Técnicas de Inspecção de Programas para Inter-relacionar as Vistas Comportamental e Operacional

Mario M. Berón[1,3]
Pedro R. Henriques[1], Maria J. Varanda[2], Roberto Uzal[3]

[1]Departamento de Informática
Universidade do Minho, CCTC, Braga, Portugal
{marioberon|prh}@di.uminho.pt
[2]Departamento de Informática
Instituto Politécnico de Bragança,Bragança, Portugal
mjoao@ipb.pt
[3]Departamento de Informática
Universidad Nacional de San Luis, San Luis, Argentina
ruzal@uolsinectis.com.ar
Grau Académico: Doutor em Ciencias da Computação
Inicio: 03/02/2006 Fim: 03/02/2009

Resumo A Compreensão de Programas (CP) é uma disciplina da Engenharia de Software destinada a criar modelos, métodos, técnicas e ferramentas, baseadas num processo da aprendizagem e num processo de engenharia, com o objectivo de alcançar um profundo conhecimento dos sistemas de software. O processo da aprendizagem implica a análise da forma como os programadores entendem os programas. Esta temática é bem descrita no contexto dos Modelos Cognitivos. O processo de engenharia necessita de ser abordado através de tópicos como: Métodos de Extracção da Informação, Administração da Informação e Visualização de Software. A pesquisa em todas estas áreas faz com que seja possível abordar a construção de boas ferramentas de CP. Além disso, esta tarefa dá a possivilidade de inter-relacionar o domínio do programa (componentes de software) e o domínio do problema (comportamento do sistema). Esta última actividade é um dos grandes desafios no contexto da compreensão de programas.
Neste artigo, descrevemos as pesquisas e os resultados obtidos nos tópicos descritos nos parágrafos precedentes. Estas tarefas fazem parte de nossa tese de doutoramento denominada: *Técnicas de Inspecção de Programas para Inter-Relacionar a Vista Operacional e a Vista Comportamental de Sistemas de Software*.
Palavras Chaves: Modelos Cognitivos, Extracção da Informação, Visualização de Software.

1 Introdução

A compreensão de programas, traduz-se na habilidade de perceber uma secção de código escrito numa linguagem de alto nível. Um programa não é mais do

que uma sequência de instruções que serão executadas de forma a garantir uma determinada funcionalidade. O leitor dum programa, consegue extrair o significado do mesmo quando compreende de que forma o código cumpre com a tarefa para a qual foi criado.
A área de Compreensão de Programas é uma das mais importantes da Engenharia de Software porque é necessaria para as tarefas de reutilização, inspecção, manutenção, migração e extansão de sistemas de software.
Pode também ser utilizada em áreas como Engenharia Inversa ou no Ensino de Linguagens de Programação. O processo de compreensão de programas pode ter diferentes significados e pode ser visto desde perspectivas diferentes.
O utilizador pode estar interessado em como o computador executa as instruções, com o objectivo de compreender o fluxo de control e o fluxo de dados, ou pode querer verificar os efeitos que a execução do programa tem sobre o objecto que está a ser controlado pelo programa.
Considerando estes níveis de abstracção, uma ferramenta versátil de inspecção visual de código é crucial na tarefa de compreensão de programas.
Para construir uma ferramenta destas características é necessário perceber temas relacionados com os Modelos Cognitivos (MC), Extracção da Informação (EI) e Visualização de Software (VS). MC ajuda a perceber as principais estruturas mentais utilizadas pelo programador para entender programas. Desta forma é possível desenhar arquitecturas de software que procurem representar essas componentes do conhecimento. EI torna possível extrair toda a informação dos sistemas de software. Isto é necessário tanto para construir arquitecturas como para implementar diferentes vistas do sistema. Finalmente, VS permite representar a informação claramente. Esta característica é muito importante porque facilita a compreensão.
Além das facilidades de inspecção de código, consideramos que uma ferramenta de compreensão deveria possuir algumas formas de inter-relacionar o domínio do programa com o domínio do problema. Uma estratégia importante para alcançar esta relação e facilitar a compreensão, consiste em criar vários domínios intermédios. Usando estes domínios intermédios, e incorporando funções de navegação entre eles, é possível construir uma relação operacional-comportamental.
Este artigo está organizado como se explica a seguir. A secção 2, descreve os trabalhos realizados no contexto dos Modelos Cognitivos. A secção 3, apresenta as pesquisas no âmbito da Visualização de Software. A secção 4, detalha a técnica usada para extrair a informação a partir dos sistemas. A secção 5, explica as técnicas da relação operacional-comportamental. Finalmente, a secção 6 apresenta a conclusão deste artigo.

2 Modelos Cognitivos

Os Modelos Cognitivos (MC) no contexto da compreesão de programas descrevem a forma como os programadores compreendem os programas. Os MC são importantes porque permitem perceber os distintos processos mentais utilizados na compreensão de programas. Desta forma é possivel desenvolver estratégias ou

ferramentas com suporte cognitivo.
Depois de realizar um estudo do estado da arte dos MC foi possível detectar que:

- Todos os autores concordam que um programador compreende um programa quando pode relacionar o domínio do programa com o domínio do problema. Com isto queram eles dizer, que para cada comportamento do sistema de estudo, o programador deve encontrar as peças de código que se utilizaram para produzir esse comportamento.
- Existe uma necessidade de uniformizar e sistematizar os conceitos usados nesta área. Isto deve-se á existencia de conceitos que produzem ambiguidade. Esta característica dificulta a possível implementação desses conceitos numa ferramenta de compreensão. Por exemplo, o conceito de *chunk* é muito utilizado para explicar como os programadores extraem informação dos sistemas. Comtudo, as bibliografías de MC não apresentam uma definição ou método para extrair chunks. Por esta razão, a automatização deste conceito fica ao criterio do programador. Esto conduz a conflitos na interpretação do conceito.
- Não existe uma única aproximação para a aprendizagem. Isto foi claramente observado através das analises das diferentes teorias de CM. Por exemplo, o Brook afirma que o processo utilizado pelos programadores para entender programas é top-down. Em quanto Soloway mantém que este processo é bottom-up. Finalmente, existem outros autores a proporem uma meta-modelo integrado que une estas duas aproximações. Todas estas teorias apresentam resultados empíricos que suportam os seus estudos. Por esta razão esta área mostra uma grande divergência neste tópico. Actualmente o Construtivismo tem tomado relevância porque é uma teoria que admite o uso misturado de estratégias da aprendizagem.

A primeira observação foi muito útil porque possibilita-nos seleccionar um abordagem para a construção de estratégias e ferramentas de compreensão. A segunda permitiu-nos fazer algumas contribuições nesta área. A terceira confirma que a abordagem seguida para entender programas é seleccionada pelo programador.
Quanto ás nossas contribuições nesta área podemos dizer que desenvolvemos definições de conceitos mais próximas ao domínio do engenheiro de software. Desta forma eliminamos as ambiguidades produzidas pelo uso de vocabulário pedagógico. A modo de exemplo, na sub-secção seguinte apresentamos uma parte de uma sistematização dum conceito utilizado no contexto de MC.

2.1 Sistematização do Conceito de Chunk

Um chunk é um conjunto de estruturas de texto. Ou seja é uma forma de construir uma abstracção desde o código fonte do programa. Com os chunks é possível construir hierarquias de chunks. Este conceito é muito utilizado no contexto dos MC contudo as bibliografias não especificam como os chunks podem ser extraídos

desde o texto do programa. Nós pensamos que uma definição mais precisa permite pensar em algoritmos que façam esta tarefa e portanto proporemos a seguinte sistematização deste conceito.
Chunk: é um conceito que descreve conjuntos de estruturas de texto. Eles podem ser classificados em : sequencial, selecção, iteração e rotina. Nos seguintes parágrafos definimos cada uns deles.

$chunk_{seq}(stmt, n, desc)$**:** Está formado por um conjunto de n sentenças genéricas (stmt) e uma descrição (desc). Neste contexto, sentença genérica representa atribuição, chamada a função ou sentença de entrada/saída. A descrição declara a funcionalidade deste grupo de sentenças. A seguinte peça de código ilustra dois exemplos de $chunk_{seq}$ de três sentenças.

```
a=temp;  a=b; b=temp;  ....
c[k]=a[i]+b[j];
i=i+1; j=j+2;
```

O primeiro chunk é denotado como $chunk_{seq}(\{a = temp; a = b; b = temp\}, 3, <$ *instercambia o valor da variavel a e o valor da variavel b* $>)$. O segundo é definido como $chunk_{seq}(\{c[k] = a[i] + b[j]; i = i + 1; j = j + 2\}, 3, <$ *soma os elementos dum vector* $>)$. O tamanho de n é determinado pelo programador e é dependente do nível de precisão desejado.

$chunk_{sel}(chunk_{seq}(stmt, 1, desc), chunck, chunck)$**:** Os chunks selecção são úteis para representar sentenças de selecção. Eles estão compostos dum $chunk_{seq}$ que representa a condição e dois *chuncks* que representam as sentenças usadas na parte *then* e na parte *else*. A seguinte peça de código é um exemplo de $chunck_{sel}$.

```
if (f(x)==0) { a=temp; a=b; b=temp;}
 else { c[k]=a[i]+b[j]; i=i+1; j=j+2; }
```

Esta peça de código pode ser descrita usando chunks na seguinte forma:
$chunk_{sel}($
$chunck_{seq}(\{f(x) == 0\}, 1, <$ *descrição da condição* $>)$,
$chunk_{seq}(\{a = temp; a = b; b = temp\}, 3, <$ *intercambia o valor da variavel a e o valor da variavel b* $>)$,
$chunk_{seq}(\{c[k] = a[i] + b[j]; i = i + 1; j = j + 2\}, 3, <$ *soma os elementos dum vector* $>))$

O leitor pode observar que estas definições são mais precisas e possibilitam construir algoritmos para extrair chunks. Por exemplo, uma forma de extrair chunks usando estas definições pode consistir em desenvolver um analisador sintático com acções semânticas que transformem o programa fonte noutro decorado com chunks. O resto das definições de chunks e doutros conceitos de MC pode ser visto em [6]. Neste artigo não foram descritas por razões de extensão.

3 Visualização de Software

A Visualização de Software (VS) [7] [2] [4] é crucial no contexto do desenvolvimento desta tese de doutoramento. Isto se deve a que é necessário representar

os diferentes domínios através da utilização de vistas. As vistas apresentam uma perspectiva do sistema de estudo que revela algum aspecto do mesmo. Para construir vistas de sistemas úteis para a compreensão é necessário definir uma representação estrutural e uma representação gráfica. A primeira faz referência á estrutura de dados que se utilizará para armazenar a informação e criar as vistas. A segunda aborda a temática das representações visuais dessa informação. Seleccionamos como vistas importantes as seguintes:

Códigos fonte e objecto: ambos os códigos são fontes básicas de inspecção e compreensão. Normalmente o programador recorre ao código fonte para entender e modificar o sistema de estudo, por este motivo, temos que possuir estratégias para aceder ao código fonte eficientemente para facilitar o proceso de inspecção e compreensão.
O código objecto é uma vista secundária que pode ser útil para tarefas como a análise do código gerado pelo compilador.

Grafo de Tipos: permite visualizar os tipos utilizados e as dependências entre eles no sistema de estudo.

Grafo de Funções: o código fonte é uma vista de baixo nível muito complexa de analisar quando o sistema de estudo é grande. Uma vista alternativa de mais alto nível é o grafo de funções. Esta vista mostra como as funções estão interligadas estáticamente no sistema. Desta forma o utilizador pode, com uma maior probabilidade, encontrar as funções mais importantes. Esta última função pode ser facilitada através dos operadores de grafos.

Grafo de Funções de tempo de execução: o grafo de funções contêm todas as funções do sistema. Contudo, em muitas ocasiões, o utilizador necessita inspeccionar só alguma delas. Por exemplo, normalmente ele gostaria de analisar só as funções que foram utilizadas para atingir algum objectivo específico. Esta informação é fornecida por um subgrafo do grafo de funções denominado grafo de funções de tempo de execução.

Grafo de Módulos: esta vista possibilita ter uma visão mais abstracta do sistema, sendo muito útil para analisar como o sistema é construído e que módulos são os mais importantes ou os mais utilizados.

Para todas estas vistas a representação estrutural é directa quando utilizam listas e grafos para armazenar toda a informação. Podem aparecer situações que impliquem a geração de novas estruturas de dados, sendo que esses tópicos estão bem descritos e documentados em diferentes livros relacionados com essa temática. O problema aparece quando deseja-mos criar a representação visual. Neste caso é necessário determinar que informação é útil e como a mostrar. Neste contexto, o nosso trabalho consistiu em definir um conjunto de atributos que sempre deveriam visualizar-se e como eles seriam visualizados. Este trabalho foi principalmente feito para as representações de grafos. Para as representações textuais necessita-mos pesquisar mais. A modo de exemplo na sub-secção seguinte apresentamos a definição da visualização das dependências de módulos.

3.1 Dependências de Módulos

As dependências de módulos mais importantes encontradas podem ser classificadas como dependências de tipos, dados e funções. A estrutura básica para modelar esta situação é um hiper-grafo dirigido, o qual é definido como:
MDG=(P,E_1,E_2,E_3) onde

$$\mathrm{P}=\{x/\ x\ \acute{e}\ um\ m\acute{o}dulo\ do\ sistema\}\ \text{e}$$
$$E_1 = \{(x,y)\ /\ x \in \mathrm{P} \wedge y \in \mathrm{P}$$
$$\wedge\ o\ m\acute{o}dulo\ x\ usa\ uma\ fun\c{c}\tilde{a}o\ definida\ no\ m\acute{o}dulo\ y\}$$
$$E_2 = \{(x,y)/x \in \mathrm{P} \wedge y \in \mathrm{P}$$
$$\wedge\ o\ m\acute{o}dulo\ x\ usa\ um\ dado\ definido\ no\ m\acute{o}dulo\ y\}$$
$$E_3 = \{(x,y)/x \in \mathrm{P} \wedge y \in \mathrm{P} \wedge o\ m\acute{o}dulo\ x\ usa\ um\ tipo\ definido\ no\ m\acute{o}dulo\ y\}$$

Estas três principais dependências podem ser mostradas usando diferentes cores e classes de arcos. Contudo é importante caracterizar o grau de inter-dependência entre os módulos. Esta actividade pode ser feita usando a largura ou intensidade da cor. A Figura 1 mostra as dependências entre um grupo de módulos de um sistema. Sobre a esquerda pode-se ver um grafo que mostra só as dependências. Sobre a direita visualiza-se um hiper-grafo decorado. Esta vista mostra diferentes dependências em diferentes cores. Em vermelho as dependências de funções, em preto as de dados e em azul as de tipos. Também é possível visualizar arcos com larguras distintas que indicam o grau de utilização de cada um destes componentes. Neste sentido pode-se observar que o módulo 2 depende mais do que o módulo 3 se tomamos como ponto de análise as funções. Por outra parte o módulo 3 usa mais dados definidos no módulo 1 do que o módulo 2. O problema com este tipo de representação das dependências é a sobrecarga da visualização. Este inconveniente pode ser resolvido usando operações tais como filtragem, zoom, etc.
Este mesmo tipo de tarefas foram feitas para as outras vistas descritas nos parágrafos precedentes e podem ser observadas em [6].

4 Métodos de Extracção da Informação

Para extrair a informação dos sistemas utilizamos Instrumentação de Código [5]. Esta técnica consiste em inserir instruções úteis no código fonte do sistema. Selecionamos como pontos estratégicos o inicio e o fim das funções. Nesses lugares inserimos funções de inspecção. Estas funções imprimem o nome da função e alguma outra informação que o programador considere importante. Os resultados obtidos com esta aproximação não foram muito bons. Isto deve-se a que as funções do sistema podem ser invocadas muitas vezes porque em certas ocasiões elas encontram-se dentro de iterações ou formam parte dum algoritmo complexo. Isto produz uma quantidade imensa de funções que não podem ser administradas com facilidade. Por esta razão foi necessário instrumentar o código para controlar as sentenças de iteração. Para atingir este objectivo esta classe de sentenças

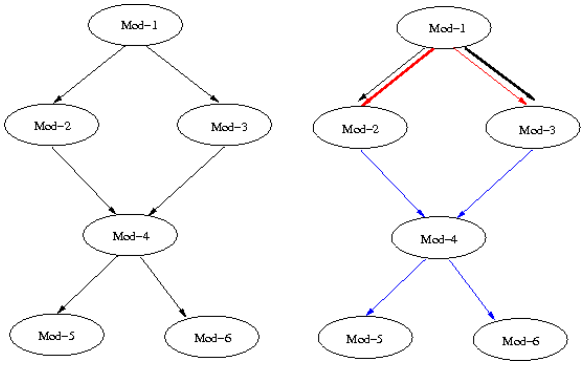

Figura 1. Dependências de Módulos

foram instrumentadas inserindo sentenças antes, dentro e depois das iterações. As sentenças previas ás iterações tem como tarefa inserir numa pilha o número que indica a quantidade de vezes que as funções dentro de uma iteração podem mostrar-se. As sentenças dentro das iteraçãoes diminuem essa quantidade em um. Finalmente as sentenças depois das iteraçãos tem como objectivo recuperar o número de vezes que as funções da iteração anterior devem-se mostrar. Este esquema permitiu reduzir a quantidade de funções reportadas pela nossa técnica de instrumentação. Além disso o número de funções pode ser controlado pelo utilizador porque ele pode decidir as vezes que as funções serão recuperadas. Este resultado foi satisfatório e possibilitou-nos avançar com o desenvolvimento da tese de doutoramento. A Figura 2.a mostra o esquema implementado para a instrumentação das funções e a Figura 2.b mostra o esquema para o controlo de ciclos.

```
void f (int a, int b)
  {
    INSPECTOR_ENTRADA("f");
    ........
    INSPECTOR_SALIDA("f");
    return;
  }
```

(a)

```
push(pila,N)
for(i=0;i<TAM;i++)
 {
  /* acciones del loop */
  .....
  decrementarTope(pila);
 }
pop(pila);
```

(b)

Figura 2. Instrumentação de Funções e Controlo de Ciclos

4.1 Outras informações extraidas do sistema

Ademais de implementar a instrumentação de código nós recuperamos toda a informação do sistema. Esta actividade permite-nos dispor de todos os componentes necessários para construir as visualizações descritas na secção 3 e inspeccionar todos as partes do código do sistema com facilidade. Além disso, nós desenhamos e criamos uma heurística de detecção de tipo de dados abstractos definidos pelo utilizador. Esta técnica baseia-se na construção dum grafo de dependências de tipos, denominado Grafo de Tipos (GT) que representa a relação "é parte de". Depois a estratégia toma cada função do sistema, extrai os tipos usados em sua assinatura e constrói um sub-grafo de GT. Este último grafo contêm os nós do GT que representam os tipos usados pela função. Nós denominamos a este conjunto de nós do grafo *nós básicos*. Além destes nós o sub-grafo contem a todos os outros nós que podem alcançar através dum caminho aos nós básicos. Finalmente, a heurística extrai os nós maximais. Se a cardinalidade deste conjunto é maior que um, então a heurística não associa a função a nenhum tipo. Noutro caso a função é associada ao tipo maximal. A Figura 3 ilustra esta técnica. Sobre a esquerda pode-se observar um grafo de tipos. No centro encontra-se uma funçaõ. Na direita esta o sub-grafo do GT para a funçao f. O tipo seleccionado pela heurística é visualizado em cor vermelha.

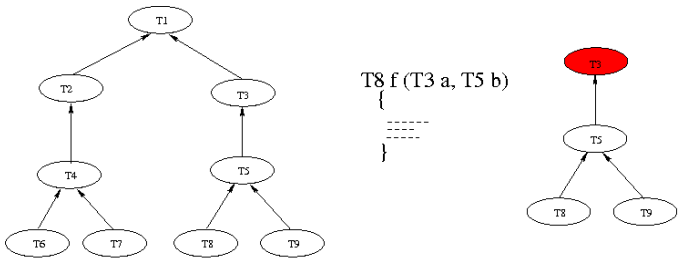

Figura 3. Heurística de detecção de grafos de tipos

5 Estratégias de Relação Operacional-Comportamental

Nesta secção descrevemos doiss procedimentos para alcançar a relação operacional-comportamental [3]. O primeiro é uma consequência de nosso esquema de instrumentação de código. O segundo é uma estratégia que usa informação estática é dinâmica para recuperar esta relação.

5.1 Efeito derivado do esquema de instrumentação de código

Esta aproximação consiste em permitir a execução paralela do sistema instrumentado e do programa que administra as funções de inspecção. Este procedimento permite visualizar, directamente, as funções que estão sendo utilizadas pelo sistema para construir a sua saída. O efeito desta estratégia é similar a uma execução passo a passo a nível de funções. Básicamente este resultado é um produto derivado da implementação do nosso esquema de instrumentação de código.

5.2 BORS

BORS (Behavioral-Operational Relation Strategy) [5] [6] tem por objectivo explicar o contexto onde as funções do sistema foram chamadas e suas funcionalidades. BORS baseia-se na seguinte observação:

"A saída dum sistema está composta de objectos do domínio do problema. Normalmente esses objectos são implementados por tipos de dados abstractos, no caso de linguagens imperativos, ou por classes, no caso de linguagens orientados a objectos. Tanto os TDAs como as classes têm objectos de dados que armazenam o seu estado e um conjunto de operações que os manipulam. Por este motivo é possível descrever cada objecto do dominio do problema utilizando os TDAs ou classes que os implementam."

BORS consta dos seguintes passos:

Detectar as funções relacionadas com cada objecto do domínio do problema: esta tarefa é feita em forma semi-automática. O utilizador selecciona os Tipos de Dados Abstractos que deseja explicar através da heurística de detecção de tipos descrita em brevemente na sub-secção 4.1. Depois todas as funções relacionadas com esses tipos são candidatas a ser explicadas e são armazenadas numa lista.

Construir a arvore de execução de funções de tempo de execução: a saída do esquema de instrumentação de código contêm toda a informação necessária (inicio e fim da execução das funções) para construir uma estrutura de dados denominada fe-tree (**F**untion **E**xecution **T**ree) [1] [5] que descreve como as funções do sistema de estudo foram executadas.

Explicar as funções encontradas no passo 1 usando a arvore construído no passo 2: combinando a informação dos passos anteriores (lista de funções a explicar e fe-tree) podem-se explicar as funções mostrando: o contexto onde foram invocadas (caminho desde a raíz até a função) e a tarefa que realizou para o sistema (sub-arvore que tem como raíz a função que está a ser explicada). Este processo aplica uma travessia por níveis sobre a fe-tree. Cada vez que se encontra um nó da fe-tree que pertence á lista de funções a explicar reporta-se o caminho desde esse nó até a raíz. Também é possível mostrar a sub-arvore que tem como raíz o nó analisado.

6 Conclução

Neste artigo apresentamos os trabalhos realizados no contexto do desenvolvimento de nossa tese de doutoramento. As pesquisas estão relacionadas com: Modelos Cognitivos, Visualização de Software e Métodos de Extracção da Informação. Cada uma destas temáticas permite-nos extrair aspectos importantes para construir estratégias e ferramentas úteis para a compreensão de programas. Como principais contribuições deste trabalho podemos mencionar: a sistematização dos conceitos utilizados no âmbito dos modelos cognitivos, a caracterização de varias vistas dum sistema de software, a elaboração dum esquema de instrumentação de código para a extracção de informação dinâmica e a criação da estratégia BORS. As primeiras duas contribuições são a nível teórico enquanto as últimas são teóricas e práticas. Os procedimentos práticos são parte dum sistema de compreensão e inspecção de programas denominado PICS. Este sistema tenta implementar inter-relação de domínios através de distintas vistas, usa BORS e instrumentação de código como estratégias para relacionar directamente os domínios de programa e do problema. Uma descrição detalhada pode ser vista em [6].

Actualmente os nossos esforços estão orientados a completar: o sistema de visualização do PICS com as caracterizações teóricas do tipo das descritas na secção 3, o esquema de instrumentação de código com a inserção de sentenças noutras partes do programa. Além disso desejamos implementar técnicas de elaboração de documentação de alto nível que usem a nossa sistematização de conceitos de Modelos Cognitivos. Por outra parte, e talvez paralelamente ás tarefas previamente mencionadas, pretendemos fazer uma análise de requisitos com o objectivo de implementar uma estratégia de relação operacional-comportamental que siga um caminho top-down.

Referências

1. Abdelwahab Hamou-Lhadj. *The Concept of Trace Summarization.*PCODA: Program Comprehension through Dynamic Analysis.(2005), 38-42.
2. Franoise Balmas, Harald Werts, Rim Chaabane. *DDGraph: a Tool to Visualize Dynamic Dependences*. Program Comprehension through Dynamic Analysis (2005), 22-27.
3. Lieberman H. and Fry C., *Bridging the Gulf Between Code and Behavior in Programming*, ACM Conference on Computers and Human Interface, (1994).
4. Maria J. Varanda and P. Henriques, *Sistematização da Animação de Programas*, Ph.D Thesis, University of Minho, (2000).
5. Mario M. Beron, Pedro Henriques, Maria J. Varanda, Roberto Uzal. *A System to understand Programs Written in C Language by Code Annotation.* European Joint Conference on Theory and Practice of Software, (2007).
6. Mario M. Beron, Pedro Henriques, Maria J. Varanda, Roberto, *Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension*, Internal Report. University of Minho - National University of San Luis, (2007).
7. Moher T. G., {*PROVIDE*}*: A Process Visualization and Debugging Environment*, IEEE Transactions on Software Engineering, 14 (1988), 849–857.