Métodos Formais para Desenvolvimento de Software: Um estudo de caso, o projeto GARDEN

Raul C. B. Martins,
Gustavo de O. Annarumma, Luiza M. F. Carneiro, Arnaldo V. de Lima,
Paschoal Molinari No., Elisabete M. B. de la Quintana, Ronaldo Stern
Centro Científico Rio IBM BRASIL
CP 4624, 20001 Rio de Janeiro - RJ - BRASIL

15 de Agosto 1989

Sumário

O trabalho pretende mostrar como um método formal de desenvolvimento de software pode vir a ser utilizado num projeto real. Apresenta-se um tutorial sobre o método em questão, simultaneamente ao projeto em discussão que consiste no desenvolvimento de um sistema de 'CAD' para 'VLSI'

Abstract

Formal methods can be as an aid in the development of a non-theoretical software projects. The development of a 'CAD' system for 'VLSI' using 'VDM' is presented as a case study.

1 Introdução

Um dos principais ingredientes de metodologias modernas para desenvolvimento de software e de ambientes de desenvolvimento encontra-se na maneira clara e precisa de se descrever as várias entidades que são manipuladas no processo de criação de software, além de expor de forma correta e objetiva todos os inter-relacionamentos destas entidades [7] [10]. Assim, tais metodologias oferecem ambientes apropriados para se tratar o processo de desenvolvimento de software de modo disciplinado, confiável e seguro. Mais ainda, e independentemente da metodologia específica onde são embutidos, os conceitos básicos que sustentam estas técnicas modernas proporcionam todo um arcabouço mental, sob o qual entidades e atividades fundamentais, tais como 'variável', 'rotina', 'recursão' e 'paralelismo', expõem sua verdadeira natureza. Isto, por sua vez, se traduz em grandes benefícios à atividade de construção de programas.

Apesar da distância entre métodos formais e informais não ter sido reduzida com o tempo, já existem métodos formais em uso, embora tais técnicas e métodos modernos

hoje só podem ser propriamente praticados e explorados em toda sua potencialidade, após dominar-se os conceitos e o ferramental matemático em que se apoiam.

De uma forma simplista estes métodos podem ser classificados em dois grandes grupos: métodos de especificação orientados a propriedades e métodos orientados a modelos. Nos métodos orientados a propriedades, como o próprio nome o diz, descreve-se, via um formalismo matemático as propriedades que o sistema deve ter. Como exemplos temos métodos algébricos, métodos apoiados em especificação em lógica. Nos métodos orientados a modelos, procura-se construir, a partir de objetos já conhecidos, um modelo que reflita as propriedades do sistema em discussão. Como exemplos, destes métodos, temos 'Z' desenvolvido em Oxford e *'VDM'*[1] [6] [5] [3] [12], cujo embasamento está fundamentado no conceito de denotações [14]. No caso específico deste trabalho, optou-se por uma metodologia orientada a modelos e finalmente por *'VDM'*.

Mantendo um cunho de praticidade, toda a apresentação fará uso de exemplos retirados do trabalho de desenvolvimento do GARDEN [8] [13] um sistema de CAD [9] ('Computer Aided Design'), para VLSI ('Very Large Scale Integration') em desenvolvimento no Centro Científico Rio da IBM Brasil.

O trabalho constitui-se num embrião natural que pode dar origem, a médio e longo prazos, a *produtos e ferramentas que sirvam a um ambiente integrado e moderno* de CAD, automatizando parte do processo. Tais produtos e ferramentas poderiam ser construídos implementando-se os resultados apresentados.

2 CAD para VLSI

2.1 Ambientes CAD

Sistemas de CAD não são apenas interfaces gráficas, coleção de rotinas matemáticas ou sistemas de gerenciamento de banco de dados. Sistemas de CAD constituem-se em coleções de ferramentas integradas objetivando a solução de complexos problemas de engenharia. Tipicamente um sistema de CAD é composto de programas tais como editores gráficos, interfaces humanas e de sistemas e programas aplicativos diretamente relacionados com o problema sendo resolvido [9]. Usualmente, o projeto e implementação de sistemas de CAD já são por si só uma tarefa difícil. O projeto de uma sistema de CAD para VLSI é, pela natureza da área de aplicação, de uma complexidade bem maior.

2.2 GARDEN

O GARDEN foi concebido como sendo um ambiente integrado para desenvolvimento de aplicações de VLSI que fosse independente dos ambiente computacionais hospedeiros do sistema. A estratégia principal do projeto GARDEN consiste em se ter uma série de interfaces especializadas (figura 1) que permitam um isolamento das ferramentas do ambiente GARDEN (aplicações) do ambiente computacional.

[1] 'VDM' é a sigla de 'Vienna Development Method'.

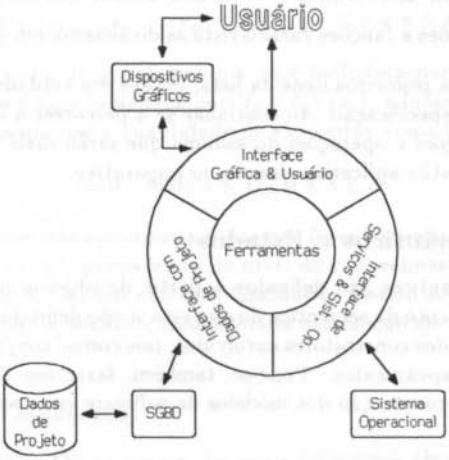

Figura 1: Ambiente GARDEN

Estas interfaces tem como objetivo permitir a evolução do sistema de CAD, em sintonia com a evolução dos ambientes computacionais, sem a necessidade de revisões e modificações nas ferramentas já existentes. Um outro objetivo destas interfaces é de orientar o ambiente GARDEN não somente para o projetista ou o ambiente de computação, mas também para o engenheiro de desenvolvimento de aplicações, permitindo que este se concentre em desenvolver algoritmos e novas ferramentas ao invés de rotinas dependentes de sistemas operacionais ou do 'hardware' disponível.

3 VDM e o GARDEN

3.1 Introdução

Como em todo método de especificação apoiada em modelos, seja formal ou não, ao construirmos uma especificação em *'VDM'* temos de construir um modelo explícito do objeto em questão. De uma forma simplista, não se estará apresentando o arcabouço matemático em que se apóia toda a técnica. Trocar-se-á a apresentação dos fundamentos por uma lista de ações a serem feitas. Essa lista depois tanto é utilizada no processo de desenvolvimento quanto no processo de revisão ou inspeção a que o produto resultante deve ser submetido. Esta lista consta de:

- definição dos estados e domínios semânticos;
- definição dos invariantes relativos aos estados já definidos;

- definição da sintaxe abstrata dos objetos em discussão e dos domínios sintáticos;
- definição das regras de boa formação relativas a sintaxe dos domínios sintáticos;
- definição das operações e funções características do sistema em projeto.

Ao completar-se os dois primeiros itens da lista, está-se em verdade completando-se o modelo do sistema em especificação. Ao continuar-se a percorrer a lista, os próximos itens caracterizam as funções e operações do sistema que serão mais tarde definidas e implementadas tanto no estilo aplicativo quanto no imperativo.

3.2 Domínios Semânticos e Estados

Estados e Domínios Semânticos são definidos a partir de objetos matemáticos bem definidos [11]. O entendimento da semântica do que está sendo definido confunde-se com a semântica, já conhecida, dos construtores envolvidos, tais como: conjuntos, seqüências, produtos cartesianos e mapeamentos. Pode-se, também, fazer uso de elementos pré-definidos que permitam a construção dos modelos de software em questão.

3.2.1 Domínios básicos

Definem elementos básicos tais como valores lógicos, naturais, inteiros, racionais, e outros conjuntos obtidos por enumeração de elementos atômicos.

3.2.2 Operações de conjuntos

As operações de conjuntos são as usuais. Listaremos estas operações mais para termos uma notação padrão de que para sua definição: enumeração, união, interseção, inclusão, conjunto potência, pertinência e outras.

3.2.3 Exemplos com conjuntos

Um repositório, que é o universo onde todos os objetos se encontram, pode ser definido como sendo a união discriminada de três conjuntos: um de bibliotecas, um de processos e outro de documentos.

Repositório = Bib-**set** + Processos-**set** + Documentos-**set**

Para testarmos se um elemento pertence a uma dessas classes de objetos temos:

$$\texttt{is-Bib}(l_i)$$
$$\texttt{is-Processos}(p_i)$$
$$\text{onde } 1 \leq i \leq n \text{ e } n \in \mathcal{N}$$

Uma biblioteca por sua vez é composta de outros conjuntos

Bib = Desenho-**set** + ...

Descrevemos uma biblioteca por enumeração, no caso:

$$l_1 = \{e_1, e_2, \ldots, e_i, \ldots, e_n\} \text{ onde } 1 \leq i \leq n \text{ e } n \in \mathcal{N}.$$

Descrevemos o acréscimo de um elemento a uma biblioteca por: $l_2 = l_1 \cup \{e_{n+1}\}$; a retirada de um elemento a uma biblioteca por: $l_3 = l_1 \setminus \{e_5\}$; a fusão de duas bibliotecas por: $l_4 = l_1 \cup l_5$ e representamos a quantidade de elementos numa biblioteca por:

$$t_i = \mathtt{card}\, l_i \text{ onde } 1 \leq i \leq n \text{ e } n \in \mathcal{N}.$$

A definição dos objetos está incompleta e estamos fazendo tanto uso da filosofia 'top-down' quanto da 'bottom-up', dependendo do nível do conhecimento de cada objeto.

A definição dos objetos Garden não está finalizada. Dentro do modelo contratual, em uso, esta definição tem evoluido e sido alterada em função do maior entendimento do problema em estudo.

3.2.4 Operações de seqüências

As operações de seqüências são as usuais. De novo, listaremos algumas operações mais para termos uma notação padrão do que para defini-las: enumeração, concatenação, primeiro ('head'), resto ('tail') e comprimento.

3.2.5 Exemplos com seqüências

Como parte constituinte de um documento variante ao longo do tempo e, com o objetivo de mantermos um histórico de trabalho, temos uma seqüência de nomes devidamente marcados por um 'time-stamp', além de arquivos de alterações que traduzem os efeitos de cada marca do tempo. Isto é expresso por:

Documento :: `seq of` TSNome + `seq of` TSDelta ...

Testamos instâncias destes objetos com:

`is- seq of` TSNome(l_i)
`is- seq of` TSDelta (d_i)
onde $1 \leq i \leq n$ e $n \in \mathcal{N}$

Uma determinada instância de uma `seq of` TSNome é descrita por:

$$l_1 = \langle t_1, t_2, \ldots, t_j, \ldots, t_n \rangle \text{ onde } 1 \leq j \leq n \text{ e } n \in \mathcal{N}$$

ou $l_2 = \mathtt{conc}\,(t_2, t_8)$ e também $l_3 = \mathtt{conc}\,(t_3, \mathtt{conc}\,(t_5, t_9))$.

3.2.6 Operações entre mapeamentos

As operações entre mapeamentos são as mais usuais: função, função das duas, domínio, contra-domínio, composição, união, etc ...

3.2.7 Exemplos com mapeamentos

Para expressar de uma maneira elegante que dentro de uma biblioteca existe um campo chave 'Id' que a identifica de uma forma unívoca, escreve-se:

$$\begin{aligned} \text{Bib} &= \text{Id} \xmapsto{m} \text{Aux} \\ \text{Aux} &= \text{Nome} \times \text{Bib-}\mathbf{set} \times \text{Documento} \times \text{Processo} \times \text{Desenho-}\mathbf{set} \end{aligned}$$

A definição de 'Id' resume-se a: Id = $\mathcal{N}$. Um Processo é definido, também por:

$$\begin{aligned} \text{Processo} &= \text{Id} \xmapsto{m} \text{Campo} \\ \text{Campo} &= \text{Nome} \times \text{Infos} \times \text{Documento} \times \text{Processo-}\mathbf{set} \end{aligned}$$

Descrevemos uma biblioteca por: $b_1 = [10 \rightarrow c_1]$.

De todos os construtores básicos de *'VDM'* este é o de mais difícil manipulação por pessoas com formação matemática mais fraca. Embora o conceito seja de uso geral, o uso deste objeto matemático e de seus derivados em especificação não tem sido trivial.

3.2.8 Operações entre árvores e produtos cartesianos

Árvores são objetos estruturados. Dependendo de como o objeto for definido temos operações que nomeiam estes objetos e, se for necessário, que selecionem componentes destes objetos. A uma equação do tipo

$$\text{NOME} :: (\text{CAMPO}_1 \times \ldots \times \text{CAMPO}_n)$$

associamos objetos notados por $\mathbf{mk}\text{-NOME}(c1, \ldots, c_n)$ onde $c_i \in \text{CAMPO}_i$ e $1 \leq i \leq n$.

A uma equação do tipo

$$\text{NOME} :: (tag_1 : \text{CAMPO}_1 \times \ldots \times tag_n : \text{CAMPO}_n)$$

associamos objetos $\mathbf{mk}\text{-NOME}(c_1, \ldots, c_n)$ ou $tag_i : \mathbf{mk}\text{-NOME}(c_1, \ldots, c_n)$ onde $c_i \in \text{CAMPO}_i$ e $1 \leq i \leq n$ e tag_i é uma função definida por:

$$tag_i : \text{NOME} \longmapsto \text{CAMPO}_i.$$

3.2.9 Exemplos com Árvores

Para exemplificar árvores pode-se modificar a definição de repositório e de biblioteca.

$$\begin{aligned} \text{Repositório} &:: \text{Bib-}\mathbf{set} \times \text{Processo-}\mathbf{set} \times \text{Documento} \\ \text{Bib} &:: \text{Nome} \times \text{Id} \times \text{Bib-}\mathbf{set} \times \text{Documento} \times \text{Processo} \times \text{Projeto-}\mathbf{set} \end{aligned}$$

Um repositório é representado por

$$\begin{aligned} &\mathbf{is}\text{-Repositório}(r_i) \text{ ou} \\ &\mathbf{is}\text{-Repositório}(\mathbf{mk}\text{- Repositório}(l_i, p_i, d_i)) \\ &\text{onde } 1 \leq i \leq n \text{ e } n \in \mathcal{N} \end{aligned}$$

3.2.10 Invariantes

Devem ser acrescidas às definições de domínio regras que permitam restringir quais elementos efetivamente devem ser considerados como pertencentes ao sistema em especificação. Acrescenta-se também propriedades que caracterizem o comportamento desses objetos no sistema. As primeiras são chamadas regras de boa formação e as segundas de invariantes.

Para fazer este tipo de definição faz-se uso de uma linguagem semelhante a uma linguagem de primeira ordem, acrescida de símbolos como `let` e `in` para facilitar a redação de sentenças, tais como: `let` $l_i \in Bib_1$ `in` ...

Invariantes são, então, restrições aos valores que os objetos que estamos especificando podem assumir. Sempre que estivermos definindo uma operação ou alterando algo no estado do sistema, precisamos verificar a validade do invariante. Podemos especificar um único invariante ou vários e, então, teremos como invariante a conjunção dos invariantes especificados. Vamos apresentar um desses invariantes:

$$\begin{aligned} \texttt{inv-Bib} \triangleq\ & \texttt{let mk-Bib}(n_i, in_i, w_i, p_i, d_i) = \text{Bib}(id_i) \\ & \wedge\ \texttt{let mk-Bib}(n_j, in_j, w_j, p_j, d_j) = \text{Bib}(id_j) \\ & \wedge\ \texttt{let } R \in \text{Repositório} \\ & \texttt{in } (id_i \neq id_j \Longrightarrow n_i \neq n_j) \wedge p_i \neq \emptyset \wedge \ldots \end{aligned}$$

Para criarmos estes invariantes, precisamos ter um conhecimento detalhado dos objetos com que estamos trabalhando. A especificação de invariantes nos cria, é claro, uma série de obrigações de prova [12].

3.3 Sintaxe Abstrata e Domínios Sintáticos

Embora sem completar as definições de todas as regiões ou domínios semânticos envolvidos num projeto, pode-se passar a definir os outros objetos envolvidos.

Um dos ítens a ser incluído numa especificação é a sintaxe abstrata dos comandos, programas e da linguagem com a qual os projetos estarão sendo desenvolvidos. Por sintaxe abstrata entende-se a definição e construção dos domínios envolvidos na sintaxe dos objetos em discussão.

Prevendo-se que a função de listar os elementos existentes num repositório, é imprescindível ao sistema em desenvolvimento, sua sintaxe abstrata é expressa por:

Listar	=	Listar-Bib | Listar-Processo | Listar-Doc
Listar-Bib	=	Bib*
Listar-Processo	=	Processo*
Listar-Doc	=	Doc*

Este tipo de definição continua até o completo detalhamento de todas as operações passíveis de serem feitas no sistema em definição.

3.3.1 Regras de boa formação

Da mesma forma que na definição da semântica, não se pode prescindir de regras de boa formação que caracterizem os objetos em questão.

$$\texttt{wwf-Listar-Bib} \triangleq \begin{array}{l}\texttt{let } l = \texttt{mk-Listar-Bib(bib)} \\ \texttt{in bib} = \emptyset \vee \forall x \in l \Longleftrightarrow x \in \text{Bib}\end{array}$$

De novo, a cada ítem especificado corresponde uma regra de boa formação. A regra colocada aqui como exemplo é extremanente simples e na continuação do projeto será alterada de tal forma que sempre reflita todas as propriedades atribuidas ao domínio em questão.

3.4 Operações e Funções

Tem-se também de se definir as funções que serão implementadas. Junto com as funções que implementam os aspectos principais da especificação, costuma-se também definir funções auxiliares que facilitam e fatoram o conjunto da primeira fase da especificação. À especificação inicial das funções segue-se um processo de concretização com obrigações de prova para cada passo realizado. O estilo de construção dessas funções pode ser o aplicativo ou o imperativo podendo haver trocas de um estilo para outro nos diversos passos do trabalho de tal forma que o conjunto total do trabalho de desenvolvimento atinja seus objetivos tais como: correção, desempenho e clareza. Em qualquer dos dois estilos as sintaxes abstrata e concreta e a semântica da linguagem de especificação podem ser bem definidas.

3.4.1 Estilo Aplicativo

O estilo aplicativo correspondente à programação funcional possui, como já dito, sua linguagem própria bem definida. No caso temos

$$\text{Listar-Bib}(l) \triangleq \begin{array}{l}\texttt{let } l = (\texttt{set of mk-Bib}(n_i, in_i, w_i, p_i, d_i)) \\ \texttt{in if } l = \emptyset \\ \quad \texttt{then } \emptyset \\ \quad \texttt{else let } x \in l \texttt{ in } x \wedge \text{Listar-Bib}(l \setminus x)\end{array}$$

$\texttt{type: Listar-Bib : Bib-set} \longrightarrow \texttt{Bib-set}$

Com a continuação do processo de implementação, a especificação tende a se aproximar de um programa escrito numa linguagem funcional.

3.4.2 Estilo imperativo: pré e pós-condições

Pré e pós condições são normalmente utilizadas para caracterizar uma função ou a operação por ela definida. Podemos caracterizar estas condições por:

$f : I \xrightarrow{m} O$

$\texttt{let } o = f(i) \texttt{ in } \forall i\big((i \in I \wedge \text{pre-}f(i)) \rightarrow (f(i) \wedge \text{pos-}f(i,o))\big)$

Num estilo imperativo tem-se:

```
pre-F
Function F
pos-F
```

onde pre-F é definida envolvendo as variáveis do domínio da função e pos-F é definida envolvendo as variáveis da função. Observe-se que pre-F e pos-F podem ser definidas como sendo funções cujo contra-domínio é BOOL. Uma função que define este objeto é:

$$\begin{aligned}&\text{Listar-Bib } (\texttt{set of mk-Bib}(n_i, in_i, w_i, p_i, d_i)) \triangleq \\ &\quad x \ ::= (\ \texttt{set of mk-Bib}(n_i, in_i, w_i, p_i, d_i)\ ) \\ &\quad \texttt{while } x \neq \emptyset \texttt{ do} \\ &\qquad\qquad \texttt{let } l \in x \\ &\qquad\qquad \texttt{output}(l) \\ &\qquad\qquad x \ ::= x \setminus l \\ &\quad \texttt{end}\end{aligned}$$

type: Listar-Bib : Bib-set $\longrightarrow$ Bib-set

As seguintes pré e pós condições podem ser acrescidas a função anterior:

```
pre-Listar-Bib   ≜   let f = Listar-Bib ( s) in
                     is-(set of  Lib(s))
pos-Listar-Bib   ≜   let f = Listar-Bib ( s) in
                     f = s
```

Passamos agora a ter a obrigação de provar que Listar-Bib é tal que, se objetos atendem à pré-condição estabelecida temos que a pós-condição é válida após a execução desta função. As atuais condições servem para exemplificar os conceitos de pré e pós condições, embora no decorrer do projeto estas condições venham a ser modificadas. Exemplos de captura de situações mais complexas também podem ser feitas. Dentro do sistema GARDEN, foi prevista uma forma de integridade entre bibliotecas, descrita por:

$$\begin{aligned}&\text{Acesso-Bib} \triangleq \ \texttt{let } z' \ = (\ \texttt{mk-Bib}(n'_i, l'_i, w'_i, p'_i, d'_i)\ ) \\ &\qquad\qquad \texttt{let } z \ = (\ \texttt{mk-Bib}(n_i, l_i, w_i, p_i, d_i)\ ) \\ &\qquad\qquad \texttt{let } p_i \ = (\ \texttt{mk-Processo}(n_j, i_j, w_j, c_j)\ ) \\ &\qquad\qquad \texttt{let } x \in l_i \ \texttt{in} \\ &\qquad\qquad \forall z', \forall x, (z' \in \text{Acesso-Bib}(z) \texttt{ if} \\ &\qquad\qquad z' \in l_i \vee \\ &\qquad\qquad z' \in \text{Acesso-Bib-aux}(x, p_i) \wedge (p_i = p'_i \vee p'_i \in c_j)))\end{aligned}$$

type: Acesso-Bib : Bib $\longrightarrow$ Bib-set

$$\begin{aligned}&\text{Acesso-Bib-aux} \triangleq \ \texttt{let } z' \ = (\ \texttt{mk-Bib}(n'_i, l'_i, w'_i, p'_i, d'_i)\ ) \\ &\qquad\qquad \texttt{let } \ldots = (\ \texttt{mk-Bib}(n_i, l_i, w_i, p_i, d_i)\ ) \\ &\qquad\qquad \texttt{let } p_i \ = (\ \texttt{mk-Processo}(n_j, i_j, w_j, c_j)\ )\end{aligned}$$

$$\texttt{let } p_j = (\ \texttt{mk-Processo}(n'_j, i'_j, w'_j, c'_j)\ )$$
$$\texttt{let } x \in l_i \texttt{ in}$$
$$\forall z', \forall x, (z' \in \text{Acesso-Bib-aux}(z, p_j) \texttt{ if}$$
$$z' \in l_i \vee$$
$$z' \in \text{Acesso-Bib-aux}(x, p_j) \wedge (p_i = p'_i \vee p'_i \in c'_j)))$$

type: Acesso-Bib-aux : Bib × Processo ⟶ Bib-set

No caso, a definição da função Acesso-Bib é de tal maneira complexa que foi definida uma função auxiliar, Acesso-Bib-aux, para completá-la. Não inclui-se a definição em linguagem natural de Acesso-Bib por questões de espaço. Observe-se também que em Jones [12] não é permitido um uso indiscriminado de atribuições como na definição de 'Listar-Bib' acima.

Figura 2: Grafo GARDEN

3.5 Um Modelo para Desenvolvimento de Software

Mesmo para se desenvolver um protótipo utilizando-se técnicas formais, faz-se uso do tradicional ciclo de vida de desenvolvimento de software [1]. Da mesma forma pode-se adaptar este esquema de ciclo de vida ao esquema de 'grafos de desenvolvimento de software' proposto por Bjorner [4] [2]. Um *grafo para desenvolvimento de software* chamado, para simplificar, apenas de grafo para desenvolvimento, é um grafo acíclico, orientado, com nós distinguíveis por rótulos. Pode-se distinguir, basicamente três tipos de grafos:

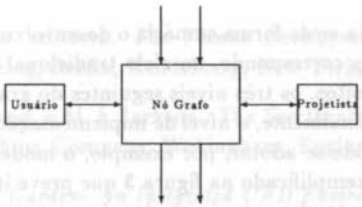

Figura 3: Modelo Contratual

- *meta-grafos*: assim chamados porque se aplicam ao desenvolvimento de uma determinada classe de software. Como exemplos, tem-se meta-grafo para desenvolvimento de ambientes de desenvolvimento de software, meta-grafos para desenvolvimento de sistemas operacionais, meta-grafos para sistemas de CAD;
- *grafos para projeto*: é um meta-grafo no qual foram incluidos detalhes para uma particular instanciação. Por exemplo, a partir de um meta-grafo para desenvolvimento de compiladores, constrói-se um grafo para o desenvolvimento de um compilador X;
- *grafo para configuração de produto*: uma vez instanciado, um meta-grafo passa a ser considerado um grafo de projeto. O passo seguinte consiste em ter-se grafos que permitam adaptar o resultado desse projeto a uma série de limitações de configuração ou de restrições de otimização ou de codificação: tem-se então uma série grafos para configuração deste produto. Por exemplo, a partir de um grafo para o desenvolvimento de um compilador X constrói-se grafos para a implementação deste compilador X para o sistema operacional Y a ser desenvolvido na linguagem T para a máquina R, ou grafos para a implementação do GARDEN para cada versão de sistema operacional ou linguagem alvo.

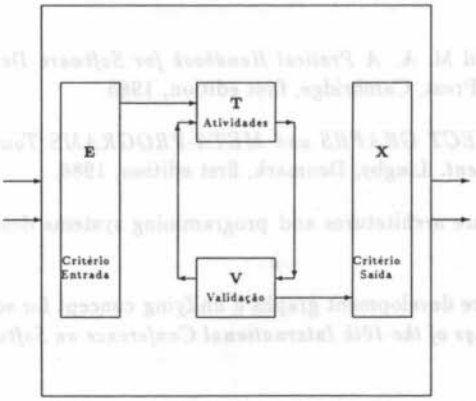

Figura 4: O Paradigma ETVX

No caso da figura 2 representa-se de forma sumária o desenvolvimento do projeto GARDEN. A definição de objetos corresponde, no ciclo tradicional de desenvolvimento de software, à Análise de Requisitos, os três níveis seguintes do grafo em discussão correspondem a fase de Projeto e, finalmente, o nível de implementação, à fase de Codificação.

Para cada nó do grafo pode-se adotar, por exemplo, o modelo contratual de desenvolvimento de software [7] exemplificado na figura 3 que preve iteração entre projetista e usuário até que haja um entendimento único dos documentos que constituem o nó. Para cada nó do grafo do projeto, exceto o nó Implementação, tem-se um contrato entre usário e o projetista. Do lado do usuário os documentos traduzem o conhecimento do domínio da aplicação, do lado do projetista o domínio da linguagem de especificação e a capacidade de transmiti-la para o usuário, pois as iterações entre as duas partes só cessam quando o usuário sinaliza que seus requisitos foram atendidos.

Para cada nó, visto do lado do projetista, toma-se como paradigma uma estensão do modelo **ETVX** [15], aplicando-o não só ao processo de arquitetura de programação, mas a todo o grafo do projeto. Deve-se, então, definir um critério de entrada, um critério de saída e critérios de validação para cada nó, para podermos completar os dados da figura 4. A estrutura formal do *'VDM'* facilita estas definições.

3.6 Conclusões e Estado Atual do Projeto

O projeto está em andamento. Existem cerca de cinqüenta páginas de documentação referentes tanto a especificação dos domínios semânticos quanto a seus invariantes e algumas das funções. Ao mesmo tempo estão sendo feitos não só os estudos necessários a adaptação do modelo **ETVX** à forma de desenvolvimento do GARDEN, como também experiências referentes à fase de implementação com o sistema WEB, WEAVE e TANGLE definido por Donald Knuth.

Referências

[1] N. D. Birrel e Ould M. A. *A Pratical Handbook for Software Development.* Cambridge University Press, Cambridge, first edition, 1985.

[2] D. Bjøner. *PROJECT GRAPHS and META-PROGRAMS Towards a Theory of Sofware Development.* Lingby, Denmark, first edition, 1986.

[3] D. Bjøner. Software architetures and programming systems design. 1989. Livro não publicado

[4] D. Bjøner. Software development graphs a unifying concept for software developement. In *Proceedings of the 10th International Conference on Software Engineering,* I.E.E.E., 1988.

[5] D. Bjøner e C. B. Jones, editores. *Formal Specification & Software Development.* Prentice-Hall International Englewood Cliffs New Jersey 1982

[6] D. Bjøner e C. B. Jones, editores. *The Vienna Development Method: The Meta-Language.* Springer Verlag, Berlin, Heildelberg, New York, first edition, 1978.

[7] B. Cohen, W. T. Harwood, e M. I. Jackson. *The Specification of Complex Systems.* Addison-Wesley Publishing Company, Wokingham, England, first edition, 1986.

[8] A. H. V. de Lima e alli. *Garden- An Integrated CAD Environement for VLSI/ULSI CAD Applications.* Rio de Janeiro, Brasil, first edition, 1989.

[9] J. E. Encarnação. *Computer Aided Design Modelling Systems. Lectures Notes in Computer Sience Vol 89,* Springer Verlag, Berlin, Heildelberg, New York, first edition, 1979.

[10] N. Gehani e A. D. McGettrick, editores. *Software Specification Techniques.* Addison-Wesley Publishing Company, Wokingham, England, 1986.

[11] M. J. C. Gordon. *The Denotational Descriptions of Programming Languages.* Springer Verlag, Berlin, Heildelberg, New York, first edition, 1979.

[12] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1986.

[13] J. Khakbaz e alli. *Garden- An Integrated CAD Environement for LSI/VLSI Design.* Rio de Janeiro, Brasil, first edition, 1988.

[14] R. C. B. Martins e Moura A. V. *Desenvolvimento Sistemático de Sistemas Corretos: A Abordagem Denotacional.* VI Escola de Computação, Campinas, 1989.

[15] R. A. Radice e alli. A programming process architecture. *IBM SYSTEMS JOURNAL,* 24(2):79–90, 1985.