Controle de Versões e Configurações em Ambientes de Desenvolvimento de Software

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pala Sra. **Eliane Zambon Victorelli** e aprovada pela Comissão Julgadora

Campinas, 19 de dezembro de 1990

Orientador: _____
Prof. Dr. Geovane Cayres Magalhães

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.

Aos meus pais e
ao Jairo

Agradecimentos

Quero deixar registrado meus mais sinceros agradecimentos ao Prof. Dr. Geovane Cayres Magalhães pela orientação eficiente e precisa, pelo incentivo recebido e por sua amizade e comprenssão, a Carmem Satie Hara por diversas discussões de idéias que contribuiram para este texto, e a Jairo Dias Junior, a Ignez Therezinha Zambon Victorelli e a Benedicto Adelino Victorelli que com seu incentivo, apoio e dedicação possibilitaram a realização deste trabalho.

Agradeço também à Capes, ao CNPq, à FAPESP e à Unicamp pelo suporte financeiro e ao projeto ETHOS pela oportunidade que me foi dada de utilizar os seus recursos de hardware e de software.

Sumário

O gerenciamento de versões e configurações é um assunto que vem sendo muito pesquisado nos últimos anos. Alguns ambientes de desenvolvimento de software oferecem suporte para esta tarefa, porém os mecanismos propostos abordam determinados aspectos do problema, deixando outros em aberto.

O objetivo deste trabalho é propor um modelo para o gerenciamento de versões e configurações, que seja adequado a vários ambientes. O enfoque foi dado para a distribuição e compartilhamento dos dados. A adaptação do modelo às características de cada ambiente é feita através da definição de alguns parâmetros.

Um gerador de gerenciadores de versões e configurações foi implementado utilizando-se, entre outras ferramentas, um protótipo de banco de dados orientado a objetos. A implementação teve como objetivo provar a viabilidade dos mecanismos propostos, bem como estudar as dificuldades envolvidas no mapeamento de suas características para o modelo de um banco de dados.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Ambientes de Desenvolvimento de Software

A crescente demanda por sistemas de software de alta qualidade e de baixo custo resultou na necessidade de aumento nos recursos de produção. A automação do processo de desenvolvimento de software através de ambientes próprios para esta finalidade é uma abordagem prática para aumentar a produtividade da mão de obra existente e a qualidade dos produtos gerados [Hoff85]. Um ambiente consiste em uma coleção de ferramentas de software e hardware que dão suporte à construção de sistemas [Dart87].

Os ambientes de desenvolvimento de software (ADS) evoluiram consideravelmente nos últimos anos. O primeiro passo neste sentido foi o suporte à fase de codificação. Surgiram, então, os ambientes de programação destinados às tarefas de construção de sistemas de pequeno porte, como edição e compilação.

Atualmente, o enfoque é dado aos ADS que procuram automatizar todas as tarefas compreendidas pelo ciclo de desenvolvimento de software, incluindo o suporte à construção de sistemas de grande porte e ao desenvolvimento distribuído [Dart87].

A automação destas tarefas implica na existência de inúmeras ferramentas, que devem ser integradas entre si de maneira que aparentem ser uma só. Para tanto, os dados do ambiente devem ser armazenados de forma centralizada e compatível com todas elas. Contudo, é comum surgir a necessidade de novas ferramentas ou de alteração nas existentes. Para satisfazer alterações nas suas especificações de requisitos, um ADS deve ser construído de maneira que possa ser estendido ou adaptado [Reed87].

Além disso, um ADS precisa oferecer meios para armazenar e gerenciar adequadamente o grande volume de dados que resulta da realização destas tarefas. No entanto, os dados manipulados em ADS não podem ser armazenados facilmente nos registros de formatos fixos suportados pelos sistemas de bancos de dados convencionais. A mesma dificuldade aparece também em outras aplicações conhecidas por não convencionais, como os sistemas de engenharia e de automação de escritórios. Este assunto é discutido em [Bato85, Katz86, Chou86, Hara90], e em muitos outros trabalhos.

Para atender às necessidades das novas aplicações, os sistemas de bancos de dados

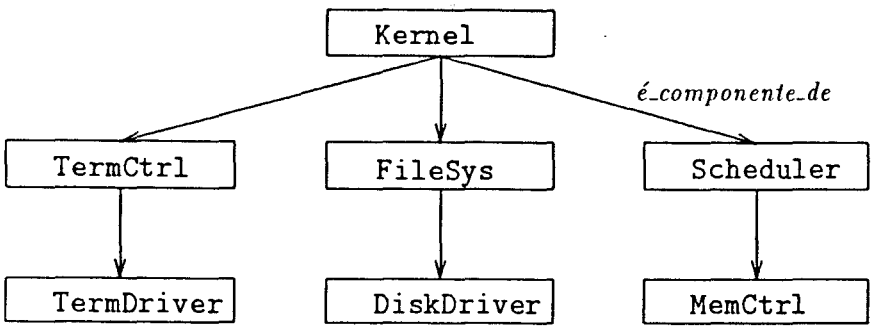

Figura 1.1: Uma hierarquia de objetos de desenvolvimento

foram estendidos com a noção de objeto complexo, que os habilita a armazenar e manipular mais adequadamente os dados envolvidos no desenvolvimento dos sistemas. Foram propostos também, modelos de dados semânticos para descrição de tipos especiais de relacionamentos entre os objetos.

1.2 Dados Manipulados em ADS

1.2.1 Composição e Equivalência

A modelagem de projetos de VLSI, proposta em [Katz86], pode ser adaptada para sistemas de software. Assim, é possível imaginar um banco de dados que armazena software como uma grande coleção de objetos, que juntos descrevem o sistema sendo desenvolvido. Objeto de desenvolvimento é um termo conveniente para identificar agregados de dados, sem se comprometer com uma implementação baseada em registros, tabelas ou arquivos. O agregado não tem um tamanho específico; ele pode ser grande como um subsistema inteiro ou tão pequeno quanto um procedimento.

Os sistemas de software podem ser descritos em diversas representações, desde as mais abstratas até as mais concretas, como especificação de requisitos, código fonte, código objeto, executável e muitas outras. No entanto, cada objeto de desenvolvimento contém dados somente de uma representação, que define o seu tipo. Os objetos de desenvolvimento em uma determinada representação podem ser identificados pelo nome seguido do tipo, como por exemplo A[código fonte] e B[executável].

Além disso, a descrição de um sistema é composta hierarquicamente, ou seja, um objeto de desenvolvimento pode ser descrito em termos de objetos componentes. A estrutura simplificada de um núcleo de sistema operacional é ilustrada na Figura 1.1. O sistema é dividido em três partes: o módulo de comunicação, o gerenciador de arquivos e o gerenciador de processos, que são representados pelos objetos de desenvolvimento

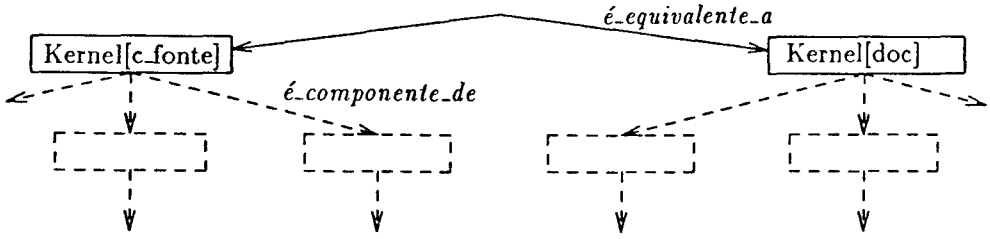

Figura 1.2: Relacionamento de equivalência

TermCtrl, FileSys e Scheduler, respectivamente. O objeto Kernel, utilizando os objetos componentes TermCtrl, FileSys, Scheduler, implementa o núcleo e fornece todas as rotinas chamadas pelos programas de aplicação e pela interface com o usuário. Cada um dos objetos do segundo nível da hierarquia usa por sua vez, os objetos que implementam as rotinas de interface com o hardware: a interface com o terminal (TermDriver), a interface com o disco (DiskDriver) e a interface com a memória física (MemCtrl).

Os objetos de desenvolvimento construídos a partir de outros, são chamados de *compostos*, enquanto que objetos *primitivos* são aqueles que não têm componentes. Um sistema é descrito por vários objetos de desenvolvimento *compostos* em representações diferentes, sendo que cada um deles é a raiz de uma hierarquia de composição. O sistema operacional poderia, por exemplo, ser descrito pelos objetos Kernel[especificação de requisitos], Kernel[código fonte], e Kernel[documentação].

É conveniente identificar os objetos que descrevem a mesma entidade em representações diferentes. Para isso, existem os relacionamentos de equivalência que fazem ligações entre as representações. A Figura 1.2 mostra que o objeto de desenvolvimento Kernel[código fonte] é equivalente ao objeto Kernel[documentação]. Outro exemplo é o da linguagem C, onde os relacionamentos de equivalência estão implícitos nos nomes dos arquivos: o código fonte do arquivo x.c corresponde ao código objeto que está no arquivo x.o e ao executável do arquivo x.

1.2.2 Versão e Configuração

Através dos relacionamentos de composição e equivalência, é possível modelar um sistema em um instante de tempo, porém não é suficiente representar apenas o estado atual de um sistema. É necessário modelar estados anteriores, bem como estados existentes paralelamente a outros. Muitas pesquisas apontam para a necessidade de representar o tempo em ADS [Lebl84, Estu84, Katz84, Klah86, Chou86]. Para representar a evolução dos sistemas foram introduzidos os conceitos de versão e configuração.

Um objeto de desenvolvimento não é estático, ele evolui com o tempo. Um estado

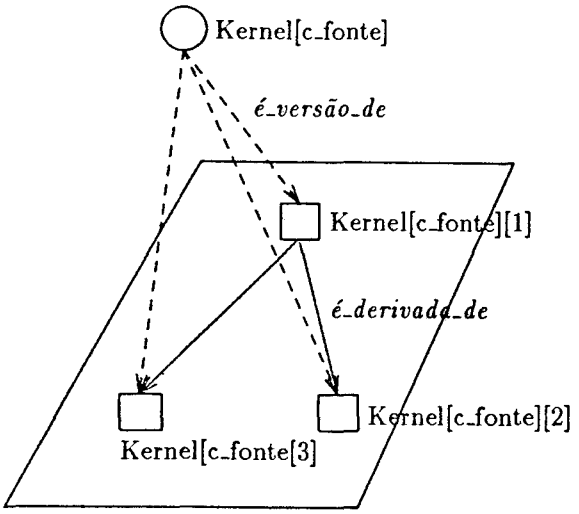

Figura 1.3: Plano de versões

particular de um objeto é chamado de versão, enquanto que o termo objeto de desenvolvimento se refere ao conjunto de todos os estados (ou versões) da entidade, em uma determinada representação. A informação é armazenada nas versões; o objeto de desenvolvimento é apenas a unidade que compreende as versões e o que elas têm em comum [Klah86]. Todas as versões associadas a um objeto de desenvolvimento têm com ele um relacionamento do tipo *é_versão_de*.

Durante o desenvolvimento de um objeto, uma nova versão pode ser derivada de uma versão antiga, e posteriormente a nova versão pode dar origem a outras. O objetivo de um histórico de versões é documentar as diversas etapas envolvidas na concepção de um objeto. A Figura 1.3 mostra um plano onde um histórico organiza as versões do objeto Kernel[código fonte] segundo os relacionamentos *é_versão_de* e *é_derivada_de*.

Se as versões criadas são sempre derivadas da última existente, o histórico de versões tem a forma linear. O histórico da Figura 1.4.a documenta que a versão TermCtrl[código fonte][3] é derivada de TermCtrl[código fonte][2], que é derivada de TermCtrl[código fonte][1].

Algumas vezes, mais de uma versão é derivada a partir de uma mesma versão antecessora. Elas são versões alternativas e representam maneiras diferentes de se desenvolver o mesmo objeto. Uma alternativa pode dar ênfase ao desempenho, enquanto a outra busca economia de espaço de armazenamento. Elas podem também surgir da necessidade de implementar um objeto para máquinas ou linguagens diferentes, e de muitas outras situações. Nestes casos, o histórico tem a forma de árvore, como é mostrado na Figura 1.4.b, onde as versões FileSys[código fonte][3] e FileSys[código fonte][4] são derivadas de

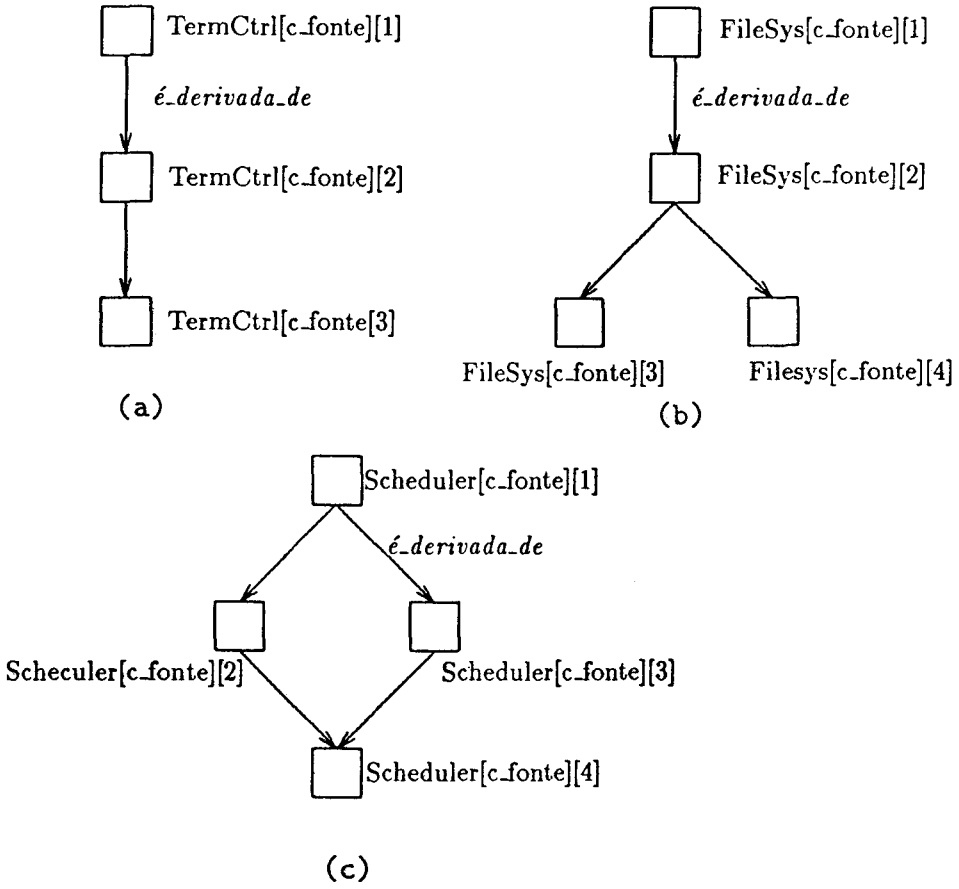

Figura 1.4: Histórico de versões: (a) linear, (b) árvore, (c) grafo acíclico

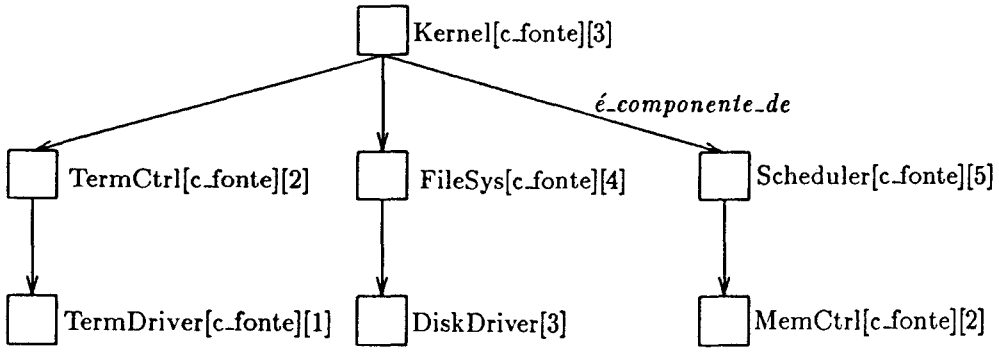

Figura 1.5: Plano de configuração

FileSys[código fonte][2].

Finalmente, um histórico de versões na forma de grafo acíclico permite representar que uma versão foi derivada a partir de duas ou mais antecessoras. O histórico da Figura 1.4.c representa o caso em que as alterações feitas nas versões Scheduler[código fonte][2] e Scheduler[código fonte][3] foram fundidas gerando a versão Scheduler[código fonte][4].

Diferentes versões do objeto Kernel[código fonte] surgem durante o desenvolvimento. E cada uma é composta com versões de TermCtrl[código fonte], de FileSys[código fonte] e de Scheduler[código fonte]. Como existem várias versões do objeto composto, bem como dos objetos componentes, é possível compor o sistema de inúmeras maneiras. Uma configuração está diretamente relacionada à composição hierárquica. Ela associa uma versão de um objeto composto a versões específicas de seus componentes.

Uma forma didática de visualizar as informações contidas em um ADS, é imaginá-las organizadas em três planos ortogonais: um para as versões, um para as configurações e outro para as equivalências.

A Figura 1.5 mostra um plano com uma configuração possível para o objeto Kernel[código fonte]. Cada objeto que participa da configuração tem suas versões dispostas dentro de seu próprio plano de versões, que é ortogonal a este.

Os relacionamentos de equivalência podem também ser estabelecidos entre versões em representações diferentes. Assim, as hierarquias de composição permanecem independentes, e não precisam ser isomorfas. Caso contrário, todas as hierarquias que representam o mesmo objeto seriam obrigadas a ter composições idênticas. Os relacionamentos de equivalência podem ser organizados em um plano ortogonal aos dois primeiros. Os diversos relacionamentos da versão TermCtrl[código fonte][2] estão ilustrados na Figura 1.6.

Deve ser notado ainda que, do ponto de vista operacional, as equivalências são restrições: a correspondência entre os objetos ou versões deve ser verificada. Algumas

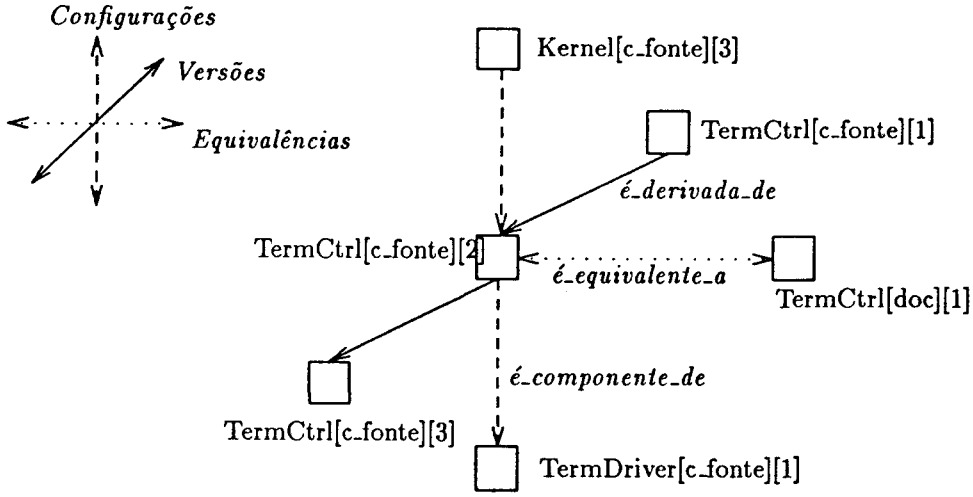

Figura 1.6: Relacionamentos de composição, derivação e equivalência de uma versão

vezes elas podem ser estabelecidas automaticamente, por exemplo: uma versão na representação código objeto é equivalente a uma versão na representação código fonte se a primeira é resultado da compilação da segunda. Outras vezes, a validade das equivalências é determinada pela execução de ferramentas de análise.

1.3 Gerenciadores de Versões e Configurações

Em geral, os sistemas de software de grande porte são desenvolvidos por muitas pessoas, sendo que cada uma é responsável por uma parte do sistema. Elas precisam compartilhar algumas versões, pois uma configuração do sistema envolve versões desenvolvidas por diferentes pessoas.

Além disso, durante o desenvolvimento do sistema é necessário *vê-lo* de diversas maneiras, isto é, composto com as versões adequadas a diferentes necessidades. Para a liberação do sistema ele deve ser composto com versões que já foram aprovadas, enquanto que na fase de testes são usadas as versões mais novas. E ainda, cada usuário tem uma necessidade diferente em relação à composição do sistema para o seu teste. Surgem assim, várias versões da configuração do sistema.

No entanto, o volume de dados aliado à complexidade dos relacionamentos faz com que a manutenção das informações a respeito das versões e das configurações se torne uma tarefa tediosa e sujeita a muitos erros, quando feita manualmente. A situação é

ainda pior, quando os dados do ADS se encontram distribuídos por uma rede.

Um ambiente que tenha como objetivo a produtividade deve livrar o usuário da manutenção dessas informações o máximo possível, para que ele possa se concentrar nas informações relativas ao sistema em si [Ditt88]. Para automatizar essa tarefa, um ADS deve contar com um gerenciador de versões e configurações.

1.4 Objetivo do Trabalho

O gerenciamento de versões e configurações é um assunto que vem sendo muito estudado nos últimos anos. Existem diversas propostas de mecanismos. Algumas são destinadas a um ambiente específico, enquanto outras deixam em aberto certos aspectos da questão. O objetivo deste trabalho é justamente propor um modelo para o controle de versões e configurações adequado a diversos ambientes

A apresentação do trabalho está contida em cinco capítulos: no primeiro foram apresentados os conceitos genéricos envolvidos no problema. No capítulo 2, é feita uma comparação das características de diversos gerenciadores. A proposta de um novo modelo é apresentada no capítulo 3, enquanto que o capítulo 4 descreve sua implementação. O capítulo 5 relata as conclusões e experiências obtidas ao longo do trabalho. Em apêndices são apresentados: um exemplo da utilização do modelo em um ambiente real, a sintaxe da linguagem de definição, a geração do esquema dos bancos de dados e as primitivas do modelo.

Capítulo 2

Gerenciadores de Versões e Configurações

Devido à importância do controle de versões e configurações em ADS, surgiram inúmeros modelos para o seu gerenciamento. Os diversos aspectos envolvidos neste tipo de controle tornam o problema bastante complexo e, conseqüentemente, não há um consenso na melhor forma de abordá-lo.

Neste capítulo, é feito um levantamento dos problemas envolvidos no controle de versões e das maneiras encontradas para resolvê-los nos diversos ambientes. Não se procura eleger o melhor modelo, o objetivo é simplesmente mostrar as diversas abordagens existentes.

2.1 Armazenamento de Versões

O armazenamento de várias versões de um mesmo objeto, embora muito conveniente, é dispendioso em termos de espaço. Portanto, um dos aspectos mais explorados nas pesquisas relacionadas com versões é o armazenamento.

2.1.1 Modo de Armazenamento

Como os objetos utilizados no desenvolvimento de software são formados por diversos componentes organizados de forma hierárquica, a sua manipulação levanta os seguintes pontos quanto ao armazenamento:

Acesso orientado a versão: Os objetos podem ser tratados como uma entidade única, e portanto seria interessante que os diversos componentes de uma versão estivessem agrupados para assegurar o acesso rápido a ela.

Acesso orientado a registro: Para que as diversas versões de um objeto pudessem ser acessadas rapidamente, seria necessário um agrupamento das versões de um mesmo objeto.

Redundância mínima: As versões deveriam ser armazenadas com um mínimo de redundância, para que o espaço de armazenamento fosse usado eficientemente.

É óbvio que os três objetivos não podem ser atingidos simultaneamente. Em geral, as propostas para armazenamento de versões privilegiam um deles, em detrimento dos outros.

Orientação por arquivos: A solução mais imediata para o armazenamento de versões, é a utilização de um arquivo para cada versão. Sempre que ocorre uma alteração, é criado um novo arquivo. A desvantagem deste método é que o nível de redundância é muito elevado. Embora a implementação seja fácil, ela não utiliza eficientemente o espaço de armazenamento. Por outro lado, o acesso aos registros de uma versão é muito rápido, pois eles permanecem agrupados.

Orientação por páginas: Em [Lori77] é proposto um mecanismo chamado *páginas sombras*, em que as versões são divididas em páginas. Quando uma página tem um de seus registros alterado, ela é duplicada com a nova versão do registro. Uma versão específica é representada por um mapa de páginas, e o banco de dados do ambiente é uma coleção de mapas de páginas.

Nesta solução a redundância ainda é muito grande. E a implementação do mecanismo pode ser difícil, pois para sistemas muito grandes, os mapas podem crescer muito e para que se tenha um tempo de resposta razoável, os mapas devem ficar em memória. Isto pode resultar em problemas de espaço, exigindo a implementação de um gerenciador de memória.

Orientação por registros: Dois tipos de mecanismos orientados a registros são conhecidos:

Árvore B: No método de armazenamento proposto em [Katz84], as versões correntes de todos os registros são mantidas agrupadas em um arquivo, enquanto todas as versões antigas dos registros estão contidas em um outro arquivo. Em um terceiro arquivo fica armazenado o índice (árvore B), que auxilia o acesso às diversas versões dos registros. O índice é baseado em identificadores únicos (*surrogates*) associados aos registros lógicos e cada folha da árvore B contém os endereços dos registros físicos que armazenam as versões de um registro lógico.

Arquivos diferenciais ou deltas: Este método usa uma técnica de compactação de dados, que consiste em armazenar uma versão básica na íntegra, enquanto as outras versões são armazenadas como diferenças a partir dela. Assim, para se recuperar uma dada versão, é necessário partir da versão básica e percorrer os arquivos delta seqüencialmente até atingir a versão desejada. Durante o percurso, os registros que sofreram modificações vão sendo atualizados conforme indicado nos arquivos diferenciais. O SCCS [Bona81] mantém na íntegra a versão original, enquanto o RCS [Tich85] mantém agrupados os registros da versão corrente. Uma estratégia diferente é adotado pelo Gypsy

[Cohe88]. Ele permite que o usuário especifique quais versões devem ser mantidas integralmente.

Orientação por objetos: O problema com algumas das abordagens anteriores é identificar o que constitui um registro. Em geral elas se referem a uma linha de código. Este não é um suporte adequado em muitos casos. Como seriam determinadas as diferenças entre arquivos binários? Em [Katz86] é proposta uma compressão de dados a nível de objetos através do uso de configurações. Uma configuração representa uma versão do sistema como um todo, e uma versão de um objeto componente pode ser compartilhada por muitas versões de objetos compostos.

2.1.2 Sistemas de Armazenamento

Além de adotar um método de armazenamento, um gerenciador de versões e configurações tem que contar com um sistema de armazenamento de dados para guardar suas informações.

Alguns gerenciadores como o **SCCS** e o **RCS** são implementados *em cima* do sistema operacional Unix. As versões são armazenadas em arquivos diferenciais. Quando um usuário quer uma versão, ela deve ser reconstruída a partir dos arquivos diferenciais correspondentes. Como não existe suporte para controle de configurações, ele tem que ser feito através de outras ferramentas como o **Make** [Feld79]. Este por sua vez, armazena os dados em *makefiles*, que são arquivos que contêm as propriedades dos objetos. A falta de integração do controle de versões e de configurações torna o mecanismo incômodo para o usuário.

Outra abordagem para o armazenamento é a *integração* do controle de versões com o sistema operacional, que é feita no **DSEE** [Lebl84] e no **Gypsy** . As vantagens da integração apontadas por [Cohe88] são:

- Os nomes das versões e arquivos são integrados. O nome de uma versão é simplesmente um nome de arquivo, estendido por um seletor de versão. O nome do arquivo inclui um caminho de diretório, que define o seu local de armazenamento.
- Um programa ou usuário não precisa extrair ou reconstruir a versão para poder ler. A versão pode ser recuperada diretamente pelo sistema operacional.
- A reconstituição e o agrupamento de versões armazenadas em arquivos diferenciais pode ser feito de modo transparente quando necessário. Quando um programa abre um arquivo, ele não precisa saber se o arquivo tem as versões controladas.

Embora o controle de versões de ambos os gerenciadores sejam integrados ao sistema operacional, o **DSEE** usa um sistema de gerenciamento de banco de dados distribuído para armazenar dados históricos. Ao passo que o **Gypsy** é construído em uma extensão do Unix, que provê mecanismos para que os usuários ampliem as definições de arquivos e diretórios.

Um dos principais requisitos de um ambiente de desenvolvimento de software é a integração. O usuário deve ver o ambiente como uma ferramenta única e todos os mecanismos que implementam a funcionalidade do ambiente devem ser encapsulados por esta

ferramenta. O uso de uma arquitetura com três camadas é uma maneira de se conseguir integração em um ambiente. A camada superior e a inferior representam respectivamente uma interface e uma base de dados compartilhada. Na camada intermediária fica o conjunto de ferramentas, que é integrado ao ambiente apenas através de comunicações verticais com a base de dados e a interface [Hara88].

O **Adele** [Belk87, Estu84] e os modelos apresentados em [Chou86] e [Katz87a] usam uma base de dados como o núcleo do sistema. Assim provêm extensibilidade ao ambiente, pois toda comunicação entre as ferramentas é feita por intermédio da base de dados, evitando a comunicação horizontal entre elas. Além disso, o armazenamento das informações geradas nas diferentes fases do ciclo de desenvolvimento de um projeto em um mesmo meio, torna a transição entre as fases mais natural e facilita a manutenção de consistência destas informações.

A maneira encontrada pelo **shape** [Mahl88] para conseguir integração foi usar uma interface generalizada, em cima da qual são construídos os comandos de controle de versões e configurações. A interface é um sistema de arquivos com atributos, que permite ligar qualquer atributo aos objetos, introduzindo um esquema generalizado para sua identificação. Através do uso da interface generalizada consegue-se uma abstração do sistema de armazenamento básico usado.

2.2 Compartilhamento das Versões

Em geral, os sistemas de software de grande porte são desenvolvidos por muitas pessoas, que precisam compartilhar alguns objetos. Portanto, o ambiente deve distribuir as informações de forma adequada e fornecer mecanismos que suportem o compartilhamento.

2.2.1 Distribuição

Apesar de existir muita diversificação quanto ao sistema de armazenamento usado, a maioria dos ambientes distribui as versões entre um sistema público e vários sistemas privados. Os dados de projeto compartilhados e os de controle de projeto são gerenciados pelo sistema público. Paralelamente, cada sistema privado gerencia os dados particulares de um usuário.

No entanto, em um ADS é comum usuários pertencentes a um mesmo projeto, precisarem compartilhar dados, aos quais os outros usuários ainda não devem ter acesso. Uma organização em que o usuário tem acesso às suas versões particulares, e às versões que tenham sido liberadas para uso público, normalmente não atende às necessidades de compartilhamento do grupo. Para que um usuário possa liberar suas versões para uso público, ele deve testá-las. O teste pode envolver versões de outros integrantes do mesmo grupo, que também não podem ser liberadas antes de serem testadas. Outra ocasião em que há necessidade de compartilhamento entre os integrantes de um grupo é quando dois usuários desenvolvem um objeto em conjunto.

Em [Chou86] é apresentada uma organização na forma de hierarquia de banco de dados, para atender a necessidade de um espaço para armazenamento das informações pertencentes ao grupo. As versões podem ser armazenadas no banco de dados geral,

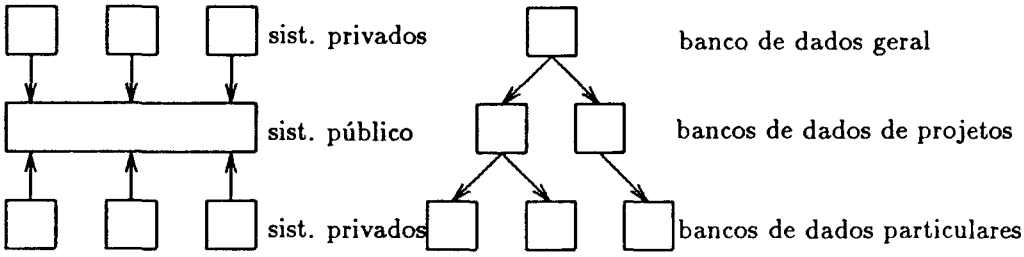

Figura 2.1: A organização tradicional de versões e uma hierarquia de bancos de dados

em bancos de dados associado a cada projeto, ou em bancos de dados particulares. Consegue-se desta forma uma maior distribuição dos dados do ambiente. A Figura 2.1 contrasta a organização tradicional das versões e a organização em uma hierarquia de banco de dados.

2.2.2 Controle de acesso

O controle de acesso aos objetos manipulados pelo gerenciador de versões e configurações é normalmente baseado no tipo de proteção oferecido pelo sistema de armazenamento usado pelo ambiente.

Alguns gerenciadores como o **DSEE** , **Cedar** [Giff88], **shape** e o **Gypsy** implementam um mecanismo baseado na lista de acesso fornecida pelo sistema operacional. Cada objeto mantém uma lista de usuários com direitos sobre ele. Por exemplo, no **Gypsy** o criador de um objeto é o seu dono e pode conceder o acesso a um usuário específico, a um grupo deles ou a qualquer usuário. Ele determina quem pode criar ramificações na estrutura de derivação de versões, quem pode dar nomes simbólicos às ramificações e quem pode definir atributos para as versões daquele objeto. O criador de uma ramificação, por sua vez, é o seu dono e determina os tipos de acesso dos outros usuários a cada versão naquela ramificação. Finalmente, quem define um atributo determina quem pode atribuir valores a ele.

Freqüentemente, os bancos de dados implementam um mecanismo baseado em uma lista de direitos (capacidades, visões). Um domínio é definido para cada usuário como um conjunto de objetos que ele pode acessar. Os modelos apresentados em [Chou86] e em [Katz87a] usam este tipo de mecanismo.

Segundo o modelo de Chou [Chou86], cada usuário tem acesso às versões que estão no seu banco de dados particular, no banco de dados do projeto do qual ele é integrante, ou no geral. Os direitos do usuário sobre as versões às quais ele tem acesso são estabelecidos de acordo com o tipo de banco de dados em que elas estão armazenadas. Por exemplo, o usuário pode atualizar uma versão armazenada no seu banco de dados particular, mas

não pode fazer o mesmo em versões armazenadas no banco de dados de projeto ou no geral.

Um mecanismo de proteção mais flexível foi implementado no **Adele** . O mecanismo usa tanto listas de acesso, como listas de direitos. Os objetos são protegidos por listas de acesso e usuários são controlados por listas de direitos. Esta abordagem considera um direito de acesso como uma síntese de dois conjuntos de privilégios. Por exemplo, o direito de um usuário U dirigir um carro C supõe que U tem licença para dirigir (definido na lista de direitos de U) e que C pode ser usado por U (definido na lista de acesso de C).

2.2.3 Ciclo de Vida

Existem gerenciadores que definem o ciclo de vida dos objetos e organizam as versões segundo este ciclo de vida. Uma versão muda de estado conforme a sua evolução, e o estado em que ela se encontra define as suas capacidades.

Segundo Katz [Katz84] uma versão inicialmente está *em_progresso*. Todos os usuários do ambiente podem modificar a versão até que ela esteja pronta para ser *liberada*. Eles podem criar *alternativas*, onde são feitas as alterações. Neste modelo, o termo alternativa se refere às variantes da versão *em_progresso*. As alternativas podem ser incorporadas à versão *em_progresso* e, quando ela está pronta para liberação, o responsável a torna *efetiva* para testes e distribuição interna. Uma versão *efetiva* não pode ser atualizada sem que seu responsável a torne mais uma vez uma versão *em_progresso*. Uma vez aprovadas, as versões *efetivas* se tornam *liberadas*; estas podem ser *arquivadas* e depois *restauradas* quando for necessário. O ciclo recomeça quando uma nova versão *em_progresso* é derivada da última versão *liberada*.

Ciclos de vida parecidos são descritos em [Lebl84], [Chou86], [Katz87a], [Ditt88] e [Mahl88]. O modelo apresentado em [Klah86] é mais genérico. Ele permite a definição de partições e classes para agrupar as versões de acordo com o seu estado. Desta forma, cada ambiente pode adotar o ciclo de vida mais adequado.

2.2.4 Controle de Ações Concorrente

Como normalmente é permitido o acesso a uma versão por mais de um usuário, é preciso controlar a concorrência para evitar conflitos. Por exemplo, quando dois usuários fazem atualizações simultâneas na mesma versão, as atualizações feitas por um deles podem ser perdidas, se não houver controle.

Uma solução para o problema é exigir que todas as versões compartilhadas sejam congeladas. Se um usuário quiser modificar uma versão compartilhada, ele realiza um *check-out*. A operação implica na criação de uma nova versão no banco de dados do usuário, que é uma cópia da outra. As alterações são feitas na nova versão e quando as modificações estiverem prontas, o usuário pode colocá-la no banco de dados geral através de um *check-in*. A partir de então, outros usuários têm acesso a nova versão, mas ninguém pode fazer outras modificações diretamente nela. Assim, quando dois usuários fazem modificações simultaneamente em uma mesma versão, são criadas duas novas

versões paralelas derivadas da primeira, resultando em uma ramificação na estrutura de derivação. Esta é a solução adotada no **Cedar** [Giff88] e nos modelos descritos em [Chou86] e [Katz87a], e ilustrada na Figura 2.2.a.

O mesmo mecanismo de criação de uma nova versão para cada atualização é usado no **Gypsy** e no **DSEE**. No entanto, eles controlam a criação de ramificações na estrutura de derivação, como se pode ver na Figura 2.2.b. Novas ramificações não devem se originar simplesmente de atualizações simultâneas da mesma versão. Existe um comando para criar ramificações, e nem sempre todos os usuários que têm permissão para criar versões de um objeto, têm permissão para criar ramificações na estrutura de derivação do mesmo objeto. Quando um usuário quer fazer modificações em uma versão, ele faz um *check-out* na versão. A partir de então, a versão original fica bloqueada, impedindo que outros usuários façam modificações criando outras versões paralelas. Para que dois usuários possam fazer modificações paralelas em uma mesma versão, alguém deve antes criar uma nova ramificação. E então, cada usuário faz um *check-out* em uma ramificação.

Segundo o modelo apresentado em [Vict89] os usuários podem criar livremente versões paralelas em seus bancos de dados particulares, mas no banco de dados geral elas são organizadas de forma linear. Quando um usuário transfere uma versão particular para o banco de dados geral, todas as versões de outros usuários que eram paralelas a ela, passam a ser suas sucessoras.

Uma abordagem diferente, feita no **Adele** e no **shape**, é ilustrada na Figura 2.2.c. As versões compartilhadas podem ser modificadas diretamente, sem a criação de novas versões. No entanto, o usuário deve fazer um *check-out* na versão para que outros não possam modificá-la ao mesmo tempo. Assim, criação de uma nova versão deve ser feita explicitamente pelo usuário; ela não ocorre através de atualizações.

2.3 Controle de Versões

2.3.1 Identificação

Quando uma versão é criada, ela deve receber uma identificação que a diferencie das outras versões existentes no ambiente, para que ela possa ser recuperada futuramente. Existem várias propostas de mecanismos para identificação de versões.

Muitos ambientes usam identificadores que, na sua forma mais simples, consistem no nome de um objeto seguido por um par de números separados por um ponto; por exemplo: 1.3 ou 2.45. O primeiro número indica a versão básica e o segundo indica a revisão. Revisões sucessivas e paralelas são distinguidas umas das outras pela forma de seus identificadores.

Na criação de uma nova revisão sucessiva é incrementado o número mais a direita do identificador da versão que deu origem a ela. A revisão que sucede a 2.2 recebe o identificador 2.3. O número mais a esquerda do identificador é alterado quando uma nova versão é criada. Quando se cria uma versão paralela é acrescentado outro "n.1" ao número da versão original, indicando que esta é a n-ésima versão paralela derivada daquela versão. Então, as versões paralelas originadas da versão 1.2, recebem identificadores 1.2.1.1, 1.2.2.1 e assim sucessivamente. Desta forma, o identificador de uma versão

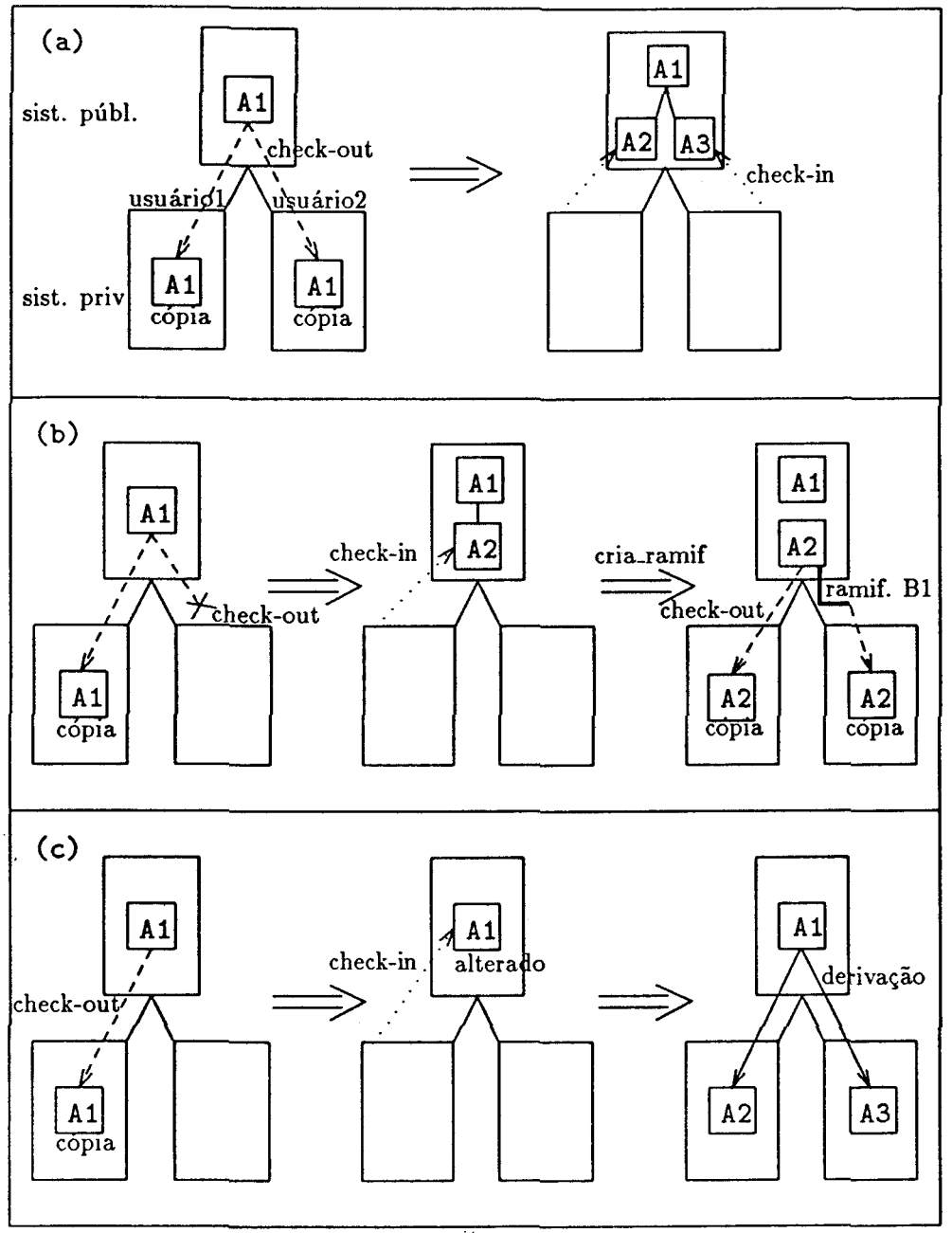

Figura 2.2: Mecanismos de controle de alterações

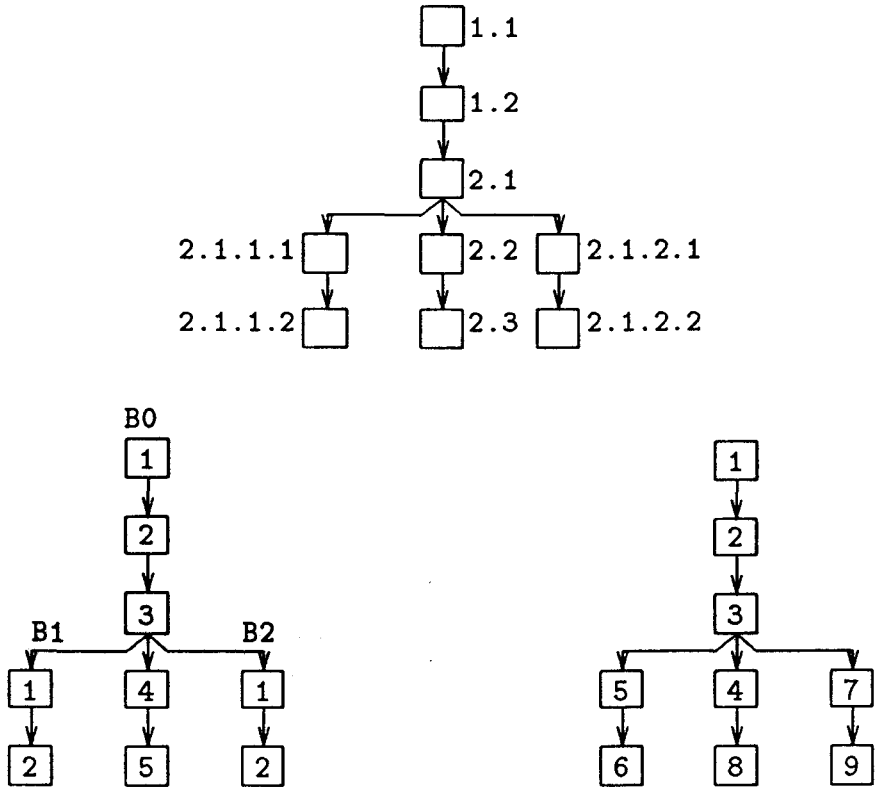

Figura 2.3: Mecanismos de identificação de versões

denota exatamente o caminho desde a raiz da árvore de derivação até ela.

No **SCCS** e no **RCS** a distinção entre versão e revisão é arbitrária; fica a cargo do usuário dizer quando uma alteração é mais significativa do que outras. O **Adele** torna esta distinção mais formal, especificando que todas as revisões de uma versão requerem as mesmas facilidades, ou seja, têm interfaces idênticas.

Outra forma de identificação é encontrada no **Gypsy**. As versões dentro de uma ramificação tem uma numeração seqüencial. A cada ramificação criada é dada outra identificação, que tem a forma B0, B1, B2, etc. As versões de um mesmo objeto são distinguidas por um par do tipo @<ramificação>@<versão> adicionado ao nome do arquivo, como por exemplo: /usr/glu/ship.c@B3@1. Um identificador não especifica a posição da versão na estrutura de derivação, mas existem métodos que fornecem informações a respeito das ramificações da estrutura de cada objeto.

Uma numeração seqüencial e independente das ramificações é proposta em [Chou86]. No entanto, além do nome do objeto e do número da versão, o nome do banco de dados também é necessário para identificar a versão, pois segundo este modelo elas estão distribuídas em uma hierarquia de banco de dados.

Outros gerenciadores propostos em [Mahl88], [Lebl84], [Belk87] e [Vict89] usam mecanismos de identificação semelhantes, com pequenas variações.

A Figura 2.3 mostra uma estrutura de derivação, com as identificações que as versões receberiam em diversos ambientes.

2.3.2 Seleção de Versões

Além dos identificadores, existem diversas outras maneiras de se distinguir as versões. Algumas vezes é necessário selecionar um subconjunto de versões que satisfaçam a alguma condição. Outras vezes se deseja uma única versão dentro de um subconjunto.

A maneira de especificar o conjunto ou de escolher uma versão, é diferente em cada gerenciador. Um método de seleção que entenda os padrões usados em comandos *shell* pode ser especialmente útil, quando o controle de versões é integrado ao sistema operacional. Um dos padrões Unix usado é o *, que pode ser substituido por qualquer string, inclusive pelo identificador da versão. Predicados baseados nos atributos identificam as versões através de suas propriedades (ex: data de criação > 06.89). Caso as versões estejam armazenadas diretamente no sistema de arquivos em diretórios diferentes, um caminho de diretório pode ser suficiente para identificar uma versão. É possível ainda, escolher versões através de rótulos pré-definidos, como ÚLTIMA_LIBERADA, ou CORRENTE. Alguns gerenciadores têm também mecanismos para definição de nomes simbólicos para as versões.

No **Adele** , um objeto é denotado por nomes da forma <domínio> <tipo> <predicado>, onde um domínio é o conjunto de todas as representações de um objeto ou um conjunto de objetos dos quais outro depende. Por exemplo, o conjunto dos objetos dos quais F depende é indicado por F*. O <tipo> especifica uma das representações do domínio. Assim, o nome A.implementação refere-se a todas as implementações do domínio A. E cada predicado é uma conjunção de predicados atômicos, que constam de <nome de atributo> <operador> <valor>. Por exemplo, B.especificação (sistema =

unix, data > 10-88, estado = distribuído) seleciona todas as versões da especificação de B criadas depois de outubro de 88 para o sistema Unix, que foram distribuídas para os clientes.

A regra de seleção usada pelo **shape** é uma seqüência de cláusulas, separadas por ponto-e-vírgula, representando expressões lógicas do tipo s por ponto-e-vírgula, representando expressões lógicas do tipo *or*. Cada cláusula consiste de uma seqüência de predicados, separados por vírgulas, representando expressões lógicas do tipo *and*. Uma regra de seleção tem sucesso, se uma de suas cláusulas identificar uma única versão de um objeto. Um exemplo de regra de seleção é.

```
#%Selection_rules
rule_1:
   afs_def.h attr(version,8.22);
   afs_hparse.c attr(version,8.17);
   *.c attr (autor, andy),
   attr (state, busy);
   *.c attrge (state, published),
   attrmax (version).
```

Se a regra rule_1 for usada para o objeto afs_def.h a versão selecionada será a 8.22. Caso ela seja usada para encontrar uma versão de um objeto cujo nome tenha o sufixo .c, e que não seja afs_hparse.c, vai resultar na versão criada por Andy e que esteja ativa. A versão publicada com o maior número será a escolhida, se não existir nenhuma versão que satisfaça as condições das cláusulas anteriores.

Além do diretório onde se encontram as versões do objeto, no **DSEE** são usados números e rótulos pré-definidos. Os rótulos LAST_RELEASE, RESERVED, REPLACED selecionam respectivamente as versões liberadas mais recentemente, as reservadas pelo usuário e as modificadas por ele. Através de rótulos é possível também especificar uma ramificação onde deve estar a versão selecionada.

O **Gypsy** armazena todas as versões de um objeto em um mesmo diretório, chamado grupo de versões. Como já foi mencionado na Secção 2.3.1, este gerenciador identifica a versão pelo caminho do diretório seguido de um par na forma @<ramificação>@<versão>. Um * no lugar da especificação da ramificação seleciona uma determinada versão em todas as ramificações. Ao passo que, se ele estiver no lugar da versão, todas as versões de uma ramificação são escolhidas. É permitido também usar predicados para especificar tanto a ramificação como a versão. Além disso, o seletor **latest** pode ser usado para selecionar a versão mais recente de um conjunto. Existe ainda um mecanismo para dar nomes simbólicos às versões. É possível, por exemplo, criar dinamicamente um rótulo mnemônico para identificar a versão da ramificação B3, mais recente e que tenha sido testada, através do comando:

```
in /usr/glu/ship.c@B3, best is latest [status = tested]
```

O rótulo criado, que neste caso é *best*, pode ser usado posteriormente para selecionar uma versão. Alguns exemplos de seleções de versões do objeto armazenado em /usr/glu/ship.c, permitidos pelo ambiente são:

```
/usr/glu/ship.c@B3@*;
/usr/glu/ship.c@*[branch]@latest[status = tested]
/usr/glu/ship.c@B3@best
```

No primeiro caso, são selecionadas todas as versões da ramificação B3. As últimas versões testadas de todas as ramificações são selecionadas no segundo exemplo. E no último, é escolhida a versão identificada pelo nome simbólico *best* na ramificação B3.

2.4 Controle de Configuração

2.4.1 Modelos de Interconexão

A composição hierárquica de um sistema de software é formada a partir das relações existentes entre seus componentes. O *modelo do sistema* representa a estrutura do software, isto é, seus componentes e os relacionamentos entre eles (é_usado_por, é_incluido_em, depende_de etc). Por outro lado um *modelo de interconexão* (MI) é uma tupla, que consiste em um conjunto de objetos que são os componentes da interconexão e um conjunto de relações que definem os tipos de interconexões que existem entre os objetos:

$$\text{MI} = (\text{[objetos]}, \text{[relações]})$$

Os mecanismos para gerenciamento da evolução de sistemas devem, portanto, basear-se em um *modelo de interconexão* para determinar os *modelos de sistemas*. Segundo [Perr87a] os modelos de interconexão usados atualmente são classificados em:

Modelo de interconexão de unidade: São definidos os relacionamentos entre várias unidades que compreendem um sistema de software. O relacionamento básico é o de dependência, e as unidades são representadas por arquivos, módulos ou objetos. Por exemplo: a unidade A depende das unidades B, C e D. Portanto, um modelo de interconexão de unidade tem um único tipo de objeto e um único tipo de relação:

$$\text{MI de unidade} = (\text{[unidade]},\text{[depende\_de]})$$

Este modelo é útil porque suporta e encoraja a construção modular de software. Ele captura a noção de encapsulamento, isolando as diversas características do sistema através da divisão do programa em partes. Ele permite também, que exista uma única cópia de cada unidade, evitando problemas no desenvolvimento e na manutenção do software.

Modelo de interconexão sintática: São descritas relações entre elementos sintáticos das linguagens de programação. Em termos genéricos o modelo consiste em:

$$\text{MI sintática} = (\text{[funções, procedimentos, tipos, variáveis, ...]},$$
$$\text{[é\_usado\_em, é\_chamado\_por, é\_parâmetro\_de, ...]})$$

O modelo de interconexão de unidades é incluido por este, e portanto eles têm a mesma utilidade. Mas este modelo oferece também a possibilidade de verificação da consistência sintática das relações existentes entre os componentes do sistema de software. É possível obter várias informações a respeito do sistema a partir de suas interconexões sintáticas, mas não se consegue nenhuma informação a respeito da sua semântica.

Modelo de interconexão semântica: O conjunto de objetos usados no modelo de interconexão sintática é aumentado pelos predicados. Por meio dos predicados, os programadores podem expressar o que têm em mente, quando criam um objeto e quando eles usam objetos para construir o sistema. Eles definem as pré-condições que devem ser satisfeitas antes de se executar uma seqüência de código. Eles especificam também as pós-condições, resultantes da execução, se as pré-condições forem satisfeitas.

MI semântica = ([funções, procedimentos, tipos, variáveis, ... predicados]
[é_usado_em, é_chamado_por, é_parâmetro_de, ... satisfaz])

Com o uso destes predicados é possível verificar a consistência do sistema de software com respeito à sua semântica. Eles podem ser usados também para formalizar a compatibilidade entre as versões, isto é, para determinar quando uma versão que faz parte do sistema pode ser substituida por outra, sem problemas.

2.4.2 Modelo de Sistema

Em um modelo de sistema existem os componentes de entrada, a partir dos quais o sistema é construído, e os componentes de saída, que são os derivados durante a construção do sistema.

As relações de um modelo de sistema podem ser representadas por *dependências fonte*, que ocorrem entre os componentes de entrada do sistema. Mas elas podem também, ser representadas por *dependências de processamento*, que ocorrem entre uma saída e uma entrada de uma etapa de construção do sistema [Schw89]. Quando se diz que inc.h é incluido em parse.c, o sistema está sendo modelado em termos de dependências fonte. E quando se diz que parse.o é derivado de parse.c e de inc.h, o modelo do sistema é composto por dependências de processamento. É importante ressaltar que dependência fonte não tem o significado de dependência entre objetos na representação código fonte. Ela é definida em relação aos componentes a partir dos quais será construído o sistema, independentemente de sua representação.

As pessoas que desenvolvem software, geralmente, preferem trabalhar com dependências fonte, pois elas correspondem aos conceitos das linguagens fonte. No entanto, uma ferramenta de construção de sistemas precisa conhecer as dependências de processamento.

2.4.3 Configuração

Como existem várias versões de cada componente, uma versão do sistema tem inúmeras possibilidades para a sua composição. Uma configuração é uma instância do modelo de sistema. Ela especifica uma versão para cada um dos seus componentes.

Configurações Baseadas em Dependências de Processamento

Muitos ambientes usam um esquema semelhante ao proposto pelo Make em [Feld79], para construção de configuração. Entre eles está o DSEE e o shape . O Make mantém as dependências de processamento entre os componentes de um sistema de software. A execução do Make após alterações nos componentes do sistema, implica na execução dos comandos necessários para garantir que o produto final seja construído a partir de componentes consistentes.

O modelo de sistema e os comandos necessários para reconstrução de cada componente são armazenados em arquivos chamados *makefiles*. Em um *makefile*, um conjunto de dependências é declarado em uma linha, onde os nomes dos dependentes são separados da lista de dependências por dois pontos. Ou seja, cada nome da esquerda depende de todos os nomes que se encontram à direita dos dois pontos. Uma linha de dependências pode ser seguida por uma ou mais linhas de comando.

O conjunto das linhas de dependências de um *makefile* forma a estrutura de dependências de processamento de um sistema, que é a representação do seu modelo. Para construir um nó N desta estrutura, é necessário construir recursivamente todos os nós dos quais N depende. Na construção de N, se qualquer um destes nós tiver sido modificado desde a última alteração de N, ou se N ainda não existir, são executados os comandos que seguem a linha de dependências de N. Um *makefile* para um sistema composto pelos arquivos parse.c e lex.c, que incluem o arquivo inc.h, consiste em:

```
pgm: parse.o lex.o
   cc -o pgm parse.o lex.o

parse.o: parse.c inc.h
   cc -c parse.c

lex.o: lex.c inc.h
   cc -c lex.c
```

Este modelo de sistema estabelece que o programa pgm depende dos arquivos parse.o e lex.o, e que qualquer modificação em parse.o ou em lex.o implica na reconstrução de pgm. Da mesma forma, parse.o é dependente de parse.c e de inc.h e deve ser reconstruído se algum deles for modificado. O mesmo se dá com lex.o em relação a lex.c e inc.h.

Assim, para construção do sistema pgm, primeiro devem ser construídos parse.o e lex.o. Como parse.o depende de parse.c e de inc.h e estes não dependem de nenhum outro componente, o comando cc -c parse.c só será executado se a data de atualização de parse.c ou de inc.h for mais recente que a de parse.o, ou se parse.o não existir. O

mesmo procedimento é usado para derivar lex.o. Uma vez construídos parse.o e lex.o, pode-se construir pgm. Novamente, o comando cc -o pgm parse.o lex.o só será executado se parse.o ou lex.o forem mais recentes que pgm, ou se pgm não existir.

Uma maneira de facilitar a descrição da estrutura de dependências, é a definição de dependências a nível de tipos. Desta forma, as dependências das instâncias são inferidas a partir das regras declaradas no nível de tipos. O Make tira proveito do fato do Unix ter como convenção que o sufixo do nome de um arquivo identifica o seu conteúdo. Um arquivo cujo nome termina em .o contém código objeto, enquanto que um sufixo .c implica em um arquivo fonte na linguagem C. Ele permite que sejam declaradas regras de dependências e de transformações implícitas, baseadas no sufixo do nome do arquivo. O usuário pode fornecer a sua própria lista de sufixos e regras implícitas, mas já existe um conjunto *default* oferecido pelo próprio **Make**.

Existe também um mecanismo de substituição de macros, que permite que a construção do sistema seja feita de maneiras diferentes através do mesmo *makefile*. Uma chamada a uma macro é denotada pelo caractere $, e o seu valor pode ser definido no próprio *makefile* ou no comando de chamada ao **Make**. Existem também as macros pré-definidas, que são dinâmicas, ou seja, seus valores podem se alterar durante o processo de construção. Por exemplo, o valor da macro $< é o prefixo do nome usado para invocar a regra, e portanto tem um valor diferente para cada regra. Para o compilador cc, a regra que determina que um arquivo que tem o nome com sufixo .o depende do arquivo cujo nome tenha o prefixo idêntico e o sufixo .c, é:

```
.c.o:
    cc -c $<.c
```

Uma grande restrição do sistema **Make** é que ele não tem nenhum controle de versões. As versões dos componentes do modelo do sistema usadas na construção da configuração são os arquivos do diretório corrente. Os outros gerenciadores que usam o mesmo tipo de modelo de sistema que o **Make**, têm mecanismos de construção de configuração integrados ao controle de versões, visando resolver este problema.

No **DSEE** uma *configuration thread* estabelece por meio de rótulos a versão de cada componente do modelo de sistema, conforme descrito na Secção 2.3.2. A construção da configuração segue o mesmo procedimento do **Make**, porém, quando um componente é citado, a versão que deve ser usada é determinada através de uma *configuration thread*.

O problema de especificar as versões que compõem a configuração é resolvido pelo **shape** dividindo os arquivos *shapefile* em várias partes. Uma dessas partes é a de regras de transformação, que é muito semelhante a um *makefile*. A diferença básica é que em cada linha de dependência pode existir um nome de uma regra de seleção. As regras de seleção, por sua vez, são descritas separadamente em outra parte.

Durante uma construção de configuração, quando o nome de uma regra de seleção é encontrado, ela é ativada. Os componentes do sistema, descritos na linha de dependências em que foi encontrado o nome da regra de seleção, terão suas versões selecionadas de acordo com ela. Um *shapefile* que contivesse a parte de regras de seleção igual à do exemplo da Seção 2.3.2, poderia ter a seguinte parte de regras de transformação:

```
#%Transformation_rules

test: rule_1 pgm

release: rule_2 pgm

pgm: parse.o lex.o

   (...)
```

Na construção da dependência test, as versões usadas para derivar pgm seriam selecionadas de acordo com a regra rule_1, descrita na Seção 2.3.2. A regra rule_2 também deveria estar definida na parte de seleção. Ela seria usada para selecionar as versões dos mesmos componentes se a dependência **release** estivesse em construção. O processo de construção, bem como o mecanismo de definição de regras implícitas e o de substituição de macros é idêntico ao do **Make**.

Configurações Baseadas em Dependências Fonte

A descrição de modelos de sistema a partir de dependências fonte é usada nos gerenciadores **Adele**, **Inscape** [Perr87b], **BiiN SMS** e nos modelos apresentados em [Wink87] e em [Katz87a]. Em geral, o processo de construção de configuração nos gerenciadores deste tipo consiste simplesmente na seleção de uma versão para cada componente do modelo de sistema. Ou seja, não existe derivação de outros componentes.

No **Adele** os objetos são divididos em interface e implementação, e tanto as versões de interface como as de implementação podem depender de outras interfaces. Uma configuração de uma interface é definida como o conjunto de implementações necessárias para implementar esta interface. Como uma configuração implementa os recursos de sua interface, ela também pode ser chamada de implementação. Assim, a construção da configuração de uma interface I consiste em selecionar uma versão para cada implementação ou configuração pertencente ao modelo de sistema, começando por I.

Cada versão tem um *manual*, que descreve os seus atributos e as interfaces das quais ela depende. Além disso, no *manual* de uma versão estão descritas as restrições sobre os atributos das versões, com as quais ela pode ser associada. Uma *descrição de configuração* não é associada a nenhuma versão em particular, mas também consiste em restrições sobre os atributos das versões que podem fazer parte do sistema. Através de restrições é possível expressar por exemplo, que a versão selecionada para um componente deve ter o valor do atributo **sistema** igual a Unix, ou ainda que uma versão que tenha o atributo **estado** definido como **inconsistente** não pode fazer parte do sistema.

A construção da configuração seleciona uma versão para cada componente, que satisfaça tanto as restrições da *descrição de configuração* como as restrições dos *manuais* das versões que dependem transitivamente dela.

Quando a própria linguagem de programação, além de descrever a relação entre os objetos, permite a descrição de informação a respeito das versões dos objetos, a configuração formada é chamada de pragmática. A abordagem apresentada em [Wink87] usa uma linguagem de configuração que deve ser incorporada pela linguagem de programação. A linguagem de configuração se baseia no fato de que uma versão de um objeto pode pertencer a mais de uma versão do sistema. Assim, as versões devem ter na parte inicial do código uma declaração de configuração que consiste em uma *definição de versão* e na *identificação das versões* dos componentes usados por ela. A definição de versão não é uma identificação única da versão do objeto, e sim a descrição das versões do sistema às quais a versão do objeto pertence. A definição de versão é composta pela lista de revisões e pelas variantes do sistema que incluem aquela versão.

A identificação da versão de um componente usado por outra versão consta do nome do componente referenciado e de uma maneira de se definir a versão deste componente. A versão do componente pode ser definida explicitamente ou por mapeamento dos valores dos atributos da versão do nível superior. O mapeamento se baseia na versão definida para o componente que faz a referência para determinar a versão do componente referenciado. O exemplo abaixo mostra o mapeamento dos atributos da versão de exe_main para a versão do componente sort. A definição de versão segue o rótulo VERS e, após o USE, vem a identificação das versões dos componentes referenciados pela versão.

```
exe_main
   CONFIG
      VERS = 1:{Speed = {High, Low}};
      USE sort VERS = SAME:{Speed = High} => {Kind = Quicksort}
                           {Speed = Low} => {Kind = Bubblesort};
   Proc exe_main is
      ...
   sort(A,B);
End exe_main

sort
   CONFIG
      VERS = 1:{Kind = QuickSort),
   Proc sort (...) is
      ...
End sort

sort
   CONFIG
      VERS = 1:{Kind = BubbleSort};
   Proc sort (...) is
      ...
End sort
```

Na identificação das versões usadas por exe_main, o rótulo SAME define que a versão

de sort selecionada deve ter a mesma revisão que a do exe_main, isto é, revisão 1. Além disso, se o componente exe_main for selecionado para fazer parte do sistema através da variante Speed igual a High a versão do componente sort usado deve ter o atributo Kind igual a QuickSort. Enquanto que se a versão de exe_main for selecionada com a variante Speed igual a Low, deve ser usada a versão de sort com atributo Kind igual a Bubblesort.

Um mecanismo de configuração muito simples é proposto em [Chou86]. Como as versões estão em uma hierarquia de bancos de dados, cada referência feita por uma versão deveria especificar além do nome do objeto, o número da versão e o banco de dados em que ela está armazenada. Mas a referência pode especificar só o nome do objeto e deixar uma ou mesmo as duas outras partes não especificadas.

O usuário pode especificar as versões a serem usadas para qualquer componente do sistema em uma *descrição de configuração*. Além disso, cada objeto pode ter uma versão *default* declarada pelo usuário. Durante a construção da configuração, se a versão de um objeto não tiver sido especificada na referência, ela é procurada na descrição da configuração. Caso a versão do componente também não tenha sido definida na descrição, a versão *default* daquele objeto será usada.

Além disso, a ordem de busca pelos bancos de dados é pré-estabelecida. Uma versão só pode referenciar outras versões que estejam no mesmo banco de dados que ela, ou em um banco de dados do nível superior ao que ela está. Uma versão em um banco de dados particular pode fazer referências às versões neste banco de dados, no do projeto relacionado ao usuário, ou no geral. As versões dos bancos de dados de projeto só podem fazer referências às versões do mesmo banco de dados e do geral. Finalmente, as versões do banco de dados geral podem referenciar exclusivamente as versões deste banco de dados.

Existem ainda, mecanismos que não se responsabilizam pela escolhas das versões dos componentes, mas são destinados a verificação da validade da substituição de uma versão por outra, em uma configuração previamente construída. A linguagem de especificação de interface **Intress** [Perr87a] é usada no ambiente **Inscape** [Perr87b], para determinar os impactos causados pela alteração de uma versão na configuração do sistema. A especificação da interface de um objeto consiste de predicados que estabelecem pré-condições, pós-condições, obrigações e exceções relacionados a sua execução. Através da comparação destes predicados é possível determinar qual o grau de compatibilidade entre duas versões dentro de uma determinada configuração.

Tradução de Dependências Fonte

As *dependências fonte* podem ser declaradas explicitamente, ou então extraídas dos objetos através de um pré-processador. Trabalhar com elas é mais cômodo para o usuário, mas requer uma tradução de *dependências fonte* para *dependências de processamento*. Por exemplo: que dependências de processamento estão implícitas quando um objeto X inclui um objeto Y, que por sua vez usa um procedimento declarado em Z? Deve ser feita uma *link-edição* de Z com X? O X deve ser recompilado quando Z é alterado?

O BiiN SMS tem um mecanismo que automatiza o mapeamento entre as dependências fonte e as dependências de processamento. No modelo do sistema o usuário

declara os componentes, as dependências fonte entre eles e as ferramentas a serem usadas na construção do sistema. Antecipadamente deve ter sido feito o registro da ferramenta, onde são declaradas as suas dependências de processamento *default* e etapas de construção de sistemas. Cada etapa de construção contém modelo do comando necessário. E ainda, para cada tipo de dependência fonte com que a ferramenta pode lidar, deve ser declarada um regra de tradução para dependências de processamento primitivas. Assim, as dependências de processamento podem ser inferidas a partir das dependências de processamento *default*, das regras de tradução declaradas no registro da ferramenta e das dependências fonte do modelo do sistema.

Uma vez identificada a estrutura das dependências de processamento do sistema, é feita a seleção das versões baseada nos critérios estabelecidos pelo usuário, juntamente com o modelo do sistema. A seleção de versões se baseia no mecanismo descrito para o **Gypsy** na Secção 2.3.2. Finalmente, o sistema é construído, identificando-se para cada dependência de processamento, uma etapa de construção adequada, no registro de ferramentas.

2.4.4 Tipos de Construção de Configuração

Outra particularidade a respeito da construção de configurações segundo Heimbginer [Heim88], é que elas podem ser feitas estática ou dinamicamente. Em uma construção estática, todos os nomes dos componentes do sistema e as relações entre eles são conhecidos a priori, isto é, existe um modelo de sistema fixo. No início de uma construção de configuração dinâmica existem muitas possibilidades para o modelo do sistema. Ele é determinado durante o processo de construção e depende das propriedades das versões escolhidas.

Para tornar as coisas mais claras, basta pensar em um mecanismo em que as dependências são definidas em relação aos objetos, e em outro no qual as dependências são definidas em relação às versões. No primeiro caso, se o objeto A depende de B, todas as versões de A dependem de versões de B. Assim, o modelo de sistema é conhecido no início da construção. No outro, enquanto uma versão de A depende de B outra versão de A pode depender de B e C, como também pode depender só de C. Ou seja, as versões de um mesmo objeto podem fazer referências a objetos diferentes. Portanto, o modelo do sistema só é conhecido depois de construída a configuração. A opção entre um tipo de construção e outro influencia significativamente nas capacidades do mecanismo de gerenciamento de configuração.

Em geral, o tipo de construção reflete a estratégia de organização das informações a respeito de versões. A construção estática é usada quase sempre pelos gerenciadores que centralizam as informações, ao passo que a construção dinâmica aparece nos que têm as informações descentralizadas.

Nos mecanismos em que as informações são centralizadas, tanto o modelo do sistema como as regras para a seleção das versões de todos os componentes estão no mesmo lugar, como é o caso do **shape**, do **DSEE** e do **Make**. Nos outros a configuração de cada versão (os objetos usados por ela e as regras para selecionar versões destes objetos) é armazenada junto com a própria versão que faz as referências. Nestes mecanismos,

uma configuração pode fazer referências a outras configurações, formando assim uma estrutura de configurações. A descentralização de informação é usada no **BiiN SMS**, no **Adele** e nos modelos propostos em [Chou86] e [Wink87].

A informação centralizada tem a vantagem de possibilitar a visão global da estrutura do sistema. Por outro lado, é muito difícil modelar um sistema em desenvolvimento com uma estrutura fixa. A simples decomposição de um módulo em dois, invalida o modelo de sistema.

A principal vantagem da abordagem descentralizada é que a informação a respeito das versões é adaptada à estrutura do sistema [Wink87]. Desta forma, as restrições técnicas ou lógicas podem ser expressas no nível em que elas são relevantes, e provavelmente são declaradas por pessoas que entendem destas restrições [Belk87].

2.5 Objetos Derivados

2.5.1 Armazenamento de Objetos Derivados

Quando um objeto derivado é reconstruído, o resultado deste processo deve substituir a versão antiga deste objeto, ou deve ser armazenado como uma nova versão? A manutenção automática de versões de objetos derivados é uma decisão muito delicada. Em geral, é interessante manter algumas versões de objetos derivados, sem ter o trabalho de salvar a versão por meio de mudança de nome, ou local de armazenamento. Por outro lado, manter todas as versões dos objetos derivados pode levar a uma proliferação de versões desnecessárias.

No **Make** os objetos derivados são armazenados no diretório corrente do usuário. Sempre que um objeto derivado estiver desatualizado em relação às suas dependências ele é reconstruído gerando uma nova versão, que substitui a antiga.

Manter uma área destinada ao armazenamento de versões de objetos derivados foi a solução adotada pelo gerenciador **DSEE** e, posteriormente, pelo **shape**. Nesta área são armazenadas todas as versões produzidas na construção dos componentes do sistema, juntamente com as informações que descrevem o contexto em que se deu a sua derivação. As informações de derivação compreendem as versões dos componentes de entrada, a definição das macros, as ferramentas e as opções usadas para derivar os objetos. Antes de reconstruir um componente, o gerenciador procura na área uma versão equivalente, isto é, uma versão que tenha sido derivada no mesmo contexto. Este método, além de evitar reconstruções desnecessárias, consegue uma seletividade maior que a oferecida pelo **Make** na determinação da reconstrução. Como um exemplo pode ser citado o caso de uma compilação onde as versões são as mesmas que as de uma compilação anterior, mas a opção de compilação foi alterada. Em uma situação assim, o **Make** não recompilaria o objeto, enquanto que os outros dois gerenciadores o fariam.

A manutenção de versões de objetos derivados não foi considerada útil no projeto do gerenciador **BiiN SMS**. Os objetos derivados e as informações de derivação são armazenados na própria configuração, que é um diretório estendido. Na construção de um componente, o gerenciador procura um equivalente somente dentro da configuração. Quando o componente é reconstruído, a versão gerada é armazenada no lugar da antiga.

2.5.2 Evitando reconstruções

Normalmente, o usuário quer que todos os componentes derivados em uma configuração sejam construídos a partir das versões selecionadas para os componentes de entrada. No entanto, algumas vezes ele pode querer que certos objetos derivados não sejam reconstruídos, mesmo sabendo que estão desatualizados em relação às versões selecionadas. Isto acontece quando o usuário acredita que as alterações feitas nos objetos fonte são irrelevantes para os objetos derivados. Por exemplo, a alteração de um comentário em um programa é, normalmente, irrelevante para o código objeto.

O **Make** permite que um usuário evite a reconstrução de certos objetos derivados através do comando *touch*, que atualiza a data da última modificação do arquivo, sem alterar o seu conteúdo. O uso do comando *touch* não permite saber quais inconsistências estão sendo ignoradas, o que pode ser inconveniente em certas situações. A mesma alteração de comentário que é irrelevante para o código objeto, pode causar impacto em outro objeto derivado como a extração de comentários. O mecanismo que evita reconstruções do **BiiN SMS** é mais seletivo. Através dele o usuário pode declarar que uma inconsistência entre uma versão de um objeto derivado e as versões dos objetos dos quais ele depende deve ser ignorada.

2.5.3 Versões de Sistema

Na manutenção de produtos de software, algumas vezes é necessário reconstruir uma configuração do sistema, exatamente como ela se encontrava em um determinado momento. Isto implica na necessidade de manter versões da configuração do sistema.

Como todas as versões dos objetos derivados são mantidas pelo **DSEE** e pelo **shape**, as versões de sistema podem ser recuperadas facilmente.

No **DSEE**, quando uma configuração vai ser construída, a *configuration thread* em questão é resolvida gerando para cada objeto derivado uma *bound configuration thread*. Nela as versões das quais o objeto derivado depende são especificadas diretamente pelos seus números e o comando usado para construção do objeto está completamente especificado. O processo de construção de configuração se baseia nestas informações e cada *bound configuration thread* é armazenada juntamente com o respectivo objeto derivado, como já foi mencionado na Secção 2.5.1. Além destas informações, cada construção do sistema é documentada em um registro que permite localizar as *bound configuration threads* correspondentes.

Uma abordagem diferente é adotada no **shape**, onde os registros sobre as versões do sistema são guardados somente quando o usuário requisitar. Isto é feito por meio de um arquivo *shapefile* em que as versões de todos os componentes da configuração são identificados por seus números e todos os comandos usados para a derivação dos componentes estão registrados. Esta abordagem permite que uma configuração do sistema seja reconstruída, mesmo se os objetos derivados já tiverem sido removidos de sua área de armazenamento.

Versões de sistema são as unidades gerenciadas pelo **Orweel** [Thom88], que controla configurações em ambientes **Smalltalk**. Cada usuário pode controlar versões de unidades menores, que no caso são classes e métodos, mas apenas no seu espaço de trabalho

particular. O compartilhamento de versões só é feito a nível de sistema. Quando se deseja compartilhar versões de classes, deve-se liberar uma versão de um sistema com uma única classe. Uma abordagem semelhante é adotado no DF, que é um mecanismo de controle de configurações construído "em cima" do sistema de arquivos **Cedar** . Porém, neste mecanismo as configurações são formadas por arquivos e não por classes e métodos.

No modelo proposto em [Chou86] versões do sistema são mantidas automaticamente, pois cada versão de objeto só tem uma configuração e todas as versões do banco de dados geral são congeladas. Contudo, o gerenciador deixa a cargo do usuário a transferência para o banco de dados geral, de todas versões referenciadas na configuração do sistema.

2.6 Controle de Alternativas

Existe o senso comum que versões paralelas ou variantes são realizações alternativas do mesmo conceito. A idéia de alternativa parece simples, mas a sua manutenção pode se tornar extremamente difícil.

Versões alternativas podem ser representadas em um único documento fonte, que deve ser pré-processado quando se quer acessar uma alternativa particular. Mas elas podem também estar fisicamente separadas, sendo que cada alternativa é uma versão. Alternativas deste tipo devem ter nomes diferentes, ou estar armazenadas em espaços diferentes [Mahl88]. A identificação de alternativas é feita pela passagem de um conjunto de *flags* para o pré-processador ou através de nomes e lugares apropriados, dependendo do seu tipo.

O primeiro tipo de alternativa pode resultar em um documento fonte com grande quantidade de código relacionado ao pré-processador, tornando difícil a sua leitura e entendimento. Conseqüentemente, a manutenção e especificação de configuração são trabalhos que estão sujeitos a muitos erros. Por outro lado, alternativas fisicamente separadas são dispendiosas em termos de espaço em disco. Além disso, para alterar as partes comuns a todas as alternativas é necessário repetir o trabalho em todas elas, fazer a junção das alterações feitas em uma alternativa nas outras, ou alterar só a versão mais importante abandonando as outras.

O **shape** suporta o uso dos dois tipos em conjunto. Ele permite a definição de alternativas no mesmo arquivo em que estão as regras de seleção de versões e o modelo de sistema. Na definição de cada alternativa são especificadas as *flags* do pré-processador adequadas para a alternativa, ou o diretório onde ela está armazenada. Caso seja necessário, tanto as *flags* como os diretórios são especificados.

Para a seleção de uma alternativa, a regra usada deve conter um predicado com a forma attrvar(nome da alternativa). As versões dos componentes selecionados por esta regra serão procuradas no diretório definido para a alternativa e as *flags* especificadas para ela serão passadas para a ferramenta que fará a transformação. Um exemplo de definição de alternativa é:

```
#% VARIANT_SECTION
vclass system ::= (vaxbsd, munix)
vclass database ::= (damokles, unixfs)
```

```
   vaxbsd:
      vflags=''-DUNIX -DBSD43 -DSTDCC -DVAX -DPSDEBUG''
      vpath =''sys/vasbsd''
   unixfs:
      vflags=''-DFS''
      vpath =''data/unixfs''
   ( ... )
#% END_VARIANT_SECTION
```

Existindo esta definição de alternativas, as regras de seleção podem usar um predicado do tipo:

```
exe_rule:
   *.(ch), atrvar (vaxbsd), atrvar (unixfs), attr (state, busy)
```

O uso da regra exe_rule para seleção de uma versão de um componente cujo nome tenha o sufixo .c resultaria na passagem das *flags*: -DFS -DUNIX -DBSD43 -DSTDCC -DVAX -DPSDEBUG para o pré-processador, e as versões seriam procuradas no diretório sys/vaxbsd e data/unixfs.

Embora este método seja bastante interessante, a definição das alternativas não está no histórico, juntamente com as versões. Elas ficam no mesmo arquivo que o modelo de sistema. Isto acarreta duplicação de informação em cada modelo de sistema em que as alternativas forem usadas.

O **Make** também suporta alternativas armazenadas em um único documento com linhas de controle de compilação condicional inseridos no código, através das macros, que podem definir valores diferentes para as *flags* do pré-processador.

As alternativas armazenadas em documentos diferentes podem ser classificadas entre as que futuramente serão fundidas em uma única versão e as que nunca sofrerão uma junção.

O **DSEE** e o **BiiN SMS** têm um mecanismo que auxilia o usuário a fazer a junção de alternativas. Esse tipo de alternativa, surge quando se descobre um erro em uma versão antiga. A manutenção deve ser feita sem afetar, ou ser afetada por outras versões em desenvolvimento. Cria-se então, uma nova ramificação a partir da versão em que foi encontrado o erro, onde serão feitas as correções necessárias. Ao mesmo tempo, o desenvolvimento de novas versões continua na ramificação principal. Terminadas as correções, pode ser feita uma junção da ramificação secundária com a ramificação principal, incorporando as correções feitas às novas versões.

O outro tipo de alternativa em documentos separados permite implementações radicalmente diferentes do mesmo módulo. Por exemplo: um módulo de controle de I/O pode ter uma implementação em assembler para duas máquinas diferentes. Muitos ambientes suportam esse tipo de alternativa, criando ramificações na árvore de derivação, sendo que alguns deixam a cargo do usuário a definição da configuração com a escolha da alternativa adequada para cada caso.

O **Adele** suporta a criação dinâmica de atributos para as versões; portanto, as versões alternativas podem ser identificadas por predicados que se baseiam nestes atributos. Os

critérios de seleção de uma versão são propagados para todas as versões que dependem dela. Assim, a especificação da alternativa pode ser feita uma só vez, e todas as versões da configuração serão selecionadas segundo esta especificação.

Uma maneira diferente de se lidar com alternativas é permitir que o usuário dê nomes às ramificações da estrutura de derivação de versões. Posteriormente, as alternativas podem ser escolhidas por parâmetros das configurações que especifiquem os nomes das ramificações. Como o valor de um parâmetro de uma configuração pode ser definido nas configurações que dependem dela, a especificação da alternativa também pode ser feita uma única vez. Esta é a abordagem encontrada no **BiiN SMS**. O gerenciador permite ainda, que a definição das *flags* para o pré-processamento de uma alternativa seja feito por meio de parâmetros. Contudo, a consistência das escolhas dos valores dos parâmetros fica a cargo do usuário que faz a construção da configuração.

2.7 O Problema do Controle de Alterações

Em uma configuração, uma versão pode referenciar e pode também ser referenciada por outras versões. Assim, uma alteração em uma versão pode causar impactos em outras versões. O usuário precisa manter um controle sobre as alterações das versões que ele está usando. Algumas vezes, ele mantém o controle trabalhando em um contexto estável, onde só as suas versões são alteradas. Outras vezes, ele prefere não fixar o contexto, mas se manter informado a respeito dos impactos causados por alterações de versões.

2.7.1 Contexto Estável

Um usuário tem que controlar as alterações não só das suas versões, mas também das versões de outros usuários que fazem parte do seu contexto. Ele precisa se manter informado a respeito das alterações feitas por outros, mas se reserva o direito de decidir quando incorporar estas alterações na sua versão do sistema. Se o usuário quiser sempre a última versão dos objetos desenvolvidos por outros, ele pode ter que reconstruir grandes porções do sistema em momentos inconvenientes. E, mais importante, as alterações nas versões que não são suas podem forçá-lo a alterar as suas para se adaptar ao novo contexto.

A única maneira de se contornar este problema em alguns ambientes é especificar exatamente o contexto desejado, fixando o número das versões que não devem mudar, e determinando regras de seleção para as outras. Assim, o usuário deve especificar exatamente o número da versão dos objetos de outros usuários, que fazem parte da sua configuração, e estabelecer regras que selecionam versões diferentes conforme a evolução dos objetos que ele está desenvolvendo.

No **DSEE** e no **shape** isto pode ser feito, desde que cada usuário tenha uma *configuration thread* ou um *shapefile* próprio. No **Adele** e no modelo proposto em [Chou86], é mais difícil implantar este mecanismo, pois as versões dos objetos usados por uma versão são especificados diretamente na configuração da versão que faz a referência. E como só existe uma configuração para cada versão de objeto, não é possível que cada

usuário especifique um contexto. Em algumas situações, ele pode se ver obrigado a derivar uma nova versão a partir de outra já existente, só para poder fazer uma configuração adequada a ele.

O comando *bind* é oferecido pelo gerenciador **BiiN SMS** para auxiliar o usuário no sentido de fixar um contexto. Quando este comando é aplicado em um componente de uma configuração, ele faz com que a regra de seleção para este componente seja avaliada e que a versão selecionada para o componente seja usada até que o usuário peça para mudá-la. As construções subseqüentes vão usar a mesma versão para o componente, em vez de reavaliar as rs vão usar a mesma versão para o componente, em vez de reavaliar as regras de seleção. O usuário pode usar o comando *bind* em todos os componentes em que ele não está trabalhando, e deixar livre para alterações os seus componentes. Ele pode ainda, pedir para receber notificações do gerenciador de eventos sempre que aparecerem novas versões dos componentes nos quais foi aplicado o comando *bind*.

2.7.2 Controle do Impacto das Alterações

Independentemente do usuário estar trabalhando em um contexto estável, é interessante que ele se mantenha informado dos impactos causados nas suas versões. Mesmo se o contexto for estável, uma alteração em uma versão do próprio usuário pode ter efeitos sobre outras versões suas. No entanto, se o contexto não for estável, a informação sobre alterações nas versões é essencial para o controle do seu trabalho.

Em alguns ambientes o usuário especifica que ele quer ser informado se determinados objetos ou versões forem modificados. Em outros, o usuário recebe informações se a modificação de qualquer versão causar algum impacto na versões determinadas por ele.

O **Gypsy** e o **DSEE** permitem que o usuário especifique os objetos nos quais ele está interessado através de um mecanismo de subscrição de eventos. Na verdade este mecanismo suporta mais que a simples notificação de alteração. Uma subscrição consiste de um objeto alvo, uma condição e uma ação. A condição é descrita por um predicado. Um evento ocorre quando o objeto alvo satisfaz uma determinada condição, e então a ação é efetuada. Através da ação relacionada ao evento, quem faz a subscrição pode determinar que quer receber uma notificação, mas pode também fornecer um programa para ser executado quando o evento ocorrer. É possivel monitorar eventos quando:

- uma operação específica (ex: eliminação, atualização, transferência) é realizada em qualquer versão que satisfaça um predicado;
- uma operação específica (ex: *check-out*, *check-in*) é feita em uma determinada ramificação; ou
- uma versão deixa de satisfazer a um determinado predicado.

O controle de alterações baseado nas referências entre os objetos ou versões é usado no **Adele** e no modelo proposto em [Chou86]. O usuário determina uma versão, e os tipos de modificações nos quais ele está interessado. O tipo de modificação pode ser atualização, eliminação ou criação de uma nova versão. A versão especificada pelo usuário pode depender de várias outras e, se uma delas sofrer uma modificação do tipo

determinado por ele, o usuário é notificado. Nestes gerenciadores as ações relacionadas ao controle de alteração consistem simplesmente no envio de mensagens aos usuários ou na atualização do valor de algum atributo. Uma alteração em uma versão pode resultar, por exemplo, na mudança do atributo estado de outra versão de consistente para obsoleta.

O **Adele** permite, ainda, especificar o momento em que se deseja receber a notificação. Se o usuário for notificado antes da alteração, ele pode avaliar o impacto que ela causaria, podendo validar ou não a alteração. Uma notificação, imediatamente após a modificação, permite que o usuário adapte suas versões às alterações logo em seguida. E se ele receber a notificação quando for usar uma versão que sofreu impacto da modificação de outra versão, ele evita usar a versão inconsistente. O usuário pode, ainda, ser notificado da alteração de uma versão somente quando ele pedir esta informação.

Capítulo 3

MVC: Um Modelo para Controle de Versões e Configurações

Neste capítulo é apresentado um modelo que visa estabelecer um suporte adequado ao controle de versões e configurações, o MVC.

3.1 Introdução

Um ADS deve, sem dúvida, suportar o controle e armazenamento de versões e configurações. Existem propostas que tratam de vários aspectos do problema, sendo que algumas delas foram apresentadas no Capítulo 2. No entanto, a maioria destas propostas não leva em conta algumas características dos ambientes, como por exemplo, a sua arquitetura e a maneira efetiva na qual os usuários e as aplicações compartilham dados.

O modelo que será proposto, o MVC, aborda os problemas envolvidos no suporte a versões e configurações, e propõe soluções consistentes para eles. Alguns dos problemas foram resolvidos a partir de soluções existentes em outros modelos, e para outros problemas novas soluções foram propostas.

Ambientes diferem uns dos outros, na organização e no acesso às informações relativas às versões e configurações. Como o MVC tem a intenção de atender às características de vários ADS, ele capta as características de cada ambiente através da definição de alguns parâmetros, nos quais se baseiam as operações de controle.

3.2 Distribuição dos Dados

Em geral, os usuários de um ADS são divididos em diferentes grupos, de acordo com o projeto que desenvolvem. Algumas informações são compartilhadas por todos eles, outras

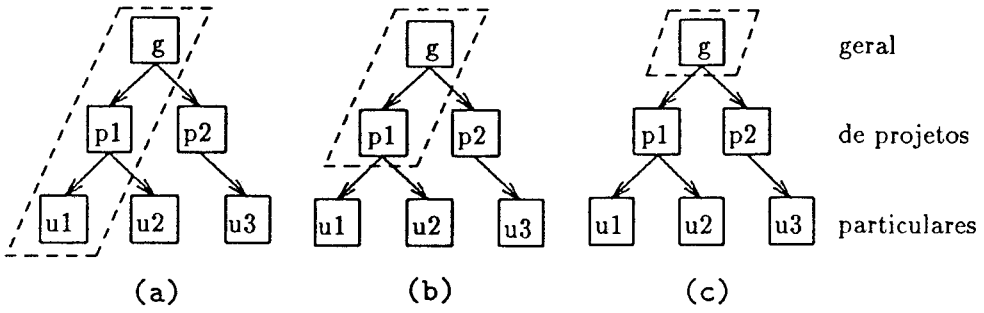

Figura 3.1: Bancos de dados acessíveis ao: (a) usuário u1, (b) responsável pelo projeto p1 e (c) responsável pelo ambiente.

devem ser restritas aos componentes de um grupo, e existem também as informações particulares de cada usuário. Assim sendo, elas serão distribuídas em uma hierarquia de banco de dados semelhante à proposta em [Chou86] e apresentada na Seção 2.2.1.

As informações são organizadas de acordo com os usuários que têm acesso a elas. Para tanto, são necessários três tipos diferentes de bancos de dados: os particulares pertencentes aos usuários comuns, os de projeto que pertencem aos grupos, e o banco de dados geral, que armazena as informações pertencentes ao ambiente todo. O representante de um grupo é o dono do banco de dados do projeto associado, enquanto o representante do ambiente é o dono do banco de dados geral.

Cada usuário tem acesso ao seu próprio banco de dados e aos que estejam em níveis superiores ao seu, na hierarquia. Contudo, ele não pode fazer acessos a bancos de dados que estejam no mesmo nível, ou em níveis inferiores ao seu, como mostra a Figura 3.1. Além do seu próprio banco de dados, um usuário comum tem acesso ao do projeto ao qual ele pertence e ao geral; um representante de grupo pode fazer acessos ao banco de dados do projeto e ao geral; finalmente, o dono do banco de dados geral tem acesso somente a ele. Desta maneira, fica estabelecida uma organização de informações que reflete a própria hierarquia dos usuários do ambiente.

Os objetos, as versões e as configurações armazenadas em um banco de dados estão divididos em várias partições. São elas que definem quais operações podem ser realizadas pelos usuários que têm acesso aos bancos de dados. Assim, a localização de um objeto na hierarquia de bancos de dados estabelece os usuários que têm a possibilidade de acessá-los, mas o tipo do acesso permitido é definido pela sua partição. O controle de acesso através das partições será detalhado na Seção 3.8.

3.3 Os Dados Manipulados no Ambiente

3.3.1 As Versões e Configurações dos Objetos de Desenvolvimento

A maneira utilizada para modelar os objetos segue a proposta de Katz [Katz86]. A adaptação desta proposta para ADS apresentada na Seção 1.2 define que um objeto de desenvolvimento pode ter várias representações, sendo que em cada representação ele tem várias versões que documentam sua evolução. As versões em representações diferentes podem ser equivalentes e uma configuração liga uma versão de um objeto composto às versões dos objetos componentes. No entanto, os relacionamentos de equivalência não serão tratados pelo modelo proposto.

A estrutura do software é definida no MVC através de referências genéricas. Cada versão faz referência a um conjunto de objetos que ela usa. A referência é feita genericamente a um objeto, e não a uma versão específica dele. Desta forma, a versão que faz a referência pode ser associada a diferentes versões do objeto referenciado. Além disso, as versões de um mesmo objeto podem fazer referências a objetos diferentes. Enquanto a versão 1 do objeto X faz referência aos objetos Y e Z, a sua versão de número 2 pode fazer referências aos objetos Z e V, como é mostrado na Figura 3.2.

A versão que faz a referência é ligada às versões dos objetos referenciados através das configurações. A exemplo do Adele [Belk87] e do modelo apresentado em [Chou86], o MVC associa as configurações diretamente às versões. Porém, de forma contrastante com estes gerenciadores, ele permite que cada versão tenha várias versões de configuração. Assim, a versão 1 de X pode ser ligada à versão 2 de Y e à 4 de Z pela sua configuração 1; enquanto que a configuração 2 a associa à versão 3 de Y e a 3 de Z.

Como um mesmo objeto de desenvolvimento pode ter várias representações, um par do tipo [nome do objeto][representação] é usado para identificar unicamente um objeto em todo o ambiente. Cada versão de um objeto, ao ser criada, recebe um número que a distingue das outras versões do mesmo objeto, e cada configuração de uma versão também tem um número próprio. Assim o objeto X na representação código fonte é identificado pelo par [X][código fonte], enquanto [X][código fonte][2] identifica a sua versão de número 2, e [X][código fonte][2][3] é a identificação da terceira versão de configuração da versão 2 deste objeto.

Para facilitar o entendimento, neste texto as versões de configuração são tratadas como configurações enquanto as versões de objetos de desenvolvimento são chamadas simplesmente de versões. Quando o assunto abordar indistintamente, objetos, versões e configurações, eles são chamados de elementos. Além disso, os exemplos simplificam a identificacão dos objetos, omitindo a especificação da representação.

Em uma configuração, a versão escolhida para o objeto referenciado pode, por sua vez, fazer referência a outros objetos. Portanto, ela também deve ter uma configuração para determinar as versões dos objetos referenciados por ela. Desta forma, uma configuração do sistema é composta por versões e subconfigurações, que também podem ser compostas, formando uma hierarquia de configurações. No exemplo da Figura 3.2, a versão X[1] faz referência aos objetos Y e Z, sendo que a configuração X[1][1] deter-

mina que as versões Y[2] e Z[4] sejam usadas na construção de X. A versão Z[4] não faz referências a outros objetos, mas o objeto W é referenciado por Y[2]. A configuração Y[2][1] determina que W[1] seja usada, ao passo que a versão W[3] é a especificada na configuração Y[2][2]. Portanto, a configuração X[1][1] pode ser composta de maneiras diferentes como é mostrado na Figura 3.2.

O uso de configurações diferentes possibilita que cada usuário componha o sistema com as versões adequadas ao seu trabalho. Ele pode especificar a composição de uma configuração através de suas entradas, que serão detalhadas na Seção 3.7.1.

Além das configurações específicas de cada versão, existem as configurações globais. Cada representação tem uma configuração global relativa a ela, e pode existir mais de uma versão de cada configuração global. A configuração global é usada sempre que uma configuração de uma versão não especificar versões para todos os objetos referenciados por ela. No exemplo acima, se a configuração X[1][1] só informasse que a versão 1 de X deveria ser ligada a versão Y[2], e não tivesse nenhum dado sobre qual versão de Z deveria ser usada, esta informação seria procurada na configuração global da mesma representação que X.

3.3.2 Propriedades

Os objetos, versões e configurações de cada ambiente, podem ter diferentes propriedades representadas por seus atributos. Alguns atributos básicos, como a identificação, autor e data de criação e de atualização são mantidos automaticamente pelo **MVC**. No entanto, novos atributos podem ser definidos para os elementos de um ambiente em particular.

Os atributos definidos podem ter diversos tipos de domínios, como pode ser observado no apêndice B. Entre eles estão os domínios ID_USUÁRIO e ID_GRUPO que permitem determinar um dos usuários ou grupos do ambiente como valor para o atributo. Existem também os domínios ID_OBJETO, ID_VERSAO, ID_CONFIGURAÇÃO, através dos quais é possível especificar, respectivamente, um objeto, uma versão ou uma configuração como valor do atributo.

Os valores dos atributos especificam as propriedades dos objetos, versões e configurações e podem ser usados para selecioná-los. O mecanismo de seleção através de valores de atributos é apresentado na Seção 3.5.

3.3.3 Estruturas de Derivação

Uma versão de um objeto pode ser criada a partir da derivação de outra versão do mesmo objeto. O processo de derivação das versões de cada objeto de desenvolvimento é documentada em um histórico. O MVC permite que os históricos de um ambiente sejam representados por estruturas na forma linear ou de árvore, e que o usuário navegue através desta estrutura por meio de operações apropriadas.

A estrutura é uma árvore, quando o ambiente permite o uso de versões alternativas. Cada ramificação da árvore pode representar uma alternativa no desenvolvimento de uma versão. No entanto, não existe nenhuma operação especial para a criação de ramificação, ela se dá pela simples derivação de uma versão que já tiver sucessoras. Assim sendo,

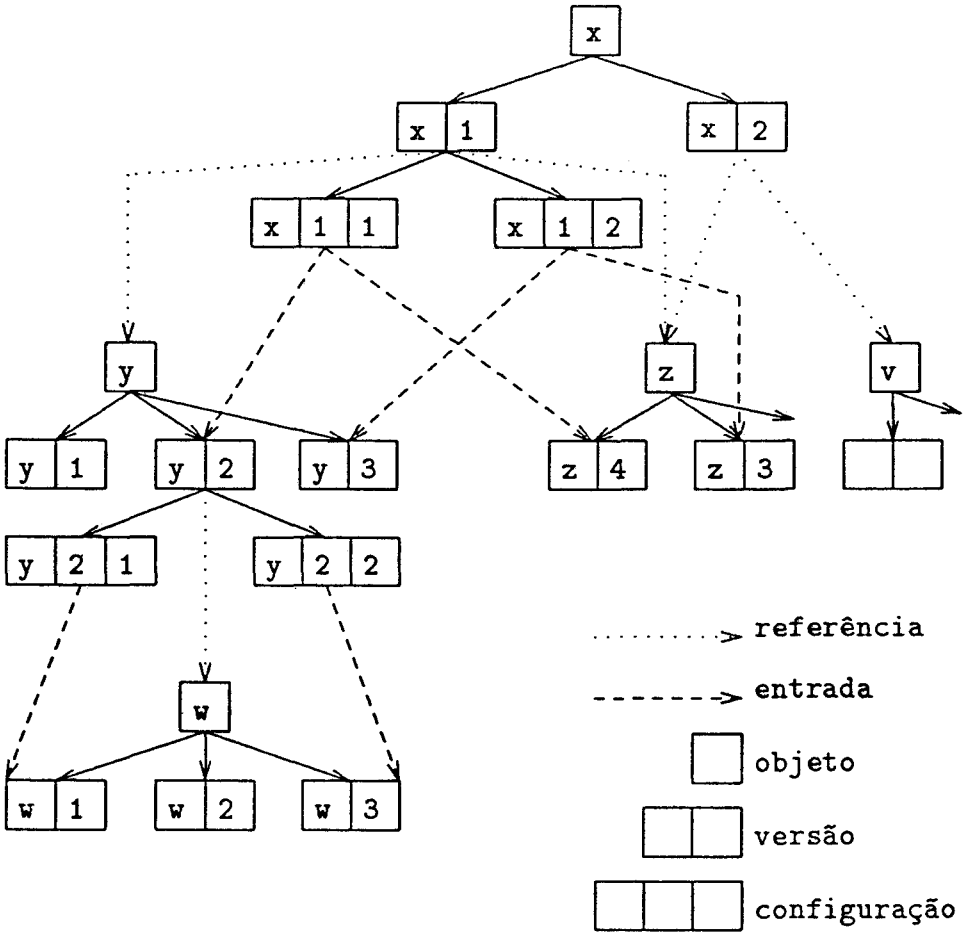

Figura 3.2: Diferentes configurações da mesma versão

é permitido que os usuários identifiquem as ramificações da estrutura de derivação por nomes escolhidos por eles, e que as versões de uma ramificação sejam selecionadas através do nome que a identifica, ou diretamente através de seus atributos. Quando o ambiente não permite o uso de versões alternativas a estrutura é linear, sendo que uma versão só pode ter uma sucessora e uma antecessora. A forma do histórico de versões é definida por um dos parâmetros que caracterizam o ambiente.

As versões de um objeto são organizadas pela sua estrutura de maneira global, isto é, não existe uma estrutura para cada banco de dados, e sim uma única estrutura que organiza todas versões existentes de um objeto, em todos os bancos de dados.

Cada estrutura organiza as versões de um objeto em uma única representação. Portanto, se um objeto tiver diferentes representações, existirão estruturas de derivação paralelas, uma para cada representação.

Durante o desenvolvimento de uma versão, pode ser necessário usar diversas configurações. Da mesma forma que as versões, as configurações de cada versão também podem ser derivadas umas da outras, e portanto são organizadas por uma estrutura. Ela tem a mesma forma que as estruturas de derivação de versões de objetos de desenvolvimento, isto é, pode ser uma árvore ou uma sequência linear. A Figura 3.3 mostra o objeto X, com suas versões, configurações e as estruturas de derivação.

Uma configuração global também pode ser derivada de outra configuração da mesma representação. Existe portanto, em cada banco de dados uma estrutura que organiza todas as configurações globais de uma mesma representação.

3.4 Inserção dos Dados

Cada usuário pode criar novos objetos, versões, configurações e configurações globais em seu banco de dados. Nas operações de criação o usuário determina, através de um parâmetro, em que partição do banco de dados o elemento criado será armazenado. A partição define as capacidades do elemento armazenado, restringindo os tipos de acesso que os diversos usuários têm sobre ele. O controle de acesso será discutido com mais detalhes na Seção 3.8.

A operação *InsObj* cria um objeto com nome e representação específicos, e inicializa a estrutura para organização de suas versões. A inserção de uma versão é feita através da operação *InsVer* e só pode se dar depois que o respectivo objeto de desenvolvimento tenha sido inserido.

Como já foi mencionado na Seção 3.3.3, o processo de criação de versões define a estrutura de derivação do objeto. Quando uma versão é derivada de outra, ela é inserida na estrutura como sucessora da versão que lhe deu origem e, a princípio, ela se apresenta como cópia da sua antecessora. Os únicos atributos que podem ter valores diferentes são *autor, data_de_criação, número e partição*. A versão criada também faz referência aos mesmos objetos que a versão antecessora. Quando a versão criada não é derivada de outra, ela é inserida na estrutura de versões como sucessora direta do objeto e não faz referência a nenhum outro objeto.

Se o ambiente tem estrutura de derivação linear, uma tentativa de derivar uma versão

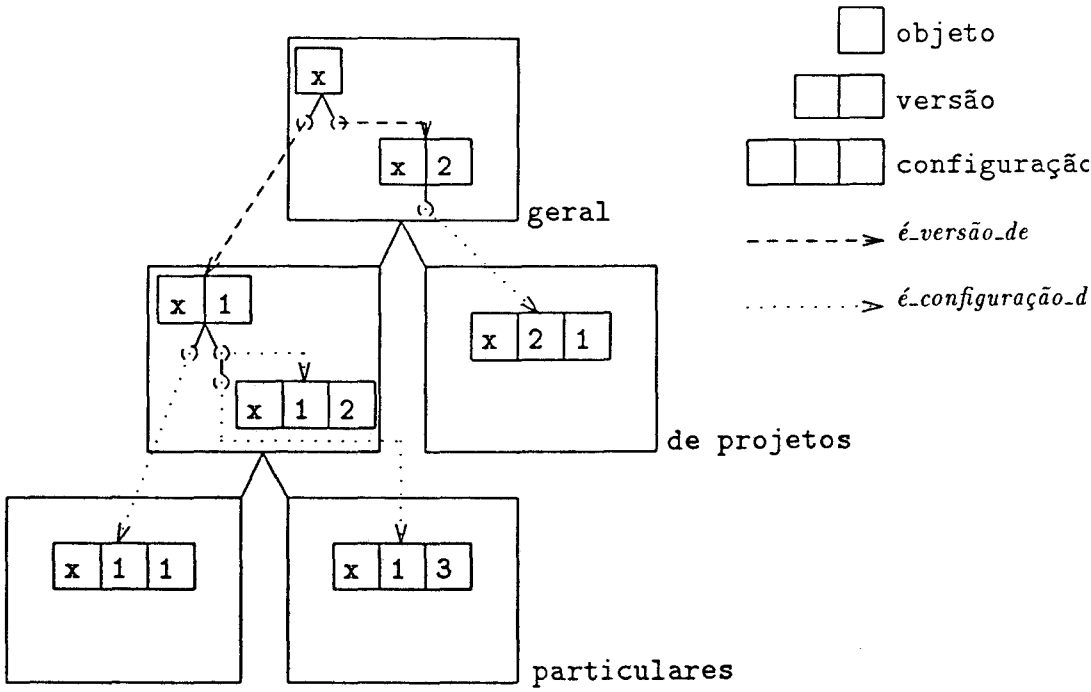

Figura 3.3: Um objeto com suas versões e configurações espalhadas pela hierarquia de banco de dados

que já tenha sucessoras não tem sucesso.

As configurações são tratadas de forma semelhante às versões. A criação de uma versão inicializa a estrutura que vai organizar as suas configurações. Assim, a operação *InsConf* só pode ser efetuada depois que a respectiva versão tenha sido criada. Se a configuração é derivada de outra, ela é inserida na estrutura que organiza as configurações como derivada da que lhe deu origem. Ela é uma cópia de sua antecessora, tendo as mesmas entradas. Caso contrário, ela é sucessora direta da versão e não contém nenhuma entrada.

Deve-se observar que, quem tem acesso a uma versão, deve ter acesso também ao objeto de desenvolvimento correspondente. Portanto, na criação de uma versão, não é suficiente que o objeto já tenha sido criado; ele deve estar armazenado no mesmo banco de dados que a versão, ou em um de nível superior a este na hierarquia. Da mesma forma, no momento da derivação de uma versão, é necessário ter acesso à versão que vai lhe dar origem e, conseqüentemente, ela também deve estar armazenada em um destes bancos de dados. Analogamente, na criação de uma configuração, a versão e a configuração antecessoras têm que estar acessíveis ao criador.

Além disso, os objetos referenciados por uma versão e as versões que compõem uma configuração devem estar acessíveis a todos usuários que tenham acesso à versão e à configuração, respectivamente. Desta forma, as operações de inserção de referência e de entradas de configuração só têm sucesso quando estas condições são respeitadas. A Figura 3.4 mostra uma distribuição possível para os elementos envolvidos na configuração X[1][1] dos exemplos anteriores.

É fácil notar que a existência das versões é dependente da existência do objeto correspondente, e o mesmo se dá com as configurações em relação a versão associada. A operação *RemObj* remove o objeto se ele não tiver versões, e a operação *RemVer* remove a versão se ela não tiver configurações. Quando uma versão é eliminada, as suas sucessoras passam a ser sucessoras da versão que a antecedia. O mesmo acontece com as sucessoras de uma configuração eliminada. E ainda, nenhum elemento pode ser removido enquanto ele estiver sendo usado por um outro usuário, ou for referenciado por alguma versão ou configuração.

3.5 Seleção

É possível selecionar objetos, versões e configurações através dos valores de seus atributos. São utilizadas para isso, regras de seleção. Uma regra de seleção é constituida por uma seqüência de cláusulas, representando expressões lógicas do tipo *or*. Cada cláusula é composta por uma seqüência de predicados que consistem em expressões lógicas do tipo *and*. Um elemento é selecionado pela regra se ele satisfizer todos os predicados de uma de suas cláusulas.

Um predicado tem a forma <atributo><operador><valor>, onde <atributo> é o nome do atributo, <operador> é representado por =, >, <, >=, <= , <>, MAX ou MIN, e <valor> é o valor do atributo no qual deve se basear a busca. Os operadores MAX e MIN selecionam os elementos que contenham o valor máximo ou mínimo encontrado

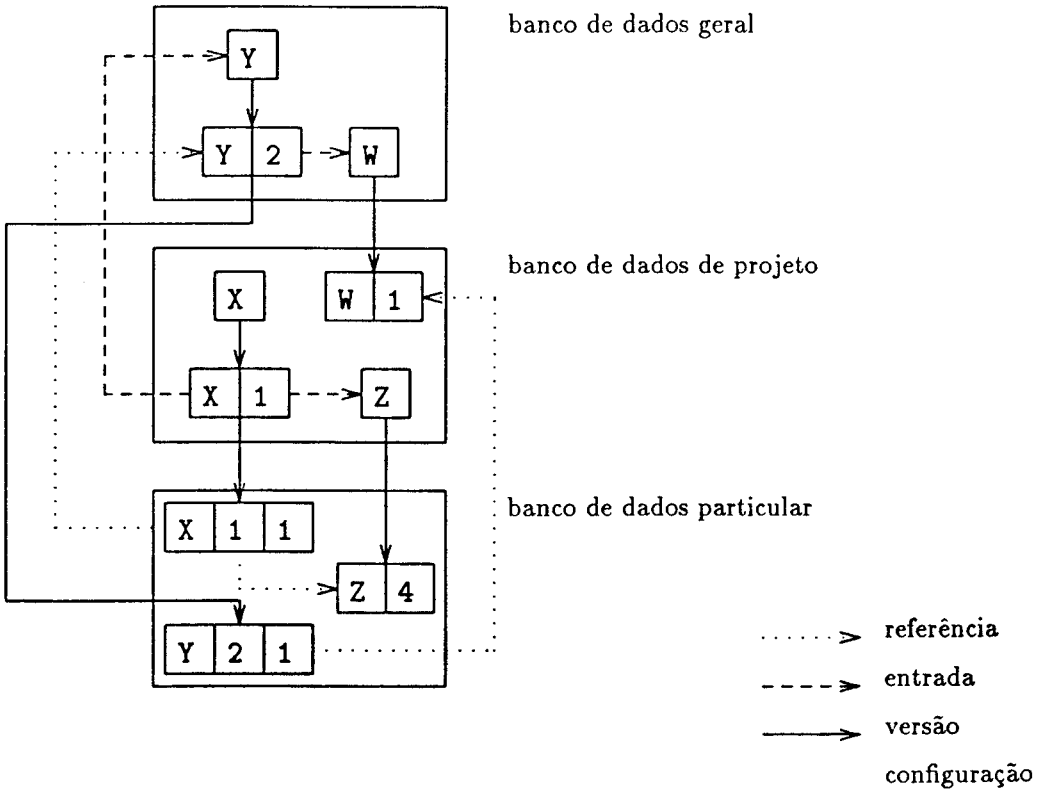

Figura 3.4: Distribuição dos componentes de uma configuração

para o atributo. Portanto, os predicados baseados nestes operadores dispensam a parte que determina o <valor>. O operador definido no predicado deve ser condizente com as comparações suportadas para o domínio do atributo. E o nome do atributo deve incluir o elemento a que ele é relacionado, isto é, deve ser especificado quando se trata de um atributo de objeto, de versão ou de configuração.

Além das cláusulas e predicados que constituem a regra de seleção, a especificação de uma consulta deve conter:

rótulo : identifica a consulta.

tipo do resultado: define se a consulta deve retornar identificadores de objetos, de versões ou de configurações.

tipo da consulta: especifica se a consulta é direta ou indireta. Uma consulta é direta quando é relacionada a um objeto específico. Para efetuar uma consulta direta, um usuário passa o identificador de um objeto e recebe o identificador de uma de suas versões ou configurações. Quando a consulta é indireta ela não precisa ser relacionada a um objeto e, ao ser efetuada, retorna um conjunto de identificadores de objetos, versões ou configurações conforme o tipo definido para o seu resultado.

parâmetros: determinam quais predicados serão baseados nos valores passados por parâmetros, quando a consulta for efetuada.

São oferecidos ainda, predicados especiais para que o usuário possa restringir a busca a determinadas partes da estrutura de derivação, ou a determinados bancos de dados.

Quando a seleção deve se restringir às versões de uma ramificação, o predicado "*ramificação* : nome de ramificação" deve ser usado. O predicado "*sucessora_direta* : número" restringe a busca às sucessoras diretas de uma versão identificadas pelo número determinado. Os predicados especiais baseados nos valores de *sucessora*, *antecessora* e *antecessora_direta* restringem a busca, respectivamente, a todas as sucessoras, a todas as antecessoras e às antecessora diretas de uma determinada versão. Estes predicados especiais podem ser usados, de forma análoga, para a seleção de configurações.

Normalmente, a busca é realizada em todos os bancos de dados a que o usuário tem acesso. Para especificar um banco de dados em que deve se dar a busca é necessário usar o predicado especial "*banco_de_dados* : tipo_bd". O tipo_bd especifica o banco de dados pelo seu tipo, isto é, seus valores podem ser CV_PARTICULAR, CV_PROJETO, CV_GERAL, indicando que a consulta deve ser efetuada no banco de dados particular, de projeto ou geral.

O rótulo ULT_DAT_LIB pode, por exemplo, ser associado a uma consulta que seleciona a última versão de um objeto do banco de dados particular que tenha sido criada depois de uma determinada data, ou a última versão do objeto em uma partição chamada LIBERADA. Esta consulta é direta e deve ter uma versão como resultado. A regra de seleção associada a consulta consistirá de duas cláusulas:

banco_de_dados : CV_PARTICULAR *and* versão.criação > CV_PARAMETRO *and* versão.número MAX *or*

versão.partição = LIBERADA *and* versão.número MAX

Uma definição da consulta é armazenada no banco de dados do usuário, podendo ser efetuada inúmeras vezes pela operação *ConsDir* se o seu tipo for direta, ou então pela operação *ConsInd*. Para efetuar uma consulta indireta, é necessário somente usar o rótulo que a identifica e fornecer os valores para os seus parâmetros. A operação retorna um conjunto de identificadores do tipo determinado para o resultado. Esse conjunto abrange os objetos, versões ou configurações que satisfazem uma das cláusulas definidas para a consulta. Se a consulta for direta, é preciso fornecer também o identificador de um objeto, e um único identificador é retornado. Uma consulta préviamente definida pode também ser usada em entradas de configuração, conforme será descrito na Seção 3.7.1.

No exemplo acima, o usuário poderia efetuar a consulta fornecendo o rótulo ULT_DAT_LIB, a data para a comparação e a identificação de um objeto. A data passada substituiria CV_PARAMETRO na regra de seleção. E então, a consulta procuraria entre as versões do objeto especificado, uma que satisfizesse a primeira cláusula. Caso não fosse encontrada nenhuma versão nestas condições, a segunda cláusula seria usada.

3.6 Operações entre Bancos de Dados

Como os dados do ambiente são distribuídos, o MVC oferece operações para a manipulação dos elementos entre dois bancos de dados diferentes. Além da operação de transferência, existe também um mecanismo para emprestar objetos, versões ou configurações. Ele permite que o usuário trabalhe com os elementos emprestados no seu banco de dados por um período prolongado de tempo. Este mecanismo envolve as transações e as operações de *check-out* e *check-in*.

3.6.1 Transação

Um usuário pode definir uma sessão de trabalho através de uma transação. O começo e o fim de uma transação são independentes da entrada e da saída do usuário no ambiente. Desta forma, o usuário pode entrar e sair do ambiente várias vezes durante uma mesma transação e pode também terminar uma transação ou começar outra sem ter que sair do ambiente.

Durante uma transação o usuário pode realizar um número arbitrário de operações de *check-out* e de resolução de configuração, que serão detalhadas nas seções seguintes.

Quando o usuário sai do ambiente e depois entra novamente, ele o encontra da mesma forma que deixou. No entanto, quando ele termina uma transação são liberados todos os elementos que tinham sofrido *check-out*, as informações sobre versões e configurações ativas, e também sobre as *configurações_resolvidas* que não haviam sido armazenadas como versões.

3.6.2 *Check-out* e *Check-in*

O controle de acessos concorrentes aos objetos, versões e configurações é feito através das operações de *check-out* e *check-in*. As operações que envolvem leitura e atualização

devem ser precedidas por uma operação de *check-out*, e esta deve estar relacionada a uma transação.

Através da operação de *check-out* o usuário faz um empréstimo de um elemento de outro banco de dados para o seu. A operação instala uma cópia do elemento no banco de dados do usuário, onde serão realizadas as futuras leituras, atualizações, resoluções e derivações. Além disso, a realização de um *check-out* em um elemento, faz com que o original fique protegido, impedindo que outros usuários o acessem de forma conflitante.

Quando o usuário realiza um *check-in* as atualizações que foram feitas na cópia são passadas para o original e a cópia é destruída. O *check-in* também faz com que a proteção do original seja liberada. Se a transação relacionada ao *check-out* for terminada sem que o usuário tenha efetuado o *check-in*, ele é realizado automaticamente.

Algumas vezes, enquanto a cópia de um elemento é lida, é necessário que o original não seja atualizado por outros usuários. Outras vezes, a atualização do original não interfere no trabalho que está sendo realizado, e assim a proteção para escrita não é requisitada na operação de leitura. Por outro lado, sempre que uma atualização está sendo feita na cópia, é necessário proteger o original de forma a impedir atualizações de outros usuários.

Além disso, quando o usuário vai derivar uma versão, é necessário que nenhum outro esteja percorrendo a estrutura de versões do objeto, pois os resultados podem ser diferentes, conforme a navegação na estrutura seja realizada antes ou depois da derivação.

Dependendo do que se deseja fazer com o elemento, e da proteção necessária, o *check-out* pode ser efetuado em diferentes modos. O modo do *check-out* determina as operações que poderão ser efetuadas com o elemento original. As operações se combinam da seguinte forma:

Modo leitura com proteção de escrita e derivação (CV_LPED): O usuário pode ler a cópia, bem como percorrer a estrutura de versões ou configurações. O elemento original está liberado para leitura de outros usuários com ou sem proteção. No entanto, outros usuários não conseguem fazer *check-out* para atualizações ou para derivações no original.

Modo leitura com proteção só de escrita (CV_LPE): O usuário pode ler a cópia e percorrer a estrutura de versões ou configurações. Contudo, o original só está protegido para atualizações de outros usuários, eles podem ler ou realizar derivações no original.

Modo leitura sem proteção de escrita (CV_LSP): Este modo de *check-out* permite que o usuário leia a cópia ou percorra a sua estrutura de versões ou configurações, mas não resulta em nenhum tipo de proteção. Portanto, qualquer modo de *check-out* é permitido no original.

Modo derivação (CV_D): É necessário dar um *check-out* nesse modo, para realizar operações de derivação. Não são permitidas atualizações nos atributos da cópia. Somente a estrutura de versões pode ser atualizada através de derivações. O original pode ser lido sem nenhuma proteção ou com proteção só de escrita.

Modo escrita (CV_E): Como é possível atualizar a cópia e derivar versões ou configurações, só é permitido ler o original sem proteção para escrita ou derivação.

3.6.3 Transferência

A transferência de um elemento consiste em removê-lo de um lugar, e reproduzí-lo em outro. Um elemento pode ser transferido para outra partição do mesmo banco de dados em que está armazenado, ou para outro banco de dados. Quando a transferência se dá entre partições de um mesmo banco de dados, na verdade, o que está sendo alterado é o tipo de acesso que cada usuário tem sobre o elemento. Por outro lado, uma transferência para outro banco de dados reflete uma alteração do conjunto de usuários que têm acesso a ele.

O sentido de uma transferência deve ser de um banco de dados para outro que esteja em um nível superior ao seu na hierarquia. Ou seja, um elemento que está em um banco de dados particular pode ser transferido para o de projeto ou para o geral, bem como um elemento que está no banco de dados de projeto pode ser transferido para o geral.

As configurações globais não podem ser transferidas para outros bancos de dados. Uma vez criadas, elas devem permanecer no mesmo banco de dados até serem removidas.

Como foi dito na Seção 3.4, a versão só pode ser inserida de forma que qualquer usuário que tenha acesso a ela, tenha também ao objeto correspondente. A operação de transferência não pode violar esta regra. Assim, a transferência de uma versão para um banco de dados superior ao banco de dados em que está armazenado o objeto correspondente, não tem sucesso. No entanto, devido a uma transferência, a versão antecessora pode ficar armazenada em um banco de dados inferior ao de sua sucessora. Desta forma, em algumas situações o usuário pode não conseguir saber qual a antecessora de uma versão, pois ele pode não ter acesso a ela. O mesmo acontece com as configurações em relação à versão correspondente e à configuração antecessora.

A operação que faz a transferência de uma versão *TraVer*, tem um parâmetro que define se alguma de suas configurações deve ser transferida juntamente com ela, e em caso positivo, qual delas.

É importante notar que, uma versão só pode ser transferida se não fizer referência a outros objetos que estejam armazenados em bancos de dados de níveis inferiores ao seu destino. E a transferência de uma configuração, seja por meio da operação *TraVer* ou pela operação *TraConf*, só tem sucesso se nenhuma de suas entradas fizer referência a uma versão ou configuração armazenada em um banco de dados de nível inferior ao seu destino.

3.7 Configuração

3.7.1 Entradas de Configuração

O trabalho de cada usuário gera uma necessidade em relação à composição do sistema, que normalmente é diferente das exigências dos outros usuários. Como o MVC suporta

diferentes configurações para a mesma versão, o usuário pode criar novas configurações do sistema que utilizem as versões adequadas para ele.

No entanto, especificar uma a uma as versões que devem compor o sistema, é muito trabalhoso. Além disso, quando for usada uma versão faz referências a outros objetos, é necessário também especificar versões para todos os objetos referenciados. Assim, uma especificação por extenso está sujeita a muitos erros. Por outro lado, compor o sistema somente através de valores *defaults* também não é suficiente.

Para facilitar o trabalho do usuário, o MVC oferece entradas de configuração de diversos tipos. Elas permitem que ele determine facilmente a composição desejada para o sistema. Conforme foi ilustrado pela Figura 3.2, as referências feitas por uma versão determinam os objetos usados por ela. No entanto, as versões dos objetos referenciados que devem compor o sistema são determinadas através das entradas de configuração.

Esta Figura ilustra entradas que ligam uma configuração a uma versão de outro objeto. Mas dependendo do seu tipo, as entradas podem determinar a composição da configuração de várias outras maneiras. São elas:

Entrada direta: A referência é feita a um objeto específico. Ela tem uma das seguintes formas:

- **total:** É determinada uma subconfiguração de um dos objetos referenciados para fazer parte da configuração. Na configuração X[1][1] da Figura 3.2, o usuário poderia inserir uma entrada direta total para a subconfiguração Y[2][2]. Desta forma, ele garantiria que a versão 2 de Y fosse usada, e que ela fosse composta por W[3].
- **parcial:** Uma versão de um dos objetos referenciados é escolhida para fazer parte da configuração. Este tipo de entrada pode ser usado tanto para as versões que não fazem referência a outros objetos, como para as que o fazem. No último caso, o gerenciador determina a subconfiguração a ser usada através de critérios que serão apresentados na Secção 3.7.3. No exemplo descrito acima, Y[2] e Z[4] são entradas diretas parciais da configuração X[1][1].
- **de consulta:** Uma consulta direta é relacionada a configuração. Ela deve ser efetuada, resultando em uma versão ou em uma subconfiguração para fazer parte da composição do sistema. O usuário poderia determinar que a última versão liberada do objeto Z seja usada na configuração X[1][1], através da entrada Z[ULTIMA_LIBERADA]. Para inserir a entrada seria necessário que o rótulo ULTIMA_LIBERADA identificasse uma consulta definida préviamente.

Entrada indireta: É determinado um conjunto de versões ou de subconfigurações que podem ser usadas, se na mesma configuração não houver entrada direta para algum dos objetos referenciados. As formas em que podem ser usadas as entradas indiretas são:

- **de inclusão:** Associa a configuração a uma outra configuração, cujas entradas podem ser usadas. Em X[1][2] poderia existir uma entrada incluindo a configuração X[1][1]. E então, se as entradas diretas de X[1][2] não fossem

suficientes para determinar versões para os objetos Y e Z, as entradas da configuração X[1][1] seriam usadas. Quando o usuário cria uma nova configuração, e pretende mudar somente algumas versões deixando o restante da mesma forma que na anterior, ele faz a inclusão da configuração anterior.

de consulta: Uma consulta indireta determina um conjunto de versões ou subconfigurações, que podem ser usados. Uma configuração pode ter uma entrada indireta, como por exemplo [ULT_DO_AUTOR][José]. Supondo que o rótulo ULT_DO_AUTOR, esteja associado a uma consulta que seleciona todas as últimas versões criadas por José, a configuração usaria a última versão criada por ele, para os objetos referenciados que não tenham entradas diretas correspondentes.

Entrada *default*: Um número de versão, um número de configuração ou uma consulta direta é especificado. Caso as entradas diretas e indiretas não forem suficientes para determinar as versões para todos os objetos referenciados, a entrada *default* é usada para cada um deles.

Objeto	Versão	Referências	Configuração		
			Número	Tipo	Entrada
Z	1	A, B	2	Direta	A[4]
				Default	[ULTIMA_LIBERADA]
	2	A, B, C	1	Direta	C[1][2]
				Indireta	[1][2]
			2	Direta	C[1][4]
					A[4][5]
				Default	ULTIMA_LIBERADA]
A	4	E	5(R)	Direta	E[3][2]
B	9	-	-	-	-
C	1	D	2	Direta	D[5]
			4	Direta	D[ULTIMA_LIBERADA]
D	5	G, H, I	1	Default	[ULTIMA_LIBERADA]
			2	Indireta	[ULT_DO_AUTOR][José]
E	3	F	2(R)	Direta	F[5]
F	5	-	-	-	-
G	9	-	-	-	-
H	3	-	-	-	-
I	6	-	-	-	-

Tabela 1 - Exemplo de algumas configurações, versões e objetos

Obs: O (R) ao lado do número significa que esta é uma configuração no estado *resolvida*.

As entradas diretas e indiretas são consideradas na ordem em que foram inseridas na configuração. Assim, se duas entradas estabelecem versões diferentes para um mesmo objeto referenciado, só a primeira delas é considerada. Além disso, nas entradas indiretas o usuário deve ter o cuidado de usar consultas que resultem em uma única versão ou configuração de cada objeto de desenvolvimento, pois os elementos são considerados na ordem em que são encontrados.

Através do exemplo da Tabela 1, é possivel compreender a diferença entre uma entrada direta total e uma entrada indireta de configuração. A versão Z[2] faz referência aos objetos A, B e C, e as versões usadas para eles devem ser determinadas através das entradas de uma configuração de Z[2]. A configuração 1 da versão Z[2], tem uma entrada direta total (C[1][2]) e outra indireta de inclusão ([1][2]). A primeira entrada determina que a versão C[1] faça parte da composição do sistema, e que as versões dos objetos referenciados por C[1], sejam determinadas através da configuração C[1][2]. No caso, a versão do objeto D seria determinada por esta configuração. A segunda especifica que as entradas da configuração Z[1][2] devem ser usadas para os objetos aos quais Z[2] faz referência e que não existam entradas diretas correspondentes em Z[2][1]. Portanto, as versões de A e B devem ser determinadas através da configuração Z[1][2].

3.7.2 Ativação

Em determinadas situações, a versão ou configuração ativa é escolhida para fazer parte da composição do sistema. Na Seção 3.7.3 estão detalhadas as condições em que elas são usadas.

Quando um usuário vai trabalhar com uma versão ele a traz para o seu banco de dados através do mecanismo de *check-out/check-in* descrito na Seção 3.6.2. Para o usuário que efetuou a operação, aquela versão se torna ativa em relação ao objeto de desenvolvimento. A versão permanece ativa para o usuário até que ele faça um *check-in* naquela versão ou um *check-out* em outra versão do mesmo objeto.

A operação de *check-out* quando efetuada em uma configuração a torna ativa em relação ao objeto de desenvolvimento e em relação à versão. Se o usuário fizer um *check-out* em outra configuração do mesmo objeto mas não da mesma versão, a primeira deixa de ser ativa em relação ao objeto mas continua sendo a configuração ativa em relação à versão. A Figura 3.5 ilustra esta situação.

Da mesma forma, as configurações globais também são ativadas pelos usuários, quando eles as utilizam para leitura ou atualização.

3.7.3 Resolução de Configuração

Para se usar um sistema é necessário determinar exatamente uma versão de cada objeto que o compõe; portanto, é preciso resolver a sua configuração. A resolução de uma configuração percorre suas entradas determinando uma versão para cada objeto referenciado, e quando a versão escolhida faz referência a outros objetos, é determinada também uma configuração para esta versão, que será resolvida recursivamente. Desta forma, quando

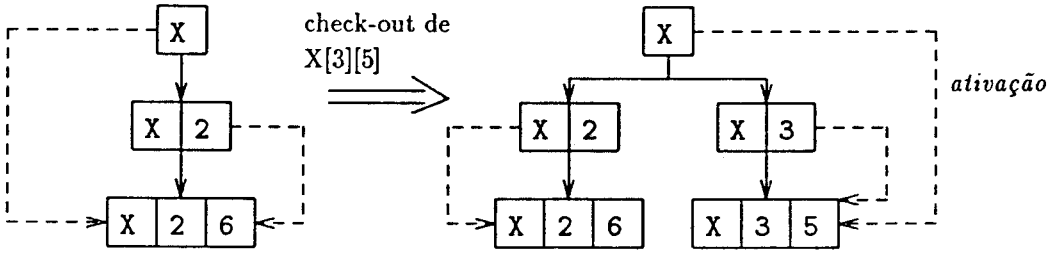

Figura 3.5: Relacionamentos de ativação de uma configuração

uma configuração é resolvida, cria-se uma nova configuração que consiste somente de entradas diretas totais ou parciais.

Na resolução das entradas de uma configuração, são consideradas em primeiro lugar as diretas. Se existirem objetos usados pela versão sendo construída, que não foram especificados nas entradas diretas, procura-se nos conjuntos definidos pelas entradas indiretas. Se ainda restarem objetos usados para os quais não foram determinadas as versões, a entrada *default* é usada.

Quando todas as entradas de uma configuração já tiverem sido resolvidas, e ainda não tiver sido determinada uma versão para cada objeto referenciado, a configuração global ativa é usada. As configurações globais são compostas pelos mesmos tipos de entradas que as outras configurações, e elas serão percorridas na mesma ordem descrita anteriormente para as configurações comuns.

A utilização das versões e configurações ativas em uma resolução pode ser feita com diferentes prioridades. Se a prioridade for 0, antes de percorrer as entradas diretas, o mecanismo procura por versões ou configurações ativas dos objetos referenciados. Se elas forem encontradas, serão usadas. Posteriormente, as entradas da configuração em resolução são percorridas para determinar versões para os objetos que não tenham versões ou configurações ativas. Na resolução com prioridade 1, as entradas diretas são resolvidas e só depois são consideradas as versões e configurações ativas. Elas são procuradas depois das entradas indiretas se a prioridade for 2, e depois da entrada *default* caso a prioridade seja 3. Quando a prioridade é 4, as versões e configurações ativas só são usadas se depois de resolvidas todas as entradas da configuração global, as versões dos objetos referenciados não estiverem todas determinadas. Finalmente, elas nunca são consideradas se a prioridade for 5. Independendemente da prioridade usada, as versões ativas são procuradas antes das configurações ativas.

Durante uma resolução, pode aparecer uma entrada direta parcial, que determina só a versão que deve ser usada, sendo que esta versão faz referência a outros objetos. Para completar o processo, a configuração desta versão deve ser resolvida. Mas qual configuração usar? Se a prioridade das configurações ativas for 4 ou 5 as entradas da

configuração global são usadas. Caso contrário, procura-se antes pela configuração ativa para aquela versão; se existir, ela será usada. As entradas da configuração global serão usadas, se não existir configuração ativa para aquela versão, ou se suas entradas forem insuficientes para a resolução.

Para permitir que o usuário possa trabalhar com uma configuração que foi resolvida, o gerenciador realiza um *check-out* em cada versão ou configuração envolvida na resolução, instalando uma cópia no banco de dados do usuário com a proteção especificada por ele. É possível evitar que outro usuário atualize as referências feitas por uma das versões, ou mesmo as entradas de uma subconfiguração. Para tanto, o usuário deve resolver a configuração com proteção de escrita, ou seja, determinando que o modo das operações de *check-out* seja CV_LPED, CV_LPE ou CV_E. No entanto, se algum destes elementos já tiver sofrido um *check-out* para escrita antes da resolução, a operação falha.

Através da operação de resolução *ConfRes*, é criada no banco de dados do usuário um objeto chamado *configuração_resolvida*, que contém exclusivamente entradas diretas. Em outras palavras, uma *configuração_resolvida* é composta por versões que não fazem referências e por outras *configurações_resolvidas*. Por meio das entradas da *configuração_resolvida* o usuário conhece a composição do sistema, e tem acesso a todas as versões e configurações que sofreram *check-out* devido à resolução. Assim, o usuário pode ler as entradas para conhecer a composição do sistema resultante. Caso ele não esteja satisfeito, ele pode fazer alterações nas versões e configurações e fazer a resolução novamente.

Este processo de resolução de configuração tem diversos detalhes, que serão melhor compreendidos com um exemplo.

3.7.4 Um exemplo de configuração

Um bom exemplo do processo de resolução é o da configuração Z[2][2] mostrada na tabela 1, pelo usuário identificado por id1. Neste resolução é usada a com prioridade 1 para versões e configurações ativas. Na tabela 2 são mostrados os resultados obtidos para as consultas efetuadas durante a resolução e na tabela 3 está a configuração global ativa para o usuário id1 sendo que suas versões e configurações ativas estão na tabela 4.

Neste exemplo, o rótulo ULTIMA_LIBERADA identifica a consulta que seleciona a última versão do objeto que esteja na partição liberada, com as seguintes cláusulas:

$$\textit{partição} = \text{liberada } \textit{and } \textit{número } \text{MAX}$$

A consulta identificada pelo rótulo ULT_DO_AUTOR seleciona todas as últimas versões que têm o valor do atributo *autor* igual ao valor passado pelo parâmetro "nome", através da regra

$$\textit{versão.autor} = \text{CV\_PARAMETRO } \textit{and } \textit{versão.número } \text{MAX}$$

Consulta	Resultado
B[ULTIMA_LIBERADA]	B[9]
D[ULTIMA_LIBERADA]	D[5]
I[ULTIMA_LIBERADA]	I[6]
[ULT_DO_AUTOR][José]	A[6]
	B[4]
	C[3]
	H[3]

Tabela 2 - Resultados das consultas efetuadas

Tipo entrada	Entrada
Direta	A[3]
Direta	B[5]
	G[9]
Default	[ULTIMA_LIBERADA]

Tabela 3 - Composição da configuração global ativa para o usuário id1

Objeto	Versão Ativa	Conf. Ativa	Versão	Conf. Ativa para a Versão
D	[3]	[3][2]		
			[4]	[3]
			[5]	[2]
B	[9]			

Tabela 4 - Versões e configurações ativas para o usuário id1

A versão 2 de Z faz referência aos objetos A, B e C, e a configuração Z[2][2] tem entradas diretas para A e C.

A primeira entrada determina que a configuração C[1][4] deve ser usada. A versão C[1] faz referência a D e, portanto, é necessário resolver sua configuração. Em C[1][4] existe uma entrada direta D[ULTIMA_LIBERADA]. Como pode ser visto na tabela 3 a consulta D[ULTIMA_LIBERADA] resulta na versão 5. Esta versão faz referência aos objetos G, H e I; portanto, deve-se resolver a sua configuração. Como a entrada que resultou em D[5] não especificou a configuração, a configuração ativa em relação a versão D[5] será usada, que segundo a tabela 4 é D[5][2].

Esta configuração não tem entradas diretas, e não existem versões ou configurações ativas dos objetos G, H e I. A configuração D[5][2] consta de uma entrada indireta [ULT_DO_AUTOR][José], e no conjunto das últimas versões de José, encontra-se a versão H[3]. Como falta determinar uma versão para G e I, a configuração global é consultada. Nela existe uma entrada direta que determina a versão G[9] como componente do sistema. A configuração global tem também a entrada *default* [ULTIMA_LIBERADA] que, aplicada ao objeto I, resulta na versão 6. Como G[9], H[3] e I[6] não fazem referências a outros objetos, não é preciso resolver as suas configurações.

Continuando a resolução de Z[2][2], encontra-se a entrada direta para a configuração A[4][5]. Esta configuração está no estado *resolvida*, cujo significado será descrito na Secção 3.7.3. Através de sua entrada direta encontra-se a configuração E[3][2], que

também está no estado *resolvida*. A configuração E[3][2] tem uma entrada direta para a versão F[5], que não faz referências.

Em Z[2][2] não existe entrada direta para B, mas existe uma versão ativa do objeto B para o usuário id1. Assim, a versão usada será B[9], que não faz referência a outros objetos.

Conclui-se dessa maneira o processo. A tabela 5 mostra qual seria a *configuração_resolvida* encontrada no banco de dados do usuário id1, após a operação. Observe que a notação [rel n] no lugar do número da configuração indica que a configuração é o resultado da resolução da configuração n.

Objeto	Versão	Configuração	
		Número	Entrada
Z	2	[rel 2]	A[4][rel 5]
			B[9]
			C[1][rel 4]
A	4	[rel 5]	E[3][rel 2]
B	9	-	-
C	1	[rel 4]	D[5][rel 2]
D	5	[rel 2]	G[9]
			H[3]
			I[6]
E	3	[rel 2]	F[5]
F	5	-	-
G	9	-	-
H	3	-	-
I	6	-	-

Tabela 5 - *Configuração_resolvida*

3.7.5 Transformação da *Configuração_resolvida*

Quando o usuário resolve uma configuração, não é gerada uma nova versão de configuração. Assim, se em seguida ele resolver outra vez a mesma configuração, a *configuração_resolvida* que estava no seu banco de dados é substituida pelo resultado da operação.

Se o usuário fizer a resolução com proteção para escrita, ele pode depois transformar a *configuração_resolvida* em uma configuração comum através da operação *StoConf*. A partir de então a nova configuração passa a ser acessível a outros usuários. Ela é disposta na estrutura de configurações como sucessora da configuração que foi resolvida inicialmente. O mesmo se dá com todas as suas subconfigurações. Se o usuário não julgar necessário guardar a versão da *configuração_resolvida*, ele deve liberá-la, tirando desta forma a proteção de todas versões e configurações que estavam protegidas devido à resolução. Isto é feito realizando uma operação de *check-in* na configuração resolvida.

Cada configuração tem um atributo que mantém o seu estado. Uma configuração está no estado *resolvida* quando é criada a partir da transformação de uma *configuração_resolvida*, ou é derivada de outra configuração neste estado. Ela conserva este valor para o atributo, até que suas entradas ou as referências da versão à qual ela é relacionada sejam atualizadas. Então, ela passa a para o estado *não_resolvida*.

Na operação *StoConf*, a subconfiguração que se originou de outra transformação e que se mantém inalterada, não é transformada. Para evitar a transformação não é suficiente que a subconfiguração esteja no estado *resolvida*. Todas as subconfigurações dos níveis inferiores a ela também devem estar neste estado. Além disso, nenhuma das versões referenciadas por entradas diretas destas configurações podem fazer referência a outros objetos. Se esta subconfiguração também fosse transformada em uma nova configuração as duas seriam idênticas. Em vez disso, a configuração transformada no nível superior tem uma entrada direta total, que a relaciona com a subconfiguração original.

Tome como exemplo a subconfiguração A[4][5] do exemplo anterior. Não há necessidade de derivar uma nova configuração para documentar a sua resolução, ao contrário de C[1][4] e D[5][2]. Se a configuração Z[2][rel 2] for transformada e se as últimas configurações existentes de Z[2], C[1] e D[5] forem, respectivamente, [5], [8], e [7], a configuração gerada teria a seguinte composição:

Objeto	Versão	Referência	Configuração		
			Número	Tipo	Entrada
Z	2	A, B, C	6	Direta	A[4][5]
					B[9]
					C[1][9]
A	4	E	5	Direta	E[3][2]
B	9				
C	1	D	9	Direta	D[5][8]
D	5	G, H, I	8	Direta	G[9]
					H[3]
					I[6]
E	3	F	2	Direta	F[5]
F	5				
G	9				
H	3				
I	6				

Tabela 6 - Configuração gerada a partir da *configuração_resolvida*

Desta forma, é possível manter o controle das configurações já resolvidas, evitando a proliferação de configurações desnecessárias. Além disso, para o usuário transformar uma *configuração_resolvida*, ele deve ter permissão para derivação da configuração original como será visto na Seção 3.8. Não haverá problemas, se ele usar uma configuração que se mantém totalmente *resolvida*, para a qual ele não tenha permissão para derivação.

Pois ele não vai criar uma nova configuração, só vai usar a existente em uma entrada de outra configuração, o que sempre é permitido quando se tem acesso à configuração.

Ainda assim, um usuário pode liberar uma configuração após a sua tranformação e outro querer usá-la novamente. Se ela ainda não foi alterada, é possível realizar de uma só vez o *check-out* de todas as suas subconfigurações e versões relacionadas. Porém, um terceiro usuário pode ter atualizado as referências feitas por uma das versões envolvidas na configuração, ou mesmo uma de suas entradas. Para evitar que alguém utilize inadvertidamente uma configuração imaginando que ela esteja no estado *resolvida*, o usuário pode se certificar do seu estado através da operação *VerRes*. Esta operação percorre recursivamente a configuração verificando o estado de todas as subconfigurações.

3.8 Controle de Acesso

Os objetos, versões e configurações têm diferentes capacidades. Existem, por exemplo, versões que podem ser atualizadas por todos os integrantes de um grupo, e outras que só podem ser lidas, sendo que qualquer usuário do ambiente pode fazê-lo. A distribuição dos dados apresentada na Seção 3.2 estabelece os usuários que têm acesso a um elemento, mas não o tipo de acesso.

O **MVC** divide os elementos de cada banco de dados em várias partições, e cada partição define o tipo de acesso permitido a cada usuário. Ou seja, através das partições são dadas permissões aos usuários de maneira a preservar as capacidades dos elementos que ela armazena. Uma operação efetuada por um usuário em um elemento só tem sucesso se ele tiver permissão para aquela operação na partição em que o elemento está armazenado.

As partições de cada banco de dados, e a lista de operações permitidas em cada uma delas são definidas pelos parâmetros que caracterizam o ambiente, segundo a sintaxe descrita no Apêndice B.

As operações controladas para cada partição são: inserção, eliminação, transferência e atualização de qualquer elemento e também a derivação de configurações e de versões. A operação de leitura é sempre permitida a todos que tenham acesso ao banco de dados.

As configurações globais são tratadas de forma especial, com respeito ao controle de acesso. O modelo não permite transferência de configuração global, só o dono do banco de dados pode inseri-las e eliminá-las. Dependendo da partição em que está uma configuração global, os usuários que têm acesso a ela podem ter direito de realizar derivações e atualizações.

Para derivar uma versão ou configuração, o usuário deve ter direito de derivação na partição em que está a antecessora, e direito de inserção na que será colocada a nova versão ou configuração. Ao passo que, a transferência envolve o direiia envolve o direito de transferência da partição onde está o elemento, e o direito de inserção na partição para onde o elemento vai. A verificação da permissão para derivação é feita no momento da derivação, e não no *check-out*. Assim, em determinadas situações, o usuário pode conseguir dar um *check-out* para derivação, e não conseguir realizar a derivação.

Normalmente, a definição das permissões se baseiam nos tipos dos usuários que têm

acesso ao banco de dados, ou seja, uma permissão é dada a todos os usuários particulares que têm acesso ao banco de dados, ou aos representantes dos projetos e do ambiente.

Conforme foi mencionado na Seção 3.3.2, os atributos que têm o domínio definido como ID_USUÁRIO e ID_GRUPO identificam um usuário ou um grupo deles. É possível também associar uma permissão a um destes atributos. Uma partição pode, por exemplo restringir a operação de atualização de uma versão somente ao seu criador, através do atributo *autor* mantido pelo gerenciador. No entanto, os atributos definidos para cada ambiente também podem ser usados para controlar o acesso. Por exemplo: os objetos de um ambiente podem ter o atributo *grupo_responsavel* para registrar o grupo que é responsável pelo seu desenvolvimento, e uma partição pode dar permissão para derivação aos integrantes deste grupo.

A tabela 8 dá um exemplo de como os bancos de dados de um ambiente poderiam ser divididos em partições.

Banco de Dados	Partições
Geral	Liberadas e Em_teste
Projeto	Efetivas e Em_teste
Particular	Em_desenvolvimento

Tabela 8 - Partições dos bancos de dados de um ambiente

Cada uma destas partições e as respectivas operações seriam definidas separadamente. O controle de acesso do ambiente poderia ser definido de tal forma, que a partição *Em_teste* do banco de dados de projeto, fosse usada para o desenvolvimento dos objetos definidos pelo representante do projeto.

Assim, ele criaria o objeto, atualizaria o campo *descrição* deste objeto com a informação das suas características, e escolheria um dos integrantes do grupo para ser o responsável pelo seu desenvolvimento. Só este usuário poderia criar e atualizar versões para este objeto. Quando julgasse necessário, poderia transferir uma das versões do seu banco de dados para o de projeto.

Os outros usuários poderiam ler as versões do objeto que já estivessem no banco de dados de projeto, e usá-las em outras configurações, bem como criar novas configurações para esta versão. As configurações de outros usuários também seriam colocadas no banco de dados de grupo, quando o compartilhamento fosse necessário. Mesmo estando no banco de dados de projeto só o criador das versões e configurações poderia atualizá-las ou removê-las. Além disso, um integrante do grupo poderia inserir no seu banco de dados uma configuração global derivada de outra do banco de dados de projeto, mas só o representante do grupo poderia fazer atualizações em configurações globais armazenadas no banco de dados de projeto.

Para controlar os elementos, segundo estas regras, algumas permissões devem se basear nos atributos dos elementos. Assim, cada objeto deve ter o atributo *responsável* para controlar as derivações. E o atributo *autor* mantido pelo gerenciador é usado para controlar as atualizações. Esta partição deveria ter a seguinte definição:

```
particao Em_teste bd projeto

insercao
   objeto (projeto);
   versao (objeto.responsavel);
   configuracao (projeto, particular);

transferencia
   objeto (projeto);
   versao (projeto);
   configuracao (projeto);

remocao
   objeto (projeto);
   versao (projeto, versao.autor);
   configuracao (projeto, configuracao.autor);

atualizacao
   objeto (projeto);
   versao (versao.autor);
   configuracao (configuracao.autor);
   configuracao global (projeto);

derivacao
   versao (objeto.responsavel);
   configuracao (projeto, particular);
   configuracao global (projeto, particular);
```

A definição das propriedades de uma partição deve ser feita de modo a refletir a política de controle de acesso do ambiente em questão. No entanto, como nenhuma política em particular é adotada pelo modelo, a coerência das permissões dadas não pode ser mantida automaticamente. Este cuidado é deixado a cargo de quem faz a definição.

3.9 Considerações Finais

O MVC pode ser adaptado às características de cada ambiente através de seus parâmetros. Assim, as representações e os atributos definem as propriedades dos elementos. A forma das estruturas de versões pode restringir o mecanismo de derivação. Finalmente, as partições e as respectivas permissões refletem o controle de acesso do ambiente.

Além disso, a existência de várias configurações para cada versão e a distribuição das versões e configurações pela hierarquia de banco de dados estabelecem uma base adequada para o desenvolvimento de software em grupos de projeto.

Capítulo 4

Implementação

A implementação feita para o MVC será apresentada neste capítulo. Primeiramente serão descritas as diversas partes que compõem o software, e alguns aspectos envolvidos nas decisões que levaram a esta estrutura. Em seguida, são mostradas com mais detalhes as características de um SGBD orientado a objetos, e a influência exercida por elas na implementação do modelo. Finalmente, é detalhada a implementação das funções mais importantes.

4.1 Considerações Iniciais

O armazenamento das informações é feito em um banco de dados, pois isto oferece uma maior integração do ambiente como já foi discutido no Capítulo 2. Para a implementação do MVC, foi escolhido o SGBD **Damokles** [Abram88], que tem características muito atraentes para a implementação do gerenciador. Ele oferece facilidades para manipulação de objetos estruturados e de versões de objetos. Além disso, ele é capaz de manipular diversos bancos de dados distribuídos em uma rede. No entanto, este SGBD não oferece facilidades para alteração dinâmica do esquema, o que restringe de certa forma a implementação.

Dentre as diversas estratégias que poderiam ser utilizadas para a implementação do modelo, foram considerados o uso de uma meta base de dados e a construção de um gerador de gerenciadores de versões e configurações.

Na primeira opção, todas as informações relativas à caracterização do ambiente, isto é, os valores dos parâmetros do modelo, ficariam armazenados em um banco de dados especial, chamado de meta base de dados; enquanto os objetos, versões, configurações e os outros dados manipulados pelo gerenciador, ficariam em bancos de dados comuns. As operações oferecidas pelo modelo seriam implementadas de forma a fazer consultas à meta base de dados, obtendo as informações necessárias, para depois manipular os dados dos usuários armazenados nos bancos de dados comuns. A operação de derivação de versão, por exemplo, consistiria em duas partes. Inicialmente, seria feita uma consulta à meta base de dados para determinar o tipo de estrutura de versões e os direitos de

acesso do usuário. E então, com base nestas informações seria feita a inserção de uma nova versão no banco de dados do usuário.

Optando-se pela implementação de um gerador, a especificação dos parâmetros que caracterizam o ambiente seria processada gerando um gerenciador de versões e configurações específico para o ambiente. A especificação poderia ser feita através de declarações em uma linguagem definida para este fim, ou mesmo utilizando um outro banco de dados.

A alteração das características do ambiente seria difícil, caso o modelo fosse implementado como um gerador, pois tanto as operações do modelo como o esquema do banco de dados seriam gerados a partir da definição das características do ambiente. O processo de alteração seria especialmente difícil, em uma implementação utilizando um SGBD como o **Damokles**, que não oferece facilidades para modificações dinâmicas do esquema.

Se a meta base de dados fosse utilizada, seria fácil controlar alterações nas características do ambiente. Isto consistiria apenas em atualizações na meta base de dados. É claro que existiriam algumas restrições quanto ao tipo de atualizações permitidas. Por exemplo, poderiam ser adicionados novos atributos, mas os atributos existentes não poderiam ser eliminados se ainda houvessem valores para eles.

Provavelmente, as características do ambiente alteradas mais frequentemente seriam os atributos dos objetos, versões e configurações. Uma terceira opção considerada para a implementação foi a de restringir os tipos de domínios dos atributos, permitindo que a sua definição fosse feita dinamicamente, enquanto que o restante das definições continuaria sendo feito de forma estática. No entanto, poderia haver algum atributo essencial para um ambiente, cujo domínio não se enquadrasse entre aqueles pré-definidos.

Por outro lado, um gerenciador de versões e configurações lento, em um ambiente de desenvolvimento de software, tende a ser negligenciado pelos usuários. E embora a implementação utilizando uma meta base de dados ofereça uma flexibilidade maior em relação à outra, ela penaliza muito a eficiência do gerenciador. Portanto, a implementação de um gerador foi escolhida. Para esta decisão foi levado em conta que o problema de alteração das características do ambiente pode ser contornado razoavelmente, com a utilização de um mecanismo que salve todas as informações contidas no banco de dados em um arquivo, altere o seu esquema, e depois carregue todos os dados novamente.

4.2 Estrutura do Software

Para que a declaração dos parâmetros seja feita através de uma linguagem específica, houve a necessidade da especificação formal desta linguagem. A sintaxe da linguagem foi descrita usando-se o formalismo BNF (Backus Naur Form), resultando em uma gramática recursiva à esquerda e não ambígua. Sua descrição encontra-se no Apêndice B.

A definição das características do ambiente consiste em sentenças da linguagem, que passam pelos analisadores léxico e sintático para gerar o gerenciador. Os analisadores por sua vez, são gerados pelas ferramentas **Lex** e **yacc** [Schr] a partir da definição da gramática da linguagem.

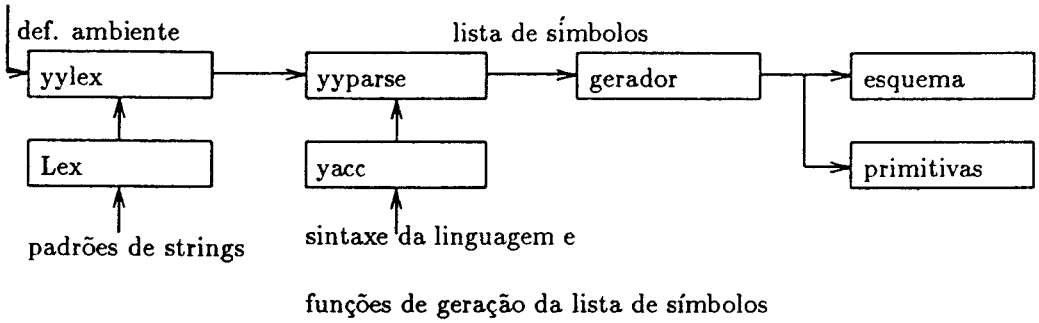

Figura 4.1: Estrutura do software

O gerador **Lex** produz um programa que reconhece expressões regulares. Ele é projetado para fazer o processamento léxico de *cadeias de caracteres*, a partir de uma especificação de alto nível para o *matching*. As expressões regulares são determinadas pelo usuário, na especificação dos fontes dada ao **Lex**. Assim, o fonte fornecido ao gerador de programas consiste em uma tabela de expressões regulares com fragmentos de programas correspondentes. A tabela é transformada em um programa chamado *yylex*, que lê uma cadeia de entrada, copia a cadeia de entrada em uma cadeia de saída, e particiona a entrada em *strings* que "casam" com as expressões regulares dadas. Além disso, o *yylex* faz com que, quando um *string* é reconhecido, o fragmento de programa correspondente a ele seja executado.

O programa **yacc** provê uma ferramenta para impor estrutura na entrada de um programa. O usuário do **yacc** fornece a especificação da entrada, que inclui as regras que descrevem a sua estrutura, o código a ser invocado quando estas regras são reconhecidas, e uma rotina de baixo nível para fazer a entrada básica. O programa **yacc** então gera uma função chamada *yyparser* para controlar o processo de entrada. Esta função chama uma rotina de baixo nível suprida pelo usuário (o analisador léxico) para reconhecer os itens básicos ou *tokens* da cadeia de entrada. Estes *tokens* são organizados de acordo com as regras de estrutura inseridas pelo usuário. Quando uma destas regras for reconhecida, o código especificado pelo usuário para esta regra é invocado. As ações têm a habilidade de retornar valores e podem também fazer uso dos valores de outras ações.

É particularmente fácil integrar o uso do **Lex** e do **yacc**. O **Lex** é usado como um pré-processador para o **yacc**. Ele particiona a cadeia de entrada, enquanto o **yacc** atribui estrutura às partes resultantes.

Da integração do **Lex**, do **yacc** e de funções para a manipulação dos símbolos reconhecidos, é gerado um programa que analisa uma cadeia de entrada e constrói uma lista ligada. Cada elemento da lista contém um símbolo e um apontador para uma estrutura que armazena o valor deste símbolo. O tipo da estrutura depende do tipo do

símbolo. Por exemplo, um símbolo do tipo representação tem como valor uma lista de atributos específicos dos objetos, versões e configurações desta representação. Enquanto que um símbolo do tipo partição, tem como valor uma estrutura que armazena todas as permissões relacionadas a esta partição. Paralelamente à lista com todos os símbolos da definição, são formadas listas auxiliares que agrupam os símbolos segundo o seu tipo. Assim, existe um lista dos símbolos do tipo representação, uma lista dos símbolos do tipo partição, e uma lista para cada um dos outros tipos de símbolos.

Quando toda a cadeia de entrada tiver sido analisada, a lista de símbolos construída é usada para gerar o esquema do banco de dados que vai armazenar os objetos, versões e configurações. O Apêndice C descreve a geração do esquema.

Além do esquema, a partir da lista de símbolos são geradas funções que serão utilizadas pelas operações de manipulação dos elementos do modelo. A lista de símbolos do tipo partição é usada para gerar a função de verificação de permissão. A implementação das operações que fazem a inserção, derivação, atualização etc, chama a função de verificação de permissão gerada. Através da identificação do usuário, do tipo da operação que ele pretende realizar, dos atributos dos elementos envolvidos na operação e das partições que os contêm, a função concede ou não a permissão para que a operação seja realizada.

Desta forma, o usuário do gerador de gerenciadores de versões e configurações especifica as características do seu ADS através da linguagem de definição. O gerador processa esta especificacão e devolve as funções que serão usadas posteriormente, pelos usuários do gerenciador de versões e configurações de um ADS específico. Um esquema das ferramentas utilizadas e do fluxo de dados pode ser visto na Figura 4.1.

4.3 Utilização do Banco de Dados

4.3.1 Controle de Acesso

O SGBD **Damokles** manipula uma hierarquia de bancos de dados que podem estar distribuídos em uma rede, sendo que cada um deles tem um proprietário. Um banco de dados é particular se seu proprietário for um só usuário, ou público se ele for de propriedade de um grupo de usuários. É permitido ainda formar uma hierarquia de grupos, ou seja, um grupo pode ser composto por outros grupos, não existindo limites para o número de níveis de agrupamento que compõem a hierarquia.

Os direitos de acesso aos dados são determinados a partir da composição da hierarquia de bancos de dados e dos respectivos proprietários. Estes direitos podem ser descritos de acordo com a Figura 4.2:

Escrita: Todo usuário tem todos os direitos em seu próprio banco de dados. O de *escrita* é o único direito relacionado exclusivamente ao dono do banco de dados.

Leitura: Um usuário ou um grupo tem o direito de *leitura* para todos os bancos de dados dos grupos, dos quais ele é membro direto ou indireto. Por exemplo: o usuário U1 tem direito de leitura nos bancos de dados D1, D2 e D3.

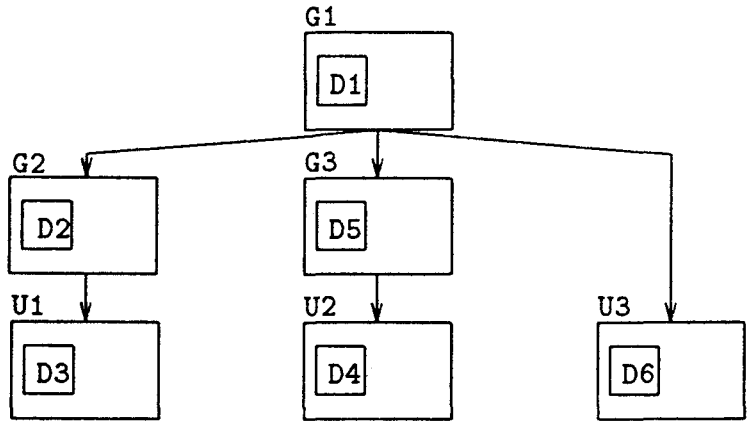

Figura 4.2: Uma estrutura de grupos de usuários do Damokles

Transferência: A *transferência* é um direito dos membros diretos de um grupo para os bancos de dados do grupo, ou para os de outros membros do mesmo grupo. Isto significa que o usuário U1 pode transferir ou copiar um objeto do seu próprio banco de dados D3 para D2 ou para D4.

Projeto: É o direito que um usuário tem de *emprestar* objetos dos bancos de dados públicos dos quais ele é membro direto através das operações de *check-out* e *check-in*. No exemplo, o usuário U1 tem direito de projeto para o banco de dados D2, enquanto que um usuário que tenha aberto o banco de dados D1 com a identificação do grupo G2 tem direito de projeto para o banco de dados D1.

Não existe nenhuma facilidade para alteração destes direitos, e portanto só é possível utilizá-lo quando ele se enquadra perfeitamente no controle de acesso da aplicação.

No **MVC**, o controle de acesso é determinado pelas propriedades de cada partição, o que impossibilita o uso do controle feito pelo **Damokles**. Na implementação todos os bancos de dados criados pertencem a um mesmo usuário, o ambiente. Assim, abrindo-se os bancos de dados com a identificação do ambiente, tem-se todos os direitos de acesso garantidos pelo **Damokles**, e o verdadeiro controle é feito pelas funções de verificação de permissão geradas.

Com isto, perde-se a possibilidade de usar o mecanismo de *check-out* e *check-in* oferecido pelo **Damokles**, pois um usuário não pode efetuar um *check-out* de um objeto entre dois bancos de dados seus. De qualquer forma, este mecanismo não poderia ser utilizado por outras características que serão detalhadas adiante.

4.3.2 Identificação

A implementação das operações utilizou a interface funcional da linguagem de manipulação de dados (DML) do **Damokles** embutida na linguagem de programação C. A identificação dos objetos mantidos nos bancos de dados é feita por uma chave única chamada *surrogate*. Quando um banco de dados é criado, o usuário lhe atribui um nome através do qual ele será aberto futuramente. Na abertura de cada banco de dados o usuário recebe um descritor, que será usado para sua identificação durante toda a execução do programa. Para localizar um objeto específico é necessário fornecer sua chave e o descritor do banco de dados em que ele está armazenado. A busca pode ainda ser feita seqüencialmente em todos objetos de um tipo armazenados em um banco de dados. O relacionamento também pode dirigir uma busca, mas o usuário deve especificar tanto o banco de dados em que está o relacionamento, como os bancos de dados em que estão os objetos relacionados.

A implementação utilizou uma hierarquia de bancos de dados com três níveis, iguais aos definidos no modelo. Cada banco de dados criado, é identificado por um número gerado pelo ambiente, e todas as funções implementadas se baseiam neste número, pois os descritores só identificam o banco de dados durante um processo.

O **MVC** identifica um objeto de desenvolvimento pelo nome e representação, uma versão pelo nome e representação do objeto e por seu número, e uma configuração pelo nome e representação do objeto, pelo número da versão e por seu número. No entanto, por uma questão de eficiência, a implementação oferece uma chave única para cada elemento do modelo. A chave da implementação é composta pela própria chave usada pelo **Damokles** (*surrogate*) mais o número do banco de dados em que ela se encontra. Desta forma, quando o usuário fornece a chave de um elemento, é possível localizá-lo com um único acesso ao banco de dados.

Além disso, no **MVC** cada usuário só tem acesso aos bancos de dados superiores ao seu na hierarquia. Portanto, quando um usuário entra no ambiente, é construída uma lista com os descritores e números dos bancos de dados aos quais ele tem acesso. E todas as operações do modelo que envolvem buscas, só procuram nos bancos de dados desta lista, que nunca é composta por mais de três bancos de dados.

4.3.3 Distribuição dos Dados no Damokles

A idéia de banco de dados orientado a objetos surgiu da necessidade de uma forma de armazenamento para objetos de estrutura complexa mais natural do que a permitida pelos modelos existentes. Assim, um banco de dados orientado a objetos deveria ser baseado em um modelo que permitisse o mapeamento direto dos objetos reais para as estruturas de armazenamento.

A linguagem de definição de dados (DDL) do **Damokles** permite que além dos objetos, seus atributos e relacionamentos, sejam declaradas as composições de cada objeto através da cláusula STRUCTURE IS. Ou seja, um objeto pode ser composto por outros objetos ou relacionamentos. Ela permite também que um objeto possua várias versões. As versões por sua vez, são tratadas como objetos, podendo ter atributos, relacionamentos, composição e até versões próprias. Outro conceito suportado é o de união de vários

tipos de objetos em um só tipo, que pode participar de relacionamentos como se fosse um tipo único.

O **Damokles** oferece operações para manipulação de objetos entre bancos de dados, para permitir que um grupo de usuários troque objetos entre si. São operadores para copiar, transferir e para fazer o *check-out* e *check-in* de objetos. Elas afetam os objetos envolvidos de maneira complexa, ou seja, juntamente com suas versões e subobjetos, por isso são chamadas de operações complexas.

Para definir precisamente a semântica destas operações, é introduzido o conceito de *extent* de um objeto. O *extent* ext(o) de um objeto ou uma versão "o" é definida recursivamente, como o conjunto:

- {o} se "o" não tem componentes ou versões,
- {o} U ext{c1} ... U ext{cm} U ext{v1}....U ext{vn} se "o" tem os componentes c1, ... cm e as versões v1, ... vn.

Expressando em palavras, pode-se dizer que o *extent* de um objeto ou versão "o" inclui todos os objetos, relacionamentos e versões que podem ser alcançados a partir do objeto âncora "o", através de uma seqüência arbitrária de relacionamentos de composição e de versão.

As operações complexas efetuadas sobre um objeto "o" operam também sobre os elementos do ext(o). Além disso, todos os objetos originais das versões pertencendo ao *extent* do objeto "o" são copiadas no banco de dados onde serão armazenadas as versões, idenpendentemente de pertencerem ou não ao *extent* do objeto.

Porém, existem algumas restrições quanto à distribuição dos dados que, embora possam ser razoáveis para outras aplicações, para a implementação do **MVC** foram as principais fontes de problemas. Elas restringem as possibilidades de utilização das operações complexas da seguinte forma:

- Os objetos componentes de um objeto estruturado devem estar todos no mesmo banco de dados em que está o objeto composto. Ou seja, não é possível compor um objeto com outro que esteja armazenado em um banco de dados diferente do seu.
- As versões de um objeto devem estar todas no mesmo banco de dados que o original.
- Objetos armazenados em bancos de dados diferentes podem estar relacionados, porém a instância do relacionamento é armazenada somente em um banco de dados.

Estas restrições devem ser levadas em conta na definição dos dados. Muitas vezes seria interessante declarar um relacionamento de composição ou versão para aproveitar as características das operações complexas. Por exemplo, ao fazer uma cópia de um único objeto, todos os seus componentes deveriam ser copiados automaticamente para o mesmo banco de dados, sem ter que procurar cada um deles para realizar a cópia. Mas nem sempre é possível garantir que o objeto componente ou a versão esteja no mesmo banco de dados que o objeto estruturado ou original, respectivamente.

4.3.4 Definição do esquema

Influência da Distribuição

Se não existissem as restrições do **Damokles** discutidas anteriormente, o mapeamento do modelo para um esquema de banco de dados seria praticamente direto, apenas com algumas adaptações. No entanto, as restrições quanto à distribuição dos dados devem ser levadas em conta no momento da definição do esquema, o que adiciona outras dificuldades ao mapeamento.

Uma das restrições do **Damokles** implica que uma busca baseada em um relacionamento que envolve objetos de bancos de dados diferentes deve percorrer todos os bancos de dados possíveis, até encontrar o objeto ou ter certeza que ele não existe. Isto complica a busca em um caso como o do objeto A que está relacionado ao objeto B, através do relacionamento R, sendo que R está no mesmo banco de dados que A, e B está em um banco de dados diferente. Se for necessário partir de A e encontrar B, a busca consiste em localizar o relacionamento R, e depois percorrer todos os bancos de dados possíveis para encontrar B. Se a intenção é partir de B e encontrar A, deve-se percorrer todos os bancos de dados até encontrar o relacionamento R, e depois localizar A a partir de R no mesmo banco de dados.

Isto não foi um grande transtorno para a implementação das operações, pois ela utiliza a lista de bancos de dados a que o usuário tem acesso. E quando a busca é baseada em um relacionamento, só os bancos de dados da lista são percorridos.

No entanto, na definição do esquema, quando era declarado um relacionamento entre objetos que poderiam ser armazenados em bancos de dados diferentes, foi necessário determinar em qual dos bancos de dados o relacionamento seria armazenado. E, sempre que possível, o relacionamento foi declarado componente do objeto pertencente ao relacionamento, armazenado no mesmo banco de dados que ele. O intuito foi de facilitar a transferência, cópia ou remoção do relacionamento quando uma destas operações fosse efetuada em um dos objetos relacionados. Assim, quando foi possível optar, o relacionamento foi declarado subobjeto do objeto mais sujeito a estas operações.

A exigência de ter todos os objetos componentes armazenados no mesmo banco de dados que o objeto estruturado faz com que algumas vezes os dados sejam declarados de uma forma que prejudica a utilização das operações complexas. Um exemplo disto são as referências que uma versão faz a outros objetos de desenvolvimento. Seria conveniente usar a composição para que, no caso de uma transferência ou cópia, as referências continuassem a existir na versão transferida ou copiada. Mas como a versão pode estar em um banco de dados diferente dos objetos de desenvolvimento referenciados, houve a necessidade de usar um relacionamento na definição do esquema. O relacionamento foi declarado como componente da versão, pois ele deve estar sempre no mesmo banco de dados que a versão.

A decisão de declarar um relacionamento como componente de um objeto envolve outros fatores, além de considerar se eles estarão sempre no mesmo banco de dados, e se o relacionamento deve continuar a existir num caso de cópia ou de transferência. Este procedimento pode levar a uma distribuição maior dos relacionamentos. Usando o mesmo exemplo, se o relacionamento *Referência* fosse mantido no mesmo banco de dados que o

objeto de desenvolvimento referenciado, quando fosse necessário saber *se existem* versões que referenciam um determinado objeto, bastaria uma pesquisa no banco de dados em que o objeto de desenvolvimento está armazenado. Caso contrário, a pesquisa teria que percorrer todos os bancos de dados que estejam na hierarquia abaixo do que armazena o objeto de desenvolvimento, pois as versões que o referenciam podem estar espalhadas por eles. Observe que, se fosse necessário saber *quais* versões referenciam determinado objeto, a pesquisa teria que percorrer todos os bancos de dados necessários independentemente do local de armazenamento do relacionamento. Neste caso, os objetos de desenvolvimento referenciados teriam que ser localizados. Presumindo-se que uma pesquisa do primeiro tipo seja, em geral, feita em operações de remoção, optou-se por favorecer outros tipos de operações em detrimento da remoção, sempre que esta escolha teve que ser feita.

Seria interessante aproveitar o mecanismo de versões do **Damokles** pois ele oferece facilidades para criação, remoção e navegação na estrutura de versões. No entanto, no **MVC**, as versões de um objeto de desenvolvimento podem estar espalhadas pelos bancos de dados dos níveis inferiores da hierarquia em relação ao banco de dados que armazena o objeto, o que não é permitido pelo **Damokles**.

A solução encontrada foi declarar um tipo objeto para armazenar as informações relativas aos objetos de desenvolvimento, chamado *Obj_desenvolvimento*, e outros dois tipos objetos, um para versão e outro para configuração.

As versões de um tipo objeto mantidas pelo **Damokles** são identificadas pelo nome do tipo seguido do indicador de versão *.VERSION*. Assim, cada objeto *Obj_desenvolvimento* tem uma estrutura de versões composta por vários Obj_desenvolvimento.VERSION. Mas, na verdade, cada versão da estrutura *Obj_desenvolvimento.VERSION* só armazena um apontador para um objeto *Versão* correspondente. Este objeto *Versão* guarda as informações relativas à versão e pode estar armazenado em outro banco de dados. Um relacionamento chamado *Apontador* faz a associação entre o *Obj_desenvolvimento.VERSION* e o objeto *Versão*.

Embora houvesse uma menor distribuição dos dados, se o *Apontador* fosse armazenado no banco de dados em que está o *Obj_desenvolvimento.VERSION*, ele é armazenado no banco de dados em que está o objeto *Versão*, como um subobjeto deste. Pois só os usuários que têm acesso a *Versão* precisam ter acesso ao *Apontador*. Assim, se durante uma busca, o relacionamento *Apontador* correspondente a uma *Versão* não for encontrado, já se sabe que ela está armazenada em um banco de dados em que o usuário não tem acesso.

O mesmo foi feito com as configurações. Cada objeto *Versão* tem uma estrutura de versões com elementos *Versão.VERSION*, sendo que cada um destes elementos aponta para um objeto *Configuração*.

Atributos

Em geral, os atributos dos elementos do modelo podem ser mapeados diretamente para atributos de *Obj_desenvolvimento*, *Versão* ou *Configuração* do esquema.

Alguns atributos do modelo identificam outros objetos de desenvolvimento, versões, configurações, usuário ou grupos de usuários. Neste caso os atributos são mapeados

para relacionamentos, entre o elemento que tem o atributo e o elemento identificado pelo atributo. Por exemplo, a configuração ativa de uma versão é mapeada para um relacionamento entre os objetos *Versão* e *Configuração*.

Os atributos definidos para cada ambiente, na linguagem de definição também são facilmente representados no esquema. Os atributos definidos para objetos, versões e configurações de qualquer representação são mapeados diretamente para atributos no esquema gerado, com exceção dos atributos definidos com os domínios ID_OBJETO, ID_VERSAO, ID_CONFIGURACAO, ID_USUARIO e ID_GRUPO. Estes atributos são mapeados para relacionamentos, seguindo o mesmo raciocínio utilizado para os atributos suportados diretamente pelo modelo. Assim, se em um ambiente específico for necessário manter um atributo *responsável* para identificar o usuário que tem a responsabilidade de desenvolver as versões de um objeto, é declarado um novo relacionamento entre *Obj_desenvolvimento* e *Usuário*.

Os atributos específicos de cada representação são tratados de forma diferente. O objeto *Obj_desenvolvimento* tem como componente o objeto *At_o_representação*. Este objeto por sua vez, é uma união de vários objetos, sendo que cada um deles armazena os atributos específicos de uma dada representação. Os atributos de versões e configurações específicos de cada representação foram declarados no esquema da mesma forma.

4.4 Operações

Como a implementação de todas as funções definidas para o gerenciador seria uma tarefa muito extensa, optou-se por implementar as funções necessárias para realizar uma resolução de configuração. Para tanto, foram implementadas, além da função de resolução, a criação e a atualização de objetos, versões e configurações, incluindo suas referências e entradas. Foi necessária também a implementação da definição das consultas através de cláusulas e predicados, bem como das operações que efetuam as consultas definidas. Estas funções envolvem as características centrais do modelo, e sua implementação permite a sua verificação quanto a coerência e adequação.

4.4.1 *Check-out/Check-in*

A utilização do mecanismo de *check-out/check-in* do **Damokles** foi inviabilizada logo de início. O MVC permite que um usuário de um banco de dados particular faça um *check-out* de objetos do banco de dados geral, e o **Damokles** só permite que um usuário faça *check-out* de objetos que estejam em bancos de dados de grupos dos quais ele seja membro direto. Para tentar aproveitar este mecanismo tão poderoso, cogitou-se na utilização de dois níveis de bancos de dados. Assim, no nível superior, ficariam os dados de uso geral e os pertencentes aos grupos, e a propriedade deles seria representada apenas por um atributo, e não pela sua localização física em um determinado banco de dados. Mas o **Damokles** também não permite que se faça um *check-out* de um objeto de um banco de dados para ele próprio, como seria o caso de um representante de grupo que fizesse um *check-out* de um objeto de uso geral. Portanto, a implementação com dois níveis foi descartada, e com ela a possiblidade de usar o mecanismo do **Damokles**.

Quando se faz um *check-out* em um objeto, deve ser copiado no novo banco de dados, juntamente com o objeto, todos os relacionamentos e subobjetos. Além disso, deve-se controlar o uso dos originais no banco de dados antigo. Na realização do *check-in* todas as atualizações feitas na cópia utilizada para o *check-out* teriam que ser refeitas no original.

Como o objetivo principal da implementação era a operação de resolução de configuração, o *check-out* foi implementado somente de forma a permitir um controle do uso dos objetos por mais de um usuário. Assim, a operação implementada só coloca uma proteção no objeto utilizado, impedindo que outros utilizem de forma inadequada o mesmo objeto. O usuário que efetuou o *check-out* recebe uma chave correspondente à cópia. Esta chave é utilizada pelo usuário para efetuar todas as operações de leitura, atualização e derivação. Mas na verdade, as operações são feitas diretamente nos originais. Conseqüentemente o *check-in* consiste apenas na liberação das proteções dos originais.

O mecanismo que controla os relacionamentos de *Ativação* de uma versão ou configuração com o objeto de desenvolvimento, ou entre uma configuração e uma versão, foi implementado conjuntamente com as operações de *check-out* e *check-in*.

Além disso, a alteração do valor do estado da configuração, devido a atualizações nas referências da versão correspondente ou nas entradas da própria configuração, também é feita de forma integrada com a operação de *check-in*.

4.4.2 Consultas

O **Damokles** pode ser classificado como um banco de dados orientado a objetos estrutural, ou seja, permite a descrição de objetos complexos, mas não permite a definição de operações particulares para cada tipo de dados. Esta facilidade é suportada pelos bancos de dados orientados a objetos comportamentais. A falta de possibilidade de definição de novas operações, resultou numa implementação bastante trabalhosa para as operações relacionadas às consultas.

A consulta definida por um usuário deve ser traduzida para o código que realize a consulta no banco de dados por meio de funções do **Damokles**. Assim, o usuário define a consulta através de inserções de cláusulas e predicados no banco de dados. Uma vez feita a definição, a operação *TraCons* deve ser usada para fazer o mapeamento das cláusulas para o código que faz a consulta no banco de dados. O código executável da consulta é inserido em um campo longo, no mesmo objeto que armazena as cláusulas. Sempre que a consulta for alterada, o mapeamento precisa ser refeito.

O código executável é carregado no banco de dados a partir de arquivos com o código fonte para alguns exemplos de consultas. No entanto, a implementação do mecanismo de mapeamento não seria difícil pois já existe um mecanismo do próprio **Damokles** muito parecido. Deveriam ser acrescentadas facilidades para operar na lista de bancos de dados acessíveis ao usuário, pois originalmente ele opera em um único banco de dados.

Depois de definida a consulta o usuário pode inserir valores para os parâmetros, e então ela pode ser efetuada pelo usuário através de uma operação própria. Esta operação traz o código armazenado no banco de dados para o sistema de arquivos para que ele possa ser executado. Posteriormente, ela escreve todos os parâmetros em um arquivo, e dispara um processo para executar o código armazenado no sistema de arquivos. O

código pré-definido lê o arquivo onde estão os parâmetros, e escreve a chave ou conjunto de chaves resultantes em um outro arquivo. Terminando o processo disparado, a operação lê o novo arquivo com a chave ou lista de chaves resultantes e as devolve em um de seus parâmetros.

4.4.3 Derivação

Quando uma versão ou configuração é derivada a partir de outra já existente, ela consiste, inicialmente, em uma cópia de sua antecessora. Na implementação desta função a operação de cópia *db_cpo* oferecida pelo **Damokles** seria particularmente útil.

Se as versões do modelo fossem mapeadas diretamente para versões do **Damokles**, esta operação não poderia ser usada. Pois quando uma versão fosse derivada de outra versão armazenada em um banco de dados diferente, a operação *db_cpo* copiaria também o objeto genérico e todas as suas versões para o novo banco de dados. Mas como as versões do **MVC** foram definidas como um tipo objeto, não haveria problemas quanto à cópia do objeto genérico.

A operação *db_cpo* do **Damokles** se encarrega de criar uma novo relacionamento entre a cópia e os objetos relacionados originais. Portanto, todos os relacionamentos de referência seriam refeitos automaticamente.

No entanto, a estrutura do objeto *Versão* que organiza as configurações daquela versão, também seria copiada, criando uma estrutura de apontadores para configurações inexistentes. Na derivação de versão só uma configuração pode ser copiada. Desta forma, optou-se por implementar a derivação de versões sem utilizar a operação *db_cpo*, copiando os relacionamentos de referência, um a um e fazendo a cópia da configuração. Além disso, na derivação de versão, devem ser criados um novo *Obj_desenvolvimento.VERSION* e um relacionamento entre ele e a *Versão*. O novo *Obj_desenvolvimento.VERSION* é localizado na estrutura como sucessor da versão que lhe deu origem.

Por outro lado, a implementação da operação de derivação de configuração pode ser feita diretamente com a operação *db_cpo*, pois o tipo objeto *Configuração* não tem versões. Desta forma, foi evitado o trabalho de percorrer todas as suas entradas para fazer a cópia. Só foi necessário criar separadamente a *Versão.VERSION* na estrutura e o relacionamento *Apontador*.

4.4.4 Resolução de Configuração

A resolução de configuração consiste em determinar uma versão, ou uma configuração para cada objeto de desenvolvimento referenciado por uma versão. Se a versão escolhida não faz referências a outros objetos não é necessário determinar a configuração, caso contrário deve-se determinar a configuração e também resolvê-la. Como se nota, este é um processo eminentemente recursivo. Para cada configuração envolvida é criado um novo objeto no banco de dados com o nome *Conf_resolvida*, que protege os originais e contém entradas diretas para todas as versões e configurações que compõem o sistema.

No entanto, antes da criação de uma *Conf_resolvida* é necessário dar um *check-out* em todas as versões e configurações usadas, para se garantir que outros usuários

não alterem as referências feitas pelas versões ou as entradas das configurações durante a resolução. Assim, a estratégia utilizada na implementação foi armazenar todas as informações obtidas durante a resolução das entradas em uma estrutura auxiliar, e fazer um *check-out* com proteção de escrita para cada versão ou configuração encontrada. Se foi possível determinar uma versão ou configuração para cada objeto referenciado e dar um *check-out* com a proteção adequada em todas elas, a estrutura auxiliar é percorrida novamente, e o objeto *Conf_resolvida* criado.

A princípio são percorridos os relacionamentos de referência da versão sendo configurada, e a estrutura auxiliar é construída. Ela consiste em uma lista ligada, que tem um registro para cada objeto referenciado. Cada registro contém campos para armazenar as chaves do objeto de desenvolvimento, da versão e da configuração, um campo que informa se já foi determinada uma versão ou configuração para aquele objeto, e um apontador para lista ligada das referências daquela versão, se houver alguma.

Como existe a possibilidade de uma subconfiguração ser usada mais de uma vez na mesma configuração, foi utilizado mais um campo em cada resgistro. Ele aponta para o registro da subconfiguração repetida, se ela existir. Assim, é usada a mesma lista de referência, e evita-se fazer novamente a mesma construção.

Depois de construída a lista de referências de uma versão, as entradas da configuração correspondente são percorridas, uma a uma, até se obter todas as informações desejadas ou terminarem as entradas. Se as informações contidas nas entradas da configuração não forem suficientes, ou não existir a configuração para se resolver as referências desta versão, as entradas da configuração global ativa do banco de dados do usuário são percorridas. As entradas são tratadas da seguinte forma:

Direta total: Representa um relacionamento entre a configuração e uma subconfiguração. Se o objeto de desenvolvimento relativo à subconfiguração fizer parte da lista de referências sendo resolvida, constrói-se uma lista de referências para a versão correspondente. A nova lista é resolvida recursivamente, baseando-se na subconfiguração. A versão e a configuração sofrem um *check-out* com proteção de escrita e têm suas chaves guardadas nos respectivos campos do registro da lista de referências. Se o objeto de desenvolvimento não fizer parte da lista de referências a entrada é ignorada.

Direta parcial: Representa um relacionamento entre a configuração e uma versão. Se o objeto de desenvolvimento relativo à versão fizer parte da lista de referências em resolução, e se a versão fizer referências a outros objetos, é construída uma nova lista com estas referências que será resolvida de acordo com sua configuração ativa. A versão e a configuração sofrem um *check-out* e têm suas chaves armazenadas nos campos adequados do registro da lista de referências. Se a versão não fizer referências a outros objetos, o único procedimento é anotar sua chave no campo adequado do registro que representa o seu objeto e dar um *check-out* na versão. Se o objeto de desenvolvimento não fizer parte da lista de referências em resolução, a entrada é ignorada.

Direta de consulta: Representa um relacionamento entre a configuração e a definição dos parâmetros de uma consulta direta. A consulta é efetuada utilizando-se os

parâmetros definidos. Conforme o tipo do seu resultado for versão ou configuração, resolve-se como uma entrada direta total ou parcial respectivamente.

Indireta de inclusão: Representa um relacionamento entre a configuração e outra configuração de inclusão. A mesma lista de referências continua a ser resolvida, com base nas entradas da configuração de inclusão.

Indireta de consulta: Representa um relacionamento entre a configuração e a definição dos parâmetros de uma consulta indireta. A consulta é efetuada utilizando-se os parâmetros definidos, resultando em uma lista de chaves de versões ou configurações. Se forem versões, cada uma delas é resolvida como uma entrada direta parcial; se forem configurações cada uma delas é resolvida como entrada direta total.

Default **de número:** Representa um número de versão ou configuração. Se um número de versão for especificado, para cada objeto da lista de referências que ainda não foi resolvido, é procurada a versão com aquele número, e se ela existir é resolvida como uma entrada direta parcial. Se um número de configuração for especificado, a configuração de cada objeto é procurada, e se for encontrada é tratada como uma entrada direta total.

Default **de consulta:** Representa um relacionamento entre a configuração e a definição dos parâmetros de uma consulta direta. A consulta é efetuada para cada objeto da lista de referências que ainda não foi resolvido. Se o tipo do resultado da consulta for versão, ela é resolvida como uma entrada parcial. Se o tipo do resultado for uma configuração, ela é resolvida como uma entrada direta total.

A prioridade das versões e configurações ativas é passada como parâmetro para a operação. Antes de começar a resolução de cada tipo de entrada, a prioridade é testada, e dependendo do seu valor as ativações são consideradas naquele momento ou deixadas para depois. Quando as ativações são consideradas, procura-se uma versão ou configuração ativa, para cada elemento da lista de referência que ainda não foi resolvido.

Depois que a estrutura auxiliar estiver completa, é inserido no banco de dados do usuário um objeto do tipo *Conf_resolvida* e a lista de referências é percorrida novamente. As informações contidas em cada registro dirigem o restante do processo.

Se o registro tem uma lista de referências associada a ele um novo objeto *Conf_resolvida* é inserido sendo ligado à *Conf_resolvida* de nível superior através de um relacionamento *Entrada_direta*. E então, sua lista de referências será percorrida pelo mesmo procedimento ligando cada elemento encontrado. O objeto *Conf_resolvida* inserido também é ligado à *Configuração* que lhe deu origem através do relacionamento *Ckrel*. Este relacionamento é do mesmo tipo do usado pela operações de *check-out* e, portanto, resulta na proteção da configuração. Se as referências foram resolvidas através da configuração global ativa, a *Conf_resolvida* é relacionada à *Versão* anotada no registro. Caso o registro esteja marcado como repetição, não é inserido um novo objeto *Conf_resolvida*. A *Conf_resolvida* do objeto, do qual ele é repetição, é inserida como

subobjeto da *Conf_resolvida* de nível superior. Se o elemento não tem uma lista de referências associada, a versão anotada no registro é ligada à *Conf_resolvida* através do relacionamento *Entrada_direta*.

Desta forma o usuário tem no seu banco de dados uma *Conf_resolvida* que protege a configuração ou versão que lhe deu origem e tem como subobjetos, outras *Conf_resolvidas* ou *Versões*. Agora ele pode percorrer esta estrutura para saber a sua composição através dos operadores para leitura de entradas de configuração.

Se o usuário resolver novamente a mesma configuração o objeto *Conf_resolvida* é removido com todos os seu subobjetos, e outro é construído no seu lugar. O objeto *Conf_resolvida* também será removido se o usuário fizer a transformação para uma configuração comum.

Capítulo 5

Conclusão

5.1 O Modelo Ideal

Devido à importância do controle de versões e configurações em ADS, vários modelos para o seu gerenciamento têm sido propostos. A complexidade dos aspectos envolvidos gera muita controvérsia quanto à melhor forma de abordar o problema.

A análise das características de diversos gerenciadores leva à conclusão de que não existe um modelo ideal. Percebe-se claramente que existe um balanço entre as características dos modelos, sendo que cada modelo enfatiza determinados aspectos do problema. Algumas facilidades interessantes oferecidas por um gerenciador geralmente o prejudicam quando ele é analisado em relação a outros aspectos.

5.2 Relação entre o MVC e Outros Gerenciadores

A modelagem apresentada no capitulo 3 teve como objetivo fornecer um mecanismo de controle de versões e configurações que não se prendesse a nenhum ambiente de desenvolvimento ou linguagem de programação específicos. Procurou-se estabelecer uma base adequada para suporte de versões e configurações em ambientes distribuídos. Algumas diferenças entre o MVC e outros modelos podem ser ressaltadas.

As versões no MVC estão armazenadas em uma hierarquia de bancos de dados como no modelo descrito em [Chou86]. Este tipo de organização oferece possibilidades para uma melhor distribuição dos dados no ambiente, e alguns problemas encontrados no seu processo de configuração foram resolvidos pelo MVC. No modelo de Chou não existem versões de configuração. Cada versão faz referências diretamente às outras versões. Assim, a transferência de uma versão de um banco de dados para outro, requer que todas versões que ela referencia também sejam transferidas. Além disso, cada banco de dados tem uma estrutura de derivação e uma numeração para as versões de um mesmo objeto. Quando uma versão é transferida, ela recebe um novo número e o usuário é obrigado a converter todas as referências feitas por outros objetos à versão transferida. Ele deve

também indicar uma nova posição para a versão na estrutura de derivação do objeto no novo banco de dados. Outro inconveniente encontrado devido à referência ser feita diretamente de uma versão a outra é a impossibilidade de construir uma configuração em que uma versão no banco de dados geral ou no de projeto faça referências a versões em bancos de dados particulares.

Uma parte dos problemas foram evitados no **MVC** através da adoção de estrutura de derivação e numeração únicas para todas as versões de um objeto em qualquer banco de dados. Desta forma, o usuário não é obrigado a converter referências e especificar novas posições na transferência de uma versão. Além disso, a estrutura usada realmente espelha o processo de derivação que, em geral, é uma informação muito útil em ADS. Por outro lado, a existência de várias configurações para cada versão, possivelmente em bancos de dados diferentes, permite construir uma configuração na qual uma versão de um banco de dados de projeto ou geral faça referências a versões particulares.

A seleção dos elementos do banco de dados pode ser feita por números ou por rótulos que identificam consultas pré-definidas. As consultas são definidas através de predicados baseados nos valores dos atributos. Embora a sua sintaxe seja muito parecida com as regras de seleção do **shape** [Mahl88], ela se destina a recuperar um objeto ou um conjunto deles com determinadas características, enquanto que no **shape** as regras são usadas como uma descrição de várias versões com características distintas, que formam uma configuração. No **MVC** isto não é necessário, pois versões diferentes são selecionadas por entradas de configuração diferentes. Ao contrário do **shape**, do **Adele** [Belk87] e do **Gypsy** [Cohe88], a definição dos predicados é feita uma única vez e pode ser usada sempre que necessário.

O **MVC** se enquadra entre os gerenciadores que controlam o acesso através dos domínios de cada usuário. Na definição do ambiente são declaradas as partições de cada banco de dados com as respectivas permissões. Embora este não seja um mecanismo tão flexível quanto o encontrado no **Adele**, ele permite um maior controle sobre o estado de uma versão. No **Adele** um usuário pode ter o direito de dar permissões a outros usuários sobre determinados objetos. Quando dois usuários têm os mesmos direitos sobre uma versão, um deles pode restringir o acesso a esta versão, não concedendo permissões a ninguém, mas ele não pode garantir que o outro faça o mesmo. No **MVC**, quando um usuário coloca uma versão em uma partição ele sabe exatamente as operações que cada usuário pode realizar com ela. E ainda, como cada ambiente pode definir suas partições, elas podem refletir o ciclo de vida dos objetos de software no ambiente, permitindo assim que a política de desenvolvimento desejada seja implantada.

A existência de várias configurações para uma mesma versão de objeto de desenvolvimento, permite que o problema de contexto estável seja resolvido facilmente no **MVC**. Quando um usuário não quer que a versão de um objeto que ele está usando mude, ele pode criar uma versão de configuração que referencie diretamente esta versão do objeto de desenvolvimento. E mais que isso, se ele quiser manter estável não só a versão de um componente, mas todas as versões referenciadas por este componente, basta resolver a configuração do componente e usar uma referência para a configuração resolvida na configuração do nível superior.

5.3 A Integração das diversas características

O MVC procurou oferecer mecanismos para que cada ambiente possa adotar as suas políticas particulares, e para permitir uma melhor distribuição dos dados no ambiente. Assim, pode-se afirmar que as maiores contribuições do trabalho foram: a captação das características de cada ambiente através dos parâmetros, em especial a definição do controle de acesso através de partições, e a introdução de diversas configurações para cada versão. No entanto, a generalidade e a distribuição causaram a maior parte dos problemas encontrados na integracão dos mecanismos.

Como o modelo não tem o controle de acesso fixo, muita responsabilidade é passada ao usuário, quanto à consistência das operações. Uma vez que o usuário tem a possibilidade de alterar configurações já resolvidas, não há como garantir que todas as configurações continuem consistentes após as alteracões. Uma configuração pode fazer referências às configurações e às versões armazenadas em bancos de dados de níveis superiores, que são compartilhadas com outros usuários, e portanto podem ser alteradas por eles. Para contornar este problema, existe uma operação através da qual o usuário pode verificar o estado da configuração. Além disso, cada ambiente pode manter uma partição, na qual não é dada permissão de atualização a ninguém. E então todas as configurações resolvidas que precisarem ser congeladas, podem ser colocadas nesta partição.

Outra conseqüência da distribuição das versões e configurações em bancos de dados diferentes do que está armazenado o objeto, é a impossibilidade de ter um mecanismo de *check-out/check-in* integrado. Em outras palavras, é impossível oferecer um mecanismo em que o usuário ao fazer um *check-out* em um objeto, tenha em seu banco de dados todas as suas versões e configurações. Isto não é possível pois o usuário pode ter acesso ao objeto, mas não ter a algumas de suas versões e configurações. Tampouco se pode garantir a um usuário que esteja com uma configuração no seu banco de dados, que o objeto e a versão estejam no mesmo banco de dados e com as mesmas proteções. Como eles podem estar armazenados em partições diferentes, o usuário pode ter permissão para atualizar uma configuração mas não uma versão. Assim, os modos das operacões de *check-out* teriam que ser diferentes, o que poderia confundir os usuários.

Além disso, o modelo permite que as versões compartilhadas sejam modificadas diretamente, sem a criação de novas versões, desde que o usuário faça um *check-out* na versão para que outros não possam modificá-la ao mesmo tempo. Desta forma, a criação de uma nova versão deve ser pedida explicitamente pelo usuário; ela não ocorre através de atualizações, ou seja, o mecanismo de *check-out/check-in* não é o mesmo de derivação de versões. O controle de acesso foi definido de forma que um usuário possa ter direito de atualizar versões, mas não de derivar versões, e vice-versa. Isso permite que cada ambiente defina o controle de acesso de maneira a refletir mais de perto a sua política. Embora seja mais prático e mais fácil criar uma versão toda vez que ocorre atualização, isso leva a uma proliferação muito grande de versões.

No MVC, como em muitos outros modelos que suportam a definição de atributos, as alternativas são diferenciadas somente por seus atributos. O modelo não faz um controle direto sobre a criação de ramificação, ele é feito através das partições definidas para cada banco de dados. O usuário nem sempre está ciente de que está criando uma

ramificação. Assim, não teria sentido um mecanismo de seleção de alternativas que se baseasse nos números das ramificações. No entanto, os nomes simbólicos para as derivações, suportados pelo MVC, constituem uma outra forma de identificação das ramificações.

Finalizando, pode-se dizer que ao mesmo tempo que o modelo oferece uma maior liberdade ao usuário, ele lhe delega maiores responsabilidades. Mas, como a intenção do MVC é servir como base para o suporte de versões e configurações, naturalmente suas primitivas não serão manipuladas diretamente por cada usuário do ambiente. O uso do controlador vai, com certeza, ser adaptado às restrições de cada ambiente. E para o usuário será apresentada uma outra camada, de forma que ele tenha o mínimo de trabalho possível.

5.4 Implementação

A implementação teve como objetivo a comprovação da viabilidade do modelo, mas outros resultados foram também obtidos.

Notou-se que a decisão pela implementação de um gerador de versões e configurações foi acertada, pois ela oferece um maior desempenho em relação a uma implementação baseada em uma meta base de dados. Segundo Leblang [Lebl84], os usuários precisam do gerenciamento de versões e configurações para assegurar a consistência da composição do seu sistema, e já esperam pagar pela segurança adicionada sacrificando o desempenho. No entanto, eles exigem que o desempenho do gerenciador seja aceitável. Alguma melhora na implementação poderia ser conseguida, com tabelas de *hash* ou com um *buffer* contendo as identificações dos elementos usados mais recentemente. Através de uma destas técnicas o usuário poderia ser poupado do uso das chaves, sem prejuízo do desempenho. Ele manipularia os objetos, versões e configurações diretamente pelo identificador definido no modelo, facilitando muito o seu trabalho.

Foi possível também avaliar a adequação de um SGBD orientado a objetos na implementação de um modelo de controle de versões e configurações. A mais grave das limitações do SGBD **Damokles** é que os esquemas dos bancos de dados só podem ser definidos estaticamente. Não é permitido fazer alterações no esquema original em tempo de execução. Assim, é necessário definir um mecanismo que permita baixar os dados contidos no banco de dados para o sistema de arquivos, alterar o esquema e carregar novamente os dados. A incapacidade de criação dinâmica de atributos pode prejudicar a identificação de alternativas, quando a propriedade que as distingüe não foi prevista na definição do modelo. No entanto, o mecanismo de criação de nomes simbólicos para as alternativas, contorna de certa forma este problema.

Ainda assim, o **Damokles**, bem como as outras ferramentas utilizadas: o **Lex** e o **yacc**, foram de grande ajuda na implementação. A geração dos analisadores léxico e sintático, foi muita facilitada pelo **Lex** e pelo **yacc**, respectivamente. O suporte a versões e objetos complexos, oferecido pelo **Damokles**, ainda que limitado, foi essencial para a implementação. É até difícil imaginar qual seria a sua complexidade caso fosse usado um banco de dados relacional.

5.5 Trabalhos Futuros

A experiência obtida na definição do modelo foi muito importante para estabelecer as dificuldades e necessidades envolvidas na manipulação de versões e configurações. No entanto, muitas outras características devem ainda ser incorporadas ao modelo definido.

Uma das mais importantes entre elas, é a manutenção de equivalência entre as versões em diferentes representações, através do suporte à derivação automática. Seria interessante um modelo que unisse as facilidades oferecidas pelos modelos de configuração baseados em dependências fonte, com as oferecidas pelos modelos baseados em dependências de processamento. O modelo deveria suportar a distribuição dos dados, tanto a nível de versões como de configurações, ao mesmo tempo que oferecesse facilidades para a derivação automática do sistema. Talvez a chave para encontrar este modelo esteja no uso de nomes lógicos não só para a definição dos componentes de um sistema, mas também para a descrição das etapas de derivação entre eles.

Com a manutenção de equivalências seria necessária uma exploração maior das interferências da alteração de uma configuração. Pois, a simples manutenção de um atributo com o estado da configuracão não seria mais suficiente. A alteração de uma configuração em uma representação poderia implicar na necessidade da reconstrução de outras configurações em outras representações.

Uma outra extensão interessante para o modelo é a manutenção da compatibilidade sintática e semântica das dependências entre as versões. Como o MVC tem a intenção de suportar diversas linguagens, seria necessário desenvolver um protocolo de especificação de interface multi-linguagem, para que isso fosse possível.

Espera-se que os argumentos apresentados sejam ao menos suficientes para o convencimento de que a manutenção de versões e configurações é um assunto complexo e digno de estudos futuros, e que a definição de um modelo que suporte adequadamente as características acima é essencial para as pesquisas em ADS.

Bibliografia

[Abram88] Abramowicz, K et alli, DAMOKLES Database Management System for Design Applications, Reference Manual Release 2.0, Forschungszentrum Informatik an der Universit'at Karlsruhe, March 1988

[Bato85] Batory, D. S. & Kim, W., Modeling Concepts for VLSI CAD Objects, ACM Transactions on Database Systems, vol 10, n.3, September 1985, p. 322 - 346

[Belk87] Belkhatir, N. & Estublier, J., Exprience with Data Base of Programs, Proceedings of the ACM SIGSOFT / SIGPLAN Software Engineering Symposium on Pratical Software Development Environments, Palo Alto, California, December 1986, p. 84 - 91

[Bona81] Bonanni, L. E. & Salemi, C. A., Source Code Control System User's Guide, Bell Telephone Laboratories, January, 1981

[Chou86] Chou, H. T. & Kim, W., A Unifying Framework for Version Control in a CAD Environment, Proceedings of the 12th VLDB Conference, Kyoto, August 1986, p. 336 - 344

[Cohe88] Cohen, E. S., et. al., Version Management in Gypsy, ACM, 1988, p. 201 - 215

[Dart87] Dart, S. A., et. al., Software Development Environments, Computer, November 1987, p. 18 - 28

[Ditt88] Dittrich, K. R. & Lorie, R. A., Version Support for Engineering Database Systems, IEEE Transactions on Software Engineering, vol. 14, n. 4, April 1988, p. 429 - 436

[Drum87a] Drummond, R. & Liesenberg, H., A_HAND Ambiente de Desenvolvimento de Software Baseado em Hierarquias de Abstração em Níveis Diferenciados, Anais do IV Encontro de Trabalhos do Projeto Ethos, Petrópolis, RJ, Abril 1987, p. 313 - 22

[Drum87b] Drummond, R. & Liesenberg, H., Requisitos para Ambiente de Desenvolvimento de Programas, I Encontro IBM de Ciência e Tecnologia em Informática, Rio de Janeiro 1987

[Estu84] Estublier, J., Ghoul, S. & Krakowiak, S., Preliminary Esperience with a Configuration Control System, ACM Proceedings of SIGSOFT / SIGPLAN Software Engineering Symp. on Pratical Development Environments, May 1984, p. 149 - 156

[Feld79] Feldman, S. I., Make - A Program for Maintaining Computer Programs, Software - Pratice and Experience, vol. 9, n. 4, April 1979, p. 255 - 265

[Giff88] Gifford, D. K., Needham, R. M. & Schroeder, M. D., The Cedar File System, Communications of the ACM, vol. 31, n. 3, March 1988, p. 288 - 298

[Hara88] Hara, C. S. & Magalhães, G. C., Banco de Dados para Ambientes de Desenvolvimento de Software, Proposta de Tese de Mestrado, DCC, Unicamp, Setembro 1988

[Hara90] Hara, C. S. & Magalhães, G. C., Utilização de um Banco de Dados Orientado a Objetos em um Ambiente de Desenvolvimento de Software, Dissertação de Mestrado, DCC Unicamp, Campinas, Outubro 1990

[Heim88] Heimbigner, D. & Krane, S., A Graph Transform Model for Configuration Management Environments, ACM, 1988, p. 216 - 225

[Hoff85] Hoffnagle, G. F. & Beregi, W. E., Automating the Software Development Process, IBM Systems Journal, vol. 24, n. 2, 1985, p. 102 - 120

[Katz84] Katz, R. H. & Lehman, T. J., Database Support for Versions and Alternatives of Large Design Files, IEEE Transactions on Software Engineering, vol SE-10, n. 2, March 1984, p. 191 - 200

[Katz86] Katz, R. H., Chang, E. & Bhateja, R., Version Modeling Concepts for Computer-aided Design Databases, ACM Proceedings of SIGMOD 86 - International Conference on Management of Data, Washington, D.C., May 1986, p. 379 - 386

[Katz87a] Katz, R. H., et. al., Design Version Management, IEEE Design & Test, February 1987, p. 13 - 22

[Katz87b] Katz, R. H. & Chang, E., Managing change in a Computer-Aided Design Database, Proceedings of the 13th VLDB Conference, Brighton, 1987, p. 455 - 562

[Klah86] Klahold, P., Schlageter, G. & Wilkes, W., A General Model for Version Management in Databases, Proceedings of the Twelfth International Conference on Very Large Data Bases, Kyoto, August 1986, p. 319 - 327

[Lebl84] Leblang, D. B. & Chase, R. P., Computer- Aided Software Engineering in a Distributed Workstation Environment, Proceedings of the ACM SIGSOFT / SIGPLAN Software Engineering Symposium on Pratical Software Development Environments, Pittsburgh, Pa, April 1984, p. 104 - 112

[Lori77] Lorie, R. A., Physical Integrity in a Large Segmented Database, ACM Transactions on Database Systems, vol. 2, n. 1, March 1977, p. 91 - 104

[Mahl88] Mahler, A. & Lampen, A., An Integrated Toolset for Engineering Software Configurations, ACM, 1988, p. 191 - 200

[Marz87] Marzullo, K. & Wiebe, D., Jasmine - A Software System Modelling Facility, Proceedings of the ACM SIGSOFT / SIGPLAN Software Engineering Symposium on Pratical Software Development Environments, p. 121 - 130

[Perr87a] Perry, D. E., Software Interconnection Models, Proceedings of 9th International Conference on Software Engineering, Monterey, California, April 1987, p. 61 - 69

[Perr87b] Perry, D. E., Version Control in the Inscape Environment, Proceedings of 9th International Conference on Software Engineering, Monterey, California, April 1987, p. 142 - 149

[Rama86] Ramamoorthy, C. V., Garg, V. & Prakash, A., Programming in the Large, IEEE Transactions on Software Engineering, vol. SE-12, n. 7, July 1986, p. 769 - 783

[Reed87] Reed, K., Pratical Software Engineering Environments: Report on the ACM SIGSOFT/SIGPLAN Software ENgineering Symposium, Software Engineering Notes, vol. 12, n. 1, January 1987, p. 56 - 61

[Schr] Schreiner, A. T., & Friedman, Jr., Introduction to Compiler Construction with UNIX, Prentice-Hall, Inc.

[Schw89] Schwanke, R. W., et. al., Configuration Management in BiiN SMS, ACM 1989, p. 383 - 393

[Silv88] Silva, F. Q. B., Liesenberg, H. & Drummond, R., Programação em Cm, Relatório de acompanhamento de bolsa para a FAPESP, Março 1988

[Thom88] Thomas, D. & Johnson, K., Orwell - A Configuration Management System for Team Programming, OOPSLA 88 Proceedings, September 1988, p. 135 - 141

[Tich85] Tichy, W. F., RCS - A System for Version Control, Software - Pratice and Experience, vol. 15(7), July 1985, p. 637 - 654

[Vict89] Victorelli, E. Z., Magalhães, G. C. & Drummond, R., Mecanismo de Gerenciamento de Versões e Configurações do A_HAND, Anais do III Simpósio Brasileiro de Engenharia de Software, SBC, Recife, PE, Outubro 1989, p. 269 - 280, também publicado na Revista Brasileira de Computação, vol 5, n. 2, dezembro 1989, p. 3 - 9

[Wink87] Winkler, J. F. H., Version Control in Families of Large Programs, Proceedings of 9th International Conference on Software Engineering, Monterey, California, April 1987, p. 150 - 161

Apêndice A

Usando o MVC

A.1 Introdução

O **A_HAND** é um ADS que está sendo desenvolvido no Departamento de Ciência da Computação da Unicamp. O seu mecanismo de gerenciamento de versões e configurações é uma extensão ao modelo proposto em [Ditt88] e se encontra descrito em detalhes em [Vict89].

O modelo proposto no Capítulo 3 estabelece um base para a implantação de controladores de versões e configurações. O mapeamento de algumas características do mecanismo existente no **A_HAND** para o **MVC**, ilustra sua utilidade.

A.2 Características do Ambiente A_HAND

A linguagem Cm - C modular e polimórfico - foi projetada para ser a linguagem básica do ambiente **A_HAND**. Em Cm [Silv88] os elementos básicos manipulados são classes. As classes contém todas as definições de estruturas e funções necessárias para a sua utilização, constituindo-se em unidades de software que podem ser desenvolvidas separadamente e depois integradas para formar um sistema mais complexo.

Uma classe é potencialmente um programa e pode importar um conjunto de outras classes. As classes importadas são chamadas de subprogramas ou subsistemas e, por sua vez, podem importar outras classes, até se chegar a classes básicas que não fazem importações [Silv88]. A última classe a fazer importações é a raiz do grafo que representa o sistema.

O conceito de herança múltipla de tipos também é suportado pela linguagem. Uma classe pode herdar várias características de outras classes, e ainda passar suas características juntamente com as que foram herdadas para seus próprios herdeiros. Como em Cm, as classes definem construtores de tipos, as relações de herança dão origem a uma hierarquia de tipos.

Assim, um sistema de software em Cm consiste de um grafo acíclico, definido pelas relações de importação e herança.

A.3 Controle de Versões

A.3.1 Tipos de versões

No **A_HAND**, os usuários do ambiente são divididos em grupos, resultando na existência de três tipos de versões: liberadas, efetivas e edições. Uma versão pode ter diferentes capacidades conforme o seu tipo. As edições só podem ser usadas por quem as criou, isto é, seu dono. As versões efetivas podem ser usadas pelos integrantes do grupo a que o seu criador pertence. Finalmente, as versões liberadas podem ser referenciadas por todos os usuários do sistema.

Tanto as versões liberadas como as efetivas estão congeladas, isto é, não podem ser atualizadas. Quando algum usuário quer fazer uma modificação em uma versão liberada ou efetiva, tem que derivar uma nova edição.

Uma versão efetiva é criada pela promoção de uma edição, para ser testada dentro de um grupo. Esta promoção deve ser feita pelo dono da versão. Da mesma forma, a liberação de uma versão efetiva é responsabilidade do representante do grupo.

A criação de uma edição pode se dar através da derivação da última versão, ou a partir de um arquivo de trabalho. Quando a edição é criada por derivação, inicialmente ela consiste em uma cópia da versão da qual ela se originou. Em uma derivação, se o usuário já tem edições do objeto, a última delas é derivada. Caso contrário, a derivação é feita a partir da última versão congelada, seja ela efetiva ou liberada.

Vários usuários podem ter edições da última versão congelada de um objeto de projeto. No entanto, quando uma destas edições for congelada, todas as outras passam a ser derivadas desta que foi recentemente congelada. Isto restringe a concorrência somente ao último nível.

A.3.2 Numeração

O número da versão de uma classe consiste de quatro campos, atualizados automaticamente pelo ambiente. São eles:

E.I+Ed, onde

E (identificação de versão externa): Sempre que uma versão sofre alterações na interface da classe (parâmetros formais, lista de classes referenciadas e funções exportáveis) o campo E é acrescido de 1;

I (identificação de versão interna): O campo I é acrescido de 1 quando a versão sofre alterações somente nas informações internas à classe, mantendo inalterada a sua interface em relação a versão anterior. Quando E é incrementado, o campo I é zerado;

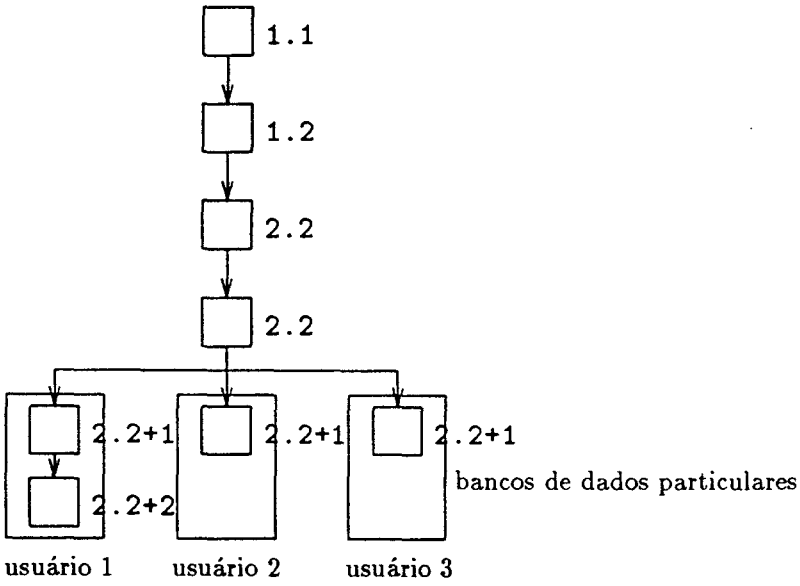

Figura A.1: Estrutura de derivação de um objeto do A_HAND

+ (indicação de derivação de versão): Significa que a edição é derivada da versão indentificada por E.I;

Ed (identificação das edições de um usuário): É um campo adicional ao número da versão da qual a edição é derivada. As edições de uma versão serão numeradas em seqüência neste campo.

Este tipo de controle é necessário para evitar recompilações desnecessárias, pois uma classe alterada somente internamente precisa ser recompilada, mas as classes que a usam não necessitam de recompilação [Drum87a, Drum87b].

A Figura A.1 mostra o histórico de um objeto de projeto, onde suas versões são apresentadas com os respectivos números.

Quando uma edição é efetivada, ela recebe um novo número, que é subseqüente ao número da versão congelada que lhe deu origem. As outras edições, passam então a ser sucessoras desta, e portanto têm os números alterados também. No exemplo da Figura A.1, se a edição 2.2+1 do usuário 2 fosse efetivada, ela receberia o número 2.3. Simultaneamente, as edições do usuário 1, passariam a ter os números 2.3+1 e 2.3+2. O mesmo se daria com a edição do usuário 3.

A.4 Configurações

As classes Cm têm relacionamentos de importação e herança com outras classes, e todas elas podem possuir várias versões. Assim, é necessário uma *definição de configuracão*, que faça a ligação da versão de uma classe a versões específicas das classes importadas ou herdadas. No A_HAND, as definições de configuração não são associadas a uma classe específica, elas definem globalmente as versões que devem compor um sistema.

Para usar uma classe é necessário ter a sua *visão de execução*. Uma visão de execução é composta por um conjunto S de pares do tipo (identificador de objeto, número de versão), ou (OP, v), onde v identifica univocamente uma versão. O número da versão é determinado pelas entradas que compõem a definição de configuração. Em outras palavras a definição de configuração determina como o número das versões são encontradas e a visão de execução origina-se da resolução de uma definição de configuração para uma determinada classe. Neste modelo, as entradas de uma definição de configuração podem ser de três tipos:

Uma entrada direta (OP, i): Este tipo de en trada especifica de forma explícita a versão v que deve ser usada para um objeto de projeto OP. O i pode ser representado por um número de versão ou por um rótulo. Quando ele é um número, o par (OP, i) é diretamente inserido no conjunto S. Os rótulos existentes no ambiente são ULTLIB, ULTEF e CORRENTE. Caso seja usado o rótulo ULTLIB, o número de versão correspondente à última versão liberada é procurada e o par (OP, número de versão) é inserido em S. O rótulo ULTEF corresponde ao número da última versão efetiva, enquanto que o rótulo CORRENTE é substituído pela última edição do usuário, se ela existir. Caso contrário é procurada a última versão efetiva e depois a última liberada.

Uma entrada indireta (OP, dc): A versão do objeto OP a ser inserida no conjunto S, é a mesma que a determinada na definição de configuração dc.

Uma entrada de inclusão (dc, p): A inclusão especifica uma prioridade p para uma definição de configuração dc. A prioridade define a ordem das entradas de inclusão. A cada definição de configuração incluída é construído um conjunto de pares S', S", etc ... É feito então, o *merge* de S com os outros conjunto de pares. Para cada objeto de projeto OP, que não tenha sido definido em S, será escolhido o número de versão que esteja definido no conjunto de pares de maior prioridade.

A visão de execução pode ser armazenada no banco de dados, evitando que a sua resolução seja repetida cada vez que a definição for usada, para o mesmo objeto.

Além disso, no congelamento de uma versão é especificada uma visão de execução que será associada a ela. Desta forma, as suas ligações com as versões referenciadas ficam também congeladas.

A.5 Definição do Ambiente

Para que seja gerado um controlador de versões e configurações adequado ao A_HAND, deve ser feita uma declaração na linguagem de definição apresentada no Apêndice B. As sentenças que compõem a definição devem ser formadas de modo a refletir as características do ambiente.

No A_HAND, as classes Cm são traduzidas para a linguagem C, para depois serem compiladas. Em alguns casos, uma classe Cm pode ter várias traduções em C, dependendo dos valores dos seus parâmetros. Como a verificação dessa possibilidade nem sempre é fácil, faz-se uma tradução para cada conjunto de parâmetros definidos. Antes de realizar a tradução de uma classe, é necessário verificar se as classes das quais ela depende já estão compiladas para os parâmetros adequados. Se não estão, elas devem ser compiladas antes da classe que depende delas. Assim, é possível armazenar as informações usando duas representações: uma para armazenar o código fonte em Cm, e outra que armazena o código C, código objeto e executável.

A numeração, os rótulos e as configurações são usados só para as classes Cm, portanto os atributos que representam os números de versão externa, de versão interna e edição, são exclusivos da representação que armazena o código fonte Cm. Para facilitar as comparações entre as interfaces, elas serão armazenadas em um atributo específico em cada versão. Um outro atributo será definido para guardar a data da atualização da interface, enquanto que atributo *atualização* mantido pelo próprio gerenciador armazenará a data de atualização de qualquer parte da versão.

As versões geradas pela tradução serão armazenadas no banco de dados, mas não serão mantidas configurações para elas. Cada uma delas será ligada à configuração da classe Cm que deu origem a sua tradução, através de um de seus atributos. Elas terão atributos especiais para armazenar o código objeto e executável, já que o código fonte em C será armazenado no atributo *informação* existente em todas as versões. A data da tradução será guardada no atributo *atualização* da versão traduzida.

As configurações não precisam de numeração especial, pois a numeração usada pelo A_HAND também é seqüencial.

```
representacao Cm
RATR_VER
   numero_v_externa : INT
   numero_v_interna : INT
   numero_v_edicao : INT
   interface : LONG_FIELD
   atualizacao_interface: TIME

representacao codigo_traduzido
RATR_VER
   codigo_objeto : LONG_FIELD
   codigo_fonte : LONG_FIELD
   traducao_de : ID_CONFIGURACAO
```

As versões congeladas são organizadas de forma linear, mas vários usuários podem ter edições derivadas paralelamente de uma mesma versão congelada. Como a estrutura de derivação do MVC, organiza todas as versões de um mesmo objeto, ela deve ter a forma de árvore para poder documentar o processo de derivação do A_HAND.

```
estrutura ramificada
```

As versões serão classificadas de acordo com o tipo de banco de dados em que estão armazenadas. As edições estarão nos bancos de dados particulares de cada usuário, enquanto que as versões efetivas estarão nos de projeto, e as liberadas ficarão no banco de dados geral.

Esta classificação induz à definição de três partições, cada uma controlando o acesso a um tipo de banco de dados. Contudo, entre as versões liberadas e efetivas de um objeto, só uma pode ser derivada: a última delas. Da mesma forma, entre as edições de uma versão particular só a última pode ser derivada. Portanto, é necessário criar mais uma partição em cada banco de dados para armazenar as versões que podem ser derivadas.

As *definições de configuração* são centralizadas, e para representá-las só as configurações globais serão usadas. As configurações ligadas as versões terão que existir para guardar as *visões de execução* do A_HAND. Os usuários poderão atualizar as configurações globais do seu banco de dados, e derivar as de outros bancos de dados. Como as visões de execução serão geradas pelo processo de resolução, não será dada permissão para atualização de configurações ligadas às versões. Todavia, os usuários poderão inserir e derivar configurações, para que seja possível fazer resoluções que gerem novas visões de execução.

No A_HAND os representantes de projeto e do ambiente têm funções puramente administrativas, portanto eles não têm permissões para derivar ou atualizar versões e configurações. Eles têm permissões para inserir, eliminar ou transferir objetos versões e configurações, para poderem realizar as tarefas de organização do seu banco de dados. Eles podem também derivar e atualizar configurações globais.

Em geral, os usuários precisam ter certeza que as configurações globais dos bancos de dados de projeto e público, que eles estão usando não serão alteradas. Por isso, só as configurações globais de uma partição poderão ser atualizadas, e ainda assim a permissão é dada somente ao dono do banco de dados.

Como a liberação de uma versão é feita pelo representante de grupo, ele deve ter permissão para inserir e transferir versões e configurações no banco de dados geral. O mesmo acontece com os usuários particulares, pois ao efetivar uma de suas versões eles têm que inserí-la no banco de dados de projeto, e transferir a versão antecessora para uma partição onde não são permitidas derivações, no banco de dados de projeto ou geral.

Assim, a definição do ambiente é finalizada com as seguintes declarações de partição:

```
particao liberada
banco de dados geral
propriedades
```

```
   insercao
      objeto (projeto, geral);
      versao (particular, projeto, geral);
      configuracao (particular, projeto, geral);
   eliminacao
      objeto (geral);
      versao (geral);
      configuracao (geral);
   transferencia
      objeto (geral);
      versao (projeto, geral);
      configuracao (projeto, geral);
   derivacao
      configuracao (particular);
      configuracao global (particular, projeto, geral)

particao liberada_derivavel
banco de dados geral
propriedades
   insercao
      objeto (projeto, geral);
      versao (projeto, geral);
      configuracao (projeto, geral);
   atualizacao
      configuracao global (geral);
   eliminacao
      objeto (geral);
      versao (geral);
      configuracao (geral);
   transferencia
      objeto (geral);
      versao (particular, projeto, geral);
      configuracao (geral);
   derivacao
      versao (particular);
      configuracao (particular)
      configuracao global (particular, projeto, geral);

particao efetiva
banco de dados projeto
propriedades
   insercao
      objeto (projeto);
      versao (particular, projeto);
```

```
      configuracao (particular, projeto);
   eliminacao
      objeto (projeto);
      versao (projeto);
      configuracao (projeto);
   transferencia
      objeto (projeto);
      versao (projeto);
      configuracao (projeto);
   derivacao
      versao (particular);
      configuracao (particular);
      configuracao global (particular, projeto)

particao efetiva_derivavel
banco de dados projeto
propriedades
   insercao
      objeto (particular, projeto);
      versao (particular, projeto);
      configuracao (particular, projeto);
   atualizacao
      configuracao global (projeto);
   eliminacao
      objeto (projeto);
      versao (projeto);
      configuracao (projeto);
   transferencia
      objeto (projeto);
      versao (particular, projeto);
      configuracao (particular, projeto);
   derivacao
      versao (particular);
      configuracao (particular)
      configuracao global (particular, projeto);

particao reservada
banco de dados particular
propriedades
   insercao
      objeto (particular);
      versao (particular);
      configuracao (particular);
   atualizacao
```

```
      objeto (particular);
      versao (particular);
      configuracao (particular);
   eliminacao
      objeto (particular);
      versao (particular);
      configuracao (particular);
   transferencia
      objeto (particular);
      versao (particular);
      configuracao (particular);
   derivacao
      configuracao (particular);
      configuracao global (particular)

particao reservada_derivavel
banco de dados particular
propriedades
   insercao
      objeto (particular);
      versao (particular);
      configuracao (particular);
   atualizacao
      objeto (particular);
      versao (particular);
      configuracao (particular);
      configuracao global (particular);
   eliminacao
      objeto (particular);
      versao (particular);
      configuracao (particular);
   transferencia
      objeto (particular);
      versao (particular);
      configuracao (particular);
   derivacao
      versao (particular);
      configuracao (particular);
      configuracao global (particular)
```

A.6 Utilização das Primitivas

O arquivo contendo as declarações acima, serve como entrada para a ferramenta que gera o esquema dos bancos de dados que armazenam as informações do ambiente, e as primitivas do modelo. Feita a geração, o responsável pelo ambiente deve cadastrar os grupos e os seus integrantes. A partir de então qualquer usuário pode realizar o seu trabalho através das primitivas geradas. A descrição detalhada de todas elas estão no Apêndice D.

Inicialmente, cada usuário deve começar uma transação, que só vai ser encerrada quando o trabalho terminar, pois a maioria das operações esta relacionada a uma transação. Entre o começo e o fim de uma transação, ele pode ter saído e entrado inúmeras vezes no ambiente, e vice-versa.

```
OpeEnv(''Maria'');
BrgTra(''transacao'', chave_transacao);
   ...
```

O representante do ambiente deve também definir as consultas CORRENTE, ULTLIB e ULTEF, para que todos usuários possam usá-las em suas definições de configuração. A definição mais complexa é a da consulta identificada pelo rótulo CORRENTE. Ela é uma consulta direta, que tem como resultado uma versão e consiste em três cláusulas:

versão.partição = reservada_derivável *or*
bando_de_dados : CV_PRJ *and* versão.número : MAX *or*
banco_de_dados : CV_GER *and* versão.número : MAX

A versão selecionada será a que está na partição *reservada_derivável* do banco de dados do usuário. Se não existir, procura-se pela última versão efetiva. E finalmente, a última versão liberada será procurada. A definição das primeiras cláusulas e predicados seria feita da seguinte forma:

```
erro = DefCons(CORRENTE, CV_VERS, CV_DIRETA, chave_cons);

/* Funcao que verifica o codigo de erro retornado, e deve ser
chamada depois de cada operacao realizada */
if (err_occurred(erro, CV_OK, ''DefCons''))
   return(erro);
erro = CheOutCons(chave_cons, CV_E, chave_transacao,
          chave_consulta);
erro = InsCla (chave_consulta, CV_NCHAVE, CV_PRIMEIRA,
          chave1_clausula);
erro = InsPre (chave_consulta, chave1_clausula, CV_NCHAVE,
          CV_PRIMEIRA, ''particao'', CV_IG, ''reservada_derivavel'',
          0, FALSE, chave1_predicado);
```

```
erro = InsCla (chave_consulta, chave1_clausula, CV_PROXIMA,
          chave2_clausula);
erro = InsPre (chave_consulta, chave2_clausula, CV_NCHAVE,
          CV_PRIMEIRA, ''bd'', CV_PRJ, null, 0, FALSE,
          chave2_predicado);
erro = InsPre (chave_consulta, chave2_clausula, chave2_predicado,
          CV_PROXIMA, ''numero'', CV_MAX, null, 0, FALSE,
          chave3_predicado);
```

Seria interessante também definir a função *Insere_versão*, pois o usuário só pode criar uma nova edição, como sucessora da última edição, ou da última versão congelada de cada objeto. Esta função faria as verificações necessárias pelo usuário. Ela procuraria por uma versão daquele objeto armazenada na partição *reservada_derivável* do seu banco de dados. Se não encontrasse, procuraria por uma versão deste mesmo objeto na partição *efetiva_derivável* e depois na *liberada_derivável*. A nova versão seria inserida na partição *reservada_derivável* como sucessora da versão encontrada. Caso não existisse nenhuma delas, a nova versão seria inserida na mesma partição, mas como sucessora direta do objeto genérico. Para resolver este problema, poderia ser definida uma consulta direta ULT_VER_PART que recebesse como parâmetro um nome de partição, e retornasse a chave da última versão de um determinado objeto que estivesse naquela partição. A função seria definida com as seguintes chamadas:

```
particao = reservada_derivavel;
strcpy(parametro.id, ''particao'');
parametro.valor = &(particao);
parametro.tam = sizeof(particao);
parametro.proximo = NULL;
erro = Consdir(ULT_VER_PART, CV_NCHAVE, &(parametro),
          chave_ck_objeto, chave_antecessora);
if (erro == CV_NENC){
   particao = efetiva_derivavel;
   erro = Consdir(ULT_VER_PART, CV_NCHAVE, &(parametro),
      chave_ck_objeto, chave_antecessora);
   if (erro == CV_NENC){
      particao = liberada_derivavel;
      erro = ConsDir(ULT_VER_PART, CV_NCHAVE, &(parametro),
        chave_ck_objeto, chave_antecessora);
   }
}
if ((erro != CV_NENC) && (erro != CV_OK))
   return(erro);
if (erro == CV_NENC)
   erro == InsVer(chave_ck_objeto, CV_NCHAVE, reservada_derivavel,
      chave_vers\~{a}o, numero)
```

```
else{
   erro = CheOutVer(chave_antecessora, CV_DPE, chave_transacao,
      chave_ck_antecessora);
   erro = InsVer(chave_ck_objeto, chave_ck_antecessora,
      reservada_derivavel, chave_versao, numero);
}
return(erro);
```

Supondo agora que o usuário queira, criar em Cm uma versão do objeto chamado Msort que usa os objetos Sort e Merge. E que depois de definida a nova versão, ele queira testá-la. Para isso, será usada a última versão liberada do objeto Merge, e a versão 5 de Sort.

A princípio, ele cria o objeto Msort. Depois ele carrega no endereço apontado por buffer1 a descrição da sua funcionalidade, e atualiza o atributo *descrição* do objeto com este valor. Observe que o *check-out* poderia ser realizado no modo CV_D, se fosse só para a derivação de versões. Mas como os atributos também vão ser atualizados é necessário realizá-lo no modo CV_E.

```
erro = InsObj(''Msort'', Cm, reservada, chave_objeto);
erro = CheOutObj(chave_objeto, CV_E, chave_transacao,
   chave_ck_objeto);
erro = UpdObjAtt(chave_ck_objeto, descricao, tamanho_buffer1,
   buffer1)
```

E então, ele insere a primeira versão através da função *Insere_versão*, atualiza o seu código com o conteúdo do buffer2 e a interface com o conteúdo de buffer3 e cria referências para o objeto Sort e Merge. Antes porém, ele deve procurar estes objetos para saber a sua chave.

```
erro = Insere_versao(chave_ck_objeto, chave_versao);
erro = CheOutVer(chave_vers\~{a}o, CV_E, chave_transacao,
   chave_ck_versao);
erro = UpdVerAtt(chave_ck_versao, informacao, tamanho_buffer2,
   buffer2);
erro = UpdVerAtt(chave_ck_versao, interface, tamanho_buffer3,
   buffer3);
 ...
erro = FinObjNam( ''Sort'', Cm, chave_obj1);
if (erro == CV_OK){
   erro = CheOutObj(chave_obj1, CV_E, chave_transacao,
      chave_ck_obj1);
   erro = InsRef(chave_ck_versao, chave_ck_obj1);
}
else
```

```
   return(erro);
erro = FinObjNam( ''Merge'', Cm, chave_obj2);
if (erro == CV_OK){
   erro = CheOutObj(chave_obj2, CV_E, chave_transacao,
      chave_ck_obj2);
   erro = InsRef(chave_ck_versao, chave_ck_obj2);
}
else return(erro);
```

Como não é permitido trabalhar com configurações locais, ele deve inserir uma configuração global, com as entradas para as versões que devem compor o sistema. Assim, ele pode procurar a versão desejada de um objeto referenciado para inserir em uma entrada de configuração, por exemplo a versão 5 de *Sort*. Se quiser, ele pode ler a sua interface para se certificar de que a chamada foi feita corretamente.

```
erro = InsOveConf(Cm, CV_NCHAVE, reservada_derivavel,
   chave_config, numero_config);
erro = CheOutConf(chave_config, CV_E, chave_transacao,
   chave_ck_config);
erro = FinVerNum(chave_ck_obj1, 5, chave_ver1);
erro = CheOutVer(chave_ver1, CV_LPE, chave_transacao,
   chave_ck_ver1);
erro = VerAttSiz (chave_ck_ver1, interface, tamanho_buffer4);
```

O usuário deve declarar uma espaço com as dimensões de tamanho_buffer4, e colocar a descrição da interface da versão 5 de Sort neste espaço, para pode ler e verificar se ela é compatível com a chamada de Msort criada por ele.

```
erro = GerVetAtt(chave_ck_ver1, interface, buffer4);
...
erro = InsDirEnt(chave_ck_config, CV_NCHAVE, CV_PRIMEIRA,
   CV_NCHAVE, chave_ck_ver1, NULL, chave_ent1);
```

Para usar a última versão liberada do objeto Merge, o usuário deve inserir uma entrada direta de consulta.

```
strcpy(parametro.id, ''particao'');
particao = liberada;
parametro.valor = &(particao);
parametro.tam = sizeof(particao);
parametro.proximo = NULL;
erro = FinCons(ULT_VER_PART, chave_consulta);
erro = CheOutCons(chave_consulta, CV_LPE, chave_transacao,
   chave_ck_consulta);
erro = InsDirEnt(chave_ck_config, chave_ent1, CV_PROXIMA,
   chave_ck_obj2, chave_ck_consulta, &(parametro), chave_ent2);
```

Quando ele for testar a sua versão, ele tem que resolver a configuração. Na resolução ele não precisa dar a chave da configuração que deve ser usada, pois neste caso o gerenciador vai usar a configuração global ativa. Após a resolução todas as versões que compõem o sistema estariam protegidas, permitindo que ele acesse qualquer uma elas.

```
erro = ConfRes(chave_ck_versao, CV_NCHAVE, 5, CV_LPE,
   chave_conf_resolvida);
erro = FinDirEnt(chave_conf_resolvida, CV_NCHAVE, CV_PRIMEIRA,
   chave_entrada, &(entrada);
```

Ao percorrer a *configuração_resolvida*, a primeira entrada encontrada é a do objeto Merge. No campo **ck_ver_ch** da estrutura que armazena a entrada esta registrada a chave da cópia correspondente ao *check-out* da última versão liberada deste objeto. O mesmo ocorre com a entrada correspondente ao objeto sf Sort.

Desta forma, ele pode usar todas as versões envolvidas para a compilação. Se ele estiver satisfeito com os resultados ele pode transformar a *configuração_resolvida* em uma configuração comum. A informação resultante da compilação pode ser armazenada em uma versão do objeto Msort na representação código traduzido. Depois de feito isso, o campo *tradução_de* da nova versão receberia a chave da configuracão que deu origem a ela.

```
erro = InsRes(chave_conf_resolvida, reservada,
   chave_conf_transformada);
erro = UpdVerAtt(chave_ver_traduzida, traducao_de,
   sizeof(chave_conf_transformada), chave_conf_transformada);
```

Estas operações realizam uma pequena parte do trabalho que o usuário teria que fazer. No entanto, através deste exemplo é possível compreender como as primitivas do modelo seriam utilizadas em um ambiente real.

Apêndice B

Sintaxe da Linguagem de Definição

```
definicao
     : AMBIENTE id_ambiente def_objetos FIM
     ;

id_ambiente
     :IDENTIFICADOR
     ;

def_objetos
     :
     |def_objetos dec_constantes
     |def_objetos dec_dominios
     |def_objetos dec_atributos
     |def_objetos dec_representacao
     |def_objetos dec_particao
     |def_objetos dec_estrutura
     ;

dec_constantes
     :CONST defs_constantes
     ;

defs_constantes
     :def_constante
     |defs_constantes ';' def_constante
     ;
```

```
def_constante
     :id_constante ':' expr_constante
     ;

expr_constante
     :expr_constante '+' termo
     |expr_constante '-' termo
     |termo
     ;

termo
     :termo '*' fator
     |termo '/' fator
     |fator
     ;

fator
     :'-' constante
     |constante
     ;

constante
     :'('expr_constante')'
     |nome_constante
     |INT_CONSTANTE
     |CHAR_CONSTANTE
     |STRING_CONSTANTE
     ;

id_constante
     :IDENTIFICADOR
     ;

nome_constante
     :IDENTIFICADOR
     ;

dec_dominios
     :DOMINIOS defs_dominios
     ;

defs_dominios
     :def_dominio
```

```
      |defs_dominios ';' def_dominio
      ;

def_dominio
      :id_dominio ':' dominio
      ;

id_dominio
      :IDENTIFICADOR
      ;

dominio
      :nome_dominio
      |ENUM '{' list_id_constante '}'
      |STRING '[' expr_constante ']'
      |BYTES '[' expr_constante ']'
      |nome_dominio SUBR '[' expr_constante PP expr_constante ']'
      |nome_dominio ARRAY '[' expr_constante ']'
      |STRUCT list_atributo END
      |UNION list_atributo END
      ;

list_id_constante
      :id_constante
      |list_id_constante ';' id_constante
      ;

dec_atributos
      :atributos_objetos
      |atributos_versao
      |atributos_configuracao
      ;

atributos_objetos
      :ATR_OBJ list_atributo
      ;

atributos_versao
      :ATR_VER list_atributo
      ;

atributos_configuracao
      :ATR_CONF list_atributo
      ;
```

```
list_atributo
     :def_atributo
     |list_atributo ';' def_atributo
     ;

def_atributo
     :id_atributo ':' nome_dominio
     ;

id_atributo
     :IDENTIFICADOR
     ;

nome_dominio
     :INT
     |CHAR
     |BOOL
     |TIME
     |LONG_FIELD
     |ID_USUARIO
     |ID_GRUPO
     |ID_OBJETO
     |ID_VERSAO
     |ID_CONFIGURACAO
     |IDENTIFICADOR
     ;

dec_representacao
     :REPRESENTACAO id_representacao
     |REPRESENTACAO id_representacao rdec_atributos
     ;

rdec_atributos
     :ratributos_objetos
     |rdec_atributos ratributos_objetos
     |ratributos_versao
     |rdec_atributos ratributos_versao
     |ratributos_configuracao
     |rdec_atributos ratributos_configuracao
     ;

ratributos_objetos
     :RATR_OBJ list_atributo
```

```
      ;

ratributos_versao
      :RATR_VER list_atributo
      ;

ratributos_configuracao
      :RATR_CONF list_atributo
      ;

id_representacao
      :IDENTIFICADOR
      ;

dec_particao
      :PARTICAO id_particao BD id_b_dados PROPRIEDADES list_prop
      ;


id_particao
      :IDENTIFICADOR
      ;

id_b_dados
      :PARTICULAR
      |PROJETO
      |GERAL
      ;

list_prop
      :def_propriedade
      |list_prop ';' def_propriedade
      ;

def_propriedade
      :id_operacao '('list_permissoes')'
      ;

id_operacao
      :O_INSERCAO
      |V_INSERCAO
      |C_INSERCAO
      |O_ELIMINACAO
      |V_ELIMINACAO
```

```
      |C_ELIMINACAO
      |O_ATUALIZACAO
      |V_ATUALIZACAO
      |C_ATUALIZACAO
      |O_TRANSFERENCIA
      |V_TRANSFERENCIA
      |C_TRANSFERENCIA
      |V_DERIVACAO
      |C_DERIVACAO
      ;


list_permissoes
      :permissao
      |list_permissoes ',' permissao
      ;


tipo_usuario
      :PARTICULAR
      |PROJETO
      |GERAL
      ;


permissao
      :tipo_usuario
      |OBJETO'.'nome_atributo
      |OBJETOI'.'nome_atributo
      |VERSAO'.'nome_atributo
      |VERSAOI'.'nome_atributo
      |CONFIGURACAO'.'nome_atributo
      |CONFIGURACAOI'.'nome_atributo
      ;

nome_atributo
      :IDENTIFICADOR
      |AUTOR
      ;

dec_estrutura
      :ESTRUTURA forma
      ;
```

```
forma
      :LINEAR
      |RAMIFICADA
      ;
```

Apêndice C

Geração do Esquema

```
SCHEMA <id_ambiente>

    CONST
       TAM_NOME = 25;
       enquanto <def_constante>
          imprime <def_constante>;


     VALUE_SET
        Cv_nome : STRING [TAM_NOME];
        Cv_modo : INT;
        Cv_operador : ENUM {IG, MA, ME, MAIG, MEIG, DIF, MAX, MIN};
        Cv_estado: ENUM {RESOLVIDA, NAO_RESOLVIDA};
        Cv_resultado : ENUM {OBJ, VER, CONF};
        Cv_t_consulta : ENUM {C_DIRETA, C_INDIRETA};
        Cv_representacao : ENUM {
        enquanto <dec_representacao>
           imprime <id_representacao,
                                    }
        Cv_particao : ENUM {
        enquanto <dec_particao>
           imprime <id_particao>,
                                  }
```

```
      enquando <def_dominio>
         imprime <def_dominio>;


OBJECT TYPE Obj_desenvolvimento
   ATTRIBUTES
      nome : Cv_nome;
      criacao : Time;
      atualizacao : Time;
      particao: Cv_particao;
      descricao : LONG_FIELD;
      representacao : Cv_representacao;
      ult_versao : INT;
      se <atributos_objetos>
         enquanto <list_atirbuto>
            imprime <def_atributo>
   VERSIONS <forma>
      (
          ATTRIBUTES
             numero : INT
      )
   STRUCTURE IS
      Ck_rel
   se existe <dec_representacao> com <ratributos_objeto>
       ,  At_o_representacao
END Obj_desenvolvimento;


se existe <dec_representacao> com <ratributos_objetos>{
   OBJECT TYPE At_o_representacao
      IS UNION OF
         para cada <dec_representacao> com <ratributos_objeto>
            O_<id_representacao>,
   END At_o_representacao;

   para cada <dec_representacao> com <ratributos_objeto>{
      OBJECT TYPE O_<id_representacao>
         ATTRIBUTES
            enquanto <list_atributo>
               imprime <def_atributo>
       END O_<id_representacao>
   }
}
```

```
OBJECT TYPE Versao
   ATTRIBUTES
      numero : INT;
      criacao : Time;
      atualizacao : Time;
      particao: Cv_particao;
      representacao: Cv_representacao;
      informacao : LONG_FIELD;
      ult_configuracao : INT
      se <atributos_versao>
         enquanto <list_atirbuto>
            imprime <def_atributo>
   VERSIONS <forma>
      (
          ATTRIBUTES
             numero : INT;
             estado : Cv_estado
      )
   STRUCTURE IS
      Ck_rel,
      V_local,
      Referencia
   se existe <dec_representacao> com <ratributos_versao>
       ,  At_v_representacao
END Versao;


se existe <dec_representacao> com <ratributos_versao>{
   OBJECT TYPE At_v_representacao
      IS UNION OF
         para cada <dec_representacao> com <ratributos_versao>
            V_<id_representacao>,
   END At_v_representacao;

   para cada <dec_representacao> com <ratributos_versao>{
      OBJECT TYPE V_<id_representacao>
         ATTRIBUTES
            enquanto <list_atributo>
               imprime <def_atributo>
       END V_<id_representacao>
   }
}
```

```
OBJECT TYPE Configuracao
   ATTRIBUTES
      numero : INT;
      criacao : Time;
      atualizacao : Time;
      particao: Cv_particao;
      representacao: Cv_representacao;
   STRUCTURE IS
      Ck_rel,
      C_local,
      Glo_local,
      Ent_direta,
      Ent_indireta,
      Ent_default,
      Ent_consulta,
      Numero
   se existe <dec_representacao> com <ratributos_configuracao>
       ,  At_c_representacao
END Configuracao;


se existe <dec_representacao> com <ratributos_configuracao>{
   OBJECT TYPE At_c_representacao
      IS UNION OF
         para cada <dec_representacao>com <ratributos_configuracao>
            C_<id_representacao>,
   END At_c_representacao;

   para cada <dec_representacao> com <ratributos_configuracao>{
      OBJECT TYPE C_<id_representacao>
         ATTRIBUTES
            enquanto <list_atributo>
               imprime <def_atributo>
       END C_<id_representacao>
   }
}


OBJECT TYPE Global_conf
      ATTRIBUTES
         representacao : Cv_representacao;
```

```
            ult_configuracao : INT
         VERSIONS <forma>
         (
             numero : INT;
         )
      END Global_conf;

OBJECT TYPE U_escrito
   IS UNION OF
      Obj_desenvolvimento,
      Versao,
      Configuracao
END U_escrito;


OBJECT TYPE U_autor
   IS UNION OF
      Ambiente,
      Grupo,
      Usuario
END U_autor;

RELSHIP TYPE Autoria
   RELATES
      obra : U_escrito,
      criador : U_autor
END Autoria;

OBJECT TYPE C_resolvida
   STRUCTURE IS
      C_resolvida, Resolucao_de, Ent_direta
END C_resolvida;

OBJECT TYPE U_resolucao
   IS UNION OF
      Versao,
      Configuracao
END U_resolucao;

RELSHIP TYPE Resolucao_de
   RELATES
      resol : C_resolvida,
      def : U_resolucao
END Resolucao_de;
```

```
RELSHIP TYPE V_c_ativacao
   RELATES
      ativado : U_ativa,
      ativa : Ck_copia
END V_c_ativacao;

RELSHIP TYPE O_c_ativacao
   RELATES
      ativado : Obj_desenvolvimento,
      ativa : Ck_copia
END O_c_ativacao;

RELSHIP TYPE O_v_ativacao
   RELATES
      ativado : Obj_desenvolvimento,
      ativa : Ck_copia
END O_v_ativacao;

OBJECT TYPE U_ativa
   IS UNION OF
      Versao,
      Global_conf
END U_ativa;

OBJECT TYPE Ambiente
   ATTRIBUTES
      nome : Cv_nome;
      ult_bd : INT
END Ambiente;

OBJECT TYPE Grupo
   ATTRIBUTES
      nome: Cv_nome;
      responsavel : Cv_nome;
      bd : INT;
END Grupo;

OBJECT TYPE Usuario
   ATTRIBUTES
      nome : Cv_nome;
      bd : INT;
END Usuario;
```

```
RELSHIP TYPE Coordenacao
   RELATES
      chefe : Grupo,
      chefiado : Usuario
END Coordenacao;

RELSHIP TYPE V_local
   RELATES
      aponta : Obj_desenvolvimento.VERSION,
      armazena : Versao
END V_local;

RELSHIP TYPE C_local
   RELATES
      aponta : Versao.VERSION,
      armazena : Configuracao
END C_local;

RELSHIP TYPE Glo_local
   RELATES
      aponta : Global_conf.VERSION,
      armazena : Configuracao
END Glo_local;

OBJECT TYPE Ck_copia
   ATTRIBUTES
      bd : INT
   STRUCTURE IS
      O_v_ativacao, O_c_ativacao, V_c_ativacao
END Ck_copia;

RELSHIP TYPE Ck_rel
   RELATES
      original : Ck_original,
      copia : U_copia
   ATTRIBUTES
      modo : Cv_modo
END Ck_rel;

OBJECT TYPE U_copia
   IS UNION OF
      Ck_copia,
      C_resolvida
END U_copia;
```

```
OBJECT TYPE Ck_original
   IS UNION OF
      Obj_desenvolvimento,
      Versao,
      Configuracao,
      Global_conf
END Ck_original;

RELSHIP TYPE Referencia
   RELATES
      faz_ref : Versao,
      eh_ref : Obj_desenvolvimento
END Referencia;

OBJECT TYPE Dconsulta
   ATTRIBUTES
      rotulo : Cv_nome;
      codigo : LONG_FIELD;
      resultado : Cv_resultado;
      tipo : Cv_t_consulta
   STRUCTURE IS
      Clausula
END Dconsulta;

OBJECT TYPE Clausula
   STRUCTURE IS
      Predicado
END Clausula;

OBJECT TYPE Predicado
   ATTRIBUTES
      nome : Cv_nome;
      operador : Cv_operador;
      valor : Cv_buffer;
      eh_parametro : BOOL;
      nome_dominio : Cv_nome
END Predicado;

OBJECT TYPE Parametro
   ATTRIBUTES
       nome : Cv_nome;
       valor : Cv_buffer
END Parametro;
```

```
OBJECT TYPE Ent_consulta
   STRUCTURE IS
      Parametro, Dcon_consulta, Obj_consulta
END Ent_consulta;


RELSHIP TYPE Dcon_consulta
   RELATES
      dcon : Dconsulta,
      cons : Ent_consulta
END Dcon_consulta;

RELSHIP TYPE Obj_consulta
   RELATES
      obj : Obj_desenvolvimento,
      cons : Ent_consulta
END Obj_consulta;

OBJECT TYPE U_conf
   IS UNION OF
      Configuracao,
      C_resolvida
END U_conf;

RELSHIP TYPE Ent_direta
   RELATES
      conf : U_conf,
      def  : U_direta
END Ent_direta;

OBJECT TYPE U_direta
   IS UNION OF
      U_conf,
      Versao,
      Ent_consulta
END U_direta;

RELSHIP TYPE Ent_indireta
   RELATES
      conf : Configuracao,
      def  : U_indireta
END Ent_indireta;
```

```
OBJECT TYPE U_indireta
   IS UNION OF
      Configuracao,
      Ent_consulta
END U_indireta;

RELSHIP TYPE Ent_default
   RELATES
      conf : Configuracao,
      def  : U_default
END Ent_default;

OBJECT TYPE U_default
   IS UNION OF
      Ent_consulta,
      Numero
END U_default;

OBJECT TYPE Numero
   ATTRIBUTES
      n_ver : INT;
      n_conf: INT
END Numero;
```

Apêndice D

Detalhamento das Funções do Modelo

Cada seção deste apêndice descreve as funções relacionadas a um elemento específico do modelo.

Todas as funções retornam um código de erro, que descreve como se comportou a sua realização. No entanto, por uma questão de espaço os códigos não serão detalhados.

Os nomes escritos em itálico, representam parâmetros cujos valores são retornados pela função, enquanto os outros são parâmetros de entrada.

Observe também que, os identificadores terminados em "ckkey" se referem a chave da cópia feita por uma operação de *check- out*. Contudo, isto não será dito em cada descrição para evitar repetição.

D.1 Objetos

InsObj (objnam, objrep, part, *objkey*)

Insere um novo objeto, com nome objnam e representação objrep na partição part do banco de dados do usuário que realiza a operação. A operação só tem sucesso, se não houver outro objeto no mesmo banco de dados com o mesmo nome e representação. A chave do objeto inserido é retornado em objkey.

RemObj (objkey)

Remove o objeto identificado por objkey se ele não tiver nehuma versão, não for referenciado por versões de outros objetos, e não for identificado por atributos de outros elementos.

FinObjNam (objnam, objrep,*objkey*)

Procura um objeto que tenha o nome objnam e a representação objrep. Se o objeto for encontrado no banco de dados do usuário ou em algum dos quais o usuário tem acesso, sua chave é retornada em objkey.

ObjAttSiz (objckkey, attrnam, *num*)

Determina o tamanho do valor do atributo attrnam do objeto identificado por objckkey, e retorna em num.

GetObjAtt (objckkey, attrnam, *buffer*)

Coloca o valor do atributo attrnam do objeto identificado por objckkey, no espaço apontado por buffer.

UpdObjAtt (objckkey, attrnam, sizattr, buffer)

Atualiza o atributo attrnam do objeto identificado por objckkey como o valor armazenado no espaço apontado por buffer. Se o atributo tiver o domínio LONG_FIELD, o seu tamanho deve ser fornecido em sizattr.

GetObjAts (objckkey,*objbuffer*)

Coloca o valor dos atributos do objeto identificado por objckkey no espaço apontado por objbuffer. Os atributos que tem o domínio LONG_FIELD são ignorados.

UpdObjAts (objckkey, objbuffer)

Atualiza o valor dos atributos do objeto identificado por objckkey como o valor armazenado no espaço apontado por objbuffer. Os atributos que tem o domínio LONG_FIELD são ignorados.

GetRefObj (objckkey, anokey, mode, *verkey*)

Recupera uma versão que faz referência ao objeto identificado por objckkey, e que seja acessível ao usuário. A referência é procurada em uma posição especificada por mode com relação a referência feita pela versão anokey.

CheOutObj (objkey, mode, trakey, *objckkey*)

Realiza um *check-out* do objeto identificado por objckkey, do seu banco de dados original para o banco de dados do usuário. O objeto original fica com uma proteção cujo tipo é especificado por mode. Uma cópia do objeto é instalada no banco de dados do usuário, na qual ele poderá efetuar operações dependendo do tipo de mode. Esta cópia é identificada por objckkey e está associada a transação identificada por trakey. O parâmetro mode pode ter os valores CV_LPED, CV_LPE, CV_LSP, CV_E, CV_D.

CheInObj (objckkey)

Faz o *check-in* de um objeto que havia sofrido *check-out* anteriormente. O objeto original é atualizado de acordo com as alterações feitas na sua cópia identificada por objckkey. A cópia é então destruída e o original liberado.

TraObj (objkey, dbtype)

Faz a transferência de um objeto do banco de dados onde está armazenada para o banco de dados especificado por dbtype da lista de bancos de dados acessíveis pelo usuário. A operação só tem sucesso de dbtype especificar um nível superior ao banco de dados em que o objeto está armazenado, e se não houver nenhum outro objeto com o mesmo nome e representação neste banco de dados.

D.2 Versões

InsVer (objckkey, predckkey, confckkey, part, *verkey, vernum*)

Insere uma nova versão do objeto identificado por objckkey na partição part do banco de dados do usuário que realiza a operação. Se predkey é diferente de CV_NCHAVE, e identifica outra versão do mesmo objeto, a nova versão é inserida como sucessora da versão identificada por esta chave, e ela se apresenta inicialmente como uma cópia desta. Caso contrário ela é inserida como sucessora direta do objeto de desenvolvimento. Se confkey é igual a CV_NCHAVE, inicialmente a versão não tem nenhuma configuração, senão uma cópia da configuração identificada por confkey é inserida na nova versão. A chave da versão inserida é retornada em verkey e o seu número em vernum.

RemVer (verkey)

Remove a versão identificada por verkey. Todas as referências feitas por ela são também removidas. Se a versão tiver sucessoras, estas ficarão no grafo como sucessoras da versão que antecedia a versão removida. Se a versão tiver configurações, for referenciada em uma configuração ou for identificada por outro elemento através de um atributo cujo domínio seja VERKEY, a operação falha.

FinVerNum (objkey, num, *verkey*)

Procura a versão com o número num do objeto identificado por objkey. Se a versão for encontrada nos bancos de dados aos quais o usuário tem acesso, a sua chave é retornada em verkey.

GetNum (verckkey, *num*)

Dada a chave de uma versão retorna o seu número em num.

VerAttSiz (verckkey, attrnam, *num*)

Determina o tamanho do valor do atributo attrnam da versão identificada por verkey, e retorna em num.

GetVerAtt (verckkey, attrnam, *buffer*)

Coloca o valor do atributo attrnam da versão identificada por verckkey no espaço apontado por buffer.

UpdVerAtt (verckkey, attrnam, sizattr, buffer)

Atualiza o atributo attrnam da versão identificada por verckkey como o valor contido em buffer. Se o atributo tiver o domínio LONG_FIELD, o seu tamanho deve ser fornecido em sizattr.

GetVerAts (verckkey, *verbuffer*)

Coloca o valor dos atributos da versão identificada por verckkey no espaço apontado por verbuffer. Os atributos que tem o domínio LONG_FIELD são ignorados.

UpdVerAts (verckkey, verbuffer)

Atualiza o valor dos atributos da versão identificada por verckkey como o valor armazenado no espaço apontado por buffer. Os atributos que tem o domínio LONG_FIELD são ignorados.

ObjVerNum (objckkey, *num*)

Retorna em num o número de versões existentes do objeto identificado por objckkey.

GetObjVer (objckkey, *verlist, num*)

Recupera todas as versões do objeto identificado por objckkey, acessíveis ao usuário e organiza as chaves em uma lista, cujo endereço é retornado em verlist. O número de versões encontradas é retornado em num.

GetVerDes (verckkey, *verlist, num*)

Recupera todas as versões acessíveis ao usuário que sejam descendentes diretas da versão identificada por verckkey, e organiza as chaves em uma lista cujo endereço é retornado em verlist. O número de descendentes encontradas é retornado em num.

GetVerAnt (verckkey, *antkey*)

Recupera a versão antecessora da versão identificada por verckkey, e retorna sua chave em antkey.

FinVerSuc (antckkey, anokey, mode, *verkey*)

Recupera a sucessora da versão identificada por antckkey, que tenha uma posição especificada por mode em relação a versão identificada por anokey. A chave da versão encontrada, é retornada em verkey.

InsVerAlt (objckkey, verckkey, altname, *altkey*)

Insere um nome altname para alternativa do objeto identificado por objckkey, que inicia na versão identificada por verckkey. A chave da identificação é retornada em altkey.

RemVerAlt (objckkey, verckkey, altkey)

Remove o nome da alternativa do objeto identificado por objckkey, iniciada na versão verckkey e identificada pela chave altkey.

GetVerAlt (objckkey, anokey, mode, *altkey, altname*)

Recupera a identificação de ramificação do objeto objckkey, que esteja em uma posição relativa a identificação anokey especificada por mode. A chave da identificação encontrada é retornada em altkey e o nome em altname.

GetVerAltNam (verckkey, anokey, mode, *altkey, altname*)

Recupera a identificação de ramificação que designa a versão identificada por verckkey, que esteja em uma posição relativa a identificação anokey especificada por mode. A chave da identificação encontrada é retornada em altkey e o nome em altname.

GetVerAltDes (objckkey, altkey, *verlist, num*)

Recupera todas as versões do objeto identificado por objkey, sejam descendentes da ramificação identificada por altkey, e que o usuário tenha acesso. A versão que inicia a ramificação também é recuperada. As chaves das versões são organizadas em uma lista cujo endereço é retornado em verlist. O número de versões encontradas é retornado em num

GetVerAntAlt (objckkey, altkey, *verkey*)

Recupera a versão do objeto identificado por objckkey de onde se origina a ramificação identificada por altkey. A chave da versão é retornada em verkey

GetVerAlt (objckkey, altkey, *verkey*)

Recupera a versão que inicia a ramificação identificada por altkey do objeto identificado por objckkey.

FinVerAlt (objckkey, altnam, *altkey, verkey*)

Recupera a versão que inicia a ramificação identificada pelo nome altnam do objeto identificado por objckkey. A identificação da alternativa é retornada em altkey, e a chave da versão em verkey.

GetRef (verckkey, *objlist*)

Recupera todos objetos referenciados pela versão identificada por verckkey. As chaves dos objetos referenciados são organizadas em uma lista cujo endereço é colocado em objlist.

FinRef (verckkey, anokey, mode, *objkey*)

Recupera o objeto referenciado pela versão identificada por verckkey. O objeto é procurado em uma posição especificada por mode em relação ao objeto identificado por anokey.

InsRef (verckkey, objckkey)

Cria uma referência da versão identificada por verckkey, para o objeto objckkey. Faz com que todas as configurações desta versão tenham o valor do atributo estado mudado para *não_resolvida*, quando a versão sofrer *check-in*.

RemRef (verckkey, objckkey)

Remove a referência da versão identificada por verckkey, para o objeto objckkey. Faz com que todas as configurações desta versão mudem para o estado *não_resolvida*, quando a versão sofrer *check-in*.

GetObjVer (verckkey, *verbuffer*, objkey)

Recupera todos os atributos da versão identificada por verckkey no espaço apontado por verbuffer, e recupera também o objeto de desenvolvimento relativo a versão. A chave do objeto é retornada em objkey.

FinObjVer (verckkey, *objkey*)

Recupera o objeto de desenvolvimento relativo a versão identificada por verckkey, e retorna sua chave em objkey.

CheOutVer (verkey, mode, trakey, *verckkey*)

Realiza um *check-out* da versão identificada por verkey, do seu banco de dados original para o banco de dados do usuário. A versão original fica com uma proteção cujo tipo é especificado por mode. Uma cópia da versão é instalada no banco de dados do usuário, na qual ele poderá efetuar operações dependendo do valor de mode. Esta cópia é identificada por verckkey, e é associada a transação trakey.

CheInVer (verckkey)

Faz o *check-in* de uma versão que havia sofrido *check-out* anteriormente. A versão original é atualizada de acordo com as alterações que foram feitas na sua cópia. A cópia é então destruída e a original liberada. Se a versão sofreu alterações nas suas referências, o atributo estado de todas as suas configurações passarão a ter o valor *não_resolvida*.

TraVer (verkey, dbtype)

Faz a tranferência de uma versão identificada por verkey, do banco de dados onde está armazenada para o banco de dados especificado por dbtype da lista de bancos de dados acessíveis pelo usuário. A operação só tem sucesso se dbtype especificar um nível superior ao banco de dados em que a versão está armazenada, e se o objeto de desenvolvimento relativo a versão estiver armazenado no banco de dados destino ou em um de nível superior a este. É possível, que a versão sucessora de verkey, fique armazenada em um banco de dados de nível inferior a este. Além disso, a versão não pode fazer referências a objetos que estejam armazenados em bancos de dados inferiores ao banco de dados destino.

D.3 Configurações

InsConf (verckkey, predckkey, part, *confkey, confnum*)

Insere uma nova configuração da versão identificada por verckkey na partição part do banco de dados do usuário que realiza a operação. Se predckkey é diferente de CV_NCHAVE. a nova configuração é inserida como sucessora da configuração identificada por esta chave, e ela se apresenta inicialmente como uma cópia desta. Caso contrário ela é inserida como sucessora direta da versão. A chave da configuração inserida é retornada em confkey e o seu número em confnum.

InsGloConf (rep, predckkey, part, *num, confkey*)

Cria uma configuração global na partição part do banco de dados do usuário. Se predckkey é diferente de CV_NCHAVE, a nova configuração global é inserida como sucessora da identificada por esta chave, e ela se apresenta inicialmente como uma cópia desta. O número da configuração global é retornado em num, e a chave em confkey.

RemConf (confkey, mode)

Remove a configuração identificada por confkey. Todas as suas entradas também são removidas. Se a configuração tiver sucessoras, estas ficarão no grafo como sucessoras da configuração que antecedia a configuração removida. Se a configuração fizer parte de outra configuração, ou for referenciada por outros elementos através de atributos que tenham como domínio CONFKEY, a operacão falha.

FinConfNum (verckkey, num, *confkey*)

Procura a configuração de número num da versão identificada por verckkey. Se a configuração for encontrada nos bancos de dados aos quais o usuário tem acesso, a sua chave é retornada em confkey.

GetNum (confckkey, *num*)

Dada a chave de uma configuração retorna o seu número em num.

ConfAttSiz (confckkey, attrnam, *num*)

Determina o tamanho do atributo attrnam da configuração identificada por confckkey, e retorna em num.

GetConfAtt (confckkey, attrnam, *buffer*)

Coloca o valor do atributo attrnam da configuração identificada por confckkey no espaço apontado por buffer.

UpdConfAtt (confckkey, attrnam, sizattr, buffer)

Atualiza o valor do atributo attrnam da configuração identificada por confckkey com o valor armazenado no espaço apontado por buffer. Se o atributo tiver o domínio LONG_FIELD o seu tamanho deve ser fornecido em sizattr.

GetConfAts (confckkey, *buffer*)

Le o valor dos atributos da configuração identificada por confckkey no espaço apontado por buffer. Os atributos que tem o domínio LONG_FIELD são ignorados.

UpdConfAts (confckkey, buffer)

Atualiza o valor dos atributos da configuração identificada por confckkey como o valor armazenado no espaço apontado por buffer. Os atributos que tem o domínio LONG_FIELD são ignorados.

VerConfNum (verckkey, *num*)

Retorna em num o número de configurações existentes para a versão identificada por verckkey.

GetVerConf (verckkey, *conflist*, *num*)

Recupera todas as configurações da versão identificada por verckkey, acessíveis ao usuário e organiza as chaves em uma lista, cujo endereço é retornado em conflist. O número de configurações encontradas é retornado em num.

GetConfDes (confckkey, *conflist, num*)

Recupera todas as configurações acessíveis ao usuário que sejam descendentes diretas da configuração identificada por confckkey, e organiza em uma lista retornada em conflist. O número de descendentes encontradas é retornado em num.

GetConfAnt (confckkey, *antkey*)

Recupera a configuração antecessora da configuração identificada por confckkey, e retorna sua chave em antkey.

FinConfSuc (antckkey, anokey, mode, *confkey*)

Recupera a sucessora da configuração identificada por antckkey, que tenha uma posição especificada por mode em relação a configuração identificada por anokey acessível ao usuário.

InsConfAlt (verckkey, confckkey, altname, *altkey*)

Insere um nome altname para alternativa da versão identificada por verckkey, que inicia na configuração identificada por confckkey. A chave da identificação é retornada em altkey.

RemConfAlt (verckkey, confckkey, altkey)

Remove o nome da alternativa da versão identificada por verckkey, iniciada na configuração confckkey e identificada pela chave altkey.

GetConfAlt (verckkey, anokey, mode, *altkey, altname*)

Recupera a identificação de ramificação da versão verckkey, que esteja em uma posição especificada por mode em relação a ramificação identificada por anokey. A chave da ramificação encontrada é retornada em altkey e o nome em altname.

GetConfAltNam (confckkey, anokey, mode, *altkey, altname*)

Recupera a identificação de ramificação que designa a configuração identificada por confckkey, que esteja em uma posição especificada por mode em relação a ramificação anokey. A chave da ramificação encontrada é retornada em altkey e o nome em altname.

GetConfAltDes (verckkey, altkey, *conflist, num*)

Recupera todas as configurações da versão identificada por verckkey, ascessíveis ao usuário que sejam descendentes da ramificação identificada por altkey. A configuração que inicia a ramificação também é recuperada. As chaves das configurações são organizadas em uma lista retornada em conflist. O número de configurações encontradas é retornado em num.

GetConfAntAlt (verckkey, altkey, *confkey*)

Recupera a configuração da versão identificada por verckkey, de onde se origina a ramificação identificada por altkey. A chave da configuração é retornada em confkey.

GetConfAlt (verckkey, altkey, *confkey*)

Recupera a configuração da versão identificada por verckkey que inicia a ramificação identificada por altkey.

FinDirEnt (confckkey, anokey, mode, *entkey, entry*)

Recupera a entrada direta da configuração identificada por confckkey. A entrada recuperada é encontrada em uma posição especificada por mode em relação a entrada direta identificada por anokey. A chave da entrada é retornada em entkey, e as informações detalhadas da entrada são retornadas em entry. Os campos da estrutura entry armazenam as seguintes informações: chave e nome do objeto, chave e número da versão, chave e número da configuração, chave, rótulo e parâmetros da consulta e tipo da entrada. Os campos preenchidos variam de acordo o tipo da entrada.

FinIndEnt (confckkey, anokey, mode, *entkey, entry*)

Recupera a entrada indireta da configuração identificada por confckkey. A entrada recuperada se encontra em uma posição especificada por mode em relação a entrada indireta identificada por anokey. A chave da entrada é retornada em entkey, e as informações detalhadas da entrada são retornadas em entry. Os campos da estrutura entry armazenam as seguintes informações: chave e nome do objeto, chave e número da versão, chave e número da configuração, chave, rótulo e parâmetros da consulta e tipo da entrada. Os campos preenchidos variam de acordo o tipo da entrada.

FinDefEnt (confckkey, *entkey, entry*)

Recupera a entrada default da configuração identificada por confckkey. A chave da entrada é retornada em entkey, e as informações detalhadas da entrada são retornadas em entry. Os campos da estrutura entry armazenam as seguintes informações: chave e nome do objeto, chave e número da versão, chave e número da configuração, chave, rótulo e parâmetros da consulta. Os campos preenchidos variam de acordo o tipo da entrada.

InsDirEnt (confckkey, anokey, mode, objckkey, refckkey, parlist, *entkey*)

Insere uma entrada direta na configuração identificada por confckkey. A entrada será inserida em uma posição especificada por mode em relação a entrada identificada por anokey. Se o tipo de refckkey for uma versão será inserida uma entrada direta parcial e se for uma configuração será inserida uma entrada direta total. Se for uma definição de consulta, será inserida uma entrada direta de consulta na configuração confckkey, que se relaciona ao objeto identificado por objckkey e que tem os parâmetros definidos de acordo com parlist. A chave da entrada direta inserida é retornada em entkey.

InsIndEnt (confckkey, anokey, mode, refckkey, parlist, *entkey*)

Insere uma entrada indireta na configuração identificada por confckkey. A entrada será inserida em uma posição especificada por mode em relação a entrada

identificada por anokey. Se o tipo de refckkey for uma configuração será inserida uma entrada indireta de inclusão e se for uma definição de consulta, será inserida uma entrada indireta de consulta que tem os parâmetros definidos de acordo com parlist. A chave da entrada indireta inserida é retornada em entkey.

InsDefEnt (confckkey, consckkey, parlist, numver, numconf, *entkey*)

Insere a entrada default na configuração identificada por confckkey, se ainda não existir entrada default nesta configuração. Se conskey for diferente de CV_NCHAVE, será inserida uma entrada default de consulta, que tem os parâmetros definidos de acordo com parlist. Caso contrário será inserida uma entrada default de número, com número da versão igual a numver e número da configuração igual a numconf. A chave da entrada inserida é retornada em entkey.

RemEnt (confckkey, entkey)

Remove a entrada entkey da configuração identificada por confckkey. Faz com que a configuração mude para o estado não_resolvida, quando sofrer *check-in*.

GetObjConf (confckkey, *confbuffer, verkey, objkey*)

Recupera todos os atributos da configuração identificada por confckkey no espaço apontado por confbuffer, e recupera também a versão o objeto de desenvolvimento relativo a configuração. A chave do objeto é retornada em objkey, enquanto que a chave da versão é retornada em verkey.

FinVerConf (confckkey, *verkey*)

Recupera a versão relativa a configuração identificada por confckkey, e retorna sua chave em verkey.

CheOutConf (confkey, mode, trakey, *confckkey*)

Realiza um *check-out* da configuração identificada por confkey, do seu banco de dados original para o banco de dados do usuário. A configuração original fica com uma proteção cujo tipo é especificado por mode. Uma cópia da configuração é instalada no banco de dados do usuário, na qual ele poderá efetuar operações dependendo do valor de mode. Esta cópia é identificada por confckkey e é associada a transação trakey.

CheInConf (confckkey)

Faz o *check-in* de uma configuração que havia sofrido *check-out* anteriormente. A configuração original é atualizada de acordo com as alterações que foram feitas na sua cópia. A cópia é então destruída e a original liberada. Se a configuração sofreu alterações nas suas entradas, o seu estado passará para *não_resolvida*.

TraConf (confkey, dbtype)

Faz a tranferência de uma configuração do banco de dados onde está armazenada para o banco de dados especificado por dbtype da lista de bancos de dados acessíveis pelo usuário. A operação só tem sucesso se dbtype especificar um nível

superior ao banco de dados em que a versão está armazenada, e se a versão relativa a configuração estiver armazenada no banco de dados destino ou em um de nível superior a este. É possível, que a configuração sucessora de confkey, fique armazenada em um banco de dados de nível inferior a este. A operação falha se a configuração tiver entradas para versões ou configurações que estejam armazenadas em bancos de dados inferiores ao banco de dados destino.

ConfRes (verckkey, confckkey, actprior, prot, *reskey*)

Resolve a configuração confckkey da versão verkey, determinando uma versão para cada objeto referênciado. Se confckkey for igual a CV_NCHAVE as entradas da configuração global ativa serão usadas. O parâmetro anterior determina com que prioridade devem ser consideradas as versões e configurações ativas, durante a resolução. A operação insere no banco de dados do usuário uma configuração_resolvida. Todas as versões e configurações envolvidas na resolução ficam protegidas, e o usuário tem acesso a elas através das entradas da configuração_resolvida. Quando o usuário especifica proteção para escrita CV_E, as versões e configurações envolvidas, para as quais o usuário não tem a permissão para a atualização, ficaram com proteção CV_LPE. A chave da estrutura que representa a configuração_resolvida é retornada em reskey.

InsRes (reskey, part, *confkey*)

Transforma a configuração resolvida em uma configuração comum, acessível aos outros usuários. A configuração gerada e suas subconfigurações serão colocadas na partição part. Todas as versões e configurações que estavam protegidas devido a resolução de reskey, são liberadas. A configuração gerada e suas subconfigurações têm o atributo estado com o valor *resolvida*.

VerRes (reskey, *ifres*)

Verifica se a configuração tem o valor do atributo estado definido como *resolvida*, e recursivamente se todas as configurações referenciadas também têm este valor para o atributo. Verifica ainda, se nenhuma das versões referenciadas diretamente faz referência a outros objetos. De acordo com o resultado da resolução, retorna TRUE ou FALSE em ifres.

FinVerRes (reskey, *verckkey*)

Recupera a chave que identifica a cópia de *check-out* da versão relativa a configuração resolvida identificada por reskey, em verckkey.

FinResEnt (reskey, anokey, mode, *entkey, entry*)

Recupera a entrada direta da configuração_resolvida identificada por reskey. A entrada direta recuperada é encontrada em uma posição especificada por mode em relação a entrada direta identificada por anokey. A chave da entrada é retornada em entkey, e as informações detalhadas da entrada são retornadas em entry.

CheOutRes (confkey, mode, trakey, *reskey*)

Realiza um *check-out* da configuração identificada por reskey, do seu banco de dados original para o banco de dados do usuário e associa a cópia de *check-out* a transação identificada por trakey. A operação assume que a configuração identificada por reskey, bem como suas subconfigurações, estejam no estado *resolvida*. E cria uma estrutura que representa a configuração_resolvida, e faz a proteção de todas as versões e configurações envolvidas. O tipo da proteção é especificado por mode. Esta estrutura é identificada por reskey. Quando o usuário especificar a proteção CV_E, e ele não tiver permissão para atualização em alguma versão ou configuração, esta ficará com a proteção CV_LPE. Através da estrutura formada o usuário poderá ler as entradas da configuração original, bem como navegar por toda a estrutura da configuração.

CheInRes (reskey)

Faz o *check-in* de uma configuração_resolvida que havia sofrido *check-out* anteriormente. A configuração original é atualizada de acordo com as alterações que foram feitas na sua cópia. A cópia é então destruída e a original liberada. Se a configuração sofreu alterações nas suas entradas, o seu estado passará para *não_resolvida*.

D.4 Consulta

DefCons (label, resul, type, *conskey*)

Insere no banco de dados do usuário, a definição de consulta identificada pelo rótulo label. O parâmetro resul, determina se a efetuação da consulta deve resultar em chaves de objeto, versões ou configurações. Através do parâmetro type, o usuário especifica se a consulta é direta (relacionada a um só objeto) ou indireta (relacionada a diversos objetos). A chave da consulta é retornada em conskey.

InsCla (consckkey, anokey, mode, *clakey*)

Insere uma nova cláusula na definição de consulta consckkey. A cláusula é inserida numa posição especificada por mode em relação a outra cláusula anokey. A chave da cláusula é retornada em clakey.

InsPre (consckkey, clakey, anokey, mode, attrid, operator, value, size, ispar, *predkey*)

Insere um predicado na cláusula clakey da consulta consckey. O predicado é inserido em uma posição especificada por mode em relação a outro predicado. O nome do atributo ao qual se refere o predicado é passado em attrid. O parâmetro operator define o operador, enquanto value define o valor em que se basear a comparação. O tamanho do valor é passado através do parâmetro size. Se o valor do predicado, for baseado em um parâmetro da consulta, ispar deve ser verdadeiro, caso contrário ispar deve ser falso. A chave do predicado é retornada em predkey.

RemCons (conskey)

Remove a definição de consulta, juntamente com suas cláusulas e predicados. Se alguém estiver utilizando a consulta, ou ela for referenciada em uma configuração, a operação falha.

RemCla (consckkey, clakey)

Remove a cláusula clakey na definição de consulta consckkey.

RemPre (consckkey, clakey, predkey)

Remove o predicado predkey da cláusula clakey da consulta consckey.

FinCons (conslabel, *conskey*)

Dada o rótulo de identificação de uma consulta retorna a sua chave.

FinCla (consckkey, anokey, mode, *clakey*)

Recupera uma cláusula da definição de consulta consckkey. A cláusula se encontra na definição em uma posição especificada por mode em relação a outra cláusula anokey. A chave da cláusula é retornada em clakey.

FinPre (consckkey, clakey, anokey, mode, *attrid, operator, value, size, ispar, predkey*)

Recupera um predicado da cláusula clakey da consulta consckkey. O predicado se encontra em uma posição especificada por mode em relação a outro predicado. O nome do atributo ao qual se refere o predicado é retornado em attrid. O parâmetro operator define o operador, enquanto value define o valor em que se basear a comparação. O tamanho do valor é retornado através do parâmetro size. Se o valor do predicado, for baseado em um parâmetro da consulta, ispar é verdadeiro, caso contrário ispar é falso. A chave do predicado é retornada em predkey.

CheOutCons (conskey, mode, trakey, *consckkey*)

Realiza um *check-out* da consulta identificada por conskey, do seu banco de dados original para o banco de dados do usuário. O tipo da proteção é especificado por mode. A cópia de *check-out* é identificada por consckkey e está associada à transação trakey. O usuário só pode requerer a proteção CV_E, se a consulta estiver armazenada no seu banco de dados.

CheInCons (consckkey)

Faz o *check-in* de uma consulta que havia sofrido *check-out* anteriormente. A consulta original é atualizada de acordo com as alterações que foram feitas na sua cópia. A cópia é então destruída e a original liberada. Se a configuração sofreu alterações posteriores a sua última tradução, ela será traduzida novamente.

TraCons (consckkey)

Faz a tradução da consulta consckkey, definida através de cláusulas e predicados, para o código que vai fazer efetivamente a consulta no banco de dados.

ConsDir (conslabel, consckkey, parlist, objckkey, *reskey*)

Faz uma consulta direta, baseando-se na definição de consulta identificada por conslabel ou consckkey, no objeto objckkey, e nos valores dos parâmetros definidos em parlist. A chave da versão ou configuração resultante é retornada em reskey.

ConsInd (conslabel, consckkey, parlist, *reslist*)

Faz uma consulta indireta, baseando-se na definição de consulta identificada por conslabel ou consckkey e nos valores dos parâmetros definidos em parlist. As chaves dos objetos versões ou configurações resultantes são retornadas na lista reslist.

D.5 Transação

BegTra (tranam, trakey)

Começa a transação tranam. A ela serão associadas todas as chaves de *check-out* que forem realizados fazendo referência a trakey.

EndTra (trakey)

Termina a transação identificada por trakey e faz o *check-in* de todos os elementos que haviam sido associados a ela que ainda não tinham sofrido *check-in*.

D.6 Lista

GetKey (list, anokey, mode, *key*)

Recupera uma chave que esta na lista identificada por list, em uma posição especificada por mode relativa a outra chave identificada por anokey. A chave encontrada é colocada em key.

D.7 Usuário

InsGro (groname, *gronum*)

Cadastra o grupo de usuários de nome groname, cria seu banco de bados e retorna o número do grupo em gronum. O número identifica tanto o grupo como seu banco de dados.

InsUse (usename, gronum, *usenum*)

Cadastra o usuário de nome usename no grupo identificado por gronum, cria seu banco de dados e retorna o número do usuário em usenum. O número identifica tanto o usuário como seu banco de dados.

RemUse (usenum)

Remove o cadastro e o banco de dados do usuário ou grupo identificado por usenum. No caso de grupo, a operação só tem sucesso se não existir nenhum usuário que seja integrante do grupo. Os bancos de dados de nível superior ao do removido, serão atualizados, com a retirada de todas os relacionamentos que os unia.

OpeEnv (usenum)

Inicializa o ambiente e abre todos os bancos de dados acessíveis ao usuário ou grupo identificado usenum. Se usenum for igual a zero, somente o banco de dados geral será aberto.

CloEnv Fecha todos os bancos de dados que haviam sido abertos para o usuário e termina sua sessão de trabalho.