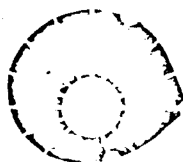
Um Gerenciador de Banco de Dados

Voltado para Projeto de Circuitos Integrados

Julho 1986



UNICAMP

UNIVERSIDADE ESTADUAL DE CAMPINAS

INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E CIÊNCIA DA COMPUTAÇÃO

CAMPINAS - SÃO PAULO

BRASIL

TÍTULO DA TESE

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. ANDRE M. DOLENC e aprovada pela Comissão Julgadora.

Campinas, de de 19

Prof.Dr. [signature]
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de Mestre em CIÊNCIA DA COMPUTAÇÃO

Universidade Estadual de Campinas - UNICAMP
Instituto de Matemática, Estatística e Ciência da Computação
IMECC - Departamento de Ciência da Computação

Autor: André M. Dolenc
Orientador: prof. Hans Liesenberg

Tese submetida à comissão de pós-graduação do IMECC/UNICAMP como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Campinas (SP), Brasil
8 de Julho de 1986

To Maria,
whom I love
and wish she where here.

SUMÁRIO

Os objetivos desta tese são:

1. explorar soluções para o problema de gerenciamento de projetos no contexto de projeto de circuitos integrados,

2. e propor um método de integração de programas de aplicação já existentes em torno de um mesmo banco de dados.

O sistema de banco de dados é dividido em camadas. A primeira camada, chamada DMS, é responsável pelo gerenciamento do projeto. As camadas seguintes podem ser gerenciadores de bancos de dados orientados para aplicações específicas (p.ex., manipulação de traçados). Propomos um gerenciador chamado OMS que facilitaria a integração de programas já existentes.

SUMMARY

The objective of this thesis is twofold:

1. to explore solutions to the problem of design management in the design of integrated circuits,

2. and to propose a method for integrating already existing application programs around a common database.

The database system is then divided in layers. The first layer, called DMS, is responsable for the design management. The following layers could be database managers oriented towards specific applications (such as the manipulation of layouts). We propose one such database manager called OMS which would ease the task of integrating already existing application programs.

CONTEÚDO

PÍTULO 1 INTRODUÇÃO

1.1 O PROJETO DE CIRCUITOS INTEGRADOS 1-1
1.2 PARALELO ENTRE BDC E UM BD ORIENTADO PARA PROJETO DE CIS 1-7
1.2.1 A Natureza Dos Dados E Seu Uso 1-8
1.2.2 Transações E Consistência 1-8
1.2.3 Acesso Concorrente 1-9
1.3 REQUISITOS ADICIONAIS 1-10
1.4 DISCUSSÃO E LEVANTAMENTO BIBLIOGRÁFICO 1-14

PÍTULO 2 DMS, UM GERENCIADOR DE PROJETOS

2.1 INTRODUÇÃO 2-1
2.2 MECANISMOS DE GERENCIAMENTO DE REPRESENTAÇÕES . . 2-2
2.2.1 Validação 2-3
2.2.2 Criação De Novas Versões 2-4
2.2.3 Hierarquia Das Representações 2-5
2.2.4 Relações, Propriedades E Relações De Afetação . 2-6
2.3 TABELAS OU ATRIBUTOS 2-8
2.4 SINÔNIMOS 2-9
2.5 SUPORTE A SISTEMAS DISTRIBUÍDOS 2-9
2.6 IMPLEMENTAÇÃO 2-10
2.6.1 Uso De Um BD Relacional 2-12
2.7 USO 2-13
2.7.1 Criação De Um Projeto 2-13
2.7.2 Configurando Um Projeto 2-14
2.7.3 Utilizando Um Projeto 2-15
2.7.4 Acesso A Representações De Entidades 2-16
2.7.5 Uso Do "path" 2-18
2.7.6 Uso Dos "Sets" 2-19

PÍTULO 3 SCHC: O COMPILADOR DE ESQUEMAS

3.1 INTRODUÇÃO 3-1
3.2 A LINGUAGEM DE DESCRIÇÃO DE ESQUEMAS 3-2
3.2.1 A Diretiva INCLUDE 3-2
3.2.2 Extensões Úteis Mas Não Implementadas 3-3
3.3 AS ROTINAS DE LEITURA/ESCRITA DE UMA ESTRUTURA DE DADOS 3-3
3.3.1 Uso 3-5
3.3.2 Implementação 3-8
3.3.2.1 Tratamento De Apontadores Para Registros 3-8
3.3.2.2 Tratamento De Apontadores Para Não Registros . . 3-10

.PÍTULO 4 OMS, UM GERENCIADOR DE ESTRUTURA DE DADOS

4.1 INTRODUÇÃO 4-1
4.2 FUNCIONAMENTO DO OMS 4-1
4.2.1 Geração Da Descrição Da Estrutura De Dados . . . 4-3
4.2.2 O "Mapper" 4-3
4.2.3 O RTOMS 4-5
4.3 RESTRIÇÕES 4-5

NCLUSÕES

RADECIMENTOS

ÊNDICE A EXEMPLO DE USO DO COMPILADOR DE ESQUEMAS

A.1 EXEMPLO 1 . . . . A-1
A.1.1 DEXAMPLE1.PAS A-2
A.1.2 DEXAMPLE1.INC A-3
A.1.3 SEXAMPLE1.PAS A-3
A.1.4 GEXAMPLE1.PAS A-5

BLIOGRAFIA

CAPÍTULO 1

INTRODUÇÃO

Esse trabalho ocupa uma "zona cinzenta" compreendida entre a atividade de projeto de circuitos integrados (CIs) auxiliado por computador e o projeto de bancos de dados (ou base de dados, BD). Estaremos, portanto, discutindo um BD não convencional.

Na tentativa de tornar o texto compreensível a ambas as partes, faremos uma breve exposição do que consiste o projeto de CIs, apresentaremos a terminologia usada no projeto de CIs, e faremos um paralelo entre os requisitos básicos necessários a um BD orientado para aplicações em projeto de CIs (BDCI) e um BD convencional (BDC).

1.1 O PROJETO DE CIRCUITOS INTEGRADOS

Várias metodologias de projeto podem ser adotadas [16,17][+]. No entanto, qualquer uma que seja adotada exige a interação do projetista com o computador. Logo, a forma pela qual o CI é concebido depende das ferramentas (programas de aplicação ou PAs) disponíveis em sua instalação.

[+] Existem inúmeros artigos sobre esse assunto. Não temos nenhum motivo especial em ter apontado esses dois em particular.

O objetivo final é obter a especificação geométrica de um conjunto de máscaras (traçado ou, em inglês, "layout") a partir de uma especificação funcional. As máscaras determinam as posições físicas ocupadas pelos elementos ativos do circuito (transistores, capacitores, resistores), suas características físicas e como estão interligados. As especificações funcional e geométrica são duas representações em níveis de abstração diferentes do mesmo circuito.

Existem representações intermediárias, e o número dessas depende da metodologia de projeto adotada. Tipicamente, as representações existentes são as especificações lógica e elétrica, além das outras duas já mencionadas.

O processo de descrever um CI num nível de abstração mais baixo a partir de um nível mais alto é o processo de síntese (p.ex., o processo de obter uma descrição lógica a partir de uma especificação funcional). Isso é análogo à atividade de programação, onde a partir de uma especificação procura-se obter um algoritmo escrito em uma linguagem de programação conveniente que, por sua vez, pode ser traduzido automaticamente para uma linguagem de mais baixo nível. (Para uma analogia mais completa veja [5].)

Infelizmente, nem sempre o projetista possui uma ferramenta que permita a transformação automática de uma representação de um CI em outra. Em consequência, a maior parte dos programas voltados para o projeto de CIs tem como função dar subsídios ao projetista na sua tarefa de validar uma representação, ou seja, auxiliar na verificação da consistência entre as representações.

Normalmente, esta validação é feita através de uma simulação, ou através de uma extração [*] seguida de uma comparação dos resultados obtidos com a representação do CI num nível de abstração mais alto. Por exemplo, a partir da especificação geométrica das máscaras é possível obter-se uma descrição elétrica equivalente. A descrição elétrica extraída pode então ser simulada e os resultados comparados com os resultados obtidos pela simulação da descrição elétrica original, usando-se em ambas as simulações os mesmos estímulos a fim de exercitar as duas representações supostamente equivalentes.

Além de traduzir uma representação em um nível de abstração numa representação em outro nível, o projetista costuma decompor o circuito descrito num dado nível em várias partes ou células. Novamente, temos uma analogia com programação onde a técnica de modularização é amplamente utilizada [+].

Da mesma forma que um procedimento, uma célula é composta de uma interface e uma descrição do seu funcionamento. Existem várias formas de se descrever uma célula, e entre elas temos

1. as linguagens comportamentais,

2. as linguagens estruturais (gráfica ou textual).

[*] Extração consiste no reconhecimento de componentes elementares e suas interações de um nível de abstração mais elevado a partir de uma representação de um nível de abstração mais baixo. Veja exemplo que se segue.

[+] Devemos salientar que para certas metodologias de projeto essa afirmação não é totalmente válida. No entanto, para esses casos podemos considerar o circuito todo como uma célula.

A tabela abaixo [9] lista as diversas possibilidades:

NÍVEL de REPRESENTAÇÃO \ DOMÍNIO	COMPORTAMENTO	ESTRUTURAL	FÍSICO
ARQUITETURA	especificações de performance	UCPs memórias controladores barramentos	partições físicas
ALGORÍTMICO	algoritmos (manipulação de estruturas dados)	modulos de "hardware" estruturas de dados	"clusters"
FUNCIONAL	operações transferências entre regis. mudanças de estados	ULAs, MUXs registradores microseq. microarmaz.	"floorplans"
LÓGICO	equações booleanas	portas flip-flops chaves	estimativas de células
ELÉTRICO	equações diferenciais	transistores capacitores resistores	estimativas de células traçados

De uma maneira geral, uma descrição de comportamento é composta de uma parte estática que descreve as operações a serem efetuadas sobre os sinais (dados), e uma parte dinâmica que descreve o sequenciamento e temporização das operações (controle). Há consenso de que uma descrição no domínio do comportamento está num nível de abstração mais alto, já que não há preocupação com a implementação.

Por outro lado, uma linguagem estrutural é usada para especificar como a célula será implementada numa dada representação. Observe na figura 1.1a a intenção do projetista de implementar a célula FFJK utilizando células do tipo AND e um flip-flop do tipo S-R, enquanto

que na figura 1.1b nada se pode concluir sobre sua estrutura. (O exemplo da figura abaixo foi extraído de [22].)

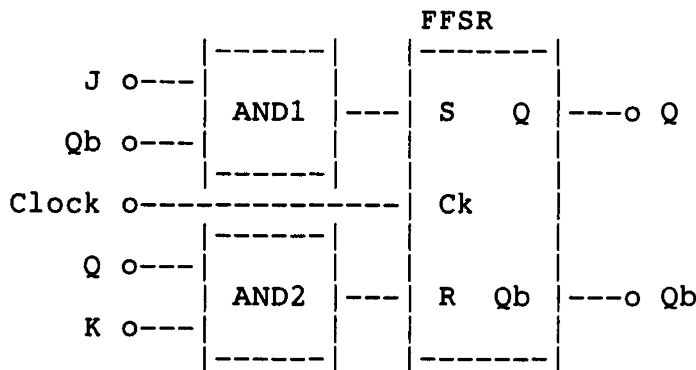


Figura 1.1a

Um flip-flop do tipo J-K chamada FFJK.

Jn	Kn	Qn+1
0	0	Qn
1	0	1
0	1	0
1	1	Qn'

Figura 1.1b

Uma descrição funcional de um FF do tipo J-K.

Na figura 1.1a, as células AND1 e AND2 são os componentes da célula FFJK e são instanciações de uma célula do tipo AND. Os rótulos J, K, Q, Qb, e Clock são chamados de terminais da célula FFJK. Os nós descrevem como os componentes estão interligados entre si através de seus terminais. O conjunto de todos os nós forma a lista de sinais ("netlist").

Esses elementos (componentes, terminais, nós) descrevem a topologia da célula. Associado a cada elemento há um conjunto de atributos que determinam o seu comportamento (ou melhor, que modelam o seu comportamento real) num dado nível de abstração. Por exemplo, a nível lógico é necessário especificar a direção em que um sinal pode fluir por um terminal, enquanto que a nível elétrico essa informação é sempre implícita devido à natureza do componente ou tipo do terminal.

Quando num dado nível uma célula não pode (ou não precisa) ser descrita a partir de outras ela é chamada de uma célula primitiva. Por exemplo, uma célula do tipo AND é uma primitiva para um simulador lógico, mas não para um simulador elétrico.

Além de células, o projetista deve manipular outros tipos de informações:

o Informações de controle

Cada programa de aplicação normalmente requer um conjunto de informações de controle que determinam como ele deve tratar a entrada. Esse conjunto pode ser bastante extenso e complexo e assumir as mais diversas formas.

o Formas de onda

O principal resultado de uma simulação é um conjunto de formas de onda. Uma forma de onda é uma representação gráfica do comportamento de uma grandeza física no tempo num certo ponto do circuito.

Por exemplo, durante uma simulação elétrica o projetista pode requerer a variação no tempo da tensão no terminal Clock da célula FFJK.

Além disso, numa simulação é necessário descrever na entrada as formas de onda que servirão de estímulo ao circuito.

o Descrição de tecnologias

Isso engloba principalmente regras de projeto e modelos para simulação. Cada processo de fabricação de CIs tem um conjunto de regras de projeto. Essas regras determinam, por exemplo, regras de espaçamento entre polígonos numa mesma máscara, larguras mínimas de canais de transistores, etc.

Os modelos são conjuntos de parâmetros que definem os modelos matemáticos das células primitivas de uma certa tecnologia para um simulador elétrico. Cada simulador elétrico utiliza-se de um conjunto de parâmetros diferente para cada tecnologia que suporta.

Essas informações normalmente são fornecidas pela "foundry" que irá fabricar o CI.

1.2 PARALELO ENTRE BDC E UM BD ORIENTADO PARA PROJETO DE CIS

A literatura demostra que se usarmos somente um BDC para desenvolver um sistema de PAC [+] orientado para projeto de CIs obteremos resultados desanimadores [1][2][7]. No entanto, em algumas situações um BDC pode ser útil [8].

Um BDC torna-se ineficiente quando colocado num ambiente de projeto de CIs devido às diferenças entre esse tipo de aplicação e aplicações

·] PAC é a abreviação de Projeto Assistido por Computador.

tradicionais de BD. Essas diferenças estão enumeradas abaixo.

1.2.1 A Natureza Dos Dados E Seu Uso

Num BD relacional, por exemplo, a unidade básica de informação é uma tupla ou um "registro". As operações sobre tuplas são bem definidas e é utilizada uma única estrutura de dados (a relação) para armazenar todo tipo de dado. Podemos afirmar, portanto, que todos os dados são tratados de uma maneira uniforme. Isso se aplica de maneira análoga a outros tipos de BDCs.

No entanto, não é possível obter-se uma situação tão elegante para projetos de CIs. Num dado instante, o projetista pode estar manipulando uma certa representação de uma célula (que pode ser uma estrutura de dados bastante complexa e variada), ou forma de onda, ou um conjunto de regras de projeto.

Além disso, não é possível manipular-se todas as formas de se representar uma célula da mesma maneira. Por exemplo, os algoritmos usados para manipular traçados (informações geométricas) determinam a forma como estes devem ser armazenados em memória secundária (disco), já que normalmente não é viável armazenar um traçado completo em memória principal.

1.2.2 Transações E Consistência

Iremos reproduzir aqui a definição de transação como descrita em [19], que por sua vez usa [20]. Uma transação é, basicamente, uma

sequência atômica de operações de leitura e escrita num BDC. Um evento que cause a finalização anormal de uma transação faz com que o BDC volte ao estado no qual se encontrava antes do início da transação [*].

Transações num ambiente de VLSI PAC[+] podem durar muito tempo (algumas horas ou dias) e implicar no uso de um volume de dados muito grande. Em contraste, espera-se que uma transação num BDC seja de curta duração e na maior parte das vezes o volume de dados manipulados durante uma transação é comparativamente pequeno.

Além disso, assume-se que o conteúdo de um BDC permaneça consistente antes e após a execução da transação. Ou seja, uma transação preserva a integridade do BDC verificando se todas as regras de consistência (estáticas e temporais) foram respeitadas.

Porém, um projeto de um CI só atinge um estado consistente quando ela termina. Durante todo o processo de concepção, síntese e verificação, o BD deve armazenar dados parciais e inconsistentes. Um dos desafios é justamente esse: implementar mecanismos no sistema de gerenciamento que auxiliem e orientem o projetista a manter a consistência dos seus dados.

*] Existem propostas de adotar um outro modelo de transação para ambientes de PAC [19]. Por enquanto, somos da opinião de que tais modelos impõem restrições inconvenientes quando aplicados em VLSI PAC.

+] VLSI é a abreviação de "Very Large Scale Integration". Circuitos bipolares e CMOS com mais de 30000 elementos ativos são classificados dessa forma.

1.2.3 Acesso Concorrente

Normalmente, projetistas distintos atualizam partes distintas do CI, e conflitos são raros. Transações que causem a colocação de "locks" conflitantes nos dados podem ser cancelados, notificando-se devidamente o usuário.

Esta estratégia minimiza (ou praticamente elimina) a possibilidade de "deadlocks" e simplifica a implementação do conceito tradicional de transações.

Num BDC há uma grande preocupação com esse problema, já que acesso concorrente é absolutamente imprescindível. O "overhead" resultante é uma das razões que tornam um gerenciador de BDC ineficiente para VLSI PAC.

1.3 REQUISITOS ADICIONAIS

Além de atender às exigências de VLSI PAC enunciadas na seção anterior, o sistema de gerenciamento de um BDCI deve apresentar as seguintes características:

1. Suportar diferentes Linguagens de Programação

   A nossa experiência e a de outros [4] mostra que numa mesma instalação são utilizados programas de aplicação (PAs) escritos em várias linguagens de programação (Fortran, Pascal, C, Lisp, e mais recentemente Prolog).

Vemos diversos motivos para o surgimento dessa situação:

- o VLSI PAC abrange uma grande classe de problemas e para resolvê-los adota-se uma linguagem de programação adequada a cada problema.

- o Não é possível desenvolver todos os PAs numa mesma instalação. Alguns devem ser adquiridos e nesse caso a linguagem adotada pode ser outra do que aquela normalmente utilizada em certa instalação.

- o Certos PAs levam muito tempo (homens/ano) para serem desenvolvidos. Nem sempre é possível deixar de utilizá-los, ou reescrevê-los simplesmente porque uma outra linguagem foi adotada.

2. Generalidade e Adaptabilidade

O BDCI não deve estar orientado para nenhum sistema específico de projeto de CIs, ou seja, ele deve adaptar-se

1. ao conjunto de programas de aplicação disponíveis,

2. à metodologia de projeto adotado e, consequentemente,

3. aos níveis de abstração e estruturas de dados usados.

Porém, deve possuir mecanismos que induzam o projetista a utilizar uma certa metodologia, por exemplo, descendente (em inglês,

"top-down"). Esses mecanismos devem ser, no entanto, opcionais a fim de não inibir a liberdade do projetista.

O sistema de gerenciamento deve também se adaptar aos níveis de sofisticação dos PAs. Haverá sempre aqueles que se utilizarão apenas das funções básicas. Um PA, porém, evolui e poderá passar a usar funções mais sofisticadas tornando o sistema mais "inteligente" e útil ao projetista.

3. Suporte a Sistemas Distribuídos

Deve ser possível o acesso a projetos localizados em outros nós de um sistema distribuído de forma simples e transparente para os PAs.

Há uma tendência crescente em se utilizar estações de trabalho ("workstations") para projeto de CIs, tal que porções diferentes de um mesmo circuito possam ser projetados em estações distintas. Neste caso, as estações podem estar interligadas e o processamento "pesado" é realizado num computador de grande porte ("mainframe").

4. Suporte a Processamento Concorrente

Durante uma sessão o gerenciador de BD (GBD) deverá ser capaz de atender e controlar os pedidos de vários PAs concorrentemente. Isso facilita a implementação de um sistema onde o projetista pode ativar vários PAs simultaneamente, permitindo assim um ambiente de

trabalho mais flexível.

5. Facilidade de Uso

Não podemos assumir que o usuário (no caso, um projetista) tenha conhecimentos sólidos de programação. O usuário será o mesmo indivíduo que irá criar, consultar, atualizar e organizar as informações no BDCI.

Portanto, é importante que a visão que ele possui do sistema e das funções oferecidas seja bem adequada às suas necessidades e de fácil uso. No entanto, o BDCI deve ser passível de adaptação ao grau de sofisticação do usuário.

Entendemos também que um sistema não confiável, sem capacidade de recuperação de erros, e sem mensagens de erro apropriadas não é "fácil" de ser usado.

Outros:

6. Um BDCI deve permitir o acesso a um número ilimitado de projetos durante uma sessão. Além disso, objetos num BDCI devem poder referenciar facilmente qualquer outro objeto em todos os outros projetos disponíveis.

7. A movimentação de dados para dentro e fora do BD deve também ser fácil.

8. Não deve haver distinção entre um projeto e uma biblioteca (p.ex., de células). Essa distinção deve ficar a cargo do usuário. Isso facilita tanto o seu uso como a sua implementação.

1.4 DISCUSSÃO E LEVANTAMENTO BIBLIOGRÁFICO

Podemos separar a evolução nessa área em diversas fases:

1. A tentativa, frustrada, de usar somente um BD convencional num ambiente de VLSI PAC (uso do modelo de redes).

2. Caracterizar as funções de gerenciamento necessárias num ambiente de VLSI PAC, e separando esse aspecto do problema da representação dos dados.

   Surge então a proposta de dividir o sistema de BD em duas camadas. Há propostas, por exemplo, de se usar um sistema de gerenciamento orientado para VLSI PAC e usar um modelo de dados convencional para a representação dos dados.

3. Quanto aos modelos propostos, podemos identificar várias tendências:

- o Uso do modelo relacional com extensões.

- o Uso de uma mistura do modelo relacional com o de redes.

- o Uso de outros modelos (p.ex., entidade-relacionamento), e tentativas não ortodoxas (p.ex., uso de linguagens funcionais).

O primeiro trabalho de que temos conhecimento que caracteriza bem os problemas encontrados no uso exclusivo de um BDC num ambiente de VLSI PAC é o de Sidle[7].

Zintl[8] reconhece as dificuldades, mas os problemas considerados fundamentais (inflexibilidade face às mudanças nas estruturas de dados e acesso a um volume pequeno de dados num dado instante) foram contornados criando-se um sistema de interface. No entanto,

1. as funções de gerenciamento a nível de projeto permanecem inadequados (p.ex., não há controle de versões), e

2. o sistema é flexível mas provavelmente ineficiente para certas aplicações (p.ex., manipulação de traçados)

As funções de gerenciamento foram também bem descritas por Bennett[10], Katz[6,11], e Claesen[5].

Katz[6] também faz um levantamento bibliográfico de 1974 até 1981. Ele aponta uma das deficiências fundamentais do modelo relacional: a impossibilidade de representar os dados de forma hierárquica. Porém, é importante distinguir entre problemas no modelo em si daqueles decorrentes da implementação. Wilkins et al[18] mostram que para certas aplicações (p.ex., expansão) a eficiências dos modelos relacional e entidade-relacionamento (ER) pode ser melhorada se os métodos de acesso forem orientados para aplicações em VLSI PAC.

Dos modelos tradicionais, o relacional é o mais adaptável a novos esquemas e estruturas. Portanto, vários esforços foram feitos para estender o modelo relacional ou facilidades especiais foram implementadas a fim de usá-lo num ambiente de VLSI PAC.

Katz[6] divide o sistema de BD em três níveis:

1. "Database Component"

   É o nível mais baixo, baseado no modelo relacional. Apenas um registro por vez pode ser obtido (portanto, é ineficiente para o acesso a um grande volume de dados). Ele reconhece que os métodos de acesso tradicionais (sequencial, indexado baseado em arvores-B) não são suficientes, e que técnicas especiais para atender problemas específicos devem ser implementados (p.ex., manipulação de textos).

   Além disso, há um subsistema de gerenciamento que implementa a noção tradicional de transação, armazenamento confiável, e

recuperação de erros.

2. "Design Management"

Este componente implementa as funções de gerenciamento orientado para VLSI PAC, a noção de hierarquia e múltiplas representações. Sua implementação usa as funções do nível anterior.

3. "Design Interpreters"

Este último nível é um conjunto de interfaces que permite manipular os dados de forma conveniente a cada aplicação. Por exemplo, haveria uma interface para um editor de esquemático e outro para um extrator de circuitos a partir de um traçado.

Bennett[10] apresenta uma proposta análoga a Katz[6], e também adota o modelo relacional.

Haynie[12] faz uma análise do modelo relacional. São apontadas as soluções adotadas por vários autores para resolver os problemas citados (limitações quanto aos métodos de acesso, manipulação de objetos complexos, múltiplas representações, etc).

Porém, há uma tendência atual que parece apontar para uma mistura do modelo relacional e o de redes. Zara e Henke[13] apresentam argumentos bastante claros e convincentes para isso. Uma das razões que tornam o modelo relacional ineficiente é a necessidade constante de se deduzir (computar) o relacionamento entre os dados.

O modelo relacional tem a seu favor o fato de ser flexível, já que o PA identifica apenas os dados desejados e não como obtê-los. No entanto, essa flexibilidade nem sempre é desejável (p.ex., quando certas estruturas já atingiram um estado estável). Por outro lado, a presença de apontadores no modelo de redes permite o acesso rápido a dados relacionados entre si. Ou seja, o apontador é o resultado de uma computação já feita anteriormente e que não precisa ser refeita.

Assim, dependendo da aplicação podemos utilizar o modelo que julgarmos mais conveniente e/ou eficiente.

Além disso, esse trabalho é o primeiro que estuda o problema de adaptabilidade e flexibilidade não do gerenciador ou modelo, mas dos programas de aplicação. Ou seja, quando uma aplicação exige mudanças na representação dos dados, o problema não é se o BD se adaptará, mas sim se as ferramentas antigas sofrerão modificações ou não.

Basicamente, o que é proposto é um modelo de programação baseado no modelo mensagem/objeto (presente na linguagem SMALLTALK), de tal forma que alterações no modelo de representação de dados causa modificações na interface do BD com o PA, mas não no PA.

Batory[14] expõem uma proposta elegante usando o modelo ER para modelar certos aspectos de gerenciamento (controle de versões, múltiplas representações) e para representação de dados, incluindo objetos (p.ex., células) parametrizados.

Uma das motivações para o uso de modelos semânticos é que eles permitem a manipulação de objetos complexos, ao contrário dos modelos tradicionais que estão orientados para a manipulação de registros (essa é a razão pela qual é necessária uma camada entre o usuário e/ou PA e o BDC).

Vamos agora às tentativas não ortodoxas.

Um dos problemas intrigantes em VLSI PAC é a consistência entre as diversas representações. (Duas representações de um mesmo circuito são consistentes se elas obedecem a uma dada especificação. Intuitivamente, os dois circuitos se comportam da mesma forma e implementam a mesma função.)

Façamos uma distinção entre consistência topológica e consistência funcional, onde esta última incorpora a noção tradicional de consistência (definida acima). Consistência funcional é um problema bastante complexo que não pode ser resolvido dentro do contexto de banco de dados. Sua solução é justamente o desenvolvimento de ferramentas de verificação e síntese de representações (p.ex., simuladores lógicos e elétricos, e geradores de PLAs [+]). Isso porque o relacionamento entre os dados que determinam o comportamento do circuito é muito difícil de ser formalizado. No momento não existe nenhuma proposta geral deste tipo de formalização.

·] PLA é a abreviação de "Programmable Logic Arrays".

Já a consistência topológica (CT) é mais "fácil" de ser verificada. Normalmente, diz-se que duas descrições de um circuito são consistentes (topologicamente) entre si se o CG ("circuit graph") de ambos forem isomorfos. O CG de uma descrição de circuito é formado da seguinte forma:

1. V(CG), o conjunto de vértices do grafo, é dado por

   V(CG) = {Ci | Ci é uma componente do circuito}.

2. Se o terminal "a" da componente Ci está ligado ao terminal "b" da componente Cj então o arco (Ci,Cj) pertence ao conjunto de arcos de CG, N(CG).

(Não achamos essa definição muito adequada. O mapeamento de uma descrição estrutural de um circuito para hipergrafos é imediata.)

A complexidade desse problema é NP-completo, mas existem procedimentos heurísticos que dão bons resultados na prática. Beetem[15], porém, adota uma solução original para o problema. Ele implementa um sistema que impõe o isomorfismo entre todas as representações permitidas.

Em linhas gerais, o sistema parte de um estado onde todas as representações estão consistentes entre si (o banco de dados está vazio). À medida que o usuário modifica uma representação R1 utilizando um conjunto pré-definido de operações, o sistema armazena junto às outras uma lista de ações ("action list"). Quando uma outra

representação R2 é modificada ela se tornará consistente com R1 quando as ações sobre R2 forem equivalentes àquelas na sua lista de ações.

Esse sistema impõem uma restrição muito grande que é o isomorfismo entre TODAS as representações, que na prática não é tolerada pelos usuários.

Uma outra abordagem ao problema da representação de dados que consideramos não ortodoxa é o uso de linguagens funcionais. Um exemplo é o uso de "frames" e "demons". Explicando brevemente, um "demon" é uma função associada a um valor de um "registro". Se esse valor não estiver presente quando requisitado, a função é executada, e o resultado de tal execução pode ser, por exemplo, um valor default.

O uso de Lisp foi mais explorado em Claesen[23]. Foi desenvolvida uma linguagem baseada em Lisp, chamada Cellist, que é utilizada para descrever os dados (tanto células como modelos, regras de projeto, etc). Cada descrição é um conjunto de funções. Para se obter uma informação do gerenciador de dados, é necessário identificar a função desejada e um conjunto opcional de parâmetros [+].

As vantagens de tal método de representação são:

1. Poder de expressão: tem-se à disposição todas as facilidades da linguagem Lisp. Assim, é possível com uma única linguagem e interface descrever e ter acesso a dados referentes a qualquer

[+] Parece haver uma semelhança entre esta proposta e tipos abstratos de dados. Em tipos abstratos de dados o acesso aos dados é também feito através de chamadas a funções.

entidade (objeto) em qualquer nível de abstração.

2. Flexibilidade: se a natureza dos dados (semântica) ou a estrutura de dados mudam, basta reescrever as funções. Não há, portanto, alterações na interface com os programas de aplicação. As funções antigas podem ser preservadas a fim de manter a compatibilidade com as várias versões de bancos de dados.

O uso de linguagens funcionais pode ser encontrado em outras aplicações na área de projeto de circuitos integrados. Uma das dificuldades apontadas tem sido a sua ineficiência inerente. No entanto, observou-se que essa ineficiência é devido ao fato do "hardware" utilizado não ser adequado a esse tipo de aplicação. Ou seja, o uso de linguagens funcionais pode ser uma boa solução, mas na prática ela deve ser adotada apenas quando há disponibilidade de "hardware" apropriado.

Essa é uma das principais razões pelas quais deixamos de lado essa abordagem, e passamos a desenvolver uma proposta que culminou com o gerenciador de dados chamado OMS.

CAPÍTULO 2

DMS, UM GERENCIADOR DE PROJETOS

2.1 INTRODUÇÃO

O DMS (Design Management System) está encarregado de gerenciar os arquivos de dados. Ele não se preocupa com o conteúdo dos mesmos, mas apenas com a sua localização, com como eles se relacionam entre si, e com seus atributos e propriedades.

O DMS manipula representações de "entidades". Uma entidade é uma célula ou, p.ex., uma descrição de um conjunto de regras de projeto. As entidades são organizadas como mostra a figura abaixo.

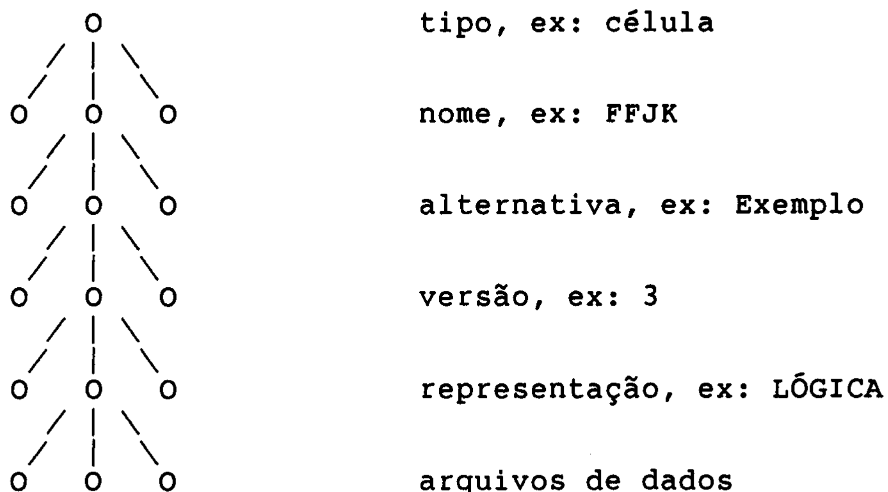


Figura 2.1
Organização das entidades

Cada nível possui uma função pré-definida. Os quatro primeiros identificam uma entidade. Assim, a cadeia de caracteres

"BIB:célula.FFJK[Exemplo];3"

identifica a entidade indicada na figura acima, onde "BIB" é o nome do projeto. Não há restrição quanto ao número de elementos em cada nível.

Os tipos dividem as entidades em grupos distintos. Eles servem para separar, por exemplo, as informações referentes a células daquelas referentes à descrição de tecnologia. As alternativas podem ser usadas para identificar diferentes implementações de uma mesma célula. As versões devem ser utilizadas para agrupar representações consistentes entre si. Voltaremos a esse assunto mais adiante.

Finalmente, associada a cada entidade pode haver um conjunto de representações, e cada representação está, por sua vez, associada a um conjunto de arquivos.

O DMS permite que representações de entidades sejam relacionadas entre si. Com isso, pode-se saber quais foram as informações utilizadas para se gerar uma dada representação, ou o inverso. Como veremos adiante, isso facilita a manutenção da consistência entre as diversas representações.

2.2 MECANISMOS DE GERENCIAMENTO DE REPRESENTAÇÕES

Note que um dos objetivo do gerenciador é indicar para o usuário como as suas ações afetam as informações usadas para projetar um determinado CI.

Seguindo essa linha de pensamento, foram implementadas as funções abaixo.

2.2.1 Validação

Introduzimos aqui o conceito informal de "validação" de uma representação. Considera-se uma representação como validada quando ela é consistente com sua especificação. Essa informação é dada ao DMS pelo projetista ou por programas que sintetizam representações automaticamente como, por exemplo, um gerador de PLAs. Uma representação validada não pode ser alterada.

Quando há uma tentativa de atualizar uma representação validada, o sistema cria automaticamente uma outra versão, como mostra a figura abaixo. Outros critérios podem ser utilizados [14] mas que normalmente exigem que o gerenciador tenha conhecimento da estrutura interna dos arquivos e dos objetos sendo manipulados pelo programa.

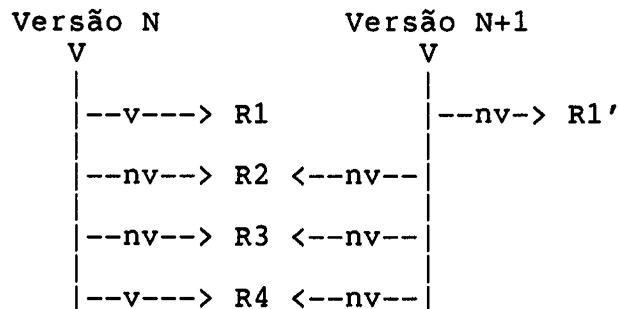


Figura 2.2
As letras "v" indicam que a representação foi validada e "nv" o oposto.

2.2.2 Criação De Novas Versões

Suponha que haja um pedido de atualização da representação R1 (fig. 2.2). Ao final da transação, o sistema criará uma nova versão que apontará para uma outra descrição R1' da representação R1. No entanto, observe que as outras representações são compartilhadas por ambas as versões. Porém, a representação R4 não será mais considerada validada quando utilizada via versão "N+1". Se posteriormente houver uma tentativa de atualização da representação R4, não será criada uma nova versão, mas sim, faz-se a versão "N+1" apontar para uma nova descrição R4'.

Normalmente, quando uma representação não validada é atualizada não é criada uma nova versão. O objetivo das versões é permitir o agrupamento de representações validadas consistentes entre si. Por exemplo, suponha que uma célula possua uma representação lógica e elétrica, ambas validadas. Se a representação lógica é alterada, é desejável que na nova versão a representação elétrica não conste como

validada, já que ela representa um nível de abstração mais baixo obtida a partir da representação lógica.

Por outro lado, se a representação elétrica é atualizada, a representação lógica poderia constar como validada na nova versão. Para que isso seja possível, é necessário que o DMS aceite uma descrição hierárquica dos níveis de abstração. Se essa descrição não existir, é assumido o pior caso, isto é, todas as representações são invalidadas na nova versão.

De qualquer forma, no caso acima, as representações que utilizam a representação elétrica alterada serão notificadas.

2.2.3 Hierarquia Das Representações

A fim de que o mecanismo de validação e criação de novas versões seja mais adequado para VLSI PAC é necessário que o DMS tenha uma descrição da hierarquia das representações.

A hierarquia é uma forma de descrever como uma representação se relaciona com as outras em função de seu nível de abstração. Como cada tipo tem um conjunto distinto de representações, é possível haver uma descrição hierárquica para cada tipo.

A linguagem de descrição segue a seguinte sintaxe:

```
<hierarquia> :=  nome
               | nome (<lista_de_nomes>)

<lista_de_nomes> := nome
                  | nome <lista_de_nomes>
                  | nome (<hierarquia>)
```

Por exemplo, suponha que a descrição abaixo seja referente ao tipo "cell".

terminals (*)

functional (floorplan logic (electric (layout)))

floorplan (layout)

O símbolo "*" indica que a representação "terminals" é o nível da hierarquia de representações mais alto (isso é apenas "açúcar sintático"). Ou seja, se uma representação "terminals" validada de alguma entidade do tipo "cell" for atualizada, então todas as outras representações na nova versão da entidade também serão invalidadas.

Podemos representar graficamente parte do exemplo acima como mostra a figura abaixo (a representação "terminals" não foi colocada devido à dificuldade de montagem da figura):

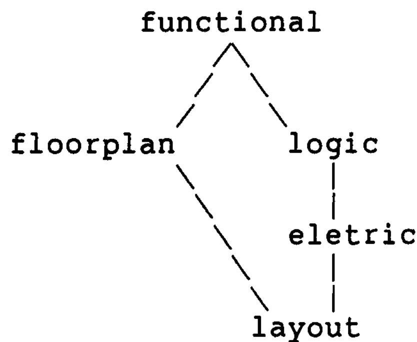


Figura 2.3

2.2.4 Relações, Propriedades E Relações De Afetação

Já mencionamos a necessidade de haver relações entre as diversas

representações. Para ilustrar, suponha a existência das relações "usa" e "usado_por", onde

- o "usa(E1:R1)" identifica todas as representações R2 das entidades E2 que usam a representação R1 da entidade E1, e
- o "usado_por(E1:R1)" é a inversa, ou seja, quem são as representações usadas por R1 da entidade E1.

Assim, não deve ser permitida a destruição de uma representação E1:R1 que é usada por outras, isto é, se o conjunto usa(E1:R1) não for vazio. Além disso, se ela é alterada, uma notificação deve ser colocada nas representações que a usam.

Por outro lado, se uma representação é invalidada, então as representações que ela "usa" também devem ser invalidadas.

Associada a cada representação podem existir uma ou mais propriedades. Por exemplo, a representação "StickDiagram" pode possuir a propriedade "tamanho". Além disso, pode-se especificar como uma propriedade "afeta" outras propriedades. Por exemplo,

```
Afeta(E1:StickDiagram.tamanho) => Afeta(E2:R.tamanho)
        sse Pertence(E2:R,Usa(E1:StickDiagram))
```

Isso significa que todas as propriedades "tamanho" de todas as representações que usam E1:StickDiagram são também afetadas.

Observamos uma afinidade dessa proposta com outras áreas, tais como lógica matemática e programação funcional.

2.3 TABELAS OU ATRIBUTOS

Se uma representação possui um determinado atributo ela é colocada numa certa tabela. Existe uma relação biunívoca entre tabelas e atributos, e podemos então usar ambos os termos indistintamente.

Para ilustrar o uso de atributos, utilizaremos um exemplo. Considere uma sessão com um editor gráfico, com o qual pode-se criar os desenhos das figuras 1.1a e 2.4.

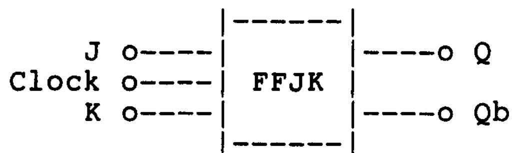


Figura 2.4

Na figura 2.4 temos uma representação "black-box" da célula FFJK. Uma das facilidades que o editor gráfico deve oferecer é a possibilidade de listar para o usuário todas as células que podem ser utilizadas para o projeto de outras.

Essas células são todas aquelas que possuem uma representação "black-box" validada. Utilizando o modelo apresentado até agora, o editor gráfico seria obrigado a pesquisar o BDCI todo em busca dessas entidades. Pode-se, no entanto, criar um atributo para as entidades que possuam uma representação "black-box" validada, e bastará fazer uma busca na tabela correspondente.

A implementação é bastante simples, e a única dificuldade é listar de forma eficiente as entidades que não estão numa certa tabela.

2.4 SINÔNIMOS

A maioria dos programas de aplicação identificam células diferentemente da forma como é feito pelo GBD. Por exemplo, o simulador elétrico SPICE2G (largamente utilizado no mundo todo) permite que sejam utilizados apenas 6 caracteres para o nome dos elementos. Além disso, suponha que seja necessário utilizar a cadeia "FFJK[Exemplo];3" em todas as representações que referenciam aquela célula. Se for necessário utilizar outra célula, então as referências àquela célula deverão ser substituídas.

Para simplificar esta identificação de células, o DMS aceita que uma tabela de sinônimos seja associada a um projeto. Assim, é possível associar a "FFJK" a cadeia "FFJK[Exemplo];3" e usá-lo para referenciar a célula desejada. Se for necessário utilizar uma outra célula, bastará alterar a tabela de sinônimos. (Isso é análogo as tabelas de nomes lógicos no sistema VAX/VMS.)

2.5 SUPORTE A SISTEMAS DISTRIBUÍDOS

Dependendo da complexidade do circuito, torna-se desejável alocar partições do circuito a projetistas diferentes para acelerar o seu desenvolvimento. Desta forma, o circuito seria projetado simultaneamente em mais de um processador. Pode também ser necessário

o acesso a uma biblioteca localizado em um outro nó da rede.

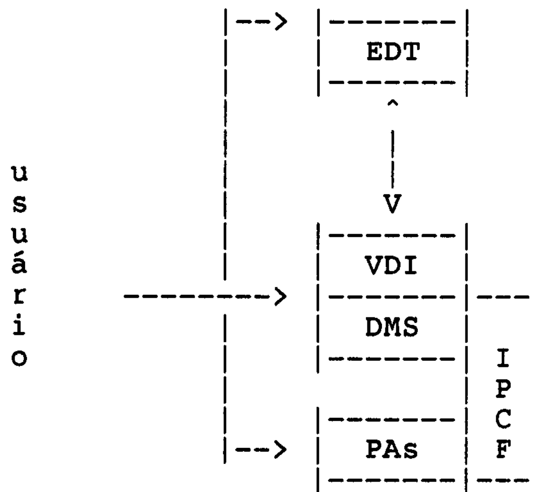
O único pré-requisito exigido pelo DMS para suportar sistemas distribuídos é a possibilidade de acesso a arquivos em outros nós utilizando rotinas já disponíveis no sistema operacional (no caso, VMS/RMS).

2.6 IMPLEMENTAÇÃO

O DMS foi implementado num sistema VAX/VMS utilizando a linguagem VAX Pascal, e é composto de aproximadamente 43 módulos num total de 15000 linhas, incluindo comentários (e sem nenhum comando GOTO).

Devido à natureza do problema foi necessário fazer uso de funções específicas do sistema VAX/VMS, restringindo assim a sua portabilidade. Pior do que isso, o DMS foi concebido assumindo a existência de certas facilidades (p.ex., para tratamento de erros).

Nem todas as funções descritas foram implementadas devido a restrições de tempo. No entanto, o DMS já foi usado e testado no desenvolvimento de um sistema com a seguinte configuração:



O VDI ("VLSI Design Interface") é um programa desenvolvido também por nós para servir de interface entre o usuário e o sistema. Através dele o usuário tem acesso a funções do DMS (criar um projeto, configurar um projeto, listar informações contidas num projeto, etc), ativar vários PAs, e ter acesso a todas as funções do sistema operacional (VMS).

A linguagem de comandos é bastante simples e inspirada no DCL (linguagem de comandos do VAX/VMS). Há estudos em andamento para a implementação de uma interface baseada em janelas e menus. Uma descrição mais detalhada do VDI pode ser encontrada em [21].

O EDT (editor de textos do VAX/VMS) pode ser utilizado para a entrada de dados textuais no BDCI.

O IPCF ("InterProcess Communication Facility") representa as facilidades de comunicação entre processos oferecidas pelo sistema VAX/VMS. As facilidades utilizadas são "mailboxes" e eventos ("event flags"). Pode-se imaginar um "mailbox" como sendo um arquivo

sequencial localizado em memória virtual.

Quando o usuário utiliza um comando que causa a execução de um PA, o VDI pede ao DMS que seja criado um "pipe". Um "pipe" é um par de "mailboxes" ligado a um subprocesso, onde um é utilizado para leitura e o outro para escrita. Associado ao "mailbox" de leitura há um procedimento assíncrono que é ativado sempre que o PA executa no mesmo uma operação de escrita.

Há três modos de operação para um "pipe". Num deles o controle volta ao VDI somente quando o "pipe" é destruído (o PA termina sua tarefa). No outro, o controle pode voltar ao VDI assim que o PA é ativado e ambos podem ser executados concorrentemente. Finalmente, o VDI e o PA podem ser executados como co-rotinas, ou seja, sob comando do usuário, enquanto o VDI está sendo executado o PA permece em estado suspenso, e vice-versa.

O PA desconhece o fato de estar sendo executado num subprocesso. A diferença está na biblioteca de rotinas que foi utilizada quando a imagem foi criada.

2.6.1 Uso De Um BD Relacional

Gostaríamos de comentar brevemente o uso de um BD relacional conjuntamente com o DMS.

Da forma como o DMS foi implementado, há restrições quanto à variedade de consultas que podem ser feitas. Algumas funções como tabelas e sinônimos poderiam ser implementadas usando um BD relacional. Além disso, a implementação de novas consultas exige que novas rotinas sejam incorporadas e, normalmente, também exige mudanças na estrutura de dados em disco, o que faz com que uma versão do DMS não seja compatível com a anterior.

Por esses motivos, somos da opinião de que seria muito útil que o DMS tivesse acesso a um BD relacional, a fim de que sua funcionalidade e flexibilidade fossem maiores.

A razão pela qual isso não foi feito desde o início é que não se tinha pleno conhecimento da natureza do problema, ou seja, quais seriam as atribuições e funções do DMS.

Isso nos aproxima do sistema proposto por Katz[6].

2.7 USO

2.7.1 Criação De Um Projeto

A criação de um projeto é muito simples. Basta utilizar a função CreateDesign especificando um nome e sua localização. O nome é uma cadeia de caracteres e a localização é um diretório qualquer (apenas um projeto pode residir num dado diretório).

Em seguida, é necessário configurá-lo. Isso consiste em especificar os tipos de entidades e representações válidas e, opcionalmente, a hierarquia das representações.

Essas informações são dependentes de cada instalação. Portanto, cada uma deve possuir um conjunto de comandos que possa ser utilizado pelo usuário para configurar seus projetos. O programa VDI possui um mecanismo de execução automática de tal conjunto de comandos.

2.7.2 Configurando Um Projeto

Configurar um projeto corresponde a especificar os ítens seguintes:

1. O gerente do projeto

   O gerente é o único que possui certos privilégios, como, por exemplo, criar e apagar tipos, alterar a proteção de entidades e representações, etc.

2. Criar os tipos que poderão ser utilizados nesse projeto.

   Numa instalação deve haver um consenso quanto ao nome dos tipos, já que normalmente certos PAs irão procurar dados em tipos específicos.

3. Identificar as representações válidas para cada tipo.

   O DMS exige que os nomes das representações sejam definidas antes de serem usadas. Novamente, deve haver um consenso quanto aos

nomes, pois PAs irão armazenar e obter dados em representações específicas.

4. Especificar a hierarquia das representações.

2.7.3 Utilizando Um Projeto

Antes que um projeto possa ser utilizado, ele deve ser carregado. Ao ser carregado, o projeto é colocado em duas listas:

1. Uma é a lista de projetos carregados. O DMS só termina sua execução normalmente quando essa lista estiver vazia.

2. A segunda é uma lista ordenada utilizada para busca de entidades, e é chamada de "path". Seu uso será explicado mais adiante.

Duas informações devem ser dadas no momento da carga:

1. A especificação do projeto, e

2. um diretório onde estão as informações particulares do projetista.

A especificação do projeto compreende o nome e o diretório VMS/RMS onde ele se encontra. No diretório particular podem ser armazenadas informações tais como:

1. O nome do tipo default, que é utilizado sempre que na especificação final o tipo for omitido.

2. O nome da representação default, que é utilizado sempre que ele for omitido em qualquer chamada a qualquer função do DMS.

3. O nome do "Set" default. A função do "Set" será descrita mais adiante.

4. Arquivos de dados particulares.

2.7.4 Acesso A Representações De Entidades

O acesso é feito através de três funções básicas:

1. OpenTransaction

   Os argumentos são:

   1. Especificação da entidade: é uma cadeia de caracteres que identifica a entidade. Sua sintaxe já foi descrita na seção 2.1.

   2. Nome da representação: esse nome deve ser válido para o tipo dado na especificação da entidade.

   3. Modo de acesso: as opções possíveis são leitura ou escrita. O modo de acesso influencia a forma como o "path" é pesquisado.

4. Opções: o DMS permite que o usuário modifique certas decisões face a certos eventos.

5. Identificação da transação: esse é um valor retornado pelo DMS que deve ser utilizado pelo usuário ao utilizar outras funções sobre a mesma transação.

2. CloseTransaction

Os argumentos são:

1. Identificação da transação

2. Opções: o usuário pode, por exemplo, cancelar a transação.

3. RequestFileSpec

Os argumentos são:

1. Identificação da transação

2. Identificação do arquivo: associada a cada representação há um conjunto de arquivos. Logo, o usuário deve identificar de qual deles ele deseja a especificacão VMS/RMS.

3. Especificação do arquivo: é onde o DMS retornará a especificação VMS/RMS do arquivo.

Observe que a interface acima é bastante simples e de baixo nível. Com ele o programa de aplicação tem acesso de leitura ou escrita a uma representação de uma entidade. Uma vez que o PA obtem a especificação VMS/RMS, não há garantia de que o arquivo será utilizado da forma como indicado (leitura ou escrita).

A conceito de transação usado nesse contexto é bem simplificado. A estrutura de dados mantido em disco pelo DMS só é atualizado no momento em que um CloseTransaction é efetuado. O que se garante é que essa estrutura de dados não será corrompida por uma falha do tipo "crash" uma vez que o DMS se compromete a efetuar o término da transação.

Esse procedimento, no entanto, não é concorrente. Ou seja, uma vez que o DMS se compromete a finalizar a transação ele não devolve o contrôle ao programa de aplicação e, em seguida, a atualizar a estrutura de dados concorrentemente.

2.7.5 Uso Do "path"

O "path" é utilizado sempre que for omitido o nome do projeto na especificação de uma entidade. O uso do "path" depende do modo de acesso, portanto, descreveremos cada caso separadamente.

1. Modo de acesso é de leitura

   Nesse caso, cada projeto no "path" é pesquisado na ordem como eles estão na lista. A pesquisa termina quando a lista chega ao fim ou

quando é encontrada uma entidade com o mesmo tipo, nome, alternativa, e versão (caso ela tenha sido especificada).

2. Modo de acesso é de escrita

A pesquisa é feita na mesma ordem que no caso anterior. Se a pesquisa terminar devido ao fato da lista chegar ao fim, então, como a entidade ainda não foi criada, ela será criada no projeto default, que é o primeiro na lista. A pesquisa também pode terminar quando num projeto for encontrada uma entidade de mesmo tipo e nome (a alternativa e versão são ignorados).

Em ambos os casos, o "set" de cada projeto pesquisado é utilizado para se obter a especificação final. Além disso, os defaults associados a cada projeto são também utilizados caso os dados no "set" não preencham todos os campos da especificação.

Nem todos os projetos carregados precisam constar do "path".

2.7.6 Uso Dos "Sets"

Duas tabelas de sinônimos ("Sets") podem estar associadas a cada projeto. Uma delas é a tabela definida pelo gerente do projeto, e a outra é a tabela do projetista. A primeira a ser utilizada durante uma busca é a do projetista.

Uma tabela de sinônimos (do usuário ou do gerente) é implementada através de um arquivo textual criado via um editor de textos qualquer. A primeira vez que uma tabela é referenciada ela é compilada e armazenada de forma que o acesso possa ser mais rápido (ela é convertida num arquivo indexado-sequencial). Além disso, as entradas mais utilizadas são mantidas em memória principal. Evidentemente, se a tabela é alterada ela deve ser recompilada.

CAPÍTULO 3

SCHC: O COMPILADOR DE ESQUEMAS

3.1 INTRODUÇÃO

O Compilador de Esquemas (SchC) serve a dois propósitos:

1. Gerar rotinas que lêem uma estrutura de dados de um arquivo em disco e colocá-lo à disposição do programa de aplicação, ou vice-versa.

2. Servir de pré-processador para o OMS, ou seja, gerar uma descrição compilada de uma estrutura de dados.

O objetivo deste capítulo é dar uma visão geral do compilador. A seção 3.2 é uma breve introdução à linguagem de descrição de esquemas. A seção 3.3 descreve as rotinas de leitura e escrita de uma estrutura de dados e mostra como são gerados e usados pelo programa de aplicação. A forma como o compilador de esquemas gera uma descrição para o OMS é descrito no próximo capítulo.

Para maiores detalhes sugerimos a leitura de [25].

3.2 A LINGUAGEM DE DESCRIÇÃO DE ESQUEMAS

A linguagem de descrição de esquemas é composta de três partes, da mesma forma que a parte declarativa da linguagem Pascal: declaração de constantes, tipos e variáveis [+]. Veja o exemplo abaixo.

```
CONST
   max_ident_length = 31;

TYPE
   ident_vc_type  = VARYING [max_ident_length] OF CHAR;
   ident_p_type   = ^ident_rec_type;
   ident_rec_type =
     RECORD
       ident : ident_vc_type;
       left, right : ident_p_type;
     END;

VAR
   ident_list : ident_p_type;
```

Exemplo 3.1
Um esquema que representa um nó de uma árvore binária.

Descreveremos aqui apenas as extensões à linguagem Pascal que achamos necessárias para a nossa aplicação.

3.2.1 A Diretiva INCLUDE

Um esquema pode utilizar identificadores definidos em outros esquemas com a diretiva INCLUDE. Por exemplo, a declaração "INCLUDE A;" torna visível todos os identificadores declarados no

[+] Na implementação atual o analisador sintático foi extraído do compilador P4 desenvolvido por U.Ammann, K.Nori, C.Jacobi do Institut fuer Informatik eidg., Technische Hochschule CH-8096, Zurich.

esquema A.

Observe que essa diretiva introduz um pequeno problema: esquemas recursivos. Na sua forma mais simples, é a tentativa de um esquema Si incluir um esquema Sj que por sua vez inclui Si.

A implementação atual não prevê a prevenção destes casos, mas que pode ser solucionado armazenando-se todos os esquemas que são incluidos direta ou indiretamente por um esquema Si. Quando um esquema Sj tenta incluir o esquema Si basta verificar se Sj consta na lista de esquemas já incluídos por Si.

A diretiva INCLUDE é equivalente a inserir a descrição textual do esquema no ponto onde a diretiva se encontra. Na realidade, apenas a descrição compilada do esquema é utilizada.

3.2.2 Extensões Úteis Mas Não Implementadas

- o Facilidades como macros e compilação condicional, que estão presentes na linguagem C e em quase todas as linguagens de montagem, seriam muito úteis se implemetadas.

- o Um gerenciador de esquemas seria útil para gerenciar as descrições e as respectivas rotinas de leitura/escrita geradas pelo compilador de esquemas.

3.3 AS ROTINAS DE LEITURA/ESCRITA DE UMA ESTRUTURA DE DADOS

A finalidade dessas rotinas é facilitar o desenvolvimento de sistemas de grande porte onde há programas de aplicação possivelmente escritos em linguagens de programação distintas. Num ambiente de VLSI PAC é comum a existência de programas escritos em Fortran, Pascal, C, e, com frequência crescente, Lisp.

Os benefícios trazidos pelas rotinas de leitura/escrita são:

1. Aliviar o programador da tarefa de escrever rotinas que lêem/escrevem sua estrutura de dados. Assim, se dois programas (ou um programa executado em diversas fases) se comunicam através de um arquivo em disco, então as rotinas de interface podem ser geradas imediatamente e de forma confiável. Os dois programas poderiam inclusive ser programados em linguagens distintas, mas dentre aquelas previstas pelo compilador de esquemas.

2. Estas rotinas podem ser usadas para armazenar o estado do programa, e se houver uma queda do sistema então a execução pode prosseguir a partir do último estado armazenado. Essa é uma característica muito desejável em VLSI PAC, visto que algumas tarefas podem consumir muitas horas de CPU.

3. Estas rotinas também podem ser usados como rotinas de inicialização.

O próprio compilador de esquemas utiliza rotinas geradas por ele mesmo e observamos que isto facilitou o seu desenvolvimento e sua confiabilidade.

3.3.1 Uso

A figura abaixo mostra os diversos passos da geração e do uso das rotinas de leitura/escrita.

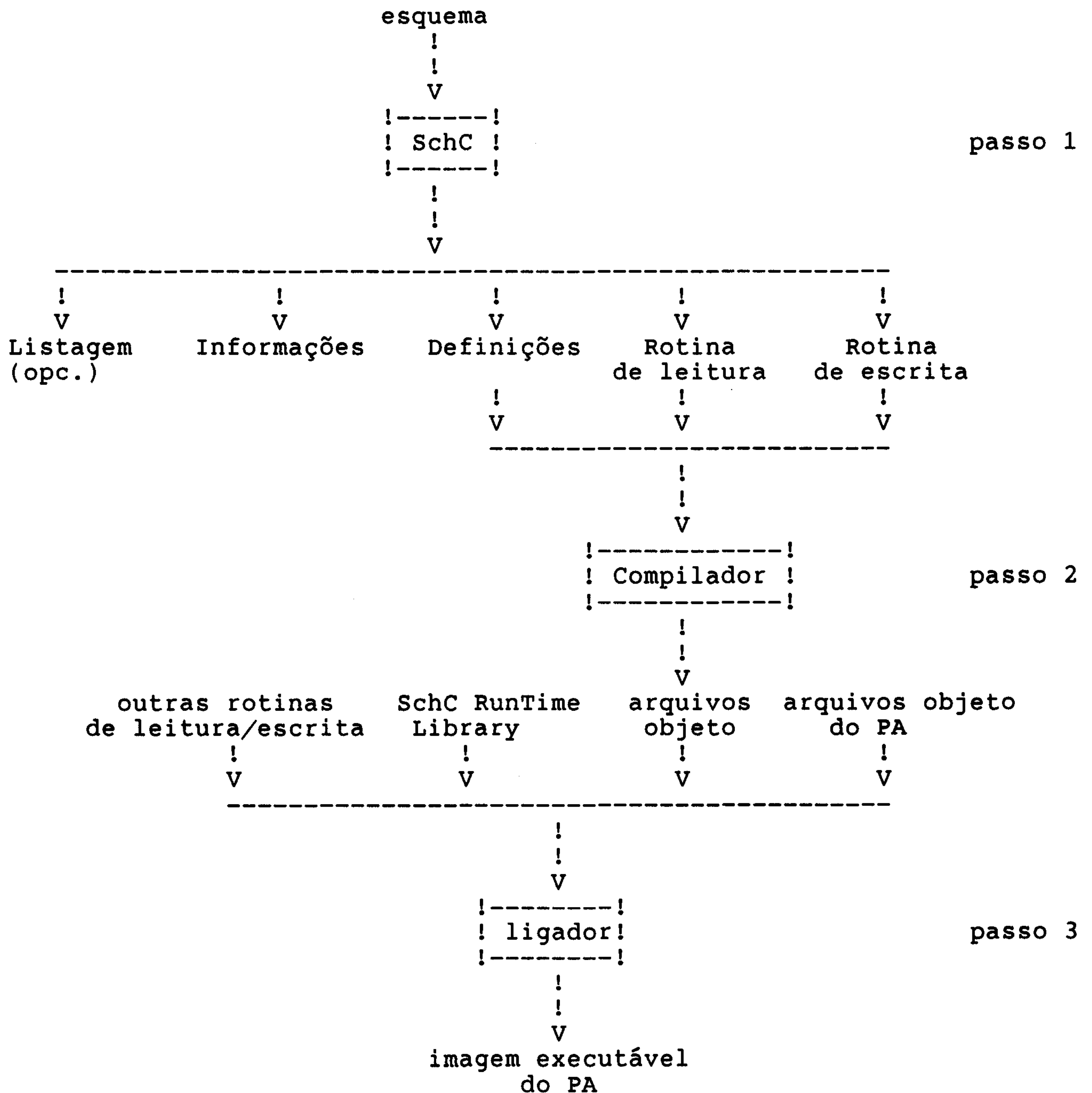


figura 3.1
Gerando as rotinas de leitura/escrita

O arquivo de definições e as rotinas de leitura/escrita variam de linguagem para linguagem. Além disso, as figuras acima para cada linguagem diferem em pequenos detalhes. A seguir, discutiremos brevemente cada passo.

1. Os arquivos gerados a partir do esquema são:

   o Uma listagem (opcional) que apresentará os erros (se houver) e as estatísticas de performance do compilador de esquemas.

   o O esquema compilado (opcional) que será usado se o esquema for utilizado por outro através da diretiva 'INCLUDE', ou se ele for utilizado pelo OMS.

   o Arquivos dependentes da linguagem:

      . Um arquivo de definições que é uma descrição equivalente da estrutura de dados na linguagem desejada;

      . e as rotinas de leitura/escrita também na linguagem desejada.

2. O código gerado deve ser compilado. Essa fase difere dependendo da linguagem a ser utilizada. Por exemplo, no caso da linguagem VAX Pascal, o arquivo de definições deve ser o primeiro a ser compilado, enquanto que na linguagem C ele é apenas incluído nos módulos que o referenciam. Há um detalhe adicional: cada arquivo de definições referencia um outro pré-definido para cada linguagem.

3. O último passo consiste na criação da imagem. Observe que as rotinas de leitura/escrita utilizam rotinas de uma biblioteca, sendo que há uma biblioteca para cada linguagem. Na figura, elas estão indicadas como "SchC RunTime Library".

Além disso, nada impede que várias rotinas de leitura/escrita sejam utilizadas num mesmo programa.

3.3.2 Implementação

Discutiremos brevemente o funcionamento das rotinas que compõem a "SchC RunTime Library". Há dois conjuntos de rotinas: as de entrada/saída e as de manipulação de apontadores. As rotinas de entrada/saída são triviais.

As rotinas de manipulação de apontadores podem ser divididas em duas classes:

1. uma classe trata dos apontadores para registros, e

2. a outra trata daqueles que não são para registros.

Em ambos os casos, o problema consiste em, dado um apontador, decidir se o objeto para o qual ele aponta já foi lido ou armazenado, dependendo da operação sendo efetuada (leitura ou escrita).

3.3.2.1 Tratamento De Apontadores Para Registros -

Nesse caso, o compilador de esquemas cria um campo adicional no registro, como pode ser observado no Apêndice A. Esse campo é referenciado pelo identificador "SCC_nr" [+], e deve ter um valor inicial igual a zero. Assume-se que os PAs são bem comportados e, sabendo da existência de tal campo, encarregam-se de colocar um valor inicial adequado quando um novo registro é alocado. Há também um contador global cujo valor inicial é igual a zero.

Numa operação de escrita, ao armazenar um apontador "p", verifica-se se o campo "p^.SCC_nr" é igual a zero ou não. Se não for igual a zero, então o valor armazenado para o apontador é "p^.SCC_nr". No caso de ser igual a zero, o contador global é incrementado e o seu valor é copiado em "p^.SCC_nr", e voltamos ao caso anterior. Chamaremos esse valor de valor virtual.

Após armazenar o valor virtual do apontador, o registro apontado é armazenado a seguir.

Numa operação de leitura, ao ler o valor associado a um apontador pela rotina de escrita temos que decidir se esse valor já foi lido ou não, e no caso afirmativo, qual é o seu valor real.

Observe que os valores virtuais se apresentam em ordem crescente. Portanto, para decidir se um valor virtual já foi lido basta compará-lo com o maior valor virtual lido até então.

+] Todos os identificadores que iniciam com a cadeia "SCC_" são reservados para uso do compilador de esquemas.

Se o valor ainda não foi lido anteriormente, devemos alocar memória para o registro que se segue, e armazenar o valor real associado a esse valor virtual. Para o mapeamento entre esses dois valores é utilizado uma lista de vetores de tamanho M. Dado um valor virtual "n", o valor real é armazenado em:

vetor = (n-1) div M

offset = (n-1) mod M + 1

Observe que o campo "SCC_nr" não é utilizado numa operação de leitura. Só devemos ter o cuidado de que seu valor inicial seja igual a zero.

3.3.2.2 Tratamento De Apontadores Para Não Registros -

A operação de leitura é análoga ao caso anterior, exceto que a lista de vetores é outra. Entretanto, numa operação de escrita não temos a "mordomia" do campo "SCC_nr". Para esse caso, é utilizado uma estrutura de dados semelhante, ou seja, uma lista de vetores. A diferença está no fato de que cada posição do vetor possui um par de valores, o valor virtual e o valor real. Os vetores estão ordenados em ordem crescente do valor real. Entretanto, essa propriedade não é válida se concatenarmos todos os vetores.

A implementação garante que todos os vetores estão completos, exceto possivelmente o último.

CAPÍTULO 4

OMS, UM GERENCIADOR DE ESTRUTURA DE DADOS

4.1 INTRODUÇÃO

Como já foi mencionado na seção 1.3 do capítulo I, existem vários programas de aplicação desenvolvidos para VLSI PAC. No entanto, o esforço necessário para adaptá-los a um sistema integrado é, em muitos casos, grande.

A finalidade do OMS ("Object Management System") é minimizar esse esforço, e além disso:

- o resolver parcialmente o problema da indenpendência dos dados [+],
- o permitir múltiplas visões a uma mesma estrutura de dados, e
- o permitir que o PA utilize diretamente os dados.

Esse capítulo deve ser considerado uma proposta, já que nada do que será descrito foi implementado.

[+] Qualificar a proposta de uma solução parcial não nos parece ser um aspecto negativo. Mesmo com um BDC, a independência dos dados é um problema complexo. Nenhum modelo isoladamente resolve a questão, pois é necessário também saber como usá-lo [24].

4.2 FUNCIONAMENTO DO OMS

O OMS é composto de três módulos:

1. O compilador de esquemas, que fornece uma descrição compilada das estruturas de dados. Esse módulo já foi descrito no capítulo anterior. Nesse capítulo descreveremos como o compilador de esquemas gera uma descrição compilada para o OMS.

2. Um mapeador ("mapper"), que, dadas duas estruturas de dados, gera uma terceira (chamada de "map") que descreve como uma deve ser mapeada na outra.

3. Um conjunto de rotinas (chamadas de RTOMS, ou seja, "RunTime OMS"), que devem ser utilizadas pelo PA para obter e colocar os dados na sua estrutura de dados.

A figura abaixo esquematiza o funciomento do OMS:

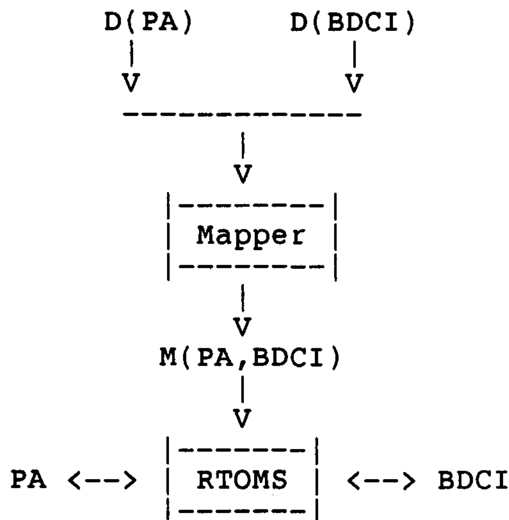


Figura 4.1

D(X) indica a descrição da estrutura de dados X produzida pelo compilador de esquemas. M(PA,BDCI) é uma estrutura de dados que guia o RTOMS no mapeamento de uma estrutura de dados na outra.

4.2.1 Geração Da Descrição Da Estrutura De Dados

A descrição compilada gerada pelo compilador de esquemas não possui uma informação muito importante para o OMS: a quantidade de memória ocupada por cada tipo utilizado (em bytes).

A razão pela qual o compilador de esquemas não executa essa tarefa é muito simples: foi mais fácil escrever um utilitário que gera uma rotina que calcula esses valores do que implementar o cálculo no compilador. Esse utilitário se chama "Convert Schema Utility" (CvtSchC), e maiores detalhes são apresentados em [25].

4.2.2 O "Mapper"

Dadas duas descrições de esquemas, uma de um programa de aplicação e outra de um banco de dados, a função do "Mapper" é:

1. detectar as partes em comum e gerar para o RTOMS um mapa de conversão de um para outro,

2. e gerar rotinas de alocação e destruição de registros.

Vamos descrever como o "Mapper" detecta as partes em comum. Num esquema há um conjunto de variáveis. O "Mapper" só considera variáveis de ambos os esquemas que tenham o mesmo nome, e estas são emparelhadas. Em seguida, verifica-se para cada par o tipo associado a essas variáveis.

Classificamos os tipos em dois grupos: (1) registros e apontadores para registros, e (2) os demais. No segundo grupo, as variáveis são ditas compatíveis se os tipos ocuparem a mesma quantidade de espaço em memória.

No caso de registros e apontadores para registros, o esquema do programa de aplicação pode selecionar um subconjunto dos campos especificados no esquema do banco de dados. Os critérios de emparelhamento dos campos é o mesmo definido para variáveis.

As rotinas de alocação e destruição de registros se fazem necessárias já que, como veremos na próxima seção, se o programa de aplicação tem a intenção de atualizar o banco de dados então ele não tem como decidir qual é o tamanho dos registros.

4.2.3 O RTOMS

A manipulação de apontadores é feita de forma análoga às rotinas utilizadas pelas rotinas geradas pelo compilador de esquemas (as rotinas que compõem a "SchC RunTime Library").

Um dos problemas fundamentais do RTMOS não é mapear as estruturas de dados do banco de dados nas estruturas de dados de um PA, mas o contrário. Se o PA tem a intenção de atualizar o banco de dados, então, quando a operação de escrita for efetuada, é necessário recompor o registro de acordo com o esquema do banco de dados. Assim, as partes não requisitadas dos registros devem ser mantidas. A descrição do registro no esquema do PA é chamada de parte visível. As partes não requisitadas são colocados acima da parte visível, e essa região é chamada (obviamente) de parte invisível.

O caso de registros de tamanho variável também merece tratamento especial. O RTOMS deve alocar apenas a quantidade de memória necessária para acomodar o registro quando este estiver sendo lido. Portanto, os campos que determinam o seu tamanho são sempre lidos primeiro, e o mapa gerado pelo "Mapper" guia o RTOMS no cálculo do tamanho.

4.3 RESTRIÇÕES

O OMS tem alguns aspectos negativos. Primeiramente, ele não pode ser utilizado para efetuar consultas. Além disso, se um programa de aplicação desejar inserir ou destruir um registro, ele deve atualizar

todos os apontadores relevantes.

Há uma outra restrição talvez pior. Pode ser que um programa de aplicação necessite de apenas um registro numa lista. O OMS não é capaz de tratar deste caso. Ou seja, da forma como ele se apresenta, o OMS pode ser considerado apenas como um reformatador de registros.

Uma possível extensão que estamos considerando para dar maior flexibilidade ao OMS é a possibilidade de associar a cada variável e campo de um registro uma função ou um procedimento. Por exemplo, uma função associado a um campo numa operação de escrita receberia como argumento o registro correspondente do banco de dados, e retornaria o valor para o campo. Isso daria liberdade ao programador para resolver problemas como os citados acima, e talvez até para efetuar mudanças na estrutura de dados.

Primeiramente, procuramos apresentar no que consiste a atividade de projeto de circuitos integrados assistido por computador. Devido às suas peculiaridades, são necessários um sistema de gerenciamento, um modelo de representação de dados, e métodos de acesso adequados, que divergem daqueles usados em sistemas convencionais.

Em particular, o sistema de gerenciamento de banco de dados orientado para projeto de circuitos integrados difere profundamente dos sistemas convencionais, e parece já haver um consenso nas funções que devem ser oferecidas. Por outro lado, a literatura parece indicar que o modelo relacional com extensões, uma simbiose do relacional com o de redes, e os modelos semânticos são apropriados para a representação dos dados. No entanto, vários métodos de acesso devem ser implementados a fim de satisfazer eficientemente certas aplicações e não há consenso em torno de um modelo.

Damos especial destaque ao uso de programação funcional para a representação e manipulação dos dados devido à sua flexibidade e poder de expressão. A ineficiência dessas abordagens deve-se principalmente ao "hardware" convencional utilizado. Cremos que essa desvantagem logo deixará de existir, visto que chegaremos ao ponto em que o "hardware" se adaptará ao "software", e não o contrário.

Em seguida, apresentamos a nossa proposta de também dividir o sistema em camadas, como o fizeram outros autores. O sistema consiste de um módulo chamado DMS (implementado em Pascal) que exerce as função de gerenciamento do sistema, e de vários gerenciadores de dados. Foi proposto um gerenciador de dados chamado OMS que permitiria o

mapeamento de certas estruturas de dados em outras. Parte desse gerenciador consiste de um compilador de esquemas chamado SchC (implementado em Pascal) que serve a dois propósitos: (1) a criação de uma descrição compilada da estrutura de dados a ser usada pelo OMS e, (2) a geração de rotinas de leitura/escrita de uma estrutura de dados arbitrária.

O DMS é um programa que controla o acesso, o relacionamento, e os atributos das representações de entidades. Uma entidade pode ser, por exemplo, uma célula. É possível que vários programas de aplicação (PAs) usem a mesma cópia do DMS simultaneamente, permitindo assim que o usuário possa utilizar vários PAs ao mesmo tempo durante uma sessão.

Um sistema protótipo foi desenvolvido consistindo de um programa de interface chamado VDI, o editor de textos EDT do VAX/VMS, e vários PAs. Essa experiência mostrou que, mesmo sem certas funções de gerenciamento de "alto nível" (p.ex., relações), o DMS facilita o desenvolvimento simultâneo de um circuito integrado por vários projetistas.

Algumas extensões no modelo hierárquico implementado pelo DMS são necessárias para acomodar, por exemplo, células parametrizadas quanto à estrutura. Há outras extensões, tais como:

- Iniciar uma transação englobando várias representações de várias entidades.

- o Nada foi discutido quanto à proteção e controle de acesso aos dados. Um mecanismo básico deve existir, porém, cada instalação ou grupo adota uma política adequada às suas necessidades. É necessário também implementar um mecanismo que permita que o sistema possa ser facilmente adaptável.

- o Seria interessante poder manter um histórico dos eventos ocorridos, em vários níveis de detalhe. Num nível mais alto, manter-se-ia um registro de quem utilizou um dado projeto. Em seguida, quais os programas de aplicação utilizados e se sua execução terminou normalmente ou não. E, num nível mais baixo, quais as transações efetuadas pelos programas de aplicação.

Quanto ao futuro, sugerimos que o uso de programação funcional e lógica para a representação e manipulação dos dados seja investigada em maior detalhe. O DMS poderia ser re-escrito e re-avaliado tendo à sua disposição um banco de dados relacional e dando maior atenção à sua portabilidade.

É importante que exista um utilitário capaz de reconstruir um projeto com arquivos corrompidos ou destruídos, e cujo "backup" seja muito antigo ou não disponível.

Quanto a "backups", atualmente é necessário utilizar um utilitário do sistema, p.ex. BACKUP, e armazenar o diretório todo onde se localiza o projeto. Porém, se no futuro for necessário utilizar essa cópia, os diretórios particulares podem estar inconsistentes. É preciso, portanto, armazenar também todos os diretórios particulares.

Não é possível armazenar um subconjunto de um projeto. Assim, um procedimento de "backup" mais adequado a esse sistema poderá ser muito útil.

Somos da opinião de que o compilador de esquemas poderia também ser usado para gerar interfaces com o usuário. Um dos problemas num sistema para VLSI PAC é o desenvolvimento de uma interface uniforme com todos os programas de aplicação. (Novamente temos o problema do uso de programas já existentes.) Ou seja, a partir de uma descrição de uma estrutura de dados acompanhada de uma especificação de como esses valores devem ser apresentados e obtidos do usuário, e quais os procedimentos que devem ser chamados quando certos dados são fornecidos, o compilador geraria um conjunto de rotinas que poderiam ser usados pelo programa de aplicação para obter os dados desejados do usuário.

Muitas pessoas tiveram participação direta ou indireta nesse trabalho, a começar pelos meus pais. Há também todos os meus professores da Graduação, e dentre eles eu gostaria de mencionar o prof. Nelson Machado, prof. Tomasz Kowaltowski, e, em especial, prof. Cláudio L. Lucchesi.

O prof. Hans Liesenberg foi de um auxílio inestimável, e se há algo de absurdo nessa tese, compartilho com ele a culpa. Mas não só dele. Parte deste trabalho foi desenvolvido na Universidade Católica de Leuven (KUL, Bélgica) durante o período de janeiro de 1984 a junho de 1985, como resultado de um convênio entre o CPqD/Telebrás e ESAT.

Devo muito a Luc Claesen, que me orientou durante aquele período, e à supervisão do prof. Hugo De Man (atualmente ambos trabalham no IMEC vzw). Evidentemente, o apoio do CPqD/Telebrás foi indispensável.

Aplausos para todos, por favor.

APÊNDICE A

EXEMPLO DE USO DO COMPILADOR DE ESQUEMAS

A.1 EXEMPLO 1

Este é o exemplo 3.1 apresentado no capítulo 3. O comando VMS/DCL

```
$ schema  example1
```

produziu os seguintes arquivos:

EXAMPLE1.LIS   Listagem compilada.

EXAMPLE1.INF   Este é um arquivo de informações sobre a compilação do esquema. Ele é utilizado quando um esquema é usado por outro através da diretiva INCLUDE. Por exemplo, ele indica se houve ou não erros de compilação, se as rotinas de leitura ou escrita forma geradas e em quais linguagens, e assim por diante.

DEXAMPLE1.PAS   Arquivo com as definições equivalentes em VAX Pascal.

DEXAMPLE1.INC Idem. A diferença com o anterior é que esse deve ser incluído e o anterior deve ser herdado usando o atributo INHERIT do VAX Pascal (observe o INHERIT no cabeçalho das rotinas de leitura/escrita).

SEXAMPLE1.PAS Essa é a rotina de escrita.

GEXAMPLE1.PAS Essa é a rotina de leitura.

A listagem de cada um desses arquivos (exceto a listagem compilada e o arquivo de informações) segue na ordem em que foram apresentados acima. Uma vez gerados, eles devem ser compilados ficando, assim, disponíveis para uso.

A.1.1 DEXAMPLE1.PAS

```
(**********************************************************************
  Definition file for schema EXAMPLE1
  Input from B:[ACDOLENC.TESE.EXAMPLES]EXAMPLE1.SCH;1
  Produced by Schema Compiler V1.1
**********************************************************************)

[ENVIRONMENT ('B:[ACDOLENC.TESE.EXAMPLES]DEXAMPLE1.PEN;'),
 INHERIT ('SCCLIB_VAXPASCAL:SCCPASDEF')]
 MODULE DEXAMPLE1;

%INCLUDE 'B:[ACDOLENC.TESE.EXAMPLES]DEXAMPLE1.INC;/NOLIST'

END.
```

A.1.2 DEXAMPLE1.INC

```
(**********************************************************************
  Include file for schema EXAMPLE1
  Input from B:[ACDOLENC.TESE.EXAMPLES]EXAMPLE1.SCH;1
  Produced by Schema Compiler V1.1
**********************************************************************)

CONST

max_ident_length = 31;

TYPE

ident_vc_type  = VARYING [31] OF CHAR;
ident_p_type   = ^ident_rec_type;
ident_rec_type =
  RECORD
    SCC_nr : SCC_integer;
    ident : ident_vc_type;
    left : ident_p_type;
    right : ident_p_type
  END;

VAR

ident_list : ident_p_type;
```

A.1.3 SEXAMPLE1.PAS

```
(**********************************************************************
  Save routines for schema EXAMPLE1
  Input from B:[ACDOLENC.TESE.EXAMPLES]EXAMPLE1.SCH;1
  Produced by Schema Compiler V1.1
**********************************************************************)

{=====================================================================}
[Environment ('B:[ACDOLENC.TESE.EXAMPLES]SEXAMPLE1.PEN;'),
 Inherit ('SCCLIB_VAXPASCAL:SCCPASDEF', 'SCCLIB_VAXPASCAL:SCCOUT',
 'B:[ACDOLENC.TESE.EXAMPLES]DEXAMPLE1.PEN;')]
 MODULE SEXAMPLE1;
{=====================================================================}

{---------------------------------------------------------------------}
[Global] PROCEDURE Save_EXAMPLE1
     (VAR SCC_filevar    : SCC_Text;
```

```
          SCC_fileformat : SCC_e_fileformat);
{------------------------------------------------------------------}

%INCLUDE 'SCCLIB_VAXPASCAL:SCCRPSAV.INC/NOLIST'

PROCEDURE SCC_St_ident_p_type (SCC_Item : ident_p_type); Forward;
PROCEDURE SCC_St_ident_rec_type (SCC_Item : ident_rec_type); Forward;
PROCEDURE SCC_St_ident_vc_type (SCC_Item : ident_vc_type); Forward;

{------------------------------------------------------------------}
  PROCEDURE SCC_St_ident_p_type;
{------------------------------------------------------------------}
BEGIN
   IF SCC_Item = NIL
      THEN SCC_OutWord (SCC_filevar, SCC_fileformat, 0)
      ELSE
        { Has structure pointed to already been stored ? }
        IF SCC_Item^.SCC_Nr = 0
           THEN
            BEGIN
              SCC_PtrIdx := SCC_PtrIdx + 1;
              SCC_Item^.SCC_Nr := SCC_PtrIdx;
              SCC_OutWord (SCC_filevar, SCC_fileformat, SCC_PtrIdx);
              SCC_St_ident_rec_type(SCC_Item^);
            END
           ELSE
              SCC_OutWord (SCC_filevar, SCC_fileformat,
                                 SCC_Item^.SCC_Nr)
END;

{------------------------------------------------------------------}
  PROCEDURE SCC_St_ident_rec_type;
{------------------------------------------------------------------}
BEGIN
   SCC_St_ident_vc_type(SCC_Item.ident);
   SCC_St_ident_p_type(SCC_Item.left);
   SCC_St_ident_p_type(SCC_Item.right);
END;

{------------------------------------------------------------------}
  PROCEDURE SCC_St_ident_vc_type;
{------------------------------------------------------------------}
VAR
   SCC_I : SCC_Integer;
BEGIN
   IF SCC_fileformat = SCC_Textual
      THEN WRITELN (SCC_filevar, LENGTH(SCC_Item))
      ELSE
       SCC_OutWord (SCC_filevar, SCC_fileformat, LENGTH(SCC_Item));
   SCC_I := 1;
   WHILE SCC_I <= LENGTH (SCC_Item) DO
    BEGIN
      IF SCC_fileformat = SCC_Textual
```

```
          THEN WRITELN (SCC_filevar, SCC_Item[SCC_I])
          ELSE
           SCC_OutByte (SCC_filevar,SCC_fileformat, SCC_Item[SCC_I]);
       SCC_I := SCC_I + 1;
     END;
 END;

 BEGIN
    SCC_InitRPtrIdx;
    SCC_St_ident_p_type (ident_list);
 END;

 END.
```

A.1.4 GEXAMPLE1.PAS

```
(*********************************************************************
  Get routines for schema EXAMPLE1
  Input from B:[ACDOLENC.TESE.EXAMPLES]EXAMPLE1.SCH;1
  Produced by Schema Compiler V1.1
*********************************************************************)

{===================================================================}
[Environment ('B:[ACDOLENC.TESE.EXAMPLES]GEXAMPLE1.PEN;'),
 Inherit ('SCCLIB_VAXPASCAL:SCCPASDEF', 'SCCLIB_VAXPASCAL:SCCIN',
 'SCCLIB_VAXPASCAL:SCCPTRGET',
 'B:[ACDOLENC.TESE.EXAMPLES]DEXAMPLE1.PEN;')]
 MODULE GEXAMPLE1;
{===================================================================}

{-------------------------------------------------------------------}
 [Global] PROCEDURE Get_EXAMPLE1
   (VAR SCC_filevar    : SCC_Text;
        SCC_fileformat : SCC_e_fileformat);
{-------------------------------------------------------------------}

%INCLUDE 'SCCLIB_VAXPASCAL:SCCRPGET.INC/NOLIST'

PROCEDURE SCC_Get_ident_p_type (VAR SCC_Item : ident_p_type); Forward;
PROCEDURE SCC_Get_ident_rec_type (VAR SCC_Item : ident_rec_type);
 Forward;
PROCEDURE SCC_Get_ident_vc_type (VAR SCC_Item: ident_vc_type);Forward;

{-------------------------------------------------------------------}
  PROCEDURE SCC_Get_ident_p_type;
{-------------------------------------------------------------------}
VAR
   SCC_Dptr : SCC_integer;
   SCC_Ptr  : SCC_Ref;
```

```
      FUNCTION New_ident_p_type: ident_p_type; External;

   BEGIN
      SCC_InWord (SCC_filevar, SCC_fileformat, SCC_Dptr);
      IF SCC_Dptr = 0
         THEN SCC_Item := NIL
         ELSE
           { Has structure pointed to already been stored ? }
           IF NOT SCC_RAlreadyLoaded (SCC_rpinfo, SCC_Dptr)
              THEN
               BEGIN
                 SCC_Item := New_ident_p_type;
                 SCC_Item^.SCC_Nr := 0;
                 SCC_StoreRPtr (SCC_rpinfo, SCC_Item::SCC_Ref);
                 SCC_Get_ident_rec_type (SCC_Item^);
               END
              ELSE
               BEGIN
                 SCC_Ptr  := SCC_GetRPtr (SCC_rpinfo, SCC_Dptr);
                 SCC_Item := SCC_Ptr::ident_p_type;
               END;
   END;

   {-------------------------------------------------------------------}
     PROCEDURE SCC_Get_ident_rec_type;
   {-------------------------------------------------------------------}
   BEGIN
      SCC_Get_ident_vc_type (SCC_Item.ident);
      SCC_Get_ident_p_type (SCC_Item.left);
      SCC_Get_ident_p_type (SCC_Item.right);
   END;

   {-------------------------------------------------------------------}
     PROCEDURE SCC_Get_ident_vc_type;
   {-------------------------------------------------------------------}
   VAR
      SCC_I  : SCC_Integer;
      SCC_ch : SCC_char;
   BEGIN
      IF SCC_fileformat = SCC_Textual
         THEN READLN (SCC_filevar, SCC_I)
         ELSE SCC_InWord (SCC_filevar, SCC_fileformat, SCC_I);
      SCC_Item := '';
      WHILE SCC_I > 0 DO
       BEGIN
         IF SCC_fileformat = SCC_Textual
            THEN READLN (SCC_filevar, SCC_ch)
            ELSE SCC_InByte (SCC_filevar, SCC_fileformat, SCC_ch);
         SCC_Item := SCC_Item + SCC_ch;
         SCC_I    := SCC_I - 1;
       END;
   END;
```

```
BEGIN
   SCC_InitRPtrInfo (SCC_rpinfo);
   SCC_Get_ident_p_type (ident_list);
   SCC_RstRPtrInfo (SCC_rpinfo);
END;

END.
```

[1] Hutchings, A.F., et al., "Integrated VLSI CAD Systems at Digital Equipment Corporation", Proc. of the 22nd Design Automation Conference.

[2] Barabino, G.P., et al., "A Module for Improving Data Access and Management in an Integrated CAD Environment", Proc. of the 22nd DA Conference, June 1985.

[3] Teorey, T.J., Fry, J.P., "Design of Database Structures", Prentice-Hall, Inc., 1982, Chap. 17.

[4] Elias, N.J., et al., "The ITT VLSI Design System: CAD Integration in a Multinational Environment", Proc. of the 22nd Design Automation Conference, June 1985.

[5] Claesen, L., De Man, H., "A database for VLSI design", relatorio interno, ESAT/K.U.Leuven, Belgica, out 1983.

[6] R.Katz, "Managing the Chip Design Database", IEEE Computer, dez 1983.

[7] Sidle, T.W., "Weakness of Commercial Data Base Management Systems in Engineering Applications", Proc. of the 17th Design Automation Conference, Minneapolis, June 1980.

[8] Zintl, G., "A CODASYL CAD Data Base System", Proc. of the 18th Design Automation Conference, Tennessee, June/July 1981.

[9] Walker, R., Thomas, D., "A Model of Design Representation and Synthesis", Proc. of the 22nd Design Automation Conference, Junho 1985, Las Vegas, Nevada.

[10] Bennett, J., "A Database Management System for Design Engineers", Proc. of the 19th Design Automation Conference, Junho 1982, Las Vegas, Nevada.

[11] Katz, R.H., "A Database Approach for Managing VLSI Design Data", Proc. of the 19th Design Automation Conference, Junho 1982, Las Vegas, Nevada.

[12] Haynie, M.N., "Tutorial: The Relational Data Model for Design Automation", Proc. of the 20th Design Automation Conference, Junho 1983, Miami Beach, Florida.

[13] Zara, R.V., Henke, D.R., "Building a Layered Database for Design Automation", Proc. of the 22nd Design Automation Conference, Junho 1985, Las Vegas, Nevada.

[14] Batory, D.S., Kim, W., "Modelling Concepts for VLSI CAD Objects", ACM Trans. on Database Systems, vol.10, num.3, Setembro 1985.

[15] Beetem, J.F., "Structured design of Electronic Systems using Isomorphic Multile Representations", Ph.D. dissertation IC Labs, Stanford University, Dezembro 1981.

[16] Ahdoot et al., K., "IBM FSD Chip Design Methodology", Proc. of the 20th Design Automation Conference, Junho 1983, Miami Beach, Florida.

[17] Elias et al., N.J., "The IC Module Compiler, A VLSI System Design Aid", Proc. of the 20th Design Automation Conference, Junho 1983, Miami Beach, Florida.

[18] Giambiasi et al., N., "An Adaptive and Evolutive Tool for Describing General Hierarchical Models, Based on Frames and Demons", Proc. of the 22nd Design Automation Conference, Junho 1985, Las Vegas, Nevada.

[19] Kim et al., W., "A Model of CAD Transactions", Proc. of VLDB 85, Stockholm.

[20] Gray, J., "Notes on Data Base Operating Systems", IBM Research Report: RJ2188, IBM Research, Calif., Fevereiro 1978.

[21] Dolenc, A., Severyns, R., "VDI: VLSI Design Interface Program", Reference Manual, IMEC/ESAT/K.U.Leuven, Belgium, 1st edition 1985.

[22] Millman-Halkias, "Integrated Electronics: Analog and Digital Circuits and Systems", McGraw-Hill Electrical and Eletronic Engineering Series, International Student Edition, 1972.

[23] Claesen, L., Dolenc, A., De Man, prof.H., "An Integrated DataBase Management System for the Complete CAD cycle of IC's", IMEC/ESAT, K.U.Leuven, Bélgica, Novembro 1984.

[24] Wilmot, R.B., "Foreign Keys Decrease Adaptability of DataBase Design", CACM vol.27 num.12, dez.1984.

[25] Dolenc, A., Claesen, L., "SchC UTILITIES: The Schema Compiler and the Convert Schema Utility", Manual de Usuário, 1a. edicão, IMEC, Leuven, Bélgica, julho de 1985.