Gerenciamento de Transações: Um Estudo e Uma Proposta

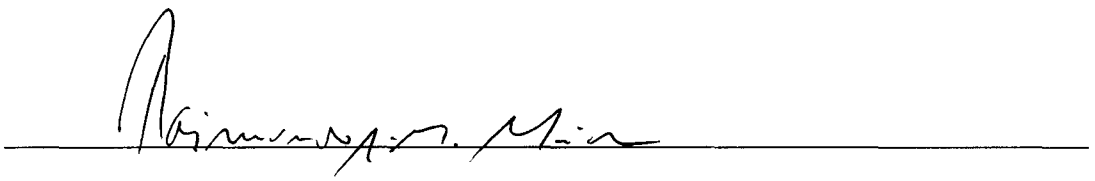
Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. George Marconi de Araújo Lima e aprovada pela Comissão Julgadora.

Campinas, 19 de setembro de 1996.

Prof. Dra. Maria Beatriz Felgar de Toledo
Orientadora

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 16 de setembro de 1996 pela Banca Examinadora composta pelos Professores Doutores



Prof. Dr. Raimundo José de Araújo Lima

Profa. Dra. Claúdia Maria Bauzer de Medeiros

Profa. Dra. Maria Beatriz Felgar de Toledo

Gerenciamento de Transações: Um Estudo e Uma Proposta[1]

George Marconi de Araújo Lima[2]

Instituto de Computação
UNICAMP

Banca Examinadora:

- Maria Beatriz Felgar de Toledo[3] (Orientadora)
- Raimundo José de Araújo Macedo[4]
- Claudia M. Bauzer Medeiros[3]
- Edmundo R. M. Madeira[3] (Suplente)

---

[1]Dissertação apresentada ao Instituto de Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]O autor é Bacharel em Processamento de Dados pelo Instituto de Matemática - Universidade Federal da Bahia.

[3]Professor(a) Dr(a). do Instituto de Computação - UNICAMP.

[4]Pesquisador Dr. do Centro de Processamento de Dados - Universidade Federal da Bahia.

Àqueles sem os quais nada seria possível, meus pais, Grinaldo e Iva.

Agradecimentos

Ao CNPq e à Fapesp pelo apoio financeiro a este trabalho, o qual faz parte do projeto Protem Geotec.

Às minhas irmãs Ivana, Itana e Míriam, que mesmo longe, fizeram-se sentir perto.

À amiga, Kathia Marçal, por sempre acreditar, incondicionalmente, no meu esforço.

À Fabiola Greve, pelo crédito e incentivo, sem os quais essa jornada torna-si-ia mais difícil.

Aos amigos de convívio diário, com os quais muito aprendi e, de certa forma, contribuíram com esse trabalho.

Aos muito mais que amigos, Nuccio Zuquello e Karen Cristina, que deixaram fotografias de um convívio, as quais reduzirão a distância geográfica de km a cm.

Em especial, a Verônica Cadena, pelos momentos únicos e nunca findos.

E finalmente, a Deus pelo pulsar da vida.

"A curiosidade faz modelar a realidade tentando explicar o que nos cerca. Alguns modelos tornam-se realidade e instigam, mais uma vez, a curiosidade."

Resumo

A fim de preservar a consistência dos dados, pode-se utilizar técnicas de controle de concorrência e recuperação de falhas oferecidas por transações. Os primeiros mecanismos foram desenvolvidos para um ambiente em que as aplicações têm curta duração e requerem isolamento. Contudo, com o aparecimento de novas aplicações, tais como CAD/CAM, CASE e SIG, outros requisitos, tais como longa duração e trabalho cooperativo, tiveram que ser considerados.

A presente dissertação apresenta um estudo de vários modelos para gerenciamento de transações e a integração de gerenciamento de transações e versões. Baseados nesse estudo, descrevemos uma solução para o gerenciamento de transações adaptável aos requisitos de vários tipos de aplicações: aplicações de curta duração que requerem gerenciamento tradicional; aplicações de longa duração que requerem gerenciamento mais flexível para permitir a liberação antecipada de recursos e aplicações baseadas em trabalho de grupos que requerem o aninhamento de transações e a transferência de objetos entre áreas de trabalho de transações cooperativas.

O modelo proposto é baseado numa hierarquia de classes que define gerenciadores de transações para cada uma das categorias de aplicações descritas acima. Cada aplicação pode criar instâncias do gerenciador mais adequado para suas características ou, então, estender algum gerenciador existente através do mecanismo de herança.

Abstract

In order to preserve data consistency, concurrency control and failure recovery techniques supported by transactions may be used. The first mechanisms were developed for an environment in which applications have short duration and require isolation. However, with the appearance of new applications such as CAD/CAM, CASE and GIS, other requirements have to be considered.

This dissertation studies several models of transaction management and the integration of transaction and version management. Based on this study, a solution for adapting transaction management to several styles of applications is described: short-duration applications requiring traditional management; long-duration applications requiring flexible transaction management to allow early release of resources and applications based on group work requiring nested transactions and facilities to exchange objects between transaction work areas.

The proposed model is based on a hierarchy of classes defining transaction managers for each of the above categories of applications. Each application can instantiate the manager more suitable for its characteristics or extend an existing manager using the inheritance mechanism.

Conteúdo

[1] Tradução para *two-phase locking*.

[2]Tradução para *two-phase commit protocol.*
[3]*Altruistic Lock.*

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A área de computação, devido ao rápido avanço tecnológico ao qual está submetida, é extremamente dinâmica. O computador, inicialmente concebido para realizar trabalhos simples e repetitivos, ocupa cada vez mais o lugar do ser humano em tarefas mais especializadas. Isso se deve ao surgimento de novas tecnologias e sua conseqüente utilização em novas aplicações. Essas aplicações, por sua vez, impõem novos requisitos devido aos diferentes tipos de dados manipulados e aos diferentes tipos de tarefas que devem ser realizadas.

Inicialmente, os *sistemas de arquivos* forneciam suporte adequado à manipulação dos dados, pois atendiam às necessidades de armazenamento e recuperação das aplicações existentes. As tarefas eram do tipo monousuário, e portanto, não havia preocupação com a concorrência de acesso aos dados. À medida que a tecnologia foi se desenvolvendo, surgiram sistemas de tempo compartilhado que necessitavam de controle da concorrência, recuperação de falhas, manutenção de integridade e segurança, a fim de evitar *problemas de inconsistência* e *acessos indevidos*. Com o objetivo de suprir essas necessidades surgiram os *sistemas de gerenciamento de banco de dados*, que tradicionalmente estavam ligados ao suporte de aplicações comerciais e financeiras.

Posteriormente, associados ao aumento da capacidade de armazenamento de dados, foram sendo incorporadas aos sistemas de gerenciamento de banco de dados, ferramentas integradas para o desenvolvimento de aplicações, tais como *automação de escritório*, *projeto assistido por computador*, *sistema de informações geográficas* etc. Essas aplicações são chamadas *avançadas*.

Aplicações tradicionais e avançadas diferem nos seguintes aspectos:

- quanto ao tipo de dados:
  - os dados manipulados pelas aplicações tradicionais são, geralmente, *alfanuméricos*, possuem tamanho fixo e estrutura plana. Em contrapartida, as aplicações avançadas manipulam dados de estrutura complexa. Podem ser constituídos por vários atributos que, por sua vez, podem ser estruturados recursivamente;
  - determinadas aplicações avançadas, principalmente aquelas ligadas ao desenvolvimento de projetos, tais como CAD/CAM[1] e CASE[2] e SIG[3], necessitam manipular vários estados do mesmo dado, representando as diferentes alternativas e evoluções do projeto. Além disso, o mesmo item de dados pode possuir várias representações *equivalentes* entre si, como por exemplo, o código fonte e objeto do mesmo módulo de programa;
- quanto à duração:
  - as aplicações tradicionais têm curta duração, enquanto que as aplicações avançadas, geralmente, são longas;
- quanto à interação entre usuários:
  - aplicações tradicionais requerem isolamento entre si, enquanto que aplicações avançadas são baseadas no trabalho de grupo, requerendo, portanto, interação entre usuários.

A seguir o problema alvo e a proposta dessa dissertação são apresentados.

1.1 O Problema: Garantia de Consistência

Existem dois problemas que podem causar inconsistência dos dados: um dos problemas está relacionado com a existência de conflitos entre as operações das aplicações que compartilham o mesmo item de dados e o outro está relacionado com a ocorrência de falhas durante a execução das aplicações. Para resolvê-los, agrupa-se um conjunto de operações em unidades, chamadas de *transações*.

[1]Computer Aided Design/Computer Aided Manufacture.
[2]Computer Aided Software Engineering.
[3]Sistema de Informação Geográfica.

Cada transação obedece a certas propriedades que estão relacionadas com o controle de concorrência e recuperação de falhas. Mecanismos para controle de concorrência evitam, através do gerenciamento do escalonamento das operações, que conflitos entre as operações gerem inconsistência nos dados. Já os mecanismos de recuperação de falhas são responsáveis por garantir a consistência dos dados na presença de falhas.

O problema de gerenciar o escalonamento das operações é *np-completo*. Em outras palavras, para um conjunto de operações existe um grande número de possibilidades de escalonamento, dentre os quais, apenas um subconjunto não causaria inconsistência. É computacionalmente inviável avaliar quais os escalonamentos corretos.

Impor critérios restritivos ao escalonamento das operações é uma solução viável. Pode-se assim, garantir que um subconjunto dos escalonamentos corretos seja aceito. Isso penaliza a concorrência do sistema, mas para as aplicações tradicionais essa solução é perfeitamente aceitável.

O mesmo não ocorre para as aplicações avançadas. Essas possuem diferentes requisitos que impossibilitam o uso das mesmas técnicas convencionais. Portanto, são necessários novos modelos de transações.

A alternativa seguida pelos modelos de transações para aplicações avançadas foi a de restringir a aplicabilidade a um conjunto de aplicações, tornando-se desta forma, específicos. Essa especificidade se deve à existência de requisitos conflitantes entre os diversos tipos de aplicação.

1.2 A Proposta: Gerenciamento de Transações Adaptável

A diversidade de modelos de transações específicos para cada classe de aplicação, motivou-nos a elaborar uma solução que fosse abrangente para atender os requisitos de uma variedade de aplicações. Para isso, é definida uma estrutura baseada no paradigma de objetos, que define vários tipos de gerenciadores de transações e cada aplicação deve escolher o gerenciador que melhor se adapte a suas características. Uma aplicação pode também estender algum gerenciador já existente, usando o mecanismo de herança.

1.3 Organização desta Dissertação

Esta dissertação estrutura-se da seguinte forma:

- **Capítulo 2.** Introdução ao problema de gerenciamento da consistência dos dados através de transações. São apresentadas as propriedades de uma execução correta e os critérios adotados para garantir a consistência. Também são descritos, resumidamente, os mecanismos de controle de concorrência e recuperação convencionais.

- **Capítulo 3.** São apresentados alguns dos modelos de transações não convencionais existentes na literatura. Estão classificados em dois grupos: transações para aplicações de longa duração e transações para aplicações cooperativas. São descritos ainda alguns modelos de gerenciamento de versões e configurações, por estarem fortemente ligados às transações cooperativas.

- **Capítulo 4.** Descreve as características das aplicações avançadas mais significativas, tais como *Ambientes de Desenvolvimento* e *Sistemas de Informações Geográficas*. Com base nesse estudo, são indicados os requisitos a serem atendidos por um modelo de transações destinado a suportá-las.

- **Capítulo 5.** Os requisitos apontados no capítulo 4 servem de base para a construção de um modelo de transações. Nesse capítulo, apresentamos esse modelo. Para tanto, foi adotada uma abordagem orientada a objetos por fornecer maior clareza. Juntamente, com o modelo de transações, descrevemos um modelo de versões e configurações. Finalmente, uma comparação entre os modelos descritos no capítulo 3 e o modelo proposto é apresentada.

- **Capítulo 6.** Apresenta as considerações finais, incluindo o resumo de todo o trabalho, suas contribuições e sugestões para extensões.

Capítulo 2

Transações Convencionais

Quando aplicações executam concorrentemente, o acesso a dados compartilhados pode gerar uma série de interferências entre essas aplicações produzindo, então, resultados incorretos. Falhas como queda de processador podem também impedir o término normal de aplicações e a conseqüente violação de restrições de consistência. Dessa forma, são necessários mecanismos de suporte que garantam automaticamente a manutenção de consistência dos dados mesmo quando ocorra compartilhamento e falhas.

Sistemas para processamento de transações fornecem esses mecanismos. O modelo mais simples, apresentado nesse capítulo, suporta transações não hierárquicas[1]. Foi desenvolvido para aplicações bancárias e é utilizado na maioria dos produtos comerciais existentes [GR93].

As transações chamadas convencionais caracterizam-se principalmente pela curta duração e pela existência de barreiras que garantem o isolamento entre transações concorrentes. Tais características permitem que um critério de correção restritivo como a *seriabilidade*[2] seja adotado.

Um sistema para processamento de transações pode ser dividido em dois componentes: *controle de concorrência* e *recuperação de falhas*. O primeiro tem por objetivo evitar algumas intercalações de operações que produzam execuções incorretas. Para tanto, existem vários mecanismos, dentre os quais serão apresentados na seção 2.1 os mecanismos baseados em trancas[3], *timestamps*, grafo de serialização e certificação. O outro componente restaura o estado dos dados após a ocorrência de falhas. No reinicio do sistema, os procedimentos de recuperação são executados a fim de restabelecer a consistência dos dados acessados por transações interrompidas.

[1] Tradução para *flat transactions*.
[2] Tradução para *serializability*.
[3] Tradução para *locks*.

Antes de apresentar os diferentes mecanismos de controle de concorrência e recuperação, é necessário conceituar as propriedades de transações no modelo convencional e definir os tipos corretos de execuções. Isso é feito nas seções 2.1 e 2.2 respectivamente seguido de mecanismos de controle de concorrência e recuperação em 2.3, 2.4 e 2.5.

2.1 Propriedades

Uma transação é um grupo de operações de consulta ou atualização a itens de dados. Esse grupo é iniciado pela operação *Begin* e finalizado pelas operações *Commit* (no caso de término normal) ou *Abort* (no caso de término mal sucedido) [Dat88].Tanto o usuário quanto o sistema podem invocar *Abort*. No caso de término normal de uma transação, todas as atualizações realizadas pela transação são atomicamente efetivadas. Caso contrário, são descartadas.

Transações têm quatro propriedades conhecidas na literatura como ACID [GR93, CP85]:

- **A**tomicidade. A transação é vista como uma unidade de trabalho. Ou todas suas operações são executadas com sucesso ou nenhuma é executada.
- **C**onsistência. O conjunto de itens de dados do sistema obedece algumas assertivas que permanecem válidas após a execução de uma transação.
- **I**solamento. Estados intermediários de uma transação não devem ser observados por outras transações.
- **D**urabilidade. Uma vez confirmadas as atualizações de uma transação, estas devem ser mantidas mesmo se falhas ocorram após a validação da transação[4].

2.2 Históricos Corretos

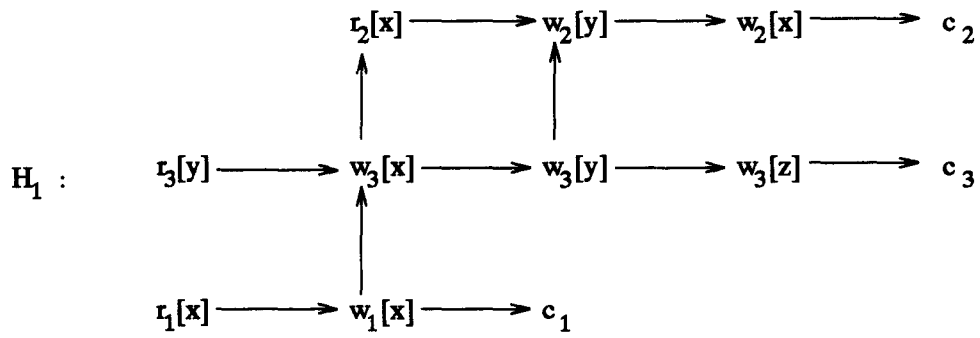
Um *histórico* descreve, para um conjunto de transações, a seqüência em que suas operações foram executadas. Pode ser representado por um grafo orientado no qual as arestas indicam a ordem em que as operações são executadas.

Por exemplo, a figura 2.1 ilustra dois históricos para um conjunto de transações T_1, T_2 e T_3. O histórico H_2 representa a execução seqüencial dessas transações, enquanto que H_1 contém uma execução concorrente. As setas representam a ordem em que as operações

[4]Tradução para *transaction commitment*.

são executadas. Assim, $r_2[x]$ é executada após $w_3[x]$, entretanto, $w_3[z]$ e $w_2[x]$ podem ser executadas concorrentemente.

T_1 : $r_1[x] \longrightarrow w_1[x] \longrightarrow c_1$
T_2 : $r_2[x] \longrightarrow w_2[y] \longrightarrow w_2[x] \longrightarrow c_2$
T_3 : $r_3[y] \longrightarrow w_3[x] \longrightarrow w_3[y] \longrightarrow w_3[z] \longrightarrow c_3$



H_2 : $r_1[x]\ w_1[x]\ c_1\ r_3[y]\ w_3[x]\ w_3[y]\ w_3[z]\ c_3\ r_2[x]\ w_2[y]\ w_2[x]\ c_2$

Figura 2.1: Exemplo de dois históricos com três transações. A leitura é representada pela letra **r**, escrita por **w** e *Commit* por **c**. Para simplificação, as arestas foram retiradas em H_2.

Na execução concorrente de um conjunto de transações, há um entrelaçamento das suas operações que pode causar interferências entre as transações e então tornar o sistema inconsistente. Exemplos de interferências e as propriedades de uma execução correta são apresentados a seguir. A propriedade *seriabilidade* é usada como critério de correção. Devido à possibilidade de ocorrência de falhas, a propriedade *recuperabilidade* também deve ser garantida. Ambas asseguram a consistência dos dados.

2.2.1 Exemplos de Interferências

Suponha a seqüência de operações executadas por T_1 e T_2 mostrada na figura 2.2. Note que o valor do item **x**, atualizado por T_2, foi perdido na atualização do mesmo por T_1. Tal interferência é conhecida como *atualização perdida.*

Outra forma de interferência, chamada *recuperação inconsistente*, ocorre quando uma transação consulta itens de dados e uma segunda os atualiza. A figura 2.3 apresenta uma intercalação de operações que resulta numa interferência desse tipo.

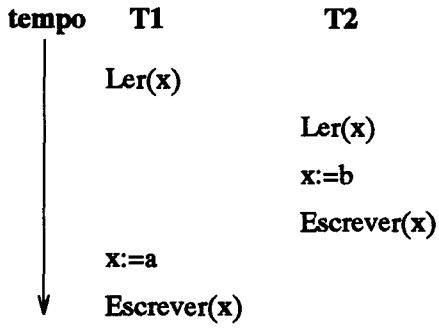


Figura 2.2: Exemplo de interferência: *atualização perdida.*

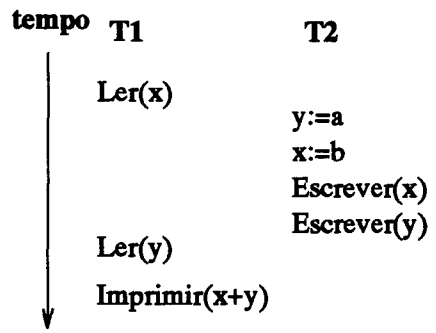


Figura 2.3: Exemplo de interferência: *recuperação inconsistente.*

Ambas as situações representam execuções incorretas. O objetivo do gerenciamento de transações é controlar as intercalações de operações evitando assim interferências entre transações.

2.2.2 Seriabilidade

Numa execução serial, ou seja, sem concorrência, não há interferência entre transações e, portanto, a consistência dos dados é mantida. Porém, o desempenho do sistema é prejudicado. O ideal seria executar transações concorrentemente mas obter o mesmo resultado de uma execução serial. Em outras palavras, deseja-se obter um histórico equivalente a algum histórico *serial.* Esse histórico é chamado *serializável* e representa uma execução *serializável* [BHG87].

Dois históricos são equivalentes [BHG87] se: (a) representam o mesmo conjunto de transações e (b) qualquer par de operações conflitantes aparece na mesma ordem em ambos. Uma operação o_i é dita conflitante com o_j se uma delas é de escrita, ambas operam sobre o mesmo item de dados e são invocadas por transações diferentes. Portanto, para todo par de operações conflitantes o_i e o_j, se o_i precede o_j em H e também H', então H é equivalente a H'.

Dado um histórico qualquer, é possível saber se esse histórico é serializável através da análise do *grafo de serialização* ou *grafo de espera*[5]. Um grafo de espera é um grafo orientado cujos nodos representam transações e as arestas, suas dependências. Uma aresta de uma transação T_1 para uma transação T_2 indica que T_2 espera por algum recurso alocado por T_1. Um grafo de espera sem ciclos representa um histórico serializável. A existência de um ciclo, no entanto, indica que o histórico correspondente não é serializável.

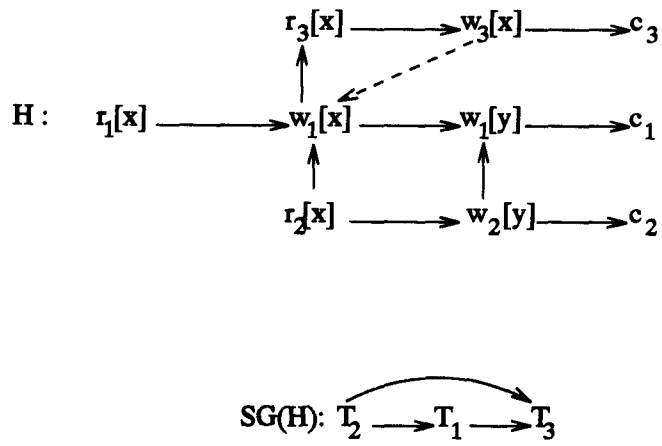


Figura 2.4: Grafo de serialização GS construído a partir do histórico H.

Analisando o histórico H mostrado na figura 2.4, o grafo GS(H) foi obtido da seguinte maneira: $T_2 \rightarrow T_1$, pois $r_2[x] \rightarrow w_1[x]$; $T_2 \rightarrow T_3$, pois $r_2[x] \rightarrow w_3[x]$; e $T_1 \rightarrow T_3$, pois $w_1[x] \rightarrow r_3[x]$. Portanto, H é serializável por ser equivalente à execução serial T_2, T_1, T_3. Se a aresta tracejada de $w_3[x]$ a $w_1[x]$ fosse incluída, o grafo GS(H) passaria a conter um ciclo ($T_3 \rightarrow T_1$ e $T_1 \rightarrow T_3$) e portanto H não seria serializável.

2.2.3 Recuperabilidade

Até agora nada foi mencionado a respeito da ocorrência de falhas e das características que um histórico deve possuir para preservar a consistência dos dados em caso de falha. Simplesmente desfazer as operações de uma transação que falhou pode ocasionar interferência em outras transações que acessaram algum dado atualizado pela primeira. Se isso for permitido, cancelamentos em cascata[6] podem ocorrer.

Considerando a ocorrência de falhas, históricos podem ser classificados em:

- **históricos recuperáveis.** Um histórico é recuperável se cada transação é validada após a validação de todas as transações que lhe fornecem um item de dados para

[5]Tradução para *wait-for graph*.
[6]Tradução para *cascading aborts*.

leitura;

- **históricos que evitam cancelamentos em cascata**. Se cada transação só pode ler valores atualizados por transações validadas ou pela própria transação;
- **históricos rigorosos**. Se itens de dados não podem ser lidos ou escritos até que as transações que os atualizaram terminem.

As interferências resultantes de uma falha são ilustradas na figura 2.5. Nesse exemplo, pode-se notar que no histórico H_1 não há como cancelar T_1, pois T_2 leu um item de dados escrito por T_1 ($w_1[y] \rightarrow r_2[y]$) e já confirmou suas operações (c_2). Qualquer tentativa de desfazer as operações de T_1 violará a propriedade de *durabilidade* de T_2. Já em H_2, o cancelamento de T_1 obrigatoriamente causaria o mesmo com T_2, pois T_2 leu um item de dados após T_1 tê-lo escrito. H_1 é, portanto, um histórico *irrecuperável* e H_2 permite *cancelamentos em cascata*. Ambas as situações são indesejáveis.

Uma alternativa é adotar um *critério rigoroso* de escalonamento. Tal critério restringe o número de históricos corretos. Restrição essa, aceitável para as *transações convencionais*. Os históricos H_3 e H_4 da figura 2.5 são recuperáveis. H_4, por sua vez, é o único *rigoroso* [BHG87]. Note que, com o critério rigoroso, H_3 é excluído, pois T_2 tem acesso a **x** antes de T_1 terminar.

A figura 2.6 ilustra como estão relacionados os diferentes tipos de históricos. O conjunto dos históricos serializáveis é composto pelo conjunto dos seriais e um subconjunto de cada um dos demais.

T_1: $w_1[x]\ w_1[y]\ w_1[z]\ c_1$
T_2: $r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ c_2$

H_1: $w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ c_2\ w_1[z]\ a_1$
H_2: $w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ w_1[z]\ a_1\ c_2$
H_3: $w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ w_1[z]\ c_1\ r_2[y]\ w_2[y]\ c_2$
H_4: $w_1[x]\ w_1[y]\ r_2[u]\ w_1[z]\ c_1\ w_2[x]\ r_2[y]\ w_2[y]\ c_2$

Figura 2.5: Históricos para as transações T_1 e T_2.

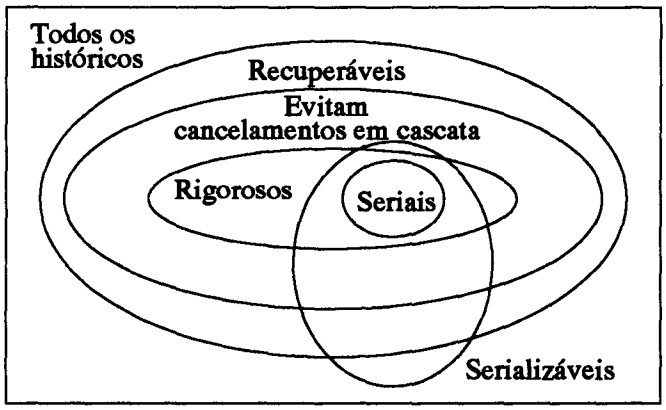


Figura 2.6: Relação entre os diferentes históricos.

2.3 Controle de Concorrência

O mecanismo de controle de concorrência é responsável pela manutenção da *recuperabilidade* e da *seriabilidade* de transações concorrentes. O principal objetivo desses mecanismos é escalonar as operações sobre itens de dados de forma a construir um histórico rigoroso. Quando uma operação é recebida, o mecanismo de controle de concorrência pode:

- atrasá-la, bloqueando a transação requisitante;
- rejeitá-la, cancelando a transação requisitante; ou
- escaloná-la imediatamente.

A opção adotada depende do mecanismo e da possibilidade da ocorrência de conflito.

Fazendo uma classificação dos mecanismos que serão apresentados temos:

$$
\textit{Mecanismos} \begin{cases} \textit{Pessimistas} \begin{cases} \textit{Bloqueantes} \begin{cases} \textit{Bloqueio em} \\ \textit{Duas Fases} \end{cases} \\ \textit{Não Bloqueantes} \begin{cases} \textit{Teste do Grafo} \\ \textit{de Serialização} \\ \textit{Ordenação} \\ \textit{por Timestamp} \end{cases} \end{cases} \\ \textit{Otimistas} \begin{cases} \textit{Certificadores} \end{cases} \end{cases}
$$

Os modelos *pessimistas* assumem que conflitos ocorrem freqüentemente e tentam evitá-los. Nessa abordagem, alguns utilizam *trancas* antes de fazer acesso a um item de dado. São conhecidos como *bloqueantes* por bloquearem uma transação que entrar em conflito com outras transações. Os outros dois métodos pessimistas cancelam transações para resolver conflitos.

Os modelos *otimistas* não se preocupam com conflitos durante a execução da transação. Ao final desta, há uma verificação dos conflitos ocorridos. Caso algum seja detectado, a transação é cancelada.

2.3.1 Bloqueio em Duas Fases[7]

Esse modelo é o mais comumente utilizado nos produtos comerciais existentes. A idéia básica reside na obtenção de trancas sobre os itens de dados antes de usá-los. As trancas são requisitadas em dois modos: compartilhado e exclusivo. Antes de acessar um item de dados para leitura, uma transação deve obter uma tranca em modo compartilhado. Quando um item de dados vai ser atualizado, a tranca deve ser requisitada em modo exclusivo. Dessa forma, várias transações podem acessar um item para leitura enquanto somente uma transação pode atualizar um item.

É necessário alguma política para obter e liberar as trancas a fim de evitar inconsistências. Para isso, transações devem ter duas fases. Na primeira, chamada fase de *crescimento*, as trancas necessárias são obtidas. A segunda, chamada de fase de *encolhimento*, inicia com a liberação da primeira tranca. Essas duas fases estão bem definidas, ou seja, não é possível uma transação adquirir um recurso após ter liberado o primeiro. Na prática, a liberação dos recursos ocorre no término da transação, pois é o único ponto no qual se sabe que esta não precisará de nenhum outro recurso. Além disso, o problema de *cancelamentos em cascata* é evitado, obtendo-se uma *execução rigorosa* [BHG87].

Uma desvantagem desse modelo é permitir a ocorrência de *deadlocks*: transações podem bloquear-se mutuamente permanecendo num ciclo de espera. Por exemplo, suponha que a transação T_1 tenha alocado o item x e requisite uma tranca sobre y, enquanto a transação T_2 já tenha alocado y e requisite uma tranca sobre x. Se isso ocorrer, as transações T_1 e T_2 ficam bloqueadas esperando a liberação de y e x, respectivamente, e nenhuma das duas pode prosseguir. Existem vários algoritmos de *detecção* ou *prevenção* de *deadlocks*, tais como os apresentados em [RSI78, Obe82, HR82].

Cada tranca está associada a um *recurso*, que pode ser um campo (granularidade fina), um arquivo ou até mesmo uma base de dados (granularidade grossa). A política baseada em granularidade grossa oferece baixo custo de gerenciamento devido ao pequeno

[7]Tradução para *two-phase locking*.

número de trancas requisitadas. Entretanto, há pouca concorrência. Por outro lado, com granularidade fina, o aumento da concorrência é acompanhado pelo maior custo na manipulação das trancas.

A melhor solução para o problema parece estar relacionado ao balanceamento entre o custo de gerenciamento e a concorrência. Transações que utilizam muitos itens de dados poderiam manter trancas com granularidade grossa, enquanto aquelas que utilizam poucos requisitam trancas com granularidade fina. Para tanto são necessárias *trancas de intenção* que não permitam ao mesmo item de dados ser utilizado em diferentes níveis. Por exemplo, ao fazer acesso a um registro, uma transação requisita uma tranca de intenção no nível superior (arquivo). Desta maneira, outra transação que requisite trancas sobre todo o arquivo não necessitará verificar se alguns dos seus registros já foram alocados previamente. Para maiores detalhes, veja [BHG87, Dat88].

2.3.2 Ordenação por *Timestamp*

Juntamente com outros modelos de controle de concorrência apresentados a seguir é conhecido como *modelo não bloqueante*, pois cancela transações para evitar inconsistências ao invés de bloqueá-las.

O objetivo desse modelo é ordenar as operações das transações para eliminar execuções incorretas. A chave de ordenação é uma marca de tempo atribuída a cada transação, chamada de *timestamp*. Em cada operação é incluído o *timestamp* que identifica a transação que a submeteu. Baseado nesse *timestamp*, o controle de concorrência segue a seguinte regra para garantir a produção de históricos serializáveis:

- Se $p_i[x]$ e $q_j[x]$ são operações conflitantes das transações T_i e T_j, respectivamente, sobre o item de dados x, então $p_i[x]$ é executada antes de $q_j[x]$ se e somente se $timestamp(T_i) < timestamp(T_j)$.

Essa regra é verificada à medida que as operações são invocadas. As operações podem chegar atrasadas, ou seja, seu *timestamp* pode ser menor que o máximo registrado para um determinado item de dado. Nesse caso, a transação deve ser cancelada. Porém, o escalonamento das operações pode sofrer um atraso a fim de diminuir o número de cancelamentos. O tempo de espera representa um compromisso entre a degradação do desempenho, devido ao atraso, e o número de cancelamentos. Mais detalhes sobre mecanismos baseados em *timestamp* podem ser encontrados em [BHG87].

2.3.3 Teste do Grafo de Serialização

Esse mecanismo, assim como o mecanismo de ordenação por *timestamp*, baseia-se em cancelamento de transações. A idéia básica é manter o grafo de serialização das transações em execução no sistema.

Antes que uma operação $p_i[x]$ invocada pela transação T_i seja escalonada, é adicionado um nodo representando T_i no grafo, caso esse ainda não exista. É também criada uma aresta de T_j a T_i representando o conflito existente entre qualquer operação já escalonada $q_j[x]$ e $p_i[x]$. Podem então ocorrer dois casos:

1. O grafo conter um ciclo. Isso significa que $p_i[x]$ provocará uma execução não serializável [BHG87]. Assim, o escalonador rejeitará $p_i[x]$ e cancelará T_i. Então, o nodo que representa T_i e as arestas incidentes nesse nodo serão eliminados.

2. O grafo ser acíclico. Nesse caso a operação $p_i[x]$ será aceita.

O grafo de serialização usado pelo modelo não necessariamente é completo por não incluir os nodos correspondentes a todas as transações terminadas. Mesmo assim, o espaço ocupado pelas estruturas para a implementação do modelo é considerável e deve ser, portanto, minimizado [BHG87].

2.3.4 Certificação

Esquemas baseados em *certificação* escalonam imediatamente todas as operações recebidas. São também conhecidos como *esquemas otimistas* por assumirem que é pequena a probabilidade de ocorrência de conflitos. Assim, permitem que transações executem até o final sem interrupção. Somente no fim da transação, a existência ou não de conflitos é verificada. Se houve conflitos com outras transações em execução, a transação sendo validada é cancelada. Caso contrário, as atualizações realizadas pela transação são efetivadas.

Uma transação passa, portanto, por três fases:

- **fase de leitura**. Os dados são lidos e atualizados em uma área de trabalho.
- **fase de validação**. No final da transação, verifica-se se as operações realizadas pela transação não conflitam com as operações de outras transações em execução.
- **fase de atualização**. Se a transação passa no teste de validação, suas atualizações são efetivadas.

Quanto ao algoritmo de escalonamento, certificadores podem adotar: bloqueio em duas fases, ordenação por *timestamp* ou teste do grafo de serialização [BHG87]. Também em [BHG87] são descritas extensões dos esquemas baseados em certificadores para ambientes distribuídos.

2.4 Recuperação de Falhas

Essa seção descreve o processo de recuperação, responsável pelo restauração de consistência após a ocorrência de uma falha. O mecanismo de recuperação deve ser capaz de detectar quais transações foram interrompidas por uma falha e corrigir possíveis inconsistências por elas deixadas.

Nas seções seguintes, são descritos as estruturas e procedimentos utilizados para a recuperação de falhas. Estamos tratando aqui de falhas do tipo queda de processador[8]. Falhas de memória secundária e de comunicação são tratadas em [BHG87]. A seguir descreveremos as técnicas baseadas em *logs* e *pontos de recuperação*[9]. Sistemas distribuídos requerem um protocolo especial de validação, descrito na seção 2.4.4.

2.4.1 *Logs*

O histórico das transações é fundamental para o suporte aos procedimentos de recuperação. Ele é armazenado em uma estrutura especial chamada *log* que contém a seqüência das operações realizadas, uma identificação das transações que as invocaram e informações para desfazê-las/refazê-las. Além das operações que fazem acesso aos dados, são também registradas as operações de controle de transações como *Begin*, *Abort* e *Commit*.

Com base nesse histórico, é possível voltar a algum estado consistente depois de uma falha como queda de processador. O procedimento de recuperação percorre o *log* e, usando informações nele contidas, refaz transações validadas[10] e desfaz transações incompletas, antes que qualquer acesso a itens de dados seja feito.

Existem duas modalidades de *log*:

- **físico**. Registra todas as mudanças feitas em cada item de dados através das imagens anteriores e posteriores a cada atualização. Tal procedimento requer muito espaço de armazenamento;

[8]Tradução para *processor crash*.
[9]Tradução para *checkpoints*.
[10]Tradução para *committed transactions*.

- **lógico**. Necessita menos espaço em disco. Ao invés de armazenar a imagem dos itens de dados, registra a operação executada. Por exemplo, registra-se *inserir registro* **R** *na página* **P**, indicando a inclusão de algum registro **R**. Diferentes soluções para o problema de recuperação baseadas em *log* lógico podem ser vistas em [BHG87], pois não é trivial recuperar transações usando tal tipo de estrutura.

2.4.2 Pontos de Recuperação

As operações invocadas por uma transação nem sempre operam sobre memória secundária. Por questão de eficiência, podem estar atuando sobre um *cache* em memória volátil. Porém, periodicamente devem ser transferidas para disco.

A utilização de *cache* melhora o desempenho, mas torna-se problemático saber quais atualizações foram efetivamente realizadas se ocorrer alguma falha que interrompa a execução de transações. Tal problema é resolvido usando-se *pontos de recuperação*. Quando um ponto de recuperação é invocado, as atualizações realizadas em *cache* são transferidas para disco. Após a transferência, deve ser armazenado no *log* um registro de ponto de recuperação, indicando que todas as atualizações realizadas até o momento foram transferidas para o disco.

2.4.3 Reinício após Falha

Foi mencionado anteriormente que o mecanismo de recuperação garante a consistência dos dados depois da ocorrência de falhas. O processo de reiniciar o sistema consiste em percorrer o *log*, verificar o que foi ou não transferido para disco (baseado nos pontos de recuperação) e chamar os procedimentos para *desfazer* ou *refazer* as operações de transações ativas no momento da falha. Note que o reinício deve ser *idempotente*, pois também pode ser interrompido por alguma falha.

Após uma falha, o procedimento de recuperação obtém o endereço do registro de ponto de recuperação mais recente, localiza-o no *log* e passa a processar, a partir daquele ponto até o final, as informações contidas no *log*. Durante a varredura do *log*, é capaz de determinar, não somente as transações a serem desfeitas, mas também as transações a serem refeitas.

A figura 2.7 ilustra a execução de cinco transações, T_1 a T_5 ao longo do tempo. t_c indica o instante de gravação do último ponto de recuperação e t_f, o instante da ocorrência de uma falha. No reinício, após a ocorrência da falha, as transações T_3 e T_5 devem ser desfeitas, pois não realizaram a operação *Commit* antes de t_f. Porém, as transações T_2 e T_4 deverão ser refeitas, pois não há a garantia de que suas atualizações tenham realmente sido

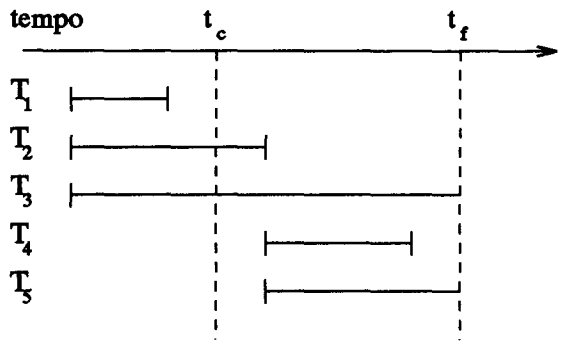


Figura 2.7: Tipos de transações quanto ao momento de ocorrência de falha.

transferidas para disco. A transação T_1, por sua vez, não entra no processo de recuperação, pois suas atualizações foram gravadas em disco no tempo t_c ou anteriormente.

Existem implementações que utilizam diferentes filosofias de recuperação [BHG87]. Há aquelas que nunca desfazem atualizações e, para tanto, estas últimas são gravadas no instante em que a transação invoca a operação *Commit*. Em outros esquemas, as atualizações nunca necessitam ser refeitas, já que todas as operações são gravadas em disco à medida em que são executadas.

2.4.4 Protocolo de Validação em Duas Fases[11]

Num ambiente distribuído, o processamento de transações é mais complexo, pois uma transação pode possuir vários agentes - um em cada nó no qual um item de dados foi utilizado. Tal cenário impõe algumas dificuldades:

- **Falhas.** Qualquer um dos nós pode falhar ou podem ocorrer falhas na comunicação, como mensagens perdidas, atrasadas, corrompidas ou ainda partição de rede.
- **Estado global.** O assincronismo imposto pelo sistema distribuído impossibilita o conhecimento do estado global num determinado instante. Qualquer troca de informações deve ser realizada via rede.
- **Garantia de atomicidade.** Todas as operações de uma transação devem ser confirmadas atomicamente em todos os nós que contêm agentes dessa transação ou todas elas devem ser desfeitas.

Assim, a validação num ambiente distribuído não é uma operação trivial. É necessário um protocolo de comunicação entre os diferentes nós da rede para sincronizar os seus

[11]Tradução para *two-phase commit protocol*.

procedimentos. O protocolo em duas fases é o mais simples e popular. Nesse protocolo, um dos agentes assume o papel de *coordenador*, enquanto os demais são chamados *participantes*. O protocolo possui os seguintes tipos de mensagens:

- **ReqVoto**. Requisição de voto enviada pelo coordenador aos participantes.
- **Voto**. Resposta à requisição de voto. Pode conter SIM, se o participante concordar em efetivar a transação, ou NÃO, caso contrário.
- **Decisão**. Mensagem enviada pelo coordenador aos participantes, indicando qual a decisão tomada: efetivar ou cancelar a transação.

A versão mais simples do protocolo de validação em duas fases pode ser resumida da seguinte forma:

1. O coordenador envia ReqVoto a todos os participantes e espera pelas respectivas respostas.
2. Ao receber ReqVoto, o participante vota SIM indicando que pode efetivar a transação ou NÃO para indicar que decidiu cancelar suas operações.
3. O coordenador coleta os votos de todos os participantes. Caso todos os votos sejam SIM, inclusive o seu, decide por efetivação e envia a decisão final a todos os participantes. Caso contrário, decide por cancelamento, enviando a decisão a todos os participantes que votaram SIM.
4. Cada participante que votou SIM espera pela decisão final do coordenador.

As duas fases do protocolo são: *fase de votação* (itens 1 e 2) e *fase de decisão* (itens 3 e 4).

No protocolo descrito acima, não há preocupação com falhas. Por exemplo, se um nó espera por uma mensagem perdida, poderá esperá-la para sempre. Para resolver tal problema, é estabelecido um tempo máximo de espera, caso a mensagem não chegue nesse tempo algumas ações podem ser tomadas [BHG87]. No caso de o coordenador não receber todos os votos, envia a mensagem de cancelamento para todos os participantes.

Uma outra questão com relação a falhas está relacionada à recuperação de algum participante. Por exemplo, se um participante parar no meio do algoritmo, na recuperação, precisará saber a decisão do coordenador para não ficar inconsistente [BHG87].

Uma desvantagem do protocolo em duas fases é ser um protocolo *bloqueante*. Se o coordenador falhar depois de receber os votos dos participantes e antes de enviar a

decisão final, os participantes que votaram SIM ficarão esperando até que o coordenador se recupere. Alguns protocolos de *terminação* reduzem a probabilidade de bloqueio [BHG87]. Existe uma vasta literatura sobre variações do protocolo de validação, tais como [ML83, RP90].

2.5 Controle de Concorrência e Recuperação Integrados

Essa seção introduz os mecanismos baseados em versões que tratam o controle de concorrência e a recuperação de forma integrada. Para cada item de dados é mantido um histórico das atualizações realizadas sobre este. Uma operação de escrita acrescenta uma nova versão a este histórico e operações de leitura nunca são rejeitadas, pois podem ser realizadas sobre qualquer versão. Apesar de aumentar a concorrência, o mecanismo baseado em versões tem um alto custo, devido ao espaço gasto com o armazenamento e à necessidade de descartar as versões mais antigas. A seguir, apresentaremos uma proposta de implementação.

2.5.1 Exemplo de Implementação

Uma possível implementação desse mecanismo foi proposta por Reed em [Ree79]. Nessa implementação, cada transação tem um único *timestamp* que é associado a cada uma de suas operações. As versões são rotuladas com um *timestamp* de escrita igual ao *timestamp* da transação que as criou e com um *timestamp* de leitura igual ao maior *timestamp* entre os *timestamps* das transações que leram a versão.

Antes do processamento de uma operação de leitura, $r_i[x]$, invocada pela transação T_i, é traduzida para $r_i[x_k]$, onde x_k é a versão de **x** com o maior *timestamp* de escrita que seja menor ou igual ao *timestamp* da transação T_i. O *timestamp* de leitura é atualizado para o valor máximo entre o *timestamp* de leitura e o *timestamp* da transação que invocou a operação de leitura.

Para processar uma operação de escrita invocada pela transação T_i, verifica-se se uma leitura $r_j[x_k]$ já foi processada sobre a versão x_k. Se o *timestamp* de leitura de x_k for maior que o *timestamp* de T_i, então $w_i[x]$ é rejeitada e T_i cancelada. Caso contrário, $w_i[x]$ é traduzida para $w_i[x_i]$ e executada.

A recuperabilidade exige que a operação *Commit* c_i da transação T_i seja atrasada até que as operações de *Commit* de todas as transações que escreveram versões lidas por T_i tenham sido processadas.

Para assegurar atomicidade, cada escrita é realizada em duas etapas. Primeiramente, cria-se uma nova versão do item de dados que foi atualizado. No final da transação, todas as versões criadas pela transação são incorporadas aos históricos dos itens atualizados, se a transação terminar com sucesso. Caso a transação seja cancelada, essas versões são descartadas [Ree79].

Capítulo 3

Transações Não Convencionais

Devido às vantagens oferecidas por sistemas que suportam transações, essa construção foi também adotada em novos ambientes que fornecem suporte a aplicações cooperativas como CAD/CAM[1], CASE[2]. Embora a manutenção de consistência continue relevante para essas aplicações, outras propriedades características do modelo de transações convencionais passaram a ser inadequadas, exigindo uma revisão dos mecanismos tradicionais para controle de concorrência e recuperação de falhas.

Aplicações cooperativas estão geralmente ligadas a desenvolvimento de projetos complexos envolvendo vários projetistas. Esse trabalho de desenvolvimento terá longa duração podendo se estender por semanas.

Outra característica que diferencia aplicações cooperativas das aplicações comerciais está relacionada com a estrutura e representação dos dados. No ambiente cooperativo, é importante a capacidade de modelar objetos do mundo real e de representar a evolução de itens de dados ao longo do projeto. Para isso, é necessário suporte à representação de dados complexos e versionamento.

Em vista dessas características serão analisados nesse capítulo trabalhos que resolvem os seguintes problemas:

- **Suporte a longa duração**. Transações longas podem durar horas ou até semanas. Nesse caso, a probabilidade de conflitos aumenta consideravelmente. Se mecanismos baseados em bloqueios forem adotados, a espera por recursos tende a degradar o desempenho do sistema. Quando conflitos são resolvidos por cancelamento de transações, o número de cancelamentos pode ser grande. Além disso, em caso de falhas, desfazer totalmente uma transação longa é muito caro.

[1]Computer Aided Design/Computer Aided Manufacture.
[2]Computer Aided Software Engineering.

- **Suporte a grupos**. Em ambientes de projeto, existe a necessidade de cooperação entre os usuários. As barreiras de visibilidade impostas pelo modelo tradicional impedem que um usuário fique ciente dos trabalhos desenvolvidos por outros.
- **Suporte a versões e configurações**. Muitas aplicações necessitam de suporte à representação da evolução de itens de dados assim como suporte à ligação entre a versão de um item de dados composto e a versão de cada um dos seus componentes.

Nesse capítulo são descritos os modelos que tratam de aplicações de longa duração na seção 3.1 e modelos que tratam do problema de cooperação entre usuários no desenvolvimento de projetos na seção 3.2. Convém ressaltar que o gerenciamento de versões e configurações não está isolado do modelo de transações. Portanto, na seção 3.3, há um resumo de alguns dos principais conceitos nessa área e apresentação de alguns modelos.

3.1 Suporte a Longa Duração

A probabilidade de ocorrência de falhas e conflitos é proporcional à duração de uma transação. Quando falhas ocorrem, o trabalho de muitas horas tem que ser desfeito. Isso torna a recuperação cara, prejudicando o desempenho do sistema. Quanto à concorrência, transações longas podem acessar muitos recursos com um tempo de retenção longo. Isso aumenta a probabilidade de ocorrência de *deadlocks* quando o controle de concorrência é baseado em trancas.

Uma alternativa para diminuir o tempo de retenção dos itens de dados é liberá-los antes do término da transação. Alguns modelos usam esse artifício, porém exigem cancelamentos em cascata para garantir a consistência [SGMA89, PKH88]. Outros modelos estruturam transações hierarquicamente permitindo que novas formas de controle de concorrência e recuperação sejam implementadas [Mos81, SW90]. Existem ainda aqueles que liberam seus recursos mais cedo, salvam atualizações periodicamente e usam *funções de compensação* como forma de recuperação [GMS87].

3.1.1 Trancas de Salem[3]

O método de trancas de Salem [SGMA89], uma adaptação do bloqueio em duas fases, permite a liberação antecipada de trancas quando itens de dados não são mais necessários a uma transação.

[3] *Altruistic Lock.*

O protocolo de trancas deve obedecer às seguintes condições a fim de manter a seriabilidade:

- Duas transações não mantém trancas sobre o mesmo item de dados simultaneamente, a menos que uma das transações já tenha liberado o item de dados. Nesse caso, diz-se que a segunda transação que o alocou *depende*[4] da primeira que o liberou.
- Se uma transação *depende* de outra ela deve *depender* completamente. Em outras palavras, se T_x aloca o item w que foi liberado por T_y, então qualquer dado que tiver sido alocado por T_x deve ter sido liberado por T_y (se T_y termina, esse requisito pode ser relaxado).

A utilidade desse modelo fica mais clara com o seguinte exemplo: considere uma aplicação bancária que soma R$ 1,00 a cada saldo de conta corrente. Essa computação pode ser considerada de longa duração para um banco de dados com um grande número de registros. Como nessa aplicação cada conta será acessada somente uma vez, pode-se depois da atualização, liberar a conta. Portanto, outras transações que esperam pelo acesso a uma conta corrente podem prosseguir.

Embora exista um ganho com relação à concorrência, pode-se citar também algumas desvantagens:

- Se uma falha ocorrer quando a transação estiver, digamos, processando a penúltima conta corrente, então todas as atualizações realizadas anteriormente à falha estarão perdidas. Em outras palavras, a recuperação ainda mantém o caráter inflexível do modelo convencional.
- Todas as transações no rastro de uma transação que for cancelada também terão que ser canceladas, desencadeando assim cancelamentos em cascata.

3.1.2 Transações Aninhadas

Uma versão mais flexível de transações convencionais foi proposta por [Mos81]. Nesse modelo as transações são organizadas hierarquicamente, permitindo novas formas de recuperação e controle de concorrência.

Conceitualmente uma transação aninhada é uma árvore de transações. Suas subárvores ou são folhas ou são transações aninhadas. Uma transação ancestral é chamada de *pai* e seus descendentes de *filhas*. No modelo original [Mos81, Mos82], os nós intermediários

[4]Tradução para *in the wake*.

da árvore só forneciam o fluxo de controle, pois as únicas transações que realizavam operações sobre os itens de dados eram as folhas. Porém, esse conceito pode ser estendido, fazendo com que qualquer nível possa ter acesso aos dados [GR93].

As seguintes regras descrevem o modelo:

- **Regra Commit.** A operação *Commit* de uma subtransação torna seus resultados acessíveis a seu pai. Uma transação só efetiva os seus resultados (torna-os visíveis às outras transações concorrentes) quando a transação raiz realiza a operação *Commit.*
- **Regra Rollback.** Se uma (sub)transação sofre cancelamento então todas as suas subtransações também sofrerão. Mas, a transação pai não precisa ser cancelada. Isso permite que haja maior flexibilidade no processo de recuperação, pois outras ações podem ser disparadas pela transação pai caso alguma subtransação falhe.

Esse modelo também prevê o uso de trancas sobre os itens de dados. As regras abaixo definem o protocolo de aquisição e liberação de trancas:

- Uma (sub)transação pode adquirir uma tranca sobre um objeto em modo de leitura se todos os possuidores de trancas as obtiveram em modo de leitura, ou se todas as trancas de escrita foram obtidas por seus ancestrais;
- Uma (sub)transação pode adquirir trancas em modo de escrita sobre um objeto se todas as trancas existentes sobre esse objeto pertencem a seus ancestrais;
- Quando uma subtransação realiza a operação *Commit*, então todas as suas trancas são herdadas no mesmo modo (leitura ou escrita) por seu pai;
- Quando uma subtransação é cancelada, todos as suas trancas são descartadas. Se seus ancestrais possuem algumas dessas trancas, elas continuam sendo mantidas no mesmo modo.

O conceito de transações aninhadas apresenta alguns benefícios em relação ao modelo convencional apresentado no capítulo anterior. É possível executar subtransações paralelamente, permitindo um aumento no nível de concorrência do sistema. Se alguma subtransação falhar, a transação pai pode tentar recuperação invocando outras subtransações. Desta forma, pode-se evitar que toda a transação seja desfeita.

A proposta de organizar ações hierarquicamente surgiu com [Dav78], porém o modelo de Moss foi o primeiro que apresentou regras possíveis de serem implementadas. Desta forma, esse modelo acabou inspirando muitos outros, tais como [BSW88, Wal84, FZ89, SZ89, Ska89].

3.1.3 Sagas

Sagas, modelo proposto em [GMS87], fornece suporte a transações longas. Esse modelo, na verdade, é uma evolução de *transações encadeadas* [GR93], pois divide a transação numa cadeia de subtransações. Entretanto, acrescentou-se a idéia de *função de compensação* como forma de recuperação.

Suponha que tenhamos que alterar um campo de todos os registros de uma base de dados. Se essa atualização for feita por várias transações encadeadas (uma iniciando após o final da outra), então é possível liberar os recursos mais cedo e o trabalho perdido devido a falhas ficaria restrito a uma transação. Mas o preço pago é a perda de atomicidade.

Tendo em vista esse problema, foi proposto um novo modelo [GMS87] com uma nova filosofia de recuperação. Ao invés de desfazer as operações, pode-se submeter operações inversas, cuja finalidade é *compensar* os efeitos das operações já realizadas. A tarefa de compensar está associada à execução de *subtransações de compensação.*

Uma Saga é uma transação que pode ser dividida em um conjunto de subtransações encadeadas. A cada subtransação S_i está associada sua compensação CS_i. A execução de uma Saga pode assumir três configurações:

1. S_1,S_2,...,S_i,...,S_{n-1},S_n. Em caso de término normal.
2. S_1,S_2,...,$S_i(falha)$,CS_{i-1},CS_{i-2},...,CS_1. Em caso de falha quando processando S_i.
3. S_1,CP,S_2,S_3(falha),CS_2,S_2,S_3. Em caso de falha quando processando S_3.

No caso 1, todas as subtransações são executadas até o fim com sucesso. No caso 2, houve uma falha durante a execução de S_i. Nesse momento, as operações realizadas por S_i são desfeitas pelo processo convencional (através de cancelamento) e as funções de compensação são executadas a partir de CS_{i-1}. No caso 3, a recuperação é por avanço[5] e exige o uso de pontos de recuperação a fim de salvar os contextos intermediários. Nesse exemplo, a Saga foi dividida em três subtransações e existe um ponto de recuperação (após S_1). Houve uma falha durante a execução de S_3. Na recuperação S_3 é abortada e S_2 é compensada. A Saga pode ser reiniciada a partir do ponto de recuperação.

Desta forma, a atomicidade fica garantida, pois ou todas as operações são executadas ou todas são compensadas. A execução também pode ser paralela, ou seja, em vez de a Saga ser decomposta em uma cadeia serial, as subtransações podem ser submetidas paralelamente [GMS87].

[5]Tradução para *forward recovery.*

Esse modelo trata o problema de recuperação de transações longas. Porém não trata dos efeitos parciais de Sagas que posteriormente são compensados e que são observados por outras Sagas. Requer grande esforço para o desenvolvimento das subtransações de compensação. Além disso, nem todas as operações podem ser compensadas. Ações que interagem com o ambiente externo, tais como disparo de um míssil, saque de uma conta corrente não são passíveis de serem compensadas.

3.1.4 Transações de Pu

O modelo proposto por Pu [PKH88][6] tem a finalidade de suportar aplicações tais como CAD/CAM e desenvolvimento de software. Essas aplicações são conhecidas por possuírem as seguintes características:

- **Duração incerta**. As execuções podem durar de horas a meses.
- **Desenvolvimento indeterminado**. As decisões são tomadas à medida que a transação está em execução. Essa característica requer interação do usuário com a transação.
- **Cooperação**. Interação com outras atividades concorrentes.

Esse modelo permite dividir dinamicamente uma transação em duas outras, através da operação *split_transaction*. Os recursos são particionados entre as duas transações resultantes que podem prosseguir independentemente. Se uma dessas transações resultantes terminar, o seu subconjunto de recursos pode ser liberado.

Com a operação inversa de *split_transaction*, chamada de *join_transaction*, pode-se unir duas transações e então efetivar ou descartar os resultados de ambas atomicamente.

Considere T_A e T_B transações resultantes de uma operação *split_transaction* da transação T; T_A_escrita e T_A_leitura os conjuntos de escrita e leitura da transação T_A respectivamente e, T_B_escrita e T_B_leitura os conjuntos de escrita e leitura da transação T_B respectivamente. T_A precede T_B quando as seguintes condições são atendidas:

1. $T_A\_escrita \cap T_B\_escrita \subseteq T_B\_escrita\_final$;
2. $T_A\_leitura \cap T_B\_escrita = \emptyset$;
3. $T_B\_leitura \cap T_A\_escrita = Conj\_compartilhado$.

[6] *Split-Transactions.*

A propriedade 1 indica que T_A não pode escrever sobre a saída de T_B, mas é permitido a T_B escrever sobre a saída de T_A. Pela propriedade 2, T_A não vê qualquer resultado produzido por T_B. A propriedade 3 indica que T_B pode ver os resultados de T_A produzidos sobre o conjunto de dados em $Conj\_compartilhado$. No caso de T_A ser abortada, então T_B também deverá ser abortada.

Analisando o efeito dessas propriedades, obtém-se as seguintes situações:

1. $T_B\_escrita\_final = \emptyset$ e $Conj\_compartilhado = \emptyset$. Não existem conflitos entre as transações T_A e T_B. Assim, elas podem ser escalonadas em qualquer ordem, pois T_A e T_B são independentes.

2. $Conj\_compartilhado \neq \emptyset$. Nesse caso há conflitos do tipo escrita/leitura entre as transações T_A e T_B. Em outras palavras, T_B lê itens de dados que são escritos por T_A. Portanto, se houver cancelamento de T_A, então T_B também será cancelada.

3. $T_B\_escrita\_final \neq \emptyset$ e $Conj\_compartilhado = \emptyset$. Nesse caso, as operações de escrita de T_B devem ser escalonadas depois das operações de escrita de T_A.

Para entender melhor o modelo suponha o seguinte exemplo: uma transação T é responsável pelo desenvolvimento de dois módulos de programa, A e B. O módulo A já está codificado e testado, enquanto o módulo B ainda não. Duas situações podem ocorrer:

- Por causa de alguma falha durante o desenvolvimento de B, as modificações feitas em A podem ser perdidas;
- Se não houver falhas, o módulo A ficará retido até que T termine.

Usando o modelo de Pu, o usuário poderia dividir a transação T em outras duas, após a codificação e teste do módulo A. Os recursos também ficariam particionados entre as novas transações, o módulo A com T_A e o módulo B com T_B. Assim, após a divisão, T_A pode terminar liberando o módulo A e T_B continua modificando o módulo B.

Esse modelo atende aos requisitos mencionados no início da seção. Suporta transações de duração incerta permitindo que os recursos sejam liberados antecipadamente. Possibilita a re-estruturação dinâmica de transações e o trabalho cooperativo através das operações *split* e *join*.

3.1.5 Ações Multicoloridas

Assim como Sagas [GMS87] e as transações de Pu [PKH88], o modelo proposto em [SW90] permite a liberação de trancas antes do final da transação. Com respeito à recuperação, a propriedade de atomicidade não precisa ser necessariamente mantida.

Esse modelo estende transações aninhadas acrescentando cores às transações e suas trancas. Ele se baseia em três estruturas: *ações seriais*, *aglutinadas* e *independentes*, que podem ser combinadas a fim de fornecer o fluxo de controle requerido pela aplicação.

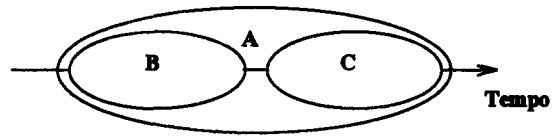


Figura 3.1: Representação de ações seriais.

Dada uma ação A composta por outras ações B e C (figura 3.1), uma *ação serial* permite que as trancas obtidas por B sejam passadas a C através de A. As atualizações que B realizou são mantidas, independentemente do resultado de C. Resumindo, três situações são possíveis:

- nenhum efeito é produzido (B é cancelada);
- os efeitos de B e C são mantidos (As ações B e C terminam normalmente);
- só os efeitos de B são mantidos (B termina normalmente, mas C é cancelada).

Ações aglutinadas permitem que as trancas sejam liberadas mais cedo. Porém, ao contrário do método de Salem [SGMA89] não requer cancelamentos em cascata. Cada ação efetiva suas operações de maneira independente (figura 3.2), liberando, então, algumas trancas. Com relação a ações seriais, existe aumento de concorrência.

Figura 3.2: Representação de ações aglutinadas.

A terceira estrutura, ilustrada na figura 3.3, chamada *ação independente*, oferece uma grande flexibilidade na modelagem de transações. Permite que uma ação invoque outra terminando independentemente da ação invocada. Pode ser síncrona, como ilustrado na figura 3.3 a), ou assíncrona, como representado em 3.3 b). Na primeira versão, A espera

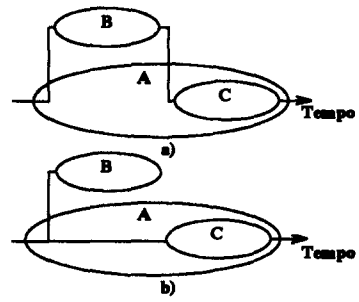


Figura 3.3: Representação de ações independentes.

pelo término de B para iniciar C. Já na versão assíncrona, A prossegue normalmente após o início de B.

Para implementar esse modelo são necessários três tipos de trancas: tranca de *escrita*, tranca de *leitura* e tranca de *leitura exclusiva*. Essa última é usada para se ter acesso exclusivo para leitura.

As estruturas descritas podem ser implementadas associando-se o atributo cor a cada ação e a cada conjunto de itens de dados. Antes de mostrar a implementação de uma das estruturas apresentadas, seguem abaixo as regras de aquisição de trancas propostas por [SW90]:

- A ação somente pode manter e adquirir trancas em algumas de suas cores;
- Uma ação colorida pode adquirir uma tranca sobre um objeto em modo de escrita usando uma de suas cores α, se todos os que possuem trancas (de todas as cores e modos) sobre esse objeto são seus ancestrais e todas as trancas de escrita mantidas sobre o objeto são de cor α. Isso significa que se um ancestral de uma determinada ação colorida tem uma tranca de escrita de cor α sobre um objeto, então essa ação somente pode adquirir uma tranca de escrita de cor α.
- Uma ação colorida pode adquirir uma tranca sobre um objeto em modo de leitura usando uma de suas cores se todas as trancas sobre esse objeto são para leitura, ou se todos os possuidores de trancas de escrita ou leitura exclusiva são seus ancestrais;
- Uma ação colorida pode adquirir uma tranca sobre um objeto em modo leitura exclusivo usando uma de suas cores se todos os possuidores de trancas, de todas as cores e modos, sobre o objeto são seus ancestrais;
- Quando uma ação de cores $\alpha_1, \alpha_2, ..., \alpha_n$ realiza a operação *Commit*, suas trancas de cores α_i $(1 \leq i \leq n)$ são herdadas pelo mais próximo ancestrais de cor α_i. Isso significa que os ancestrais retém as trancas no mesmo modo e cores que seus filhos vinham mantendo;

- Quando uma ação colorida é cancelada, todas as trancas mantidas são descartadas. Se qualquer de seus ancestrais mantém a mesma tranca, eles continuam a mantê-la como antes do cancelamento (do mesmo modo e cor).

Basicamente, as regras são semelhantes àquelas apresentadas para transações aninhadas. Apenas, as cores dão maior poder e flexibilidade ao modelo.

Por exemplo, seja a configuração de ações seriais ilustrada na figura 3.4 e R e W os conjuntos de objetos que são respectivamente lidos e escritos pela ação B de cores azul e amarela. Os objetos pertencentes a W são mantidos com trancas do tipo escrita azul e leitura exclusiva amarela. Os objetos em R são adquiridos no modo leitura com as cores azul e amarela. Desta forma, a ação A de cor amarela retém as trancas amarelas de B quando B termina e libera as azuis. A ação C de cor azul pode adquirir trancas azuis de escrita sobre qualquer objeto mantido por A com trancas do tipo leitura exclusiva e trancas azuis de leitura sobre qualquer objeto mantido por A. Note que se as ações e as trancas fossem coloridas com a mesma cor, o comportamento desta configuração seria similar ao modelo *transações aninhadas.*

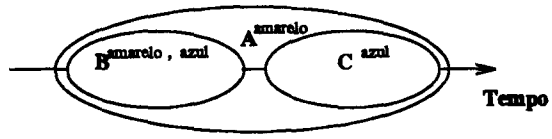


Figura 3.4: Exemplo de coloração para implementação de ações seriais.

A implementação de ações aglutinadas e independentes segue a mesma idéia. Sua coloração pode ser vista em [SW90].

3.2 Suporte a Grupos

A cooperação é um fator predominante nas aplicações relacionadas com desenvolvimento de projetos. Um projeto envolve o trabalho de vários usuários que interagem entre si e com o sistema em prol de um objetivo comum. Esse processo pode ser longo podendo durar dias ou até meses. Geralmente, cada usuário possui tarefas dentro de um determinado projeto. Por exemplo, num projeto de um automóvel a construção do carburador fica a cargo de um projetista e deve estar de acordo com a especificação do consumo de combustível e potência do motor que muitas vezes está sob responsabilidade de outros projetistas. Assim, formam-se grupos de trabalho, cada um desenvolvendo um subprojeto e tendo, por vezes, a necessidade de conhecer partes do projeto desenvolvidas por outros. Existe, portanto, a necessidade de se modelar a hierarquia de grupos de usuário que cooperam entre si. Isso pode ser feito através de transações aninhadas.

As propriedades do modelo convencional de transações não são, portanto, adequadas para o ambiente cooperativo. A seriabilidade é um bom critério de correção para transações curtas. Entretanto, quando há cooperação entre tarefas é necessário que cada usuário fique ciente do trabalho realizado pelos outros. Em outras palavras, transações não podem isolar os usuários como acontecia com transações convencionais. A atomicidade também não é uma propriedade adequada para o ambiente de projetos onde transações são longas. Essa propriedade pode ser relaxada através de pontos de recuperação ou subtransações [BKK85].

Nessa seção serão apresentados alguns modelos encontrados atualmente na literatura. Os modelos de Bancilhon [BKK85], Nodine [NRZ94] e [KSUW85] são baseados em transações aninhadas para representar hierarquias de grupos de projetistas. O quarto modelo [NG92] discute uma nova forma de consistência estabelecida pelo grupo.

3.2.1 Transações para CAD de Bancilhon

Bancilhon [BKK85] apresentou um modelo para dar suporte a aplicações em ambiente CAD que foi implementado no ORION [GK88]. É baseado na construção de uma hierarquia de transações de diversos tipos. Os tipos das transações são os seguintes:

- **Transação de projeto**. Transações de projeto distintas acessam partições disjuntas do banco de dados. Haverá pouco compartilhamento entre duas transações de projeto.

- **Transação de projetista**. Em um ambiente de desenvolvimento, um usuário pode iniciar transações em paralelo através da manipulação de várias janelas de trabalho ao mesmo tempo. Nesse ambiente multitarefa, uma transação de projetista é um conjunto de transações curtas invocadas de várias janelas.

- **Transações cooperantes**. Uma transação cooperante é um conjunto de transações de projetistas. Cada projeto complexo é dividido em várias subtarefas alocadas a grupos de projetistas. Como projetistas trabalham numa subtarefa relativamente pequena, existe maior compartilhamento de dados entre projetistas. Cada transação curta de um projetista só espera por alguma outra transação curta de outro projetista.

- **Transação cliente/subcontratado**. Num projeto complexo, algumas tarefas são subcontratadas. O cliente fornece a especificação das tarefas que serão realizadas pelo contratado e limita o acesso aos dados, fornecendo só o necessário. Esse tipo de transação é na verdade uma transação cooperante. Ela só existe para dar suporte a contratação de tarefas que são vistas pelo cliente como transações atômicas.

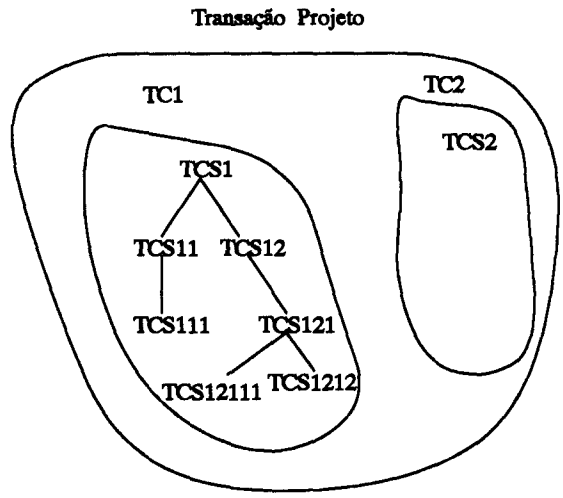


Figura 3.5: Hierarquia de transações de um projeto CAD.

A combinação dos tipos fornece a estrutura do modelo, como mostra a figura 3.5. Resumindo, um ambiente de projeto consiste de:

- um conjunto de transações de projeto, onde uma transação de projeto é
- um conjunto de transações cooperantes, onde uma transação cooperante é
- uma hierarquia de transações cliente/subcontratado, onde cada uma delas é
- um conjunto de transações de curta duração, onde cada uma delas é
- uma seqüência de operações sobre o banco de dados e cada operação é
- uma seqüência de operações do sistema.

Cada um dos níveis da hierarquia possui diferentes requisitos quanto à concorrência:

- Um controle de concorrência baseado em bloqueio em duas fases de longa duração[7] é aplicado ao nível de operação de banco de dados. Assim, a seriabilidade é mantida entre as transações de projeto. No caso de haver conflitos entre transações de projeto, a transação que requisitou a tranca conflitante não deve esperar. Ou ela deve ser notificada ou a transação curta que requisitou a tranca deve ser cancelada.
- As transações cooperantes requerem seriabilidade das transações de curta duração que as constituem. Isso é feito através de controle de concorrência baseado em bloqueio em duas fases[8].

[7]As trancas de longas duração são mantidas em *logs* para possibilitar a recuperação de transações cooperantes.

[8]Nesse caso, as trancas são mantidas em memória dinâmica.

- A hierarquia cliente/subcontratado requer um esquema de concorrência multinível que é implementado no nível de operações de banco de dados. Bancilhon [BKK85] descreve um algoritmo baseado num protocolo de *checkout* em duas fases que utiliza as diferentes áreas do banco de dados para armazenar os dados. O banco de dados é dividido em três tipos de áreas: *espaço do cliente*, *espaço privado* e *espaço do subcontratado*. O espaço privado de uma transação não é compartilhado com outras. Ela pode ler e atualizar os dados nesse local livremente. O espaço do subcontratado é onde uma transação coloca seus dados privados para que suas subtransações possam utilizá-los. Esse espaço é único por transação. O espaço do cliente é o espaço do subcontratado de um cliente ou o banco de dados público (se não existem clientes). Como uma transação cooperante é uma hierarquia de transações cliente/subcontratado, então mais que uma transação pode compartilhar o mesmo espaço do cliente. Nesse algoritmo, uma operação *checkout* corresponde a uma requisição de trancas e uma operação *checkin* a uma liberação de trancas. Um segundo algoritmo baseado em *timestamp* também é descrito em [BKK85].

- As operações de banco de dados devem ser executadas atomicamente.

3.2.2 Transações Cooperativas de Nodine

No modelo descrito em [NRZ94], uma transação é estruturada hierarquicamente e o critério de correção estabelecido a cada nível conforme a semântica da aplicação. Atribuir aos usuários a definição do critério de correção para substituir o critério da seriabilidade não é uma idéia nova. Em [GM83] o usuário era o responsável por definir tipos de transações e a compatibilidade entre eles.

Nesse modelo, uma hierarquia de transações reflete a organização também hierárquica das tarefas. Os nós internos são *grupos de transações* (*TG - Transaction Group*) e as folhas são *transações cooperativas*. Cada grupo de transações (TG) contém um conjunto de membros cooperando para realizar uma tarefa e especificações de correção que determinam as interações entre seus membros. A figura 3.6 ilustra o projeto de um carro organizado como uma hierarquia de transações.

A figura 3.7 fornece a idéia da função de uma TG. Note que o critério de correção, encapsulado na estrutura, é usado para sincronizar operações requisitadas por seus membros a fim de produzir históricos corretos.

Membros de um grupo acessam e atualizam objetos através de operações. Uma operação é uma ação atômica sobre um objeto e seus efeitos podem ser propagados para a raiz ou desfeitos por um membro. Antes de operar sobre um objeto cada membro obtém, do seu ancestral, uma cópia do mesmo (essa é uma das características que diferencia esse

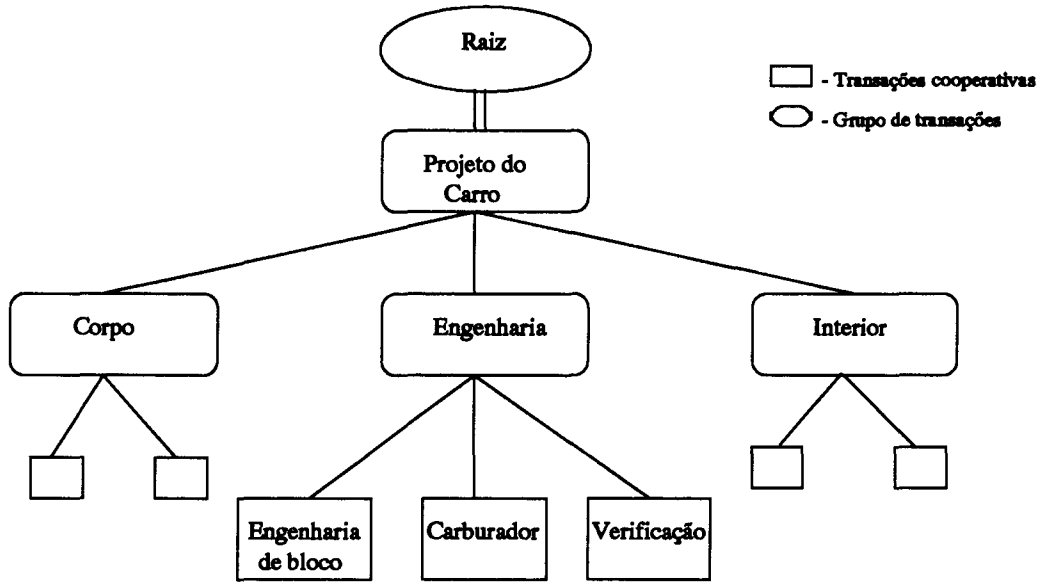


Figura 3.6: Exemplo de uma transação cooperativa.

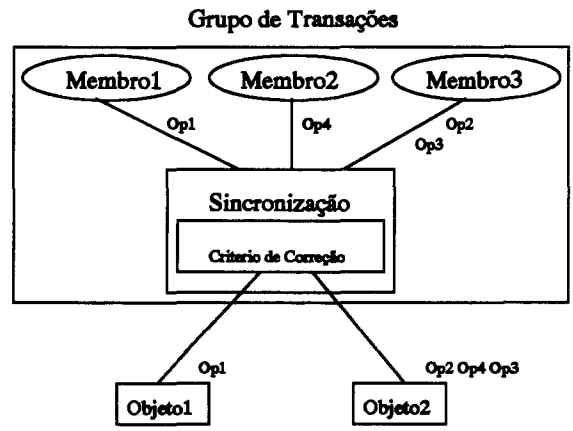


Figura 3.7: Estrutura de uma transação de grupo.

modelo do modelo proposto por [SZ89, Ska89]). Portanto, a raiz da hierarquia possui as versões mais antigas dos objetos que estão sendo manipulados. Quando as operações vão sendo confirmadas[9], as versões são propagadas das folhas para a raiz.

Quando um membro submete uma operação ao seu ancestral, ela pode ser aceita ou rejeitada. Quando uma operação é aceita e executada, as mudanças realizadas serão salvas de forma segura (utilizando pontos de recuperação) pelo TG responsável. Entretanto, isso não garante que elas sejam definitivamente mantidas, pois um membro pode cancelar alguma operação forçando o cancelamento de outras operações dependentes.

O critério de correção utilizado para aceitar ou rejeitar operações é baseado na especificação de *padrões* e *conflitos*. Padrões especificam entrelaçamento de operações que devem ocorrer. Por exemplo, *se o objeto A for modificado, então o objeto B deve ser verificado a fim de confirmar se ainda está de acordo com a especificação.* Os conflitos são seqüências de operações que não devem ocorrer. Por exemplo, *se a última versão de B não foi lida, então o objeto A não poderá ser atualizado.* Os padrões e conflitos restringem o número de possíveis entrelaçamentos de operações e fornecem mecanismos para garantir execuções corretas. Portanto, um histórico é dito correto se está de acordo com todos os padrões e não contém conflitos.

Tanto os padrões quanto os conflitos são definidos através de uma gramática LR(0). Portanto, reconhecer históricos corretos é equivalente a reconhecer palavras geradas por essa gramática. Isso pode ser feito através de um *autômato push down determinístico* que é uma máquina de estados com uma pilha a ela associada [ASU88]. Pode-se analisar as palavras acompanhando o reconhecimento pelo autômato. Desta forma, consegue-se sincronizar as operações requisitadas pelos membros de um grupo de transações.

Um autômato correspondente a um padrão pode aceitar, rejeitar ou ignorar uma dada operação. Se uma operação causa uma transição para um estado *morto*, então ela é rejeitada. Por outro lado, se a transição é para um estado final, a operação pode ser aceita. Porém, se não existe nenhuma transição que corresponda a essa operação ela pode ser ignorada, pois possivelmente deve estar sendo tratada por outro autômato.

Análise semelhante pode ser feita para o autômato responsável pelas definições de conflitos. Porém, nesse caso, o reconhecimento indica a rejeição da operação.

Resumindo, um histórico está correto em relação ao conjunto de autômatos de reconhecimento se todas as suas operações foram submetidas em seqüência e todos os autômatos associados com a definição de padrões estão num estado final e as respectivas pilhas vazias e todos os autômatos referentes aos conflitos estão em configuração indicando o não reconhecimento.

[9]Tradução para *committed*.

3.2.3 Consistência para CASE

O modelo proposto em [NG92][10] suporta o desenvolvimento de *software* de forma consistente através de um protocolo de comunicação entre usuários.

A violação da consistência ocorre quando são processadas modificações conflitantes dos objetos. Se mais de um usuário modifica um objeto, o sistema tem que fornecer suporte à difusão das mudanças de maneira coerente, a fim de manter a integridade do banco de dados. As mudanças são difundidas antes que elas sejam realmente efetivadas. Na verdade, divulga-se a *proposta de mudanças.* Assim, os programadores podem revisar a proposta e ter a oportunidade de aprová-la, reprová-la ou modificar o seu conteúdo.

Além do modelo de transações, há um protocolo de comunicação que tem a finalidade tanto de notificar as propostas de mudanças quanto de transmitir as respostas dos usuários. Quando todos os usuários aprovam a proposta é que a mudança é realmente efetivada. Portanto, é necessário que o protocolo de comunicação interaja com o mecanismo de transação.

Existem relações de dependência e responsabilidade que são utilizadas para definir a abrangência das mudanças e identificar quais programadores serão notificados e indagados na divulgação das propostas. Os objetos mantém entre si relações de dependência. Assim, é possível saber quando uma modificação em um objeto provoca uma alteração em outro. As relações de responsabilidade são, na verdade, ligações entre os usuários e os objetos sobre os quais eles são responsáveis.

O modelo de transação baseia-se no conceito de *etapas de evolução*, pois é necessário agrupar as mudanças dos objetos em conjuntos e efetivar as modificações atomicamente. Uma etapa de evolução é, portanto, um conjunto de propostas de mudanças. Assim que todos as propostas de mudanças forem aceitas pelos usuários que estão participando dessa tarefa, a etapa de evolução é confirmada e efetivada.

A violação de consistência pode ocorrer dentro de uma etapa ou entre duas etapas. A consistência dentro de uma etapa de evolução é garantida através do mecanismo descrito acima. Entretanto, duas etapas podem colidir. Nesse caso, o grupo de usuários envolvidos pode: abandonar ou suspender o processamento de uma das etapas; mesclar as duas em apenas uma; ou fazer que apenas um usuário seja responsável pelo objeto comum às etapas conflitantes.

[10]Lazy Consistency.

3.2.4 Modelo de Klahold

Em [KSUW85] é descrito um modelo de transações para ambientes cooperativos. É um modelo hierárquico em dois níveis. O primeiro representa uma *transação de grupo* que possui vários usuários participantes. No segundo nível estão as *transações de usuários.*

Existem três níveis de banco de dados: *público*, onde residem todos os objetos acessados por várias transações de grupo; *de grupo*, onde residem os objetos alocados pelas transações de grupo e; *privado*, que constitui uma área associada a cada transação de usuário. A figura 3.8 ilustra as hierarquias.

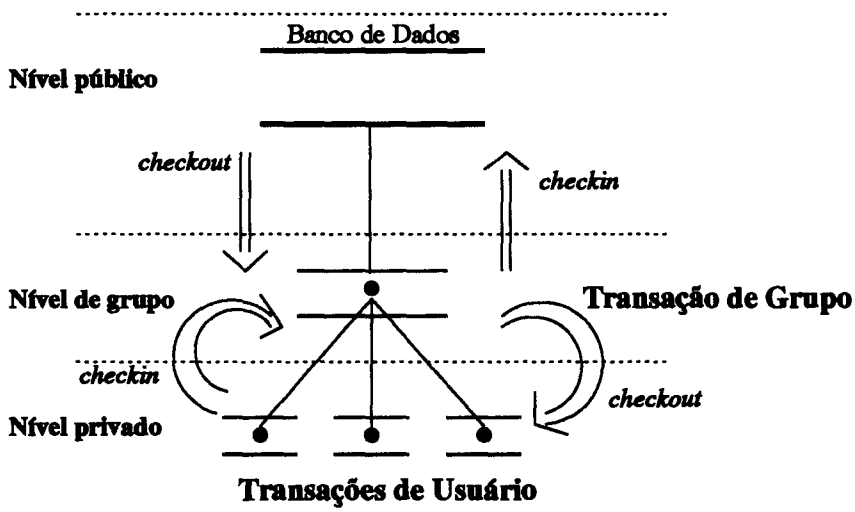


Figura 3.8: Estrutura hierárquica de transações e as áreas de trabalho associadas.

A transferência de objetos entre os níveis é realizada através das operações:

- ***checkout.*** Copia um objeto para um nível descendente, ou seja, do público para o nível de grupo ou desse para o privado;
- ***checkin.*** Transfere a cópia de um objeto para o banco de dados do nível superior.

A estrutura em três níveis de banco de dados, associada com a hierarquia de transações e com as operações *checkin* e *checkout*, garante o isolamento entre áreas de trabalho, permitindo que várias cópias do mesmo objeto sejam manipuladas paralelamente. Além disso, pode-se permitir que a execução das transações de usuário não obedeçam à propriedade de *seriabilidade*, aumentando a flexibilidade quanto à cooperação.

Devido à longa duração das trancas sobre os objetos, o modelo provê operações para *conceder* ou *emprestar* um objeto entre transações de usuários do mesmo grupo. A primeira operação transfere permanentemente o objeto e os direitos sobre os mesmos entre

áreas de trabalho. A operação de empréstimo apenas os transfere temporariamente a outra transação, pois essa, após operar sobre o objeto, deve retorná-lo à transação de origem.

Posteriormente, em [Unl94, US94], esse modelo é estendido. As hierarquias de transações e de banco de dados são generalizadas para vários níveis. Um extenso conjunto de trancas permite definir várias formas de compartilhamento de objetos. Além disso, é permitido que haja múltiplas políticas de controle de concorrência um para cada nível na hierarquia de transações.

3.3 Suporte a Versões e Configurações

Muitos dos ambientes cooperativos como aqueles destinados a auxiliar no desenvolvimento de projetos necessitam de mecanismos de gerenciamento de *versões* e *configurações*.

Existem várias motivações para o uso de versões:

- **Controle de Concorrência**. Com o uso de versões, o número de conflitos diminui. Uma aplicação pode ter acesso a um item para leitura enquanto uma segunda atualiza esse mesmo item, criando uma nova versão. Ou ainda, duas aplicações podem atualizar o mesmo item, criando duas novas versões [BHG87, Ree79, PK84].
- **Recuperação**. Usa-se versões para representar diferentes estados de um item de dados. Em caso de falha, o sistema pode restaurar algum estado consistente representado por um conjunto de versões [BHR80].
- **Evolução**. Durante o desenvolvimento de um projeto, é necessária a representação de várias alternativas de desenvolvimento e sua evolução ao longo do tempo [Kat90, DL88, BKK85, CJ94].

Com respeito a configurações, ambientes de desenvolvimento devem suportar a modelagem de objetos do mundo real construídos hierarquicamente de componentes. As referências para objetos componentes podem ser resolvidas tanto estática como dinamicamente. No caso de configuração estática, objetos compostos identificam explicitamente as versões de seus componentes. No caso de configuração dinâmica, as referências são resolvidas à medida que a hierarquia de composição é percorrida.

3.3.1 Modelo de Katz

O modelo proposto em [Kat90] é baseado em objetos de projeto e seus relacionamentos. Os relacionamentos existentes são os seguintes:

- **é-parte-de.** Objetos podem ser hierarquicamente construídos de outros objetos componentes (figura 3.9). Esse relacionamento é responsável pela construção da hierarquia de composição: objetos complexos e seus componentes;
- **é-derivado-de.** Durante o projeto, um objeto pode sofrer revisões. O relacionamento *é-derivado-de* constrói a hierarquia de derivação, relacionando cada versão com suas versões ancestrais e descendentes (figura 3.10);
- **é-um-tipo-de.** É um relacionamento entre um objeto genérico e suas diferentes versões (figura 3.10);
- **é-equivalente-a.** Um objeto de projeto pode ter uma variedade de representações. Cada uma dessas representações é, por sua vez, um objeto independente. Um objeto de equivalência é introduzido para inter-relacionar as diversas representações de um objeto de projeto.

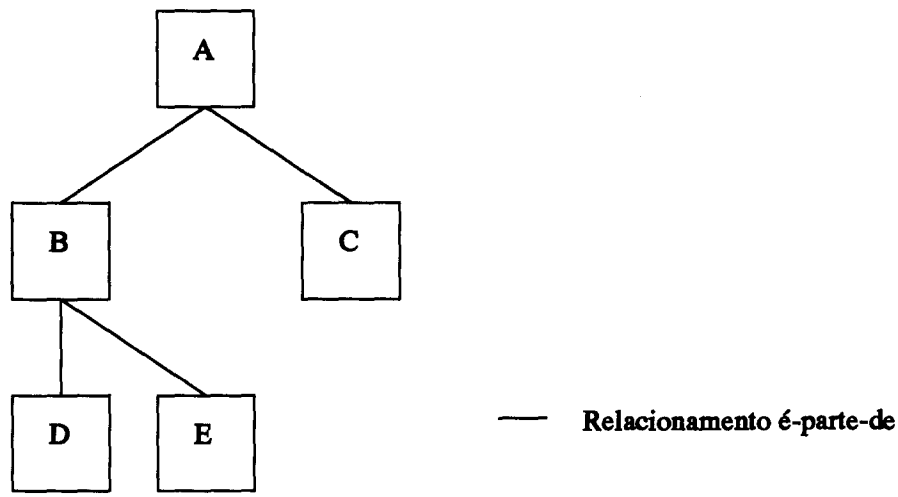


Figura 3.9: Hierarquia de composição de objetos.

As operações para gerenciamento de versões são as seguintes:

- **identificação da versão corrente.** Freqüentemente a versão corrente não é a mais recente. Assim, é útil possuir operações que diferenciem a versão corrente da versão criada mais recentemente;

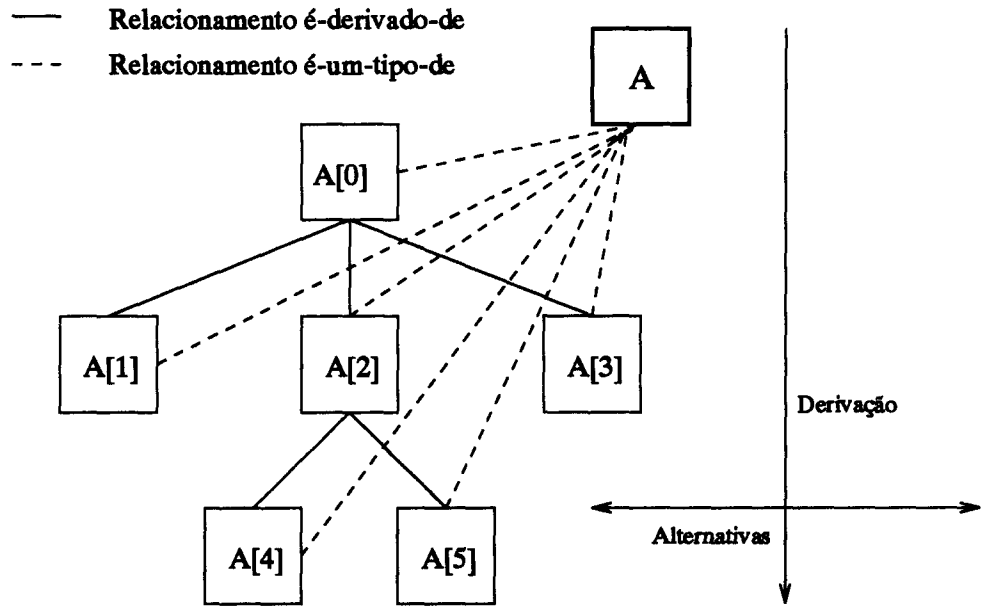


Figura 3.10: Histórico multiversão do objeto A.

- **criação de configurações dinâmicas.** Uma versão de um objeto complexo é composta de versões específicas de seus objetos componentes. As referências para objetos componentes são resolvidas à medida que a hierarquia de composição é percorrida. Históricos de versões de objetos relacionados são distribuídas em camadas. Versões de objetos que devem ser usadas juntas são colocadas na mesma camada e o contexto define a ordem de busca nessas camadas. Por exemplo, na figura 3.11, a versão 0 de A e a versão 1 de B são compatíveis pois estão na mesma camada. Se a ordem de busca em um contexto for nível 3/nível 2/nível 1/nível 0, uma referência para o objeto A é resolvida para versão 2 enquanto para o objeto B resulta na versão 3;

- **transferência de objetos entre espaços de trabalho**. São definidos três tipos de espaço de trabalho: público, semi-público e privado. Os objetos são criados e/ou atualizados na área privada que pertence a um projetista específico. Na área semi-pública, são armazenadas as versões em progresso de usuários de um grupo. Por fim, quando as modificações forem validadas podem ser movidas para a área pública[11]. Essa área pode ser acessada por qualquer usuário;

- **propagação de mudanças**. Para que as atualizações sejam realizadas nos objetos é necessário uma política de *propagação das mudanças*, pois uma alteração em um dos componentes pode requerer mudanças nos objetos que os utilizam. Existem duas questões fundamentais: como limitar o escopo das propagações e como decidir

[11] A transferência de objetos entre as diferentes áreas é realizada por protocolos *checkin/checkout*.

o caminho das propagações no caso de haver bifurcações. Várias estratégias são descritas em [Kat90].

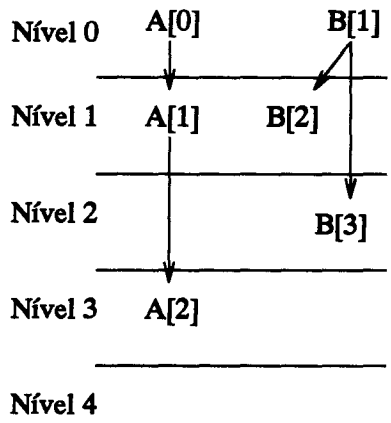


Figura 3.11: Resolução dinâmica de referências.

3.3.2 Modelo de Dittrich e Lorie

O modelo apresentado em [DL88] é baseado nos conceitos de agrupamentos lógicos de versões e de ambientes para resolver referências dinamicamente.

Um ambiente é uma relação binária que associa o identificador do objeto (Oid) à sua versão (Vid), portanto é formado pelo conjunto de pares (Oid,Vid) mais uma opção *default*. Cada par especifica qual versão do objeto será utilizada quando ele é referenciado. Se esse par não existir, a opção *default* é usada.

Um ambiente pode ser criado a qualquer momento. Entretanto, somente um pode estar ativo por vez. No momento em que um ambiente é ativado, o sistema constrói uma visão de execução a partir de uma visão de definição. A visão de definição consiste de:

- **nome do ambiente**;
- **opção *default***. Indica qual versão será usada se o número da versão não é especificado. Ela pode ser, por exemplo, a versão corrente ou a versão congelada[12];
- **conjunto de entradas diretas**. Cada entrada é um par (Oid, Vid) que especifica a versão a ser usada quando o objeto identificado por Oid é referenciado. Pode ser um número explícito, versão corrente ou versão congelada;

[12]Quando o projeto de um objeto atinge um estágio consistente, uma versão do objeto pode ser congelada tornando as atualizações e remoções ilegais.

- **conjunto de entradas indiretas.** Cada entrada é definida pelo par (Oid, A) e indica que a versão de Oid a ser usada é a mesma usada pelo ambiente A, se esse estivesse ativo;
- **conjunto de entradas de inclusão.** Especifica, através do par (A,P), uma prioridade associada ao ambiente A. Desta forma, é estabelecida uma ordem de busca a outros ambientes quando não se resolve a referência no ambiente ativo.

O algoritmo para construir a visão de execução [DL88] percorre os conjuntos de entradas diretas, entradas indiretas e entradas de inclusão, nessa ordem, inserindo para cada entrada na visão de definição nenhuma, uma ou várias entradas na visão de execução. Na visão de execução, todas as entradas são da forma (Oid, Vid) especificando identificador do objeto e o número explícito de versão.

O modelo também oferece um mecanismo para agrupamento lógico de versões organizando versões em uma hierarquia ou mesmo grafos direcionados. Tipos de grupo também podem ser criados além dos tipos pré-definidos objeto de projeto e versão. Por exemplo, é possível, para um objeto de projeto, criar várias alternativas, para cada alternativa várias revisões e agrupar diversas versões para uma cada revisão.

3.3.3 Modelo de Cellary

O trabalho apresentado em [KBGW91] é baseado nos conceitos de constelação e configuração. Uma constelação é uma coleção de objetos e seus componentes. Constitui a unidade de alocação de objetos para projetistas. Uma configuração é uma versão de uma constelação compreendendo uma versão para cada objeto dessa constelação. Constitui a unidade de consistência, i.e., todas as versões de objetos na configuração são mutuamente consistentes.

Uma configuração é isolada de todas as outras configurações da mesma constelação, i.e., a atualização de uma versão não tem efeito em outras configurações. Contudo, configurações de constelações distintas podem compartilhar uma versão de um mesmo objeto. Nesse caso, uma atualização dessa versão aparece em todas essas configurações.

Resumindo os conceitos, temos que:

- cada objeto é um par (Oid, V), onde Oid é o identificador do objeto e V é o conjunto de versões associado ao objeto;
- uma constelação é um par (Cid, S), onde Cid é o identificador da constelação e S é o conjunto dos objetos que ela contém, tal que, se um objeto composto pertence a S, todos os seus componentes também pertencerão;

- uma configuração é uma tripla (CFid, Cid, Ver), onde CFid é o identificador da configuração, Cid é o identificador da constelação e Ver é o conjunto das versões dos objetos de Cid pertencentes à CFid. Nota-se com essa definição que as diferentes configurações de uma constelação formam planos paralelos, todos eles representando diferentes estados dos objetos (suas versões);

- uma versão de um objeto é identificada por (Oid, CFid), portanto pode-se dizer que uma versão é constituída pela tripla (Oid, CFid, valor), ou seja, seu identificador e o valor que ela contém;

- finalmente, um banco de dados multiversão é definido como um conjunto finito de configurações, tal que, cada objeto deve pertencer a, no mínimo, uma constelação e para cada versão de um objeto existe uma configuração que a contém.

Dependendo do contexto em que operam, as transações são classificadas em:

- **o-transações**. Operam sobre versões de objetos. As operações disponíveis são leitura, atualização, criação e remoção. Uma transação desse tipo leva uma configuração de um estado consistente para um novo estado consistente;

- **cf-transações**. Operam sobre configurações. As operações disponíveis são de derivação de uma nova configuração a partir de uma constelação e de destruição de uma configuração;

- **cs-transações**: Operam sobre constelações. As operações disponíveis são de criação/remoção de uma constelação, adição/remoção de objetos de uma constelação.

Capítulo 4

Estudo das Aplicações Avançadas

Existe uma grande variedade de aplicações para as quais é fundamental a preservação de restrições de consistência sobre o conjunto de dados com os quais interagem. A maior parte dessas restrições não precisa ser especificada. Elas são implicitamente mantidas pelo sistema uma vez que programadores conhecem essas restrições e desenvolvem aplicações de forma a preservá-las. Contudo, a manutenção de consistência torna-se mais difícil quando as aplicações executam concorrentemente ou quando falhas interrompem a execução de uma aplicação. O sistema deve, portanto, fornecer suporte ao controle de concorrência e recuperação de falhas facilitando assim o desenvolvimento de aplicações. Transações ficam responsáveis pela manutenção de consistência e programadores não precisam, então, preocupar-se com problemas como controle de concorrência e recuperação de falhas resolvidos automaticamente pelos mecanismos de transação.

Embora o suporte a transações seja importante, não se pode adotar o mesmo modelo de transações para todas as classes de aplicações. Conforme as características das aplicações podemos distinguir duas grandes classes:

- **aplicações tradicionais**. nessa categoria estão as aplicações comerciais/financeiras como, por exemplo, sistemas para banco e reserva de vôos. São caracterizadas por:
  - *curta duração*. As aplicações podem durar, em geral, frações de segundos;
  - *acesso competitivo a dados compartilhados*. O compartilhamento é controlado de forma a evitar interferências entre as aplicações;
  - *estrutura de dados simples*. Os tipos de dados acessados são geralmente registros ou arquivos;
- **aplicações avançadas**. Nessa categoria estão as aplicações para desenvolvimento

de projeto e outras aplicações baseadas em trabalho de grupo. São caracterizadas por:

- *longa duração.* As aplicações podem durar horas ou semanas. Entre os exemplos de aplicações com longa duração estão as aplicações que realizam processamento em lote e as aplicações para desenvolvimento de projeto;
- *acesso cooperativo a dados compartilhados.* Usuários trabalhando cooperativamente precisam estar cientes de alterações realizadas por outros usuários no seu grupo de trabalho. O modo de interação pode ser tanto síncrono como assíncrono. Aplicações de desenvolvimento de projeto como CAD/CAM/CASE são baseadas na forma assíncrona enquanto aplicações como reunião eletrônica necessitam de interação síncrona;
- *estrutura de dados complexa.* As aplicações avançadas geralmente lidam com dados que são agregados de vários componentes.

Esse capítulo descreve as características das aplicações avançadas de nosso interesse na seção 4.1, apontando os requisitos relevantes ao desenvolvimento de um modelo de transações na seção 4.2. As estratégias existentes para se lidar com aplicações avançadas são apresentadas em 4.3.

4.1 Características de Aplicações Avançadas

Essa seção descreve as características de algumas aplicações avançadas. A preocupação principal é enfatizar aquelas que necessitam de suporte à cooperação. Entre as aplicações cooperativas, discutiremos os ambientes de desenvolvimento (seção 4.1.1) e os sistemas de informações geográficas (seção 4.1.2) para os quais o modelo de transações proposto nessa dissertação fornecerá suporte.

4.1.1 Ambientes de Desenvolvimento

O projeto de sistemas complexos envolve um grande número de pessoas trabalhando cooperativamente ao longo das etapas seguintes (não necessariamente nessa ordem):

- análise de requisitos;
- especificação das funções do sistema;
- determinação dos componentes e suas funções específicas;

- implementação dos componentes e integração;
- testes;
- documentação;
- avaliação de desempenho;
- ajustes e extensões.

Para tornar possível a construção de sistemas complexos é necessário o suporte de ferramentas computacionais nas diversas etapas de desenvolvimento. Essas ferramentas podem ser classificadas em três grandes categorias [Kat85]:

- **ferramentas de síntese**. Auxiliam o projetista na criação de objetos de projeto. Exemplos de ferramentas dessa classe são editores de texto, editores de geometria de layout e compiladores;
- **ferramentas de análise**. Auxiliam o projetista na verificação de correção do projeto. Entre as ferramentas de análise estão os simuladores, analisadores topológicos e analisadores de tempo;
- **ferramentas de gerenciamento de informação**. São responsáveis pela criação e manutenção de um banco de dados com a descrição do projeto. Um banco de dados para desenvolvimento de projetos deve ser capaz de organizar as várias representações de um projeto, sua evolução ao longo do tempo, implementações alternativas além de controlar o compartilhamento de dados e a recuperação de falhas.

Devido ao grande número de itens de dados envolvidos em um projeto e à complexidade de inter-relações entre eles, as ferramentas de gerenciamento de informação assumem um papel de destaque no ambiente de desenvolvimento de projetos.

Primeiramente, objetos de projeto são geralmente compostos de objetos mais elementares. Por exemplo, num projeto de circuito integrado, um microprocessador é composto de unidade de controle, unidade de aritmética/lógica e registradores.

Outro aspecto a ser considerado é a representação da evolução do projeto ao longo do tempo. Um projetista desenvolve um ou mais objetos de projeto de forma gradual fazendo melhoramentos a uma implementação já existente. Novas versões de um objeto não devem sobrepor as versões já existentes. Ao invés disso, a evolução de um objeto deve ser mantida como uma seqüência de versões possibilitando ao projetista o retorno a um estágio anterior válido se erros forem detectados na versão atual. Somente as versões que passaram nos testes de validação se tornam visíveis a outros projetistas.

As ferramentas para gerenciamento de informação devem também fornecer suporte à configuração mantendo informação sobre a ligação das versões de objetos complexos e as versões de seus objetos componentes.

Finalmente, um projeto deve ser especificado em várias representações equivalentes, uma para cada ponto de vista. Para apressar o desenvolvimento do produto, essas representações são refinadas simultaneamente. Um sistema de gerenciamento de informação deve, portanto, fornecer mecanismos para organizar as diversas representações de um projeto e as dependências entre elas. Embora seja difícil propagar as mudanças de uma representação para outra automaticamente, a estrutura do banco de dados deve ajudar na identificação das mudanças necessárias. O ambiente de suporte ao desenvolvimento também fornece ferramentas de verificação para cada representação e de teste de equivalência entre as várias representações. O teste de equivalência pode ser realizado através de simulação de cada representação e comparação dos resultados das várias simulações. Por exemplo, num projeto de circuito integrado, a visão de arquitetura lida com o comportamento de partes de projeto; a visão lógica especifica os tipos de componentes e suas interconexões e a geometria de layout especifica formas e representações espaciais das partes do projeto. O ambiente de projeto além de manter as representações para cada visão fornece ferramentas para garantir a consistência entre elas. No ambiente de projeto de software, um sistema é representado usando-se o modelo entidade-relacionamento, o diagrama de fluxo de dados, códigos fonte, códigos objeto e códigos executáveis. Ferramentas como Make analisam as dependências entre as representações de um programa e executam os comandos especificados pelo programador quando essas dependências são violadas.

4.1.2 Sistemas de Informações Geográficas

Um sistema de informações geográficas (SIG) é um sistema computadorizado que tem por finalidade coletar, armazenar e auxiliar na análise de fenômenos nos quais a localização geográfica é um fator importante [Aro89].

As motivações para utilização de computadores em SIGs vêm sobretudo da grande quantidade de dados a ser processada além do desenvolvimento tecnológico proporcionando maior velocidade de processamento e tecnologia de satélite para a coleta de informações.

A seguir apresentamos os componentes de um SIG, as características dos dados armazenados por um SIG, a evolução dos SIGs com relação à capacidade de integração dos dados e uma classificação das aplicações geográficas.

Componentes de um SIG

Um SIG consiste dos quatro componentes abaixo[Aro89]:

- **Entrada de dados.** O componente que converte os dados de sua forma natural (mapas, fotos, imagens de satélite) para um formato no qual possam ser processados pelo SIG.
- **Gerenciamento de dados.** O componente para gerenciamento de dados inclui funções de armazenamento e recuperação. Os métodos usados para implementá-las determinam a eficiência do sistema.
- **Manipulação e análise.** Esse componente usa os atributos espaciais e não espaciais dos dados para responder questões sobre o mundo real. As funções de análise determinam as informações que podem ser geradas pelo SIG.
- **Saída.** Assim como a entrada, a saída das informações pode assumir várias formas como mapas, tabelas, textos ou arquivos. Tanto a funcionalidade quanto a especificação da interface devem ter sido definidas com a participação dos usuários.

Características de Dados Geográficos

Dados geográficos têm características diferentes de dados usados em sistemas de informação para aplicações mais tradicionais. A um dado geográfico estão associados:

- **localização geográfica.** A localização de um objeto é armazenada usando-se algum sistema de coordenadas geográficas. A definição de localização pode ser complexa, pois os fenômenos geográficos tendem a ocorrer em padrões irregulares, tais como linhas sinuosas da costa marítima ou planos de rotas de transporte;
- **atributos não espaciais**;
- **relacionamentos espaciais.** As relações existentes entre aspectos geográficos são geralmente numerosas e nem sempre simples. Por exemplo, é importante conhecer noções relativas à localização, tais como *perto* e *longe*, que são intuitivas para alguém que olha num mapa, mas de difícil implementação em um computador por dependerem do contexto e do tipo do usuário [HFC93]. Por exemplo, questões como: "quais vegetações ficam perto do ponto X?" assumem um significado no escopo de uma cidade e outro em um estado ou país. Além dessas dificuldades, não é possível armazenar informações sobre todas as relações espaciais: algumas são armazenadas explicitamente, outras são calculadas;

- **tempo**. As informações geográficas estão relacionadas a um ponto ou intervalo no tempo e só têm validade nesse período. Assim, a representação do tempo adiciona complexidade à manipulação das informações geográficas ([MJ94]).

Um banco de dados para SIGs deve organizar dados geográficos para armazenamento, recuperação e análise eficientes. Portanto, a escolha do modelo de representação de dados é fundamental. Existem duas abordagens para a representação de dados espaciais: modelo vetorial e modelo raster. No modelo vetorial, os objetos são vistos como um conjunto de pontos, linhas ou polígonos que representam as fronteiras dos objetos. No modelo raster, o espaço é representado por uma matriz de células e a localização dos objetos é definida pelas posições de linha e coluna das células que ocupam.

Evolução dos SIGs

À medida que o desenvolvimento tecnológico avança proporcionando maior poder de processamento, aumenta a integração dos dados geográficos. Pode-se apresentar um perfil dessa evolução ao longo dos estágios abaixo [Aro89]:

1º estágio. SIGs são baseados em procedimentos para manipulação de dados espaciais armazenados em sistemas de arquivos. A maioria dos SIGs é desse tipo;

2º estágio. Um SIG é constituído de um banco de dados comercial (geralmente um banco de dados relacional) para armazenar atributos não espaciais e pacotes externos para lidar com as características espaciais;

3º estágio. Bancos de dados relacionais ou orientados a objeto são estendidos para lidar com características espaciais;

4º estágio. Suporte ao trabalho cooperativo é incorporado aos SIGs [Arm93, Arm94]. Muitas das aplicações em SIGs estão relacionadas com o desenvolvimento de projetos complexos por um grupo de projetistas (por exemplo, na área de telecomunicações) ou envolvem aspectos não rigorosamente definidos requerendo a integração de informações de diversas fontes analisadas por especialistas de diferentes áreas.

Classificação de SIGs

Pode-se classificar as aplicações geográficas em dois grupos principais [Cif95]: *gerenciamento de utilidades* e *controle ambiental*. O primeiro grupo destina-se a manter informações sobre o limites de ruas, avenidas e propriedades para que se possa desenvolver

atividades tais como planejamento viário, controle de cobrança de impostos, serviço de correio, planejamento e manutenção de serviços de água, esgoto, telefone, cabos elétricos, entre outros. O modelo de dados mais adequado a esse grupo é o vetorial. O segundo grupo caracteriza-se por manter informações sobre os recursos naturais disponíveis, a fim de possibilitar a utilização racional de tais recursos e garantir a qualidade do meio ambiente. Esse grupo utiliza o modelo *raster* para mapear o mundo real.

Aplicações em SIGs são geralmente complexas podendo envolver um grupo de usuários possivelmente de diversas áreas. Contudo, a interação entre usuários pode acontecer de formas diferentes dependendo da aplicação.

Muitas das aplicações para gerenciamento de utilidades como planejamento de cabos elétricos ou serviços telefônicos, por exemplo, têm semelhanças com as aplicações para desenvolvimento de projetos descritas anteriormente. Elas compreendem tarefas dentro de um projeto complexo alocadas entre vários usuários. Cada usuário desenvolve sua tarefa de forma gradual incorporando melhoramentos pouco a pouco mas interagindo com outros usuários que também realizam tarefas inter-dependentes dentro do mesmo projeto.

Outras aplicações como controle ambiental, por exemplo, usam técnicas de sobreposição de mapas para produzir soluções. Em muitos casos, várias soluções podem ser propostas, pois muitos aspectos devem ser considerados. Um grupo de usuários trabalhando na solução de um determinado problema necessita, portanto, de ferramentas que auxiliem na avaliação de diversas soluções e na obtenção de consenso entre os elementos do grupo.

Em ambos os casos, as aplicações requerem um espaço compartilhado onde cada usuário pode colocar informações para o grupo e um espaço privado onde um usuário desenvolve sua parte do projeto ou avalia sua solução antes de apresentá-la ao grupo. Atualmente, a visualização de efeitos sobre o espaço de trabalho em tempo real é proibitiva devido à complexidade das operações de análise sobre dados geográficos. Com o desenvolvimento de algoritmos paralelos e a conseqüente melhoria de desempenho interações em tempo real podem se tornar uma necessidade.

As duas classes de aplicações acima diferem na maneira como o problema pode ser subdividido em tarefas e nas atribuições de cada usuário no grupo. No primeiro grupo de aplicações, fica mais fácil dividir o problema em subtarefas menos complexas. Cada uma dessas subtarefas pode ser alocada para um usuário do grupo. No segundo caso, cada usuário busca uma solução para o problema considerando sua área de conhecimento. A complexidade também pode ser evitada dividindo-se a área geográfica a ser estudada em regiões menores e atribuindo cada uma dessas regiões a grupos de usuários distintos.

4.2 Requisitos de Aplicações Avançadas

As características das aplicações avançadas impõem novos requisitos aos mecanismos de controle de concorrência e recuperação de falhas exigindo revisões do modelo tradicional de transações. Os requisitos que devem ser considerados no desenvolvimento de mecanismos para gerenciamento de informação são discutidos a seguir.

4.2.1 Suporte a Diversidade

O gerenciamento de transações deve ser adaptável às características de cada aplicação. Podem existir aplicações que consistem de apenas algumas atualizações e/ou consultas, outras que fazem atualizações sobre uma grande porção do banco de dados e ainda algumas que requerem trabalho de grupo. O gerenciamento de transação deve atender a qualquer tipo de aplicação.

4.2.2 Suporte a Longa Duração

Existem aplicações como, por exemplo, aplicações para desenvolvimento de projeto que podem durar horas ou semanas. Os métodos de controle de concorrência tradicionais são baseados no bloqueio ou cancelamento de transações para resolver conflitos. Os métodos de recuperação tradicionais são também baseados em cancelamentos de transações de forma a garantir a propriedade de atomicidade. Quando transações têm longa duração, é inaceitável fazer transações esperarem trancas por um longo período ou desfazer totalmente uma transação que já dura semanas. Essas estratégias degradariam o desempenho do sistema.

4.2.3 Suporte ao Trabalho de Grupo

Aplicações cooperativas ligadas ao desenvolvimento de projetos complexos envolvem vários projetistas. As tarefas são divididas em subtarefas e atribuídas a um projetista ou grupo de projetistas. Cada projetista pode interagir mais intensivamente com um subconjunto do banco de dados mas pode também precisar interagir com outros projetistas e seus subconjuntos de dados. O trabalho desenvolvido por um projetista pode exigir consultas ou atualizações no trabalho em andamento de outros projetistas.

Para ambientes cooperativos, o modelo de transações não deve, portanto, impor barreiras de visibilidade entre os usuários trabalhando cooperativamente. Métodos de controle

de concorrência baseados no critério de seriabilidade restringiriam a cooperação entre usuários de um grupo. Outros critérios de correção devem ser adotados.

O suporte para interação entre usuários pode ser fornecido nas formas:

- **assíncrona**. A interação entre usuários é realizada através de um espaço de trabalho especial. Cada projetista tem seu espaço privado mas pode trocar informações com outros projetistas através de um espaço do grupo onde itens de dados compartilhados podem ser armazenados e recuperados através de protocolos de *checkin/checkout*.
- **síncrona**. A interação entre usuários é realizada em tempo real. Uma ação de um usuário sobre um contexto compartilhado é imediatamente propagada para todos os membros do grupo. Nesse caso, a granularidade de compartilhamento é mais fina que na forma assíncrona.

4.2.4 Suporte a Critérios Flexíveis de Correção

Critérios de correção baseados em seriabilidade são adequados para aplicações de curta duração e padrão de acesso competitivo. Já aplicações avançadas requerem critérios mais flexíveis que levam em conta sua longa duração e a necessidade de interações entre os membros de um grupo. Novos critérios baseados em semântica dos dados foram propostos [SZ89, Ska89, NRZ94]. Contudo, a dificuldade de especificar restrições de consistência e de detectar automaticamente as violações dessas restrições em tal ambiente provocou o desenvolvimento de novas propostas. Em [NG92], o grupo de usuários participando do desenvolvimento de um sistema define dinamicamente as alterações que podem ser aplicadas sobre o banco de dados.

4.2.5 Suporte a Dados Complexos

Algumas aplicações avançadas precisam modelar objetos do mundo real que são compostos de outros objetos. É necessário, portanto, o suporte à representação de dados complexos.

4.2.6 Suporte a Evolução de Dados

Em ambientes de desenvolvimento de projetos é necessário manter os vários estados de um item de dados durante a evolução do projeto. Refinamentos são acrescentados gradualmente a um projeto gerando novas versões para objetos de projeto já existentes. Dessa forma, uma cadeia de versões pode ser criada para um objeto se cada versão tiver somente uma versão ancestral e gerar somente uma versão descendente ou ainda um grafo pode

representar a evolução de um objeto se uma versão puder ser gerada de várias versões e/ou puder ter várias versões descendentes.

No caso de dados complexos, o sistema deve fornecer suporte à ligação entre a versão de um objeto complexo e a versão de cada um de seus componentes.

4.2.7 Suporte a Múltiplas Representações

Um projeto deve ser especificado em várias representações equivalentes. Cada uma dessas representações expressa um ponto de vista distinto mas deve haver consistência entre elas.

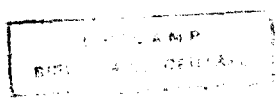
Ambientes para desenvolvimento de projeto devem ser capazes de organizar as diversas representações de um projeto e as dependências entre elas.

4.3 Suporte a Aplicações Avançadas

Existem vários modelos de transações desenvolvidos para essas novas aplicações. Muitos desses modelos foram discutidos no capítulo 3. Eles relaxam as propriedades do modelo convencional flexibilizando os mecanismos de controle de concorrência e recuperação tradicionais.

As estratégias adotadas para lidar com aplicações avançadas são descritas brevemente a seguir:

- **mecanismos para controle de concorrência**:
  - permitem a liberação antecipada de trancas [SGMA89, PKH88, SW90, GMS87]. Em alguns casos [SGMA89, PKH88], é necessário o cancelamento em cascata para manter a consistência dos dados;
  - adotam critério de correção mais flexível para possibilitar o aumento de concorrência e visibilidade entre usuários de um grupo. O critério de correção adotado em [SZ89, Ska89, FZ89, NRZ94] é baseado na especificação de padrões e conflitos. Essa especificação determina as interações possíveis entre os membros de um grupo. Em [NG92], os usuários envolvidos no desenvolvimento de um projeto determinam em tempo de execução as atualizações que podem ser aplicadas através de um protocolo de comunicação;
- **mecanismos para recuperação**. Tornam persistentes partes do trabalho já realizadas através de pontos de recuperação ou transações aninhadas [GMS87, SW90, BKK85].



Além dos aspectos de controle de concorrência e recuperação, existem outros que devem ser considerados dentro do contexto de transações:

- **mecanismos para cooperação:**
  - permitem comunicação entre usuários através de notificações. As notificações podem ser anteriores às atualizações [NG92] ou podem ser enviadas à medida que atualizações são invocadas [HZ87];
  - modelam hierarquias de grupos através de transações aninhadas [BKK85, FZ89, NRZ94]. Transações de grupo gerenciam as interações entre os usuários do grupo;
  - propagam mudanças sobre o contexto compartilhado. Existem duas formas de propagação: síncrona e assíncrona. A forma assíncrona é baseada em três tipos de áreas de trabalho (pública, de grupo e privada) e protocolos de *checkin/checkout* para movimentar objetos de uma área para outra [BKK85, Kat90]. Na forma síncrona, as ações de um usuário são imediatamente propagadas para outros usuários no grupo [EGR91]. O suporte baseado em transações para aplicações síncronas tem sido criticado e substituído pelo conceito de sessão em que usuários podem entrar ou sair a qualquer momento e manipular objetos dentro do contexto compartilhado [EGR91];
- **mecanismos para versionamento e configuração.** Armazenam as diversas versões de um objeto e as ligações entre as versões de um objeto complexo e as versões de seus componentes [CJ90, Kat90, Dit87, CVJ94].

Capítulo 5

Um Modelo Flexível de Gerenciamento de Transações

Suportar diversos tipos de aplicações que muitas vezes apresentam requisitos conflitantes é o objetivo de um gerenciador de transações genérico. A proposta apresentada neste capítulo visa unir três gerenciadores, um para cada tipo de aplicação, num único modelo. Desta forma, pode-se atender os vários requisitos apresentados no capítulo anterior.

A estrutura deste capítulo é apresentada como se segue. Inicialmente, são apresentados, na seção 5.1, os objetivos do trabalho. A seção 5.2 descreve sucintamente o modelo elaborado nesta dissertação, fornecendo uma visão geral. As seções 5.3 e 5.4 apresentam, em detalhes, o gerenciamento de versões e de transações, respectivamente. Por fim, a seção 5.5 avalia o modelo proposto através de uma comparação com os vários modelos estudados.

5.1 Objetivos

A proposta apresentada nesse capítulo integra numa única estrutura o gerenciamento de transações para aplicações convencionais e avançadas. O modelo é baseado numa hierarquia de classes, definindo gerenciadores de transações com diversas características.

Os objetivos deste trabalho são os seguintes:

- atender os requisitos de aplicações, tais como suporte a longa duração, trabalho em grupo, evolução dos dados, dados complexos, analisados no capítulo 4. Contudo, apenas será fornecido suporte à cooperação assíncrona, para a qual um modelo baseado em transações é adequado. Interações síncronas são suportadas por esquemas

diferentes [EGR91];

- fornecer gerenciamento de transações adaptável aos requisitos de cada aplicação;
- integrar o gerenciamento de transações e versões, pois o gerenciamento de versões está muito ligado aos mecanismos de controle de concorrência e recuperação de falhas;
- apresentar o modelo em uma abstração de alto nível. Foi escolhida a notação OMT [RBP+94] por ser bem conhecida e, em especial, por ser baseada em objetos. Esse paradigma facilita a modelagem, oferecendo conceitos como *herança* e *agregação*;
- avaliar o modelo proposto através de uma comparação com outros trabalhos e de uma análise das facilidades oferecidas para atender os requisitos de aplicações, conforme estudo do capítulo anterior.

5.2 Visão Geral

O modelo proposto é baseado numa hierarquia de classes que define gerenciadores de transações para aplicações de curta duração, aplicações de longa duração e aplicações baseadas em trabalho de grupo. Uma determinada aplicação deve instanciar o gerenciador que melhor se adapte aos seus requisitos. A figura 5.1 ilustra esse modelo.

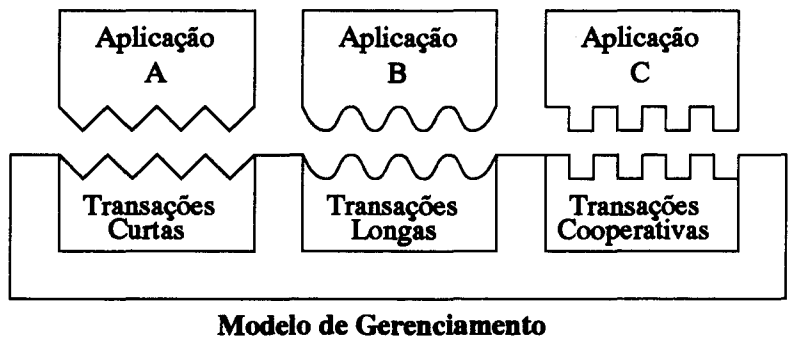


Figura 5.1: Visão geral do modelo proposto.

O modelo trata dos itens seguintes:

- gerenciamento adaptável de transações;
- gerenciamento de versões;
- manutenção de relacionamentos entre versões.

Discutiremos cada um desses itens a seguir.

5.2.1 Gerenciamento Adaptável de Transações

Aplicações tradicionais e avançadas impõem uma variedade de requisitos ao gerenciamento de transações. Todos esses requisitos não podem ser fornecidos por um único estilo de gerenciamento. No modelo proposto, cada aplicação escolhe o gerenciador de transações mais adequado às suas características ou, alternativamente, estende algum gerenciador existente através do mecanismo de herança.

Aplicações curtas utilizam gerenciamento tradicional. O controle de concorrência é baseado em trancas com modo de acesso para ler e derivar versões. Trancas podem ser requisitadas no início da transação ou durante a execução da transação e liberadas no final da transação. É possível, contudo, liberar algumas trancas antes do final da transação. Versões liberadas antes do término ficam disponíveis para leitura apenas e são descartadas se a transação abortar. A recuperação de falhas usa informações sobre o estado da transação e objetos participantes armazenadas em um log. Atualizações requisitadas sobre um objeto são armazenadas em uma nova versão que será efetivada depois da validação da transação.

Aplicações longas necessitam de extensões ao gerenciamento tradicional. No modelo proposto, é possível invocar pontos de recuperação periodicamente e liberar trancas antecipadamente.

Aplicações cooperativas podem ser estruturadas como uma hierarquia de transações. Essa hierarquia modela grupos trabalhando cooperativamente. Os mecanismos para permitir a colaboração entre usuários são baseados em protocolos de *checkin/checkout* e operações para cópia, empréstimo e concessão de versões entre transações.

5.2.2 Gerenciamento de Versões

Um objeto de aplicação para adquirir as propriedades e operações relacionadas com versões deve herdá-las de uma classe especial de suporte a gerenciamento de versões. Essa classe fornece controle de concorrência baseado em trancas e operações para criar, remover e transferir versões entre áreas de trabalho de transações cooperativas.

5.2.3 Manutenção de Relacionamentos

O modelo proposto fornece diversos relacionamentos entre as versões de objetos.

O relacionamento de derivação entre as versões de um objeto representa a história evolutiva desse objeto. No modelo proposto, a história de versões é linear. A versão atual é sempre a última versão efetiva da qual uma nova versão pode ser derivada.

Os relacionamentos de composição são dois: *é-componente-de* e *é-composto-por* (figura 5.2). Eles formam um grafo orientado, chamado de *grafo de composição*, já que uma versão pode ser componente de várias outras.

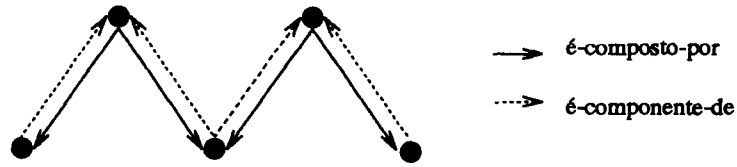


Figura 5.2: Relacionamentos de composição.

Objetos possuem várias facetas por causa de suas várias representações. Por exemplo, o código fonte e objeto de um módulo de programa são objetos distintos, porém equivalentes entre si. Assim, se o código fonte for atualizado e não for gerado outro código objeto referente a essa atualização, eles não estarão consistentes entre si. O relacionamento de equivalência faz as ligações entre versões equivalentes, facilitando o gerenciamento da consistência. No modelo proposto, a manutenção da consistência entre versões deve ser realizada pela própria aplicação.

5.3 Gerenciamento de Versões

O gerenciamento de versões é fornecido por duas classes: *Objeto Genérico* e *Objeto Versão*. Um objeto de aplicação para ser versionável deve herdar as propriedades da classe Objeto Versão. Uma instância da classe Objeto Genérico organiza a hierarquia de derivação para um dado objeto de aplicação. Além disso, um objeto de aplicação pode ser composto de outros objetos, ser equivalente a outros objetos e participar de várias transações conforme diagrama da figura 5.3. A entidade Transação nessa figura representa os vários tipos de gerenciadores de transações discutidos na seção 5.4.

Essas duas classes fornecem às aplicações:

- **identificação de objetos e suas versões.** Quando objetos genéricos e novas versões de objetos de aplicação são criados recebem um identificador único;
- **controle de concorrência.** A unidade de trancamento é a versão. São dois os modos de acesso: leitura e derivação. Modos de intenção são acrescentados para permitir o trancamento de objetos complexos. Além disso, as trancas adquiriram uma nova dimensão para permitir cooperação entre transações através de cópia, empréstimo ou concessão de versões;

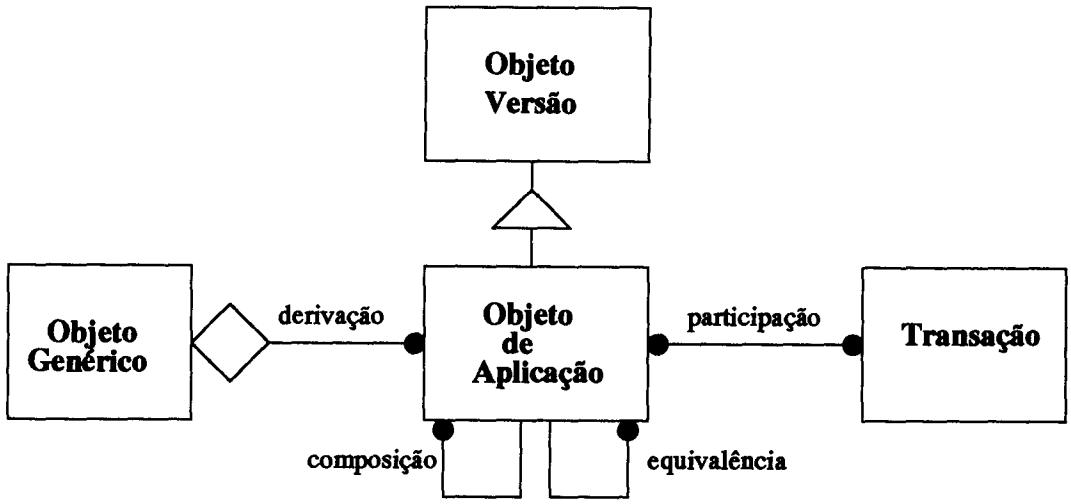


Figura 5.3: Gerenciamento de Versões.

- **resolução de referências dinâmicas**. Aplicações podem usar referências estáticas ou dinâmicas para objetos. Uma referência estática aponta para uma versão específica de um objeto enquanto uma referência dinâmica aponta para o componente Objeto Genérico do objeto. Esse componente armazena a versão atual do objeto para a qual a referência dinâmica é resolvida;

- **criação e manipulação do objeto genérico**. A classe Objeto Genérico fornece operações para criação/remoção de instâncias e cópia/transferência de instâncias de/para áreas de trabalho de transações;

- **criação e manipulação de versões**. A classe Objeto Versão fornece operações para criação/remoção de instâncias da classe e cópia/transferência de instâncias de/para áreas de trabalho de transações;

- **atualização da hierarquia de derivação**. A história de versões é linear. A versão atual é sempre a última versão efetiva da qual uma nova versão pode ser derivada. A classe Objeto Versão tem operações para inserir e remover versões da história de versões;

- **atualização da hierarquia de composição**. Cada versão de um objeto pode ser composta de outras versões. Tanto referências estáticas como dinâmicas podem ser usadas. A classe Objeto Versão fornece operações para atualizar a hierarquia de composição;

- **atualização de equivalências**. Uma versão de um objeto pode ser equivalente

a outras versões. São permitidas somente referências estáticas para facilitar a manutenção da consistência. A classe Objeto Versão fornece operações para atualizar equivalências.

A seguir são apresentados detalhes sobre a identificação de objetos, o controle de concorrência e as classes Objeto Genérico e Objeto Versão com a descrição de suas operações e atributos.

5.3.1 Identificação de Objetos e Suas Versões

Um objeto é composto de uma instância da classe Objeto Genérico e várias versões representando estágios distintos na história do objeto.

Uma instância de Objeto Genérico recebe um identificador no momento da criação consistindo de duas partes: o identificador da área de trabalho concatenado a um contador que identifica unicamente o objeto na área de trabalho. Esse contador é incrementado cada vez que um objeto genérico é criado.

Quando uma versão de um objeto é criada, recebe um identificador consistindo do identificador do objeto genérico concatenado a um contador de versão. Esse contador de versões é mantido pelo objeto genérico e incrementado toda vez que uma nova versão do objeto é criada.

Quando um objeto é copiado ou transferido de uma área para outra, seu identificador é atualizado substituindo-se a área de trabalho original pela área destino. As outras partes do identificador do objeto continuam as mesmas.

5.3.2 Controle de Concorrência

São dois os modos de trancas adotados:

- **l**. Tranca para leitura de versão;
- **d**. Tranca para derivação de versão;

Além disso, para a remoção de um objeto, são necessárias trancas exclusivas temporárias sobre todos os componentes na hierarquia de composição.

Para trancar um objeto composto, utilizamos também *trancas de intenção* [KBC+87] **il** e **id**. Se for requisitada uma tranca **l** sobre um objeto, todos seus ancestrais na hierarquia

de composição são trancados em modo **il**. Se for requisitada uma tranca **d** sobre um objeto, todos seus ancestrais na hierarquia de composição são trancados em modo **id**. Já seus descendentes na hierarquia de composição são implicitamente trancados em **l** ou **d** respectivamente. Por exemplo, veja a hierarquia de composição apresentada na figura 5.4. Cada vértice no grafo representa uma versão. Antes de alocar o vértice O_3, é necessário obter uma tranca de intenção, correspondente ao modo desejado, sobre todos os seus ancestrais, vértices O_1 e O_2. Somente depois a versão do objeto é trancada, juntamente com seus componentes, vértices O_4 e O_5. Note que esses são trancados implicitamente.

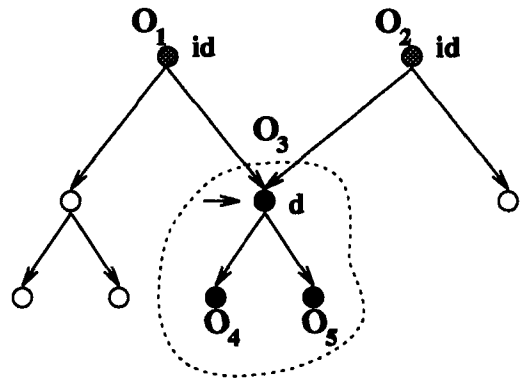


Figura 5.4: Exemplo de alocação de objeto composto utilizando trancas de intenção.

Para alocar qualquer objeto é preciso verificar a compatibilidade entre as diferentes trancas. A tabela 5.3.2 a seguir define essa compatibilidade.

Tabela 5.1: Compatibilidade entre os diferentes modos das trancas. O símbolo + indica compatível e − incompatível.

	il	l	id	d
il	+	+	+	−
l	+	+	−	−
id	+	−	+	−
d	−	−	−	−

A fim de fornecer suporte para as operações das transações cooperativas RequisitarCopia, RequisitarEmprestimo e RequisitarConcessao, discutidas em 5.4, anexamos uma segunda dimensão à tranca, que indica a permissão ou não de cooperação. São três os tipos de permissão:

- **c**. Permite cópia;

- **e**. Permite empréstimo;
- **o**. Permite concessão.

Por exemplo, se uma transação possui uma tranca sobre uma determinada versão do tipo **d/ceo**, então ela está permitindo que outras transações concorrentes requisitem a cópia, o empréstimo ou a concessão da versão derivada. As seguintes combinações são possíveis:

- permissões para concessão ou empréstimo ou cópia associadas a trancas do tipo **d**;
- permissões para cópia associadas a trancas do tipo l.

As operações associadas a trancas são TrancarVersao e DestrancarVersao. Além disso, as operações PrepararTranca, ConfirmarTranca e CancelarTranca implementam trancamento em duas fases quando um conjunto de versões deve ser trancado atomicamente - ou todas as versões são trancadas ou nenhuma é trancada. Essas operações são fornecidas pela classe Objeto Versão, descrita em 5.3.4.

5.3.3 Classe Objeto Genérico

A classe *Objeto Genérico* mantém a hierarquia de derivação (figura 5.5) fornecendo métodos para sua atualização. Além disso, são necessários métodos para criar, remover, copiar e transferir o objeto genérico entre áreas de trabalho.

Essa classe é composta dos seguintes atributos:

- **identificador do objeto**. Identifica unicamente o objeto genérico;
- **versão atual**. Identifica a versão atual do objeto;
- **vetor de versões**. Mantém uma entrada contendo o identificador da versão para cada uma das versões na história do objeto. A história é linear e as posições no vetor indicam os relacionamentos ancestral/descendente entre as versões;
- **contador de versão**. Usado na geração do identificador único de cada versão.

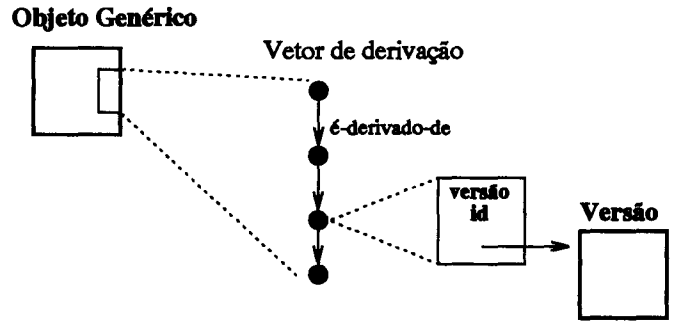


Figura 5.5: Representação da hierarquia de derivação.

As operações dessa classe são descritas a seguir:

1. **CriarNovoObjetoGenerico**
   - parâmetros: área de trabalho.
   - retorna: identificador do objeto genérico.
   - Cria o objeto genérico retornando seu identificador único.
2. **CriarObjetoGenerico**
   - parâmetros: área de trabalho, estado linearizado.
   - retorna: identificador do objeto genérico.
   - Essa operação cria um objeto genérico na área de trabalho passada como parâmetro. Inicializa o estado do objeto genérico com o estado passado como parâmetro. Cria o identificador único do objeto genérico substituindo o identificador da área de trabalho original pela área de trabalho especificada como parâmetro.
3. **RemoverObjetoGenerico**
   - retorna: resultado.
   - Remove o objeto genérico se não existirem versões a ele associadas. Caso contrário, retorna falha.
4. **CopiarObjetoGenerico**
   - parâmetros: área de trabalho destino.
   - retorna: novo identificador, resultado.

- Cria o objeto genérico na área de trabalho destino inicializando o estado do novo objeto com o estado do objeto genérico original. Para isso, lineariza o estado do objeto e invoca a operação CriarObjetoGenerico.

5. **AtualizarObjetoGenerico**

   - parâmetros: estado linearizado.
   - retorna: resultado.
   - Deslineariza o estado passado como parâmetro e atualiza o estado do objeto genérico com esse novo valor.

6. **TransferirObjetoGenerico**

   - parâmetros: área de trabalho destino.
   - retorna: resultado.
   - Essa operação é encarregada de transferir o objeto genérico para a área de trabalho destino. Primeiramente, lineariza o estado do objeto e atualiza o objeto genérico na área destino através da operação AtualizarObjetoGenerico. Finalmente, remove o objeto genérico da área de trabalho atual.

7. **InserirVetorDerivacao**

   - parâmetros: identificador da versão.
   - retorna: resultado.
   - Insere o identificador da versão passado como parâmetro no vetor de versões. Atualiza versão atual para novo identificador.

8. **RetirarVetorDerivacao**

   - parâmetros: identificador da versão.
   - retorna: versão atual, resultado.
   - Atualiza o vetor de derivação, excluindo o identificador passado como parâmetro. Se o identificador da versão corresponder à versão atual, atualiza a versão atual para a versão ancestral.

9. **FornecerVersaoAtual**

   - Retorna o identificador da versão atual.

10. **FornecerVersaoAnterior**

    - parâmetros: identificador de versão.

- Retorna o identificador da versão ancestral da versão passada como parâmetro.

11. **FornecerVersaoPosterior**

    - parâmetros: identificador de versão.
    - Retorna o identificador da versão descendente da versão passada como parâmetro.

12. **CriarIdentificadorVersao**

    - retorna: identificador de versão.
    - Soma um ao contador de versões e retorna esse valor.

5.3.4 Classe Objeto Versão

A classe Objeto Versão possui operações e atributos para manter os relacionamentos de composição e equivalência entre versões e controlar o acesso concorrente de aplicações.

Os atributos dessa classe são os seguintes:

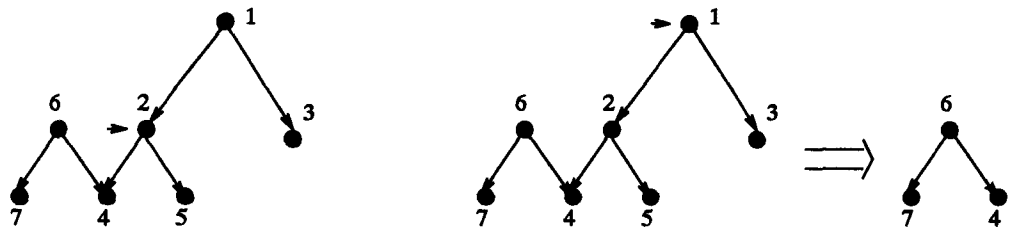
- **identificador da versão**. Identifica unicamente a versão. É formado pelo identificador do objeto genérico concatenado a um contador de versão. O identificador da versão é criado a partir de um contador incrementado toda vez que a operação CriarIdentificadorVersao é invocada;
- **vetor de compostos**. Contém uma entrada para cada versão que usa a versão em questão como componente;
- **vetor de componentes**. Contém o identificador de todas as versões componentes da versão em questão;
- **estado da versão**. Uma versão pode estar em três estados:
  - *temporário.* Indica que a versão foi criada por uma transação ainda em execução e está, portanto, disponível somente para a transação que a criou;
  - *efetivo.* Indica que a transação que criou a versão já foi validada e a versão está, portanto, disponível para outras transações;
  - *liberado para leitura.* Uma transação pode liberar uma versão antes do término através da operação LiberarObj. Contudo, essa versão ficará disponível somente para leitura e poderá ser removida se a transação abortar;

- **lista de transações que obtiveram trancas sobre a versão e o modo de acesso.** Essa lista implementa o relacionamento entre versões do objeto e o gerenciador de transações da aplicação;
- **lista de transações que esperam trancas sobre a versão.** Essa lista implementa o relacionamento entre versões do objeto e o gerenciador de transações da aplicação.

As operações dessa classe são descritas a seguir:

1. **CriarVersao**
   - parâmetros: identificador único do objeto genérico, estado linearizado.
   - retorna: identificador da nova versão, resultado.
   - Deslineariza o estado passado como parâmetro e inicializa o estado da nova versão com esse estado. Marca a versão como "temporária". Invoca a operação CriarIdentificadorVersao a fim de criar o identificador da nova versão. O resultado é anexado ao identificador do objeto genérico para formar o identificador único da versão.
2. **DerivarVersao**
   - retorna: identificador da nova versão, resultado.
   - Essa operação é responsável por derivar uma nova versão a partir da versão atual. Lineariza o estado da versão e requisita a criação de uma nova versão através da operação CriarVersao. Essa nova versão será inicializada com o estado da versão corrente. Finalmente, atualiza o vetor de derivação.
3. **RemoverVersao**
   - retorna: resultado.
   - Verifica se a versão não é componente de nenhuma outra. Em caso afirmativo, a versão não pode ser removida. Se a versão possui referências de composição para outras versões, todas as versões componentes são removidas. Requisita a exclusão da entrada no vetor de derivação referente à versão removida.

     A figura 5.6 resume o procedimento de remoção de versões. Cada vértice representa a versão de um objeto. O vértice 2, em 5.6 A), não pode ser removido, pois é componente do vértice 1. Em 5.6 B), a remoção do vértice 1 implica na remoção de todos os seus componentes, vértices 2, 3 e 5. Como o vértice 4 faz parte de outra configuração, apenas a referência de composição é removida.



Falha na remoção do vértice 2

A)

Remoção do vértice 1

B)

Figura 5.6: Remoção de objetos compostos e componentes.

4. **CopiarVersao**

   - parâmetros: área de trabalho destino.
   - retorna: novo identificador da versão, resultado.
   - Essa operação copia a versão para a área de trabalho destino. Se o objeto genérico não foi copiado antes essa operação falha. Para isso, lineariza o estado da versão e requisita a operação CriarVersao. Se existirem versões componentes estas também são copiadas para a área destino.

5. **AtualizarVersao**

   - parâmetros: estado linearizado.
   - retorna: resultado.
   - Deslineariza o estado passado como parâmetro e inicializa o estado da versão corrente a partir do estado passado como parâmetro.

6. **TransferirVersao**

   - parâmetros: área de trabalho destino.
   - retorna: resultado.
   - Essa operação transfere a cópia da versão para a área de trabalho destino. Entretanto, se a versão foi trancada apenas para leitura, a cópia é simplesmente descartada. Se existem versões componentes, essas também são transferidas. Primeiramente, lineariza o estado da versão corrente e requisita a operação AtualizarVersao sobre a versão na área de trabalho destino. Remove a versão e suas componentes da área de trabalho corrente.

7. **TrancarVersao**

   - parâmetros: identificador da transação, modo de acesso, requisição bloqueante/não bloqueante.
   - retorna: resultado.
   - Todos os componentes ancestrais na hierarquia de composição devem ser trancados em modo de intenção e a versão corrente no modo requisitado se houver compatibilidade com as trancas já existentes. Se existe compatibilidade, a operação retorna sucesso. Caso contrário, a transação pode ser bloqueada ou não conforme o parâmetro da operação. A transação bloqueada é inserida na lista de transações esperando trancas e a transação que obteve tranca é inserida na lista de transações que obtiveram trancas sobre a versão.

8. **PrepararTranca**

   - parâmetros: identificador da transação, modo de acesso.
   - retorna: resultado.
   - Essa operação implementa a primeira fase do processo de tranca. Verifica se há compatibilidade entre o modo de acesso requisitado e as trancas já existentes sobre a versão. Se for compatível, insere a transação na lista de transações que alocam a versão indicando que a tranca está em preparação. Caso contrário, retorna fracasso. Para trancar uma versão, deve-se primeiro trancar as versões dos objetos ancestrais na hierarquia de composição em modo de intenção. (seção 5.3.2). Os componentes são trancados implicitamente.

9. **ConfirmarTranca**

   - parâmetros: identificador da transação.
   - retorna: resultado.
   - Essa operação implementa a segunda fase do processo de tranca. Muda o estado das trancas para confirmadas.

10. **CancelarTranca**

    - parâmetros: identificador da transação.
    - retorna: resultado.
    - Essa operação implementa a segunda fase do processo de tranca, cancelando as trancas de preparação.

11. **DestrancarVersao**

- parâmetros: identificador da transação.
- retorna: resultado.
- Destranca a versão corrente e todas as suas componentes, se existirem. Retira o identificador da transação especificada como parâmetro da lista de transações. Se existirem transações a espera da versão liberada, concede a versão para a próxima transação, desbloqueando-a.

12. **MudarEstado**

- parâmetros: novo estado da versão.
- Atualiza o estado da versão para efetivo ou liberado para leitura.

13. **InserirVetorComposicao**

- parâmetros: identificador da versão.
- retorna: resultado.
- Insere o identificador da versão passada como parâmetro no vetor de componentes. Insere, no vetor de compostos da versão passada como parâmetro, o identificador da versão corrente.

14. **RetirarVetorComposicao**

- parâmetros: identificador da versão.
- retorna: resultado.
- Remove o identificador da versão do vetor de componentes. O identificador da versão corrente também é removido do vetor de compostos da versão passada como parâmetro.

15. **InserirVetorEquivalencia**

- parâmetros: identificador da versão.
- retorna: resultado.
- Constrói as relações de equivalência entre a versão em questão e a versão passada como parâmetro, inserindo uma entrada no vetor de equivalência.

16. **ExcluirVetorEquivalencia**

- parâmetros: identificador da versão.
- retorna: resultado.

- Remove o identificador da versão passada como parâmetro do vetores de equivalência.

17. **FornecerEquivalencia**

    - retorna: lista de versões equivalentes.
    - Retorna todas as versões equivalentes à versão em questão.

18. **CongelarReferencia**

    - retorna: resultado.
    - Substitui as referências dinâmicas pelos identificadores das versões atuais. Para tanto, percorre a hierarquia de composição no sentido descendente, resolvendo as referências dinâmicas de todos os seus componentes. Para cada referência indireta, consulta o objeto genérico a fim de obter o identificador da versão atual.

5.4 Gerenciamento de Transações

Nesse trabalho, distinguimos três tipos de gerenciadores de transações voltados para aplicações específicas:

- **gerenciamento de transações tradicional.** Para aplicações de curta duração;
- **gerenciamento de transações tradicional estendido.** Gerenciamento tradicional é flexibilizado possibilitando a liberação antecipada de recursos e estabelecimento de pontos de recuperação;
- **gerenciamento orientado à cooperação.** Gerenciamento de transações possibilita a modelagem de grupos através de hierarquias de transações e a criação de áreas de trabalho para cada transação na hierarquia. Para facilitar a cooperação, objetos podem ser *copiados* entre áreas de trabalho, *emprestados* de uma transação e posteriormente retornados, ou ainda, *concedidos* de uma transação para uma outra que adquire todos os direitos da primeira.

O diagrama da figura 5.7 representa o modelo de transações proposto utilizando a notação OMT [RBP+94]. Nesse diagrama apresentamos os relacionamentos das várias classes e suas operações. Uma aplicação deve criar instâncias da classe que implementa o gerenciador de transações adequado a suas características. Instâncias da classe ***Log*** devem ser persistentes, contendo informação que será usada na recuperação de transações interrompidas por uma falha. Cada uma das classes do modelo é descrita a seguir.

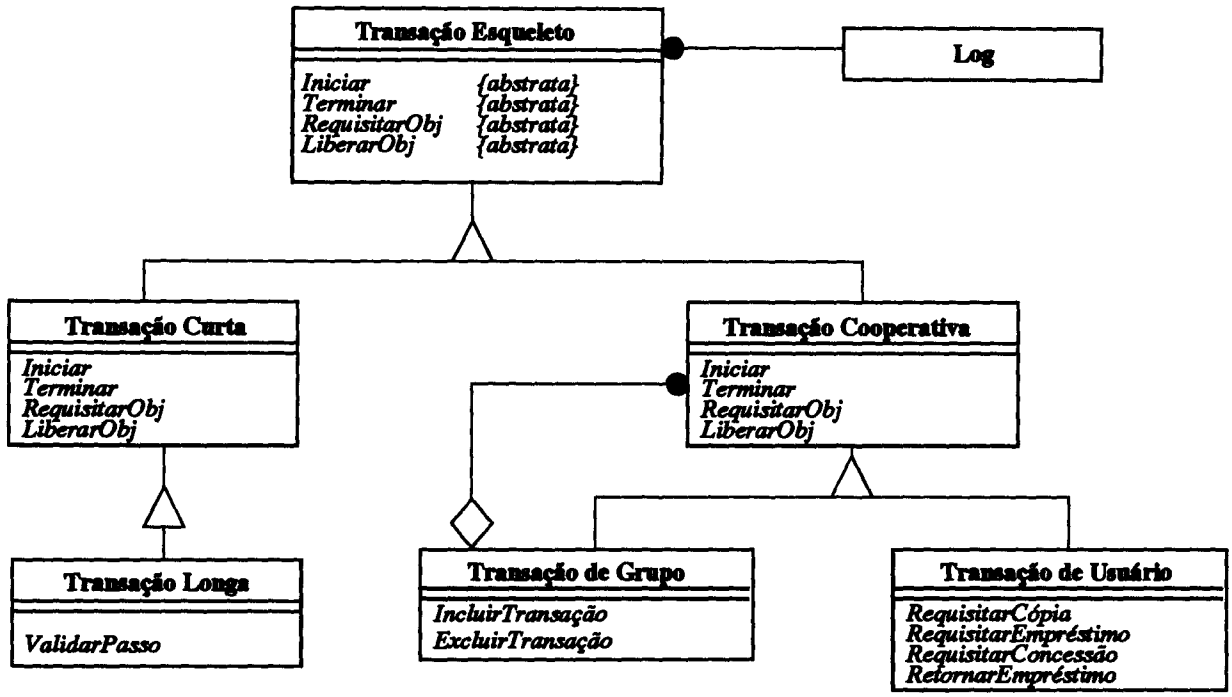


Figura 5.7: Modelo de classe dos gerenciadores de transações.

5.4.1 Classe Transação Esqueleto

A classe Transação Esqueleto define a interface das operações comuns a todos os tipos de gerenciadores de transações. Todas suas operações são abstratas e terão sua implementação posteriormente definida nas subclasses Transação Curta e Transação Cooperativa.

Os atributos dessa classe são os seguintes:

- identificador da transação;
- identificador da transação pai;
- identificador da área de trabalho;
- lista de objetos participantes que implementa o relacionamento participação na figura 5.3.

As operações desta classe são as seguintes:

1. **Iniciar**
   - parâmetros: identificadores das versões a trancar, modos de acesso, identificador da transação pai.
   - retorna: identificador da transação, área de trabalho da transação, identificadores das versões derivadas, resultado ("sucesso"/"falha").
2. **Terminar**
   - parâmetros: tipo de terminação.
   - retorna: resultado.
3. **RequisitarObj**
   - parâmetros: identificador da versão a trancar, modo de acesso, requisição bloqueante/não bloqueante.
   - retorna: identificador da versão derivada, resultado.
4. **LiberarObj**
   - parâmetros: identificador da versão.
   - retorna: resultado.

5.4.2 Classe Transação Curta

A classe Transação Curta herda a interface da classe Transação Esqueleto definindo a implementação das operações dos gerenciadores de transações curtas.

A semântica das operações é a seguinte:

1. **Iniciar**
   - Cria o identificador da transação.
   - Requisita a tranca sobre as versões dos objetos passadas como parâmetro. As trancas devem ser obtidas de forma atômica - ou todas são obtidas ou nenhuma é obtida. Se o modo de acesso for derivação, novas versões são derivadas a partir das versões atuais.
   - Se a tranca tem sucesso, insere as versões trancadas na lista de objetos da transação.

- Armazena no log um registro indicando o início da transação.

2. **Terminar**

   - Se o tipo de terminação for "validação", armazena um registro no log indicando que a transação será validada. O registro deve conter o identificador da transação e os identificadores dos objetos e versões participantes. Percorre a lista de objetos da transação, efetivando as novas versões criadas. Destranca todas as versões trancadas. Armazena registro no log indicando que a transação terminou com sucesso.
   - Se o tipo de terminação for "cancelamento", percorre a lista de objetos da transação descartando as versões temporárias e destrancando as versões trancadas. Armazena registro no log indicando que a transação terminou com cancelamento.

3. **RequisitarObj**

   - Requisita a tranca sobre a versão especificada como parâmetro (pode bloquear ou não a transação requisitante conforme o parâmetro da operação e a compatibilidade entre o modo de acesso solicitado e as trancas já existentes sobre a referida versão do objeto).
   - Se a operação de tranca tem sucesso, insere o identificador da versão na lista de objetos da transação.

4. **LiberarObj**

   - Se a versão especificada como parâmetro foi derivada, destranca a versão e libera a versão derivada somente para leitura. Se a transação abortar posteriormente, a versão derivada será descartada. Na lista de objetos da transação, indica que a versão foi liberada para leitura.
   - Se a versão do objeto foi trancada para leitura, destranca a versão e retira o objeto e sua versão da lista de objetos da transação.

5.4.3 Classe Transação Longa

A classe Transação Longa herda as operações da classe Transação Curta adicionando a operação ValidarPasso que permite a liberação de recursos antes do término da transação e a efetivação das atualizações realizadas até o ponto da transação em que essa operação é invocada.

A definição da operação **ValidarPasso** é a seguinte:

- parâmetros: identificadores das versões a liberar.
- retorna: identificadores das versões derivadas, resultado ("sucesso"/"falha").
- Essa operação armazena um registro no log contendo informação sobre a validação de um passo da transação. Esse registro deve conter informação sobre os objetos que continuam trancados pela transação. Percorre a lista de objetos da transação tornando as novas versões efetivas. Destranca as versões especificadas como parâmetro. Deriva novas versões para os objetos que continuam participando da transação.

5.4.4 Classe Transação Cooperativa

A Classe Transação Cooperativa herda as operações da Classe Transação Esqueleto definindo a implementação das operações comuns a suas duas subclasses: Transações de Grupo e Transação de Usuário. Uma aplicação não cria instâncias dessa classe. Ao invés disso, uma aplicação cooperativa pode ser estruturada como uma hierarquia de transações cujos nós internos são instâncias de Transação de Grupo e folhas são instâncias de Transação de Usuário.

Cada transação cooperativa está associada a uma área de trabalho de onde os objetos e suas versões podem ser copiados pelas transações descendentes e posteriormente transferidos de volta quando uma transação descendente terminar o trabalho sobre os objetos.

A figura 5.8 ilustra a hierarquia de transações e a alocação de objetos. T_1 é uma transação de grupo constituída pelas subtransações T_2 (transação de grupo), T_4 (transação de grupo) e T_3 (transação de usuário). T_1 alocou e requisitou cópias de cinco versões de objetos das quais três foram alocadas pelas suas transações descendentes.

As operações dessa classe são as seguintes:

1. **Iniciar**
   - Cria a área de trabalho da transação.
   - Cria o identificador da transação.
   - Inclui a transação cooperativa na lista de transações da transação ancestral.
   - Tranca atomicamente as versões dos objetos passadas como parâmetro.

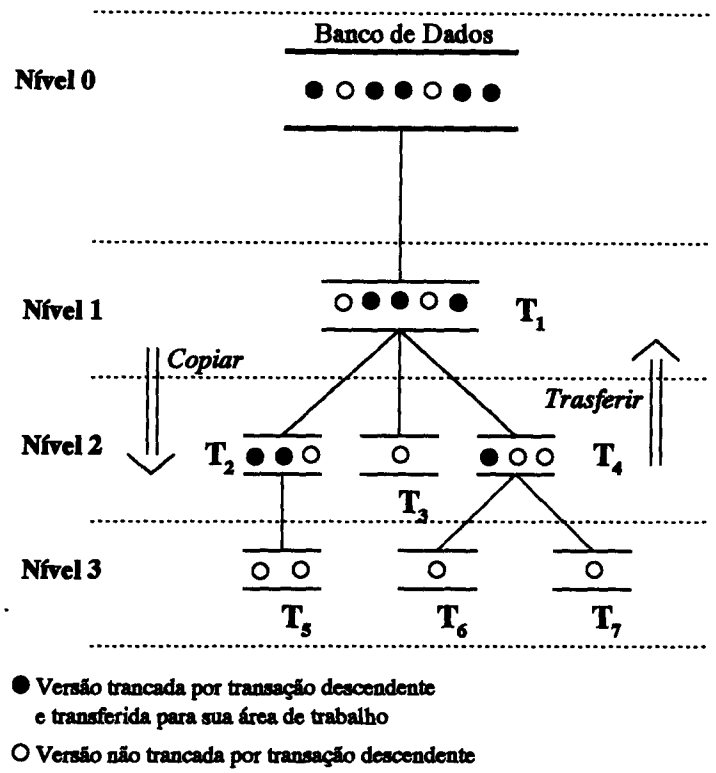


Figura 5.8: Estrutura básica das transações cooperativas.

- Se a tranca tem sucesso, insere os objetos e suas versões na lista de objetos da transação e requisita uma cópia dessas versões da área da transação pai para a área de trabalho da transação cooperativa. Se a versão não for encontrada na área da transação pai, a busca continua na hierarquia de áreas de trabalho no sentido folha-raiz. Quando a versão é encontrada, uma tranca no modo especificado é solicitada e se for concedida, uma cópia é realizada da área ancestral para a área descendente. Esse processo se repete até a área da transação cooperativa requisitante. É necessário que, para cada versão a ser copiada, haja uma cópia do objeto genérico correspondente na mesma área de trabalho destino.

  Por exemplo, a transação T_6 (figura 5.8) requisita uma cópia da área de trabalho de T_4. Se T_4 não possui a versão, ela solicita a T_1 e assim sucessivamente até chegar ao nível 0. Assim que a versão é trancada, a cópia é realizada nas áreas descendentes até chegar à área de trabalho da transação de grupo. Da mesma forma é realizada a cópia do objeto genérico correspondente, porém, sem a necessidade de mantê-lo trancado.

- Armazena registro no log indicando o início da transação.

2. **Terminar**

   - Verifica o estado das transações descendentes, se existirem. A transação cooperativa pode ser validada dependendo do estado de suas transações descendentes e do tipo de terminação especificado como parâmetro. Se o tipo de terminação for "cancelamento", a transação cooperativa será abortada. Os outros tipos de terminação possíveis são: "validação se todas as transações filhas terminarem com sucesso", "validação se a maioria das transações filhas terminarem com sucesso" e "validação das transações filhas que terminaram com sucesso".
   - Verifica o estado dos objetos participantes da transação. A transação cooperativa só poderá ser terminada se todas as versões emprestadas para outras transações tiverem sido retornadas à transação original.
   - Se o tipo de terminação for "validação", armazena registro no log indicando que a transação será validada. O registro deve conter o identificador da transação e dos objetos participantes e suas versões. Versões emprestadas são transferidas para as áreas das transações originais. Versões que foram copiadas através da operação RequisitarCopia e sobre as quais a transação só pode realizar alterações na sua área de trabalho são descartadas. Versões derivadas são transferidas para a área da transação pai ou para a área pública (dependendo do nível da transação cooperativa), juntamente com os respectivos objetos genéricos. As versões que originaram as versões derivadas são destrancadas. Versões trancadas para leitura são removidas da área da transação e destrancadas.
   - Se o tipo de terminação for "cancelamento", descarta as versões e os objetos genéricos correspondentes na área de trabalho da transação e libera todas as trancas requisitadas pela transação. Para versões emprestadas, retorna os privilégios para as transações originais sem contudo propagar as mudanças.
   - Armazena registro no log indicando que a transação foi terminada e o tipo de terminação.
   - Indica à transação pai que a transação cooperativa terminou e o tipo de terminação.
   - Descarta a área de trabalho da transação cooperativa.

3. **RequisitarObj**

   - Requisita a tranca sobre a versão do objeto no modo especificado (pode bloquear ou não a transação requisitante conforme o parâmetro da operação e a compatibilidade entre o modo de acesso solicitado e as trancas já existentes sobre a referida versão).

- Se a tranca tem sucesso, insere o identificador do objeto e sua versão na lista de objetos da transação, copia a versão e o objeto genérico correspondente da área da transação pai para a área da transação cooperativa. Busca em níveis superiores e cópias em vários níveis podem ser necessárias como explicado anteriormente.

4. **LiberarObj**

   - Se a versão do objeto foi derivada, transfere a versão para a área da transação pai. A versão liberada fica disponível somente para leitura.
   - Se a versão do objeto foi trancada para leitura, destranca a versão e descarta sua cópia da área da transação cooperativa.

5.4.5 Classe Transação de Usuário

A Classe Transação de Usuário herda as operações da Classe Transação Cooperativa adicionando as operações que relaxam a seriabilidade e melhoram a visibilidade entre usuários trabalhando cooperativamente. Essas operações permitem a transferência de versões temporárias de uma área de trabalho para outra. Um usuário pode ficar ciente de alterações sobre um objeto realizadas por um segundo usuário requisitando a cópia da versão do objeto na área de trabalho da transação de usuário através da operação RequisitarCopia. Além disso, dois usuários podem trabalhar alternadamente sobre uma mesma versão invocando a operação RequisitarEmprestimo antes de aplicar modificações e RetornarEmprestimo para permitir que o segundo usuário continue trabalhando sobre a versão. Finalmente, uma versão e privilégios associados podem ser transferidos de uma transação para outra quando a operação RequisitarConcessao é invocada. A versão passa a pertencer à segunda transação que pode descartar ou efetivar essa versão.

1. **RequisitarCopia**

   - parâmetros: identificador do objeto.
   - retorna: resultado.
   - Essa operação busca a versão mais atual do objeto copiando-a para a área de trabalho da transação requisitante. A transação original que derivou a versão mantém sua cópia e seus privilégios. A cópia tem sucesso somente se a versão ancestral que originou a versão tiver sido trancada em modo que permita a cópia por outras transações. Após a cópia, a transação de usuário pode realizar qualquer operação sobre a versão do objeto, mas essa cópia será descartada no final da transação.

A figura 5.9 a seguir ilustra o processo de cópia. A transação T_y requisita uma cópia do objeto O_1. Uma versão do objeto foi derivada por T_x. Se em todo o caminho de alocação (T_s) houver permissão de cópia, então é fornecida uma cópia a T_y. A busca se faz no sentido de T_y até a transação ancestral que tiver uma cópia de O_1.

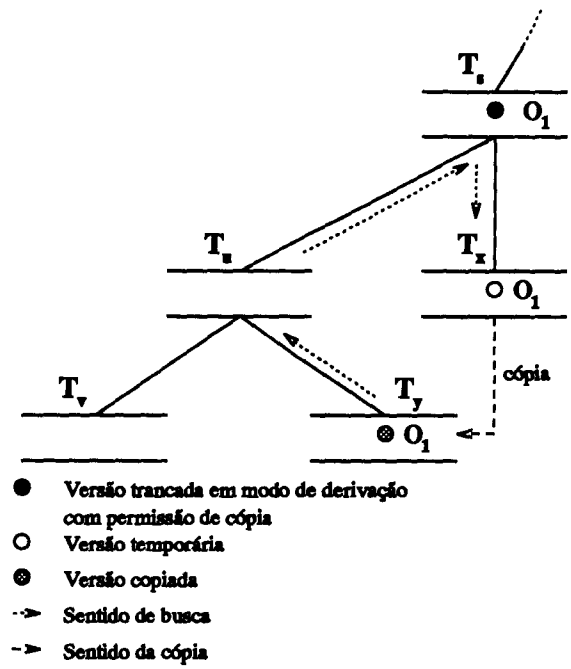


Figura 5.9: Hustração do processo de cópia

2. **RequisitarEmprestimo**

- parâmetros: identificador do objeto, *timeout.*
- retorna: área da transação original, resultado.
- Essa operação busca a cópia mais atual da versão copiando-a para a área da transação requisitante. O empréstimo tem sucesso se o modo de acesso permitir empréstimo. A transação original mantém uma cópia e pode continuar aplicando atualizações sobre ela. Contudo essa cópia será substituída pela segunda transação quando invocar a operação RetornarEmprestimo. Se o *timeout* especificado como parâmetro da operação esgotar sem que a segunda transação tenha retornado a versão emprestada, a transação original volta a ter os privilégios iniciais e a transação que tomou emprestado perde o direito de alterar o objeto na área origem. Não é permitido o empréstimo de versões que estejam emprestadas a outras transações e nem que tenham sido adquiridas por empréstimo ou cópia.

Essa operação segue a mesma regra de busca descrita para a operação RequisitarCopia. Veja o exemplo da figura 5.10.

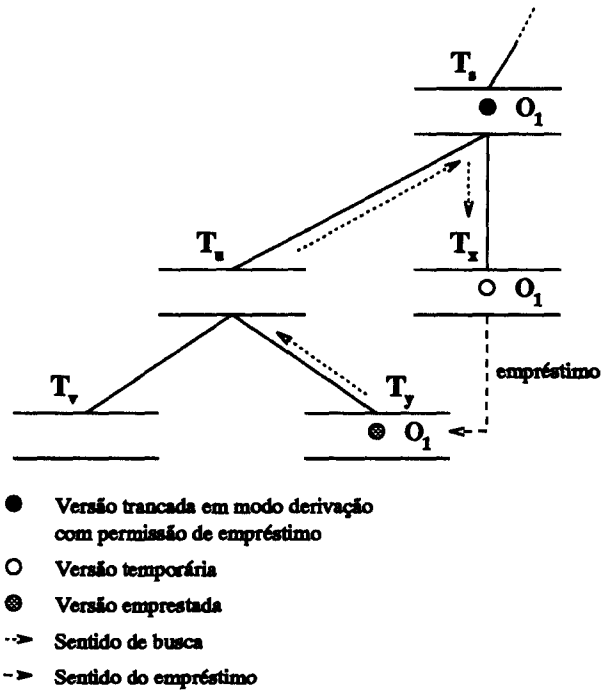


Figura 5.10: Ilustração do processo de empréstimo

3. **RetornarEmprestimo**

   - parâmetros: identificador da versão, área da transação original, propaga estado/não propaga estado.
   - retorna: resultado.
   - Se for especificado que o estado deve ser propagado, a versão do objeto é transferida para a área de trabalho da transação original que volta a ter todos os privilégios sobre o objeto. Caso contrário, a versão é descartada e a transação original volta a ter todos os privilégios sobre a sua cópia.

4. **RequisitarConcessao**

   - parâmetros: identificador do objeto.
   - retorna: resultado.
   - A versão mais atual do objeto é transferida para a área da transação requisitante. A transação original concede todos seus direitos sobre o objeto para a transação requisitante. A concessão é obtida se os modos de acesso desde a transação raiz até a transação descendente que possui a cópia mais recente

do objeto permitirem. Cópias do objeto são realizadas na hierarquia de áreas de trabalho da mesma forma descrita para a operação RequisitarObj. O objeto genérico associado também é transferido. Não é permitida a concessão de versões que estejam emprestadas a outras transações e nem que tenham sido adquiridas por empréstimo ou cópia.

A figura 5.11 ilustra o processo de concessão. A transação T_y requisita a concessão do objeto O_1. A busca é feita de T_u a T_s, onde O_1 é encontrado. Como O_1 foi alocado por T_x, verifica-se se o modo de acesso permite concessão. Em caso afirmativo, a versão é concedida a T_y.

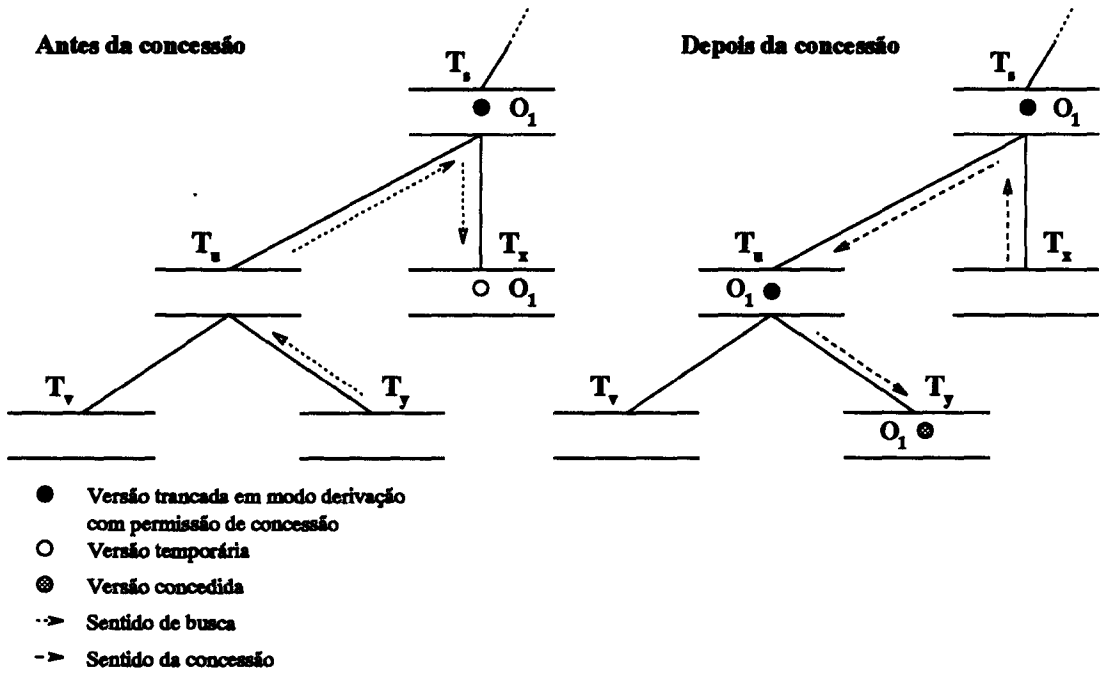


Figura 5.11: Ilustração do processo de concessão

5.4.6 Classe Transação de Grupo

Esta classe herda as operações da Classe Transação Cooperativa adicionando duas operações que permitem construir a hierarquia de transações cooperativas. As instâncias dessa classe representam os vértices não-folhas da árvore de transações (figura 5.8).

Os atributos da classe são os seguintes:

- **lista de transações**. Implementando o relacionamento entre entidades de Transação Cooperativa. Esse relacionamento representa as transações filhas de determinada transação de grupo. (na figura 5.7).

As operações da classe são as seguintes:

1. **IncluirTransacao**
   - parâmetros: identificador da transação filha.
   - retorna: área da transação pai, resultado ("sucesso"/"falha").
   - Essa operação inclui o identificador da transação filha na lista de transações da transação pai.
2. **ExcluirTransacao**
   - parâmetros: identificador da transação filha, tipo de terminação.
   - retorna: resultado ("sucesso"/"falha").
   - Essa operação muda o estado da transação filha na lista da transações da transação pai indicando o tipo de terminação.

5.5 Avaliação

O capítulo 3 apresentou alguns modelos de transações existentes na literatura. Nessa seção, procuraremos compará-los com nossa proposta, através da apresentação de suas principais características.

O modelo de Salem [SGMA89] obedece a mesma estrutura plana das transações convencionais. Apenas a política de liberação de trancas é relaxada a fim de permitir maior concorrência. Entretanto, o custo para se manter a consistência é alto: muitas transações podem ser canceladas e muito trabalho pode ser desfeito. Além disso, não oferece suporte algum à cooperação. Em contrapartida, é um modelo simples quando comparado aos demais.

O modelo de transações aninhadas de Moss [Mos81] surge como uma extensão do modelo convencional. A natureza hierárquica permite melhor estruturação das tarefas e maior versatilidade na recuperação, além de permitir paralelismo entre subtransações *irmãs*. Esses conceitos constituem a base de vários outros modelos, muitos deles destinados à cooperação. Entretanto, assim como nas transações convencionais, ainda são mantidas as propriedades ACID. Isso impede a utilização desse modelo em tarefas cooperativas e/ou longas devido ao caráter restritivo do critério de correção.

O relaxamento da propriedade de isolamento do Sagas [GMS87], proposto por Garcia-Molina, difere da idéia de Salem. É permitido que transações vejam resultados parciais de suas concorrentes, entretanto, os cancelamentos em cascata não ocorrem devido à

utilização de recuperação semântica. Assim, o modelo de transações começa a se aproximar da aplicação a fim de ganhar concorrência, idéia defendida pelo mesmo autor em [GM83]. Contudo, o processo de recuperação fica sob responsabilidade do usuário, através da criação das *funções de compensação*, o que gera algumas desvantagens, como visto no capítulo 3. Esse modelo não trata cooperação.

Calton Pu [PKH88] propôs um modelo baseado em operações que possibilitam a reestruturação dinâmica das transações. Essa idéia fornece versatilidade à execução, pois possibilita a liberação de resultados intermediários. O apoio a atividades cooperativas é oferecido através de operações para unir e dividir transações. Cancelamentos em cascata também podem ocorrer nesse modelo.

A proposta de Shrivastava [SW90], Ações Multicoloridas, também baseia-se na liberação antecipada das trancas. No entanto, pode haver o relaxamento da atomicidade, evitando-se cancelamentos em cascata. Na verdade, esse modelo é uma extensão das transações aninhadas de Moss, onde um atributo *cor* é fornecido às ações e aos dados alocados. Novamente, não há a preocupação aqui com cooperação.

O modelo de Bancilhon [BKK85] está baseado na estruturação das tarefas de um projeto CAD. Portanto, seu enfoque principal refere-se à cooperação. Vários tipos de transações foram definidos, com controles de concorrência específicos, permitindo que uma aplicação cooperativa seja estruturada hierarquicamente.

O modelo de transações cooperativas de Nodine [NRZ94] também é um modelo de transações aninhadas. Contudo, a principal diferença do modelo anterior é o controle de concorrência semântico, baseado na compatibilidade entre as operações da aplicação. Desta forma, obtém-se um maior nível de concorrência e flexibilidade que possibilita cooperação entre projetistas. Porém, surgem algumas desvantagens, observadas por [US94]: especificidade do controle de concorrência; dificuldade de remodelagem e; dependência do conhecimento do programador do controle de concorrência sobre as operações da aplicação.

O modelo apresentado em [NG92] destina-se a fornecer suporte a aplicações do tipo CASE. Baseia-se nas relações de dependência existentes entre os módulos e nas relações de responsabilidade dos usuários para com os mesmos. A idéia é difundir as propostas de alterações dos módulos antes de efetuá-las. Os usuários, pertencentes a um grupo, opinam sobre as alterações e, apenas quando todos as aceitam, essas são efetuadas atomicamente. Um protocolo de comunicação é responsável por notificar e transmitir as propostas para os usuários. Portanto, informações do nível da aplicação (relações de responsabilidade e dependência) são utilizadas pelo sistema de transações para auxiliar na manutenção da consistência.

O modelo de transações de Klahold [KSUW85] é estruturado numa hierarquia de dois níveis. Associado a cada nível, há áreas de trabalho pertencentes a transações de grupo

(primeiro nível) ou a transações de usuário (segundo nível). Ao invés de usar controle de concorrência semântico, esse modelo provê um conjunto de trancas, dando suporte às operações de leitura, leitura exclusiva, derivação, atualização e remoção. A concorrência entre as transações do primeiro nível baseia-se em seriabilidade, mas as demais não obedecem a tal critério de correção, tornando o sistema mais flexível e adequado à cooperação.

O modelo apresentado nesta dissertação fornece três gerenciadores de transações que mesclam algumas características dos modelos estudados. Foi apresentada uma hierarquia de classes que representa os vários tipos de gerenciadores: transações curtas, longas e cooperativas. Dependendo das necessidades das aplicações, devem-se criar instâncias do tipo adequado.

As transações curtas são semelhantes às convencionais, exceto pela operação LiberarObj e pela capacidade de manipulação de versões de objetos complexos. A operação LiberarObj permite que recursos sejam liberados antes do final da transação, mas ao contrário de vários modelos, não ocorrem cancelamentos em cascata.

As transações longas, uma especialização de transações curtas, assemelham-se às ações aglutinadas do modelo de Shrivastava [SW90]. A operação ValidarPasso marca o fim de uma subtransação atômica e o início de outra com possível liberação de recursos.

Por fim, o suporte à cooperação e a grupos hierárquicos é fornecido pelas transações cooperativas, estruturadas de forma aninhada. A cooperação é realizada por intermédio de operações de *empréstimo*, *cópia*, *concessão*, *checkin* e *checkout*, que permitem a transferência de objetos entre as diferentes transações.

As tabelas 5.2 e 5.3, a seguir, apresentam as diferentes abordagens dos modelos estudados, juntamente com a proposta dessa dissertação.

Tabela 5.2: Comparação entre diferentes modelos de transações.

Modelo	Estrutura	Critério de Correção	Recuperação	Suporte a Grupos	Atomicidade	Suporte a Cooper.	Propósito	Isolamento
Moss [Mos82]	Aninhada	Seriabilidade	Cancelamento	Não	Sim	Não	Genérico	Sim
Salem [SGMA89]	Plana	Seriabilidade	Cancelamento (4)	Não	Sim	Não	Aplic. Longas	Relaxado
Garcia Molina [GMS87]	Aninhada 2 Níveis	Não Serializável	Compensação	Não	Relaxada (2)	Não	Aplic. Longas	Relaxado
Pu [PKH88]	Plana ou Aninhada	Seriabilidade	Cancelamento (4)	Não	Relaxada	Restrito	Ambientes de Projeto	Relaxado
Shrivastava [SW90]	Aninhada	Seriabilidade	Cancelamento	Não	Relaxada (5)	Não	Aplic. Distrib.	Relaxado
Bancilhon [BKK85]	Aninhada	Não (1) Serializável	Cancelamento	Hierárquicos	Relaxada (5)	Sim	Ambientes de Projeto	Sim
Nodine [NRZ94]	Aninhada	Semântico	Cancelamento e Compensação	Hierárquicos	Relaxada (2)	Sim	Ambientes de Projeto	Relaxado
Narayanas. [NG92]	Plana .	Decisão dos Usuários	Cancelamento	Planos	Sim	Sim	CASE	Relaxado
Klahold [KSUW85]	Aninhada 2 Níveis	Seriabilidade no 1ºNível	Cancelamento	Em 2 Níveis	Relaxada (5)	Sim	Ambientes de Projeto	Relaxado (3)

(1) Depende do tipo de transação. (2) Uso de pontos de recuperação.
(3) Devido as operações de *empréstimo* e *concessão*. (4) Permite cancelamentos em cascata.
(5) Através do aninhamento de transações.

Tabela 5.3: Características da proposta apresentada neste trabalho.

Modelo	Estrutura	Critério de Correção	Recuperação	Suporte a Grupos	Atomicidade	Suporte a Cooper.	Propósito	Isolamento
Trans. Curtas	Plana	Seriabilidade	Cancelamento	Não	Sim	Não	Aplic. Curtas	Relaxado (1)
Trans. Longas	Plana	Seriabilidade	Cancelamento	Não	Relaxada (2)	Não	Aplic. Longas	Relaxado (1)
Trans. Coop.	Aninhada	Não Serializável	Cancelamento	Hierárquicos	Relaxada (4)	Sim (3)	Ambientes de Projeto	Relaxado (2)(3)(4)

(1) Liberação antecipada de recursos. (2) Uso de pontos de recuperação.
(3) Através das operações de *empréstimo*, *cópia* e *concessão*. (4) Através do aninhamento de transações.

Capítulo 6

Conclusões e Trabalhos Futuros

Este capítulo resume a dissertação, apresenta suas contribuições e algumas sugestões para possíveis extensões. Inicialmente, a seção 6.1 descreve, em linhas gerais, as diferentes abordagens seguidas pelos modelos de transações mais significativos. Posteriormente, são apresentados na seção 6.2 os requisitos das aplicações avançadas e na seção 6.3 resumimos as características de nossa proposta. A seção 6.4 comenta as contribuições deste trabalho e a seção 6.5 apresenta as possíveis futuras extensões.

6.1 Modelos de Transações Estudados

Toda a pesquisa sobre transações gira em torno de como garantir a consistência dos dados. Descobrir uma maneira adequada de fazê-lo é a preocupação básica dos diferentes modelos, pois nem sempre se consegue descrever modelos de transações, sem impor restrições que prejudiquem o desempenho dos sistemas.

O critério de correção baseado em seriabilidade e isolamento de transações constitui um método adequado para atender aos requisitos das aplicações tradicionais, pois elas têm curta duração e não exigem cooperação. Assim, critérios rigorosos de correção podem ser impostos. Mas, não se pode aplicar o modelo convencional para suportar aplicações de longa duração ou cooperativas, pois:

- em aplicações longas o desempenho é prejudicado pelos seguintes motivos:
  - aumento do número de transações bloqueadas e de *deadlocks*, caso a implementação se baseie em mecanismos bloqueantes;
  - aumento do trabalho perdido em caso de cancelamento de transações;

- em aplicações cooperativas, além da longa duração, há necessidade de interação entre usuários que não podem se manter isolados.

Conclui-se, então, que as restrições impostas pelas propriedades ACID das transações convencionais não se adaptam às aplicações de longa duração ou cooperativas. Assim, existem duas abordagens distintas dos modelos de transações não convencionais: suporte à longa duração e suporte à cooperação.

Existem várias propostas destinadas a fornecer suporte às aplicações longas. Geralmente, são baseadas no relaxamento da propriedade de atomicidade, através de pontos de recuperação e do relaxamento do isolamento, através de liberação antecipada de recursos. Isso provoca o aumento da concorrência, mas em contrapartida, pode causar excessivo aumento do trabalho perdido, se houver cancelamentos em cascata. Uma solução alternativa, bastante referenciada na literatura, é o Sagas [GMS87]. Esse modelo mantém a atomicidade e relaxa o isolamento. A recuperação aqui não mais se baseia em cancelamentos, mas sim em *funções de compensação*, que desfazem semanticamente as operações. Esse conceito é usado em outros modelos posteriores, tais como DOM [BOH+94], ConTract [WR94], entre outros.

Outra extensão do modelo convencional são as transações aninhadas [Mos81, Mos82]. Com esse conceito consegue-se um aumento da concorrência proporcionado pelo paralelismo entre as subtransações e maior flexibilidade na recuperação. Posteriormente, vários foram os modelos que se basearam nessa idéia, muitos dos quais destinam-se a fornecer suporte à cooperação [BSW88, Wal84, FZ89, SZ89, SW90, Ska89, NRZ94, US94]. Nesses modelos, a hierarquia de subtransações auxilia na estruturação de tarefas. Além disso, é possível associar áreas intermediárias de trabalho às subtransações, proporcionando ambientes adequados à cooperação.

No início da década de 80, Garcia-Molina [GM83] apresentou uma abordagem alternativa para o controle de concorrência. Sua idéia é levar em consideração a semântica das operações da aplicação para escalonar as operações das transações, aumentando o nível de concorrência. Posteriormente, outros modelos, também em ambientes cooperativos, basearam-se nessa idéia [SZ89, Ska89, NRZ94].

A interação entre usuários pode ser definida por critérios semânticos, como feito em [SZ89, Ska89, NRZ94]. Entretanto, critérios semânticos exigem, muitas vezes, especificações complexas que requerem do programador um bom conhecimento dos objetos da aplicação. Propostas alternativas mantêm a flexibilidade sem exigir especificações complexas [US94, NG92].

6.2 Requisitos das Aplicações Avançadas

Os diferentes modelos de transações surgiram devido aos requisitos impostos por novas aplicações e a não adequação do modelo de transações convencionais para atender esses requisitos. Entre os requisitos podemos citar:

1. Suporte a longa duração.
2. Suporte a trabalho de grupo.
3. Suporte a critérios flexíveis de correção.
4. Suporte a diversidade.
5. Suporte a dados complexos.
6. Suporte a evolução dos dados.
7. Suporte a múltiplas representações.

O itens 1 e 2 dizem respeito aos tipos de tarefas realizadas pelas aplicações avançadas. Geralmente são longas, podendo se estender por horas ou até dias. Além da longa duração, as aplicações cooperativas exigem interação entre os usuários. Esses requisitos obrigam ao relaxamento das propriedades de isolamento, seriabilidade e atomicidade.

O critério de correção usado pelos modelos de transações não podem ser rígidos. Em algumas soluções, os usuários de um grupo aprovam ou rejeitam as operações sobre objetos de sua responsabilidade [NG92]. Noutras, o critério de correção é baseado na semântica da aplicação [GM83, NRZ94, SZ89, Ska89].

Os diversos requisitos das aplicações avançadas, muitas vezes conflitantes, exigem que o modelo de transações seja adaptável às necessidades das diversas aplicações.

Os itens 5, 6 e 7 acima, referem-se aos dados manipulados pelas aplicações avançadas que podem:

- ser recursivamente compostos, ou seja, objetos mais simples são agregados formando objetos mais complexos;
- evoluir com o tempo e, por isso, exigem gerenciamento de versões;
- possuir mais de uma representação, necessitando de mecanismos para a manutenção da consistência das diversas representações.

6.3 Gerenciamento Flexível Proposto

A proposta apresentada neste trabalho oferece três tipos de gerenciadores de transações que são destinados aos seguintes propósitos: suporte às aplicações de curta duração, de longa duração e cooperativas. Uma aplicação deve então, criar instâncias da classe que melhor atende seus requisitos. Pode também definir um gerenciador que seja uma extensão de algum gerenciador já definido. Isso pode ser feito através do mecanismo de herança. Juntamente com o modelo de transações, apresentamos um modelo de gerenciamento de versões que foi conceitualmente baseado no modelo apresentado em [KBC+89, KBGW91].

A abordagem da apresentação de ambos os modelos foi baseada na notação OMT [RBP+94]. Para cada objeto, suas operações foram definidas em alto nível com um enfoque descritivo. Foram definidas entidades, representando gerenciadores de transações e objetos de aplicação, além dos relacionamentos entre essas entidades.

Resumimos as principais características dos modelos propostos:

- Gerenciamento de Transações:
  - Curtas:
    - baseado no modelo de transações convencionais. Entretanto, devido à inclusão da operação LiberarObj, há uma maior flexibilidade, pois os recursos podem ser liberados antecipadamente, ficando disponíveis apenas para leitura;
  - Longas:
    - uma especialização da classe Transação Curta;
    - relaxa a atomicidade através dos vários passos de validação e permite a liberação antecipada de recursos;
  - Cooperativas:
    - possui estrutura aninhada, oferecendo suporte a grupos hierárquicos;
    - provê áreas de trabalho associadas com as várias subtransações na hierarquia. Essa estrutura constitui um ambiente propício à cooperação assíncrona, pois usuários/grupos podem executar suas tarefas isoladamente e usar as áreas de trabalho para trocar informações;
    - provê operações de *empréstimo*, *concessão* e *cópia*, fornecendo maior flexibilidade à cooperação. Com isso, os usuários podem ter acesso a estados intermediários de objetos e até mesmo adquirir direitos sobre os mesmos;

- Gerenciamento de Versões:
  - fornece suporte a referências estáticas e dinâmicas. As referências dinâmicas são implementadas pela instância da classe Objeto Genérico, responsável por manter o identificador da versão atual;
  - fornece controle de concorrência baseado em trancas nos modos de leitura e derivação. As trancas de intenção são empregadas a fim de diminuir a sobrecarga de alocação de objetos complexos;
  - provê mecanismos para manipulação de:
    * objetos complexos, através dos relacionamentos de composição e das operações que atualizam esses relacionamentos;
    * equivalência, relacionando um ou mais objetos equivalentes;
    * derivação evolutiva, implementada pela classe Objeto Genérico, através do vetor de derivação e de operações que manipulam o relacionamento de derivação.

6.4 Contribuições da Dissertação

Este trabalho abrange um vasto espectro de modelos de transações. Foram abordados, desde mecanismos convencionais de controle de concorrência e recuperação, utilizados nos modelos tradicionais, até mecanismos mais flexíveis que suportam aplicações avançadas. Foram estudados também alguns mecanismos de gerenciamento de versões e configurações. A necessidade desse estudo adveio da forte interação entre tais mecanismos e os mecanismos de transações.

Uma comparação dos vários modelos não convencionais foi apresentada na seção 5.5.

Além disso, foram descritas as principais características de algumas aplicações avançadas. Essa descrição resultou na apresentação dos requisitos dessas aplicações. Visando atender esses requisitos, elaboramos um modelo integrado de gerenciamento de transações e versões.

Apesar de o modelo proposto ser conceitualmente baseado em modelos já descritos na literatura, contribuímos com os seguintes itens:

- integramos idéias distintas num único modelo;
- unimos três estilos de gerenciamento de transações num modelo, permitindo que a aplicação escolha o gerenciador que melhor se adapta a suas características;

- a cooperação foi flexibilizada com as operações de concessão, empréstimo e cópia. Essa última foi originalmente concebida na nossa proposta. Apesar da operação de concessão já ter sido proposta no modelo de Klahold [KSUW85], essa foi adaptada à hierarquia de vários níveis;
- a incorporação de mais uma dimensão às trancas para permitir cooperação entre transações;
- a abordagem orientada a objetos forneceu maior clareza na apresentação do gerenciamento de transações e versões e dos relacionamentos entre as diversas entidades do modelo.

6.5 Trabalhos Futuros

O tema abordado nessa dissertação se interrelaciona com muitas áreas. Várias extensões em cada uma dessas áreas poderiam ser propostas:

- **Comunicação de grupos.**
  - Definição de grupos de trabalho. É necessário estudar e definir formas adequadas de organização, atribuindo papéis e responsabilidades aos usuários, assim como definindo operações para criar e retirar membros de um grupo [CZ85, BSS91, Bir93].
- **Interação síncrona.**
  - O modelo proposto fornece interação assíncrona entre usuários através da transferência de objetos entre áreas de trabalho. Contudo, algumas aplicações exigem suporte à interação síncrona. Esse tópico relaciona-se à definição de uma interface e de um sistema de notificação. Abrange conceitos de CSCW[1].
- **Controle de concorrência.**
  - Estender o controle de concorrência e os tipos de trancas para permitir múltiplas derivações em paralelo, incorporando assim, versões alternativas.
  - Estudar maneiras de aproveitar os relacionamentos entre os objetos da aplicação, a fim de auxiliar na manutenção consistência e no gerenciamento da concorrência. Uma idéia a ser avaliada é a apresentada em [NG92].

[1] *Computer Supported Cooperative Work.*

- **Recuperação.**
  - Estender a classe Transações Longas para conter recuperação por compensação. Tal extensão deve ser primeiramente avaliada a fim de se verificar a possibilidade da coexistência dos diferentes tipos de transações com o processo de compensação.
- **Distribuição.**
  - Estudar, avaliar e propor extensões para a adaptação do modelo proposto num contexto distribuído.
- **Implementação.**
  - Detalhar os modelos de transações, versões e configurações apresentados a fim de, posteriormente, implementá-los. Um ambiente adequado seria a arquitetura CORBA [VA93, Vin93], baseada no paradigma de objetos para desenvolvimento de aplicações distribuídas.

Bibliografia

[Arm93] Marc P. Armstrong. Perspectives on the Development of Group Decision Support Systems for Locational Problem-Solving. *Geographical Systems*, 1:69–81, 1993.

[Arm94] Marc P. Armstrong. Requirements for Development of GIS-based Group Decision-Support Systems. *Journal of the Americam Society for Science*, pp. 659–667, 1994.

[Aro89] Stan Aronoff. *Geografic Information Systems: A Mangement Perspective.* WDL Publications, 1989.

[ASU88] A. Aho, R. Sethi, e J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1988.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, e Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Whesley Publishing Company, 1987.

[BHR80] R. Bayer, H. Heller, e A Raiser. Parallelism and Recovery in Database System. *ACM Transaction on Database Systems*, 5(2):139–156, junho de 1980.

[Bir93] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, dezembro de 1993.

[BKK85] F. Bancilhon, Won Kim, e Henry F. Korth. A Model of CAD Transactions. In *Proc. of 11th VLBD, Stockholm*, pp. 25–33, 1985.

[BOH+94] Alejandro Buchmann, M. Tamer Özsu, Mark Hornick, Dimitrios Georgakopoulos, e Frank A. Manola. A Transaction Model for Active Distributed Object Systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pp. 123–158, 1994.

[BSS91] K. Birman, A. Schiper, e P. Sephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, agosto de 1991.

[BSW88] C. Beeri, H. J. Schek, e G. Weikum. Multi-Level Transaction Management, Theoretical Art or Pratical Need? *Lecture Notes in Computer Science*, 303:134–154, 1988.

[Cif95] Ricardo Rodrigues Ciferri. Um Benchmark Voltado à Análise de Desempenho de Sistemas de Informações Geográficas. Master's thesis, UNICAMP-IMECC-DCC, 1995.

[CJ90] W. Cellary e G. Jomier. Consistency of Versions in Object-Oriented Databases. In *Proc. 16th VLDB*, pp. 432–441, 1990.

[CJ94] Wojciech Cellary e Geneviève Jomier. Apparent Versioning and Concurrency Control in Object-Oriented Databases. *In Proc. of the International Conference on Computing and Information*, pp. 1–19, 1994.

[CP85] Stefano Ceri e Giuseppe Pelagatti. *Distributed Database. Principles and Systems.* McGrawhill, 1985.

[CVJ94] W. Cellary, G. Vossen, e G. Jomier. Multiversion Object Constellations: A New Approach to Suport a Designer's Database Work. *Engineering with Computer. Spring Verlag*, pp. 230–244, 1994.

[CZ85] D.R. Cheriton e W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, maio de 1985.

[Dat88] C. J. Date. *Banco de Dados: Tópicos Avançados.* Editora Campus LTDA, 1988.

[Dav78] C. T. Davies. Data Processing Spheres of Control. *IBM Systems Journal*, 17(2):179–198, 1978.

[Dit87] Klaus R. Dittrich. Controlled Cooperation in Engineering Database Systems. pp. 510–515, 1987.

[DL88] Klaus R. Dittrich e Raymond A. Lorie. Version Support for Engineering Database Systems. *IEEE Transactions on Software Engineering*, 14(4):429–437, abril de 1988.

[EGR91] C. A. Ellis, S. J. Gibbs, e G. L. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, 1(34):39–58, janeiro de 1991.

[FZ89] Mary F. Fernandez e Stanley B. Zdonik. Transaction Groups: A Model for Controlling Cooperative Transactions. *In Proc. of third International Workshop on Persistent Object Systems: Their Designs, Implementation and Use.*, pp. 128–138, janeiro de 1989.

[GK88] Jorge F. Garza e Won Kim. Transaction Management in an Object-Oriented Database System. In *Proc. of the ACM SIGMOD Conference*, pp. 37–45, 1988.

[GM83] Hector Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, junho de 1983.

[GMS87] H. Garcia-Molina e K. Salem. Sagas. *In ACM SIGMOD*, 16(3):249–259, 1987.

[GR93] Jim Gray e Andreas Reuter. *Transactions Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[HFC93] Andrea S. Hemerly, Antônio L. Furtado, e Marco A. Casanova. Towards cooperativeness in geografic databases. In *Lecture Notes in Computer Science*, volume 720, pp. 373–376, 1993.

[HR82] Gary S. Ho e C. V. Ramamoorthy. Protocols for Deadlock Detection in Distributed Database Systems. *IEEE Transactions on Software Engineering*, 8(6), novembro de 1982.

[HZ87] M. F. Hornic e S. B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1):70–95, janeiro de 1987.

[Kat85] R. H. Katz. *Information Management for Engineering Design.* Surveys in Computer Science. Springer-Verlag, 1985.

[Kat90] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, dezembro de 1990.

[KBC+87] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, e Darrell Woelk. Composite Object Support in an Object-Oriented Database System. In *Proc. of the 2nd OOPSLA.*, pp. 118–125, Orlando, outubro de 1987.

[KBC+89] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, e Darrell Woelk. Features of the ORION Object-Oriented Database System. *In W. Kim, H.*

F. Lochovsky (ed). Object-Oriented Concepts, Databases, and Applications ACM Press cap 11, pp. 251–282, 1989.

[KBGW91] Won Kim, Nat Ballou, Jorge F. Garza, e Darrell Woelk. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. *ACM Transaction on Information Systems*, 9(1):31–51, janeiro de 1991.

[KSUW85] P. Klahold, G. Schlageter, R. Unland, e W. Wilkes. A Transaction Model Suporting Complex Applications in Integrated Information Systems. In *ACM SIGMOD*, pp. 388–401, maio de 1985.

[MJ94] C. B. Medeiros e G. Jomier. Using Versions in GIS. In *International DEXA Conference*. Spring Verlag Lecture Notes in Computer Science, 1994.

[ML83] C. Mohan e B. Lindsay. Efficient Commit Protocols for Tree of Processes Model of Distributed Transactions. In *2nd ACM SIGACT/SIGOPS. Sysmposium on Principles of Distributed Transactions*, pp. 1–13, Montreal, Canada, agosto de 1983.

[Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, abril de 1981.

[Mos82] J. E. B. Moss. Nested Transactions and Reliable Distributed Computing. In *The 2nd Symp. on Reliability in Distributed Software and Database Systems*, pp. 33–39, julho de 1982.

[NG92] K. Narayanaswamy e Neil Goldman. "Lazy" Consistency: A Basis for Cooperative Software Development. In *CSCW Proceedings*, pp. 257–264, novembro de 1992.

[NRZ94] Marian H. Nodine, Sridhar Rasmaswamy, e Stanley B. Zdonik. A Cooperative Transaction Model for Design Databases. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pp. 53–85, 1994.

[Obe82] Ron Obermarck. Distributed Deadlock Detection Algorithm. *ACM Transactions on Database Systems*, 2(7):187–208, junho de 1982.

[PK84] Christos H. Papadimitriou e Paris C. Kanellakis. On Concurrency Control by Multiple Versions. *ACM Transactions on Database Systems*, 9(1):89–99, março de 1984.

[PKH88] Calton Pu, Gail E. Kaiser, e N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proc. of the 14th VLDB Conference*, pp. 26–37, Los Angeles, California, agosto de 1988.

[RBP+94] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, e William Lorensen. *Modelagem e Projetos Baseados em Objetos.* Editora Campus, 1994.

[Ree79] D. P. Reed. Implementing Atomic Actions. In *Proc. 7th ACM SIGOPS Symp. on Operating Systems Principles*, dezembro de 1979.

[RP90] Kurt Rothermél e Stefan Pappe. Open Commit Protocols for the Tree Processes Model. In *The 10th International Conference on Distributed Computing Systems*, maio de 1990.

[RSI78] Daniel J. Rosenkrantz, Richard E. Stearns, e Philip M. Lewis II. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 2(3):178–198, junho de 1978.

[SGMA89] Kenneth Salem, Hector Garcia-Molina, e Rafael Alonso. Altruistic Locking: A Strategy for Coping with Long Lived Transactions. *Lecture Notes in Computer Science*, 359:175–198, 1989.

[Ska89] A. H. Skarra. Concurrency Control for Cooperating Transactions in an Object-Oriented Database. *SIGPLAN Notices*, 24(4):145–147, abril de 1989.

[SW90] S. K. Shrivastava e S. M. Wheater. Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions. *ICDCS-10. Paris.*, pp. 1–9, junho de 1990.

[SZ89] A. H. Skarra e S. B. Zdonik. Concurrency Control and Object Oriented Databases. *In W. Kim and F. H. Lochovsky, Object-Oriented Concepts, Databases and Applications. Cap. 16*, pp. 395–421, 1989.

[Unl94] Rainer Unland. TOPAZ: A Tool Kit for the Assembly of Transaction Managers for Non-standard Applications. Technical report, University of Münster, Germany, Institute of Business Informatics, novembro de 1994.

[US94] Rainer Unland e Gunter Schlageter. A Transaction Manager Development Facility for Non Standard Database Systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pp. 53–85, 1994.

[VA93] F. Vogt e C. Andrae. Middleware for Distributed Applications Support: ODP and/or CORBA. *International Conference on Open Distributed Processing*, 1993.

[Vin93] S. Vinoski. Distributed Object Computing with CORBA. *C++ Report*, julho de 1993.

[Wal84] Bernd Walter. Nested Transactions with Multiple Commit Points: An Approach to the Structuring of Advanced Database Applications. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pp. 161–171, Singapore, agosto de 1984.

[WR94] Helmut Wächter e Andreas Reuter. The ConTract Model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for advanced Applications*, pp. 219–263, 1994.