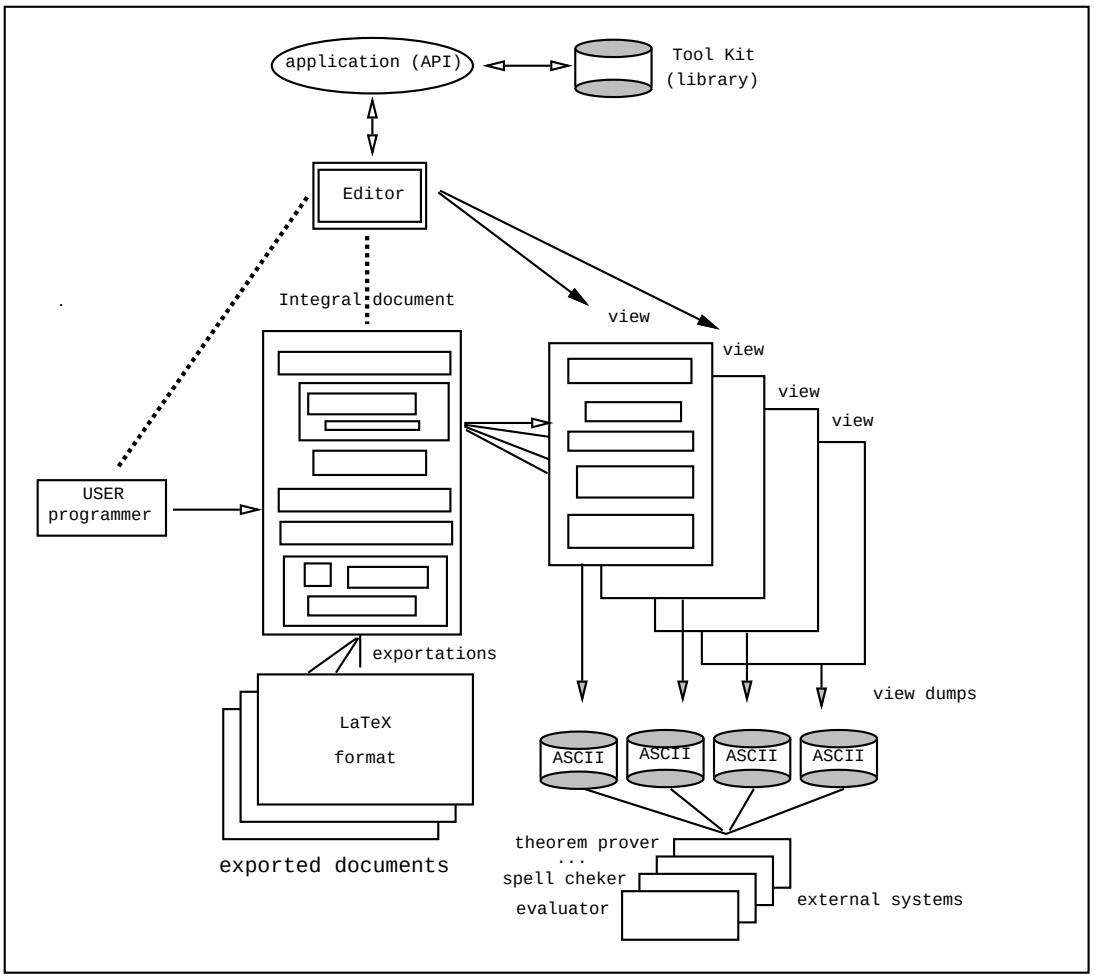

Figura 1: Arquitetura do HyperPro

do `Thot` é um conjunto de funções de edição, escritas em C, que pode ser usado na construção das API's; tais funções executam operações em ambientes estruturados Unix X-Window.

O usuário entra com programas escritos na linguagem S e na linguagem P que definem a estrutura genérica do documento. O documento todo é visualizado em uma só vista integral do documento. Além desta vista, mais quatro vistas podem ser definidas. HyperPro permite definir diferentes esquemas de exportação de documentos na forma canônica para outros tipos de formalismos, por exemplo, LaTex, ascii, etc.

4.3 Funcionalidades do HyperPro

As principais funções do HyperPro são os índices e as vistas de diferentes partes do documento associados a diferentes utilidades tais como, teste de programas e verificação sintática de linguagens lógicas.

Vistas:

O documento pode ser visto por meio de várias perspectivas chamadas vistas, que são especificadas na apresentação genérica. Essas vistas são formas de visualização de partes específicas do programa que são úteis ao programador durante o estágio do desenvolvimento da aplicação, por exemplo, a parte dos comentários.

O documento todo é visualizado em uma única vista, denominada vista integral do documento. As vistas podem ser abertas simultaneamente e são automaticamente sincronizadas. Ao invés de escrever na vista integral, o usuário pode editar em uma vista específica, e a informação aparece nas demais vistas abertas. Quatro tipos de vistas foram especificadas: comment_view, assertion_view, typing_view e program_view além da tabela de conteúdos. As Figuras 2, 3, 4, 5, 6 mostram as vistas dos comentários, asserções, tipos, tabela de conteúdos e programa, respectivamente.

Comment_View : Vista dos comentários que permite ao usuário ver apenas os comentários relativos às definições de predicado.

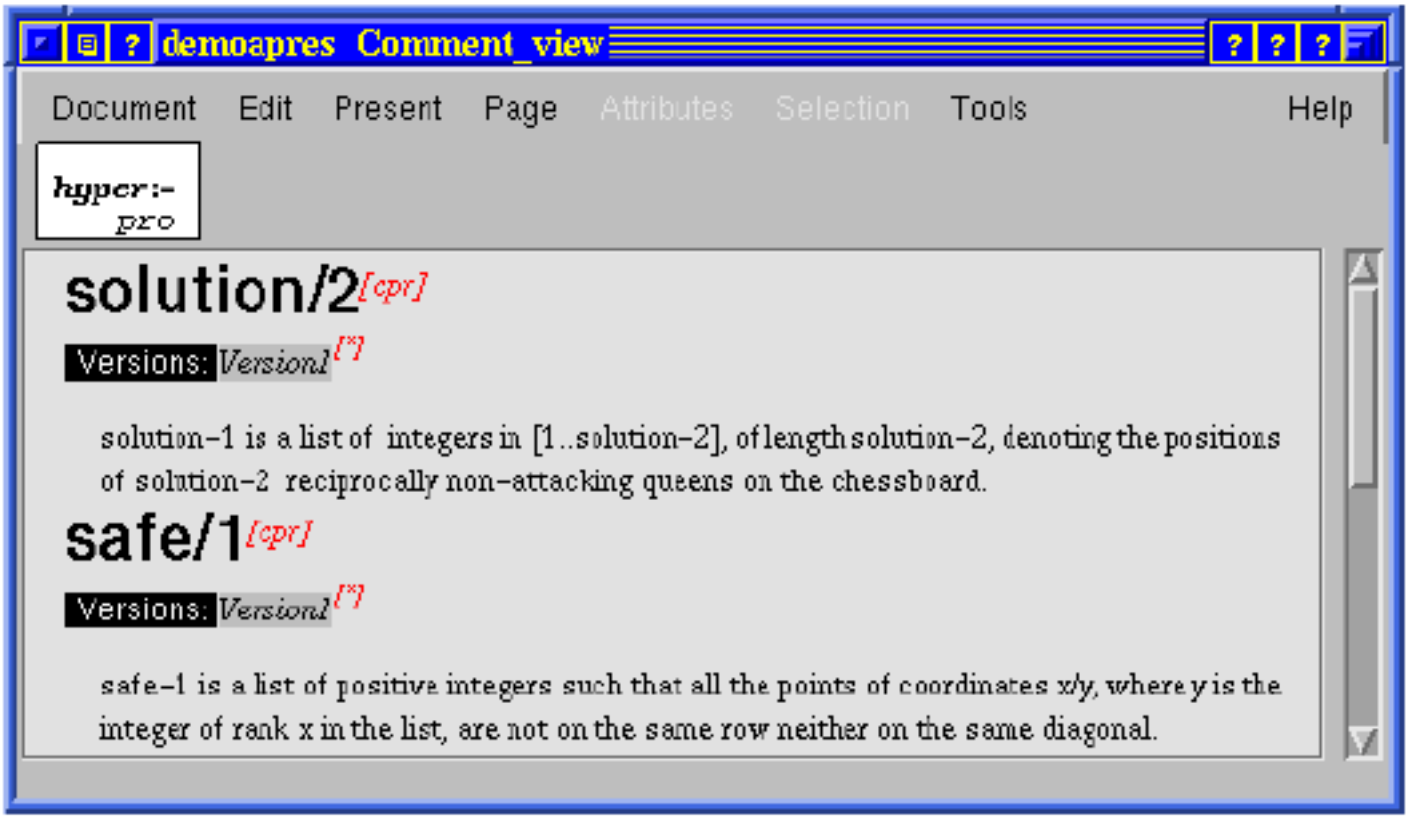

Figura 2: Visão de Comentários

Assertion_View : Vista das asserções que permite o usuário visualizar exclusivamente as partes de asserções relativas às definições de predicado.

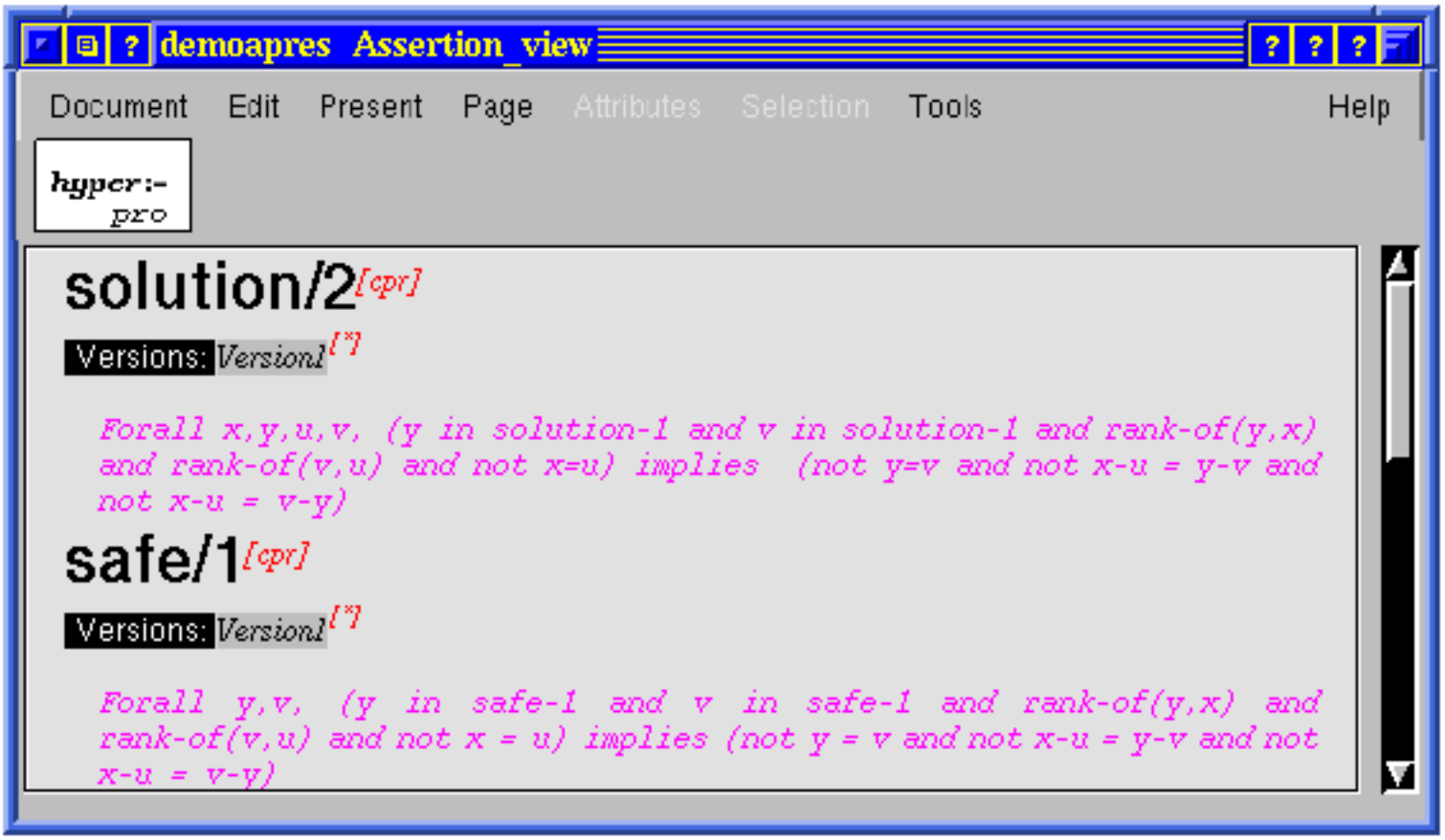

Figura 3: Visão de Asserção

Typing_View : Vista de tipos que permite o usuário visualizar apenas os tipos relacionados às definições de predicado. A vista está sendo mostrada na Figura 4.

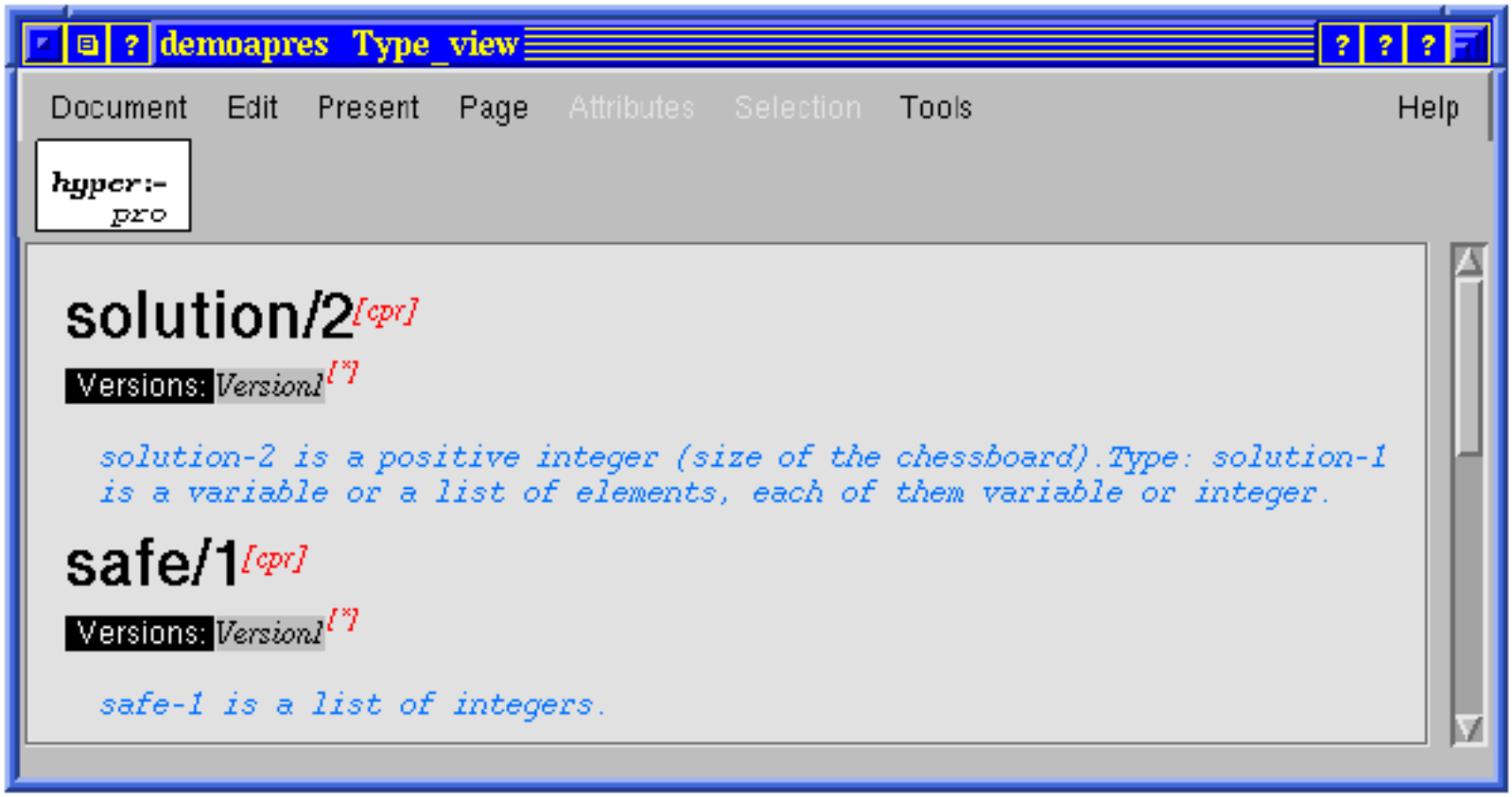

Figura 4: Visão de Tipos

Table_of_contents : Vista do programa que permite que o usuário veja a tabela de contúdos. A vista está sendo mostrada na Figura 5

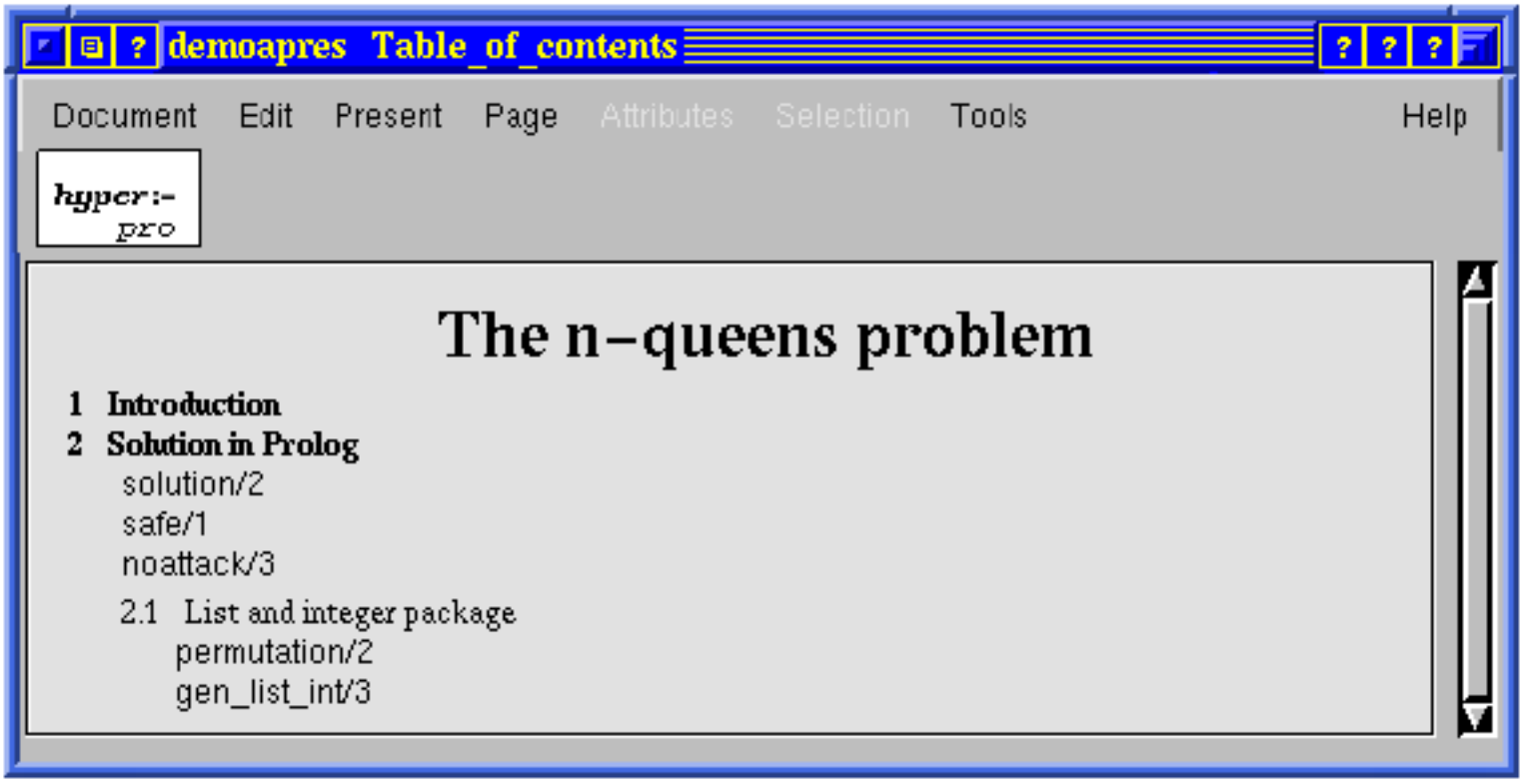

Figura 5: Visão da Tabela de Conteúdos

Program_View : Vista do programa que permite o usuário ver somente as partes de *clauses* e definição de predicados. A vista está sendo mostrada na Figura 6

O usuário pode especificar a visibilidade de um elemento e também a apresentação de certos elementos que aparecerão em cada vista.

Uma projeção mostra partes selecionadas de um documento em uma vista separada. O processo de seleção depende da projeção desejada. As projeções diferem das vistas no processo de seleção, as vistas, já estão incorporadas no `Thot` e as projeções têm que ser implementadas.

adjfafjlaflkjdffhdjhdfjksfhdk

Figura 6: Visão de Programas

Programas e Versões

Um programa é um conjunto de pacotes de cláusulas. Um documento pode conter diferentes programas. O usuário decide como o documento estará organizado, definindo seus programas por meio das seções. O usuário pode definir seus programas selecionando convenientemente no documento suas definições de predicado e colocando referências.

O HyperPro permite que o usuário teste manualmente e automaticamente seu programa.

O usuário pode definir para qualquer relação de definição, diferentes versões de uma definição de predicado que é documentada e gerenciada com as mesmas utilidades usadas para definir programas. De fato, versões de programas são programas que diferem em pelo menos uma cláusula.

Índices

O HyperPro possui o seguinte tipo de índice: índice de referência cruzada, que indica onde a relação aparece no documento, onde a sua definição de predicado é encontrada e onde a relação é usada em outros programas ou versões no documento. O índice de programas e versões que mostra onde o programa e suas versões foram primeiramente definidos está sendo projetado pela equipe francesa.

Conversões e Exportações

O HyperPro é um sistema aberto, o que significa que ele pode trocar documentos com outros sistemas através de um mecanismo de exportação.

O HyperPro produz documentos em um nível abstrato chamado de forma canônica. Pode ser definida uma série de regras de tradução de documentos. Na versão atual, foram definidos dois esquemas diferentes de tradução: exportação de documentos para LaTeXe exportação de documentos para ASCII. Também foram feitos esquemas de exportação de certas partes do documento: exportação do programa, exportação dos comentários informais, exportação das asserções e exportação dos tipos.

Os arquivos de tradução (.T) contém as regras de tradução que convertem o documento do formato PIVOT (.PIV) para os diferentes formalismos.

4.4 HyperPro Básico

Em janeiro de 1997, o grupo de pesquisadores do `HyperPro` se reuniu na França para definir os novos runos do projeto. Desse encontro surgiu a base para uma reformulação dos esquemas de estrutura, apresentação e tradução do `HyperPro` que foi chamado de `HyperPro Básico`. Também foram decididas as utilidades e funcionalidades que o sistema deveria oferecer.

As principais funcionalidades do `HyperPro` introduzidas foram: visão de diferentes partes do documento e índices associados a diferentes funções como teste de programa e verificação sintática para CLP e programação lógica.

Como o `HyperPro`, o documento `HyperPro Básico` é basicamente um documento relatório do `Thot`. Ele deve ter um título, uma seqüência de pelo menos uma seção, uma tabela de conteúdos e um índice de referência cruzada. Opcionalmente, ele pode conter a data, o nome

dos autores e suas afiliações, palavras-chave, referências bibliográficas, anexos e índices de versões.

Um documento `HyperPro Básico` difere do documento relatório do `Thot` no sentido em que um parágrafo pode ser uma também uma definição de relação. Pelo menos uma definição de relação deve estar presente no documento `HyperPro Básico`.

Uma definição de relação é definida por um título da relação e uma lista de pelo menos uma definição de predicado. O título da relação é o indicador de predicado, que é o nome do predicado e a sua aridade, ou somente o nome, desde que o pacotes de *goals* ou diretivas possa ser visto como um caso especial de ralação.

Uma definição de predicado é formada por quatro itens: comentários informais, tipos do predicado, asserções e pacotes de cláusulas. Comentários informais são uma seqüência de parágrafos. Asserções e tipos são seqüências de linhas de texto e são opcionais.

Os pacotes de cláusulas podem ser cláusulas Prolog ou CLP(FD), Clásulas incluem diretivas, *goals*, fatos e regras. Os predicados no corpo da clásula podem ou não ter referências a relações que os definem. Isto realmente definirá a versão atual de um programa no documento `HyperPro Básico`.

Pelo menos uma definição de predicado é obrigatória em cada definição de relação.

A árvore de estrutura do `HyperPro Básico` está mostrada no Apendice A

5 Projeções

Uma das funcionalidades determinadas para o `HyperPro Básico` são as projeções.

Uma projeção mostra partes selecionadas de um documento em uma vista separada. O processo de seleção depende da projeção desejada. As projeções diferem de vistas pré-definidas no processo de seleção. Além disso, vistas são estáticas e já estão incorporadas no `Thot`, as projeções são dinâmicas e devem ser implementadas no `HYperPro Básico`.

Em um ambiente integrado de programação e documentação, é importante que o usuário tenha uma maneira de testar seu programa separado do resto do documento. Daí surge a idéia da projeção. Uma projeção é uma vista do documento com partes selecionadas pelo usuário.

O `HyperPro Básico` oferece cinco projeções diferentes: a projeção manual, a projeção recursiva, a projeção de versões, a projeção automática (*por expressão regular*) e a projeção baseada em índices.

Projeção Manual: o usuário pode selecionar qualquer parte do código para ser vista separadamente.

Projeção Automática: o sistema pede para que o usuário entre com uma expressão regular e o `HyperPro Básico` mostra em uma vista separada todas as porções do documento em que esta expressão aparece.

Projeção Recursiva: o usuário seleciona uma ou mais relações e o `HyperPro` mostra em uma vista separada todos os pacotes de cláusulas CPR/PR (*Current Predicate Reference/ Predicate Reference*) nas quais a relação está inserida.

Projeção Baseada em Índices: essa projeção deverá permitir ao usuário selecionar qualquer entrada de índice e deverá mostrar em uma vista separada todas as partes do documento relacionadas ao índice.

5.1 Implementação

Para implementar essas projeções nós tínhamos pelo menos duas formas de fazê-las. A primeira, cuja solução não era trivial, consistia na adaptação da cópia de uma das funções da biblioteca do `Thot` e depois incorporar a versão modificada na biblioteca do `HyperPro`.

A segunda consistia na adição de atributos globais, por exemplo `elemento_visível`, ao esquema de estrutura do `HyperPro Básico` e na adição de vistas para as projeções no esquema de apresentação do `HyperPro Básico`.

Como a apresentação do `HyperPro Básico` é programada na linguagem P do `Thot` e para as projeções do `HyperPro Básico`, o controle de visibilidade deve ser codificado no esquema de apresentação, a regra de apresentação do atributo `elemento_visível` deve ter uma visibilidade maior que a sensibilidade da projeção. Assim, para tornar um elemento visível na projeção era necessário ligá-lo a um atributo visível, por exemplo o atributo `elemento_visível`. Essa solução era mais fácil de implementar porém apresenta dois problemas:

1. ela implica em modificar o esquema de estrutura. Se os esquemas de estrutura e apresentação são modificados, surgem alguns problema em abrir documentos antigos e isso é desconfortante por parte do usuário.

2. É necessário declarar , no esquema de apresentação n vistas para n tipos de projeções ou qualquer vista que for aberta dinamicamente. Outro problema é que não poderá haver simultaneamente duas instâncias da mesma vista. Por exemplo, não poderá ter duas projeções em versões diferentes.

Nós escolhemos implementar a primeira opção [1] pensando que seria melhor já que não precisávamos fazer mudanças nos esquemas de estrutura e apresentação, já definidos na época. Contudo, a primeira opção não funcionou pois foi difícil entender e modificar o código API (*Aplication Program Interface*) do `Thot`. Então decidimos que seria melhor implementar a segunda opção.

Para implementar essa opção, nós fizemos algumas mudanças nos esquemas de estrutura e apresentação. No esquema de estrutura, nós incluímos cinco variáveis representando o atributo `elemento_visível`, um para cada projeção.

```
ATTR
    Manual_proj_visible         = Yes;
    Recursive_proj_visible      = Yes;
    Reg_expression_proj_visible = Yes;
    Index_based_proj_visible    = Yes;
    Version_proj_visible        = Yes;
```

No esquema de apresentação, nós incluímos a declaração de cinco vistas, na seção `VIEWS`, correspondendo as cinco projeções apresentadas anteriormente. Definimos a visibilidade do `HyperPro` em zero para essas vistas e, por fim, definimos a sensibilidade para conter o valor oito para cada vista. Para definir a sensibilidade de uma vista foi usada a função da API `TtaSetSensibility(Doc, View, S)`. Para tornar um elemento visível na projeção, usamos a função da API `TtaAttachAttribute`. Abaixo estão partes do código do esquema de apresentação onde foram efetuadas as modificações:

```
VIEWS
Text_view, Table_of_contents, Program_view, Comment_view, Assertion_view,
Type_view, Manual_projection_view, Recursive_projection_view,
Regular_expr_projection_view, Index_based_projection_view,
Version_projection_view;
...
RULES
   HyperPro:
     BEGIN
       ...
       IN Manual_projection_view Visibility : 0;
       IN Recursive_projection_view Visibility : 0;
       IN Regular_expr_projection_view Visibility : 0;
       IN Index_based_projection_view Visibility : 0;
       IN Version_projection_view Visibility : 0;
     END;
...
ATTRIBUTES
    Manual_proj_visible :
          IN Manual_projection_view Visibility : 8;
    Recursive_proj_visible :
          IN Recursive_projection_view Visibility : 8;
    Reg_expression_proj_visible :
          IN Regular_expr_projection_view Visibility : 8;
    Index_based_proj_visible :
          IN Index_based_projection_view Visibility : 8;
    Version_proj_visible :
          IN Version_projection_view Visibility : 8;
END
```

Depois de feitas essas modificações nos esquemas do `HyperPro Básico`, começamos a implementar as projeções.

5.2 Interface das Projeções

Nessa seção nós apresentamos as interfaces das projeções bem como os menus e submenus.

Essa interface aparece no menu `Tools` do `Thot`, com as outras facilidades do `HyperPro Básico`. Isso acontece porque o `Thot` permite que qualquer usuário inclua suas próprias facilidades usando sua caixa de ferramentas, acessível por meio do menu `Tools`.

As vistas são selecionadas no menu `Document` do `Thot`. Lá o usuário pode escolher o sub-menu `Open a view` e então, escolhe a vista desejada, incluindo as projeções.

O menu `Tools` do Thot oferece ao usuário acesso às seguintes utilidades das projeções:

⟨*Other Thot usual menu entries*⟩	Tools	
	Make indexes	▷
	Versions	▷
	Projections	▷
	Tests	▷
	Sintax verification	▷
	HyperPro preferences	

Manual
By relation (recursively)
By regular expression
By version
By Index based

Todas as utilidades da interface foram feitas via janela de comunicação, onde o usuário controla a utilidade e onde a entrada é feita. Algumas utilidades podem abrir janelas específicas para trabalharem como suas janelas de interface.

Nas próximas seções, nós descrevemos para cada projeção sua especificação, sua implementação e como usá-la, mostrando sua janela de diálogo. As projeções podem ser encontradas no endereço: {$THOTDIR}/hyperpro/src/HyperProActions.c.

5.3 Projeção Manual

5.3.1 Descrição

A projeção manual é uma vista dinâmica na qual o usuário pode trabalhar com elementos selecionados. Nesta projeção, o usuário pode selecionar várias partes do documento, chamadas elementos, e esses elementos são mostrados em uma vista separada. O usuário pode incluir e excluir qualquer elemento nessa vista.

5.3.2 Implementação

Começamos a implementação da projeção manual com a caixa de diálogo. Para construí-la, nós usamos as seguintes funções da API do `Thot`:

- TtaSetCallback⇒ passa a função de Callback.
- TtaGetViewFrame⇒ retorna à vista do documento o *frame*.
- TtaNewSeet⇒ cria a caixa de diálogo.

Depois de implementada a caixa de diálogo, começamos a fazer a projeção manual. O primeiro ponto a ser definido foi a granularidade do elemento selecionado. Ficou definido que a granularidade seria: os parágrafos e as definições de relações. Quando o usuário seleciona qualquer um desses elementos ou seus descendentes, a projeção mostra o parágrafo inteiro ou a relação de definição inteira.

Para implementar a projeção manual, usamos as seguintes funções da API do `Thot`:

- TtaDisplayMessage⇒ mostra a mensagem na janela principal do `HyperPro`.
- TtaClickElement⇒ retorna o elemento que foi selecionado.

- TtaGetTypedAncestor⇒retorna o primeiro ancestral de um dado tipo de um dado elemento.
- TtaGetAttribute⇒ retorna se o elemento tem ou não um dado atributo.
- TtaNewAttribute⇒ cria um atributo.
- TtaAttachAttribute⇒ liga um atributo criado a um elemento.
- TtaRemoveAttribute⇒ remove o atributo.

Brevemente, incluir um elemento na projeção é o mesmo que ligá-lo a um atributo, e remover o elemento da projeção é o mesmo que remover o atributo do elemento. O algoritmo usado para implementar essa projeção consiste em:

Primeiro o usuário seleciona um elemento, depois de conferir se o elemento é ou não um parágrafo ou uma definição de relação atravessa-se a árvore de estrutura abstrata do `HyperPro Básico`, verificando se o elemento tem ou não o atributo. Se não, é criado um novo attributo e ligado ao elemento.

O Apêndice B apresenta o algoritmo completo dessa projeção.

5.3.3 Como Usar a Projeção Manual

Para usar a projeção manual, o usuário deverá selecionar no menu `Tools` a projeção. Em seguida, uma caixa de diálogo como a apresentada na Figura 7 aparece com os botões: `select`, `delete`, `cancel` e `done` onde:

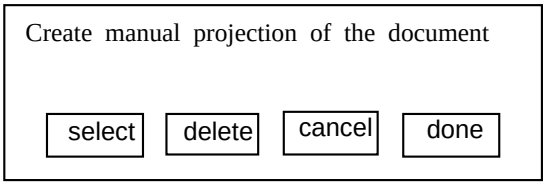

Figura 7: Caixa de Diálogo para a Projeção Manual

- select⇒ seleciona a parte do documento a ser mostrada.
- delete⇒ retira um elemento da projeção.
- cancel⇒ cancela a projeção.
- done⇒ fecha a caixa de diálogo e a projeção permanece.

Para incluir um elemento na projeção, o usuário deve selecionar o elemento com o botão `select` e automaticamente ele aparecerá na `Manual_projection_view`. É permitido incluir

vários elementos na mesma vista. O usuário pode fazer isso selecionando o botão `select` novamente.

Para incluir o elemento em uma outra vista, o sistema automaticamente fecha a projeção corrente e abre outra com o elemento desejado.

Para remover o elemento da projeção, pressione o botão `delete` e selecione o elemento a ser retirado da projeção. O elemento será retirado mas a vista permanecerá aberta porque o usuário pode querer manter os outros elementos na vista, se existirem.

5.4 Projeção Recursiva

5.4.1 Descrição

Na projeção recursiva, o usuário pode selecionar uma definição de relação e o `HyperPro Básico` mostrará em uma vista separada todos os pacotes de cláusulas relacionados àquela definição incluindo o predicado corrente e todos os predicados referenciados na corrente a qual a relação está inserida.

A projeção recursiva é útil para ajudar o usuário a detetar qual predicado tem sua referência definida ou descobrir aonde ela está definida.

5.4.2 Implementação

A projeção recursiva disponível atualmente liga o atributo a toda definção de relação portanto, não precisa de caixa de diálogo. Essa projeção funciona da seguinte forma: quando o usuário escolhe uma relação de definição, a projeção procura todas as outras relações de definição que estão relacionadas a ela e então liga o atributo a elas. Para implementar isso, algumas funções da API foram necessárias:

- TtaClickElement,
- TtaGetTypedAncestor,
- TtaNewAttribute,
- TtaAttachAttribute.

Essas funções já foram definidas na Seção 5.3.2.

Na implementação dessa projeção, também foi usada uma biblioteca que lida com listas. A lista contém todas as referências de predicados, ele começa a construir a lista de relações de definições que deve estar na projeção, encontra as referências de definições no pacote de cláusulas e as coloca na lista. As funções da biblioteca usadas são: adicionar um elemento a lista, deletar um elemento da lista e destruir a lista.

O Apêndice C mostra o algoritmo completo dessa projeção.

5.4.3 Como Usar a Projeção Recursiva

Para executar a projeção recursiva na implementação corrente, o usuário deverá selecionar no menu `Tools`, `Projections` e então, `By relation (recursively)`. Desta forma, quando o *mouse* estiver em cima do documento, ele virará uma pequena mão. Depois disso, o usuário deverá escolher uma definição de relação no documento e selecioná-la. A projeção recursiva

abrirá automaticamente com a definição de relação escolhida e todas as definições que estão relacionadas a ela.

Para completar a implementação da projeção recursiva, nosso próximo passo foi implementar uma função que permitisse ao usuário escolher entre selecionar toda a definição da relação ou somente o predicado corrente da relação.

A caixa de diálogo para esta versão é apresentada na Figura 8. Nessa caixa de diálogo, o botão `CPR` é usado para ligar o atributo ao título da relação e ao predicado corrente. O botão `All` é usado para ligar o atributo a toda relação de definição. Na verdade o botão `All` representa a implementação atual da projeção recursiva.

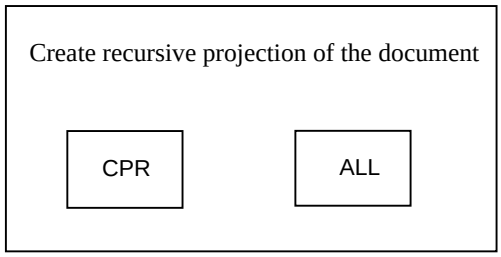

Figura 8: Nova Caixa de Diálogo para a Projeção Recursiva

5.5 Projeção de Versão

5.5.1 Descrição

O usuário pode, na projeção de versão, selecionar uma versão a ser vista separadamente. Esta vista conterá todos os pacotes de cláusulas que compõe a versão escolhida.

5.5.2 Implementação

O algoritmo de implementação da projeção de versão foi implementado da seguinte maneira: verifica se o usuário selecionou ou não em um nome de versão, caso afirmativo, a partir da raiz principal da árvore de estrutura do `HyperPro Básico` procura pelas ocorrências da versão selecionada, e retorna à raiz principal da árvore abstrata. Depois de encontrar os lugares onde as versões aparecem, descobre-se para onde ela aponta e liga o atributo ao elemento apontado.

Abaixo estão apresentadas algumas funções da API do `Thot` que foram usadas para implementar essa projeção:

- TtaGetMainRoot⇒ retorna o elemento raiz da árvore abstrataa principal.
- TtaSearchTypedElement⇒ retorna o primeiro elemento de um dado tipo.
- TtaGetFirstChild⇒ retorna o primeiro filho de um dado elemento.
- TtaGetTextLength⇒ retorna o tamanho do texto dado.
- TtaGiveTextContent⇒ retorna o conteúdo de um elemento texto.
- TtaSearchReferenceElement⇒ procura pelo próximo elemento referência.

- TtaGiveReferredElement⇒ retorna o elemento referenciado por uma dada referência.
- TtaDisplayMessage
- TtaClickElement
- TtaGetTypedAncestor
- TtaGetAttribute
- TtaNewAttribute
- TtaAttachAttribute

5.5.3 Como usar a Projeção de Versão

A projeção de versão não precisa de uma caixa de diálogo. Quando o usuário chama a função, aparece uma mensagem na janela principal do `HyperPro Básico` mostrado na Figura 9, dizendo ao usuário o que fazer.

Para usar a projeção de versão, o usuário precisa selecionar no menu `Tools`, `Projections`, a opção `By version`. Depois disso, o usuário deverá clicar em uma versão e automaticamente a projeção de versão se abrirá.

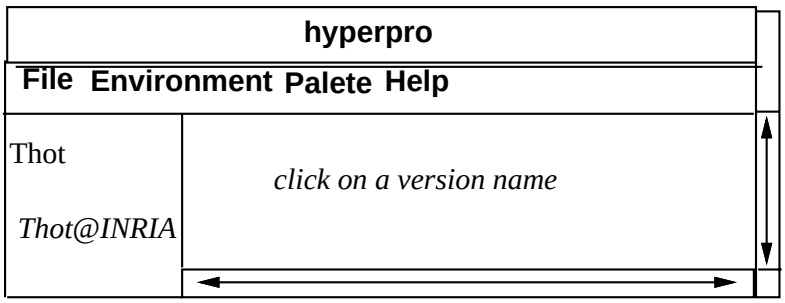

Figura 9: Mensagem da Projeção de Versões

O Apêndice C apresenta o algoritmo completo da projeção de versão.

5.6 Projeção Automática (*Expressão Regular*)

5.6.1 Descrição

A projeção automática é uma facilidade de vista onde o usuário fornece uma expressão regular e como resultado o `HyperPro Básico` mostra em uma vista separada todas as porções do documento, escolhidas pelo usuário, em que a expressão aparece. A menor granularidade para essa projeção é uma palavra.

5.6.2 Implementação

Para implementar a projeção automática, primeiramente nós procuramos nas bibliotecas do `Thot` como a função `Search` do menu `edit` era implementada. Assim, nós nos baseamos nessa função para fazermos a nossa.

Com base na caixa de diálogo da função `Search` implementamos primeiramente a nossa caixa de diálogo. Após essa implementação, usamos a função `Search` propriamente dita que retorna o elemento selecionado. Com o elemento selecionado, o usuário tem o direito de colocá-lo na projeção. Para inserirmos um elemento na projeção, pegamos o elemento selecionado e ligamos o elemento ao atributo visível.

O código desta projeção está no Apêndice E.

5.6.3 Como Usar a projeção Automática

Para usar a projeção automática, está disponível no sistema `HyperPro Básico` uma caixa de diálogo como a mostrada na Figura 10 que possui as seguintes funções disponíveis:

- Search for⇒ a expressão regular desejada para colocar na projeção.
- go⇒ acha a expressão regular procurada.
- select⇒ inclui o elemento no qual a expressão regular está inserida na projeção.
- cancel⇒ cancela a projeção.

Nessa caixa de diálogo, escolhemos uma expressão regular do documento `Hyperpro Básico` digitando na caixa de texto abaixo de *Search For*. Estão disponíveis quatro opções para direcionar o usuário na procura da palavra no documento: *after*, depois da expressão regular selecionada; *before*, antes da expressão regular selecionada; *inside definition selected*, dentro da definição selecionada e *in the document* no documento inteiro.

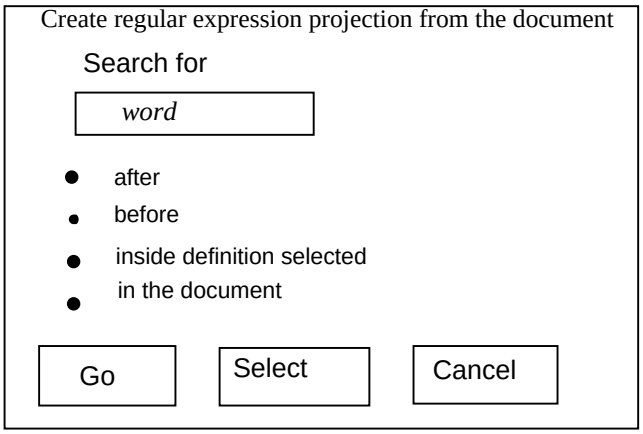

Figura 10: Caixa de Diálogo da Projeção Automática

A função que procura a expressão regular trabalha da seguinte forma: quando ela encontra a *string* desejada, ela seleciona o elemento no texto e liga o atributo ao elemento selecionado.

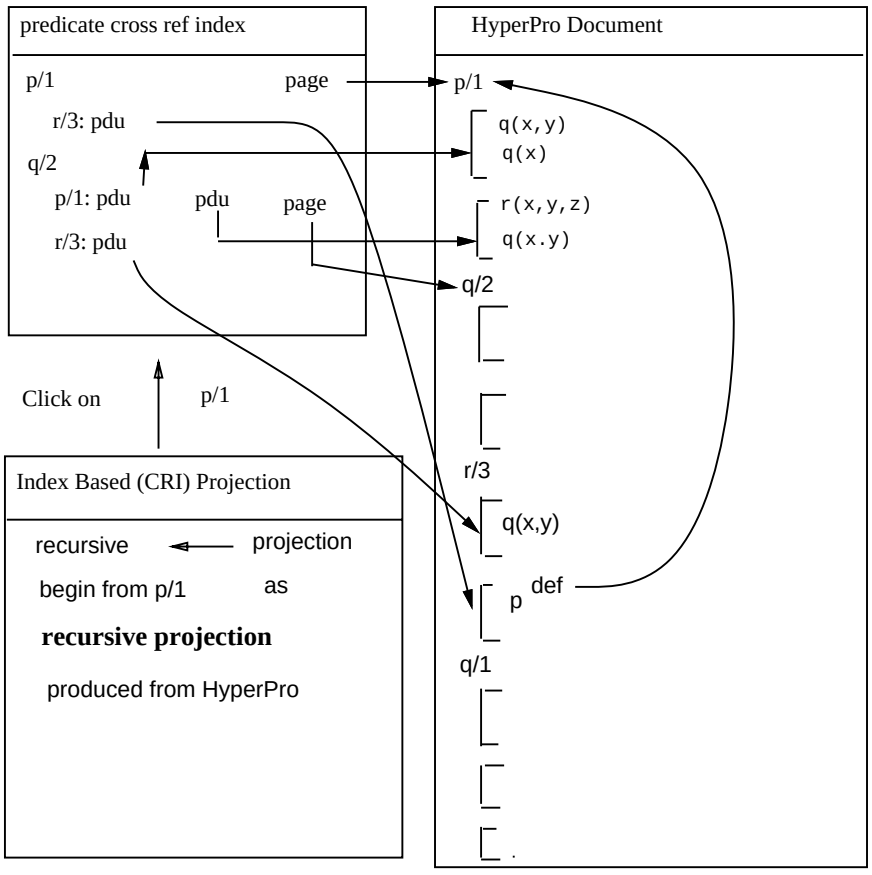

Figura 11: ESquema da Projeção de Índices Produzida pelo Índice CRI de Predicados

5.7 Projeção Baseada em Índices

5.7.1 Descrição

Nessa projeção, o usuário poderá selecionar qualquer entrada de índice e terá como resultado todas as partes relacionadas do documento em uma vista separada. Haverá dois tipos de projeções baseadas em índices: a projeção com o índice CRI (*Cross Reference Index*) de predicados explicado em 5.7.2 e a projeção produzida pelo índice de versões explicado na Seção 5.7.3.

5.7.2 Projeção Baseada em Índices Produzida pelo Índice CRI de Predicados

Nesta projeção, o usuário escolherá uma definição de predicado do índice CRI (*Cross Reference Index*) e a projeção baseada em índices mostrará, em uma vista separada, o predicado corrente da definição e cada definição na qual ela aparece.

A Figura 11 apresenta o esqueleto dessa projeção. Aqui, pdu (*predicate definition and use)* significa aonde a definição está sendo usada.

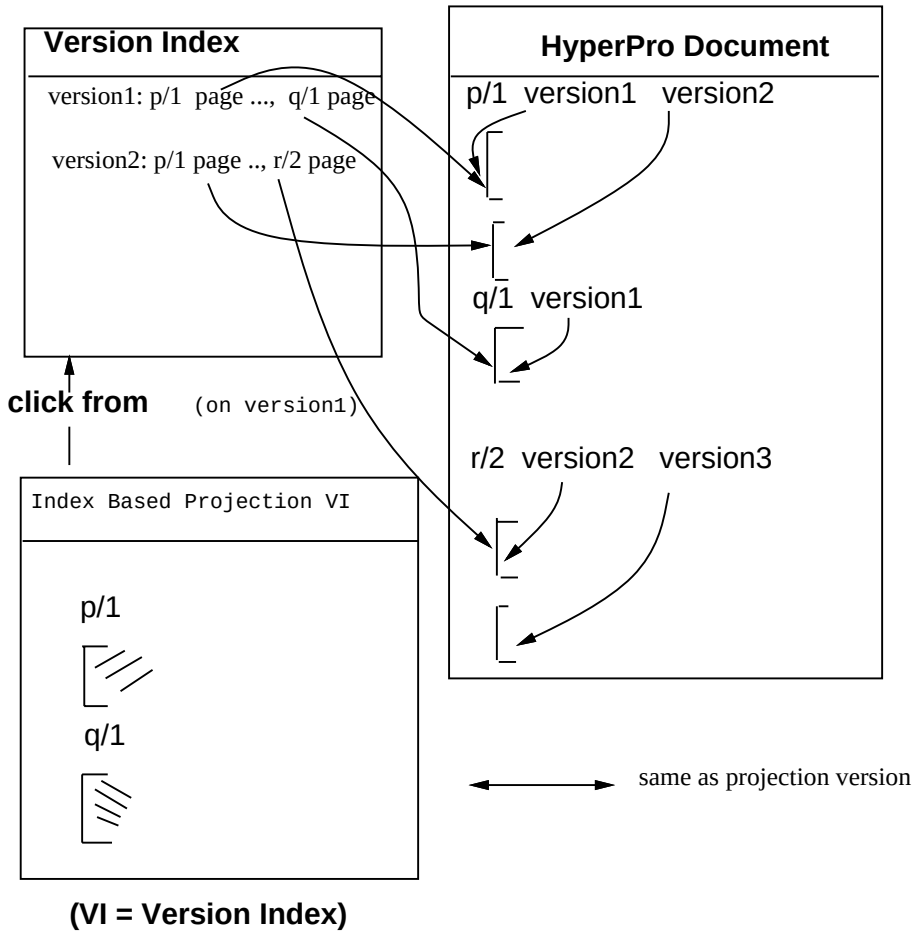

Figura 12: Esquema da Projeção de Índices Produzida pelo Índice de Versões

5.7.3 Projeção Baseada em Índices Produzida pelo Índice de Versões

Para essa projeção, o usuário precisará chamar o índice de versões. Esta vista de índice contém as versões de cada predicado definido no documento, incluindo o número da página em que eles aparecem no documento do `HyperPro Básico`. Neste índice ele selecionará a versão. Depois disso, o `HyperPro Básico` irá mostrar automaticamente em uma vista separada, como mostrado na Figura 12, todos os predicados da versão escolhida.

6 Conclusão

Durante algumas fases deste projeto, surgiram algumas dificuldades. Uma delas foi a falta de uma documentação mais elaborada. A maior parte da literatura sobre programação literária e os sistemas disponíveis foram encontradas em manuais de usuário e artigos. Estudar o manual do sistema `Thot` para que pudéssemos entender o sistema e conseguir superar essa deficiência não foi uma tarefa fácil.

Além dos problemas acima mencionados durante a implementação das projeções, ainda tivemos o problema de ter que começar do zero em novembro de 1998, pois não conseguimos

alterar o suficiente as bibliotecas do `Thot` para que as projeções ficassem prontas.

As contribuições do nosso trabalho foram:

- Instalação do Sistema `Thot`
- Instalação do Sistema `HyperPro`
- Instalação do Sistema `HyperPro Básico`
- Modificação do esquema de estrutura do `HyperPro Básica`
- Modificação do esquema de apresentação do `HyperPro Básico`
- Implementação das Projeções:
  - Manual
  - Recursiva
  - de Versões
  - Automática

Como trabalho futuro, ainda resta implementar as projeções baseadas em índices.

Referências

[1] Ribeiro, Flávia P., *HyperPro: Sistema de Programa e Documentação em um Ambiente de Programação Baseado no PAradigma de Estilo Literário*, Relatório final do projeto orientado I, ICEx, UFMG, 1998.

[2] Deransart, P., Bigonha, Roberto S, Ed-Dbali, AbdelAli, Siqueira, Jose, Bigonha, Mariza A S, *Basic HyperPro Functionalities and Utilities*, Relatório Técnico n^o 23, Departamento de Ciência da Computação, ICEx, UFMG, 1997, 17 páginas.

[3] Deransart, P., Bigonha R.S., Parot, P., Bigonha, M.A.S., Siqueira, J., *A Hypertext Based Environment to Write Literate Logic Programms*, Relatório Técnico n^o 15, Departamento de Ciência da Computação, ICEx, UFMG, 1996.

[4] Parot, Patrick, *The HyperPro Experimental System*, Relatório Técnico n^o 16/96, Departamento de Ciência da Computação, ICEx, UFMG, 1996.

[5] Deransart, P., Bigonha Roberto S., Parot, P., Bigonha, Mariza A.S., Siqueira, J., *A Literate Logic Programming System*, *Anais do I Simpósio Brasileiro de Linguagens de Programação da* SBC, Belo Horizonte, 1996, páginas:1-16.

[6] Deransart, P., Bigonha Roberto S., Parot, P., Bigonha, Mariza A.S., Siqueira, J., *A Hypertext Based Environment to Write Literate Logic Programms*, *Anais do* JICSLP'96, Bonn, Germany, setembro/1996, páginas:247-252.

[7] Quint, V., *The Thot user manual*, 1995

[8] Quint, V. and Vatton, I., *Hypertext aspects of the Grif structured editor: design and applications*, INRIA Rocquencourt, 1992, July.

[9] Quint, V., *Édition de documents structurés*, Le traitement électronique du document, 1994, 11–17, ADBS éditions, October.

[10] Quint, V., *The Languages of Thot*, Internal report, INRIA-CNRS, 1992, May.

[11] Quint, V. and Vatton, I., *The Thot Tool Kit API*, July 10, 1997.

[12] Quint, V. and Vatton, I., *The Thot Application Generation Language*, May 7, 1997.

[13] Deransart, Pierre and Ed-Dbali, Abdelali and Cervoni, Laurent, Springer Verlag, *Prolog, The Standard; Reference Manual*, 1996.

[14] Knuth, Donald D., *Literate Programming*, The Computer Journal, Vol. 27, No. 2, 1984, pp. 97-111.

[15] Knuth, Donald, *The* `web` *System of Structured Documentation*, Technical Report 980, Stanford Computer Science, Stanford, California, September 1983.

[16] Ramsey Norman, *Literate-Programming Tools Can be Simple and Extensible*, Departament of Computer Science, Princeton University, November 1993.

[17] Ramsey Norman, *Literate Programming Simplified*, IEEE Software, V.11(5), 97-105, September 1994.

[18] Ramsey Norman, *The noweb Hacker's Guide*, Departament of Computer Science, Princeton University, September 1992 (Revised August 1994).

[19] Ramsey Norman, *Literate Programming: Weaving a language-independent Web*, Communications of the ACM, 32(9): 1051-1055, September 1989.

[20] Leler, William, *Constraint Progamming Languages: their specification and generation*

[21] Lamport, Leslie, *LATEX: A Document Preparation System*, Editora Addison-Wesley Pub Co

[22] Venetianer, Tomas, *HTML - Desmistificando a Linguagem da Internet*, Editora mcGraw-Hill Ltda, São Paulo.

[23] Hennessy, John L., Patterson, David A., *Computer Architecture: A Quantitative Approach*, 2nd edition.

A Árvore de Estrutura de um Documento HyperPro

A seguir, será mostrada a figura da árvore de estrutura de elementos do `HyperPro`. Esta figura foi feita pelo Professor AbdelAli Ed-Dbali.

s

B Projeção Manual

B.1 Código da Projeção Manual

```
void HP_Util_CreateManualProjectionRun()
{
   Document document;
   Element element, relationdefinition, paragraph;
   int firstCharacter, lastCharacter;
   Attribute attribute;

   TtaDisplayMessage(INFO, TtaGetMessage(MessageTable,
                                         MESSAGES_CREATEMANUALPROJECTION));
   TtaClickElement(&document, &element);
   if (!document || !element)
      return;

/* Set global types,... for this document   */

   HP_Util_Setup(document);

/* If there is any ancestor that is paragraph or
   if there is any ancestor that is relation definition, catch it */

   paragraph = TtaGetTypedAncestor(element, type_Paragraph);
   relationdefinition = TtaGetTypedAncestor(element, type_Relation_def);
   if ((!paragraph) && (!relationdefinition))
      return;
   if (relationdefinition) {
      attribute = TtaGetAttribute(relationdefinition, attr_Manual_projection_visible);
      if (!attribute){
         attribute = TtaNewAttribute(attr_Manual_projection_visible);
         TtaAttachAttribute(relationdefinition, attribute, document);
      }
   }
   if (paragraph){
      attribute = TtaGetAttribute(paragraph,attr_Manual_projection_visible);
      if (!attribute){
         attribute = TtaNewAttribute(attr_Manual_projection_visible);
         TtaAttachAttribute(paragraph, attribute, document);
      }
   }
   HP_Util_OpenProj(document, 'M');
}

void HP_Util_DelElemFromManProj()
{
   Document document;
   Element element, relationdefinition, paragraph;
   int firstCharacter, lastCharacter;
   Attribute attribute;
```

```
   TtaDisplayMessage(INFO, TtaGetMessage(MessageTable, MESSAGES_CREATEMANUALPROJECTION));

   TtaClickElement(&document, &element);

   if (!document || !element)
      return;


/* Set global types,... for this document
*/

   HP_Util_Setup(document);


   /* ************** tit!
 PRINT("=====");
   TYPE(element);
   */
   paragraph = TtaGetTypedAncestor(element, type_Paragraph);
   relationdefinition = TtaGetTypedAncestor(element, type_Relation_def);

   if ((!paragraph) && (!relationdefinition))
      return;

   if (relationdefinition) {

      /* ************** tit!
    PRINT("~~~~~");
      TYPE(relationdefinition);
      */
      attribute = TtaGetAttribute(relationdefinition, attr_Manual_projection_visible);

      if (attribute){
         TtaRemoveAttribute(relationdefinition, attribute, document);
      }
   }
   if (paragraph){

      /* ************** tit!
    PRINT("-----");
      TYPE(paragraph);
      */
      attribute = TtaGetAttribute(paragraph,attr_Manual_projection_visible);

      if (attribute){
         TtaRemoveAttribute(paragraph, attribute, document);
      }
   }
   HP_Util_OpenProj(document, 'M');
}
```

```
   void HP_Util_RemoveAttributes(Document document, AttributeType AttributeToRemove)
   {
      Element root, element;
      Attribute attribute;

      attribute = NULL;
      root = TtaGetMainRoot(document);

      fprintf(stderr, "Type of the attribute : %s\n", TtaGetAttributeName(AttributeToRemove));
      fprintf(stderr, "Type of the attribute : %s\n", TtaGetAttributeName
(attr_Manual_projection_visible));

   /* Remove attribute (if any) from the root of the document */
      attribute = TtaGetAttribute(root, AttributeToRemove);
      if (attribute)
         TtaRemoveAttribute(root, attribute, document);

    /* Do the same for the rest of the tree */
      TtaSearchAttribute(AttributeToRemove, SearchInTree, root, &element, &attribute);
      /* ************** tit!
    PRINT("Attribute : ");
      ATTTYPE(attribute); PRINT("associated to element : ");
      TYPE(element);
      */
      while (attribute){
         TYPE(element);
         TtaRemoveAttribute(element, attribute, document);
         attribute = NULL;
         TtaSearchAttribute(AttributeToRemove, SearchForward, root, &element, &attribute);
      }
 }
```

B.2 Caixa de Diálogo da Projeção Manual

```
   {
      char *buf, *aux_buf;

   /*  let's reserve 1 element int the stack,  */
   /*  (passing the proper callback function)   */
      HPCreateManualProjectionDialogBase = TtaSetCallback(HP_Util_CreateManualProjectionCallback,
      1);

   /*  do create the sheet with buttons 'Select', 'Delete', 'Done' and 'Cancel' */
      buf = (char*)malloc(strlen(HPC_Util_GetMessage(MESSAGES_SELECT)) + strlen
      (HPC_Util_GetMessage(MESSAGES_DELETE)) + strlen(HPC_Util_GetMessage
      (MESSAGES_CANCEL)) + 4);
      aux_buf = buf;
      strcpy(aux_buf, HPC_Util_GetMessage(MESSAGES_SELECT)); aux_buf+=strlen(aux_buf)+1;
      strcpy(aux_buf, HPC_Util_GetMessage(MESSAGES_DELETE)); aux_buf+=strlen(aux_buf)+1;
      strcpy(aux_buf, HPC_Util_GetMessage(MESSAGES_CANCEL));
      TtaNewSheet (HPCreateManualProjectionDialogBase, TtaGetViewFrame(document, view),
```

```
   HPC_Util_GetMessage(MESSAGES_CREATEMANUALPROJECTIONTITLE), 3, buf, 1, 2, 'L',
   D_DONE);
   free(buf);

/*    */
   TtaNewLabel (HPCreateManualProjectionDialogBase+2, HPCreateManualProjectionDialogBase,
   TtaGetMessage(MessageTable, MESSAGES_CREATEMANUALPROJECTION));
/*  popup the dialog ! */
   TtaShowDialogue (HPCreateManualProjectionDialogBase, 0);

 /* and finally remove the attribute Manual_projection_visible from all
  *  element in the document
  */
   HP_Util_RemoveAttributes(document, attr_Manual_projection_visible);
}
```

C Projeção Recursiva

C.1 Código da Projeção Recursiva

```
void HP_Util_AddRelationToProjection(Document document, List relations)
{
   Element element;
   Attribute attribute;
   fprintf(stderr, "Number of relation: %d\n", HP_Util_ListCount(relations));
/* Runs through the list and sets the attributes to the desired elements */
   element = ((Element)HP_Util_ListFirst(relations));
   while(*element){
      if (element)
      {
         attribute =
 TtaGetAttribute((Element) *element, attr_Recursive_projection_visible);
         if (!attribute){
            TtaGetAttributeName(attr_Recursive_projection_visible));
            attribute = TtaNewAttribute(attr_Recursive_projection_visible);
            TtaAttachAttribute((Element) *element, attribute, document);
         }
         element++;}
   }
}

void HP_Util_CreateRecursiveProjectionRun()
{
   Document document;
   Element element, relationdefinition, relation;
   int firstCharacter, lastCharacter;
   Attribute attribute;
   List relations, newrelations;

   TtaDisplayMessage(INFO, TtaGetMessage(MessageTable,
                                         MESSAGES_CREATERECURSIVEPROJECTION));
```

```
   TtaClickElement(&document, &element);
   if (!document || !element)
      return;

/* Set global types,... for this document  */
   HP_Util_Setup(document);

   /* Get the main Relation Definition to be in the projection */
   relationdefinition = TtaGetTypedAncestor(element, type_Relation_def);
   if (!relationdefinition)
      return;

   /* Start building the list of Relations Definition that must be in the projection*/
   relations = HP_Util_ListNew();
   newrelations = HP_Util_ListNew();
   HP_Util_ListAdd(newrelations, (ListData)relationdefinition);
   while (relation = (Element) (*(HP_Util_ListFirst(newrelations)))){
      HP_Util_ListAdd(relations, (ListData)relation);
      HP_Util_ListDel(newrelations, HP_Util_ListSearch(newrelations,
                                              (ListData)relation));
     /* Find the definitions references in the packet of clauses and
put them in the newrelations list*/
      HP_Util_AddCallsFor(document, relation, relations, newrelations, 'D');
   }
   /* Set the attributes to show the elements */
   HP_Util_AddRelationToProjection(document, relations);
   HP_Util_ListDestroy(relations);
   HP_Util_ListDestroy(newrelations);

 /* Open the corresponding projection view */
   HP_Util_OpenProj(document, 'R');
}
```

D Projeção de Versão

D.1 Código da Projeção de Versão

```
void  HP_Util_ShowVersion(Document document, char* VersionName, ElementType type,
ElementType type_bar)
{
   Element search, basic, remember, ref, target, element, relationdef;
   Language language;
   char* vname;
   int size;
   char targetDocumentName;
   Document targetDocument;
   Attribute attribute, refAttribute;
```

```
      search = TtaGetMainRoot(document);
      search = TtaSearchTypedElement(type, SearchForward, search);
      while (search)
      {
      /* setup next step...
       * doing that now because we're maybe going to kill the 'search' element
       * which is the base of the next search
       */
         remember = search;

         search = TtaSearchTypedElement(type, SearchForward, search);

      /* Now going down to basic text to scan
       * the Version name...
       */
      TYPE(remember)
         basic = TtaGetFirstChild(remember);
       TYPE(basic);

       if (!basic)
            continue;


         size = TtaGetTextLength(basic) + 1;
         vname = (char*)malloc(size);
         TtaGiveTextContent(basic, vname, &size, &language);

      fprintf(stderr, "Version Name: %s & %s\n", VersionName, vname);

      /* Is the Version we found the one to be projected ?
       */

         if (!strcmp(VersionName, vname)) {
    PRINT("Entrei...\n");

     refAttribute = TtaGetAttribute(remember, attr_Version_ref);
     TtaGiveReferenceAttributeValue(refAttribute, &ref, &targetDocumentName, &targetDocument);
TYPE(ref);
     target = TtaGetLastChild(ref);
     target = ref;
          TYPE(target);
            attribute = TtaGetAttribute(target, attr_Version_projection_visible);
            if (!attribute){
               attribute = TtaNewAttribute(attr_Version_projection_visible);
               TtaAttachAttribute(target, attribute, document);
            }
            relationdef = TtaGetTypedAncestor(target,  type_Relation_def);
            element = TtaGetFirstChild(relationdef);
            attribute = TtaGetAttribute(element, attr_Version_projection_visible);
            if (!attribute){
               attribute = TtaNewAttribute(attr_Version_projection_visible);
               TtaAttachAttribute(element, attribute, document);
            }
```

```
      }

   }
}


void HP_Util_CreateVersionProjectionRun()
{
   Document document;
   Element element, version;
   Attribute attribute;
   char* vname;
   int size;
   ElementType type_version, type_versionsbar;
   Language language;


   TtaDisplayMessage(INFO, TtaGetMessage(MessageTable, MESSAGES_CHOOSEVERSIONPICK));

   TtaClickElement(&document, &element);

   if (!document || !element)
      return;


/* Set global types,... for this document
*/

   HP_Util_Setup(document);


   /* Looking for the Version we clicked on...
 */

   version = TtaGetTypedAncestor(element, type_Version);
   if (version) {
      type_version = type_Version;
      type_versionsbar = type_Versions_bar;
   }

   version = TtaGetFirstChild(version);

   size = TtaGetTextLength(version) + 1;
   vname = (char*)malloc(size);
   TtaGiveTextContent(version, vname, &size, &language);

   fprintf(stderr, "Version Name %s\n", vname);

   HP_Util_ShowVersion(document, vname, type_version, type_versionsbar);
     /* At\'{e} aqui funciona  */

   /* Open the corresponding projection view */
   HP_Util_OpenProj(document, 'V');
```

```
PRINT("Abri a projecao");

   /*
    *   change view title name
    */

      {
         char *buf, *docname;
         docname = TtaGetDocumentName(document);
         buf = (char*)malloc(strlen(docname) + strlen(vname) + 12);
         sprintf(buf, "%s (version %s)", docname, vname);
         TtaChangeViewTitle (document, VersionProjectionView, buf);
         free(buf);
      }
      free(vname);
   }


   void HP_Menu_About(Document document, View view)
   {
      fprintf(stderr, "About...\n");
   }

   void HP_Menu_Info(Document document, View view)
   {
      fprintf(stderr, "Info...\n");
   }
```

E Projeção Automática

E.1 Código da Projeção Automática

```
{
   PtrElement          pFirstSel;
   PtrElement          pLastSel;
   PtrDocument         pDocSel;
   int                 firstChar;
   int                 lastChar;
   PtrElement          pCurrEl;
   boolean             selectionOK;
   boolean             found, stop;
   boolean             foundString;
   boolean             error;


   error = FALSE;
 switch (ref)
       {
  case NumZoneTextSearch:
     strcpy (pSearchedString, txt);
     if (strcmp (pSearchedString, pPrecedentString) != 0)
```

```
      {
  ReplaceDone = FALSE;
  strcpy (pPrecedentString, pSearchedString);
      }
     SearchedStringLen = strlen (pSearchedString);
     break;
  case NumZoneTextReplace:
     strcpy (pReplaceString, txt);
     ReplaceStringLen = strlen (pReplaceString);
     if (!WithReplace)
       {
  WithReplace = TRUE;
  DoReplace = TRUE;
  TtaSetMenuForm (NumMenuReplaceMode, 1);
      }
     break;
  case NumMenuReplaceMode:
     switch (val)
   {
      case 0:
 WithReplace = FALSE;
 AutoReplace = FALSE;
 DoReplace = FALSE;
 break;
      case 1:
 WithReplace = TRUE;
 AutoReplace = FALSE;
 DoReplace = TRUE;
 break;
      case 2:
 WithReplace = TRUE;
 AutoReplace = TRUE;
 DoReplace = TRUE;
 break;
   }
     break;
  case NumFormSearchText:
     TtaNewLabel (NumLabelAttributeValue, NumFormSearchText,
  " ");
     if (searchDomain->SDocument == NULL)
{
TtaDestroyDialogue (NumFormSearchText);
return;
}
     if (searchDomain->SDocument->DocSSchema == NULL)
return;
     if (val == 2 && WithReplace && !StartSearch){
                          Document document;
                          Element element;
                          Attribute attribute;
                          SSchema sschema;

                         AttributeType attr_Reg_expression_projection_visible;
```

```
                        int firstCharacter, lastCharacter;


                        TtaClickElement(&document, &element);

                        if (!document || !element){
         exit(0);}

                        sschema =  TtaGetDocumentSSchema(document);

                        attr_Reg_expression_projection_visible.AttrSSchema = sschema;
      attr_Reg_expression_projection_visible.AttrTypeNum =
                          HyperPro_ATTR_Reg_expression_proj_visible;
                        TtaGiveFirstSelectedElement(document, &element, &firstCharacter, &lastCharacte
  element= TtaGetParent(element);
                        attribute = TtaGetAttribute(element, attr_Reg_expression_projection_visible );

         if (!attribute){
            attribute = TtaNewAttribute(attr_Reg_expression_projection_visible);
            TtaAttachAttribute(element, attribute, document);
                        }
                       HP_Util_OpenProj(document, 'E');
                       DoReplace = FALSE;
                     }
     else if (val == 0)
return;

     selectionOK = GetCurrentSelection (&pDocSel, &pFirstSel,
&pLastSel, &firstChar, &lastChar);
     if (selectionOK)
if (pDocSel != searchDomain->SDocument)
   selectionOK = FALSE;
     if (!selectionOK && StartSearch)
       {
  pCurrEl = NULL;
       }
     else if (pDocSel != searchDomain->SDocument)
       {
  TtaDestroyDialogue (NumFormSearchText);
  return;
       }
     else if (StartSearch)
       {
  pCurrEl = NULL;
       }
     else if (searchDomain->SStartToEnd)
       {
  pCurrEl = pLastSel;
       }
     else
       {
  pCurrEl = pFirstSel;
       }
```

```
     if (ThotLocalActions[T_strsearchgetparams] != NULL)
(*ThotLocalActions[T_strsearchgetparams]) (&error, searchDomain);
     if (!error || SearchedStringLen != 0)
       {
  found = FALSE;
  if (SearchedStringLen == 0)
    {
       if (ThotLocalActions[T_strsearchonly] != NULL)
  (*ThotLocalActions[T_strsearchonly]) (pCurrEl, searchDomain, &found);
    }
  else
    {
       pFirstSel = pCurrEl;
       do
 {
    stop = TRUE;
    if (WithReplace && DoReplace && !StartSearch
&& TextOK
     && DocTextOK == searchDomain->SDocument
&& ElemTextOK == pFirstSel
&& FirstCharTextOK == firstChar
&& LastCharTextOK == lastChar)
       if (ElementIsReadOnly (pFirstSel))
  TtaNewLabel (NumLabelAttributeValue, NumFormSearchText,
   TtaGetMessage (LIB, TMSG_EL_RO));
       else if (!pFirstSel->ElIsCopy && pFirstSel->ElText != NULL
&& pFirstSel->ElTerminal
 && pFirstSel->ElLeafType == LtText)
 {
    found = TRUE;
    if (ThotLocalActions[T_strsearcheletattr] != NULL)
       (*ThotLocalActions[T_strsearcheletattr]) (pFirstSel, &found);
    if (found)
      {
 ReplaceString (searchDomain->SDocument,
pFirstSel, firstChar, SearchedStringLen,
pReplaceString, ReplaceStringLen,
      !AutoReplace);
 ReplaceDone = TRUE;
 StartSearch = FALSE;
 if (pFirstSel == searchDomain->SEndElement)
    if (searchDomain->SEndChar != 0)
       searchDomain->SEndChar += ReplaceStringLen - SearchedStringLen;
 if (!AutoReplace)
    selectionOK = GetCurrentSelection (&pDocSel, &pFirstSel, &pLastSel,
     &firstChar, &lastChar);
      }
 }

    do
      {
 if (pFirstSel == NULL)
```

```
   {
      if (searchDomain->SStartToEnd)
{
   pFirstSel = searchDomain->SStartElement;
   firstChar = searchDomain->SStartChar;
   if (pFirstSel == NULL)
      pFirstSel = searchDomain->SDocument->DocRootElement;
}
      else
{
   pFirstSel = searchDomain->SEndElement;
   firstChar = searchDomain->SEndChar;
}
   }
 else if (searchDomain->SStartToEnd)
   {
      pFirstSel = pLastSel;
      firstChar = lastChar + 1;
   }

 if (searchDomain->SStartToEnd)
   {
      pLastSel = searchDomain->SEndElement;
      lastChar = searchDomain->SEndChar;
   }
 else
   {
      pLastSel = searchDomain->SStartElement;
      lastChar = searchDomain->SStartChar;
   }
 found = SearchText (searchDomain->SDocument,
      &pFirstSel, &firstChar, &pLastSel,
      &lastChar, searchDomain->SStartToEnd,
      CaseEquivalent, pSearchedString,
      SearchedStringLen);
 if (found)
    lastChar--;

 foundString = found;
 if (found)
   {
      stop = FALSE;
      if (ThotLocalActions[T_strsearcheletattr] != NULL)
(*ThotLocalActions[T_strsearcheletattr]) (pFirstSel, &found);
   }
      }
    while (foundString && !found);

    if (found)
      {
 if (!AutoReplace)
   {
      SelectStringWithEvent (searchDomain->SDocument,
```

```
    pFirstSel, firstChar, lastChar);
     stop = TRUE;
     lastChar++;
  }
TextOK = TRUE;
DocTextOK = searchDomain->SDocument;
ElemTextOK = pFirstSel;
FirstCharTextOK = firstChar;
LastCharTextOK = lastChar;
     }
   else
     {
TextOK = FALSE;
stop = TRUE;
if (searchDomain->SWholeDocument)
   if (NextTree (&pFirstSel, &firstChar, searchDomain))
     {
stop = FALSE;
pLastSel = pFirstSel;
lastChar = 0;
     }
     }
   StartSearch = FALSE;
}
      while (!stop);
   }
 if (found)
   {
      if (!AutoReplace)
 if (searchDomain->SStartToEnd)
    TtaSetMenuForm (NumMenuOrSearchText, 2);
 else
    /* Before selection */
    TtaSetMenuForm (NumMenuOrSearchText, 0);
      StartSearch = FALSE;
      if (ThotLocalActions[T_strsearchshowvalattr] != NULL)
 (*ThotLocalActions[T_strsearchshowvalattr]) ();
   }
 else
   {
      if (WithReplace && ReplaceDone)
 TtaNewLabel (NumLabelAttributeValue,
      NumFormSearchText,
      TtaGetMessage (LIB, TMSG_NOTHING_TO_REPLACE));
      else
 TtaNewLabel (NumLabelAttributeValue,
      NumFormSearchText,
      TtaGetMessage (LIB, TMSG_NOT_FOUND));
      StartSearch = TRUE;
   }
      }
    break;
 default:
```

```
     if (ThotLocalActions[T_strsearchretmenu] != NULL)
(*ThotLocalActions[T_strsearchretmenu]) (ref, val, txt, searchDomain);
     break;
       }
}
```

E.2 Caixa de Diálogo da Projeção Automática

```
{   boolean             ok;
   PtrDocument         pDocSel;
   PtrElement          pFirstSel;
   PtrElement          pLastSel;
   int                 firstChar;
   int                 lastChar, i;
   char                bufTitle[200], string[200];
   PtrDocument         pDoc;


   pDoc = LoadedDocument[document - 1];

   ok = GetCurrentSelection (&pDocSel, &pFirstSel, &pLastSel, &firstChar, &lastChar);
   if (ok)
      if (pDoc != pDocSel)
 ok = FALSE;
   if (!ok)
     {
pDocSel = pDoc;
pFirstSel = pLastSel = pDoc->DocRootElement;
firstChar = lastChar = 0;
     }

   /* fait disparaitre les autres formulaires de recherche qui sont affiches */
#  ifndef _WINDOWS
   TtaDestroyDialogue (NumFormSearchEmptyElement);
   TtaDestroyDialogue (NumFormSearchEmptyReference);
#  endif /* _WINDOWS */

   StartSearchProj = TRUE;

   strcpy (bufTitle, TtaGetMessage (LIB, TMSG_SEARCH_IN));
   strcat (bufTitle, pDoc->DocDName);
   strcpy (string, HPC_Util_GetMessage(MESSAGES_GO));
   i = strlen (HPC_Util_GetMessage(MESSAGES_GO)) + 1;
   strcpy (string + i, HPC_Util_GetMessage(MESSAGES_SELECT));
#  ifndef _WINDOWS
   TtaNewSheet (NumFormSearchText, TtaGetViewFrame (document, view),
bufTitle, 2, string, FALSE, 6, 'L', D_CANCEL);
   TtaNewTextForm (NumZoneTextSearch, NumFormSearchText,
   TtaGetMessage (LIB, TMSG_SEARCH_FOR), 30, 1, FALSE);
   TtaSetTextForm (NumZoneTextSearch, pSearchedStringProj);
#  endif /* _WINDOWS */
```

```
{
   int                 i;
   char                s[200];

   i = 0;
   sprintf (&s[i], "%s%s", "B", TtaGetMessage (LIB, TMSG_BEFORE_SEL));
   i += strlen (&s[i]) + 1;
   sprintf (&s[i], "%s%s", "B", TtaGetMessage (LIB, TMSG_WITHIN_SEL));
   i += strlen (&s[i]) + 1;
   sprintf (&s[i], "%s%s", "B", TtaGetMessage (LIB, TMSG_AFTER_SEL));
   i += strlen (&s[i]) + 1;
   sprintf (&s[i], "%s%s", "B", TtaGetMessage (LIB, TMSG_IN_WHOLE_DOC));
   TtaNewSubmenu (NumMenuOrSearchText, NumFormSearchText, 0,
    TtaGetMessage (LIB, TMSG_SEARCH_WHERE), 4, s, NULL, FALSE);
   TtaSetMenuForm (NumMenuOrSearchText, 2);
}
   WithReplaceProj = FALSE;
   ReplaceDoneProj = FALSE;
   AutoReplaceProj = FALSE;
   strcpy (pPrecedentStringProj, "");
   SearchLoadResourcesP ();


   if (!ok)
     {
InitSearchDomain (3, searchDomainProj);
TtaSetMenuForm (NumMenuOrSearchText, 3);
     }
   searchDomainProj->SDocument = pDoc;
   TextOKProj = FALSE;
#  else  /* _WINDOWS */
   searchDomainProj->SDocument = pDoc;
   TextOK = FALSE;
   CreateSearchDlgWindow (TtaGetViewFrame (document, view));
#  endif /* _WINDOWS */
}
```

Belo Horizonte, 05 de março de 1999.

Flávia Peligrinelli Ribeiro

Mariza Andrade da Silva Bigonha