# Ambiente para Representação de Múltiplas Visões de Requisitos: O Metamodelo e Uma Linguagem de Transformação

**Tereza Gonçalves Kirner**
**Antônio Francisco do Prado**
**Maria Adriana Vidigal de Lima**
**Rogéria Cristiane Gratão**
<tkirner, prado>@power.ufscar.br
<adriana, rogeria>@dc.ufscar.br

**Universidade Federal de São Carlos**
**Programa de Pós-Graduação em Ciência da Computação**
Rodovia Washington Luiz, Km 235 Caixa Postal 676
13565-905 São Carlos-SP Telefone: (016)274-8233

**Abstract**

This article describes an environment to represent multiple views of software requirements. Two essential components of the environment are discussed, that is, the metamodel and a transformation language. The metamodel is a unique, common representation that supports the integrated use of representation techniques for different methods. The language permits to write the metamodel, aiming to automaticaly generate the system codding in an executable language.

KEY WORDS: Engenharia de Requisitos, Métodos, Metamodelo, Linguagem Transformacional.

## 1. Introdução

A especificação de requisitos envolve a modelagem e descrição do comportamento externo de um sistema. Sua importância é amplamente reconhecida, pois erros de especificação que não forem detectados e adequadamente sanados, ocasionarão problemas que tornar-se-ão cada vez mais difíceis e dispendiosos de serem corrigidos à medida em que o desenvolvimento do sistema for avançando para as etapas subsequentes (Davis [3]).

No intuito de vencer as dificuldades de especificação, tem surgido uma série de métodos, como análise estruturada, análise orientada a objetos, Statemate, rede de Petri, etc. Cada método possui certas peculiaridades que o torna apropriado para a especificação de determinados aspectos, permitindo a geração de uma visão única dos requisitos do sistema. Esta *visão única* nem sempre é suficiente para especificar correta e completamente uma aplicação. Em contrapartida, a abordagem de múltiplas visões[1] possibilita o uso integrado de diferentes métodos na especificação de um sistema, ao mesmo tempo que propicia: garantia da consistência entre especificações preparadas utilizando-se mais de um método; migração de um método

[1] De acordo com o conceito de *múltiplas visões* assumido neste projeto, uma *visão* corresponde a uma representação dos requisitos do sistema, obtida de acordo com um determinado método ou técnica de especificação (Davis *et al.* [4]).

para outro, em situações nas quais o sistema já começou a ser desenvolvido ou passará por re-engenharia; reuso de (porções de) especificações; minimização dos custos de manutenção; facilidade de documentação; criação de especificações de alta qualidade; etc.

Este trabalho apresenta os resultados até então obtidos em um projeto de pesquisa, cujo objetivo é desenvolver um ambiente para representação de múltiplas visões de requisitos de sistemas de software. O componente central do ambiente é um metamodelo para representação de requisitos, que dá suporte ao uso integrado de técnicas provenientes de diferentes métodos. Aliada a este metamodelo, é essencial que exista uma linguagem, que viabilize a transformação das representações criadas e a consecutiva geração automática de código em linguagens executáveis. O metamodelo definido, assim como a linguagem criada são objetos do presente artigo.

O artigo está organizado de acordo com os ítens definidos a seguir. O item 2 descreve o *Sistema de Controle de Tráfego*, adotado como base nos exemplos ao longo dos ítens. O item 3 apresenta o metamodelo para especificação de requisitos, incluindo também os mapeamentos entre especificações feitas por meio de três técnicas e o metamodelo. O item 4 descreve a linguagem para escrita do metamodelo. As conclusões do trabalho são apresentadas no item 5.

## 2. Exemplo de Aplicação

O exemplo empregado é um Sistema de Controle de Tráfego, adaptado de Hull *et al.* [7] e ilustrado na Figura 1.

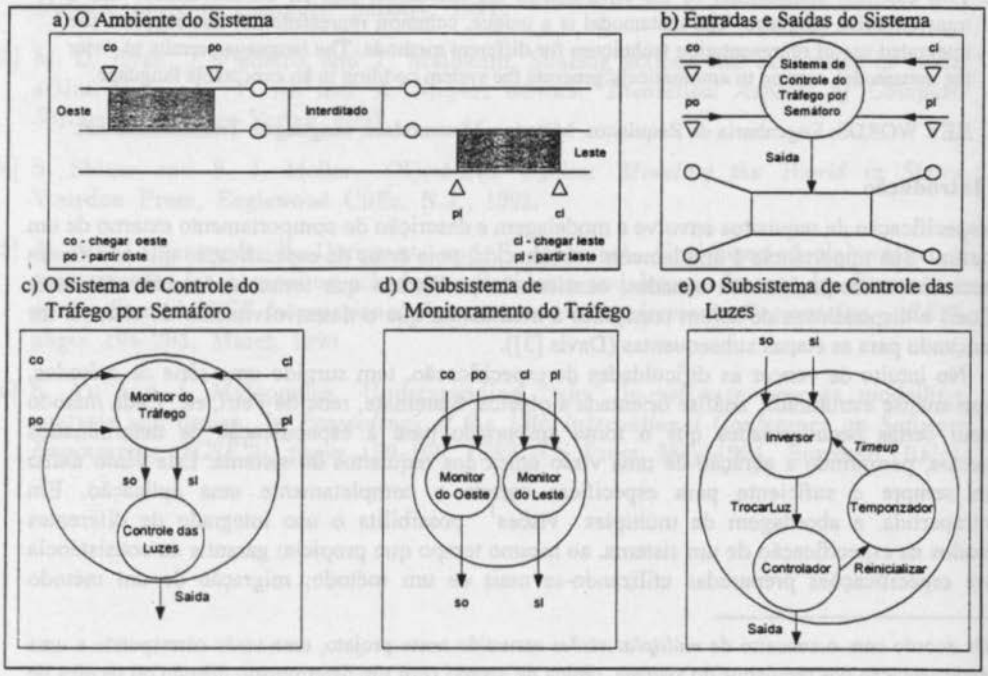



Figura 1. Visão Geral do Sistema de Monitoramento do Tráfego

A Figura 1-a ilustra o ambiente do sistema, com uma rua na qual o trânsito está funcionando no sentido Leste-Oeste e o tráfego é permitido apenas em uma direção de cada vez. As áreas hachuradas são *zonas de aproximação* para os semáforos. Os círculos mostram as posições dos semáforos e os triângulos representam Predicado sensores que monitoram o trânsito que chega e que atravessa as zonas de aproximação. As luzes do semáforo se alternam, permitindo que o trânsito flua nas duas direções. O trânsito flui em cada direção por um período de tempo de no máximo 45 segundos e até que haja carro(s) esperando na zona de aproximação oposta. Ou seja, se existir carro na zona oposta e a zona corrente estiver vazia, as luzes do semáforo mudarão, mesmo que os 45 segundos não tenham se esgotado; caso os 45 segundos tenham se passado, mas ainda existir(em) carro(s) na zona corrente e a zona oposta estiver vazia, a mudança de luzes não ocorrerá.

A Figura 1-b indica a existência de quatro entradas, vindas dos sensores para o sistema de software, e uma saída, que vai do sistema para um dispositivo único que altera todas as quatro luzes do semáforo. A Figura 1-c mostra as funções básicas do sistema (ou subsistemas), que responsáveis por monitorar o tráfego e controlar as luzes do semáforo.

Os subsistemas de Monitoramento de Tráfego e Controle das Luzes são ilustrados pelas Figuras 1-d e 1-e, respectivamente.

Detalhes sobre o funcionamento do sistema são fornecidos a seguir:

- O número inicial de carros em uma zona de aproximação é igual a zero. Cada vez que uma mensagem de chegada (*co* ou *cl*) é recebida, o número de carros é incrementado, para refletir o fato de que um carro entra na zona de aproximação. Cada vez que uma mensagem de saída (*po* ou *pl*) é recebida, o número de carros é decrementado, para indicar que um carro sai da zona de aproximação. Sempre que ocorrer uma mudança no *estado geral* da zona de aproximação (o *estado geral* indica se existe ou não algum carro na zona de aproximação considerada), o novo estado é emitido sob a forma de uma mensagem. Assim, toda vez que o número de carros mudar de zero para um, uma mensagem *"carro"* será emitida, e toda vez que o número de carros mudar de um para zero, uma mensagem *"semcarro"* será emitida.
- Existe um processo responsável pela temporização, que recebe mensagens sobre um canal de *reinicializar* e emite mensagens sobre um canal de *timeup*. Sempre que uma mensagem for recebida (as luzes são alteradas), uma contagem regressiva de 45 segundos será disparada. Quando o tempo de 45 segundos, contado a partir da última entrada, se expirar, uma mensagem de *timeup* para a direção corrente é enviada. Se uma mensagem de *reset* for recebida antes do decrementador atingir zero, este será reinicializado com o valor 45.
- Existe um processo responsável por alteração de luzes, que replica uma mensagem, passando-a para o processo *Temporizador* a fim de avisar que as luzes do semáforo precisam ser mudadas. Sempre que uma mensagem de mudança for recebida (indicando que uma decisão para alteração de luzes foi tomada), uma mensagem de *reinicializar* será enviada para o *Temporizador* e outra mensagem será enviada para as luzes.
- Existe um processo responsável pela decisão de alterar as luzes, de acordo com o estado do tráfego (*"carro"* ou *"semcarro"*) e com o *timeup*.

## 3. Metamodelo para Representação de Requisitos

Vários metamodelos tem surgido para apoiar a criação de múltiplas visões de requisitos de software, como, por exemplo, as abordagens propostas por Davis [4], Delugach [5], Meyers & Reiss [11] e Zamperoni [16]. A adoção da abordagem proposta por Davis, como ponto de partida para o presente projeto de pesquisa, deve-se ao fato deste ser o metamodelo mais completo e de oferecer suporte a um grande número de técnicas distintas de especificação de requisitos.wazzu

O metamodelo aqui adotado tem por base uma representação canônica, que permite armazenar e analisar requisitos, independentemente do método de especificação empregado. O termo representação canônica referesão -se a uma representação única, comum, que elimina redundância entre métodos, ao mesmo tempo que preserva as características peculiares de cada método (Meyers [10]). É importante salientar que o metamodelo não é uma nova técnica de especificação, mas, sim, uma forma de representação interna para a qual, senão todos, mas os principais métodos disponíveis possam ser mapeados.

### 3.1 Descrição do Metamodelo

O metamodelo sugerido baseia-se em uma *Representação Canônica* de requisitos *RC (E, R)*, que é uma estrutura composta de um conjunto de elementos $E = \{e_1, e_2, ...\}$ e de um conjunto de relações $R = \{r_1, r_2...\}$, onde cada relação $r$ conecta um par ordenado de elementos $e, f \in E$, que não são necessariamente distintos. Além disso, cada $e_i \in E$ é uma *tupla:* $e_i = (et_i, el_i)$ onde $et_i$ é o tipo do elemento, $et_i \in ET \cup FT$, onde *ET = {entidade, processo, estado, mensagem, atributo, predicado, restrição, informação, transição}*, com $FT = 2^{ET}$. Por sua vez, $el_i$ é uma identificação única para o elemento. , Também, cada $r_i \in R$ é uma quádrupla: $(rt_i, rl_i, se_i, te_i)$ onde $rt_i$ é o tipo do relacionamento, $rt_i \in RT$, com RT = {*parte* de, *instanciação*, *tem valor*, *envia*, *recebe*, *estímulo*, *resposta*, *equivalência*, *associação, operando*}, $rl_i$ é uma identificação única para a relação, $se_i$ e $te_i$ são o elemento inicial e o elemento final do relacionamento, $se_i \in E$ e $te_i \in E$.

Em resumo, o metamodelo compõe-se dos seguintes elementos e relacionamentos:

- Elementos: *entidade, processo, mensagem, estado, atributo, informação, predicado, restrição e transição.*
- Relacionamentos: *instanciação, parte-de, tem-valor, envia, recebe, estímulo, resposta, operando, equivalência e associação.*

A representação de um sistema, de acordo com o metamodelo, é feita por meio de um grafo multipartido, no qual os nós correspondem aos elementos e os arcos correspondem às relações. Além disso, cada tipo de elemento e de relacionamento possui a sua representação gráfica. As Tabelas 1 e 2 contém, as representações gráficas e definições sucintas dos relacionamentos e elementos listados.

Tabela 1. Relacionamentos do Metamodelo

| Nome | Representação Gráfica | Definição |
|---|---|---|
| Parte de | *parte de* ⟶ | Indica que um elemento *e2* é parte de um outro elemento *e1* no sistema considerado. |
| Instanciação | *inst.* ⟶ | Indica que um elemento *e1* é uma generalização de um outro elemento *e2*. |
| Tem valor | *tem valor* ⟶ | Indica que um elemento, *e1*, tem valor de um outro, *e2*, se *e2* recebe um valor específico atribuído a *e1*. |
| Envia | *envia* ⟶ | Um relacionamento *envia* capta os requisitos relativos a um elemento e gera uma mensagem. |
| Recebe | *recebe* ⟶ | Um relacionamento *recebe* capta os requisitos de um elemento, afim de aceitar uma mensagem. |
| Estímulo | *estim.* ⟶ | Um relacionamento *estímulo* capta os elementos que causam a ocorrência de uma transição. |
| Resposta | *resp.* ⟶ | Um relacionamento *resposta* capta os elementos que serão modificados pela ocorrência de uma transição. |
| Equivalência | *equiv.* ⟷ | Dois elementos *e1* e *e2* são equivalentes se eles representam conceito ou algo idêntico no mundo real. |
| Operando | *oper.* ⟶ | Representa um relacionamento entre um predicado ou restrição e seus respectivos operandos. |
| Associação | *assoc.* ⟶ | Representa um relacionamento com características distintas dos relacionamentos anteriores. |

Tabela 2. Elementos do Metamodelo

| Nome | Representação Gráfica | Definição |
|---|---|---|
| Entidade | *nome_da_entidade* | Representa algo existente no mundo real, que seja relevante para o problema em questão. |
| Processo | *nome_do_processo* | Representa uma ação, tarefa, função ou atividade a ser realizada. |
| Estado | *nome_estado* | Representa um modo no qual o problema/sistema se comporta de maneira a conservar determinadas características. |
| Mensagem | < *mensagem* > | Representa algo que está sendo movimentado de um elemento para outro. |
| Atributo | { *nome_do_atributo* } | Representa uma característica ou descrição de algum outro elemento do modelo. |
| | *prep.ou oper. Booleano* | Representa uma preposição ou um operador booleano, relacionados à aplicação tratada. |
| Restrição | *prep.ou oper.* | Define um relacionamento que deve existir obrigatoriamente entre um ou mais elementos. |
| Informação | [ *informação* ] | Representa um valor ou um conjunto de valores referentes à aplicação considerada. |
| Transição | *nome_da_trans* | Representa a conexão entre os agentes externos causadores de mudança no sistema (feita por meio de um estímulo) e os resultados gerados (por meio de uma resposta). |

### 3.2 Mapeamento de Especificações para o Metamodelo

Os métodos a serem suportados pelo metamodelo foram selecionados com base em critérios que levaram em conta métodos: (a) de uso comum ou bem aceitos entre a comunidade de engenharia de software; (b) bem difundidos na literatura; (c) que incluem técnicas de suporte gráfico; (d) que possuem diferenças substanciais nos conceitos envolvidos. Neste artigo, são consideradas técnicas pertencentes aos métodos de Análise Orientada a Objetos (Coad *et al.* [2]), Análise Estruturada (Shumate & Keller [15]) e Rede de Petri (Peterson [13]).

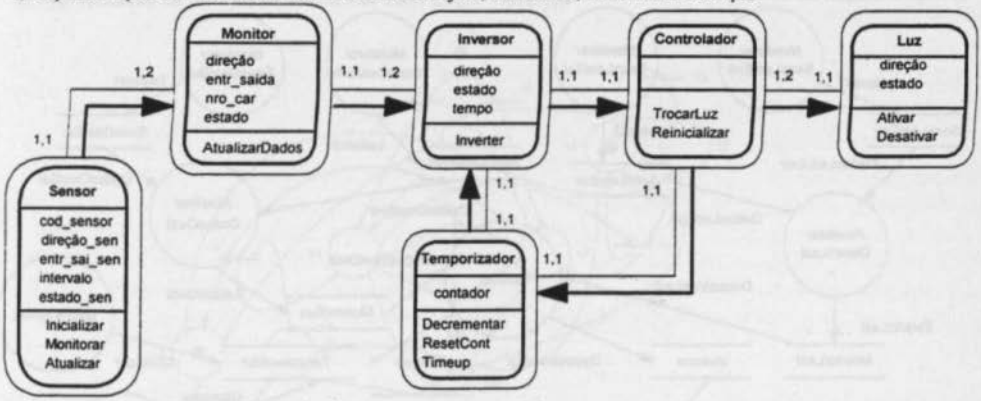
A Figuras 2, 3 e 4 apresentam as modelagens do Sistema de Controle de Tráfego, feitas em cada um dos métodos tratados, cada qual com a respectiva representação no metamodelo. Informações adicionais referentes à rede de Petri mostrada na Figura 4 são dadas na Tabela 4.

A Tabela 3 relaciona as primitivas de cada método e as suas primitivas correspondentes no metamodelo.

Tabela 3. Mapeamento entre Técnicas e o Metamodelo

| Técnicas | Metamodelo | |
|---|---|---|
| **Classe&Objeto** | **Elementos** | **Relacionamentos** |
| Classes | Entidade | ——— |
| Objetos | ——— | Parte de, Tem valor |
| Atributos | Atributo | Parte de |
| Serviços | Processo | Parte de |
| Estrutura Gen-Espec. | ——— | Instanciação |
| Estrutura Todo-Parte | ——— | Parte de |
| Conexão de Mensagem | Mensagem, Transição | Estímulo, Resposta, Envia |
| Conexão de Ocorrência (Cardinalidade) | Restrição | Parte de, Operando |
| **Diagrama de Fluxo de Dados** | **Elementos** | **Relacionamentos** |
| Processo | Processo | ——— |
| Depósito de Dados | Entidade, Atributo (interno / externo) | Parte de, Tem valor |
| Fluxo de Dados | Mensagem, *Livre* (ou seja, Informação, Entidade, Estado ou Atributo) | Associação, Envia, Recebe |
| **Redes Lugar-Transição** | **Elementos** | **Relacionamentos** |
| Lugar | Estado | ——— |
| Senha | Atributo (início/não início) | Tem valor |
| Transição | Transição, Predicados, Mensagem, *Livre* (ou seja, Informação ou Estado) | Estímulo, Operandos, Associação, Resposta |

a) Representação do Sistema na Técnica Classe&Objeto, Estruturas, Atributos e Serviços




b) Mapeamento da Representação do Sistema para o Metamodelo

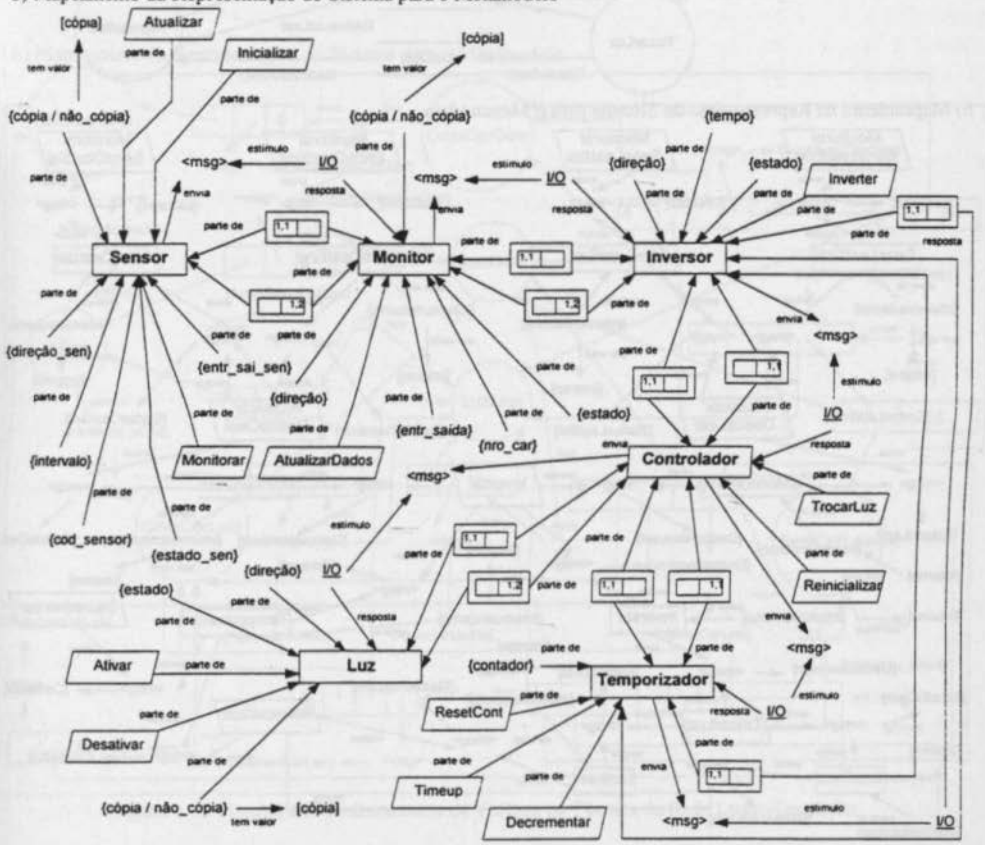
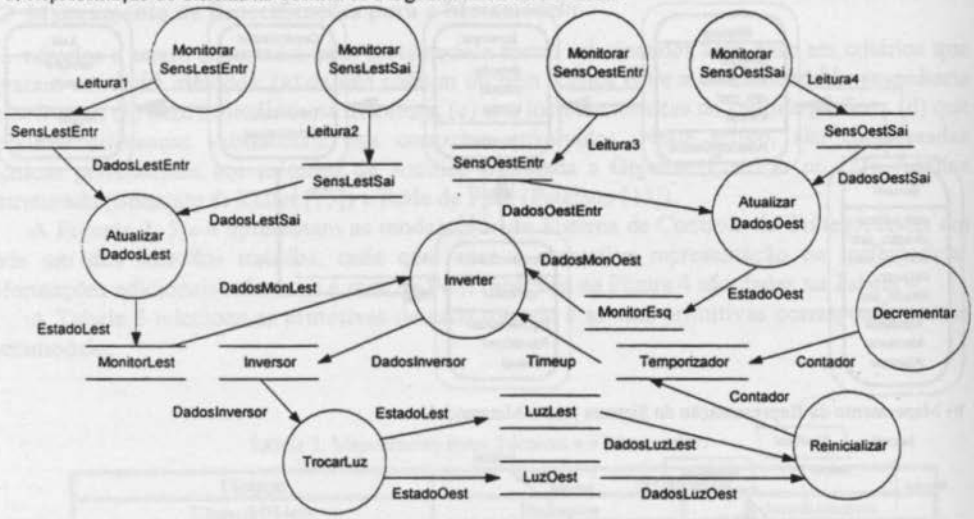



Figura 2. O Sistema de Monitoramento de Tráfego na Técnica Classe&Objeto

a) Representação do Sistema na Técnica de Diagrama de Fluxo de Dados




b) Mapeamento da Representação do Sistema para o Metamodelo

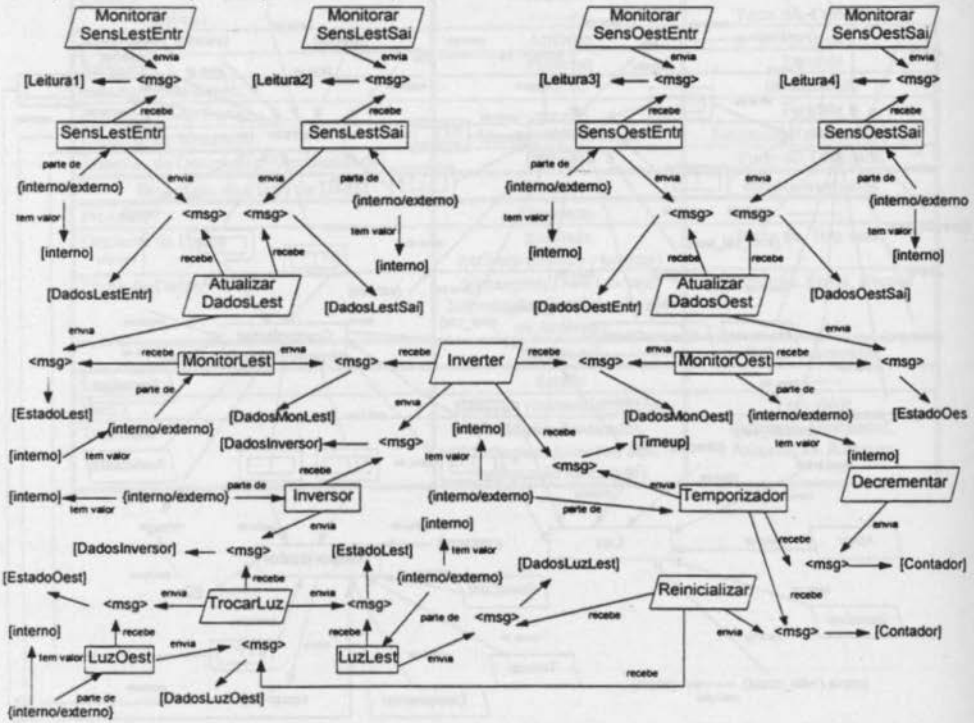
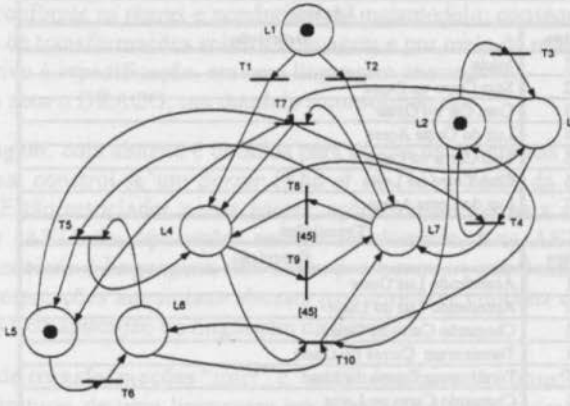



Figura 3. O Sistema de Monitoramento de Tráfego na Técnica de Diagrama de Fluxo de Dados

a) Representação do Sistema na Técnica de Rede Lugar-Transição




b) Mapeamento da Representação do Sistema para o Metamodelo

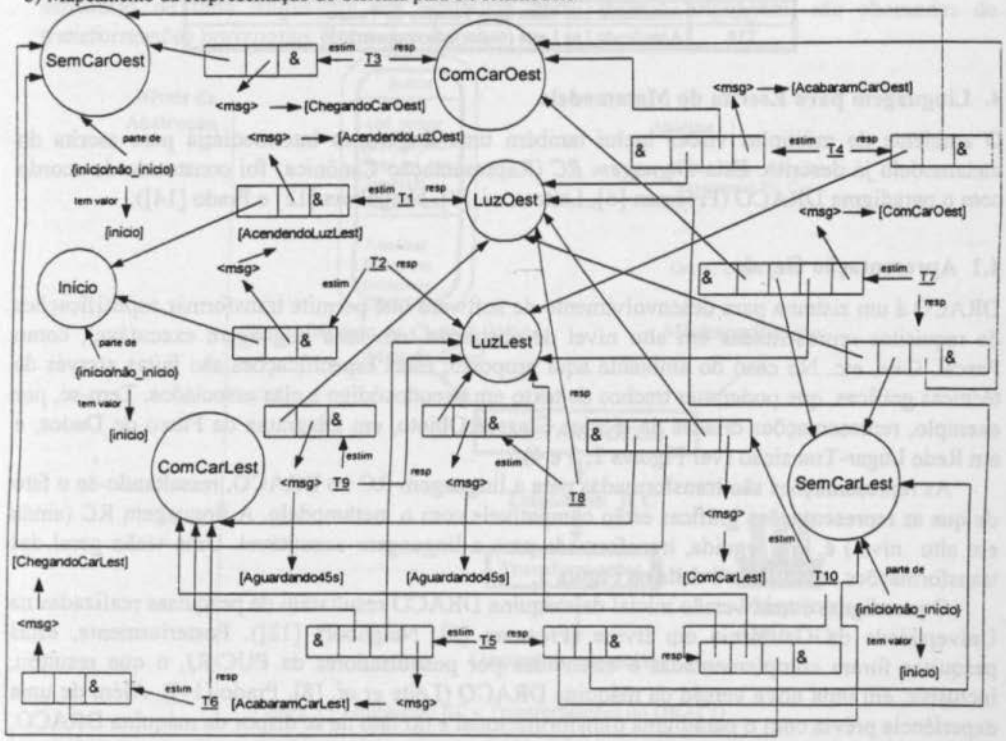



Figura 4. O Sistema de Monitoramento de Tráfego na Técnica de Rede Lugar-Transição

Tabela 4. Lista de Lugares e Transições

| Lugares | |
|---|---|
| **Código** | **Descrição** |
| L1 | Início |
| L2 | Sem Carro no Oeste |
| L3 | Com Car no Oeste |
| L4 | Luz do Oeste Acesa |
| L5 | Sem Carro no Leste |
| L6 | Com Carro no Leste |
| L7 | Luz do Leste Acesa |
| **Transições** | |
| **Código** | **Descrição** |
| T1 | Acendendo Luz Oeste |
| T2 | Acendendo Luz do Leste |
| T3 | Chegando Carro no Oeste |
| T4 | Terminaram Carros no Oeste |
| T5 | Terminaram Carros no Leste |
| T6 | Chegando Carro no Leste |
| T7 | Acendendo Luz Oeste (único lado com carro) |
| T8 | Acendendo Luz Oeste após Término do Tempo |
| T9 | Acendendo Luz Leste após Término do Tempo |
| T10 | Acendendo Luz Leste (único lado com carro)do |

## 4. Linguagem para Escrita do Metamodelo

O ambiente de múltiplas visões inclui também uma linguagem intermediária para escrita do metamodelo já descrito. Esta *linguagem RC* (Representação Canônica) foi construída de acordo com o paradigma DRACO (Freeman [6], Leite *et al.* [8], Neighbors [12] e Prado [14]).

### 4.1 Apresentação Geral

DRACO é um sistema para desenvolvimento de software que permite transformar especificações de requisitos representadas em alto nível de abstração, em uma linguagem executável, como Pascal, C++, etc. No caso do ambiente aqui proposto, estas especificações são feitas através de técnicas gráficas, que podem ter trechos de texto em pseudocódigo a elas associados. Tem-se, por exemplo, representações criadas na técnica Classe&Objeto, em Diagrama de Fluxo de Dados, e em Rede Lugar-Transição (ver Figuras 2, 3 e 4).

As representações são transformadas para a linguagem RC no DRACO, ressaltando-se o fato de que as representações gráficas estão compatíveis com o metamodelo. A linguagem RC (ainda em alto nível) é, em seguida, transformada para a linguagem executável. Uma visão geral das transformações envolvidas é dada na Figura 5.

O paradigma e uma versão inicial da máquina DRACO resultaram de pesquisas realizadas na Universidade da Califórnia em Irvine (Freeman [6], Neighbors [12]). Posteriormente, estas pesquisas foram complementadas e estendidas por pesquisadores da PUC-RJ, o que resultou, inclusive, em uma nova versão da máquina DRACO (Leite *et al.* [8], Prado [14]). Além de uma experiência prévia com o paradigma transformacional e do fato de se dispor da máquina DRACO, sua adoção no desenvolvimento do ambiente ora apresentado, justifica-se pela possibilidade de implementação automática do software e pela sua compatibilidade com a abordagem de geração de múltiplas visões de um sistema.

A Figura 6 ilustra a geração de diferentes visões de requisitos do Sistema de Controle de Tráfego. Isto inclui: especificação do sistema em uma determinada técnica; mapeamento da especificação conforme as regras e semânticas do metamodelo; consequente escrita na linguagem RC; realização de transformações sobre a linguagem e por meio da máquina DRACO; e produção do código relativo à especificação, em uma linguagem executável.

De acordo com o DRACO, um domínio é constituído de:

- Uma **linguagem**, cuja sintaxe é definida para escrita de programas no domínio. Para analisar os programas, constrói-se um *parser* (Aho *et al.* [1]) a partir da definição da gramática da linguagem. Estão associadas a este *parser*, ações semânticas para a geração de uma *Abstract Syntax Tree* (AST) das aplicações escritas no domínio. Esta AST é chamada Draco AST (DAST). Associado à linguagem, tem-se um *prettyprinter*, que é um *unparser* responsável por mapear representações em sintaxe abstrata para a sintaxe concreta da linguagem, ou seja, que exibe a DAST da aplicação na linguagem do domínio.

- Bibliotecas de **transformações** "inter" e "intra" domínios. As transformações da Figura 5, que mapeiam estruturas de uma linguagem em estruturas de uma outra linguagem, são chamadas de transformações verticais (inter-domínios). As transformações da Figura 6, que mapeiam estruturas de uma linguagem em estruturas dessa mesma linguagem, são chamadas de transformações horizontais (intra-domínios).

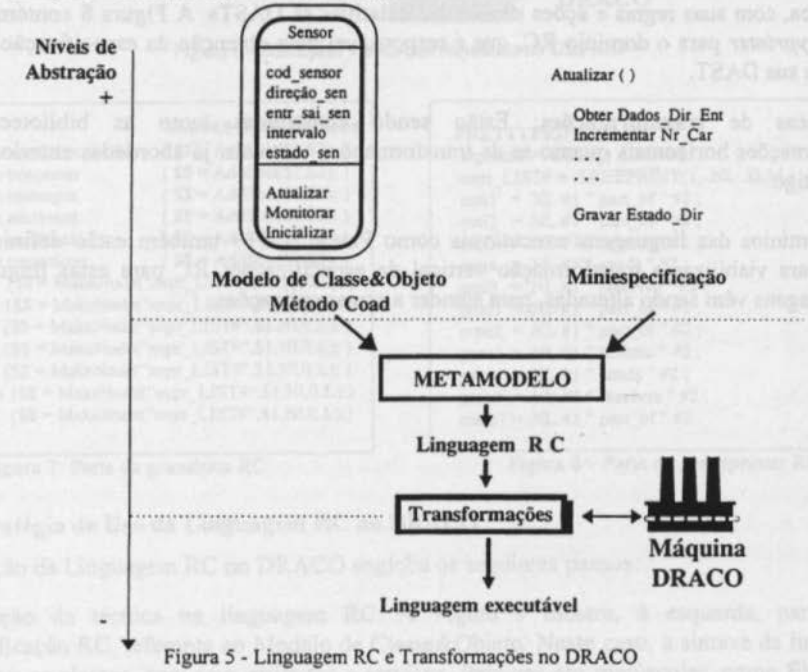



Figura 5 - Linguagem RC e Transformações no DRACO

Aplicações escritas na linguagem RC representam especificações de requisitos, sob uma determinada técnica, que podem ser transformadas para especificações vistas de acordo com outras técnicas, na mesma linguagem RC. A produção automática de um programa executável,

através da máquina DRACO, começa pela verificação de sua sintaxe pelo *parser* RC (caso a aplicação esteja escrita em mais de uma linguagem, outros *parsers* podem ser chamados), o que resultará na DAST. Sobre esta DAST, são aplicadas as bibliotecas de transformações, que transformam a aplicação no próprio domínio, ou para outros domínios suportados pelo DRACO. No caso do exemplo, as bibliotecas de transformações transformam a aplicação RC, na própria linguagem RC para obter múltiplas visões, ou em outras linguagens executáveis para obter diferentes implementações.

### 4. 2 Aspectos da Construção da Linguagem RC

A transformação de especificações, feitas segundo o metamodelo, dentro do próprio domínio RC (transformação horizontal) ou para outros domínios (transformação vertical), requer a descrição do domínio RC no ambiente DRACO. Para isso, foram definidas as seguintes partes:

- **Linguagem RC**: Esta linguagem inclui uma gramática, para a qual é gerado um *parser*. A definição da gramática seguiu as regras do metamodelo, seus elementos e relacionamentos (ver item 3.1). Após definida a gramática, foram gerados os respectivos *parser* e *unparser* na máquina DRACO. O *parser* analisa programas RC e constrói suas DASTs e o *unparser* exibe programas na forma interna DAST, usando a sintaxe RC. A Figura 7 mostra uma parte da gramática, com suas regras e ações semânticas relativas às DASTs. A Figura 8 contém parte do *prettyprinter* para o domínio RC, que é responsável pela obtenção da especificação RC a partir da sua DAST.
- Bibliotecas de **transformações**: Estão sendo construídas tanto as bibliotecas de transformações horizontais quanto as de transformações verticais, já abordadas anteriormente neste artigo.

Os domínios das linguagens executáveis como Pascal e C++ também estão definidos no DRACO para viabilizar a transformação vertical de especificações RC para estas linguagens. Estas linguagens vêm sendo alteradas, para atender a novas aplicações.

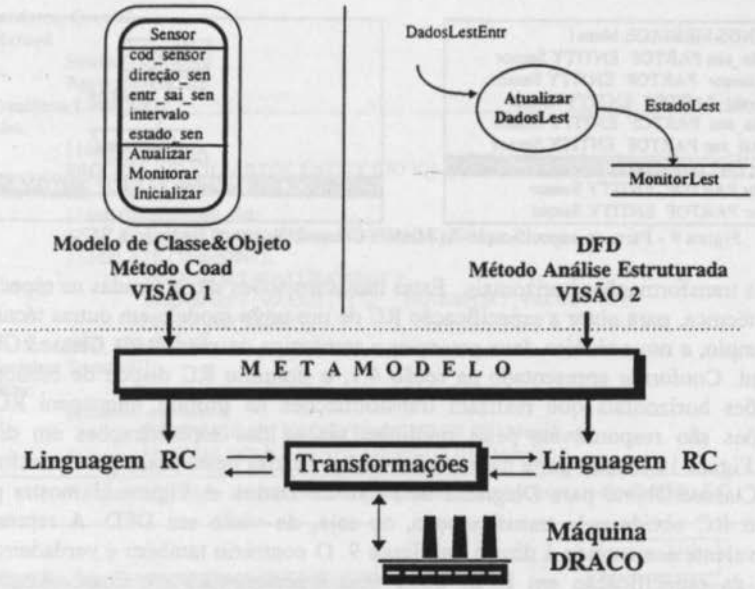



Figura 6 - Múltiplas visões dos requisitos no DRACO

```
Regras                                   Ações Semânticas
expression: expression entities     { $$ = AddSon($1,$2); }
| expression processes              { $$ = AddSon($1,$2); }
| expression messages               { $$ = AddSon($1,$2); }
| expression attributes             { $$ = AddSon($1,$2); }
| expression predicates             { $$ = AddSon($1,$2); }
| expression transitions            { $$ = AddSon($1,$2); }
| entities     {$$ = MakeNode("expr_LIST#",$1,NULL); }
| processes    {$$ = MakeNode("expr_LIST#",$1,NULL); }
| messages     {$$ = MakeNode("expr_LIST#",$1,NULL); }
| attributes   {$$ = MakeNode("expr_LIST#",$1,NULL); }
| states       {$$ = MakeNode("expr_LIST#",$1,NULL); }
| constraints  {$$ = MakeNode("expr_LIST#",$1,NULL);}
| predicates   {$$ = MakeNode("expr_LIST#",$1,NULL);}
```

Figura 7- Parte da gramática RC

```
PRETTYPRINTER RC
express1 = .COL(1) .LM #1 ;
expr_LIST# = .TREEPRINT(1, .NL .SLM,) ;
enti1  = .NL #1 " part_of " #2 ;
enti2  = .NL #1 " part_of " #2 ;
enti3  = .NL #1 " instanc " #2 ;
enti4  = .NL #1 " sends " #2 ;
enti5  = .NL #1 " receives " #2 ;
proc1  = .NL #1 " part_of " #2 ;
proc2  = .NL #1 " part_of " #2 ;
proc3  = .NL #1 " instanc " #2 ;
proc4  = .NL #1 " sends " #2 ;
proc5  = .NL #1 " receives " #2 ;
mess1  = .NL #1 " part_of " #2
```

Figura 8 - Parte do *prettyprinter* RC

## 4. 3 Estratégia de Uso da Linguagem RC no DRACO

A utilização da Linguagem RC no DRACO engloba os seguintes passos:

- Descrição da técnica na linguagem RC. A Figura 9 mostra, à esquerda, parte desta especificação RC, referente ao Modelo de Classe&Objeto. Neste caso, a sintaxe da linguagem descreve as classes, com seus atributos e serviços. Palavras em maiúsculas, como ENTITY e ATTRIBUTE, são reservadas, ou seja, são símbolos terminais da gramática. A semântica da linguagem pode ser vista através do trecho da gramática e a respectiva representação gráfica, vinda do metamodelo.

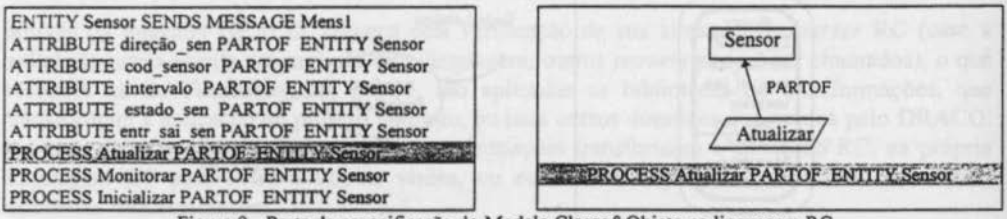



Figura 9 - Parte da especificação do Modelo Classe&Objeto na linguagem RC

- Aplicação de transformações horizontais. Estas transformações são aplicadas na especificação RC de uma técnica, para obter a especificação RC de um novo modelo em outras técnicas. No caso do exemplo, a nova técnica deve preservar a semântica do modelo de Classe&Objeto do método Coad. Conforme apresentado na seção 4.1, o domínio RC dispõe de bibliotecas de transformações horizontais que realizam transformações na própria linguagem RC. Estas transformações são responsáveis pelas múltiplas visões das especificações em diferentes modelos. A Figura 10 mostra parte das transformações usadas neste passo para transformar do modelo de Classe&Objeto para Diagrama de Fluxo de Dados. A Figura 11 mostra parte da especificação RC obtida pela transformação, ou seja, da visão em DFD. A representação gráfica equivalente é mostrada à direita na Figura 9. O contrário também é verdadeiro. Pode-se, a partir da especificação em DFD, obter uma representação em Coad. Ampliando a biblioteca de transformações horizontais, pode-se ter visões em diferentes métodos, nos quais se tenham representações equivalentes do modelo. É importante salientar que nem toda a semântica expressa em uma técnica de um método é sempre capturada na nova técnica transformada. Em certos casos, pode ser necessário completar textualmente a semântica.

- As transformações verticais são aplicadas sobre as especificações RC obtidas pelas transformações horizontais. O resultado deste passo são programas numa linguagem executável, como Pascal e C++, que implementam o sistema que está especificado.

Detalhes sobre a definição e aplicação de transformações horizontais e verticais são discutidos em Leite *et al.* [8,9] e Prado [14].

## 5. Conclusões

Este artigo apresentou um ambiente para representação de múltiplas visões de requisitos, descrevendo especificamente o metamodelo e a linguagem que estão sendo desenvolvidos. O metamodelo permite a criação de diferentes visões do sistema, sob o enfoque de diferentes técnicas de especificação. A linguagem tem por objetivo escrever o metamodelo de forma a viabilizar a implementação do sistema considerado.

```
Set Of Transforms CoadAE
    Method
            Search: Top-Down
            Apply: Single Step
    Transform Coad_AE1
    Lhs:
            {{dast RC.process
            PROCESS [[ID X]] PARTOF ENTITY [[ID Y]] }}
    Post-Match:
            {{dast cpp.statement_list
            SET_LEAF_VALUE ("Mens", expand("Men[[X]]"));
            TEMPLATE (TransfDfd1);
                    TRANSPORT_VALUE ("Mens");
                    TRANSPORT_VALUE ("Y");    TRANSPORT_VALUE ("X");
                    PLACE_AT("buffer");
            END_TEMPLATE; }}
    Template TransfDfd1
    *Rhs:
            {{dast RC.expression
                    ENTITY [[ID Y]] RECEIVES MESSAGE [[ID Mens]]
                    PROCESS [[ID X]] SENDS MESSAGE [[ID Mens]] }}
```

Figura 10 - Parte da biblioteca de transformações horizontais do domínio RC

```
ENTITY Sensor RECEIVES MESSAGE MenAtualizar
PROCESS Atualizar SENDS MESSAGE MenAtualizar
ENTITY Sensor RECEIVES MESSAGE MenMonitorar
PROCESS Monitorar SENDS MESSAGE MenMonitorar
ENTITY Sensor RECEIVES MESSAGE MenInicializar
PROCESS Inicializar SENDS MESSAGE MenInicializar
```

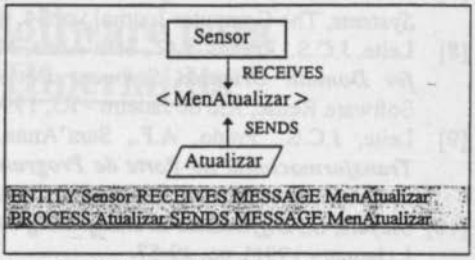



Figura 11 - Parte da especificação RC do DFD após a aplicação da transformação Coad_AE1

O trabalho descrito faz parte de um projeto de pesquisa, que envolve as seguintes atividades adicionais, em desenvolvimento ou previstas para um futuro próximo:

- Aplicação do metamodelo em diferentes estudos de casos, obtidos de áreas distintas de aplicação;
- Teste da potencialidade do metamodelo para apoiar outras técnicas de especificação de requisitos, tais como Statecharts, diagramas do método Hatley, máquinas de estado finito, etc.;
- Extensão e refinamento do metamodelo para atender requisitos de sistemas de tempo real;
- Implementação de procedimentos para translação entre cada técnica e o metamodelo, visando automatizar todo o processo de geração das múltiplas visões.
- Ampliação da biblioteca de transformações horizontais da linguagem RC, visando atender novas técnicas e modelos de outros métodos de especificação de requisitos;
- Construção da biblioteca de transformações verticais da linguagem RC, para mapear especificações RC para C++;
- Prototipação de um Banco de Dados Orientado a Objetos (BDOO) para suportar as múltiplas representações de requisitos de software;
- Geração de uma ferramenta para apoiar a especificação de requisitos de software, com o uso de múltiplas visões.

O ambiente pretendido será de grande auxílio para especificação de requisitos e trará, certamente, contribuições significativas para a geração de sistemas de software de alta qualidade.

## 6. Referências Bibliográficas


[1] Aho, A., Sethi, R., and Ulman, J., ***Compilers: Principles, Techniques and Tools***. Addison Wesley, Reading, Mass., 1986.

[2] Coad, P., North, D., and Mayfield, M., ***Object Models: Strategies, Patterns, & Applications***. Yourdon Press, New York, 1995.

[3] Davis, A.M., ***Requirements Engineering***. In Encyclopedia of Software Engineering, Vol. 2 (J. Marciniak, ed.), pp. 1043-1054, John Wiley & Sons, New York, 1994.

[4] Davis, A.M., et al., ***A Canonical Representation for Requirements***, 30 p., Technical Report, University of Colorado at Colorado Springs, 1995.

[5] Delugach, H. S., ***Specifying Multiple-Viewed Software Requirements with Conceitual Graphs***, Journal of Systems and Software, 19, 3 (March 1992), pp. 207-224.

[6] Freeman, P., ***A Conceptual Analysis of the DRACO Approach to Constructing Software Systems,*** IEEE Transactions on Software Engineering, 13, 7 (July 1987), pp. 830-844.

[7] Hull, M.E.C., O'Donoghue, P.G., Hagan, B.J., ***Development Methods for Real-Time Systems***, The Computer Journal vol 34, nº 2, Abril 1991.

[8] Leite, J.C.S., Freitas, F.G., Sant'Anna, M., ***Máquina Draco-PUC: A Technology Assembly for Domain Oriented Software Development,*** 3rd IEEE International Conference of Software Reuse, Rio de Janeiro - RJ, 1994.

[9] Leite, J.C.S., Prado, A.F., Sant'Anna, M., e Freitas, F.G., ***O Uso do Paradigma Transformacional no Porte de Programas Cobol***. IX Simpósio Brasileiro de Engenharia de Software - SBES 95, Recife, PE, Outubro de 1995, pp. 397-415.

[10] Meyers, S., ***Difficulties in Integrating Multiview Development Systems***, IEEE Software, 8, 1 (January 1991), pp. 49-57.

[11] Meyers, S. and Reiss, S., ***A System for Multiparadigm Development of Software Systems***, Proc. Sixth Int. Workshop on Software Specification and Design, IEEE, 1991, pp. 202-209.

[12] Neighbors, J., ***Software Construction Using Components***. PhD Thesis, University of California, Irvine, September 1980.

[13] Peterson, J., ***Petri Nets***, ACM Computing Surveys, 9,3 (1977), pp. 223-252.

[14] Prado, A.F., ***Estratégia de Re-Engenharia de Software Orientada a Domínios***, Tese de doutorado, PUC-RJ, Agosto 1992.

[15] Shumate, K., and Keller, M., ***Software Specification and Design: A Disciplined Approach for Real-Time Systems***. John Wiley & Sons, New York, 1992.

[16] Zamperoni, A., ***GRIDS - Graph - based, Integrated Development of Software***, 18th Int. Conference on Software Engineering, IEEE, 1996, pp. 48-59.