Construção de um compilador de JAVA para a linguagem formal VCt.

Eduardo Jorge Oliveira, J. Artur Vale Serrano

Resumo - Pretende-se neste artigo descrever um projecto de construção de um Sistema de Geração Automática de Código como ferramenta para projecção e construção de Técnicas de Modelação tais como DFD's (Data Flow Diagrams), STD's (State Transition Diagrams), Activity Diagrams (AD's) ou ER's (Entity-Relationship Diagrams).

Este Sistema denominado de VC (Visual Concepts) é projectado para Técnicas de Modelação que possuam notações de diagrama como os mencionados acima. Estas Técnicas de Modelação são formalmente descritas através da utilização de uma linguagem especialmente construída para tal, o VCt (Visual Concepts textual). Esta linguagem expressa a estrutura de uma Técnica de Modelação, através de regras, que serão posteriormente compiladas para código executável o qual, implementa uma ferramenta de desenho dedicada a essa Técnica de Modelação.

Abstract - In this paper we present a System for the Automatic Generation of Design Tools, for a wide variety of Modelling Techniques (MT's), such as Dataflow Diagrams (DFD's), State Transition Diagrams (STD's), Activity Diagrams (AD's) or Entity Relationship Diagrams (ER's). This System, also called VC System, is targeted to those MT's that have diagram based notations, such as the ones mentioned above. Those MT's are formally described using a language specially defined for this purpose - the VC^t (Visual Concepts textual). This language uses constraints to express the concept structure of a MT. Specifications written in VC^t can be parsed and automatically generate executable code, which implements an interactive design tool dedicated to the given MT.

I. INTRODUÇÃO

O aparecimento de Aplicações que utilizam uma representação visual de dados, com recurso a estrutura de grafos, começa a ser uma prática corrente. O esforço empregue para desenvolvimento deste tipo de aplicações, pode ser substancialmente reduzido, se for fornecida ao projectista uma ferramenta que incorpore potenciais funcionalidades de análise e manuseamento de grafos.

O problema consiste em desenvolver ferramentas que além das devidas capacidades de interacção visual, possuam ainda uma forma de incorporar a semântica da Técnica de Modelação suportada pela dita ferramenta. Ou seja, que os objectos gráficos que compõem o diagrama possuam um significado próprio, o que vai limitar a forma como estes podem ser combinados. O presente artigo descreve um Sistema que constitui uma solução para este problema.

As diferentes técnicas de Modelação são expressas em VC^t, uma linguagem de especificação formal, e posteriormente compiladas para Código JAVA o qual implementa as ferramentas, a serem fornecidas aos projectistas mencionados acima.

Na secção II, é apresentada a linguagem VCt, e tenta-se descrever como pode ser uma ferramenta bastante útil no desenvolvimento de aplicações como as mencionadas.

A secção III faz uma descrição da Arquitectura Geral do Sistema de Geração Automática de código. Na secção IV são descritos os pormenores mais directamente relacionados com o processo de produção de código (Java) a partir da linguagem formal de especificação VCt.

II. CARACTERIZAÇÃO DA LINGUAGEM VCT

A. Introdução

A linguagem VCt é capaz de reproduzir a semântica de uma Técnica de Modelação (TM). Tal é exclusivamente conseguido através do uso de restrições semânticas. Uma restrição semântica não é mais do que uma regra que exprime parte da semântica de uma Técnica de Modelação sob especificação, que possa ser validada. Em VCt., essas restrições são escritas num formato de predicado lógico com igualdade.

Uma especificação de TM inclui duas secções principais: A secção de ***Preamble***, onde todos os conjuntos de variáveis e suas propriedades são declarados, e a secção de ***Semantic Constraints***, onde cada restrição é especificada por uma regra em lógica. Estas duas secções são explicadas em detalhe mais à frente.

Numa especificação, cada restrição é associada a uma linguagem natural que será usada posteriormente na ferramenta de desenho (automaticamente gerada), de suporte à Técnica de Modelação, para fornecer uma realimentação semântica ao utilizador do sistema, acrescentando assim legibilidade à especificação.

A semântica de uma TM é independente da representação gráfica. Assim, os aspectos relacionados com as notações gráfica da TM são deixadas propositadamente fora do âmbito da linguagem VCt, o que contribui significativamente para a sua simplicidade.

- As especificações obtidas com VCt são, ao mesmo tempo, descrições formais das TM's e, descrições de alto nível das ferramentas de suporte às mesmas. Estas descrições apresentam maior vantagem para o utilizador quando comparado com linguagens naturais:
- Devido a serem formais, possuem um significado bem definido e, consequentemente são precisas.

- A linguagem VCt foi explicitamente desenvolvida para expressar TMs, assim sendo, as descrições possuem uma estrutura e terminologia mais consistentes que aquelas obtidas com linguagens de âmbito mais geral como a linguagem Natural.

A linguagem VCt tem como um dos seus objectivos principais permitir que o utilizador construa uma especificação de forma fácil e rápida. Tal será conseguido se a linguagem possuir uma rápida curva de aprendizagem. Deve-se ter em conta que os utilizadores desta linguagem são projectistas de TM's, o que não implica que possuam um profundo conhecimento Matemático ou de métodos formais. Assim sendo, tentou-se reduzir ao mínimo a complexidade dos aspectos formais da linguagem. A formalidade foi apenas empregue como uma ferramenta Matemática, de forma a conduzir o desenho da linguagem. O grande objectivo é o de obter uma linguagem que possa ser usada por alguém que já seja capaz de especificar uma Técnica de Modelação usando Linguagem Natural. O uso desta linguagem nunca deverá ser razão para aumentar a complexidade da especificação.

B. A estrutura de uma especificação na linguagem VCt

Como foi mencionado anteriormente, uma especificação em VCt tem duas secções principais: A secção de ***Preamble***, onde todos os conjuntos de variáveis e suas propriedades são declarados, e a secção de ***Semantic Constraints***, onde cada restrição é especificada por uma regra em lógica.

▪ A secção de Preamble

Esta secção expressa a TM em termos de todos os seus tipos de VC (*Visual Concepts*) - ou seja, os objectos usados pela TM, onde são declarados todos os conjuntos que serão usados durante a especificação. Esta Secção compreende ainda quatro subsecções as quais são sucintamente descritas de seguida:

1) MT MODEL - Produto cartesiano dos power sets de todos os VC types. Uma Técnica de Modelação Genérica poderá ser expressa da seguinte forma:
' MTX = P VCt0 x P VCt1 x ... x P VCtn '
' VAR mtx : MTX '
onde **MTX** é o identificador da Técnica de Modelação (nome), **VCt0** a **VCtn** são os VC types definidos para a TM em especificação. Uma variável do tipo **MTX** é ainda declarada para poder ser usada na secção de ***Semantic Constraints***.

2) SET EXTRACTORS DECLARATIONS - Nesta subsecção os power sets são divididos naqueles que serão representados por ícones e por ligações. Tal é necessário para que o compilador pode estabelecer o mapeamento de cada componente do diagrama (*ícones e ligações*) para o respectivo componente grafo abstracto (*nós e links*) .

3) SETS DEFINITIONS - Nesta subsecção é possível definir-se conjuntos auxiliares de VC types.

4) SET PROPERTIES - Declaração das propriedades de cada VC. As propriedades de cada VC consistem nos seus componentes restritivos. Estes componentes são usados nas frases de predicado lógico das restrições expressas na secção de ***Semantic Constraints***.

▪ A secção de Semantic Constraints

Uma ***Semantic Constraint*** (restrição semântica) tal como já foi dito, é uma regra expressa em lógica.

As restrições podem ser divididas em dois grupos: as ***instantiated predicate logic statements*** e as ***quantified predicate logic statements***. No primeiro grupo, as restrições não possuem variáveis no seu predicado o que significa que o predicado não é quantificado. No segundo grupo, os predicados das restrições usam variáveis logicamente quantificáveis. Essas quantificações podem ser *existenciais* ou *universais*. O encadeamento destas quantificações é possível, com qualquer número de níveis.

III. ARQUITECTURA GERAL DO SISTEMA:

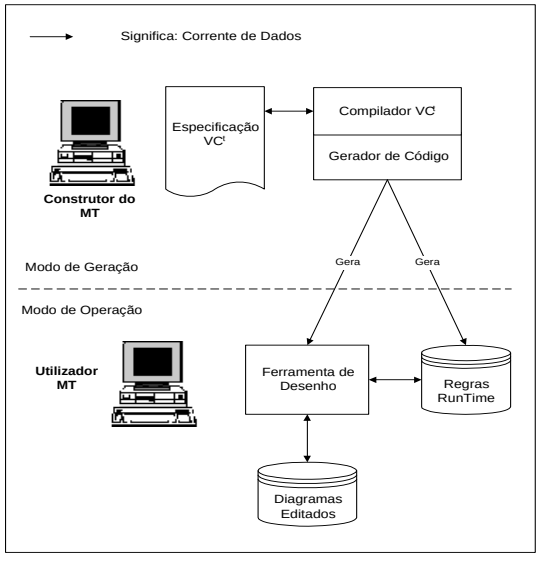


Fig. 1 – Arquitectura geral do Sistema

Nesta secção é descrita a Arquitectura Geral do Sistema VC. A única entrada existente no sistema é a especificação da Técnica de Modelação escrita na linguagem VCt. O objectivo deste Sistema é de automaticamente gerar o código implementador da

ferramenta, um editor gráfico dedicado à semântica, que suportará esse Técnica de Modelação a partir dessa especificação. O Sistema é usado tanto em ***modo de geração*** quando a especificação é compilada e traduzida para código executável, como em ***modo de operação*** quando a ferramenta de desenho que foi gerada é usada.

A. O sistema em Modo de Geração

O construtor do Sistema, especifica em linguagem VCt, o desenho da Técnica de Modelação, de seguida o sistema compila a especificação e, no caso de sucesso (ausência de erros de VCt), é criada uma representação da especificação da Técnica de Modelação em memória. Será essa representação intermédia que será posteriormente usada para gerar o código de implementação da ferramenta de suporte à Técnica de Modelação

A maior parte da complexidade do Processo de Geração de código está no gerador automático. O processo de tradução é baseado num conjunto de regras as quais cobrem todos os aspectos possíveis de uma especificação VCt e produzem código logicamente equivalente. Estas regras estão integradas no gerador de código.

B. O sistema em Modo de Operação

O Sistema também proporciona suporte em Modo de Operação. A ferramenta de desenho (ver fig. 1), gerada no Modo anterior, inclui funcionalidades para edição de diagramas suportadas por uma ferramenta genérica de manuseamento de Grafos. A ferramenta de Grafos gere os grafos e a respectiva representação visual. Também aplica ao diagrama a ser editado, as restrições geradas no Modo anterior (especificadas inicialmente em VCt) usando o seu próprio gestor de regras.

IV. GERAÇÃO AUTOMÁTICA DE CÓDIGO – COMPILADOR DE VCT

A. Princípios Gerais e alguns Problemas relacionados com a Produção de Código

A geração de restrições executáveis torna-se o principal problema no processo de Geração de Código. Cada restrição duma Especificação em VCt, deve ser traduzida para código executável. A saída do processo de tradução deve consistir num conjunto de restrições em formato executável, cada qual correspondendo a uma restrição de especificação VCt. Este processo de tradução é baseado num conjunto finito de regras, as quais cobrem todos os aspectos duma especificação em VCt produzindo código logicamente equivalente.

O principal objectivo consiste em conseguir definir-se regras de tradução correctas e ***coerentes*** que originem código legível, simples e eficiente.

Mais abaixo no capítulo *Regras de compilação*, serão apresentados os princípios gerais do processo de produção de código. Esta introdução pretende fornecer uma visão geral das regras de tradução, do processo de geração de código.

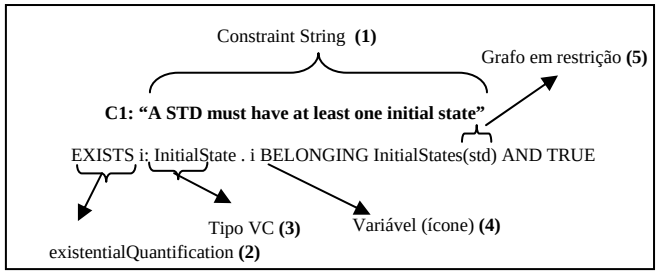
B. Exemplos de Compilação

Nesta secção, são apresentadas alguns exemplos de tradução de especificações VCt para código executável. Como já foi dito anteriormente, o principal objectivo é a produção de código que seja o mais simples e legível possível. Um código com uma boa apresentação implica regras de tradução mais complexas embora, nalgumas situações se tenha sacrificado ligeiramente a legibilidade para simplicidade de regras.

Para todas as regras que se referem a Objectos Visuais (VO's) assumiu-se que a variável nó na especificação é um ícone. As regras podem ser aplicadas da mesma maneira a especificações com variáveis de ligação (link's).

▪ Exemplo 1

Considere-se a seguinte restrição em VCt:



Na restrição anterior está-se a impor que o grafo STD (ponto 5) tenha sempre um estado inicial. Para se poder validar tal restrição, o código gerado deve efectuar uma chamada à função de navegação de grafos e verificar se existe algum nó do tipo VC ***InitialState*** (ponto 3). Caso tal não se verifique, a *string* contida na primeira linha deverá ser impressa de forma a informar que a restrição foi violada (ponto 1).

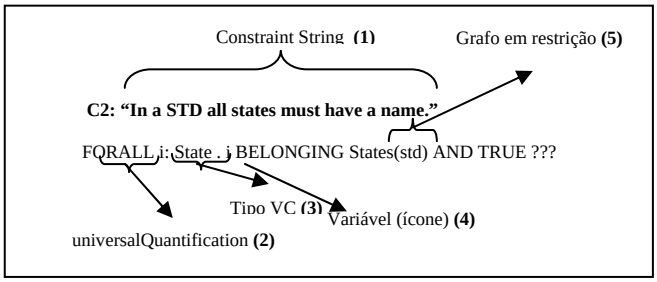
Para validar a restrição basta que se verifique a condição num nó (ícone - ponto 4) do grafo (**EXISTS**) tratando-se assim de uma ***existentialQuantification***. Tal implica que a navegação dos grafos deverá parar no primeiro nó que satisfaça o predicado e atribuir o valor de ***true*** à variável ***success*** (ponto 2).

O código JAVA gerado será então:

```
public static VCtSearch existencial_1( ){
{
UnaryPredicate predicate=new UnaryPredicate(){
        public boolean execute(Object o){
        Node vctNode=(Node)o;
        VctGraph data=(VctGraph)vctNode.getData();
        Object i=data.container;          /*(4)*/
if ( i  instance of InitialState )        /*(3)*/
{search.success=true;                     /*(2)*/
return true;}
else return false;
        };
        Traverse tr=new Traverse();
        tr.setGraph(std);                 /*(5)*/
tr.setStopAtFirst(true);  // Stop's at first true
tr.setPredicate(predicate);
tr.navigate(predicate);   // Navigate
if (search.success==false)
System.out.println("A std must have at least one
initial state"); /* (1) */
 return search;     }
```

▪ Exemplo 2

Considere-se agora a seguinte restrição:



Na restrição anterior está-se a impor que todos os nós state (ponto 3) do grafo STD (ponto 5) tenham um nome. Para se poder validar tal pedido, o código gerado deve efectuar uma chamada à função de navegação de grafos e verificar se todas as nós (ponto 4) tenham o campo *name* diferente de nulo. Caso tal não se verifique, a *string* contida na primeira linha deverá ser impressa de forma a informar que a restrição foi violada (ponto 1).

Para validar a restrição, tem que se verificar a condição em todos os nós do grafo (**UNIVERSAL**) tratando-se assim de uma ***universalQuantification***. Tal implica que a navegação dos grafos deverá verificar o predicado em todos os nós e atribuir o valor de ***true*** à variável ***violated*** no caso de violação (ponto 2).

O código JAVA gerado será então:

```
…
public static VCtSearch existencial_1( ){
{
UnaryPredicate predicate=new UnaryPredicate(){
        public boolean execute( Object o ) {
        Node vctNode=(Node)o;
        VctGraph i=(VctGraph)vctNode.getData(); /*(4)*/
        String nameg=i.name;                    /*(4)*/
        if ( nameg != "" ) return false;
else    {search.violated=true;                  /*(2)*/
  return true;}
        };
Traverse tr=new Traverse();
tr.setGraph(std);                               /*(5)*/
tr.setStopAtFirst( true );     // Stop's at first true
tr.setPredicate(predicate);
tr.navigate(predicate);       // Navigate
        if (search.violated==true)
System.out.println("In a STD all states must have a
name.");                                        /*(1)*/
 return search;}
```

C. Trabalho anterior

Foi já projectado e construído um compilador que inclui um gerador de código, para a linguagem formal VCt [3].

A sua implementação foi baseada pelas ferramentas Lex e Yacc do sistema UNIX

▪ Arquitectura

VCt - Linguagem fonte;
C - Linguagem de implementação;
Napier88 - Linguagem Alvo;

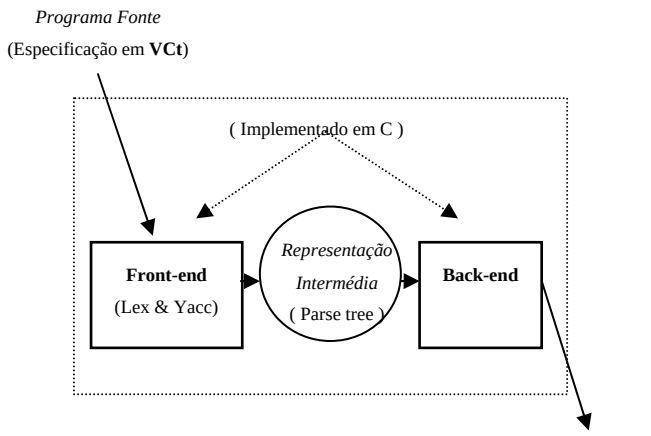


Fig. 2 – Arquitectura anterior

O modelo de compilação de VCt compreende dois blocos principais:

O bloco ***Front-end*** e o bloco ***Back-end***.

O bloco ***Front-end*** é composto pelo analisador de léxico e pelo *parser*. Efectua a fase de análise no processo de compilação ou seja, faz a tradução de VCt para uma representação intermédia (construção de uma parse tree) e efectua uma análise sintáctica e semântica.

O bloco ***Back-end*** consiste no gerador de código. Através da representação intermédia do programa fonte, gera o código que, nesta implementação consiste em NAPIER88.

A principal vantagem deste modelo consiste, além de permitir fáceis testes e manutenção, produzir um compilador redireccionável isto porque, o parser é independente da linguagem alvo, sendo assim perfeitamente portátil para outras plataformas.

Dessa forma, para transformar o compilador de forma a produzir uma linguagem alvo diferente da usada na implementação actual (NAPIER88), para JAVA ou C++ , por exemplo, basta substituir o bloco ***Back-end***.

O Código gerado implementa uma ferramenta de suporte à Técnica de Modelação especificado na linguagem VCt.

D. Trabalho Actual

A ideia actual foi de efectuar um ***upgrade*** no compilador descrito anteriormente para, não só:

- Passar a produzir JAVA como linguagem alvo, bem como,
- Implementá-lo nessa mesma linguagem.

Para tal foi necessário reconstruir tanto o bloco ***Front-end*** como o ***Back-end***.

Desta forma a arquitectura do compilador passou a ser a seguinte:

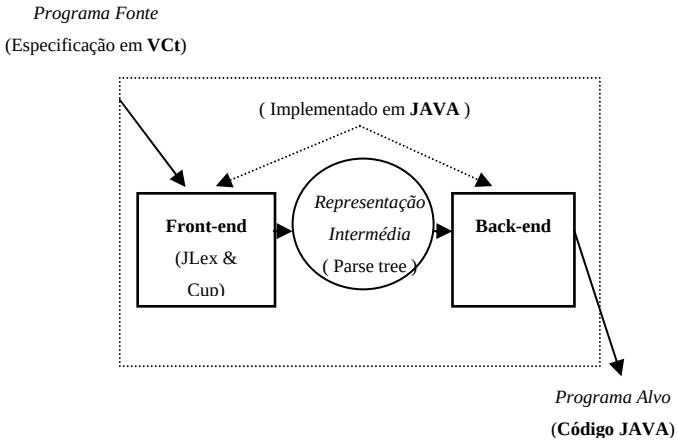


Fig. 3 – Arquitectura Actual

▪ Software Utilizado

JLex & Cup

À semelhança do uso das ferramentas Lex e Yacc (do sistema UNIX) adoptou-se o uso das ferramentas JLex e Cup para linguagem JAVA disponíveis, assim como toda a documentação correspondente, em :

http://www.cs.princeton.edu/~appel/modern/java/

O JLex foi implementado à imagem do Lex . O Lex é um gerador de analisadores de léxico que, a partir de um ficheiro pré-definido de especificação, o qual contém os detalhes específicos do analisador, cria o código (neste caso C) implementador do analisador pretendido. Assim, tal como o Lex, o JLex constrói o analisador de léxico a partir dum ficheiro de definições relativas ao analisador que se pretende, produzindo o código JAVA (em vez de C) correspondente.

Este analisador de léxico será capaz de dividir uma corrente de caracteres de entrada em Token's (conjuntos pré definidos de caracteres) os quais são enviados ao Parser (neste caso gerado pelo Cup) o qual fará o processamento adequado para análise sintáctica, e posteriormente semântica da entrada.

O Cup é uma ferramenta também muito semelhante ao Yacc do UNIX. Pode-se tentar definir este tipo de ferramentas como compiladores de compiladores (o nome Yacc provém de "**Y**et **a**nother **c**ompiler **c**ompiler") as quais permitem a construção de compiladores (ou tradutores de linguagem) de linguagens que se pretendam construir. Para tal, é também necessário definir um ficheiro de especificação no qual se define a gramática (sintáctica), em formato BNF, do compilador a gerar.

Biblioteca de Grafos

Para o manuseamento do ambiente de grafos foi utilizada uma biblioteca já construída. Refira-se que no trabalho anterior (implementação para Napier88), foi implementada uma ferramenta deste género, a GraphTool [4].

A biblioteca de funções aqui utilizada foi implementada em JAVA. Fornece além das funções básicas de manuseamento de grafos, um editor gráfico que permite a edição visual de um diagrama previamente definido. Esta ferramenta, por ainda estar numa fase de desenvolvimento, ainda não está provida de documentação.

E. Estado Actual e Desenvolvimento

Neste momento foi já completamente construído o *parser* utilizando o JLex e o Cup. O gerador de código está ainda em construção tendo no entanto sido já gerado código JAVA para especificações muito simples, tais como as exemplificadas neste artigo.

F. Aplicações

Como possível exemplo de aplicação deste Sistema VC, poderá ser um editor para a TM *(Activity Diagrams)*. Esta TM é parte integrante da UML *(Unified Modelling Language)* [8].

Um *Activity Diagram* inclui os seguintes ícones e a seguinte ligação:

Start icon, End icon, Synchronization icon, Decision Activity icon e Trigger connection.

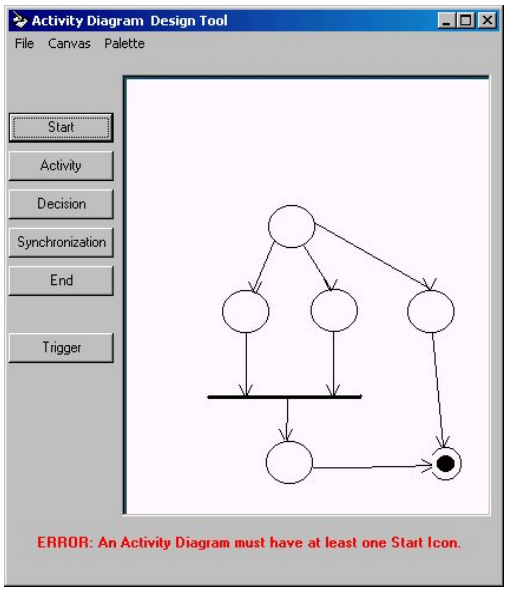


Fig. 4 – Exemplo de Aplicação

O editor a ser gerado pelo Sistema, é mostrado na figura 4. O utilizador do sistema poderá seleccionar qualquer um dos ícones ou ligação na palete do lado esquerdo e colocá-lo na área de desenho.

O exemplo da figura está ainda incompleto visto faltar-lhe o ícone Start. Quando é executada a validação do diagrama, a mensagem de erro visível na figura (a vermelho), é produzida de formas a alertar o utilizador do erro que está a cometer. Esta mensagem tem origem nas *Constraint String's* da especificação VCt referente ao grafo em questão (pontos 1 dos exemplos de compilação apresentados anteriormente).

V. REFERÊNCIAS

[1] Eckel, Bruce, "Thinking in Java", *Prentice Hall Computer Books*, ISBN: 0136597238, 1998.

[2] Serrano, J. Artur, "The Use of Semantic Constraints on Diagram Editors", in proceedings of the *11th International IEEE Symposium on Visual Languages, VL'95*, Darmstadt, Germany, 1995.

[3] Serrano, J. Artur, "Automatic Generation of Software Design Tools Supporting Semantics of Modelling Techniques", *PhD Thesis*, University of Glasgow, 1997.

[4] Serrano, J. Artur, "The GraphTool: Supporting Visual and Interactive Graph-Based Applications", *Technical Report Series*, University of Glasgow, August, 1997.

[5] Serrano, J. Artur and Welland, Ray, "A Tutorial for the VCt Formal Specification Language", *Technical Report Series*, University of Glasgow, August, 1997.

[6] Serrano, J. Artur and Welland, Ray, "VCt - A Formal Language for the Specification of Diagrammatic Modelling Techniques", published in journal *Information and Software Technology*, 40, pp. 463-474, Elsevier, 1998.

[7] Schreiner, Axel T. and Friedman, H. George, Jr., "Introduction to Compiler Construction with UNIX", *Prentice-Hall, Inc*., Englewood Cliffs, NJ 07632, 1985.

[8] Fowler, Martin, "UML Distilled, Applying the Standard Object Mobility Language", *Addison-Wesley*, September 1998.