31691-0

Centro Científico Rio

Relatório Técnico CCR-121

# Modelos de Representação e Gerência de Dados em Ambientes de Projeto de Sistemas Digitais

*por*

**Flavio Rech Wagner**

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

## Sumário

Um ambiente de projeto deve suportar um modelo de dados uniforme, que permita a representação de sistemas digitais como objetos complexos, considerando aspectos como composição e hierarquia, e ofereça recursos para a gerência dos dados de projeto, incluindo facilidades como múltiplas representações para um objeto e controle de versões e configurações. Este modelo de dados é a base sobre a qual são integradas as diversas ferramentas de projeto. Este trabalho faz uma análise dos requisitos principais para a representação e gerência de dados, e discute modelos de dados adotados em diferentes ambientes de projeto de sistemas digitais.



## Abstract

A design framework must support a uniform data model, which must allow the representation of digital systems as complex objects, where composition and hierarchy are essential issues, and offer facilities for design data management, including multiple representations for an object and version and configuration management. This data model is the basis for the tool integration process. This report analyzes the main requirements for data representation and management and discusses data models that are used in various design frameworks.



O autor está em licença da Universidade Federal do Rio Grande do Sul - Instituto de Informática e Curso de Pós-Graduação em Ciência da Computação

# Conteúdo

1 **Introdução** **1**

2 **Metodologias de projeto de sistemas digitais** **3**

3 **Requisitos de um ambiente de projeto** **7**
- 3.1 Modelos de dados x integração de ferramentas . . . . . 7
- 3.2 Requisitos para a gerência de dados . . . . . 8

4 **Requisitos de modelos de representação e gerência de dados** **10**
- 4.1 Representação de dados . . . . . 10
  - 4.1.1 Objetos complexos e primitivos . . . . . 10
  - 4.1.2 Múltiplas representações . . . . . 11
  - 4.1.3 Tratamento da interface . . . . . 13
  - 4.1.4 Atributos . . . . . 13
  - 4.1.5 Outros objetos . . . . . 15
- 4.2 Gerência de dados . . . . . 16
  - 4.2.1 Gerência da evolução do projeto . . . . . 16
  - 4.2.2 Configurações . . . . . 18
  - 4.2.3 Uso do tempo . . . . . 20
  - 4.2.4 Objetos parametrizáveis . . . . . 21
- 4.3 Relações entre objetos . . . . . 22
- 4.4 Restrições de integridade . . . . . 22

5 **Discussão de modelos** **25**
- 5.1 VHDL . . . . . 25
- 5.2 AMPLO . . . . . 26
- 5.3 Oct . . . . . 28
- 5.4 DAMASCUS . . . . . 30
- 5.5 GARDEN . . . . . 31
- 5.6 Outros modelos . . . . . 36

6 **Implicações dos modelos de representação e gerência de dados em outros aspectos de ambientes de projeto** **38**
- 6.1 Controle de metodologias de projeto . . . . . 38
- 6.2 Outros aspectos de gerência de dados . . . . . 39
- 6.3 Interface com o usuário . . . . . 40
- 6.4 Comunicação entre ferramentas . . . . . 41

# 1 Introdução

O processo de projeto de sistemas digitais complexos, em especial de circuitos VLSI, requer uma grande variedade de ferramentas. Um ambiente contendo ferramentas de diferentes fornecedores e dedicadas a vários níveis de projeto, aplicações e arquiteturas deve atender a inúmeros requisitos. Entre as principais questões a serem consideradas podem ser mencionadas a representação de sistemas como objetos complexos, a gerência de dados de projeto, a gerência de metodologias de projeto, o processo de integração de ferramentas, a uniformidade na interface com o usuário e a comunicação entre ferramentas.

Este relatório se dedica à discussão de modelos de dados para ambientes de projeto, que devem suportar recursos para a representação de sistemas digitais como objetos complexos, incluindo a modelagem de composição e hierarquia, e para a gerência de dados, incluindo facilidades como múltiplas representações para um objeto e controle de versões e configurações.

O relatório inicia na seção 2 por uma apresentação genérica do processo de projeto de sistemas digitais complexos, que procura estabelecer a motivação para os requisitos de ambientes de projeto relativos ao aspecto de modelo de dados, analisados a seguir na seção 3. Na seção 4 são então discutidas em detalhes as características desejáveis em modelos de dados. A seção 5 apresenta modelos de dados adotados em diferentes ambientes de projeto, e é feita uma análise de suas características à luz dos requisitos antes discutidos. Já durante a análise de requisitos na seção 4, várias caracteristícas dos ambientes estudados são antecipadas, a título de ilustração de diferentes soluções. Finalmente são feitas na seção 6 considerações gerais sobre as implicações dos modelos de dados em outros aspectos de ambientes de projeto não discutidos em detalhe neste relatório, tais como o controle de metodologias de projeto, o trabalho cooperativo, a comunicação entre ferramentas e a interface com o usuário.

Na terminologia hoje aceita internacionalmente, um "environment" é um conjunto de ferramentas integradas sobre um modelo de representação e gerência de dados comum e que apresentam uma interface uniforme com o usuário. Por outro lado, um "framework" é uma plataforma para a integração de ferramentas, que oferece, entre outras facilidades, um modelo de dados básico, eventualmente com a possibilidade de que o usuário possa definir esquemas conceituais particulares. Na terminologia em Português, ambos os conceitos são designados como "ambientes". Para os propósitos da discussão conduzida neste relatório, esta distinção não é essencial, já que um modelo de dados comum é necessário em ambos os tipos de ambientes. As implicações dos modelos de dados no processo de integração de ferramentas serão no entanto levadas em consideração como um dos aspectos mais importantes de ambientes de projeto.

Ao longo deste relatório será utilizada a expressão "modelo de dados" para designar o conjunto de objetos, relações, operações e restrições de integridade que

são definidos em função de uma aplicação ou conjunto de aplicações. Não se trata aqui de modelos de dados fundamentais, como o relacional ou o hierárquico, nem de modelos semânticos genéricos. Determinados ambientes de projeto, como [1,2], oferecem modelos semânticos orientados a aplicações de projeto, sobre os quais podem ser definidos diferentes esquemas conceituais. Este relatório não se destina à discussão de modelos de dados segundo estas acepções do termo.

# 2 Metodologias de projeto de sistemas digitais

Não se pode falar de uma metodologia de projeto "típica", já que a multiplicidade de sistemas possíveis de serem projetados leva a um grande número de metodologias imagináveis. Nesta seção, se tentará apesar disto apresentar aspectos comuns que podem ser abstraídos a partir destas metodologias, de modo a se estabelecer posteriormente um conjunto de características desejáveis em ambientes de projeto e, particularmente, em modelos de dados.

Segundo a terminologia proposta por Gajski e Kuhn [3] e hoje largamente aceita, o projeto de um sistema digital se dá ao longo de 3 eixos ortogonais de projeto, como se vê na Figura 1. Sobre cada um destes eixos encontram-se diversos níveis de projeto, que representam diferentes graus de abstração para a descrição do sistema digital em diferentes momentos do processo de projeto.

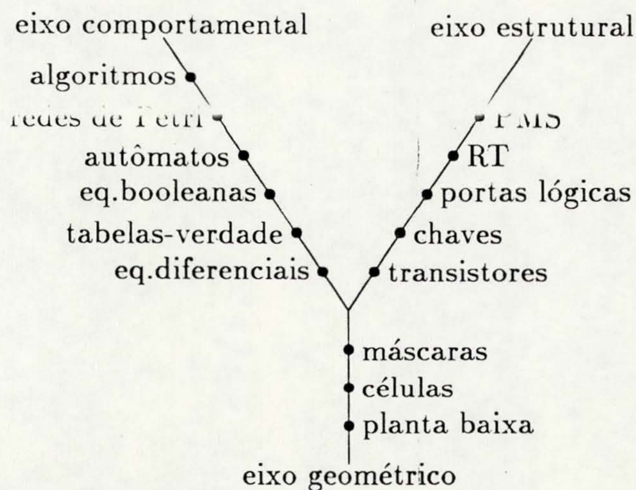


Figura 1: Eixos e níveis de projeto

No eixo comportamental descreve-se aspectos funcionais do sistema, sem qualquer ligação com a implementação física do mesmo. Pode-se descrever desde a função global do sistema em um nível bastante alto de abstração, através de um algoritmo ou de outro formalismo equivalente, como por exemplo uma rede de Petri, até a função de uma porta lógica através de um conjunto de equações diferenciais que explicam seu comportamento elétrico.

No eixo estrutural um sistema é descrito como uma composição de elementos que têm contrapartida física. Estes componentes podem ser desde módulos complexos, como processadores e memórias (no que se convenciona chamar de nível de sistema ou PMS), passando por registradores e elementos lógicos e aritméticos no nível RT, até transistores no nível mais baixo.

Sobre o eixo geométrico o sistema é descrito através de conjuntos de coordenadas que dão as localizações e dimensões exatas de módulos na implementação física. Os

diferentes níveis de abstração sobre este eixo dependem do tipo de implementação física a ser adotada, na forma de um circuito integrado ou de um sistema formado por placas e componentes discretos. No caso de um circuito integrado, por exemplo, se pode descrever o sistema desde um nível mais alto de abstração, na forma de uma planta baixa em termos de módulos como bloco de controle, bancos de registradores e unidades funcionais, até o nível detalhado da geometria interna de cada célula, em termos de polígonos nas diversas camadas de fabricação de um circuito integrado.

Em realidade, é muito difícil estabelecer uma definição de níveis de projeto que tenha aplicação universal. Para cada aplicação específica, na qual um determinado conjunto de ferramentas deve produzir um circuito com uma certa arquitetura numa determinada tecnologia, é definido um conjunto de níveis de projeto adequado. Sendo assim, determinada aplicação pode exigir uma descrição estrutural no nível lógico que inclua elementos mais complexos, tais como flip-flops e multiplexadores (o que tipicamente ocorre em bibliotecas de células para síntese em estratégias "semi-custom"), enquanto em outra aplicação o nível lógico deve ser definido unicamente como uma interconexão de portas lógicas elementares (o que é exigido em determinados procedimentos de síntese e otimização lógica).

Além disto, muitos formalismos permitem a descrição de aspectos de sistemas digitais que seriam teoricamente classificáveis em diferentes níveis e até em diferentes eixos de projeto. Isto é especialmente verdadeiro para determinadas linguagens de descrição de hardware, que procuram justamente abranger o maior espaço de projeto possível. VHDL [4], por exemplo, permite a descrição de aspectos comportamentais e estruturais a partir do nível lógico até níveis mais altos. Sendo assim, não se deve pensar em níveis de projeto como "pontos", mas sim como "regiões" do espaço de projeto.

Uma metodologia de projeto, ainda segundo Gajski e Kuhn, é uma seqüência de transformações envolvendo níveis sobre os eixos de projeto. Transformações podem ser de

- síntese automática: uma descrição num nível $i$ é gerada a partir de outra num nível $j$ superior
- verificação formal: descrições em níveis diversos são comparadas entre si
- análise: uma descrição num nível $i$ passa do estado "não validado" para o estado "validado" (exemplos são simulação, DRC e verificação de timing)
- extração: uma descrição num nível $i$ é gerada, total ou parcialmente, a partir de outra num nível $j$ inferior.

Metodologias de projeto podem seguir seqüências de transformações de diferentes naturezas:

- *top-down*, quando se parte de níveis mais altos de abstração para os mais baixos (i.e., se vai de uma idéia geral do sistema em direção aos componentes que realizam esta idéia)

- *bottom-up*, quando se parte de componentes previamente conhecidos, que vão sendo agregados de modo a se obter a função desejada
- *meet-in-the-middle*, quando se parte de um nível mais alto de abstração até se obter uma composição de módulos que foram gerados por um processo *bottom-up*.

Independentemente da estratégia adotada, todo projeto se dá através de aproximações parciais e sucessivas. Sobre uma descrição inicial vão sendo acrescentadas informações de composição, no caso de uma tarefa *bottom-up*, ou de refinamento, no caso de uma tarefa *top-down*. Neste processo, o projetista cria descrições que representam o sistema, ou partes selecionadas dele, em diferentes níveis de abstração.

Todo projeto procura explorar o espaço de projeto à procura de uma solução que melhor atenda determinados compromissos entre fatores considerados relevantes, tais como custo, velocidade de operação, potência consumida, área utilizada, etc. A exploração pode ser restrita a uma determinada tecnologia de fabricação (p.ex. apenas circuitos CMOS), ou a uma determinada estratégia de implementação (p.ex. "standard-cell" a partir de uma dada biblioteca de células), ou a uma determinada arquitetura (p.ex. o bloco de controle deve ser projetado com PLAs).

O projetista deve portanto mover-se dentro do espaço de projeto admitido, à procura da solução que melhor atenda às restrições estabelecidas na especificação. Nesta tarefa, ele certamente avaliará diversas alternativas de projeto. Esta característica "exploratória" do processo de projeto é extremamente importante, pois implica que o projetista terá para análise um número qualquer de descrições alternativas de um mesmo sistema ou módulo deste, e poderá continuar o projeto a partir de qualquer um destes, dependendo do resultado de sua avaliação.

Como se vê, o projetista manipula diferentes descrições de um mesmo sistema em função de dois processos, um deles criando descrições em diferentes níveis de abstração, o outro criando descrições alternativas a serem comparadas. A evolução do projeto ao longo destas duas dimensões segue um roteiro bastante complexo e muito dependente da aplicação particular. Determinadas decisões de projeto só fazem sentido em certos níveis de abstração. Tome-se como exemplo a escolha do número de fases de relógio: esta decisão é tipicamente tomada quando o projetista já tem uma descrição do sistema em termos de elementos unicamente estruturais do nível RT (registradores, interconexões e unidades funcionais). Assim, pode-se neste momento gerar duas alternativas, uma com relógio de 2 fases e outra com relógio de 4 fases. Não faz nenhum sentido pensar-se nestas alternativas para a descrição comportamental algorítmica inicial do mesmo sistema. A escolha do número de fases do relógio, no entanto, vai se refletir em todos os níveis de projeto inferiores em relação ao nível no qual esta decisão foi tomada. Assim, as duas alternativas deverão estar por exemplo presentes em descrições do mesmo sistema no nível de portas lógicas e no nível de *layout* das máscaras de fabricação, no caso do projeto de um circuito integrado.

É difícil estabelecer-se um padrão para a relação entre as decisões de projeto e os

níveis de abstração. Cada tipo de arquitetura, ferramenta de projeto ou tecnologia pode exigir diferentes decisões em diferentes níveis de abstração.

Outro aspecto importante no projeto de um sistema digital é o relativo à tecnologia de implementação. No caso do projeto de um circuito integrado, existem diversas tecnologias de integração (TTL, ECL, diversas tecnologias MOS, etc). Deve se considerar ainda as diferentes tecnologias de encapsulamento de um circuito integrado. No caso de um sistema composto por componentes discretos, deve ser considerada não só a tecnologia na qual foram implementados os componentes a serem utilizados como também as tecnologias de fabricação de placas (circuitos impressos em uma ou duas faces ou *multi-layer*, etc). As diferentes tecnologias de fabricação têm impacto sobre os níveis de abstração, principalmente sobre o eixo geométrico mas também sobre o eixo estrutural, e influenciam a escolha da metodologia de projeto, restringindo decisões nos níveis de abstração mais elevados.

# 3 Requisitos de um ambiente de projeto

Nesta seção são analisadas as implicações do processo de projeto de sistemas digitais, apresentado na seção anterior, na arquitetura geral de um ambiente de projeto. As implicações sobre o conteúdo do modelo de dados propriamente dito serão discutidas em detalhe na seção seguinte.

## 3.1 Modelos de dados x integração de ferramentas

Existe uma decisão inicial a ser adotada que impactará todo o projeto subseqüente do ambiente, e que diz respeito ao espectro de aplicações a ser coberto pelo mesmo. Sendo desejável um ambiente aberto a diversas tecnologias de fabricação, a diversas arquiteturas de circuitos e a diversas metodologias de projeto, é importante que se adote um modelo de dados bastante genérico e flexível e que se suporte mecanismos que permitam uma fácil integração de ferramentas.

Do ponto de vista do modelo de dados, pode-se adotar uma de duas estratégias que atendem a este requisito:

- pode-se definir um modelo que cubra todo o espectro de aplicações. Isto pode ser obtido de duas maneiras:
  1. definindo-se um modelo bastante completo, que contemple detalhadamente todos os objetos, atributos, relações e operações presentes no universo de aplicações. Esta abordagem pode levar a sérios inconvenientes na integração de ferramentas não previstas inicialmente.
  2. definindo-se um modelo bastante genérico (larga granularidade) que contenha objetos com pouca semântica própria (tais como "módulos" ou "células") e permita ao construtor da aplicação a definição de atributos e relações particulares.
- pode-se oferecer um modelo de dados básico e uma ferramenta que permita a construção de esquemas conceituais particulares para as diversas aplicações, como em [1,5].

A integração de ferramentas está intimamente associada ao modelo de dados adotado. Modelos ou esquemas conceituais detalhados permitem uma integração forte ("tight" ou "white-box integration"), na qual cada acesso a um dado é mapeado para um acesso ao banco de dados de projeto. Esta abordagem permite que sejam definidas relações e restrições de integridade referentes aos objetos primitivos dos diversos níveis de abstração que serão administradas pelo sistema de gerência de dados. Por outro lado, modelos genéricos implicam numa integração fraca ("loose" ou "black-box integration"), na qual a representação interna de um objeto num dado nível de abstração é armazenada como um "arquivo" cujo conteúdo o sistema de gerência de dados desconhece, e que é interpretada unicamente pelas ferramentas de projeto.

Alguns ambientes oferecem ferramentas que permitem simultaneamente definir a associação entre os dados acessados pelas ferramentas e os objetos do modelo de dados (como Ulysses [6] e CWS [1]), assim como definir o processo de integração de ferramentas (o sistema CWS oferece diferentes graus de integração em função de uma granularidade do modelo controlada pelo construtor da aplicação).

Dois requisitos adicionais devem ser considerados no projeto de um ambiente flexível. Em primeiro lugar, ele deve suportar a integração adequada de ferramentas bastante diversas. Ferramentas particulares podem manipular dados que não são apenas descrições de sistemas. O modelo de dados deve portanto prever (ou permitir a definição de) objetos tais como vetores de teste, resultados de simulação, informações de testabilidade, restrições de síntese, etc. Estes objetos têm atributos próprios e guardam determinadas relações com os demais objetos manipulados no ambiente. Em segundo lugar, o ambiente deve ser adaptável a diferentes tecnologias de fabricação. Tecnologias diversas (vários processos MOS, ECL, GaAs, etc) resultam em diferentes requisitos para o modelo de dados, especialmente nos níveis inferiores de abstração, e em particular em relação a informações sobre o eixo geométrico (diversidade de camadas de fabricação, diferentes regras de projeto, etc). Deve-se notar que o ambiente terá que atender simultaneamente a dois requisitos que podem parecer conflitantes, isto é, não ser restrito a nenhuma tecnologia em particular mas ser configurável para uma determinada tecnologia de modo a suportar de maneira eficiente o projeto nesta tecnologia.

Caso se tenha optado primariamente por um ambiente dedicado a uma aplicação específica, obviamente pode-se definir um modelo de dados e um processo de integração de ferramentas restritos, que resultarão eventualmente numa implementação mais eficiente do ambiente, embora às custas de perda de flexibilidade.

A integração de ferramentas não está no entanto relacionada unicamente ao modelo de dados. Exige-se ainda recursos para a definição de interfaces com o usuário uniformes, aspecto este bastante vasto e que não será discutido neste trabalho. Em alguns sistemas, a integração de ferramentas é feita dentro da definição da metodologia de projeto a ser seguida pelos usuários.

## 3.2 Requisitos para a gerência de dados

No campo da gerência de dados, o primeiro requisito é o controle da evolução do projeto. Já foi discutida na seção anterior a questão das múltiplas representações para um objeto, em função dos diversos níveis de abstração e das diversas alternativas de projeto a serem exploradas. O ambiente deve suportar estas múltiplas representações de um objeto e permitir que, quando um objeto é composto por diversos outros, possa ser escolhida para ele uma configuração constando de determinadas representações dos seus objetos componentes. A gerência de configurações também é portanto um requisito essencial em ambientes de projeto.

Num ambiente aberto, é ainda necessário que o conjunto de níveis de abstração

nos quais um determinado objeto pode ser representado seja extensível. Conforme discutido na seção anterior, o conceito de nível de abstração é bastante dependente da aplicação, de modo que o ambiente poderia ficar intoleravelmente inflexível caso fosse definido previamente um conjunto de níveis com propriedades fixas que permeassem o modelo de dados e as diversas facilidades do ambiente.

Múltiplas representações de um objeto são ainda geradas numa dimensão adicional em relação às duas já mencionadas. Toda descrição sofre consecutivas modificações, que vão aprimorando sua forma inicial até que se chegue a um resultado considerado aceitável. Não se trata aqui da análise de diferentes alternativas de projeto, mas sim de correções e aperfeiçoamentos de uma mesma alternativa. Estas representações são comumente denominadas versões de um objeto.

A gerência da evolução das múltiplas representações de um objeto está relacionada às estratégias de projeto *top-down*, *bottom-up* e *meet-in-the-middle*. Em cada um destes casos gera-se novas representações para um objeto, assim como novos objetos, a partir de representações já existentes, devendo-se manter a consistência com determinadas propriedades destas últimas.

Os aspectos de gerência de dados relacionados às múltiplas representações de um objeto e à gerência de versões e configurações serão discutidos em maior detalhe na seção seguinte. Será também discutido o inter-relacionamento entre a representação dos dados e estes aspectos de gerência de dados. Em muitos ambientes, o modelo de dados engloba de forma simultânea aspectos de representação de dados e de gerência de representações, versões e configurações.

Há ainda outros aspectos relacionados à gerência de dados que decorrem do trabalho em equipe, típico de projetos de maior porte:

- um modelo de cooperação deve ser estabelecido, implementando mecanismos que suportem o compartilhamento de objetos, mas também permitam restringir o acesso a objetos de acordo com uma determinada disciplina
- o ambiente deve ser implementado sobre uma plataforma distribuída, incluindo estações de trabalho e servidores.

Deve-se considerar ainda que um banco de dados de projeto apresenta características bastante diversas de um banco de dados convencionais em relação ao processamento de transações. Transações de projeto são tipicamente de longa duração e manipulam objetos complexos, ao contrário de transações convencionais, que são atômicas e manipulam objetos simples como registros numa tabela. Um mecanismo de transações de projeto é parte essencial de um modelo de cooperação.

Não é a intenção deste trabalho discutir detalhadamente as questões de cooperação e distribuição de dados. Na seção 6, no entanto, serão feitas algumas considerações para ilustrar as relações entre o modelo de dados e estes aspectos de gerência de dados.

# 4 Requisitos de modelos de representação e gerência de dados

## 4.1 Representação de dados

### 4.1.1 Objetos complexos e primitivos

Descrições de sistemas digitais possuem duas propriedades fundamentais: composição e hierarquia. Em função de sua complexidade, um sistema digital precisa ser descrito de forma modular, como uma composição de sub-objetos, de modo que cada sub-objeto possa ser analisado de forma separada, seja numa estratégia *top-down* ou *bottom-up*. Um objeto composto não contém as descrições de seus componentes, mas apenas referências a estas descrições. Desta forma, um objeto pode ser referenciado dentro de vários outros objetos compostos.

A segunda propriedade permite que o processo de composição (numa estratégia *bottom-up*) ou de decomposição (numa estratégia *top-down*) seja repetido sobre os novos objetos ou sub-objetos criados. Assim, todo objeto pode conter abaixo de si uma hierarquia de referências a outros objetos.

A composição deve iniciar a partir de objetos primitivos, não descritos em termos de outros sub-objetos, assim como a decomposição deve terminar por objetos primitivos.

Para a descrição de composição e hierarquia, o modelo de dados deve oferecer um conjunto básico de conceitos, que contém em princípio objetos compostos, objetos primitivos e componentes (referências a outros objetos). Estes conceitos não são no entanto suficientes para a descrição exata da composição. Todo objeto possui uma interface, composta por diversos sinais. Sinais de interfaces de sub-objetos distintos são conectados entre si. Bastante conhecido é o modelo "5-box", apresentado na Figura 2, onde a composição é precisamente descrita através dos conceitos *célula*, *pino*, *instância de célula*, *instância de pino* e *conexão*.

Este modelo possui larga granularidade. O objeto "célula" não tem uma semântica definida, podendo ser qualquer módulo, desde um processador até um único transistor. Neste tipo de modelo, um objeto primitivo pode ter uma representação interna (uma estrutura, um comportamento ou uma geometria) que não é tratada pelo modelo, sendo interpretada pelas ferramentas de projeto, que necessitam trabalhar sobre uma representação convencionada entre elas (por exemplo um padrão para intercâmbio de dados de projeto, tal como EDIF [7]).

No caso de se desejar uma granularidade fina, o modelo deve ser especificado em termos de objetos tais como portas lógicas, registradores, transistores, etc. Neste caso, todo acesso que uma ferramenta faz a dados de projeto é mapeado para acessos a objetos do banco de dados. Esta abordagem permite que o sistema de gerência de dados controle restrições de integridade bastante mais detalhadas, mas tem como desvantagem um processo de integração de ferramentas bastante mais complexo e

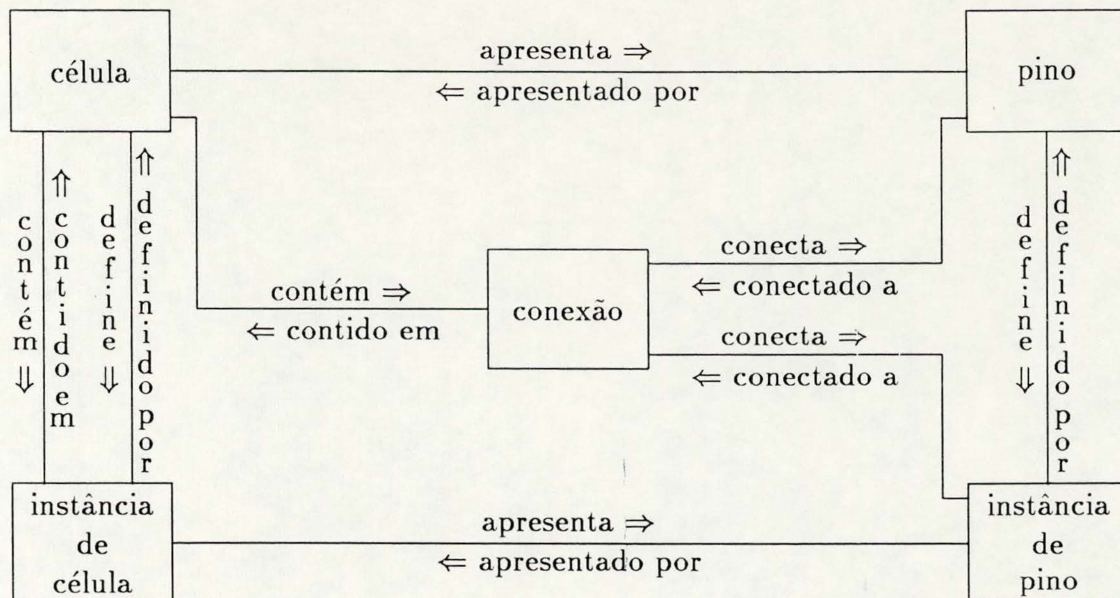


Figura 2: Modelo "5-box" para representação de dados

uma possivel inflexibilidade na integração de ferramentas que manipulem objetos não previstos quando do projeto e construção do ambiente. De modo geral, modelos com granularidade fina geralmente têm objetos primitivos estruturais (portas, transistores) e/ou geométricos (polígonos, camadas), já que é extremamente difícil estabelecer-se relações envolvendo primitivas comportamentais. Como exceção pode ser citado o modelo suportado no ambiente EVE [8], onde podem ser especificados *realization bindings* entre primitivas estruturais e primitivas geométricas e *operation bindings* entre primitivas comportamentais e primitivas estruturais. O ambiente EVE tem entre um de seus objetivos a síntese de alto nível incremental, para o que esta granularidade no eixo comportamental é necessária.

### 4.1.2 Múltiplas representações

Para a discussão que se segue, adotaremos a designação **vista** ("view") para indicar representações distintas de um objeto em diversos níveis de abstração, enquanto **alternativas** serão representações que correspondem a diferentes decisões de projeto. As versões de projeto serão discutidas posteriormente na sub-seção 4.2.1.

As dimensões da evolução do projeto estão relacionadas entre si por uma **estrutura de controle** hierárquica. É claro que o objeto em si (a célula, por exemplo) é a raíz desta hierarquia, e que eventualmente as versões estarão associadas às folhas. Cabe no entanto discutir como tratar vistas e alternativas nesta hierarquia, já que há requisitos que favorecem abordagens conflitantes.

Alternativas correspondem a decisões de projeto que tipicamente devem ser tomadas em determinados níveis de abstração. A escolha entre dois algoritmos que resultam em diferentes soluções para o compromisso entre área e velocidade, por

exemplo, é uma decisão de alto nível no eixo comportamental. A escolha entre uma arquitetura com barramentos e uma arquitetura com multiplexadores, por outro lado, é uma decisão a ser tomada no nível RT estrutural. Portanto, determinadas alternativas não fazem sentido em altos níveis de abstração: não há uma alternativa "standard-cell" no nível de sistema, por exemplo. Este argumento parece favorecer uma estrutura de controle onde vistas são agregados de alternativas, pois assim a cada vista serão associadas apenas as alternativas que fazem sentido naquele nível de abstração. Esta solução é adotada por exemplo no sistema DAMASCUS [9].

A solução não é no entanto tão simples, já que todo projeto passa por uma sucessão de decisões de projeto. Assim, um circuito integrado pode ter duas alternativas, uma rápida mas de área grande, e outra mais lenta mas de área menor. Estas alternativas resultam de decisões tipicamente tomadas no nível algorítmico. Para cada uma destas alternativas pode-se ter duas arquiteturas, uma com barramentos e outra com multiplexadores, sendo esta uma decisão tomada no nível RT estrutural. Uma próxima decisão, já no nível de composição de células no nível lógico, pode levar a mais duas alternativas para cada uma das anteriores, conforme a opção por uma estratégia "standard-cell" ou "gate-array". As oito alternativas geradas neste processo poderiam ser representadas de forma linear num mesmo nível da hierarquia da estrutura de controle, mas parece claro que a representação intrinsecamente apropriada é a de uma hierarquia de alternativas, onde os níveis da hierarquia estão relacionados a níveis de abstração (portanto a vistas). O ambiente GARDEN [10] oferece uma estrutura hierárquica flexível de ViewGroups que permite modelar adequadamente esta realidade.

A abordagem de representação de alternativas sob as vistas apresenta uma outra desvantagem, quando se considera a representação dos sinais de interface, que são atributos muito importantes dos objetos, como se verá na subseção 4.1.3. A decisão a respeito da estrutura de controle também está relacionada ao tratamento de relações de equivalência entre vistas e alternativas, conforme será visto na subseção 4.3.

Existe ainda uma relação a ser considerada entre as diversas vistas de um objeto e sua decomposição em sub-objetos. Há duas possibilidades de modelagem:

- todas as vistas compartilham a mesma decomposição, como em AMPLO [11], onde a decomposição não é representada nas vistas ("versões primitivas") mas sim numa vista especial ("versão composta") onde não há nenhuma informação específica de algum nível de abstração, mas apenas informação de composição
- cada vista pode apresentar uma decomposição própria, tal como nos ambientes GARDEN e EVE [8], de modo que as hierarquias sob as diversas vistas não são isomórficas.

Embora a segunda alternativa pareça mais flexível, ela implica em que, dentro das vistas, haja uma mistura de informações de composição com dados primitivos

de um certo nível de abstração. Esta mistura impede uma representação exata da composição, especialmente quando a vista corresponde ao eixo comportamental de projeto, o que traz desvantagens para o controle do processo de refinamento estrutural e para o controle de metodologias de projeto, conforme analisado em [12]. Em GARDEN isto foi resolvido parcialmente pela inclusão, ao lado das vistas que contêm informações de determinados níveis de abstração e que também podem fazer referências a sub-objetos, de uma vista especial ("MMHD") destinada unicamente à descrição exata de uma decomposição.

### 4.1.3 Tratamento da interface

Alternativas de projeto podem apresentar interfaces distintas entre si. A escolha de um tipo particular de registradores, por exemplo, pode levar a uma interface com relógios de 4 fases ou de 2 fases. No ambiente AMPLO [11], por exemplo, a cada objeto "agência" podem ser associadas diferentes alternativas que possuem interfaces distintas, enquanto no ambiente SDE [13] "classes" são agrupamentos de entidades VHDL que são funcionalmente semelhantes mas têm interfaces diversas.

Vistas de uma mesma alternativa também podem mostrar interfaces diversas, já que certos pinos só são relevantes em níveis inferiores de abstração. As fases do relógio aparecem apenas após a síntese estrutural para o nível RT, enquanto pinos de alimentação só precisam ser representados quando o *layout* é gerado. Esta constatação leva à idéia de se representar todos os pinos comuns às diversas vistas de uma alternativa num mesmo ponto da estrutura de controle, o que favoreceria a modelagem de alternativas como agregados de vistas.

Considerando no entanto a natureza hierárquica do processo de tomada de decisões de projeto, parece mais adequado associar os sinais de interface a níveis de uma hierarquia de alternativas. Assim, em cada nível, que corresponderia na realidade a um certo nível de abstração, seriam representados os sinais de interface acrescentados pelo projetista naquele nível. A interface de uma representação completa de um objeto seria obtida por um mecanismo de herança de sinais, no qual um nodo qualquer da hierarquia herdaria todos os sinais dos nodos colocados diretamente entre ele e a raíz (nodos ascendentes). Este esquema é adotado no ambiente GARDEN [10]. Outros mecanismos de herança de sinais também existem nos ambientes Oct [14] e DAMASCUS [9].

### 4.1.4 Atributos

Os objetos do modelo de dados possuem conjuntos de atributos bastante variados. Atributos podem ser dados de projeto, tais como valores de atrasos de propagação num certo módulo, ou dados de gerência, tais como o nome do projetista deste módulo e a data de liberação de uma certa representação do mesmo.

Os atributos que mais interessam nesta análise são obviamente aqueles que correspondem a dados de projeto. Estes atributos podem ser associados a diferentes

entidades do modelo de dados:

- a interface de uma célula em um CI pode ter como atributos suas dimensões e limites máximos de deformação em cada dimensão para uso por uma ferramenta de posicionamento ou de planejamento topológico
- sinais de interface podem ter como atributos nome, sentido (entrada, saída ou bidirecional), tipo de dado, largura em bits, posição ao longo da interface e camada do *layout* em que estão implementados
- conexões entre sinais de sub-objetos de um objeto composto podem ter como atributos traçado (seqüência de segmentos de reta), largura e camada do *layout* em que estão implementadas (eventualmente cada segmento de reta pode estar implementado numa camada distinta)
- sub-objetos em um objeto composto podem ter como atributos um nome, sua posição geométrica relativa ao objeto e fatores de escala aplicados a suas dimensões

Atributos podem ser de diferentes tipos de dados, como valores inteiros, reais ou booleanos, *strings*, registros e até eventualmente arquivos inteiros (no caso de informação de natureza documentacional).

Embora determinados atributos, tais como o nome de um objeto e o sentido de um sinal de interface, sejam suficientemente comuns para estarem presentes em todas as aplicações, e assim serem previstos de maneira fixa no modelo de dados, cada aplicação certamente terá atributos particulares em função da natureza das ferramentas a serem utilizadas, das arquiteturas a serem projetadas ou das tecnologias a serem adotadas. É assim essencial que o modelo de dados permita que o construtor da aplicação defina seus próprios atributos. Uma ferramenta apropriada para esta definição deve ser prevista.

Um aspecto relevante a ser considerado no tratamento da interface dos objetos é a visibilidade de determinados atributos para diferentes ferramentas de projeto. No projeto de um circuito integrado, por exemplo, um roteador precisa conhecer determinados atributos geométricos da interface das células, tais como as camadas do *layout* em que estão implementados determinados sinais de interface e a posição exata de cada sinal na envoltória da célula. Estes atributos são totalmente irrelevantes para um simulador elétrico, enquanto um editor de esquemáticos precisa conhecer a posição dos sinais na envoltória mas não necessita da informação sobre as camadas de fabricação. Esta propriedade de ambientes pode ser obtida através de um mecanismo de visões, à semelhança do que existe em bancos de dados convencionais. O sistema Oct [14] oferece um recurso com resultado análogo, pelo qual a uma vista de um objeto podem ser associadas diferentes "facetas", cada uma contendo determinados atributos da vista visíveis externamente.

Também é importante que se considere a relação entre os atributos e as várias dimensões da evolução do projeto. Determinados atributos podem ser particulares

de certos níveis de abstração, não sendo visíveis nos demais níveis. Outros atributos, por sua vez, são comuns a diversos níveis de abstração. Atributos definidos pela aplicação podem ser comuns a certas alternativas de projeto. Seja qual for a estrutura de controle adotada (com vistas acima de alternativas ou vice-versa, com ou sem hierarquia de alternativas), é interessante a existência de um mecanismo de herança de atributos, da raíz da estrutura em direção às folhas. A aplicação deve no entanto poder definir quais atributos devem ser herdados, embora o modelo possa prever certas heranças automáticas (como sinais de interface, por exemplo).

O sistema Fred [15] permite que sejam associados valores *default* de atributos a vistas de um objeto. Alternativamente, o usuário pode especificar uma função que calcula rapidamente um valor estimativo de um atributo quando este não está declarado na descrição da vista. Com isto o usuário pode descrever a função ou estrutura interna desta vista de forma incompleta. O sistema automaticamente procurará, para cada atributo cujo valor não foi fornecido, seu valor *default*, ou, na ausência deste, ativará a função que calcula o valor estimativo.

### 4.1.5 Outros objetos

"Células" ou "módulos" são os objetos principais de um modelo de representação de dados, contendo descrições de comportamento, estrutura e geometria de sistemas. Muitas ferramentas exigem no entanto a manipulação de objetos auxiliares adicionais. Pode-se citar como exemplos

- estímulos e resultados de simulação,
- vetores de teste,
- informações de testabilidade,
- regras de projeto e outras informações tecnológicas, e
- restrições de síntese (área, velocidade, potência, etc).

Estes objetos têm seus próprios atributos e guardam relações particulares com os demais objetos. Assim sendo, pode-se armazenar as seguintes relações:

- um *conjunto de estímulos* aplicado por um *simulador* a uma *representação* de um sistema gera *resultados de simulação*
- um *gerador de vetores de teste* gera *vetores de teste* a partir de uma *representação* de um sistema
- um *verificador de regras de projeto* aplica determinadas *regras de projeto* sobre uma *representação* de um sistema
- um programa de síntese automática para PVT (projeto visando testabilidade) gera uma nova *representação* de um sistema a partir de uma *representação* anterior do mesmo e de *informações de testabilidade*, obtidas por um *analisador de testabilidade*, inserindo *estruturas de teste* armazenadas numa biblioteca de células

Como se vê, as relações podem envolver uma ou mais representações de sistemas, um ou mais objetos auxiliares e uma ferramenta de projeto. Numa relação, determinados objetos são de entrada (são usados pela ferramenta) e outros são de saída (são gerados pela ferramenta).

Considerando a evolução constante das ferramentas, metodologias e tecnologias de projeto, é extremamente duvidoso que se possa estabelecer a priori um conjunto de objetos, atributos e relações que sirva a todas as aplicações a serem desenvolvidas durante o tempo de vida do ambiente. Parece bem mais razoável oferecer ao construtor da aplicação um mecanismo genérico para definir objetos auxiliares e seus atributos e criar relações genéricas envolvendo ferramentas e objetos (ver subseção 4.3).

## 4.2 Gerência de dados

### 4.2.1 Gerência da evolução do projeto

Na subseção anterior já foram discutidas duas dimensões da evolução do projeto, correspondendo a vistas e alternativas do projeto. Foram também analisadas as relações entre estas duas dimensões.

É bastante discutível sob qual tópico esta questão deva ser discutida, se como um aspecto de representação ou de gerência de dados. Foi mostrado que a estrutura de controle que relaciona vistas e alternativas guarda uma relação importante com o tratamento dos sinais de interface, motivo pelo qual a discussão foi inicialmente travada pelo aspecto de representação de dados.

Existe uma terceira dimensão na evolução do projeto, que corresponde às correções e melhoramentos consecutivos que vão sendo feitos sobre uma mesma representação de um módulo até que o projetista considere-se satisfeito com o resultado. As representações obtidas neste processo são denominadas genericamente de **versões**. Esta dimensão pertence claramente à esfera da gerência de dados.

Mais uma vez coloca-se a questão de como tratar esta dimensão dentro da estrutura de controle. Que representações têm versões? As vistas, as alternativas, ou ambas? As versões são geralmente associadas apenas a vistas ou apenas a alternativas, dependendo de qual destas ocupa posição inferior na hierarquia. Assim, no sistema DAMASCUS [9] as versões, denominadas revisões, são associadas às alternativas (que por sua vez são agrupadas em vistas), enquanto no sistema Oct [14] as versões são vinculadas às facetas, que são visões particulares das vistas. O armazenamento das versões nas folhas da estrutura de controle permite que, na prática, se possa guardar versões para cada combinação "alternativa – vista", independentemente da relação que estas guardam entre si.

A questão adicional que se coloca é a relação que existe entre as versões de uma mesma representação (seja uma alternativa ou uma vista). Diferentes soluções são adotadas. Assim,

- no sistema AMPLO [11] as versões são armazenadas como uma seqüência linear no tempo sob cada alternativa,
- no ambiente GARDEN [10] as versões são armazenadas na forma de uma árvore sob cada vista, onde filhos de um mesmo nodo são denominados modificações, e filhos únicos de um nodo são denominados iterações,
- no sistema DAMASCUS [9] revisões são versões de uma alternativa armazenadas na forma de um grafo acíclico, enquanto para cada revisão pode-se ainda ter uma seqüência linear de estágios de projeto.

A representação do histórico de versões na forma de um grafo permite que o usuário relembre o fato de que determinada descrição foi obtida a partir da combinação de idéias de várias descrições anteriores. A representação em árvore ou grafo permite que se registre o fato de que uma dada versão foi a geradora de várias outras.

É bastante útil que o usuário possa gerar novas versões não apenas a partir da última versão (aquela gerada mais recentemente), mas também a partir de versões anteriores. Versões devem portanto ser designadas de maneira a serem identificadas e recuperadas posteriormente. Uma representação seqüencial de versões impede esta facilidade.

À maior riqueza da representação do histórico de versões na forma de um grafo ou de uma árvore corresponde uma maior complexidade na implementação do sistema gerenciador de dados e na gerência de configurações pelo usuário (esta última questão será discutida na subseção 4.2.2).

A evolução do histórico de versões, quando este é representado por uma árvore ou por um grafo, pode ser controlada pelo usuário, que fica então responsável pela indicação da posição de cada nova versão dentro da estrutura já existente. Um controle automático pelo próprio sistema gerenciador de dados é possível de ser feito no caso de um histórico em árvore, mas torna-se complexo para o caso de grafos.

Deve-se distingüir entre o caráter documentacional do histórico de versões e a implementação do armazenamento das versões no sistema de banco de dados. A solução trivial para a implementação é o armazenamento completo de cada versão. Esta solução pode ser extremamente custosa quanto ao espaço consumido na memória secundária. Pode-se pensar em armazenar as versões na forma de arquivos diferenciais, usando-se diferenças positivas (guarda-se na nova versão apenas a diferença dela para a versão anterior) ou negativas (guarda-se a versão mais recente de forma completa e na versão anterior é calculada e armazenada sua diferença em relação à nova versão). Neste caso, o histórico pode servir de guia para o cálculo das diferenças.

Um mecanismo adicional que permite economizar memória secundária é o arquivamento de versões não mais utilizadas em meios de acesso mais esporádico, como fita magnética. O arquivamento do conteúdo propriamente dito da versão não pre-

cisa corresponder necessariamente no entanto à remoção da referência a esta versão no histórico de versões. Referências a versões não deveriam ser nunca removidas, já que o registro histórico é importante, mesmo que o objeto não seja mais de interesse.

Note-se que estas considerações em relação ao histórico e armazenamento de versões não se aplicam às alternativas e vistas. Em primeiro lugar, as modificações no projeto que geram novas versões são bem mais freqüentes e de menor envergadura do que aquelas que respondem pela criação de alternativas e vistas. Assim, o número de versões é bem maior do que o número de alternativas e vistas. Em segundo lugar, não há sentido em representar um histórico de vistas, embora o haja para o histórico de alternativas, conforme já discutido na seção 4.1.2.

### 4.2.2 Configurações

Uma representação composta $Y^k$ de um objeto $Y$ contém componentes $C_i, ...C_j$ que são instâncias de outros objetos $X_i, ..., X_j$. Como cada objeto $X_n$ pode possuir múltiplas representações (alternativas, vistas e versões), deve-se selecionar para cada uma de suas instâncias uma **configuração** válida do mesmo, que contenha a referência exata a uma folha da sua estrutura de controle. Supondo que a estrutura de controle tenha alternativas como agrupadores de vistas e estas como agrupadores de versões, é necessária a seleção de um caminho "alternativa $\Rightarrow$ vista $\Rightarrow$ versão" para o $X_n$ associado a cada $C_n$, sem o que a descrição de $Y^k$ não poderá ser utilizada por nenhuma ferramenta de projeto.

O momento desta seleção não precisa no entanto coincidir com o momento da criação da descrição de $Y^k$. A seleção pode ser postergada para o momento em que a ferramenta de projeto será aplicada sobre $Y^k$. Esta postergação tem como vantagem o armazenamento no banco de dados de uma única descrição de $Y$, genérica, que pode ser posteriormente configurada para diferentes representações possíveis de seus componentes.

Diz-se que $Y^k$ tem uma **configuração dinâmica** [16] quando a seleção das configurações de seus componentes não está fixada dentro da própria descrição de $Y^k$. Diz-se, em caso contrário, que $Y^k$ tem uma **configuração estática**. É ainda possível se definir para $Y^k$ uma **configuração aberta**, no caso em que não são associados objetos $X_n$ a seus componentes $C_n$.

No caso do modelo de dados implementar uma estrutura de controle com vários níveis, surge a possibilidade adicional de se especificar configurações parcialmente dinâmicas, onde a seleção é feita, em tempo de criação da descrição $Y^k$, até um certo nível da estrutura de controle dos objetos $X_n$. O restante da configuração, até o nível das versões (suposto como o último da hierarquia na estrutura de controle), é feito quando da aplicação de uma ferramenta sobre $Y^k$. Esta facilidade está presente no ambiente GARDEN [10]. Em [17] é ilustrado de que forma este recurso pode ser utilizado num processo de projeto.

Qualquer que seja a estrutura de controle adotada no modelo de dados, o processo de configuração depende da maneira pela qual os sinais de interface dos objetos

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

são representados nesta estrutura. Só se pode criar uma composição a partir de descrições de objetos cujas interfaces são conhecidas. Assim, a referência a um objeto numa configuração dinâmica deve alcançar o nível da hierarquia da estrutura de controle onde a interface está representada.

Esta restrição não se aplica a configurações abertas, como em VHDL [4], onde o projetista especifica uma interface para cada componente de maneira inteiramente livre, dentro da própria descrição do objeto $Y$ que contém o componente. Obviamente haverá uma restrição posterior no momento da seleção de uma representação de um dado objeto $X$ para este componente, já que as interfaces deverão se ajustar. VHDL permite, como flexibilidade adicional, que o objeto $X$ selecionado tenha mais sinais de interface do que o componente ao qual ele está associado. Neste caso, alguns sinais do objeto $X$ permanecerão não conectados.

A desvantagem das configurações abertas e dinâmicas está na necessidade de se gerenciar as configurações. Pode-se criar objetos auxiliares, que aqui serão denominados de **descrições de configurações**, tal como é feito em VHDL por exemplo, que contêm configurações válidas de $Y^k$ a partir da seleção de configurações para seus componentes $C_n$. Quando da aplicação de alguma ferramenta sobre $Y^k$, basta a seleção de uma descrição de configuração já existente para este objeto. Caso o ambiente não preveja o armazenamento de descrições de configurações como objetos no banco de dados, a cada aplicação de uma ferramenta sobre $Y^k$ terá que ser efetuado um novo processo de configuração. A maior desvantagem deste procedimento está na natureza hierárquica dos objetos. Cada representação $X^m$ selecionada para um componente de $Y^k$ pode ser também um objeto composto, com seus próprios sub-componentes, e o processo de configuração tem que ser repetido sobre cada um destes, até que se complete para $Y^k$ uma hierarquia que tem como folhas objetos primitivos.

Um recurso interessante seria a possibilidade de se montar descrições de configuração para $Y^k$ utilizando-se hierarquicamente descrições de configurações para os objetos $X^m$ associados a seus componentes. Esta facilidade é oferecida pelos ambientes PLAYOUT [18] e da empresa Cadence [19].

Outro recurso bastante útil é a construção automática de configurações a partir de critérios definidos pelo usuário. Imagine-se uma estrutura de controle onde se tenha unicamente vistas e versões. Pode-se criar uma configuração para um dado objeto selecionando-se automaticamente, para cada um dos objetos $X$ associados a seus componentes, a versão mais recente da vista de $X$ no nível de portas lógicas, por exemplo. Este tipo de construção automática de configurações é utilizado por exemplo no ambiente de simulação do sistema AMPLO [20]. Um mecanismo mais sofisticado é proposto em [13,21], onde o processo de configuração em VHDL é guiado por restrições especificadas pelo usuário e que podem envolver atributos variados dos objetos (por exemplo valores de parâmetros, ver subseção 4.2.4). Tal tipo de recurso permite que se restrinja uma configuração a uma determinada tecnologia de fabricação (selecionar apenas alternativas CMOS, p.ex.) ou a uma determinada

estratégia de implementação (selecionar apenas alternativas *standard-cell*, p.ex.).

### 4.2.3 Uso do tempo

Certos ambientes propõem o uso do tempo como recurso para o gerenciamento de versões e configurações. O exemplo a seguir, extraído de [22], ilustra a ortogonalidade dos conceitos de "versão" e de "tempo" para controlar a evolução do projeto. Considere-se uma representação $Y^k$ de um objeto $Y$ contendo um componente $C$ ao qual é associado um objeto $X$. A representação $Y^k$ possui uma configuração dinâmica, que automaticamente associa a versão mais recente de $X$ a $C$. Num tempo $T_a$ é feita uma simulação de $Y^k$ utilizando-se a versão $X_3$, que é então a mais recente. Num tempo $T_b$ posterior é criada uma nova versão $X_4$. Se o usuário for interpretar os resultados de $T_a$ recriando a configuração de $Y^k$ em um $T_c > T_b$, ele certamente fará uma análise equivocada. Os números que identificam as versões não são portanto suficientes: é necessário que se associe um tempo às versões de objetos. Assim, a configuração de $Y^k$ em $T_a$, mesmo que recriada em um tempo posterior, utilizará apenas versões com tempo de criação $T_x < T_a$.

Biliris [22] classifica atributos em **atributos temporais**, aqueles cujos valores são função do tempo e para os quais é automaticamente mantida no banco de dados toda a seqüência de valores passados, e **atributos "versionáveis"**, para os quais é mantido um histórico de versões. Estes dois grupos de atributos não são necessariamente disjuntos. Biliris propõe que seja deixada ao usuário a decisão de quais atributos são temporais, já que este tipo de controle é bastante custoso para a implementação do sistema gerenciador de dados.

Como exemplo, a evolução dos valores de atributos, tanto os definidos pelo usuário como certos atributos intrínsecos do modelo (como nomes de objetos), é controlada em GARDEN [10] através de **TimeStamps**. Sempre que o valor de um destes atributos é alterado, o sistema automaticamente associa este valor a um TimeStamp com o tempo corrente e o armazena numa seqüência de pares "valor – TimeStamp". Quando um objeto é recuperado do banco de dados por uma aplicação, o usuário pode especificar um tempo passado $T_i$ qualquer: serão recuperados os valores dos atributos cujos TimeStamps forem iguais ou mais próximos de (mas menores do que) $T_i$. A referência a um objeto genérico $X$ associada a um componente de outro objeto $Y$ tambem é um atributo temporal em GARDEN, com o que configurações dinâmicas podem ser controladas pelo tempo, mas no entanto a referência a uma versão particular de $X$ é um atributo "versionável" mas não temporal (em GARDEN estes dois grupos de atributos são disjuntos).

Em Oct [14] TimeStamps são utilizados para controlar o armazenamento de todas as modificações efetuadas sobre as facetas (visões das vistas). Oct também utiliza o tempo para gerenciar configurações [23]: uma configuração em Oct é um conjunto de representações para objetos de uma determinada área de trabalho do usuário num dado instante de tempo (ver seção 5.3). A simples referência a um valor de tempo passado automaticamente permite que se recupere todo este conjunto de

representações.

### 4.2.4 Objetos parametrizáveis

Um objeto parametrizável é um objeto que possui características variáveis. A variação se dá no valor de determinados atributos, denominados então de **parâmetros**. O valor de um parâmetro pode condicionar

- o comportamento do objeto, como no caso do valor de um atraso de propagação de uma porta lógica no interior do objeto
- a geometria do objeto, como no caso de um fator de escala a ser aplicado sobre as dimensões do objeto como um todo ou a dimensões de determinados elementos no seu interior
- a estrutura interna do objeto, quando o valor do parâmetro altera o número de sub-objetos ou a forma de interconexão entre eles.

Quando da definição de um objeto parametrizável $X$ são especificados um conjunto de parâmetros e determinados intervalos para seus valores. Quando este objeto $X$ é instanciado como componente em outro objeto composto $Y$, os valores destes atributos variáveis precisam ser fixados, para que o objeto $Y$ tenha uma configuração completamente especificada e possa ser usado por alguma ferramenta de projeto. O mecanismo de parametrização está portanto intimamente relacionado à gerência de configurações. Uma configuração válida de um objeto composto deve, além de selecionar representações para os sub-objetos, atribuir valores aos parâmetros destes sub-objetos.

O mecanismo de parametrização, alem da flexibilidade de modelagem oferecida ao projetista, permite uma grande economia de representações de objetos no banco de dados de projeto. Diversas alternativas de projeto, que eventualmente difeririam entre si apenas pelo valor de determinados atributos, podem ser representadas como uma única alternativa com um parâmetro variável. Por outro lado, a parametrização exige um mecanismo de configurações mais complexo, já que valores de parâmetros devem ser armazenados junto com as descrições de configurações.

Determinadas linguagens de descrição de hardware, como CASCADE [24] e VHDL [4], permitem a descrição de objetos compostos com estrutura regular parametrizável, i.e., onde um certo tipo de sub-objeto é replicado um número variável de vezes, controlado por um parâmetro, seguindo um determinado padrão. Este recurso de linguagem é bastante conveniente para o projetista do ponto de vista da modelagem, que fica bem mais concisa e inteligível. Do ponto de vista do modelo de dados, no entanto, este mecanismo cria uma séria dificuldade na representação do objeto composto, que não pode ser exata pois não se conhece a priori sua estrutura.

## 4.3 Relações entre objetos

Em [16] é feita uma análise sistemática das relações envolvendo objetos num modelo de dados para aplicações de projeto de sistemas digitais. São apresentadas como relações de primeira ordem:

- composição (já analisada na seção 4.1.1)
- histórico de versões (já analisado na seção 4.2.1)
- configuração (já analisada na seção 4.2.2)
- equivalência

Uma relação de equivalência é estabelecida entre duas representações de um sistema que apresentam comportamento idêntico. Dentro da terminologia adotada neste trabalho, está claro que tal tipo de relação deve ser estabelecida entre duas vistas de um objeto que correspondam à mesma alternativa de projeto. De forma mais precisa, poderia se fixar a relação entre duas versões, uma de cada uma destas vistas, embora possa se pensar como aceitável que a equivalência valha para toda e qualquer versão de cada uma das vistas.

A relação de equivalência pode ser gerada automaticamente pelo ambiente, sempre que uma ferramenta de síntese automática gerar uma representação de um objeto a partir de outra representação do mesmo objeto, ou que uma ferramenta de verificação formal estabelecer a equivalência funcional entre duas representações. No caso de uma representação gerada manualmente, com auxílio de um editor gráfico ou de textos, compete ao usuário indicar ao sistema que outra representação é equivalente àquela.

Estas relações de primeira ordem estão presentes em qualquer tipo de aplicação. Outras relações de primeira ordem podem no entanto ser dependentes de aplicação, devendo ser estabelecidas pelo usuário. Exemplos já foram dados na seção 4.1.5, envolvendo objetos auxiliares quaisquer. O ambiente deve oferecer neste caso um mecanismo genérico para suportar estas relações. Em GARDEN [10], por exemplo, uma *correlação* pode envolver um número qualquer de representações de objetos diversos, segundo um critério definido pelo usuário. Em Oct [14], um *attachment* é um *link* dirigido entre duas representações de objetos, e um objeto pode estar relacionado por *attachments* a um número qualquer de outros objetos.

Em [16] são discutidas também relações de segunda ordem, tais como versões de equivalências e configurações de equivalências, mas a utilidade prática de tais relações é duvidosa.

## 4.4 Restrições de integridade

Ao longo de um processo de projeto, diversas restrições de integridade devem ser mantidas, relacionando diferentes objetos e atributos, de modo que o projeto como um todo se encontre sempre em um estado consistente. Estas restrições são de diferentes naturezas:

- restrições referentes à representação de dados, tais como
  - sinais de interface conectados entre si numa composição de componentes devem ter tipos de dados idênticos (ou compatíveis)
  - atributos devem ter seus valores dentro de limites válidos
  - polígonos em diferentes camadas de fabricação de circuitos integrados devem obedecer regras quanto a suas dimensões e distâncias relativas
- restrições referentes à gerencia de dados, tais como
  - uma referência válida (completa) numa configuração de um componente deve alcançar uma folha da estrutura de controle do objeto instanciado
  - uma relação de equivalência deve ser estabelecida entre versões (ou vistas) de uma mesma alternativa, e não de alternativas distintas
- restrições intrínsecas a modelos de dados, tais como
  - nenhum nodo na estrutura de controle de um objeto pode ficar sem um nodo pai, exceto o nodo raiz (restrição de composição)
  - uma representação de um objeto que é instanciada em outro objeto não pode ser removida do banco de dados (restrição de referência)
- restrições referentes à metodologia de projeto, tais como
  - uma representação não validada de um objeto (por exemplo não simulada ou não gerada por síntese automática) não pode ser referenciada em outro objeto
  - determinada ferramenta de projeto não pode ser aplicada sobre uma representação de um objeto se determinados atributos desta representação não tiverem valores dentro de limites especificados

Estas diferentes categorias de restrições são verificadas normalmente em diferentes instâncias do ambiente. Assim, restrições relativas à representação de dados são verificadas pelas próprias ferramentas de projeto, tais como compiladores, editores de esquemáticos e verificadores de regras de projeto. Restrições referentes à metodologia de projeto são verificadas por um gerenciador de metodologia, quando este existe, como por exemplo em [6,25,26]. Finalmente, restrições referentes à gerência de dados e as intrínsecas ao modelo de dados são verificadas pelo próprio sistema de gerenciamento de dados (SGD).

Poderia se imaginar uma abordagem não convencional nestas soluções. No caso de um modelo de dados com granularidade bastante fina, onde polígonos nas diversas camadas de fabricação de um circuito integrado fossem objetos do modelo, poderia se imaginar que a verificação das regras de projeto ficasse a cargo do SGD, o que poderia ser feito em *batch*, quando um *layout* fosse submetido ao banco de

dados para armazenamento, ou interativamente, se o editor de *layout* estivesse fortemente integrado ao ambiente e cada acesso aos dados que representam o *layout* fosse mapeado para um acesso ao banco de dados. Este tipo de verificação dispensaria o uso de ferramentas como DRC, mas supõe um SGD extraordinariamente eficiente, sem o que o tempo dispendido seria intolerável pelo *overhead* dos acessos ao banco de dados.

Certas restrições referentes à representação de dados podem ser verificadas pelo SGD, apesar de tudo, sem que se comprometa a eficiência do ambiente. Estas restrições poderiam ser especificadas na própria descrição do modelo de dados, sendo portanto genéricas para todos os projetos realizados sobre o ambiente, ou poderiam ser descritas pelo usuário, sendo específicas para determinado projeto. O sistema Fred [15], por exemplo, permite ao usuário associar asserções a vistas de um objeto, envolvendo valores de parâmetros do objeto. Em [27] é proposto um mecanismo de *event trigger* associado a objetos e atributos, que dispara automaticamente rotinas de verificação de integridade quando os valores destes atributos são modificados. Mecanismos semelhantes têm sido adotados para o controle de metodologias de projeto, permitindo uma ativação automática de ferramentas de projeto quando certas condições são verificadas ou restrições são violadas, como em [25].

# 5 Discussão de modelos

## 5.1 VHDL

VHDL [4] é uma linguagem de descrição de hardware desenvolvida sob um contrato do DoD e adotada como padrão pela IEEE. Muitos produtos comerciais suportando a linguagem estão disponíveis. Embora VHDL não seja um ambiente, mas uma linguagem que pode ser integrada em um ambiente, ela já apresenta um esquema de gerenciamento de dados. Este esquema pode ser estendido por ferramentas externas, mas impõe algumas restrições conceituais que não podem ser evitadas.

Objetos de projeto em VHDL são modelados como **Entidades de Projeto**, que são descritas através de uma **interface** e um ou mais **Corpos Arquiteturais**, correspondendo a diferentes representações para a Entidade. Três diferentes estilos de descrição podem ser utilizados dentro de um mesmo Corpo Arquitetural: **comportamental**, **dataflow** e **estrutural**. Neste último estilo, o Corpo é descrito como uma interconexão de **componentes**.

Componentes podem ser ligados a uma determinada Entidade através de uma especificação de configuração dentro do Corpo que os contém ou através de um **Corpo Configuracional** separado. Um Corpo Arquitetural pode ser **parcialmente ligado**, se apenas Entidades são selecionadas para seus componentes, **completamente ligado**, se também Corpos Arquiteturais destas Entidades são selecionados, ou **abertos**, se a ligação é feita separadamente através de um Corpo Configuracional.

Com relação aos requisitos discutidos na seção 4, VHDL apresenta três restrições principais:

- Corpos Arquiteturais podem corresponder a vistas, alternativas e versões. Embora um gerenciador de dados externo pudesse organizar Corpos Arquiteturais como alternativas compostas de vistas (ou vice-versa) que por sua vez são compostas por versões, este esquema não seria utilizado pelo mecanismo de configuração existente na linguagem, onde para cada componente o usuário deve selecionar uma Entidade e somente um Corpo Arquitetural.
- vistos como alternativas ou vistas, Corpos Arquiteturais não podem refletir uma hierarquia de decisões de projeto, novamente porque esta hierarquia, embora pudesse ser representável através de um gerenciador externo, não seria utilizada pelo mecanismo de configuração da linguagem.
- VHDL não oferece um esquema de herança de portas. Todos os Corpos Arquiteturais de uma Entidade têm exatamente os mesmos sinais de interface. Portas podem no entanto ser deixadas não conectadas quando a Entidade tem mais portas do que o componente que a instancia.

O histórico de versões, representado como uma seqüência, uma árvore ou um grafo, poderia ser facilmente implementado por um gerenciador externo à linguagem.

VHDL oferece recursos adicionais de modelagem que são úteis num ambiente de projeto. Sinais podem ser de tipos definidos pelo usuário por um mecanismo de tipos abstratos de dados. Quando dois componentes são conectados entre si, os sinais conectados podem ser de tipos diferentes, desde que o usuário tenha especificado, junto aos tipos de dados, as operações que permitem a conversão entre eles. VHDL permite também que atribuições simultâneas a um sinal de interface sejam feitas de dentro de componentes distintos. Para isto ser possível, o usuário deve ter especificado *funções de resolução*, que decidem como resolver o conflito entre as atribuições. Todas estas facilidades (tipos abstratos de dados e funções de conversão e de resolução) podem ser definidas dentro de **packages**, objetos auxiliares que podem ser associados a Entidades ou a Corpos Arquiteturais destas.

Em [13,28,29,30,31] são encontrados exemplos de ambientes baseados em VHDL. Em [13] o modelo de dados intrínseco de VHDL é estendido com diversos conceitos, dos quais dois merecem ser citados:

- "classes" são agregados de Entidades funcionalmente semelhantes mas que apresentam interfaces distintas
- restrições de integridade podem ser utilizadas para controlar o processo de configuração (ver subseção 4.2.2)

## 5.2 AMPLO

AMPLO [11] é um ambiente integrado de projeto em desenvolvimento na Universidade Federal do Rio Grande do Sul, baseado em um modelo uniforme para representação e gerência de dados.

Todo sistema é representado como uma **Agência**. Cada Agência pode ter várias **Alternativas** de projeto, que correspondem a diferentes definições para a interface da Agência. A cada Alternativa podem ser associadas várias **Versões** de projeto. Todas as Versões de uma Alternativa compartilham a mesma definição da interface.

Versões podem ser **primitivas** ou **compostas**. AMPLO exige a atribuição de um determinado nível de abstração a cada Versão primitiva (um nível é definido por uma linguagem de descrição de hardware). Construções de diferentes linguagens não podem ser utilizadas dentro de uma mesma Versão primitiva. Versões compostas são redes de componentes que são ocorrências de outras Agências. Embora a separação explícita entre Versões primitivas e compostas seja restritiva do ponto de vista de modelagem, ela apresenta vantagens para o processo de integração de ferramentas e para o controle de metodologias de projeto, conforme analisado em [12].

Descrições compostas podem conter ocorrências de Alternativas ou de Versões de outras Agências. No primeiro caso, a descrição tem uma **Configuração Dinâmica**, e Versões devem ser selecionadas para seus componentes antes que ela possa ser usada por alguma ferramenta de projeto. No segundo caso, a descrição tem uma **Configuração Estática**.

Num projeto *top-down*, Alternativas podem ficar temporariamente sem Versões. Uma Agência, e implicitamente sua primeira Alternativa, pode ser criada dentro da descrição composta que faz a primeira referência a esta Agência - Alternativa.

O modelo de dados prevê um conjunto fixo de atributos associados aos sinais de interface: um tipo de dados, um sentido (entrada, saída, bidirecional), uma largura em bits e uma posição geométrica ao longo da envoltória da Agência (tendo em vista que descrições de Agências, tanto primitivas como compostas, podem ser feitas através de editores gráficos). A fim de permitir a interconexão de sinais de Agências descritas em níveis de abstração diversos, é definida uma compatibilidade entre tipos de dados de linguagens distintas.

O modelo de dados do AMPLO apresenta as seguintes restrições:

- Versões (de AMPLO) servem para representar tanto alternativas como vistas e versões. Não há nenhuma maneira de se distingüir entre estas situações, pois todas as Versões de uma dada Alternativa são organizadas linearmente sob esta Alternativa.
- Como decorrência do item anterior, AMPLO não permite a representação da hierarquia de decisões de projeto.
- AMPLO representa o histórico de Versões de uma forma linear sob cada Alternativa, e não permite que se represente o histórico em cada um dos níveis de abstração.
- AMPLO não suporta configurações abertas. Componentes devem sempre ser associados ao menos a uma Alternativa de uma Agência, e a interface do componente deve ser idêntica à interface da Alternativa escolhida.
- AMPLO não prevê o armazenamento no banco de dados de *descrições de configurações*.

A decisão de assumir que todas as Versões sob uma mesma Alternativa apresentem a mesma interface foi tomada para simplificar o mecanismo de configuração, pois, como um componente tem a mesma interface que a Alternativa que ele instancia, qualquer Versão desta Alternativa pode ser ligada a este componente sem que seja necessária qualquer verificação de consistência entre a interface do componente e a interface da Versão ligada.

AMPLO oferece uma facilidade de gerenciamento não comumente encontrada em outros sistemas, que é a possibilidade de se dar um nome comum a objetos com diferentes definições de interface. Isto evita a explosão de nomes de objetos devida a alterações de projeto que afetam a definição da interface. Do ponto de vista do mecanismo de configuração esta facilidade não traz nenhum benefício, pois a definição da interface está inteiramente contida nas Alternativas, sendo impossíveis referências apenas a Agências.

## 5.3 Oct

A Figura 3 mostra os objetos básicos manipulados pelo gerenciador de dados Oct [14], desenvolvido pela Universidade de Berkeley. **Células** podem conter qualquer número de **Views**. Não há um conjunto pré-definido de tipos de Views. Cada View tem um número qualquer de **Facetas**. Uma Faceta especial, denominada **Contents Facet**, contém a descrição completa da estrutura e geometria internas da View, além da definição de sua interface básica. As restantes Facetas de uma View, denominadas **Interface Facets**, herdam esta interface básica e acrescentam atributos adicionais, de modo a implementar *protection frames*, que definem informações visíveis externamente à View e dedicadas a ferramentas especializadas. Exemplos são a definição geométrica da envoltória da Célula, para um editor de esquemáticos, regiões de roteamento (tanto ao longo da interface como regiões de transparência sobre a Célula) para um roteador, e a geometria que precisa ser verificada contra as Células vizinhas, para um DRC. Cada Faceta pode ter várias **Versões**. Componentes em descrições compostas fazem referência a Views de Células.

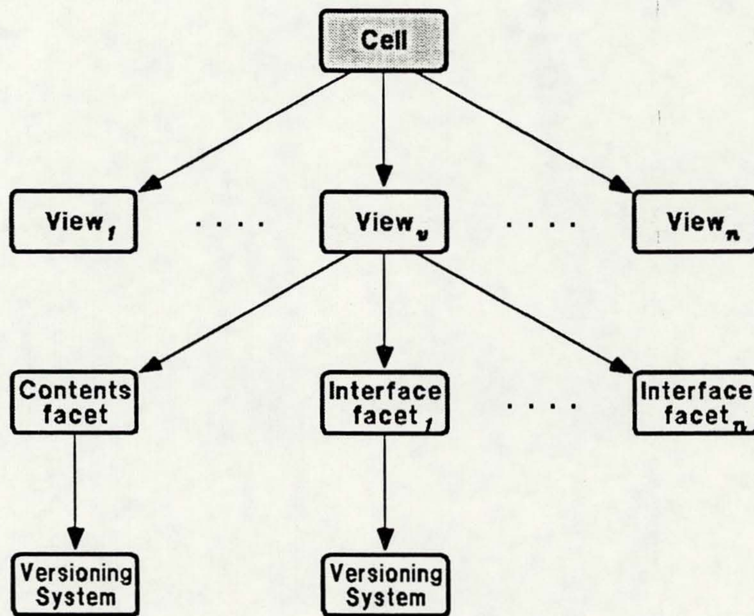


Figura 3: O modelo de dados do gerenciador Oct

O modelo de dados Oct oferece um pequeno conjunto fixo de objetos básicos a partir dos quais pode-se descrever a estrutura e geometria das Facetas: *boxes*, polígonos, *labels*, círculos, caminhos (linhas com vários pontos), camadas de *layout*. O ambiente é claramente dedicado à integração de ferramentas para os níveis infe-

riores de abstração, e em particular para os eixos estrutural e geométrico de representação.

Relações genéricas entre objetos são implementadas através de **Attachments**, que são *links* dirigidos entre dois objetos. Um objeto pode ser ligado a um número arbitrário de outros objetos e pode ter um número qualquer de objetos ligados a ele.

Este modelo de dados básico foi estendido em [23] para permitir um melhor gerenciamento da evolução do projeto e para suportar trabalho cooperativo. Células são agrupadas em **Workspaces**, que podem ser arbitrariamente organizados de acordo com critérios definidos pelo usuário, inclusive numa estrutura hierarquizada. Uma **Facet-family** é uma coleção de Facetas com mesmos objetos Célula, View e tipo de Faceta, mas com distintas Versões. Uma **Configuração** é um conjunto de Facetas em um Workspace (embora a mesma denominação, este conceito não é idêntico ao de *configuração* dado na seção 4) e tem duas propriedades principais. Em primeiro lugar, duas Versões de uma mesma *Facet-family* não podem aparecer numa mesma Configuração. Em segundo lugar, uma Configuração deve conter uma Versão para cada *Facet-family* do Workspace. Configurações portanto implementam um mecanismo de "versionamento" de todas as Células contidas em um Workspace. Uma Configuração é selecionada como a correntemente ativa. Os objetos desta Configuração são os únicos que podem ser modificados. Quando uma ferramenta faz referência a uma View de uma Célula em um Workspace, a *Interface Facet* é escolhida de acordo com a aplicação, enquanto a Versão é automaticamente selecionada de acordo com a Configuração ativa. Uma **Época** é o intervalo de tempo no qual uma dada Configuração é a ativa.

Oct apresenta as seguintes restrições em relação aos requisitos previamente discutidos:

- Views e Versões em Oct correspondem a vistas e versões, respectivamente, mas Oct não suporta alternativas de projeto, e portanto não permite que se represente a hierarquia de decisões de projeto
- Oct não provê um mecanismo que permita a herança de portas através de Views descrevendo uma mesma Célula em níveis de abstração diferentes
- Oct não suporta configurações abertas
- Oct suporta configurações dinâmicas de uma maneira restrita, pois a Configuração de Oct seleciona automaticamente Versões para todas as tuplas {*Célula, View, Faceta*} segundo um critério único e pré-estabelecido (a Época), não sendo possível ao usuário selecionar Versões, para diferentes tuplas de um mesmo Workspace, sob diferentes critérios.

Oct apresenta duas características que o distingüem dos demais modelos. Em primeiro lugar, ele tem um mecanismo de Facetas explicitamente dedicado a especialização de interfaces de acordo com a ferramenta a ser aplicada sobre a descrição.

Em segundo lugar, o conceito de Configuração permite simultaneamente uma representação (embora linear) da evolução do projeto (Configurações se sucedem em Épocas) e a seleção de versões, embora esta se dê segundo um critério restrito, conforme já comentado.

Oct oferece dois mecanismos para monitorar modificações em Facetas. **TimeStamps** marcam o momento em que foi feita a última operação básica de cada tipo (criação, remoção, *attachment*) na Faceta, assim como o momento em que foi feita a última alteração na interface da Faceta. **ChangeLists** contêm o registro de todas as operações que foram efetuadas sobre objetos básicos em uma Faceta desde a criação da ChangeList. Este mecanismo é utilizado para implementar uma comunicação entre ferramentas sendo executadas concorrentemente no ambiente, conforme explicado na seção 6.

## 5.4 DAMASCUS

A Figura 4 mostra a estrutura de controle de um objeto de projeto no sistema DAMASCUS [9], desenvolvido na Universidade de Karlsruhe. Esta hierarquia contém **Representações** (que correspondem a vistas), **Alternativas**, **Revisões** e **Estágios de Projeto**, estes dois últimos correspondendo a versões. Alternativas sob uma mesma Representação são em geral independentes umas das outras, mas o usuário pode estabelecer relações *derivada-de* entre elas. Revisões de uma mesma Alternativa estão organizadas na forma de um grafo acíclico, que implementa a relação *é-revisão-de*. Cada Revisão pode evoluir ao longo do tempo, originando uma seqüência linear de Estágios de Projeto, onde estão de fato armazenados os dados de projeto.

A interface de um objeto é definida nos níveis de Alternativas e Revisões. A interface de uma Alternativa é herdada por todas suas Revisões, enquanto cada Revisão pode adicionar novos atributos à interface. Como a interface é definida nestes níveis, um objeto não pode ser referenciado no nível de Representação.

Relações de equivalência podem ser estabelecidas entre Alternativas ou entre Revisões de diferentes Representações. Esta relação cria uma dependência dirigida entre as versões. Se a versão independente de uma tal relação for modificada, então a versão dela dependente perderá a validade.

Uma **Configuração** é definida como uma seleção de uma Alternativa para cada Representação de um objeto (note-se que esta denominação não corresponde ao conceito de *configuração* definido na seção 4). As Alternativas selecionadas para uma mesma Configuração devem ser necessariamente equivalentes.

O mecanismo de herança de interface de DAMASCUS, combinado com a coleção de Alternativas sob uma mesma Representação, apresenta uma vantagem mas duas desvantagens:

- Sob cada Representação é possível colecionar apenas as Alternativas que correspondam a decisões de projeto relevantes naquele nível de abstração.

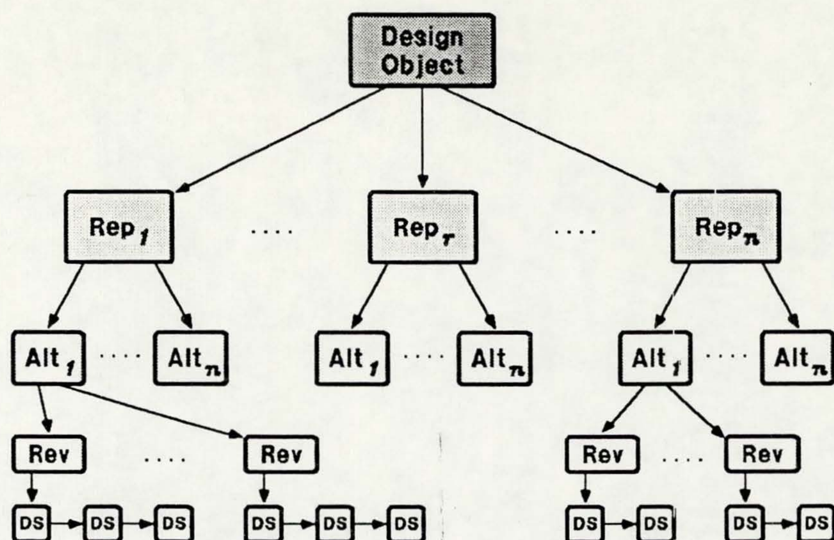


Figura 4: Modelo de dados do sistema DAMASCUS

- O sistema não consegue representar o fato de que, para uma mesma Alternativa, cada Representação compartilha um conjunto comum de portas e apresenta algumas portas adicionais. As portas comuns às várias Representações estarão duplicadas nas Alternativas equivalentes sob as várias Representações.
- A hierarquia de decisões de projeto, refletida seja pelas Representações, seja pelas Alternativas, não é representada. Uma referência válida contém apenas uma Alternativa, e não uma hierarquia de Alternativas como seria desejável. Relações *derivada-de* entre as Representações minoram esta deficiência.

DAMASCUS tem como característica principal ser um dos únicos modelos que contempla as três dimensões da evolução do projeto (alternativas, vistas e versões). O sistema inclusive tem um esquema mais refinado de representação de versões, já que, para cada nodo do grafo de Revisões, pode ser representada uma seqüência de Estágios de Projeto. Pode-se portanto associar modificações mais substanciais (mas que não caracterizem Alternativas, que devem corresponder a decisões de projeto importantes, como escolher entre uma abordagem "standard-cell" e uma "gate-array") a Revisões, enquanto pequenas correções ou aperfeiçoamentos serão representados como Estágios de Projeto.

## 5.5 GARDEN

O núcleo do modelo de dados do ambiente GARDEN [10,32], em desenvolvimento no Centro Científico Rio da IBM, é mostrado na Figura 5. Sua estrutura básica oferece uma hierarquia para a representação e gerência de objetos complexos, composta pelos conceitos **Design**, **ViewGroup**, **View**, **Modificações** e **Iterações**, estes dois

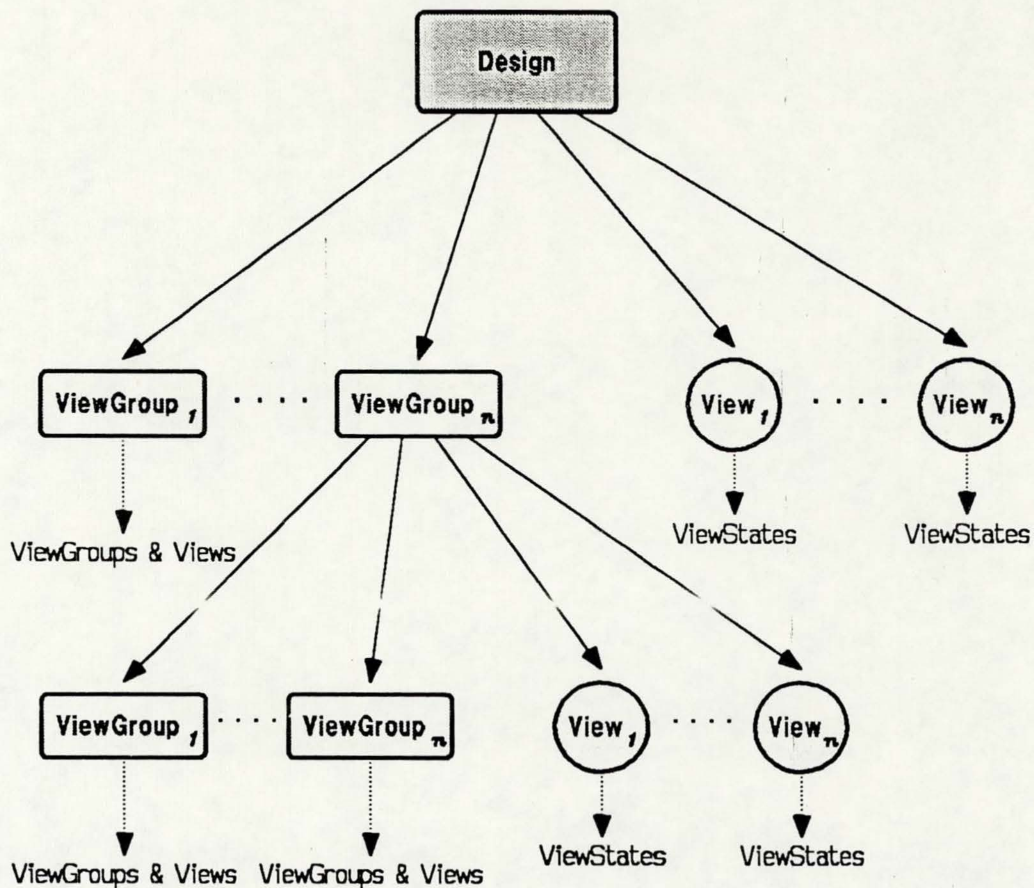


Figura 5: Objetos básicos do modelo de dados do GARDEN

últimos formando **ViewStates**. ViewGroups podem conter outros ViewGroups ou Views, podendo ser criada assim uma representação hierárquica com profundidade qualquer.

Views podem ser de três tipos: **HDL**, **Layout** e **MMHD** ("Mixed-Mode Hierarchical Description"). Todos os tipos de Views podem conter componentes, que fazem referência a outros Designs, mas apenas Views MMHD representam as interconexões entre sinais de interface dos componentes. A referência pode ser feita a qualquer nível da hierarquia de descrição destes outros Designs. É possivel fazer-se uma referência a um ViewState particular de uma View ou uma referência genérica ao ViewState mais recente.

Conforme ilustrado na Figura 6, Modificações são descrições alternativas para uma determinada View, enquanto Iterações são refinamentos sucessivos de uma mesma Modificação. Na notação $i.j$, $i$ identifica a Modificação, enquanto $j$ identifica a Iteração. Este controle mais refinado de versões é comparável ao esquema de

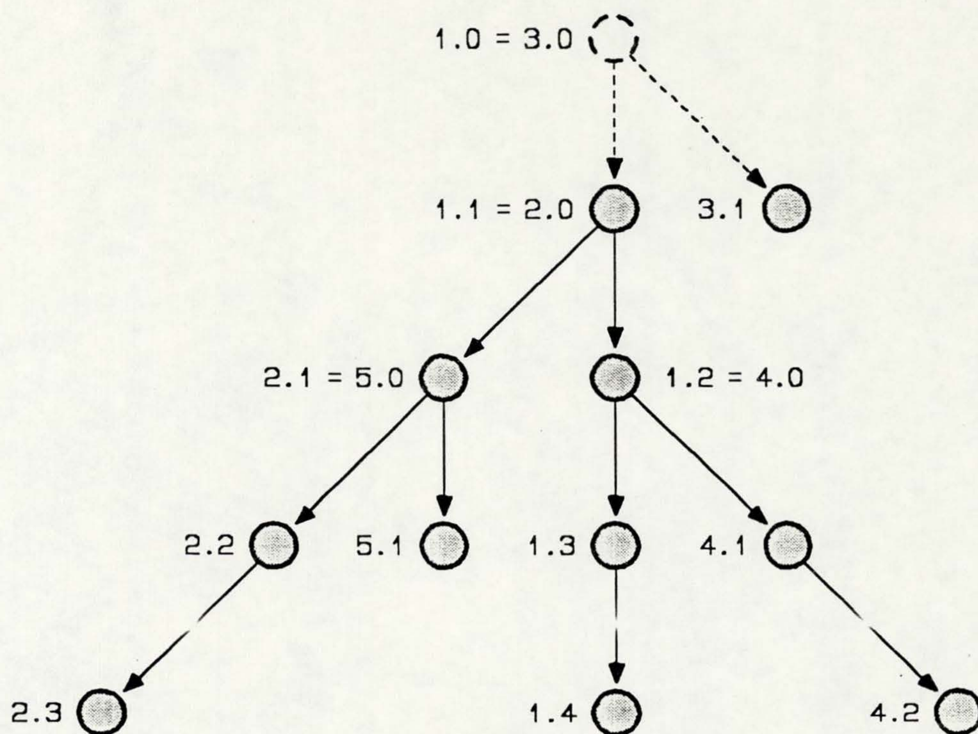


Figura 6: Modelo de dados do GARDEN – modificações e iterações

Revisões e Estágios de Projeto suportado em DAMASCUS, embora lá Revisões se relacionem na forma de um grafo e Estágios estejam organizados linearmente, enquanto em GARDEN Modificações e Iterações estão organizadas na forma de uma árvore.

Sinais de interface (**Ports**) podem ser definidos nos níveis de Design, ViewGroup e View, sendo herdados pelos níveis inferiores da hierarquia. Com isto é possível a representação de portas apenas nos níveis de abstração nos quais elas realmente têm um significado. Além disto, é possivel representar-se decisões de projeto alternativas que herdam portas comuns mas acrescentam portas específicas.

Além de portas, o usuário pode definir outros **Atributos** para um nodo $N_i$ qualquer da estrutura de controle. Estes Atributos são automaticamente herdados por todos os nodos descendentes de $N_i$. Atributos têm um nome e seu valor pode ser um *string* ou estar armazenado em um arquivo cujo conteúdo não é detalhado no modelo de dados.

**Correlações** podem ser estabelecidas entre dois ou mais objetos quaisquer, em qualquer nível de suas respectivas estruturas de controle. Correlações podem representar relações de equivalência entre ViewGroups e/ou Views e/ou ViewStates de um mesmo Design. Pode-se também representar o fato de que dois ou mais

objetos (p.ex. *DataPath* - $VG_i$ e *ControlUnit* - $VG_j$) correspondem a uma mesma decisão de projeto (p.ex. relógio de 4 fases), ou ainda que um objeto, p.ex. uma View MMHD $V_k$ de um Design $D_n$, composta de portas lógicas, foi obtida acrescentando-se informações de *timing* a outra View MMHD $V_l$ de $D_n$ (processo conhecido por *back-annotation*), informações estas extraídas de outra View Layout $V_m$ de $D_n$.

O modelo de dados do GARDEN é bastante flexível, já que ViewGroups têm uma semântica aberta. O critério de agrupamento de Views (ou outros ViewGroups) sob um ViewGroup pode ser definido pelo usuário e a hierarquia de ViewGroups pode ter uma profundidade qualquer, com o que pode-se criar na prática diferentes esquemas conceituais. Em [32], por exemplo, é sugerido de que forma poderiam ser mapeados para o GARDEN os modelos dos sistemas Oct e DAMASCUS e da linguagem VHDL. Em [17] é simulado um processo de projeto que resulta na criação de objetos GARDEN atendendo a um critério pelo qual cada ViewGroup é o agrupador de descrições que resultam de uma mesma decisão de projeto, conforme ilustrado na Figura 7. Estes ViewGroups estão organizados hierarquicamente, de um modo que reflete a seqüência de níveis de abstração pela qual passa o projeto. Neste esquema, portas vão sendo acrescentadas à interface de um objeto à medida que decisões de projeto são tomadas.

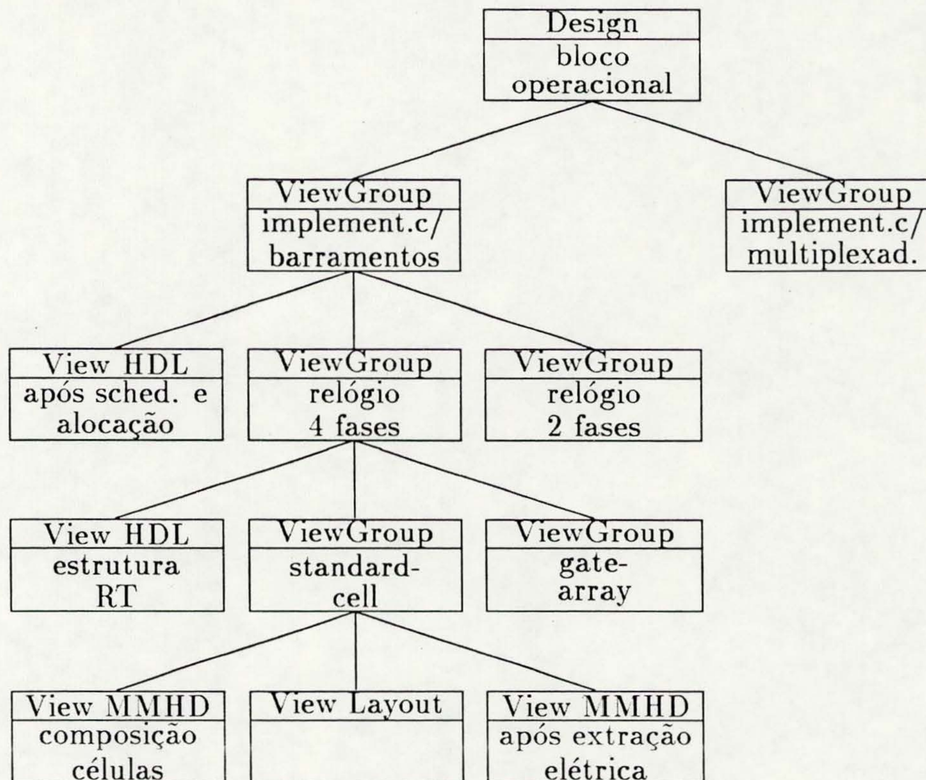


Figura 7: Exemplo de estrutura de controle implementada em GARDEN

Em relação aos requisitos discutidos na seção 4, pode-se dizer que:

- GARDEN suporta as três dimensões da evolução de projeto (vistas, alternativas e versões), sendo o controle de versões refinado na forma de uma árvore
- GARDEN permite a representação da hierarquia de decisões de projeto, na forma de uma hierarquia de ViewGroups, na qual um mecanismo de herança de portas permite que portas sejam associadas aos níveis de abstração nos quais elas são realmente acrescentadas ao projeto
- GARDEN suporta configurações estáticas e dinâmicas, sendo estas últimas bastante flexíveis, em função da profundidade variável da hierarquia de ViewGroups
- GARDEN não suporta configurações abertas

GARDEN apresenta ainda outras características adicionais, não comumente encontradas em outros modelos, referentes ao suporte à gerência de tecnologias de fabricação e ao controle, por um mecanismo de TimeStamps, de versões passadas.

O banco de dados do GARDEN é denominado de **Repositório**, e contém **Bibliotecas** e definições de **Processos**. Bibliotecas agrupam Designs segundo critérios estabelecidos pelos usuários. Um Processo consiste basicamente de um arquivo contendo informações sobre uma determinada tecnologia de fabricação. Uma Biblioteca pode ter um Processo associado a ela, caso em que este Processo será aquele no qual serão implementados todos os Designs existentes na Biblioteca. Um Processo $P_i$ pode ter outros Processos nele **incluíveis**. Estes Processos incluídos definem tecnologias compatíveis com a tecnologia de $P_i$. Uma ferramenta de configuração poderia restringir referências dentro de uma View de um Design $D_j$, contido numa Biblioteca com Processo $P_i$, a outros Designs contidos em Bibliotecas cujos Processos estão incluídos em $P_i$. Este mecanismo de compatibilidade de processos de fabricação é particularmente útil no projeto físico de sistemas discretos, onde devem ser considerados aspectos de tecnologias de fabricação de componentes, de encapsulamento e de montagem em placas.

Um Processo pode alternativamente ser associado a Designs, ViewGroups ou Views, de forma individual. Isto é possível quando o objeto (Design, ViewGroup ou View) estiver contido em outro ao qual não está associado um Processo. Pode-se por exemplo associar um Processo a um ViewGroup desde que o Design ao qual este ViewGroup pertença não tenha Processo associado diretamente a si ou à sua Biblioteca.

**TimeStamps** permitem o controle da evolução de diversos atributos no Repositório. Atributos que podem ser controlados desta forma têm armazenada no Repositório uma história de alterações, numa forma linear. Num tempo qualquer, uma referência a um objeto pode ser feita à sua versão corrente, com o que são buscados os valores de atributos para o tempo corrente (aqueles com maior valor de TimeStamp), ou a uma versão passada, com o que são buscados valores de atributos com TimeStamp mais próximo do (mas menor do que o) tempo no qual se quer fazer a referência. Pode-se recuperar configurações passadas de objetos, num

efeito bastante semelhante ao obtido pelas Configurações do gerenciador Oct (ver a sub-seção 5.3), já que referências em configurações dinâmicas são também atributos temporais. Também o nome do Processo associado a uma dada Biblioteca ou Design é um atributo temporal, permitindo-se com isto um mecanismo de configuração inteiramente controlado pelo tempo (lembrar relação entre Processo e configuração aludida no parágrafo anterior).

## 5.6 Outros modelos

Nesta subseção serão apresentadas brevemente características de outros ambientes que são distintas daquelas presentes nos modelos analisados em detalhe nesta seção 5. Eventualmente são repetidas informações já fornecidas anteriormente na seção 4.

### CWS

O ambiente Cadlab Workstation - CWS - [1], desenvolvido pelo Cadlab em Paderborn, implementa um modelo de dados genérico sobre o qual podem ser implementados diferentes esquemas conceituais, como por exemplo em [33], utilizando-se a linguagem TIDL [34]. O modelo permite diferentes graus de integração de ferramentas, conforme a modelagem dos objetos da aplicação como "objetos complexos não estruturados" (estruturação interna dos objetos é completamente desconhecida) ou "objetos complexos estruturados" (granularidade fina). Objetos estruturados são definidos a partir do instanciamento de objetos primitivos de um repertório pré-definido.

### PLAYOUT

O ambiente PLAYOUT [18] implementa uma **árvore de projeto** para o tratamento das alternativas de projeto. PLAYOUT se distingüe dos demais ambientes por implementar explicitamente o conceito de domínio, que corresponde aos eixos de projeto mencionados na seção 2. A cada **célula** podem ser associadas diferentes interfaces em diferentes **domínios** de projeto (estrutural, comportamental, "floorplan" e "masklayout"). Para uma interface num certo domínio podem ser geradas diversas **alternativas** de projeto, correspondendo a diferentes conteúdos. Para cada um destes conteúdos podem ser geradas, num outro domínio de projeto, novas interfaces, e para estas novamente vários conteúdos. Em verdade, cada nodo "interface" nesta árvore de projeto possui atributos específicos de um determinado domínio de projeto. Em cada domínio existem ainda várias **views**, que compartilham necessariamente a mesma interface para cada objeto.

### FACE

FACE [35] é um ambiente para o desenvolvimento de ambientes de CAD baseado no paradigma de orientação a objetos. O núcleo do ambiente oferece as estruturas de dados (baseadas em grafos e hipergrafos), algoritmos e ferramentas necessários ao desenvolvimento de novas ferramentas de projeto. A partir deste núcleo pode-se definir, usando-se um mecanismo de heranças múltiplas, os objetos e métodos primitivos nos diversos níveis de abstração, o modelos de dados para o ambiente de projeto e o mecanismo de controle de metodologia de projeto.

### EVE

O modelo oferecido pelo ambiente EVE [8] permite a especificação de relações (*operation bindings*) entre objetos primitivos estruturais e comportamentais, o que dá condições ao ambiente de suportar a síntese de alto nível incremental.

### SDE

O ambiente SDE [13] é baseado na linguagem VHDL e implementa uma extensão desta na declaração de Corpos Configuracionais que permite a especificação de restrições (de área, velocidade, potência, etc) a serem observadas pelas Entidades e Corpos Arquiteturais a serem selecionadas para componentes nas configurações.

### Fred

O sistema Fred [15] permite que sejam associados valores *default* de atributos a vistas de um objeto. Alternativamente, o usuário pode especificar uma função que calcula rapidamente um valor estimativo de um atributo quando este não está declarado na descrição da vista. Com isto o usuário pode descrever a função ou estrutura interna desta vista de forma incompleta. O sistema automaticamente procurará, para cada atributo cujo valor não foi fornecido, seu valor *default*, ou, na ausência deste, ativará a função que calcula o valor estimativo.

Fred permite ainda a associação de asserções a vistas de um objeto, envolvendo valores de parâmetros deste, com o que podem ser verificadas automaticamente pelo ambiente restrições de integridade relativas a estas vistas.

### Kemper e Wilkes

Kemper e Wilkes [36] propõem uma hierarquia de "versões" de profundidade qualquer, com herança de propriedades para os nodos descendentes. Em cada nível da hierarquia, versões podem ser classificadas em **partições** de acordo com um critério organizacional qualquer, e podem ser relacionadas por grafos que representam o histórico de sua evolução. Este esquema oferece uma flexibilidade semelhante à do ambiente GARDEN, embora com menos recursos.

# 6 Implicações dos modelos de representação e gerência de dados em outros aspectos de ambientes de projeto

O modelo de dados está também bastante relacionado ao controle de metodologias de projeto, aos mecanismos de comunicação entre ferramentas, à interface com o usuário e a outros aspectos de gerência de dados, tais como o modelo de cooperação, o mecanismo de transações e a hierarquização do banco de dados. A discussão a seguir objetiva unicamente ilustrar estes relacionamentos, sem procurar sistematizá-los. Ela também não visa discutir genericamente estes outros aspectos de um ambiente de projeto, nem apresentar soluções para estes problemas.

## 6.1 Controle de metodologias de projeto

Ferramentas de controle de metodologia de projeto [6,25,26] têm por objetivo orientar ou forçar o usuário na execução de uma determinada seqüência de ferramentas de projeto. Pode-se por exemplo decidir qual a próxima ferramenta a ser executada em função

- de condições informadas pelo usuário (p.ex. resultado de simulação adequado ou não adequado), que dificilmente poderiam ser derivadas automaticamente pelo ambiente
- de condições calculadas pelo ambiente e dependentes dos valores de determinados atributos
- de restrições de integridade especificadas pelo usuário e violadas como resultado da execução anterior de outra ferramenta
- de seqüências obrigatórias de ferramentas (p.ex. após a geração de um *layout* por qualquer método devem se seguir necessariamente uma extração de parâmetros elétricos e uma verificação de regras de projeto)
- de modificações em determinadas representações de objetos (p.ex. se uma vista RT é modificada deve ser feita obrigatoriamente uma nova síntese lógica para criação de uma vista lógica equivalente)

Os exemplos acima demonstram que o controle da metodologia de projeto é exercido em função do estado de objetos ou de valores de seus atributos. Este controle depende portanto do modelo de dados: o ambiente não pode testar condições sobre informações não disponíveis no modelo (embora possa eventualmente ativar automaticamente ferramentas de avaliação que testem condições referentes a informações internas a determinadas representações de objetos, que não aparecem no nível do modelo em função da granularidade deste).

## 6.2 Outros aspectos de gerência de dados

A gerência de dados não está restrita ao controle de versões e configurações, embora estes aspectos tenham sido enfatizados neste relatório tendo em vista que fazem parte do modelo de dados. Na seção 3 já foram introduzidos os aspectos de cooperação entre projetistas e de distribuição de dados numa rede de estações e servidores como requisitos essenciais de um ambiente de projeto.

Um modelo de cooperação envolve várias questões, entre as quais um mecanismo de transações longas e a organização do banco de dados em áreas com diferentes direitos de acesso para diferentes usuários.

Transações de projeto são tipicamente de longa duração. Diferentes mecanismos de transações para bancos de dados de projeto têm sido propostos (p.ex. [37,38,39]). Um objeto requisitado em uma transação não pode ficar completamente bloqueado para acesso por outros usuários em outras transações. O bloqueio além disto é altamente dependente do modelo de dados, já que os objetos são complexos, compostos por sub-objetos (de forma hierárquica com uma profundidade qualquer) e possuindo diversas representações. Ações de requisição, bloqueio e liberação de objetos devem portanto tratar de maneira adequada e diferenciada objetos e seus sub-objetos, assim como as várias representações. É imaginável que o tratamento dispensado às vistas não seja o mesmo dispensado às versões, por exemplo. O mecanismo de transações está também muito associado ao histórico de versões: como se administra a criação de versões para um mesmo objeto em transações distintas mas concorrentes?

O banco de dados deve ser organizado em alguma forma hierárquica que contemple áreas privativas para os usuários e áreas compartilhadas, implementando por exemplo uma única área pública e várias áreas restritas a determinadas equipes de projetistas (p.ex. [37,39,40]). Esta estruturação do banco de dados deve estar associada ao mecanismo de transações, já que os usuários devem, em uma transação, ter visibilidade a objetos limitada pelas restrições de acesso impostas pelo ambiente. As questões relativas ao tratamento diferenciado das dimensões da evolução do projeto e de objetos e sub-objetos devem portanto ser consideradas em função dos diversos níveis da hierarquia do banco de dados e suas diferentes propriedades.

Em [22], por exemplo, é descrito um modelo de cooperação onde são associados estados às versões. Os estados restringem os níveis da hierarquia de banco de dados onde podem estar armazenadas as versões. Além disto, o estado de uma versão composta por sub-objetos restringe não só o tipo de configuração possível para estes sub-objetos como também os estados das versões selecionáveis numa destas configurações, da maneira descrita a seguir.

Uma versão pode estar *em progresso* (é visível apenas pelo seu usuário proprietário), *estável* (não pode mais ser alterada, mas pode conter ainda configurações dinâmicas), *congelada* (todas as referências abertas devem ser resolvidas, i.e. só pode conter configurações estáticas), ou *liberada* (não pode mais ser removida). Além disto, versões liberadas só podem referenciar outras versões liberadas nas

configurações de seus sub-objetos (outras restrições se aplicam a versões em outros estados).

A estruturação do banco de dados de projeto em uma hierarquia apresenta também problemas adicionais, como o já mencionado na subseção 4.2.3. Se uma nova versão de um objeto $X$ torna-se visível num certo nível do banco de dados num tempo $T_i$ e $X$ já era referenciado numa configuração num objeto $Y$ que sempre seleciona a versão mais recente para seus componentes, então a descrição de $Y$ obtida num tempo $T_j < T_i$ será diferente daquela obtida em $T_i$, o que poderia levar o projetista a interpretar erradamente, por exemplo, resultados de simulação gerados em $T_j$ para $Y$, mas analisados no tempo atual.

A distribuição dos dados em uma rede (ver p.ex. [37,41]) está obviamente relacionada às questões do modelo de transações e da estruturação do banco de dados. A requisição de um objeto, armazenado em um servidor, para uma dada estação deve acarretar a cópia de todas as suas representações e as de seus componentes? Ou é mais eficiente buscar apenas determinadas representações? Neste caso, qual critério adotar? Problemas semelhantes são colocados no momento da liberação de um objeto de uma estação para um servidor.

## 6.3 Interface com o usuário

No momento em que o banco de dados contém uma infinidade de objetos e representações destes, é importante que o usuário possa visualizar os objetos disponíveis, assim como as relações (de composição, de estruturação em dimensões da evolução do projeto, de equivalência) de maneira adequada. Interfaces gráficas para consulta e navegação no banco de dados têm sido propostas em todos os ambientes apresentados na literatura (ver p.ex. [42,43]). Estas ferramentas, denominadas genericamente de *browsers*, oferecem recursos, entre outros, para

- visualizar grupos de objetos ou representações destes segundo diferentes critérios, sempre a partir de um objeto "corrente",
- apontar um novo objeto corrente na tela, alterando assim os demais objetos e representações apresentados,
- percorrer as diferentes relações (de composição, de estruturação, de equivalência).

Obviamente os mecanismos de consulta e navegação devem ser adequados ao modelo de dados implementado pelo ambiente. Pode-se no entanto visualizar duas estratégias para a implementação de um *browser*:

- desenvolver um *browser* específico, dedicado para um certo modelo de dados, ou
- desenvolver um *browser* genérico, configurável para diferentes modelos de dados, como em [44], por exemplo.

O *browser* pode também apoiar o processo de configuração, já que neste é necessária a seleção de representações para instâncias de objetos utilizadas em descrições compostas de outros objetos, e o *browser* permite justamente a visualização das representações disponíveis para um objeto.

## 6.4 Comunicação entre ferramentas

Em ambientes multi-tarefa, é importante que seja oferecido um mecanismo que permita a ferramentas sendo executadas concorrentemente comunicarem entre si alterações efetuadas sobre objetos aos quais elas têm acesso simultâneo. Obviamente imagina-se que apenas uma destas ferramentas tenha acesso de escrita aos objetos compartilhados, enquanto as demais têm acesso de leitura. Este recurso é útil por exemplo em situações como:

- executar a verificação de regras de projeto concorrentemente com a edição do *layout*,
- permitir que valores de sinais gerados numa simulação sejam mostrados sobre a posição exata destes sinais no esquemático, cujo processo de visualização é controlado por uma ferramenta de edição (o processo de salientar no esquemático sinais nos quais ocorreram eventos significativos – numa simulação ou numa síntese – é conhecido como *highlight*)
- numa aplicação mais sofisticada, permitir alterações num esquemático concorrentemente com sua simulação, de modo que o simulador, a partir do tempo em que a modificação foi feita, passe automaticamente a tratar a nova descrição.

Dependendo da sua forma de implementação, este mecanismo pode estar intimamente relacionado ao modelo de dados. A comunicação de um evento sobre um objeto pode ser disparada automaticamente pelo ambiente para todas as ferramentas que estiverem lendo o objeto naquele dado momento, através de um *trigger* definido junto ao objeto no banco de dados. A implementação de um tal mecanismo depende obviamente da granularidade do modelo, já que apenas alterações sobre objetos representados no modelo poderão ser comunicadas.

Na interface SPI do ambiente do IMEC [45], ao contrário, a comunicação deve ser disparada explicitamente pela ferramenta que efetuou a modificação sobre um objeto. A especificação dos dados a serem comunicados através da SPI define implicitamente o modelo de dados da aplicação. SPI oferece recursos para *highlight*, seleção e navegação de objetos, para expansão de hierarquias (no caso de uma ferramenta trabalhar sobre descrições hierarquizadas e outra sobre descrições expandidas) e para configuração de descrições parametrizadas.

No sistema Oct [14] é oferecida ainda uma terceira solução, na qual as ferramentas que lêem os objetos é que devem tomar a iniciativa de detectar alterações sobre

eles. Elas o fazem acessando periodicamente *change lists*, que registram automaticamente todas as operações que foram executadas sobre um dado objeto desde que a *change list* foi criada.

## Agradecimentos

As idéias contidas neste relatório amadureceram após incontáveis e frutíferas discussões sobre os mais variados aspectos de ambientes de projeto com os colaboradores dos projetos GARDEN (A.H.V.de Lima, G.O.Annarumma, E.B.de la Quintana, P.Molinari Neto, R.C.B.Martins, R.Stern, L.Carneiro e A.Osso), em desenvolvimento no Centro Científico Rio da IBM Brasil, e AMPLO (L.G.Golendziner, C.Iochpe, C.M.D.S.Freitas, V.Boklis, K.Becker, M.Oliveira Neto e muitos outros), em desenvolvimento no Instituto de Informática e Curso de Pós-Graduação em Ciência da Computação da UFRGS.

As figuras 3, 4, 5 e 6, extraídas de [32], foram gentilmente cedidas por A.H.V.de Lima.

## Referências

[1] K. Gottheil et al. The CADLAB workstation CWS – an open, generic system for tool integration. In F.J. Rammig, editor, *IFIP Workshop on Tool Integration and Design Environments*, North-Holland, 1988.

[2] P. van der Wolf et al. Data management for VLSI design: conceptual modeling, tool integration, and user interface. In F.J. Rammig, editor, *IFIP Workshop on Tool Integration and Design Environments*, North-Holland, 1988.

[3] D. Gajski and R. Kuhn. Guest's editors introduction. *IEEE Computer*, December 1983.

[4] *IEEE Standard VHDL Language Reference Manual.* IEEE, New York, 1988.

[5] D. Rieu and G.T. Nguyen. Semantics of CAD objects for generalized databases. In *23rd Design Automation Conference*, ACM/IEEE, 1986.

[6] M.L. Bushnell and S.W. Director. VLSI CAD tool integration using the Ulysses environment. In *23rd Design Automation Conference*, ACM/IEEE, 1986.

[7] *Electronic Design Interchange Format Version 2 0 0.* EDIF Steering Committee, 1987.

[8] H. Afsarmanesh, E. Brotoatmodjo, K.J. Byeon, and A.C. Parker. The EVE VLSI Information Management Environment. In *International Conference on Computer Aided Design*, IEEE, 1989.

[9] J.A. Mulle, K.R. Dittrich, and A.M. Kotz. Design management support by advanced database facilities. In F.J. Rammig, editor, *IFIP Workshop on Tool Integration and Design Environments*, North-Holland, 1988.

[10] E.B. de la Quintana, G.O. Annarumma, and P. Molinari Neto. *GARDEN – the Design Data Interface.* Technical Report CCR-107, IBM Rio Scientific Center, Rio de Janeiro, 1990.

[11] L.G. Golendziner and F.R. Wagner. Modeling digital systems as complex objects. In *9th International Symposium on Computer Hardware Description Languages and their Applications*, IFIP, 1989.

[12] F.R. Wagner. *Integrating a VHDL Dialect into the AMPLO Design Framework.* Technical Report, CPGCC / UFRGS, Porto Alegre, 1991.

[13] M.J. Chung and S. Kim. An object-oriented VHDL design environment. In *27th Design Automation Conference*, ACM/IEEE, 1990.

[14] D.S. Harrison et al. Data management and graphics editing in the Berkeley Design Environment. In *International Conference on Computer Aided Design*, IEEE, 1986.

[15] W. Wolf. An object-oriented procedural database for VLSI chip planning. In *23rd Design Automation Conference*, ACM/IEEE, 1986.

[16] R.H. Katz et al. Design version management. *IEEE Design & Test*, February 1987.

[17] F.R. Wagner. *Mapeamento de um Processo de Projeto de Circuitos VLSI para o Modelo de Dados do Ambiente GARDEN*. Technical Report CCR-109, IBM Rio Scientific Center, Rio de Janeiro, 1990.

[18] E. Siepmann and G. Zimmermann. An object-oriented datamodel for the VLSI design system PLAYOUT. In *26th Design Automation Conference*, ACM/IEEE, 1989.

[19] L.-C. Liu, P.-C. Wu, and C.-H. Wu. Design data management in a CAD framework environment. In *27th Design Automation Conference*, ACM/IEEE, 1990.

[20] F.R. Wagner. Um ambiente integrado para a simulação de sistemas digitais. In *IV Simpósio Brasileiro de Concepção de Circuitos Integrados*, SBC, Rio de Janeiro, 1989.

[21] S. Kim and M.J. Chung. The constraint-driven design in an object-oriented VHDL CAD. In F.J. Rammig and R. Waxman, editors, *2nd IFIP International Workshop on Electronic Design Automation Frameworks*, North-Holland, 1991.

[22] A. Biliris. Database support for evolving design objects. In *26th Design Automation Conference*, ACM/IEEE, 1989.

[23] M. Silva, D. Gedye, R.H. Katz, and A.R. Newton. Protection and versioning for Oct. In *26th Design Automation Conference*, ACM/IEEE, 1989.

[24] D. Borrione and C. LeFaou. Overview of the CASCADE multi-level hardware description language and its mixed-mode simulation mechanisms. In *7th International Symposium on Computer Hardware Description Languages and their Applications*, IFIP, 1985.

[25] D.W. Knapp and A.C. Parker. A design utility manager: the ADAM planning engine. In *23rd Design Automation Conference*, ACM/IEEE, 1986.

[26] J. Daniell and S.W. Director. An object-oriented approach to CAD tool control within a design framework. In *26th Design Automation Conference*, ACM/IEEE, 1989.

[27] A.M. Kotz, K.R. Dittrich, and J.A. Mulle. Supporting semantic rules by a generalized event / trigger mechanism. In *International Conference on Extending Data Base Technology*, Venice, 1988.

[28] E. Marschner. VHDL design environment. *VLSI Systems Design*, September 1988.

[29] R.M. da Costa. Integrating VHDL. *High Performance Systems*, February 1989.

[30] L.F. Saunders. The IBM VHDL design system. In *24th Design Automation Conference*, ACM/IEEE, 1987.

[31] R.D. Acosta et al. The role of VHDL in the MCC CAD system. In *25th Design Automation Conference*, ACM/IEEE, 1988.

[32] F.R. Wagner, A.H.V. de Lima, G.O. Annarumma, E.B. de la Quintana, and P. Molinari Neto. Design version management in the GARDEN framework. Paper submitted to the 28th Design Automation Conference.

[33] B. Kleinjohann and E. Kupitz. Tight integration approach in a hardware synthesis system. In F.J. Rammig and R. Waxman, editors, *2nd IFIP International Workshop on Electronic Design Automation Frameworks*, North-Holland, 1991.

[34] K. Groening et al. From tool encapsulation to tool integration. In F.J. Rammig and R. Waxman, editors, *2nd IFIP International Workshop on Electronic Design Automation Frameworks*, North-Holland, 1991.

[35] W.D. Smith et al. FACE Core Environment: the model and its application in CAE/CAD tool development. In *26th Design Automation Conference*, ACM/IEEE, 1989.

[36] F. Kemper and W. Wilkes. Basic mechanisms to support versioning in the database component of a CAD framework. In F.J. Rammig and R. Waxman, editors, *2nd IFIP International Workshop on Electronic Design Automation Frameworks*, North-Holland, 1991.

[37] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. Transaction mechanism for engineering databases. In *International Conference on Very Large Data Bases*, Singapore, 1984.

[38] R. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Database Week - Engineering Design Applications*, IEEE, 1983.

[39] K. Dittrich. Controlled cooperation in engineering database systems. In *Data Engineering Conference*, IEEE, 1987.

[40] R.H. Katz and E. Chang. Managing change in a computer-aided design database. In *International Conference on Very Large Data Bases*, Brighton, 1987.

[41] R.H. Katz and T.J. Lehman. Database support for versions and alternatives of large design files. *IEEE Transactions on Software Engineering*, March 1984.

[42] D. Gedye and R.H. Katz. Browsing the chip design database. In *25th Design Automation Conference*, ACM/IEEE, 1988.

[43] N. Delisle and M. Schwartz. Neptune: a hypertext system for CAD applications. *SIGMOD Record*, June 1986.

[44] E. Siepmann. A data management interface as part of the framework of an integrated VLSI design system. In *International Conference on Computer Aided Design*, IEEE, 1989.

[45] L. Claesen et al. Open framework of interactive and communicating CAD tools. In F.J. Rammig, editor, *IFIP Workshop on Tool Integration and Design Environments*, North-Holland, 1988.