UNIVERSIDADE CATÓLICA DE PELOTAS
CENTRO POLITÉCNICO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Uma Linguagem de Domínio Específico para Programação de Memórias Transacionais em Java

por
Marcos Gonçalves Echevarria

Dissertação apresentada como
requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Orientador: Prof. Dr. Adenauer Corrêa Yamin
Co-orientador: Prof. Dr. André Rauber Du Bois

DM-2010/1-005

Pelotas, abril de 2010

Dedico este trabalho a minha família.

As pessoas mudam através do que alcançam. — WYSTAN AUDEN.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BGGA	Extensão Java que suporta *funções anônimas* e *closures*
CAS	*Compare-And-Swap*
HTM	*Hardware Transactional Memory*
JIT	*Just-In-Time*
JVM	*Java Virtual Machine*
STM	*Software Transactional Memory*
TCC	*Transactional Memory Coherence and Consistency*
TL2	*Transactional Locking II*
TM	*Transactional Memory*

RESUMO

As arquiteturas multi-core influenciam diretamente no desenvolvimento de software. Para que os programas possam tirar proveito dessas arquiteturas é necessário que estes possuam várias atividades concorrentes e que possam ser alocadas aos *cores* disponíveis. Programas concorrentes para máquinas multi-core são geralmente implementados usando threads e se comunicam através de uma memória compartilhada. Para evitar que threads interfiram de maneira errada no trabalho de outras threads, as linguagens fornecem mecanismos de sincronização, como por exemplo bloqueios. Mas sincronizações baseadas em bloqueios apresentam algumas armadilhas que dificultam a programação e são propensas a erros (PEYTON JONES, 2007; HERLIHY; MOSS, 1993).

Memórias transacionais fornecem um novo modelo de controle de concorrência que não apresenta as mesmas dificuldades encontradas no uso de bloqueios. Elas trazem para a programação concorrente os conceitos de controle de concorrência usados hà décadas pela comunidade de banco de dados. Construções em linguagens transacionais são fáceis de serem usadas e podem gerar programas altamente escaláveis (ADL-TABATABAI; KOZYRAKIS; SAHA, 2006).

O objetivo deste trabalho é apresentar a linguagem CMTJava. CMTJava é uma linguagem de domínio específico para programação de memórias transacionais em Java e foi criada visando facilitar a programação de máquinas multi-core. Seu sistema foi todo desenvolvido em Java e a forma de implementação pode ser aplicada em qualquer outra linguagem orientada a objetos que suporte *closures*, como por exemplo C#.

CMTJava faz uso das chamadas mônadas para compor ações transacionais. Ações transacionais podem ser combinadas para gerar novas transações e vale ressaltar que o sistema de tipos da linguagem CMTJava garante que ações transacionais somente serão executadas através da primitiva `atomic`. CMTJava apresenta todas as construções de memórias transacionais (`atomic`, `retry`, `OrElse`) e é a primeira extensão Java para transações que suporta a construção OrElse.

Palavras-chave: Memórias Transacionais, Linguagem de Domínio Específico, Programação Concorrente, Linguagem Java.

TITLE: "A DOMAIN SPECIFIC LANGUAGE FOR COMPOSABLE MEMORY TRANSACTIONS IN JAVA"

ABSTRACT

The multi-core architecture directly influences software development. The revolution caused by these architectures is a revolution in software development and not in hardware as it forces programmers to write parallel programs.

For programs to take advantage of multi-core architectures they must have concurrent activities that can be allocated to the available cores. Concurrent programs are usually implemented using threads that communicate through a shared memory. To prevent threads from interfering in the work of other threads, languages provide mechanisms for synchronization such as locks. However, synchronization based on locks has some traps that make programming harder and error prone.

Transactional memory provides a new model for concurrency control that does not presents the same problems encountered in the use of locks. It brings to the concurrent programming concepts used for a long time in databases. Transactional languages are easy to use and generate highly scalable programs.

The purpose of this work is to present the CMTJava. CMTJava is a domain specific language for composable memory transactions in Java. Although the system was implemented in Java, the ideas could also be applied to any object oriented language that supports closures.

Transactions in CMTJava are implemented as a state passing monad. Most of the simplicity of the implementation comes from the fact that we use a Java extension for closures to implement CMTJava. Transactional actions can only be executed by the `atomic` method and they are composable: transactions can be combined to generate new transactions. CMTJava supports all of the Transactional Memory constructs (`atomic`, `retry`, `OrElse`) and it is the first Java extension to provide the OrElse construct.

Keywords: Transactional Memory, Domain Specific Language, Concurrent Programming, Java.

1 INTRODUÇÃO

Durante as últimas décadas, o desempenho alcançado pelos microprocessadores cresceu exponencialmente. Esse crescimento deveu-se principalmente ao avanço da tecnologia de fabricação de chips, que possibilitou a construção de transistores menores e mais rápidos, além de permitir a adição de mais funcionalidades à microarquitetura. A partir do começo do século 21 esse panorama vem se alterando. Embora o processo de fabricação ainda possibilite transistores menores, mais rápidos e em maior quantidade, seguindo a lei de Moore, fatores novos estão limitando o crescimento do desempenho dos microprocessadores: aumento da energia dissipada, limite na extração de paralelismo no nível de instruções e complexidade de projeto. A tendência atual é desenvolver projetos mais simples, com freqüência de operação mais baixa, e integrar em um mesmo chip dois ou mais núcleos de processamento, ou seja, hoje tem predominado o projeto de processadores multi-core (também conhecidos como *single chip multiprocessors*).

É importante notar que, até então, o aumento da velocidade de operação dos microprocessadores implicava um aumento proporcional no desempenho do software existente. Com os processadores multi-core isso não é mais verdade. Para se explorar o potencial desta arquitetura, as aplicações devem ser concorrentes/paralelas, de modo que existam tarefas para serem distribuídas entre os cores.

Programar um computador paralelo é mais difícil que programar um computador seqüencial, principalmente pelo desafio de construir um código correto e seguro (PEYTON JONES, 2007). Erros não determinísticos que podem ocorrer durante sua execução são mais difíceis de serem detectados e corrigidos. Deste modo, a revolução causada por essas arquiteturas é na verdade uma revolução na área de software e não apenas na de hardware porque força os desenvolvedores de software a escrever aplicações paralelas que possam tirar proveito do incremento no número de processadores que cada nova geração multi-core irá prover.

Tradicionalmente máquinas multi-core são programadas usando um modelo de memória compartilhada, onde threads concorrentes se comunicam através de uma área de memória em comum. Acessos às mesmas posições de memória precisam ser sincronizados para evitar interferências entre as threads. O mecanismo mais usado hoje para fazer essa sincronização são os bloqueios (*locks*).

As Memórias Transacionais surgiram como uma alternativa aos métodos baseados em bloqueios, usando uma nova abstração para programação concorrente baseada na idéia de transações. Uma *transação de memória* é uma seqüência de operações que modificam a memória e que podem ser executadas completamente ou podem ter nenhum efeito (*podem ser abortadas*) (HERLIHY; MOSS, 1993). O uso de memórias transacionais facilita

a programação concorrente porque o programador não precisa se preocupar em garantir a sincronização como nas abordagens baseadas em bloqueios. Todo o controle de acesso à memória compartilhada é feito automaticamente pelo sistema transacional.

O objetivo geral deste trabalho consiste em facilitar a programação de máquinas multi-core. Para alcançar esse objetivo, foi desenvolvida uma extensão para a linguagem orientada a objetos Java para programação concorrente usando memórias transacionais, denominada CMTJava (DU BOIS; ECHEVARRIA, 2009). CMTJava foi desenvolvida adaptando a linguagem STM Haskell (HARRIS et al., 2008) para um contexto orientado a objetos. As principais caraterísticas da CMTJava são:

- CMTJava provê a abstração de *objetos transacionais*. Os atributos dos objetos transacionais somente podem ser acessados por métodos especiais de `get` e `set` que são introduzidos automaticamente pelo compilador. O sistema de tipos da linguagem garante que os atributos de um objeto transacional somente podem ser acessados dentro de um método `atomic`. O método `atomic` recebe uma transação como argumento e a executa atomicamente, concorrentemente com as outras transações do sistema. Como transações não podem ser executadas fora de um método `atomic`, propriedades como atomicidade (a execução de uma transação somente terá efeito se ela for efetivada, caso ela seja abortada nenhuma alteração terá efeito) e isolamento (transações não vêem resultados intermediários produzidos por outras transações) sempre serão mantidas.

- CMTJava apresenta todas as construções de memórias transacionais (*atomic*, *retry*, *OrElse*) e é a primeira extensão Java para transações que suporta a construção *OrElse*.

- Transações são implementadas como uma mônada de passagem de estados (DU BOIS; ECHEVARRIA, 2009). Muito da simplicidade de sua implementação vem do fato de que a CMTJava usa uma extensão para *closures* (JAVA CLOSURES, 2009). Algumas *regras de tradução* são usadas para traduzir a CMTJava para Java puro + *closures*. Outro detalhe importante é que mesmo o sistema sendo desenvolvido em Java, as idéias apresentadas podem ser implementadas em qualquer outra linguagem orientada a objetos que suporte *closures*, como por exemplo C#.

Mais especificamente, este trabalho busca:

- Realizar uma revisão bibliográfica do assunto, apresentado os conceitos de memórias transacionais;

- Desenvolver a linguagem CMTJava, uma extensão da linguagem Java para programação concorrente utilizando memórias transacionais;

- Desenvolver dois protótipos para implementar a linguagem CMTJava;

- Realizar testes com os dois protótipos desenvolvidos para avaliação do desempenho dos mesmos;

1.1 Estrutura do trabalho

Este trabalho está organizado da seguinte forma. No Capítulo 2, são apresentados os conceitos de memórias transacionais. Nesse capítulo são discutidos os problemas na programação concorrente, bem como as vantagens do uso de memórias transacionais para esse tipo de programação. Além disso, são apresentados os modelos de memórias transacionais existentes.

O Capítulo 3 mostra alguns trabalhos relacionados com a linguagem CMTJava. Esses trabalhos são extensões da linguagem Java para memórias transacionais. Elas estendem a linguagem Java com novas construções para suportar memórias transacionais. O objetivo é apresentar o modo como cada uma é implementada e como se programa utilizando essas extensões.

O Capítulo 4 apresenta a linguagem CMTJava. A linguagem é mostrada através de alguns exemplos com o objetivo de mostrar como se programa utilizando a linguagem.

O Capítulo 5 apresenta dois protótipos desenvolvidos para implementar a linguagem CMTJava. Além disso, são apresentados alguns testes realizados com os dois protótipos de modo a fazer uma comparação entre o desempenho de cada um deles.

Por fim, no Capítulo 6 encontram-se as conclusões do trabalho. Nesse capítulo são apresentados também os trabalhos futuros.

2 MEMÓRIAS TRANSACIONAIS

O termo "Memória Transacional" (*Transactional Memory*) surgiu em 1993, definido como "uma nova arquitetura para multiprocessadores que objetiva tornar a sincronização livre de bloqueios tão eficiente (e fácil de usar) quanto técnicas convencionais baseadas em exclusão mútua" (HERLIHY; MOSS, 1993). Esse termo define qualquer mecanismo de sincronização que utilize o conceito de transação para coordenar acessos à memória compartilhada.

Este capítulo apresenta o conceito de memórias transacionais. Primeiramente, na seção 2.1 são discutidos os problemas na programação concorrente sem sincronização e os problemas surgidos com a sincronização baseada em bloqueios. Na seqüência, a seção 2.2 mostra as vantagens do uso de memórias transacionais em relação ao uso de bloqueios. A seção 2.3 apresenta o que são transações e como elas funcionam. A seção 2.4 mostra algumas construções transacionais básicas. A seção 2.5 mostra o aninhamento de transações e as diversas formas de se fazer aninhamento. A seção 2.6 apresenta alguns requisitos de implementação para a abstração de memórias transacionais e como o modo de implementação de cada requisito pode alterar o modelo de programação e até mesmo sua performance. Finalmente, na seção 2.7 são apresentados os modelos de memória transacional existentes.

2.1 Problema com o uso de bloqueios

Um modelo concorrente de programação deve oferecer abstrações para a especificação, comunicação e sincronização de atividades concorrentes. O modelo mais utilizado atualmente é o *multi-threaded*, onde a thread é a unidade de concorrência, a comunicação é feita via memória compartilhada e a sincronização é baseada em bloqueios.

Threads são uma extensão da programação seqüencial que permite linhas de execução simultâneas em um programa. Nos programas tradicionais, existe apenas uma linha de execução definida pela função principal do programa, como a função `main()` em C e C++. Um programa com várias threads é mais ou menos como um programa que possui vários `main()` executando ao mesmo tempo. Linguagens de programação precisam de poucas modificações para suportar threads. O hardware e sistemas operacionais disponíveis hoje em dia possuem em geral um bom suporte para a implementação desse tipo de abstração, mas programar utilizando este recurso, quando existe a necessidade de comunicação e sincronização entre as threads, não é tão fácil.

Um programa concorrente é composto de duas ou mais threads. Cada thread

consiste de uma seqüência de instruções, executadas seqüencialmente. A execução de um programa concorrente pode ser visto como uma intercalação das instruções executadas pelas suas threads. Desta forma, a execução de um programa é válida se todas as suas intercalações são válidas. Threads se comunicam por meio de uma memória compartilhada, efetuando operações de leitura e escrita. Quando múltiplas threads acessam uma mesma faixa de endereços de memória, é possível que alguma intercalação de suas instruções resulte em uma execução inválida.

Considere o código abaixo como exemplo:

```
int c;     // Variável compartilhada
(...)
temp = c;
temp++;
c = temp;
(...)
```

A variável `c` é compartilhada entre as threads, enquanto a variável `temp` é local. O trecho de código mostrado acima incrementa o valor da variável `c` em `1`. Supondo que a variável `c` tenha o valor `zero` e que duas threads `t1` e `t2` estejam executando o mesmo código, pode-se ter uma simulação da possível execução passo a passo dessas duas threads na Tabela 2.1.

Tabela 2.1: Simulação de execução concorrente de duas threads sem sincronização

Thread 1 (t1)	**Thread 2 (t2)**
temp = c;	-
-	temp = c;
-	temp++;
-	c = temp;
temp++;	-
c = temp;	-

Analisando a tabela nota-se que exatamente quando a thread `t1` acabou de executar a instrução `temp = c`, a thread `t2` executou as instruções `temp = c`, `temp++` e `c = temp`. Quando a thread `t2` terminou sua execução o valor da variável `c` ficou igual a `1`, mas quando a thread `t1` chamou a instrução `temp = c` ela recebeu o valor `zero`. Então, a thread `t1` ao retornar sua execução vai incrementar `zero`, terminando sua execução com `c = 1`. Como duas threads passaram pelo código, `c` deveria ser incrementada duas vezes, terminando a computação com o valor `2`. No entanto, repare que `c` foi incrementada apenas uma vez nesta seqüência de operações e o valor real de `c` após a computação é 1. Essa intercalação na execução do código por duas threads tornou o estado do programa inconsistente.

O objetivo da sincronização é evitar intercalações indesejáveis entre instruções de threads concorrentes. Há basicamente duas formas de sincronização: exclusão mútua (*mutual exclusion*), que permite que um conjunto de instruções seja executada de forma atômica; e sincronização de condição (*condition synchronization*), que permite postergar a execução de uma thread até que alguma condição seja satisfeita. Essas duas formas

são suficientes para evitar qualquer tipo de interferência entre threads concorrentes (ANDREWS, 1991).

A sincronização tem tradicionalmente sido feita com o uso de bloqueios (*locks*). De forma geral, um bloqueio é uma variável booleana que aceita as operações LOCK e UNLOCK. A operação LOCK altera atomicamente a variável de falsa para verdadeira. Caso a variável já seja verdadeira antes da alteração, a operação deve primeiro esperar que ela se torne falsa. A operação UNLOCK atribui, incondicionalmente, falso à variável.

O código abaixo mostra como o problema descrito acima pode ser resolvido através do uso de bloqueios, garantindo que as instruções entre LOCK e UNLOCK sejam executadas de maneira atômica. Nesse caso, somente uma thread em um determinado instante terá acesso à variável `c`.

```
int c;     // Variável compartilhada
(...)
bool b;    // Lock
(...)
LOCK(&b);
  temp = c;
  temp++;
  c = temp;
UNLOCK(&b)/
(...)
```

Tendo como base este código, pode-se ter uma simulação da possível execução passo a passo dessas duas threads, usando sincronização baseada em bloqueios, na Tabela 2.2.

Tabela 2.2: Simulação de execução concorrente de duas threads com sincronização

Thread 1 (t1)	**Thread 2 (t2)**
LOCK(&b);	-
temp = c;	-
-	Não consegue adquirir o lock
temp++;	bloqueada
c = temp;	bloqueada
UNLOCK(&b);	bloqueada
-	LOCK(&b);
-	temp = c;
-	temp++;
-	c = temp;
-	UNLOCK(&b);

Vários mecanismos com níveis de abstrações mais altos foram desenvolvidos durante os últimos 40 anos, como semáforos e monitores. Este trabalho não tem como objetivo estudar a fundo estes mecanismos, mas vale ressaltar que todos eles usam bloqueios da mesma forma para implementar sincronização e, portanto, compartilham dos mesmos problemas apresentados a seguir (RAJWAR; GOODMAN, 2003; LEE, 2006; PEYTON JONES, 2007):

- **Dificuldade de Programação**: Programar usando bloqueios é complicado. É comum acontecerem erros como utilizar *locks* em excesso, o que acaba com a concorrência ou pode gerar um bloqueio eterno do programa (*deadlock*). Adquirir menos *locks* do que o necessário pode permitir que mais de uma thread entre na região crítica. Outro erro é adquirir os *locks* errados, erro de programação comum pois pode ser difícil de perceber quais os dados que estão sendo protegidos pelo *lock*. Ou adquirir os *locks* na ordem errada: os *locks* devem ser adquiridos na ordem correta para evitar *deadlock*. Programar usando *locks* é tão baixo nível quanto programar em *assembly*, porém é ainda mais difícil de depurar pois todos os erros possíveis como condições de corrida e *deadlocks* são imprevisíveis e de difícil reprodução (TANENBAUM; WOODHULL, 2006). Por isso é muito difícil de se criar sistemas que sejam confiáveis e escaláveis;

- **Dificuldade de Reuso de Código**: Trechos de código corretamente implementados usando bloqueios, quando combinados, podem gerar erros. No exemplo mostrado anteriormente, tínhamos um método, corretamente implementado usando bloqueios, onde a variável $\subset$ era incrementada. Porém, se o programador resolve incrementar duas vezes a variável $\subset$ usando esse método, não podemos garantir que o resultado estará correto. Como o método libera o lock depois de ser chamado a primeira vez, alguma thread pode se aproveitar da oportunidade e realizar alguma operação que modifique a variável $\subset$ antes desse método ser chamado pela segunda vez.

- **Gargalo Serial**: Uma região crítica protegida por bloqueios permite o acesso de apenas uma thread de cada vez. Quando uma thread tenta acessar uma região de código protegida que já está sendo acessada, essa thread fica bloqueada até a outra liberar o bloqueio causando gargalos seriais nos programas. Um erro comum de programação é criar grandes regiões de código protegidas por bloqueios, o que reduz o paralelismo do programa. Porém, quando se trabalha com regiões menores, o código de sincronização fica mais difícil de se escrever e mais sujeito a erros.

2.2 Vantagens do uso de memórias transacionais

A Memória Transacional (MT) surgiu como alternativa aos métodos baseados em bloqueios, oferecendo as seguintes vantagens (RAJWAR; GOODMAN, 2003):

- **Facilidade de programação**: A programação torna-se mais fácil porque o programador não precisa se preocupar em como garantir a sincronização, e sim em especificar o que deve ser executado atomicamente. Na sincronização por bloqueios, o programador inicialmente precisa criar uma associação entre variáveis e bloqueios e, posteriormente, garantir que todo acesso a uma determinada variável seja protegido pelo seu respectivo bloqueio. Além disso, a fim de evitar *deadlock*, o programador deve atentar para o ordem em que os bloqueios são adquiridos. Com Memória Transacional, basta especificar o trecho de código que deve ser executado atomicamente e o sistema de execução garante a sincronização. Ou seja, a responsabilidade pela sincronização passa do programador para a entidade que implementa as transações;

- **Escalabilidade**: Transações que acessem um mesmo dado para leitura podem ser executadas concorrentemente. Também podem ser executadas em paralelo as transações que modifiquem partes distintas de uma mesma estrutura de dados. Essa característica permite que mais desempenho seja obtido com o aumento do número de processadores porque o nível de paralelismo exposto é maior. Já com bloqueios, o mesmo desempenho só pode ser obtido através de técnicas sofisticadas como a criação de bloqueios de leitura e escrita e fatoração do código. Essas técnicas, no entanto, possuem alta complexidade de implementação e introduzem erros difíceis de serem identificados, como *deadlock*;

- **Composabilidade**: Transações suportam naturalmente a composição de código. Para criar uma nova operação com base em outras já existentes, basta invocá-las dentro de uma nova transação. Novamente, o sistema de execução garante que as operações sejam executadas de forma atômica.

2.3 Transações

Enquanto o paralelismo tem sido um problema difícil para a programação em geral, os sistemas de bancos de dados tem tido sucesso na exploração do mesmo ao longo dos anos. Esses sistemas alcançam uma boa performance em computações concorrentes executando várias *queries* simultâneas.

O conceito de transação é fundamental nos sistemas de banco de dados. O ponto essencial de uma transação é englobar várias ações em uma única operação de tudo ou nada. Ou todas as ações são realizadas com sucesso, ou nenhuma ação é realizada. Vale ressaltar que os estados intermediários entre essas ações não são vistos pelas demais transações.

O advento dos processadores multi-core fortaleceram o interesse na idéia de incorporar o conceito de transações no modelo de programação usado para a construção de programas paralelos. Enquanto transações em linguagens de programação apresentam certas semelhanças com transações de bancos de dados, a implementação e execução das mesmas ocorre de forma muito diferente porque bancos de dados armazenam os dados em discos e programas armazenam os dados em memória. Essa diferença deu origem a uma nova abstração chamada de memórias transacionais.

Uma transação é uma sequência de ações executadas sequencialmente por uma thread que aparecem de forma indivisível e instantânea para um observador externo. Uma transação possui as seguintes propriedades:

- **Atomicidade**: Ou todas as instruções da transação são executadas ou nenhuma é executada. No primeiro caso, diz-se que a transação sofreu efetivação (*commit*) e o resultado da computação é permanente. No segundo caso, diz-se que a transação sofreu cancelamento (*abort*) e qualquer valor parcialmente computado é invalidado;

- **Isolamento**: Transações concorrentes não vêem resultados intermediários produzidos por outras transações. Ou seja, o resultado da execução de diversas transações concorrentes equivale ao resultado da execução dessas mesmas transações em alguma ordem serial;

2.4 Construções Transacionais Básicas

Em um programa paralelo exitem várias threads rodando simultaneamente, isso faz com que seja necessário um controle de concorrência para previnir que uma thread não interfira no trabalho de outra quando acessa dados de uma área compartilhada.

A coordenação é outra razão para controlar a concorrência. Um programa pode necessitar que duas threads sejam executadas em uma ordem específica. Então, por exemplo, a thread 2 deve executar depois da thread 1. Sem coordenação, as threads podem rodar de forma independente e a thread 1 pode terminar ao mesmo tempo ou até mesmo depois da thread 2. Memória transacional provê um mecanismo de controle de concorrência que controla esses dois aspectos.

Transações não são a única forma de controlar computações paralelas, mas muito do crescente interesse por memórias transacionais é conseqüência de uma crença generalizada de que as transações oferecem alto-nível e menor capacidade de introduzir erros no modelo de programação paralela que algumas alternativas conhecidas como *mutexes*, bloqueios, semáforos, monitores, etc (LARUS; RAJWAR, 2006).

Para fazer uso das memórias transacionais, de alguma forma deve-se indicar que um determinado objeto ou um determinado trecho e código fará parte de uma transação. Isso pode ser feito tanto de forma *explícita* quanto de forma *implícita*. Algumas propostas obrigam que o programador indique explicitamente quais objetos ou quais trechos do código serão partes de uma transação. Entretanto, outras propostas fazem isso de forma implícita. Para isso são usadas algumas construções transacionais como: bloco atômico, *retry* e *orelse*. Abaixo serão mostradas cada uma dessas construções separadamente.

2.4.1 Bloco Atômico

Um bloco `atomic {}` delimita um bloco de código que deve ser executado em uma transação:

```
atomic {
  if ( x != null)
    x++;
  y = true;
}
```

Um bloco atômico sempre executa garantindo as propriedades de atomicidade e isolamento de uma transação. Uma grande vantagem do bloco atômico é que não é necessário criar manualmente variáveis específicas para controlar cada bloco crítico de um programa, diferentemente de outras construções como bloqueios. Com isso, o programador se preocupa apenas em definir quais regiões do programa devem ser executadas em paralelo e a implementação de *atomic* é responsável por garantir a sincronização dos blocos. Já com o uso de outras construções, o programador é que deveria fazer esta sincronização.

Quando as transações são executadas, o programador tem uma ilusão de que existe um único bloco de execução onde as transações estão sendo executadas de forma sequencial. Claro que essa ilusão não pode ser verdade porque isso acabaria com a concorrência do programa. O que acontece realmente é que o sistema de execução da memória transacional executa as diversas threads ao mesmo tempo, respeitando as propriedades de

atomicidade e isolamento. Mais tarde nesse trabalho será apresentado como o sistema de execução da memória transacional é implementado para que respeite essas propriedades.

Na perspectiva do programador, uma transação pode estar em três estados possíveis. Ela pode estar em execução. Ela pode estar efetivada, o que faz com que seu resultado se torne visível para as demais threads em execução, ou ela pode estar abortada. Uma transação pode ser abortada tanto pelo programa, quanto pelo sistema de tempo de execução da memória transacional. Uma transação pode ser abortada pelo programa usando um comando do tipo retry (ver Seção 2.4.2) e se uma transação acessa um dado que conflita com outra transação, o sistem de tempo de execução aborta a mesma. Nos dois casos, o sistema aborta a transação e re-executa ela automaticamente na esperança de que o problema não volte a acontecer. Em geral, o processo de re-execução não é visível para o programador.

2.4.2 Retry

A construção *retry* faz com que uma transação seja cancelada e re-executada quando alguma variável associada à ela seja modificada por outra thread.

Considere a implementação de uma classe buffer em Java com os métodos para inserção e remoção de elementos, onde a inserção deve verificar se há espaço para inserção de novos elementos e, de forma análoga, a operação de remoção deve verificar se o buffer não está vazio:

```
public synchronized int remover() {
  int resultado;
  while (itens == 0)
    wait();
  itens--;
  resultado = buffer[itens];
  notifyAll();
  return resultado;
}
```

Observando o código acima, o uso do *synchronized* indica que o processo que chamar o método remover deverá obter o bloqueio associado ao objeto buffer, antes de prosseguir com a operação de remoção. A aquisição e liberação do bloqueio é transparente em Java quando o método é qualificado como *synchronized*. O método *wait* serve para garantir que não seja removido um elemento de um buffer vazio e o método *notifyAll* avisa os outros processos que um elemento foi removido do buffer.

Esse mesmo método *remover()* pode ser re-escrito, usando construções transacionais, da seguinte forma:

```
public int remover() {
  atomic {
    if (itens == 0)
      retry;
    itens--;
    return buffer[itens];
  }
}
```

Sempre que um buffer estiver vazio a transação chama `retry`, fazendo com que a transação seja cancelada e re-executada quando a variável `itens` for modificada por outro processo. Este exemplo também mostra um problema comum com bloqueios: a baixa concorrência. Como um bloqueio associado ao objeto buffer é adquirido ao se invocar `remover`, qualquer outro método do objeto que eventualmente seja invocado será bloqueado até que `remover` libere o bloqueio. Ou seja, duas ou mais operações não acontecerão concorrentemente no buffer mesmo quando há uma possibilidade de paralelismo. Por exemplo, é possível que uma operação de inserção ocorra simultaneamente a uma de remoção se ambas acessarem elementos disjuntos no buffer. Portanto o uso de transações consegue explorar mais paralelismo, já que a detecção de conflitos é feita de forma dinâmica (em tempo de execução).

2.4.3 OrElse

Outra construção das memórias transacionais é o *orElse*. Essa construção permite que transações sejam compostas como alternativas, ou seja, somente uma transação entre várias será executada. Por exemplo, uma chamada T1 orElse T2, primeiramente será tentada a execução da transação T1. Se T1 chamar *retry*, então T2 é executada e o resultado de toda computação feita por T1 será descartada. Caso T2 também chamar *retry*, então todo o bloco atômico será executado novamente. Caso T1 termine sua execução normalmente, T2 não será executada.

Agora, consideremos a seguinte situação: ler um valor entre dois buffers transacionais. Como visto na seção anterior, ao chamar o método remover da classe buffer a transação pode invocar *retry* caso o buffer esteja vazio. Uma implementação possível para essa situação seria usar a construção *orElse* para que se o buffer1 estiver vazio, então removemos do buffer2. Segue abaixo a implementação do exemplo:

```
atomic {
    { x = buffer1.remover(); }
  orElse
    { x = buffer2.remover(); }
}
```

Este código remove um elemento do buffer1. Caso o mesmo esteja vazio, o elemento será removido do buffer2. Por fim, caso os dois buffers estejam vazios, o bloco atômico é bloqueado até que pelo menos um dos buffers tenha um elemento e aí sim esse bloco será executado novamente.

2.5 Aninhamento de transações

Em memória transacional, o aninhamento de transações é essencial para a composição de código. Isso permite que novas transações possam ser criadas a partir de operações encapsuladas em bibliotecas transacionais. Uma transação T2 é dita aninhada quando é definida sob o contexto de uma transação T1, diz-se então que T1 é pai de T2 e T2 é filha de T1.

As transações podem ser aninhadas de três maneiras diferentes. A primeira maneira é conhecida como *flattened*. Nessa abordagem a transação filha é parte integrante da transação pai, como se ambas fossem uma única transação e nenhum aninhamento

existisse. Dessa forma, se a transação filha abortar a transação pai também aborta. Por exemplo, o código abaixo mostra uma variável x sendo incrementada, x inicia com o valor 1. Existe um bloco atômico que representa uma transação T1 e dentro dele um novo bloco atômico representando uma transação T2. Como a transação T2 aborta, T1 também irá abortar e a variável x no fim da execução continuará com o valor 1.

```
int x = 1;

atomic {
   x = 2;
   atomic {
      x = 3;
      abort;
   }
}
```

Transações *flattened* são fáceis de serem implementadas, mas essa abordagem tende a criar transações grandes, aumentando a probabilidade de conflitos e limitando a concorrência (LARUS; RAJWAR, 2006).

A segunda maneira é conhecida como *aninhamento fechado* (*closed nesting*). Nessa abordagem cada transação aninhada pode acessar os estados da transação pai. Desta forma, um conflito detectado durante a execução de uma transação filha não causa o cancelamento da transação pai, já que a filha pode ser cancelada e reiniciada de forma independente. Por exemplo, no código abaixo, ao fim da execução, a variável x terá o valor 2 porque o cancelamento da transação filha (T2) não irá alterar a transação pai (T1).

```
int x = 1;

atomic {
   x = 2;
   atomic {
      x = 3;
      abort;
   }
}
```

Com aninhamento fechado, quando uma transação filha é efetivada suas alterações e seus conjuntos de leitura e escrita são mesclados com as alterações e conjuntos de escrita da transação pai. Todos os dados alterados pela filha só se tornarão permanentes (visíveis para as outras transações) quando a transação pai for efetivada. Caso a transação pai seja abortada a execução da transação filha será descartada. O aninhamento fechado expõe mais paralelismo porque permite que transações aninhadas sejam canceladas e reiniciadas de forma independente.

A terceira maneira é conhecida como *aninhamento aberto* (*open nesting*). Essa abordagem expõe ainda mais paralelismo ao relaxar a propriedade de isolamento das transações. Sempre que uma transação filha é efetivada, suas alterações ficam visíveis para todas as threads mesmo que a transação pai ainda esteja em execução. Por exemplo, no fim da execução do código abaixo a variável x terá o valor 3 se a transação filha não for abortada porque mesmo a transação pai sendo abortada o resultado da transação filha se torna permanente.

```
int x = 1;

atomic {
   x = 2;
   atomic {
      x = 3;
   }
   abort;
}
```

Com aninhamento aberto é possível realizar resultados permanentes em transações mais internas, mesmo abortando as transações mais externas. A vantagem disso é que podemos executar códigos em transações internas que não possuem relação alguma com o código das transações mais externas. Um exemplo é um coletor de lixo que é executado no meio de uma transação e faz mudanças permanentes na memória.

A desvantagem do aninhamento aberto é que em determinadas circunstâncias a transação pai deve fornecer ações compensatórias. Por exemplo, se existe um contador compartilhado por várias transações e cada transação possui uma transação filha que incrementa esse contador, de modo que não haja conflitos no acesso a essa variável. Toda vez que uma transação pai falhar e esse contador já ter sido incrementado por sua transação filha, a transação pai deverá decrementar esse contador para que seu resultado não fique inconsistente.

2.6 Requisitos de implementação

As diferentes maneiras de implementação de memórias transacionais podem afetar diretamente o modelo de programação usado ou até mesmo o desempenho. Então, essa seção vai mostrar como um sistema de memória transacional consegue executar as transações, respeitando as propriedades de atomicidade e isolamento, e quais as diferentes maneiras de implementação que garantem essas propriedades.

Dois mecanismos chave para garantir as propriedades de atomicidade e isolamento são: versionamento de dados e detecção de conflitos. Para a detecção de conflitos é necessário que seja identificado qual o nível de detecção do conflito, chamado de granularidade do conflito. Outro parâmetro que deve ser levado em consideração é qual o nível de isolamento usado entre as transações. Todos esses conceitos são explicados a seguir.

2.6.1 Granularidade do conflito

A granularidade de detecção de conflitos é um dos requisitos importantes para a implementação de memórias transacionais. Essa granularidade pode ser em três níveis:

- **Detecção no nível de objeto**: É quase o mesmo raciocínio de um programador em um ambiente orientado a objetos. A vantagem desse nível é que dependendo do tamanho do objeto, ele pode reduzir o *overhead* em termos de espaço e tempo necessários para detectar um conflito. Como desvantagem, esse nível permite detectar falsos conflitos, por exemplo, quando duas transações operam em diferentes partes de um objeto muito grande como um array multidimensional;

- **Detecção no nível de palavra**: Esse nível elimina a detecção de falsos conflitos, mas requer mais tempo e espaço para monitorar e comparar as escritas e leituras;

- **Detecção no nível de linha de cache**: Esse nível provê um acordo entre a freqüência de detecção de falsos conflitos e o *overhead* em termos de tempo e espaço. Infelizmente linhas de cache e palavras não são entidades no nível de linguagem (diferentemente de objetos), o que dificulta para os programadores otimizar os conflitos nos seus códigos, principalmente em ambientes gerenciados em tempo de execução porque escondem a localização dos dados para os programadores;

2.6.2 Níveis de isolamento

Quando uma transação está rodando, os seus resultados intermediários não são visíveis para as outras transações que estão sendo executadas concorrentemente. Isso significa que uma transação roda em isolamento.

Existem dois tipos de isolamento, chamados de isolamento forte e isolamento fraco. Isolamento fraco somente garante isolamento entre as transações. Isolamento forte garante isolamento entre as transações e o código fora das transações.

Com isolamento fraco, um conflito de memória pode ocorrer fora de uma transação, o que não será visto pelo sistema transacional. Consequentemente, o resultado da execução, tanto do código não transacional como das transações, poderá ter resultados inconsistentes. A idéia do isolamento fraco é que todo o acesso as regiões compartilhadas de memória seja feito dentro das transações.

Já o isolamento forte automaticamente converte cada operação, fora das transações, em uma transação. Então, todos os acessos a regiões compartilhadas de memória serão feitos por transações, garantindo que se existir algum confitlo, ele será detectado.

2.6.3 Versionamento de dados

O versionamento de dados visa garantir a atomicidade das transações. Sempre que uma transação está sendo executada é necessário manter tanto a versão corrente quanto a versão antiga de um dado modificado durante a execução. Caso a transação seja efetivada, o valor corrente do dados se tornará permanente e o valor antigo será descartado. Caso contrário, se a transação falhar, o valor corrente será descartado e o antigo não sofre modificação.

Existem duas maneiras de controlar o versionamento de dados: *versionamento adiantado*, onde os valores alterados são modificados diretamente em memória enquanto o valor antigo é armazenado em um log; e *versionamento tardio*, onde os valores alterados são armazenados em um *buffer* enquanto que a memória contém o valor antigo.

Versionamento adiantado

Com o versionamento de dados adiantado, sempre que um dado sofre alteração o novo valor do dado é atualizado diretamente na memória e o valor antigo do dado é guardado em um log. Se a transação for efetivada, o valor do log (valor antigo) é descartado e o valor na memória fica permanente. Caso contrário, se a transação falhar o valor que estava na memória é descartado e o valor do log é escrito na memória.

A Figura 2.1 mostra um exemplo de uma transação rodando com o versionamento adiantado. A figura está dividida em 4 partes (A, B, C, D). Na parte A, a transação é iniciada, a memória contém um valor 2 e o log está vazio. Quando a transação é executada (parte B), o valor será alterado para 33. Essa alteração é feita diretamente na memória e o

valor antigo será passado para o log. Se a transação for efetivada (parte C), nada precisa ser feito, o valor do log é descartado e o valor da memória ficará permanente. Caso a transação falhe (parte D), o valor do log deve ser escrito na memória e o valor que estava na memória é descartado.

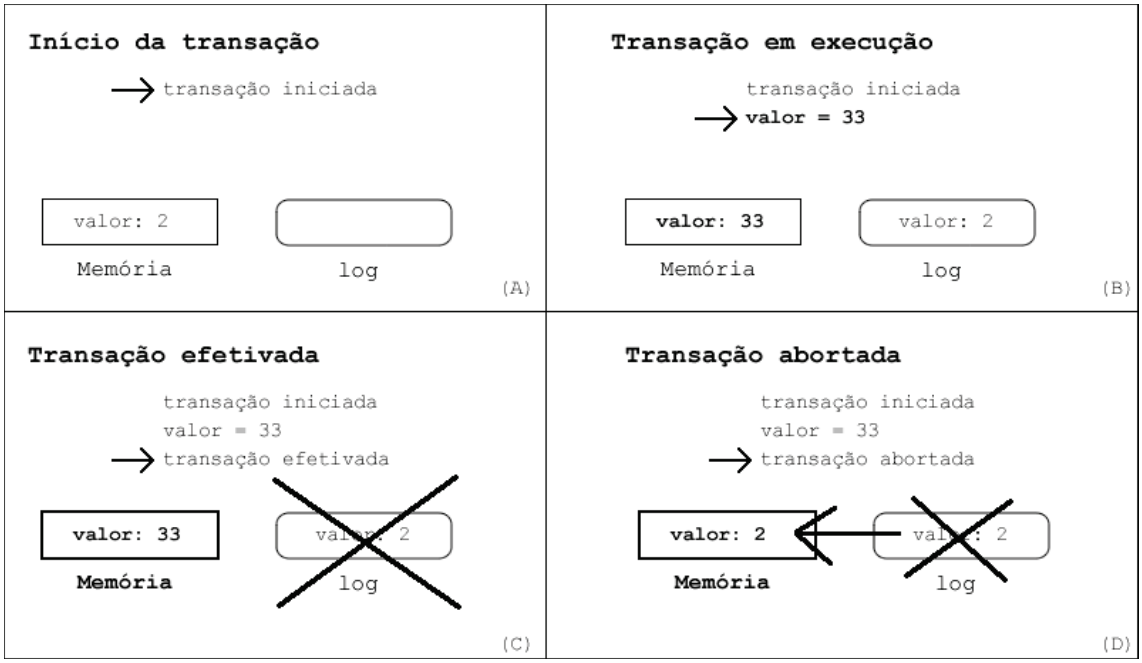

Figura 2.1: Versionamento adiantado

Vale ressaltar que o versionamento adiantado torna mais rápida a efetivação de uma transação porque os valores corretos já estão na memória e nada precisa ser feito. No entanto, com o cancelamento de uma transação, a memória deve ser restaurada com os valores armazenados no log.

Versionamento tardio

Com o versionamento de dados tardio, quando um dado sofre alteração o novo valor será escrito em um *buffer* e o valor que está na memória não sofre alteração. Se a transação for efetivada, o valor do *buffer* (valor novo) é escrito na memória e se torna permanente. Caso contrário, se a transação falhar, nada precisa ser feito e o valor que estava no *buffer* será descartado.

A Figura 2.2 mostra um exemplo de uma transação rodando com o versionamento tardio. A figura está dividida em 4 partes (A, B, C, D). Na parte A, a transação é iniciada, a memória contém um valor 2 e o *buffer* está vazio. Quando a transação é executada (parte B), o valor será alterado para 33. O novo valor será escrito no *buffer* e a memória segue com o valor antigo. Se a transação for efetivada (parte C), o valor do *buffer* será escrito na memória e se torna permanente. Caso a transação falhe (parte D), nada precisa ser feito e o *buffer* será descartado.

Vale ressaltar que o versionamento tardio torna mais lenta a efetivação de uma transação comparado com o versionamento adiantado porque os valores corretos devem ser transferidos do *buffer* para a memória. No entanto, com o cancelamento de uma transação, nada precisa ser feito e o *buffer* será descartado.

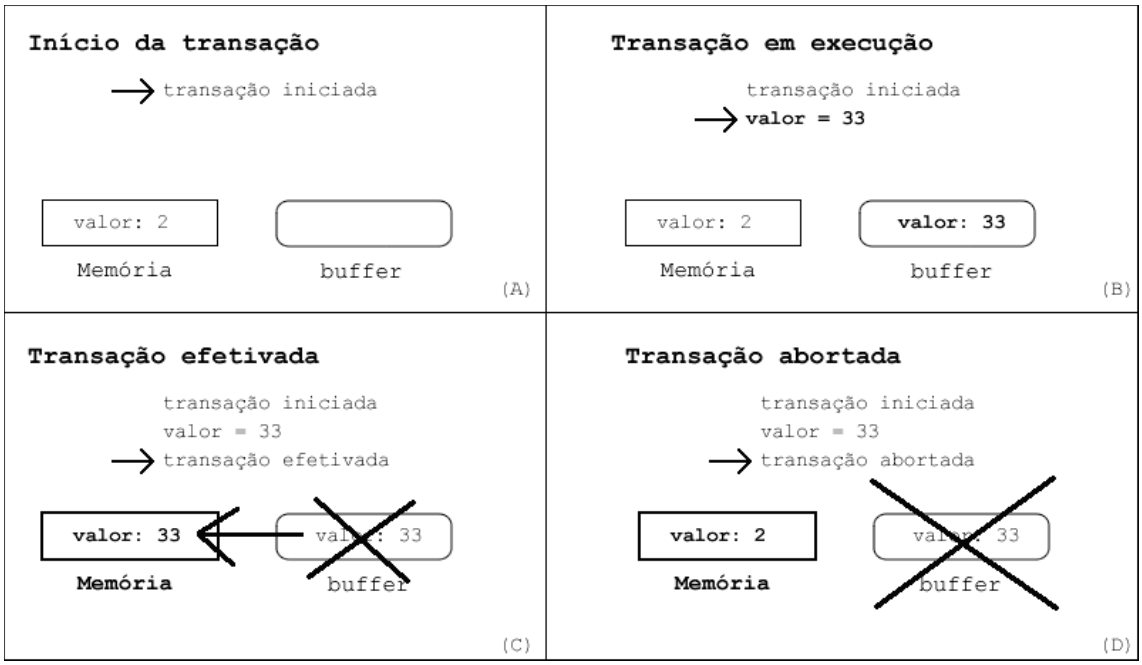

Figura 2.2: Versionamento tardio

2.6.4 Detecção de conflitos

A detecção de conflitos visa garantir o isolamento das transações. Uma solução simples para isso seria executar somente uma transação por vez, mas esse comportamento não explora nenhum paralelismo. Então, o isolamento das transações é feito com base na detecção e resolução de conflitos. Cada transação possui um conjunto de leitura e um conjunto de escrita, onde são armazenados todos os endereços acessados pela transação na leitura e na escrita de dados, respectivamente. Diz-se que uma transação T1 conflita com uma transação T2 quando a intersecção entre os conjuntos de escrita das duas transações e/ou a intersecção entre o conjunto de escrita de T1 e de leitura de T2 (e/ou vice-versa) não é vazia. Sempre que duas transações manipularem conjuntos de dados disjuntos não haverá conflito.

Existem duas maneiras de detectar os conflitos entre as transações: *detecção adiantada*, onde o conflito é detectado no momento em que uma posição de memória é acessada; e *detecção tardia*, onde o conflito só é detectado no momento da efetivação da transação.

Detecção adiantada

Na detecção de conflitos adiantada, se uma transação T1 escrever em uma posição de memória e uma transação T2 acessar essa posição, tanto para leitura como escrita, um conflito será detectado. Por exemplo, a Figura 2.3 mostra 3 exemplos de conflito de dados em uma detecção adiantada. No exemplo A, uma transação T1 escreve na posição X, quando a transação T2 tenta ler essa posição um conflito é detectado e a transação T2 será reiniciada automaticamente. Quando a transação reinicia, ela tenta ler a posição X novamente, não detecta nenhum conflito e é efetivada. No exemplo B, uma transação T1 lê a posição X, quando T2 escreve nessa posição é detectado um conflito e a transação T1 será reiniciada. Quando T1 reinicia e faz a leitura da posição X novamente, não é

detectado nenhum conflito e ela será efetivada. Já no exemplo C, uma transação T1 lê e escreve na posição X, quando uma transação T2 também tentar ler e escrever na posição X será detectado um conflito e T1 será reiniciada. Quando T1 reinicia e tenta ler e escrever na posição X novamente, será detectado um conflito com T2 e T2 será reiniciada. Nesse exemplo, podemos notar que as transações T1 e T2 nunca vão ser efetivadas e o sistema entrará em um estado de *livelock*.

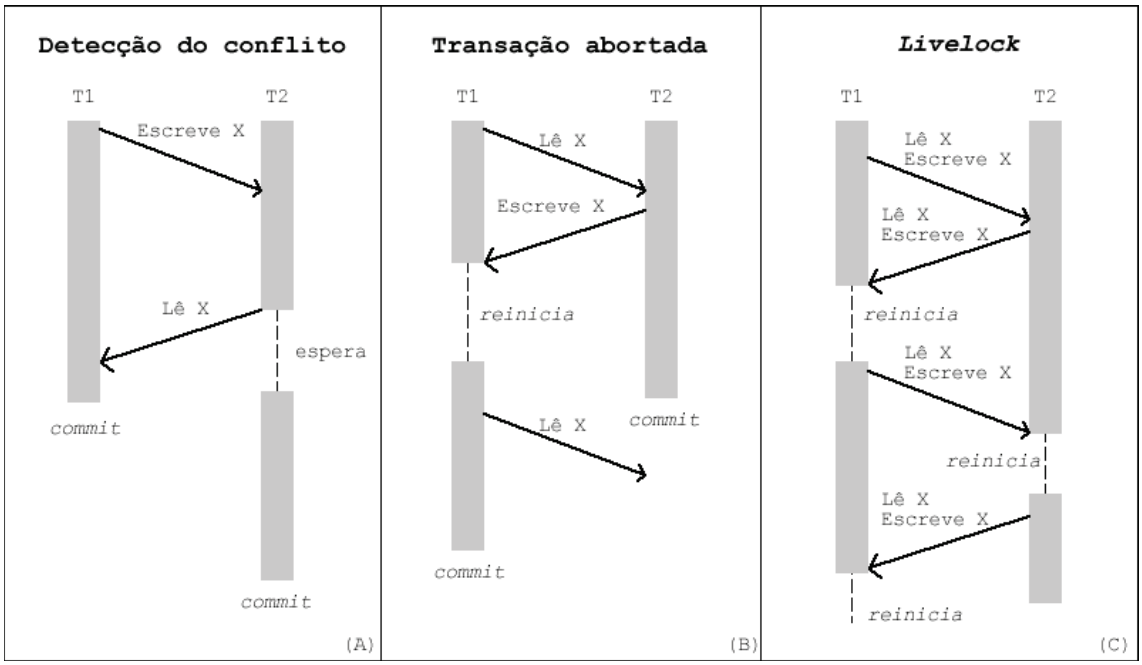

Figura 2.3: Detecção adiantada

Detectar conflitos adiantadamente pode evitar que uma transação seja executada desnecessariamente, mas também pode cancelar transações que, dependendo do progresso de outras, poderiam ser efetivadas normalmente.

Detecção tardia

Na detecção tardia de conflitos, os conflitos somente serão detectados no momento da efetivação de uma transação. Se uma transação T1 escrever em uma posição de memória e uma transação T2 acessar essa posição, tanto para leitura como escrita, um conflito somente será detectado no momento da efetivação de uma delas. Por exemplo, a Figura 2.4 mostra 2 exemplos de conflito de dados em uma detecção tardia. No exemplo A, uma transação T1 escreve na posição X e uma transação T2 lê essa posição X, quando a transação T1 é efetivada, é detectado um conflito com a transação T2, a transação T1 será efetivada e a transação T2 é reiniciada. Depois de re-executar T2 e não detectar nenhum conflito ela é efetivada. No exemplo B, uma transação T2 lê e escreve na posição X, logo em seguida uma transação T1 também lê e escreve na posição X. Quando T1 é efetivada, é detectado um conflito com a transação T2, a transação T1 é efetivada e T2 será re-executada. Depois de T2 ser executada sem conflitos ela será efetivada.

A detecção tardia não permite que ocorra *livelock* como ocorre com a detecção adiantada, mas se no exemplo B a transação T2 fosse muito longa, ela poderia ficar sendo cancelada a todo momento por transações menores e nunca conseguir efetivar suas

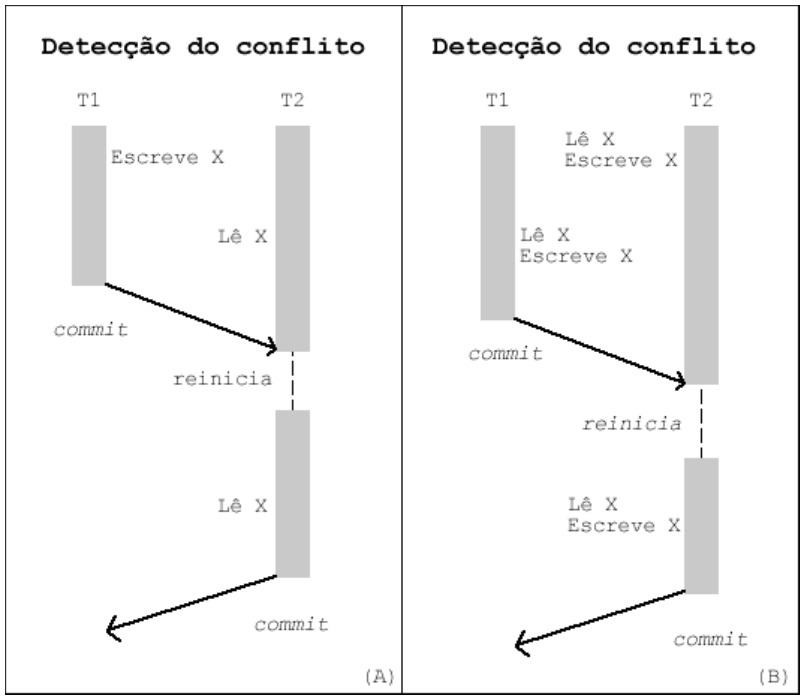

Figura 2.4: Detecção tardia

alterações, entrando no estado de *starvation*.

2.7 Modelos de Memória Transacional

Existem três modelos de memória transacional. O primeiro, conhecido como Memória Transacinal em Hardware (*Hardware Transactional Memory*), trabalha com o desenvolvimento de suporte arquitetural para a execução de transações. O segundo, conhecido como Memória Transacional em Software (*Software Transactional Memory*), implementa um sistema de execução transacional totalmente em software. E o terceiro, conhecido como Memória Transacional Híbrida (*Hybrid Transactional Memory*), visa explorar a união dos modelos de memória transacional em software e em hardware.

Memória Transacional em Hardware

Um dos primeiros modelos de memória transacional em hardware foi descrito por Herlihy e Moss (HERLIHY; MOSS, 1993). Herlihy e Moss modificaram o conceito de transações de bancos de dados para aplicá-lo a operações concorrentes sobre memória compartilhada, permitindo leitura e escrita simultânea em múltiplas posições de memória não-relacionadas, sem o uso de locks. Para dar suporte a transações, eles estenderam a arquitetura do conjunto de instruções de um processador base com 6 novas instruções, adicionaram uma cache especial, chamada de cache transacional, e alteraram o protocolo de coerência. Resultados da simulação para 3 benchmarks sintéticos mostraram um desempenho ligeiramente superior dessa proposta sobre o uso de bloqueios. A implementação de Herlihy e Moss funciona bem sob a suposição de que transações tem vida curta e alteram conjuntos pequenos de dados.

Várias abordagens do modelo de memória transacional em hardware foram de-

senvolvidas após o modelo descrito por Herlihy e Moss, como: Remoção Transacional de bloqueios (TLR - *Transactional Lock Removal*) (RAJWAR; GOODMAN, 2002), Coerência e Consistência em Memória Transacional (TCC - *Transactional Memory Coherence and Consistency*) (HAMMOND et al., 2004), Memória transacional Ilimitada (UTM - *Unbounded Transactional Memory*) (ANANIAN et al., 2005), Memória Transacional Virtualizada (VTM - *Virtualizing Transactional Memory*) (RAJWAR; HERLIHY; LAI, 2005), entre outras.

Memória Transacional em Software

O modelo de Memória Transacional em Software implementa o conceito de transação puramente em software. As duas grandes vantagens desse modelo dizem respeito a sua facilidade de implementação em um grande número de arquiteturas já existentes e sua subseqüente portabilidade.

Shavit e Touitou (SHAVIT; TOUITOU, 1995) cunharam o termo Software Transactional Memory em 1995, propondo uma solução totalmente em software para o mecanismo em hardware de Herlihy e Moss (1993). O sistema mantém, para cada palavra em memória compartilhada, um registro de posse (*ownership record*) que aponta para o registro da transação (*transaction record*) detentora da palavra. O processo de efetivação de uma transação consiste da aquisição de posse de todas as palavras a serem alteradas, efetuação das alterações e liberação da posse. Um conflito acontece caso alguma palavra já tenha sido adquirida por outra transação.

Existem várias abordagens do modelo de memória transacional em software propostos depois de Shavit e Touitou, como: Memória Transacional em Software para Estruturas de Dados Dinâmicas (DSTM - *Dynamic Software Transactional Memory*) (HERLIHY et al., 2003a), Transações em Memória Otimizadas (OMT - *Optimizing Memory Transactions*) (HARRIS et al., 2006), entre outras. Algumas abordagens são baseadas em objetos e disponibilizam uma API transacional diretamente para o programador, como é o caso da abordagem DSTM. Outras são integradas a um sistema de execução e compilador, permitindo a otimização de código e mais controle na execução das transações, como é o caso da abordagem OMT.

Memória Transacional Híbrida

As abordagens híbridas para memória transacional combinam os modelos de software e hardware na esperança de obter o melhor dos dois mundos, ou seja, o desempenho do modelo de hardware e os recursos do modelo de software. Nessas abordagens, uma transação pode ser executada em um de dois modos: hardware, com alto desempenho; ou software, com recursos ilimitados. Uma transação é geralmente iniciada em modo hardware, mas pode ser reiniciada em modo software caso os recursos de hardware sejam exauridos. Um sistema híbrido precisa corretamente detectar e resolver conflitos entre transações em modo hardware e transações em modo software, já que ambos os tipos podem coexistir no sistema.

Existem várias propostas para o modelo de memória transacional híbrida, como: Modelo Transacional Híbrido de Kumar et al. (KUMAR et al., 2006) que visa otimizar o modelo de software DSTM, originalmente proposto por Herlihy et al. (HERLIHY et al., 2003a), Memória transacional Híbrida (HyTM - *Hybrid Transactional Memory*) (DAMRON et al., 2006) que não depende de nenhuma implementação específica de HTM, assu-

mindo apensa que o modelo HTM tenha suporte para instruções primitivas que permitam inicializar, finalizar e cancelar transações, dentre outras propostas.

2.8 Observações finais

Este capítulo apresentou os conceitos relacionados às memórias transacionais. Foram apresentados os problemas encontrados no uso de bloqueios para sincronização de atividades concorrentes, as construções transacionais básicas, as formas de aninhar transações, os requisitos para implementação e os modelos existentes de memórias transacionais.

O capítulo seguinte apresenta alguns trabalhos relacionados com a linguagem CMTJava. O objetivo é mostrar algumas extensões da linguagem Java que foram desenvolvidas utilizando o modelo de memória transacional em software, visto que a implementação da CMTJava foi toda feita na linguagem Java sem o suporte de hardware transacional.

3 TRABALHOS RELACIONADOS

Existem várias extensões da linguagem Java para memórias transacionais que foram desenvolvidas utilizando o modelo de memória transacional em software. Este capítulo tem como objetivo apresentar algumas dessas extensões, focando em como programar e no modo como cada uma é implementada.

Para cada extensão apresentada será usado um exemplo de um programa básico (*buffer* de um elemento) para ilustrar seu funcionamento e sua implementação. O programa buffer de um elemento possui dois métodos: o método de inserção, que só pode colocar um elemento no buffer quando ele estiver vazio; e o método de remoção, que só pode retirar um elemento do buffer se o buffer estiver cheio. O controle do buffer é realizado por uma variável booleana chamada `livre`. Quando a variável `livre` for verdadeira, um elemento pode ser inserido, quando for falsa, um elemento pode ser retirado.

Vale ressaltar que algumas extensões apresentadas neste capítulo utilizam uma operação CAS (*Compare-And-Swap*) em sua implementação. Essa operação é uma operação especial que compara um valor qualquer com um valor localizado na memória e se forem iguais, escreve um novo valor nessa posição de memória, de forma atômica. Tipicamente essa operação CAS é usada pelas linguagens para fazer alterações na memória de forma atômica. Quando uma transação modifica um valor, ela armazena em um buffer o valor antigo da memória e o valor novo gerado pela transação. Ao efetivá-la, a operação CAS irá, atomicamente, verficiar se o valor da memória continua o mesmo (comparando o valor antigo com o valor atual da memória) e alterar essa posição para receber o novo valor gerado pela transação. Caso a comparação seja falsa nada será feito.

3.1 DSTM

A linguagem DSTM (*Dynamic Software Transactional Memory*) é uma interface de programação de aplicativos (API - *Application Programming Interface*) para sincronizar o acesso a dados compartilhados sem o uso de bloqueios (HERLIHY et al., 2003b). Sua implementação visa uma boa adaptação para implementações que possuam estruturas de dados de tamanhos dinâmicos como listas e árvores.

DSTM é uma extensão da linguagem Java e usa um protótipo experimental da biblitoteca *java.util.concurrent* para chamar operações nativas de *Compare-And-Swap* (CAS). Muito da simplicidade dessa implementação é devido às escolhas de condições de progresso não bloqueantes. O mecanismo de sincronização é livre de obstrução (*obstruction-free*), o que garante que threads bloqueadas não impeçam o progresso de outras threads.

Para programar usando a linguagem DSTM se faz necessário entender alguns conceitos básicos sobre a mesma. Nessa linguagem o programador cria as transações de forma explícita e ainda deve indicar quais objetos farão parte de uma transação. DSTM gerencia uma coleção de "objetos transacionais"que são acessados pelas transações. Um objeto transacional nada mais é que um container para um objeto Java e quando uma transação acessa um objeto, ela acessa o container do mesmo. Objetos transacionais são implementados pela classe `TMObject`.

`TMThread` é a unidadade que controla as transações e extende a classe Thread do Java com algumas operações como: `starting`, `commiting` e `aborting`. Assim como a Thread do Java, o método `run()` faz todo o trabalho.

Para ilustrar o uso da linguagem DSTM, será apresentado um programa básico de um buffer de um elemento. Esse programa possui duas variáveis compartilhadas entre as transações e implementa dois métodos: o primeiro para inserir e o segundo para remover um elemento. A Figura 3.1 mostra o código na linguagem DSTM para a implementação dos dois métodos.

Primeiramente vale observar que no momento da criação das variáveis compartilhadas (linhas: 1 a 5) é necessário criar uma instância de `TMObject` para indicar que essas variáveis serão modificadas por uma transação. O buffer de um elemento é representado por uma variável do tipo inteiro chamada `buffer` e é criado um `tmBuffer` para controlá-lo. Do mesmo modo, a variável booleana chamada `livre` é criada para indicar se o buffer está cheio ou não e é criado para ela um `tmLivre` para controlá-la. Vale ressaltar que toda variável compartilhada entre as threads deve possuir um `TMObject`.

Como o programador cria as transações de forma explícita, quando criamos os métodos para inserir e remover um elemento se faz necessário declarar uma thread para controlar o bloco atômico. Essas threads criadas (linhas: 8 e 27) extendem a classe `TMThread` para executar as ações de iniciar, efetivar ou abortar uma transação. Nas linhas 10 e 29, uma transação é iniciada através do método `beginTransaction()`. Uma vez que um transação é iniciada ela fica ativa até que seja efetivada ou abortada.

Na criação do método `inserir()` (linhas: 7 a 24), após a transação ser inicida, deve-se acessar a variável `livre` para verificar se o buffer está vazio e assim inserir um elemento. Uma transação pode acessar essa variável chamando o método `open()` do TMObject criado para ela (linha 12). Sempre que o método `open()` é invocado, será retornada uma cópia do objeto requerido associado a uma versão. Essa versão é válida somente para essa transação.

Para testar se o buffer está vazio, a variável `disponivel` recebe o resultado de uma leitura da variável `livre` (linha 12). Se o resultado for falso é porque o buffer está cheio e deve-se esperar até o buffer esvaziar para inserir um elemento. Como DSTM não possui a instrução `retry` se faz necessário criar um bloco `while` para ficar lendo a variável até que ela se torne verdadeira. O método `release()` é usado para desconsiderar a última leitura executada. Se o resultado for verdadeiro, o buffer está vazio e podemos inserir um elemento. Então, é dado uma permissão de escrita na variável `livre` (linha 17) e seu valor é alterado para falso (linha 18) e é dado uma permissão de escrita no `buffer` (linha 19) e o elemento é inserido (linha 20). Após a execução de todas as ações é invocado o método `CommitTransaction()` para efetivar a transação.

Na criação do método `remover()` (linhas: 26 a 44), após a transação ser inicida, deve-se acessar a variável `livre` para verificar se o buffer está cheio e assim remover um elemento. A leitura da variável `livre` é semelhante ao método `inserir()`, exceto

```
1  int buffer;
2  TMObject tmBuffer = new TMObject(buffer);
3
4  Boolean livre;
5  TMObject tmLivre = new TMObject(livre);
6
7  public int inserir(int valor) {
8     TMThread thread = (TMThread)Thread.currentThread();
9     while(true) {
10       thread.beginTransaction();
11       try {
12          Boolean disponivel = (Boolean)tmLivre.open(READ);
13          while(!disponivel) {
14             tmLivre.release();
15             disponivel = (Boolean)tmLivre.open(READ);
16          }
17          disponivel = (Boolean)tmLivre.open(WRITE);
18          disponivel = false;
19          int elemento = (int)tmBuffer.open(WRITE);
20          elemento = valor;
21       } catch (Denied d) {}
22       thread.CommitTransaction();
23    }
24 }
25
26 public int remover() {
27    TMThread thread = (TMThread)Thread.currentThread();
28    while(true) {
29       thread.beginTransaction();
30       try {
31          Boolean disponivel = (Boolean)tmLivre.open(READ);
32          while(disponivel) {
33             tmLivre.release();
34             disponivel = (Boolean)tmLivre.open(READ);
35          }
36          disponivel = (Boolean)tmLivre.open(WRITE);
37          disponivel = true;
38          int elemento = (int)tmBuffer.open(READ);
39       } catch (Denied d) {}
40       if (thread.CommitTransaction()) {
41          return elemento;
42       }
43    }
44 }
```

Figura 3.1: Implementação dos métodos *get* e *set* de um buffer de um elemento na linguagem DSTM

o fato de testar se a variável disponível é verdadeira ao invés de falsa. Se o buffer estiver cheio é dado uma permissão de escrita na variável `livre` (linha 36) e seu valor é alterado para true (linha 37) e é dado uma permissão de leitura no `buffer` (linha 38). Após a execução de todas as ações é invocado o método `CommitTransaction()` para efetivar a transação e tendo sucesso o elemento do buffer é retornado.

3.1.1 Implementação

Um objeto transacional é um container para um objeto. Esse container aponta para o próprio objeto e para um Locator que é uma instância da classe `TMObject` que contém três atributos: `transaction`, indica o status da transação que chamou esse objeto; `oldObject`, indica a versão antiga do objeto; e `newObject`, indica a nova versão do objeto. A Figura 3.2 apresenta a estrutura de um objeto transacional.

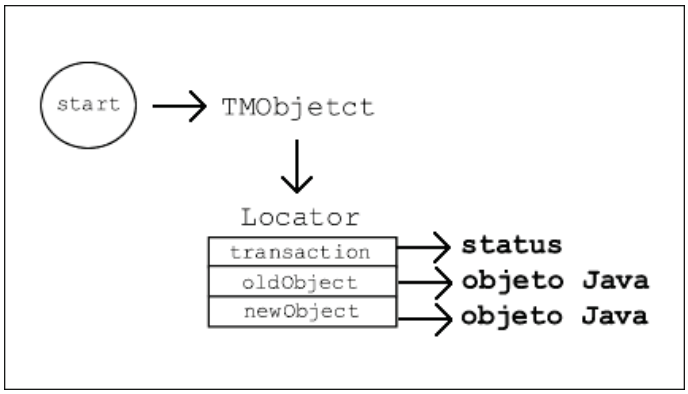

Figura 3.2: Estrutura de um objeto transacional na linguagem DSTM

A parte interessante dessa implementação é como uma transação pode seguramente encontrar uma versão de um objeto transacional sem que exista alguma inconsistência na mesma. Para isso, é preciso que o acesso aos três campos da classe `TMObject` seja atômico e essa atomicidade é feita através de um caminho indireto, no qual cada `TMObject` tem apenas uma única referência para o objeto real (*start*) que aponta para um único `Locator`.

Com o intuito de detalhar mais o funcionamento da linguagem DSTM, suponhemos que uma transação `T1` chame um objeto transacional em modo de escrita e seja `T2` a última transação a abrir esse mesmo objeto em modo escrita. `T1` cria um novo *Locator* para o objeto. Supondo que `T2` esteja efetivada, `T1` aponta o `oldObject` de seu *Locator* para o `newObject` do *Locator* de `T2`, o `newObject` do *Locator* de `T1` aponta para uma cópia do `newObject` do *Locator* de `T2` e `T1` chama uma operação CAS para que a referência do objeto real (*start*) aponte agora para o *Locator* de `T1` e não mais para o *Locator* de `T2`. Caso essa operação falhe, a transação `T1` tenta tudo novamente.

Supondo que `T2` esteja abortada, `T1` aponta tanto o `oldObject` quanto o `newObject` de seu *Locator* para o `oldObject` do `Locator` de `T2` e `T1` chama uma operação CAS para que a referência do objeto real (*start*) aponte agora para o *Locator* de `T1` e não mais para o *Locator* de `T2`.

Por último, supondo que `T2` esteja ativa ainda. `T2` pode tanto ser efetivada quanto abortada a qualquer momento antes de `T1` e `T1` não consegue definir qual a versão corrente do objeto. Nesse caso, entra em cena a sincronização livre de obstrução, onde `T1`

pode tanto optar por abortar `T2` quanto pode optar por esperar que `T2` termine. Para decidir qual a melhor escolha a fazer, existe uma propriedade chamada de gerenciamento de disputa (através de uma interface *contentionManager*) que é executada automaticamente pelo sistema DSTM e através de um algoritmo define qual a melhor escolha.

Acessos somente leitura (*read-only*) são implementados de forma um pouco diferentes. Quando `T1` abre um objeto transacional "o"com permissão de leitura, o método `open()` retorna a versão corrente do objeto ("v") da mesma forma que retornaria para uma permissão de escrita. No entando, ao invés de criar um novo Locator para o objeto, `T1` adiciona o par (o,v) em um tabela local com todas as leituras realizadas. Além do par lido a tabela armazena um contador de leituras para cada par. Quando um par é lido e o mesmo já está na tabela, apenas seu contador será incrementado. Esse contador é decrementado através de um método `release()` e quando chega a zero o par é retirado da tabela.

Validando e efetivando

Depois que um método `open()` determina qual versão de um objeto será retornada, antes de retorná-la o sistema DSTM deve validar a chamada da transação assegurando que um estado inconsistente não será retornado. Existem dois passos para essa validação: primeiro, para cada par (o,v) na tabela de leituras deve ser verificado se `v` continua sendo a versão mais recente para `o`; segundo, checar se o status da transação no *Locator* do objeto continua ativo.

Para efetivar uma transação também são necessários dois passos: primeiro, todas as entradas na tabela de leitura devem ser validadas como descrito no parágrafo anterior; segundo, chamar uma operação CAS para mudar o status do objeto transacional de ativo para efetivado.

3.2 WSTM

A linguagem WSTM (*Word-Granularity STM*) foi criada por Harris e Fraser (HARRIS; FRASER, 2003) e como o próprio nome diz, *Word-Granularity*, possui granularidade no nível de palavra. Ela implementa um mecanismo de concorrência para condições em regiões críticas (*Conditional Critical Regions*, CCR), permitindo que um bloco atômico somente seja executado se uma determinada condição for verdadeira.

WSTM é integrada na linguagem Java e extende a linguagem com novas contruções. Para programar usando essa extensão basta identificar as regiões críticas no código e inserí-las dentro de um bloco atômico. A sintaxe básica para um bloco atômico é:

```
atomic ( condição ) {
   instruções;
}
```

O bloco atômico pode possuir uma condição ou não. Se possuir, ele somente será executado quando a condição for verdadeira, caso contrário ele será executado automaticamente. Dentro do bloco atômico põe-se as intruções que deverão ser executadas dentro de uma transação.

Para exemplificar o uso do bloco atômico, obeserve o exemplo da Figura 3.3 que mostra o programa de um buffer de um elemento. Esse programa possui duas variáveis

```
1   public int remover() {
2      atomic (!livre) {
3         livre = true;
4         return buffer;
5      }
6   }
7
8   public int inserir(int valor) {
9      atomic (livre) {
10        buffer = valor;
11        livre = false;
12     }
13  }
```

Figura 3.3: Implementação dos métodos *get* e *set* de um buffer de um elemento na linguagem WSTM

compartilhadas entre as transações e implementa dois métodos: o primeiro para inserir e o segundo para remover um elemento.

O método *remover()* só será executado quando o buffer estiver cheio, enquanto isso o bloco atômico fica aguardando uma modificação na variável livre para começar a executar. Quando é executado, a variável livre recebe o valor verdadeiro (linha 3) e o elemento do buffer é retornado (linha 4). Já no método *inserir()*, o bloco atômico só será executado quando o buffer estiver livre e então, o buffer recebe o novo elemento (linha 10) e a variável livre recebe falso (linha 11).

3.2.1 Implementação

Transações escrevem suas modificações em *Transaction Entries* e existe uma *transaction entrie* para cada palavra modificada na heap. Uma *transaction entrie* possui cinco atributos: endereço da palavra (*ADDR*), versão antiga (*Old Version*), valor antigo (*Old Value*), versão nova (*New Version*), valor novo (*New Value*).

Cada palavra possui um número de versão associada. Quando uma palavra é lida a versão antiga guarda a versão corrente da mesma. Quando a transação tentar efetivar, se a versão antiga do valor for diferente da versão atual será detectado um conflito, o que caracteriza detecção de conflitos tardia.

Uma transação possui um descritor de transação (*Transaction Descriptor*) para guardar o status atual da transação, o nível de aninhamento e um vetor com todas as *transaction entries* acessadas.

Cada endereço da heap é mapeado (usando uma função HASH) para um OWNERSHIP RECORD (OR). Um OR pode conter um número de versão do valor ou apontar para uma *transaction entrie* no caso de uma transação estar sendo efetivada e usar esse endereço. A Figura 3.4 apresenta a estrutura da linguagem WSTM.

Todo o código escrito em um bloco atômico dentro da linguagem WSTM será traduzido pelo compilador para operações transacionais específicas da linguagem. Existem cinco operações para controlar as transações e duas para controlar a leitura e escrita em palavras compartilhadas. As operações para controle das transações são:

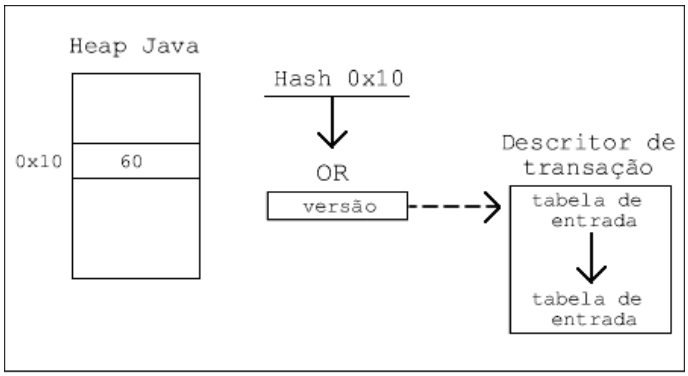

Figura 3.4: Estrutura da linguagem WSTM

- **void STMStart()**: Cria um novo descritor de transação, se não existe ainda, senão incrementa o aninhamento;

- **void STMAbort()**: Muda o estado da transação para abortada e desaloca todas as estruturas;

- **boolean STMCommit()**: Adquire o OR (usando uma operação CAS (*Compare-And-Swap*)) de todas as palavras modificadas pela transação e modifica o estado da transação para efetivada. Após isso, copia os valores para o heap e libera todos os ORs. Toda essa execução parece atômica pois a transição do estado ativo para efetivado é feita de forma atômica. Dessa maneira quando outra transação tenta ler um valor que está sendo efetivado, ela irá ler direto do descritor da transação efetivada;

- **boolean STMValidate()**: Verifica se a versão antiga de uma palavra não foi alterada por outra transação na hora de ser efetivada;

- **void STMWait()**: Implementa a espera condicional, se a condição do bloco atômico for falso, ela muda o estado da transação para *ASLEEP* e adquire o OR das variáveis (como se fosse um STMCommit). Quando outra transação tentar ser efetivada a transação que estava dormindo será acordada;

E as operações para controle de leitura e escrita de palavras compartilhadas são:

- **stm_word STMRead(addr a)**: Retorna o valor corrente de um endereço. Se já existe uma *transaction entry* para ele, o valor lido será retirado da tabela. Caso contrário, será criada uma nova *transaction entry* para a palavra e ela será lida do heap. E na hipótese de uma outra transação estar sendo efetivada e tenha adquirido o endereço, o valor deve ser retirado do descritor de transação da transação que está sendo efetivada;

- **void STMWrite(addr a, stm_word w)**: Escreve o novo valor da palavra dentro da *transaction entry*, no campo novo valor (*New Value*) e incrementa a versão, no campo nova versão (*New Version*);

```
1   public int remover() {
2      atomic {
3         if (livre) {
4            watch livre;
5            retry;
6         }
7         livre = true;
8         return buffer;
9      }
10  }
11
12  public int inserir(int valor) {
13     atomic {
14        if (!livre) {
15           watch livre;
16           retry;
17        }
18        buffer = valor;
19        livre = false;
20     }
21  }
```

Figura 3.5: Implementação dos métodos *get* e *set* de um buffer de um elemento na linguagem ATOMOS

3.3 Atomos

A linguagem Atomos é a primeira linguagem de programação criada com transações implícitas e forte atomicidade (CARLSTROM et al., 2006). Atomos é derivada da linguagem Java, mas substitui suas estruturas de sincronização e espera condicional por alternativas transacionais simples. Vale salientar que para fornecer transações implícitas e forte atomicidade, Atomos faz uso de um modelo de memória transacional em hardware, chamado de consistência e coerência transacional (TCC - *Transactional Coherence and Consistency*) (MCDONALD et al., 2005).

Outra característica interessante é que essa linguagem suporta aninhamento aberto de transações (*open nested*), o que permite que transações aninhadas sejam efetivadas independentes da transação pai. Atomos também disponibiliza duas contruções, *retry* e *watch*, para fazer uma espera condicional em uma determinada palavra compartilhada.

Para mostrar o uso da linguagem Atomos, obeserve o exemplo da Figura 3.5 que mostra o programa de um buffer de um elemento. Esse programa possui duas variáveis compartilhadas entre as transações e implementa dois métodos: o primeiro para inserir e o segundo para remover um elemento.

Atomos sempre executa um bloco atômico automaticamente, diferente da linguagem WSTM que possui uma espera condicional para executar um bloco. Note que no método *remover()* quando a variável livre for verdadeira (linha 3), é executada a instrução *watch* na variável *livre* (linha 4) e um *retry* (linha 5). A instrução *watch* faz com que o sistema adicione o endereço da variável em uma lista local e o *retry* faz com que o sistema aguarde uma modificação nessa variável para ser re-executada. No caso de a variável livre

ser falsa, então ela receberá verdadeiro (linha 6) e o elemento do buffer é retornado (linha 7).

No método *inserir()* o funcionamento é o mesmo. Se a variável livre for falsa (linha 14), será executado um *watch* e depois um *retry*. Caso seja verdadeira, o buffer recebe o novo elemento (linha 18) e a variável livre recebe falso (linha 19).

3.3.1 Implementação

A implementação da linguagem Atomos é feita com base na *Jikes Research Virtual Machine* (JikesRVM), a qual roda sobre um modelo de memória transacional em hardware, chamado de consistência e coerência transacional (TCC - *Transactional Coherence and Consistency*). Todas as operações para se trabalhar com transações são fornecidas por esse modelo de consistência e coerência transacional.

Sempre que uma instrução *watch* é chamada, ela adiciona o endereço da variável em uma lista local. Quando a thread chama *retry*, ela usa uma transação de aninhamento aberto para mandar o endereço da variável para o escalonador. O escalonador confirma o recebimento desse endereço e a thread original reinicia e fica aguardando. Quando é feita uma escrita na variável, o escalonador é violado, fazendo com que a thread original acorde e volte a executar.

Atomos também fornecem duas interfaces, *CommitHandler* e *AbortHandler*, que podem ser associadas a uma thread e podem executar ações em um novo contexto de transações após uma determinada thread ser efetivada ou abortada. Esses *handlres* podem ser usados para tratar exceções ou até mesmo para compensar ações de uma transação que rodou em aninhamento aberto. Por exemplo, se existe um contador compartilhado por várias transações e cada transação possui uma transação filha para incrementá-lo, toda vez que uma transação pai falhar e esse contador já ter sido incrementado, ele deverá ser decrementado para que não fique com um resultado inconsistente.

3.4 Observações finais

Este capítulo apresentou algumas extensões da linguagem Java que estendem a mesma com novas contruções para memórias transacionais. Para cada extensão, foi apresentado seu funcionamento e o modo como cada uma é implementada.

O capítulo seguinte apresenta a linguagem CMTJava. O objetivo é, além de apresentar a linguagem, apresentar os aspectos que a diferencia das demais extensões apresentadas.

4 A LINGUAGEM CMTJAVA

CMTJava (DU BOIS; ECHEVARRIA, 2009) é uma linguagem de domínio específico para programação de memórias transacionais em Java. Ela foi desenvolvida adaptando a linguagem funcional STM Haskell (HARRIS et al., 2008) para um contexto orientado a objetos. Seu desenvolvimento foi todo feito em Java, mas as idéias apresentadas podem ser implementadas em qualquer outra linguagem orientada a objetos que suporte *closures*, como por exemplo C#.

Este capítulo tem como objetivo apresentar a linguagem CMTJava. Serão apresentados alguns simples exemplos com o objetivo de mostrar como se programa utilizando a linguagem. Na Seção 4.1 é apresentada a linguagem STM Haskell, a qual serviu de inspiração para a linguagem CMTJava. Na Seção 4.2, é apresentada a linguagem CMTJava e na Seção 4.3 é apresentada uma comparação entre as linguagens apresentadas no trabalho, com o objetivo de destacar as características da CMTJava.

4.1 STM Haskell

STM Haskell (HARRIS et al., 2008) é uma extensão da linguagem funcional Haskell para programação concorrente usando memórias transacionais.

Uma linguagem de programação funcional, tipo Haskell, é perfeita para se implementar memórias transacionais por dois motivos (HARRIS et al., 2008):

- O sistema de tipos consegue separar as ações que possuem efeitos colaterais das que não possuem. Nem todo o tipo de ação pode ser realizada dentro de uma transação. O sistema de tipos de linguagens como o Haskell consegue garantir que somente as ações corretas serão executadas dentro de uma transação.

- Em uma linguagem funcional, a maior parte das computações são puras no sentido de que não possuem efeitos colaterais. Esse tipo de ação não modifica a memória e não precisa ser logada pelo sistema transacional. As ações puras da linguagem nunca precisam ser desfeitas, elas podem simplesmente ser repetidas no caso de uma transação abortar.

A primitiva `atomically` do STM Haskell serve para marcar um bloco de código que deve ser executado atomicamente com relação a todos os outros blocos atômicos do programa e possui o seguinte tipo:

```
atomically :: STM a -> IO a
```

A primitiva `atomically` recebe como argumento uma *ação transacional* de tipo `STM a` e gera uma ação de entrada e saída (*IO*) que quando executada também executa a transação de forma atômica. Basicamente, dentro de uma ação `STM a` pode-se apenas ler e escrever em váriaveis transacionais (`TVar a`). Programadores definem *variáveis transacionais* (`TVars`) que podem ser lidas e escritas usando duas primitivas:

```
readTVar :: TVar a -> STM a
writeTVar :: Tvar a -> a -> STM a
```

A primitiva `readTVar` recebe uma `TVar` como argumento e retorna uma ação transacional que, quando executada, retorna o valor corrente da `TVar`. De forma similar, a primitiva `writeTVar` é usada para escrever um novo valor em uma `TVar`. STM Haskell também permite que ações transacionais possam ser compostas para gerar novas transações usando a mesma notação usada em ações de *IO* em Haskell (PEYTON JONES, 2003).

4.1.1 Exemplo: O jantar dos filósofos

Esta seção tem como objetivo mostrar como implementar o problema do jantar dos filósofos utilizando a linguagem STM Haskell. Essa implementação foi retirada do artigo (HUCH; KUPKE, 2005).

O jantar dos filósofos é um problema clássico de programação concorrente que foi proposto por Dijkstra em 1965. Cinco filósofos estão sentados em uma mesa redonda e cada um possui um prato de comida em sua frente. Para poder comer, um filósofo necessita de dois garfos, só que existe somente um garfo entre cada prato, assim como na Figura 4.1. No contexto do problema, um filósofo faz apenas duas ações: comer ou pensar. Toda a vez que sente fome, o filósofo tenta pegar dois garfos, um à direita e outro à esquerda para comer. Se conseguir pegar os dois garfos então o filósofo come um pouco, coloca os garfos novamente na mesa e volta a pensar.

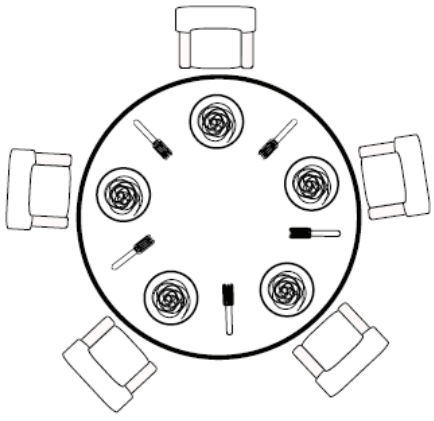

Figura 4.1: O Jantar dos Filósofos (DU BOIS, 2008)

Para implementar esse problema usando a linguagem STM Haskell, primeiramente os garfos são representados por uma TVar que guarda um valor booleano indicando se o garfo está disponível (`True`) ou não (`False`).

```
type Garfo = TVar Bool
```

A palavra reservada `type` em Haskell serve para se criar *sinônimos de tipos* no sentido de que o tipo `Garfo` é equivalente ao tipo `TVar Bool`. Sinônimos de tipo servem para facilitar a leitura do código.

Um filósofo pode ser representado pela seguinte função:

```
filosofo ::    Int          -- Nome do Filosofo
            -> Garfo        -- Garfo da Esquerda
            -> Garfo        -- Garfo da Direita
            -> IO ()
filosofo n e d = do
                 atomically ( do
                              pegaGarfo e
                              pegaGarfo d)
                 print ("O filósofo " ++ (show n)
                         ++ " esta comendo!")
                 atomically ( do
                              devolveGarfo e
                              devolveGarfo d)
                 filosofo n e d
```

O `filosofo` recebe como argumentos um inteiro que o identifica e duas referências a garfos, um que está à sua esquerda e outro à direita. Primeiramente o filósofo realiza uma ação atômica que serve para adquirir os dois garfos. Uma vez adquiridos, o filósofo pode comer. Em seguida, este realiza outra ação atômica que serve para devolver os garfos para a mesa e então tenta novamente comer fazendo uma chamada recursiva à função `filosofo`.

A função `pegaGarfo` serve para ler a `TVar` que contém o garfo:

```
pegaGarfo :: Garfo -> STM ()
pegaGarfo g = do
              b <- readTVar g
              if(!b)
                then retry
                else writeTVar g False
```

Se o garfo não se encontra disponível na `TVar` então é chamado `retry`. A implementação da função `retry` faz com que esta ação seja executada novamente no momento em que a `TVar` contendo o garfo seja modificada.

A função `colocaGarfo` possui uma implementação mais simples:

```
colocaGarfo :: Garfo -> STM ()
colocaGarfo g = writeTVar g True
```

Como no momento em que a função `colocaGarfo` é chamada os garfos estão com o filósofo, então sabe-se que o valor guardado na `TVar` é `False`, ou seja, o garfo não está disponível. Como o objetivo de `colocaGarfo` é devolver o garfo para a `TVar`, a função simplesmente escreve `True` na variável transacional.

Por último, usa-se um programa um pouco mais complicado, que serve para inicializar as threads que contêm os filósofos passando para elas as referências às variáveis transacionais que contêm o estado dos garfos:

```
iniciaFilosofos :: Int -> IO ()
iniciaFilosofos n =
          do
          garfosIO <- atomically (criaGarfos n)
          mapM_ (\(l,r,i)->forkIO (filosofo i l r))
                (zip3 garfosIO (tail garfosIO) [1..n-1])
          filosofo n (last garfosIO) (head garfosIO)
```

Além disso, uma função auxiliar `criaGarfos` é usada para criar a lista contendo todos os `Garfos`. Inicialmente todos os garfos estão disponíveis, ou seja, o valor guardado nas `TVars` é `True`:

```
criaGarfos :: Int -> STM [Garfo]
criaGarfos n =
          do
          garfos <- mapM (const (newTVar True)) [1..n]
          return garfos
```

4.2 CMTJava

Nesta seção são apresentados alguns exemplos simples com o objetivo de descrever como se programa na linguagem CMTJava.

4.2.1 Exemplo: O jantar dos filósofos

O primeiro exemplo a ser mostrado é o jantar dos filósofos, descrito na Seção 4.1.1. Para implementar esse problema usando a linguagem CMTJava, primeiramente definimos a classe `Garfo`:

```
class Garfo implements TObject {
  private volatile Boolean garfo = true;
}
```

A interface `TObject` funciona como um alerta para o compilador, que irá gerar automaticamente os códigos necessários para que essa classe seja acessível dentro das transações. Para cada atributo da classe são gerados dois métodos especiais de `get` e `set`. Esses métodos são a única forma de acessar os atributos da classe. Para a classe `Garfo`, são gerados os métodos:

```
STM<Void> setGarfo (Boolean garfo);
STM<Boolean> getGarfo();
```

Esses métodos retornam *ações transacionais* como resultado. O tipo `STM<A>` representa uma *ação transacional* que quando for executada gerará um valor do tipo `A`. Nota-se que apenas implementando a interface `TObject` o programador já está tornando essa classe acessível dentro de uma transação, o que diferencia a CMTJava da linguagem DSTM (Seção 3.1), onde o programador, além de criar as transações de forma explícita, deve criar uma instância da classe `TMObject` para cada objeto que será acessado por uma transação.

```
class Filosofo implements Runnable {

  int id;
  Garfo direita;
  Garfo esquerda;

  Filosofo (int i, Garfo e, Garfo d) {
    id = i;
    esquerda = e;
    direita = d;
  }

  public void run() {
    STM<Void> t1 = STM {
      adquireGarfo(d);
      adquireGarfo(e);
    }
    atomic(t1);

    System.out.println("Filósofo"+id+"comendo!");

    STM<Void> t2 = STM {
      devolveGarfo(r);
      devolveGarfo(l);
    }
    atomic(t2);
  }
  (...)
}
```

Figura 4.2: A classe `Filosofo`

Já a classe `Filósofo` pode ser descrita conforme a Figura 4.2:

O bloco `STM{...}` é usado para compor transações, assim como a notação `do` na linguagem STM Haskell (HARRIS et al., 2008). A notação `STM{a1;...;an}` contrói uma ação `STM` que une pequenas operações `a1;...;an` em seqüência. Usando blocos `STM{...}` é possível implementar os métodos `STM<Void> adquireGarfo(Garfo g)` e `STM<Void> devolveGarfo(Garfo g)` que estão faltando na classe `Filosofo`.

O método `adquireGarfo` primeiro verifica o estado atual do `Garfo` usando o método `getGarfo`. Se o garfo não está disponível, a transação será pausada chamando `retry`. Se o garfo estiver disponível, ele será setado para falso, tornando-o indisponível. A implementação do método `adquireGarfo` é apresentada a seguir:

```
public STM<Void> adquireGarfo(Garfo g) {
  STM<Void> r = STM {
    Boolean disponivel <- f.getGarfo();
    if (!disponivel) {
      retry();
    } else {
      f.setGarfo(false);
    }
  }
  return r;
}
```

Uma chamada a `retry` irá pausar a transação que será abortada e re-executada desde o início. A transação não deve ser re-executada enquanto não houver alguma

alteração em seus TObjects. Por exemplo, no método `adquireGarfo`, quando ele chama `retry` o filósofo ficará esperando até que o seu vizinho coloque o garfo de volta na mesa para ele poder comer.

Na linguagem Atomos (Seção 3.3) essa espera por uma alteração na variável compartilhada deveria ser feita usando uma instrução auxiliar `watch`. Essa instrução é usada para adicionar a variável compartilhada em uma lista local, de modo que o método `retry` fica aguardando uma modificação nessa variável presente na lista para poder re-executar a transação. Nota-se que na CMTJava não é necessário usar uma instrução auxiliar, tudo é feito de forma implícita. Quando uma transação chama `retry` o sistema automaticamente fica aguardando uma modificação em todas as variáveis acessadas pela transação durante sua execução. Assim que uma dessas variáveis sofre alteração, a transação é executada novamente.

As variáveis criadas dentro de uma ação transacional são variáveis de *atribuição única*. Essas variáveis são usadas somente para guardar um estado intermediário de uma transação, não tendo necessidade de manter essa variável no log da transação. Para enfatizar que essas variáveis são diferentes das variáveis padrões do Java é usado um símbolo diferente para atribuição (<-).

O método `devolveGarfo` possui uma implementação simples e é apresentado a seguir:

```
public STM<Void> devolveGarfo(Garfo g) {
  return f.setGarfo(true);
}
```

Esse método retorna uma ação transacional que quando executada irá setar o valor do garfo para verdadeiro incondicionalmente. Quando um filósofo chama o método `devolveGarfo`, significa que o mesmo já possui o garfo. Então, esse método altera o valor do garfo para verdadeiro incondicionalmente, sem a necessidade de verificar o valor corrente da variável.

4.2.2 Exemplo: Conta Bancária

CMTJava também provê o método `OrElse` que é usado para compor transações como alternativa. O método `OrElse` recebe duas transações como argumento e retorna uma nova transação:

```
public static <A> STM<A> orElse (STM<A> t1, STM<A> t2)
```

Sendo `OrElse(t1,t2)`, a primeira transação executada será `t1`. Se `t1` chamar `retry`, `t1` é descartada e `t2` é executada. Se `t2` também chamar `retry`, `t2` é descartada e todo o método será executado novamente.

Então, o segundo exemplo a ser mostrado utilizando a linguagem CMTJava é o exemplo de conta bancária. A classe `ContaBancaria` descrita conforme a Figura 4.3, representa uma simples implementação de uma conta bancária.

Usando o `orElse` pode-se estender a classe `Conta` com o método:

```
public STM<Void> saque2contas (Conta c1, Conta c2) {
  return orElse ( c1.saque(1000), c2.saque(1000) );
}
```

```
public class ContaBancaria implements TObject {

  private volatile Double saldo;

  public STM<Void> saque (Double n) {
    STM<Void> t = STM { Double b <- getSaldo();
      if (b < n) {
        retry()
      else
        setSaldo(b-n)
      };
      return t;
    }

  public STM<Void> deposito (Double n) {
    return STM { Double b <- getSaldo();
      setSaldo(b + n); };
    }

  public STM<Void> transferencia (Conta c1, Conta c2, Double n) {
    return STM { c1.saque(n);
      c2.deposito(n)
    };
  }
}
```

Figura 4.3: Classe `Conta`

O método `saque2contas` primeiro tenta sacar R$ 1.000,00 da conta `c1`. Caso não tenha dinheiro suficiente nessa conta, ele tenta sacar a mesma quantidade da conta `c2`.

4.3 Comparação

Esta seção tem como objetivo apresentar uma comparação entre as linguagem de programação para memórias transacionais apresentadas neste trabalho, com o objetivo de destacar as características da linguagem CMTJava.

Foram escolhidas cinco características das linguagens de programação para memórias transacionais apresentadas, levando em consideração se a linguagem possui transações implícitas, se ela é orientada a objetos, se ela possui a construção `OrElse`, se implementa o modelo de memória transacional em software (STM) e, por fim, se ela roda em processadores multi-core. A Tabela 4.1 apresenta a comparação das linguagens.

Tabela 4.1: Comparação entre as linguagens para memórias transacionais

Linguagem	**DSTM**	**WSTM**	**Atomos**	**STM Haskell**	**CMTJava**
Transações Implícitas	Não	Sim	Sim	Sim	Sim
Orientada a Objetos	Sim	Sim	Sim	Não	Sim
Construção OrElse	Não	Não	Não	Sim	Sim
STM	Sim	Sim	Não	Sim	Sim
Multi-core	Sim	Sim	Sim	Não	Sim

A primeira característica compara as linguagens pelo modo como são criadas as transações, se as transações são criadas implicitamente ou não. Dizer que uma transação

é criada de modo implícito significa que ao criar um bloco atômico, uma transação será criada automaticamente. Nota-se que a linguagem DSTM não cria as transações implicitamente, é necessário utilizar alguns comandos da classe `TMThread` para criar as transações.

A segunda característica compara as linguagens pelo paradigma de orientação a objetos. Nota-se que somente a STM Haskell não é orientada a objetos porque é uma linguagem funcional, já as outras são extensões da linguagem Java que estendem a linguagem com novas construções para utilizar memórias transacionais.

A teceira caracterísa verifica quais linguagens possuem a construção `OrElse`. Somente a linguagem STM Haskell e a linguagem CMTJava possuem.

A quarta característica verfica as linguagens que são implementadas utilizando o modelo de memórias transacionais em software. Nesse item, pode-se notar que a linguagem Atomos não é toda implementada em software, ela roda sobre um modelo de memória transacional em hardware, chamado de consistência e coerência transacional (MCDONALD et al., 2005), onde todas as operações para se trabalhar com transações são fornecidas por esse modelo.

A quinta característica verifica quais linguagens são implementadas para rodarem em processadores multi-core. Vale ressaltar que a linguagem STM Haskell já possui uma versão implementada para rodar em máquinas multi-core (HARRIS; MARLOW; Peyton Jones, 2005), mas a versão original da linguagem, a qual serviu de inspiração para a CMTJava, é implementada para rodar em máquinas com um único processador. Já as outras linguagens apresentadas são todas implementadas para rodarem em processadores multi-core.

Analisando a Tabela 4.1, pode-se notar que a CMTJava possui todas as características apresentadas. Ela é uma linguagem implementada no paradigma de orientação a objetos, utilizando o modelo de memória transacional em software e roda em processadores multi-core. As transações são criadas de modo implícito e ela possui a construção `OrElse`.

4.4 Observações finais

Este capítulo apresentou a linguagem CMTJava através de alguns simples exemplos para mostrar como se programar usando a mesma.

O capítulo seguinte apresenta os dois protótipos desenvolvidos para implementar a linguagem CMTJava, bem como alguns testes realizados com os mesmos.

5 IMPLEMENTAÇÕES DA CMTJAVA

Hoje a CMTJava conta com dois sistemas transacionais para rodar a linguagem. O primeiro sistema transacional desenvolvido foi baseado no sistema transacional da linguagem STM Haskell (HARRIS et al., 2008). Já o segundo sistema transacional foi desenvolvido baseado em um algoritmo moderno para transações, denominado TL2 (*Transactional Locking II*) (DICE; SHALEV; SHAVIT, 2006).

O primeiro sistema transacional da linguagem CMTJava é uma implementação simples, desenvolvida com base no sistema transacional da linguagem STM Haskell. O fato dessa implementação ser simples é que as transações precisam adquirir um bloqueio global sempre que vão validar o log da transação ou efetivar a mesma, o que gera um gargalo serial, limitando o paralelismo.

Já o segundo sistema transacional foi desenvolvido com a intenção de não usar um bloqueio global, afim de aumentar o paralelismo da linguagem e consequentemente aumentar seu desempenho. Para isso foi usado o algoritmo TL2, um algoritmo que combina o uso de um relógio global para controle de versão dos dados, com uma técnica de aquisição do bloqueio dos objetos em tempo de efetivação.

Levando em consideração os requisitos de implementação, descritos na Seção 2.6, que devem ser levados em consideração quando se implementa um sistema transacional para uma linguagem, os dois sistemas transacionais desenvolvidos possuem as mesmas características da implementação original da STM Haskell, como podemos observar na Tabela 5.1.

Tabela 5.1: Requisitos de implementação dos sistemas transacionais

Req. de implementação	**Sist. Trans. 1**	**Sist. Trans. 2**	**STM Haskell**
Granularidade do conflito	Palavra	Palavra	Palavra
Nível de isolamento	Forte	Forte	Forte
Versionamento de dados	Tardio	Tardio	Tardio
Detecção de conflitos	Tardia	Tardia	Tardia

Este capítulo tem como objetivo apresentar os detalhes da implementação dos dois sistemas transacionais da linguagem CMTJava, bem como uma comparação entre os dois sistemas. Na Seção 5.1 é apresentado os conceitos gerais das implementações da CMTJava. Na Seção 5.2 são apresentados os detalhes da implementação baseada na STM Haskell. Na Seção 5.3 são apresentados os detalhes da implementação baseado

no algoritmo TL2. Por fim, na Seção 5.4 é apresentada uma comparação entre as duas implementações de CMTJava.

5.1 Conceitos gerais das implementações

Esta seção apresenta os conceitos gerais das implementações da CMTJava. É importante ressaltar que a linguagem CMTJava foi toda desenvolvida em software, utilizando a linguagem de programação Java. Algumas linguagens desenvolvidas para fazer uso das memórias transacionais rodam sobre um modelo de memória transacional em hardware, como por exemplo a linguagem Atomos (Seção 3.3).

5.1.1 Java Closures

Para implementar a CMTJava foi usada *BGGA Closures*, uma extensão Java que suporta *funções anônimas* e *closures* (JAVA CLOSURES, 2009).

Usando BGGA, uma função anônima pode ser definida usando a sintaxe: { `parâmetros formais => expressão` } , onde tanto os *parâmetros formais* quanto as *expressão* são opcionais. Por exemplo:

```
{ int x => x + 1 }
```

A função definida acima, é uma função que recebe um inteiro e retorna seu valor incrementado de um. Para chamar uma função anônima usamos o método `invoke`, por exemplo:

```
String s = { => "Hello!"}.invoke();
```

Uma função anônima também pode ser referenciada por variáveis. Uma variável pode possuir um tipo que é uma função e quando chamada executará algo:

```
{int => void} func = {int x => Sytem.out.println(x)};
```

A variável `func` possui o tipo {`int => void`} o que significa que é uma função de `int` para `void`. Com a extensão BGGA, podemos também passar funções como argumentos para métodos Java.

Um *closure* é uma função anônima, ou seja, uma função criada dinamicamente em tempo de execução do programa, que captura as variáveis livres automaticamente:

```
public static void main(String args[]) {
  int x = 1;
  {int=>int} func = {int y => x+y};
  x++;
  System.out.println(func.invoke(1)); //vai imprimir 3
}
```

Um *closure* pode usar variáveis que estão dentro de um escopo, mesmo que esse escopo não esteja ativo no momento de sua chamada. Se um closure é passado como um argumento para um método, ele vai continuar usando as variáveis do escopo onde ele foi criado.

5.1.2 Mônadas

Uma mônada (ALL ABOUT MONADS, 2009) é usada para descrever computações que podem ser combinadas gerando novas computações. Por esta razão, mônadas são freqüentemente usadas na implementação de linguagens de domínio específico, principalemente em linguagens funcionais, como por exemplo: parsers (HUTTON; MEIJER, 1998), controle de robôs (PETERSON; HUDAK; ELLIOTT, 1999) e transações de memória (HARRIS et al., 2008).

Uma mônada pode ser implementada como um tipo de dado abstrato que representa um container para uma computação. Essas computações podem ser criadas e combinadas usando três operações básicas: *bind*, *then* e *return*. Para qualquer mônada `m`, essas funções têm os seguintes tipos em Haskell:

```
bind :: m a -> (a -> m b )-> m b
then :: m a -> m b -> m b
return :: a -> m a
```

O tipo `m a` representa uma computação dentro da mônada `m` que quando executada produz o valor `a`. As funções `bind` e `then` são usadas para combinar computações em uma mônada. A função `bind` executa seu primeiro argumento e passa o resultado para o segundo argumento (uma função), produzindo uma nova computação. A função `then` recebe duas computações como argumento e produz uma computação que irá executar uma depois da outra. A função `return` cria uma nova computação para um valor.

5.1.3 Blocos STM

Blocos STM são traduzidos para chamadas para os métodos `bind` e `then`, usando *regras de tradução* mostradas na Figura 5.1.

```
STM{ type var <- e; s} = STMRTS.bind( e, { type var => STM { s }})

STM{ e ; s }           = STMRTS.then( e,  STM{ S })

STM{ e }               = e
```

Figura 5.1: Esquema básico de tradução para blocos STM

Regras de tradução de blocos STM são similares às regras de tradução descritas na notação `do` da linguagem Haskell (PEYTON JONES, 2003).

Por exemplo, considere a seguinte implementação de um método depósito:

```
public STM<Void> deposito (Conta c, Double n) {
  return STM { Double saldo <- c.getSaldo();
               c.setSaldo(saldo + n)};
}
```

O método acima é traduzido para:

```
public STM<Void> deposito (Conta c, Double n) {
  return STMRTS.bind( c.getSaldo(),
                      { Double saldo => c.setSaldo(saldo + n)});
}
```

5.2 Implementação baseada na linguagem STM Haskel

O primeiro sistema transacional a ser apresentado é muito similar ao sistema transacional original da linguagem STM Haskell. Como mostrado na Seção 4.3, a linguagem STM Haskell já possui uma versão implementada para rodar em máquinas multi-core (HARRIS; MARLOW; Peyton Jones, 2005), mas a versão original da linguagem, a qual serviu de inspiração para a CMTJava, é implementada para rodar em máquinas com um único processador. O fato de ela ter sido desenvolvida para rodar em um único processador permite que sejam criadas várias *threads* que rodam concorrentemente, mas não em paralelo, o que torna mais simples a sua implementação. Já o sistema transacional para a CMTJava foi desenvolvido para rodar em máquinas multi-core.

O sistema transacional da linguagem CMTJava possui uma implementação simples, onde as transações precisam adquirir um bloqueio global sempre que vão validar seu log ou serem efetivadas, o que limita o paralelismo, gerando um gargalo serial.

5.2.1 O sistema transacional da STM Haskell

O sistema transacional da linguagem STM Haskell utiliza uma detecção tardia de conflitos e um versionamento de dados tardio.

Todas as variáveis acessadas por uma transação devem ser declaradas como variáveis transacionais. Sempre que uma transação é iniciada, um log para ela é criado. Para cada variável transacional lida/escrita pela transação é inserido no log os seguintes campos:

- `valor antigo`: contém o valor atual da variável transacional, presente na memória;
- `valor novo`: contém o novo valor alterado pela transação;

Quando uma transação chama o método `get()` de uma variável transacional, o método verifica se essa variável está no log da transação. Se sim, será retornado o valor que está no campo `valor novo`. Se não, o método adiciona essa variável no log e retorna o valor da variável. Vale salientar que nesse caso, tanto o campo `valor novo` como o campo `valor antigo` receberão o valor corrrente da variável presente na memória.

O método `set()` funciona de forma similar. Se a variável transacional não está no log da transação, ele será inserida no log. O campo `valor antigo` receberá o valor corrrente da variável presente na memória e o campo `valor novo` receberá o novo valor alterado pela transação. Se a variável transacional já estiver no log da transação, apenas o campo `valor novo` será atualizado com o novo valor da variável.

Como esse sistema transacional utiliza uma detecção tardia de conflitos, seu log somente será validado ao fim da execução da transação. Quando uma transação vai ser efetivada, para cada variável transacional presente no log, é verificado se o campo `valor antigo` é igual ao valor presente na memória. Se sim, é porque nenhuma transação alterou esse valor durante essa execução e ela pode ser efetivada. Se não, é detectado um conflito e a transação deve ser re-executada porque estava sendo executada com valores inconsistentes.

Após o log da transação ser efetivado com sucesso, todas as variáveis transacionais presentes no log são atualizadas na memória. O que caracteriza o versionamento de dados tardio. O log é percorrido e o valor do campo `valor novo` é copiado para memória.

5.2.2 A mônada STM

A mônada para ações STM é implementada através de uma mônada de passagem de estados. A mônada de passagem de estados é usada para passar um estado pelas computações, onde cada computação retorna uma cópia alterada desse estado. No caso das transações, esse estado é um log.

A classe STM é implementada como segue:

```
class STM<A> {
  { Log => STMResult<A> } stm;

  STM ({ Log => STMResult<A> } stm) {
    this.stm = stm;
  }
}
```

A classe STM é usada para descrever transações e possui apenas um atributo: uma função que representa a transação. Uma transação `STM<A>` é uma função que recebe um `Log` como argumento e retorna um `STMResult<A>`. O `Log` representa o estado corrente da transação durante sua execução e `STMResult` descreve o novo estado da transação depois de sua execução.

A classe STMResult possui três atributos:

```
class STMResult<A> {

  A result;
  Log newLog;
  int state;

  (...)
}
```

O primeiro atributo (`result`) é o resultado da execução de uma ação STM, o segundo (`newLog`) é o log resultante e o terceiro (`state`) é o estado corrente da transação. A execução de uma ação STM pode colocar a ação em dois estados: `ACTIVE`, que significa que a transação pode continuar; ou `RETRY`, que significa que a construção `retry` foi chamada e a transação deve ser abortada.

O método `bind` é usado para compor ações transacionais:

```
public static <A,B> STM<B> bind (STM<A> t, {A=>STM<B>} f){
  return new STM<B> ( {Log l1 =>
    STMResult<A> r = t.stm.invoke(l1);
    STMResult<B> re;
    if (r.state == STMRTS.ACTIVE) {
      STM<B> nst = f.invoke(r.result);
      re = nst.stm.invoke(r.newLog);
    } else {
      re = new STMResult(null, r.newLog, STMRTS.RETRY);
    }
    re
  } );
}
```

O método `bind` recebe como argumento uma ação `STM<A> t` e uma função `f` do tipo {`A => STM<B>`} e retorna como resultado uma nova ação `STM<B>`. O objetivo do método `bind` é combinar ações STM gerando novas ações. Uma ação `STM<A> t` é executada recebendo um log (`l1`) como um argumento e sendo invocada através da chamada (`t.stm.invoke(l1)`). Se a chamada de `t` não der `retry` (seu estado é válido), então `f` é chamada gerando o resultado `STM<B>`. Caso contrário, a execução é abandonada e `bind` retorna um `STMResult` com o estado `RETRY`.

O método `then` é implementado de uma forma similar:

```
public static <A,B> STM<B> then (STM<A> a, STM<B> b) {
  return new STM<B> ( {Log l1 =>
    STMResult<A> r = a.stm.invoke(l1);
    STMResult<B> re;
    if (r.state == STMRTS.ACTIVE) {
      re = b.stm.invoke(r.newLog);
    } else {
      re = new STMResult(null, r.newLog, STMRTS.RETRY);
    }
    re
  } );
}
```

O método `then` é uma combinação seqüencial: ele recebe como argumento duas ações STM e retorna uma ação que vai executar uma depois a outra.

Finalmente, o método `stmReturn` é usado para *inserir* um objeto qualquer `A` dentro de uma mônada STM:

```
public static <A> STM<A> stmReturn (A a) {
  return new STM<A>({Log l => new STMResult(a,l,STMRTS.ACTIVE)});
}
```

O método `stmReturn` funciona como o método `return` do Java só que para blocos STM. Ele recebe um objeto como argumento e cria uma simples transação que retorna esse objeto como resultado. Ele também pode ser usado para criar novos objetos dentro de uma transação. Por exemplo, o método `addToTail` da classe `List`, conforme Anexo B, retorna uma transação que insere um novo elemento no fim de uma lista encadeada.

5.2.3 Compilando TObjects

Uma classe como a classe `Garfo` descrita na seção 4.1.1 é compilada para a classe descrita na na Figura 5.2.

Todo `TObject` possui um `PBox` para cada um de seus atributos. Um `PBox` funciona como um ponteiro para o atributo. O construtor da classe `PBox` recebe dois argumentos: uma função que altera o valor corrente do atributo e outra que retorna o valor corrente do atributo. Essas funções são usadas pelo método `atomic` para efetivar uma transação (Seção 5.2.6).

Um log é uma coleção de entradas, definidas por uma classe `LogEntry`:

```
class Garfo implements TObject {

   private volatile Boolean garfo = true;
   private volatile PBox<Boolean> garfoBox =
       new PBox<Boolean> ( {Boolean b => garfo = b; } , { => garfo} );

       public STM<Void> setGarfo (Boolean b) {
            return new STM<Void>({Log l =>
            LogEntry<Boolean> le = l.contains(garfoBox);
            if(le!=null) {
              le.setNewValue(b);
            } else {
              l.addEntry(new LogEntry<Boolean>(garfoBox,garfo,b));
            }
            new STMResult(new Void(), l,STMRTS.ACTIVE)} );
       }

       public STM<Boolean> getGarfo() {
            return new STM<Boolean> ({Log l =>
            Boolean result;
            LogEntry<Boolean> le = l.contains(garfoBox);
            if(le!=null) {
              result = le.getNewValue();
            } else {
              result =garfo;
              l.addEntry(new LogEntry<Boolean>(garfoBox,garfo,garfo));
            }
            new STMResult(result, l,STMRTS.ACTIVE)} );
       }
}
```

Figura 5.2: Classe Garfo depois de ser compilada

```
class LogEntry<A> {
  PBox<A> box;
  A oldValue;
  A newValue;
  (...)
}
```

Cada `LogEntry` representa o estado corrente de um atributo durante a execução de uma transação. Ela contém uma referência para o `PBox` do atributo, o valor original do atributo (`oldValue`) e o valor corrente do atributo (`newValue`).

Os métodos `setGarfo` e `getGarfo` são usados para acessar o atributo `garfo` dentro de uma transação. O método `setGarfo` primeiro verifica se o `PBox` do atributo está presente dentro do log da transação usando o método `contains` do `Log`. Se o log já contém uma `LogEntry` para o `garfo`, então essa `LogEntry` é alterada para que o atributo `newValue` contenha o novo valor vindo do método `set`. Se o log não contém uma `LogEntry` para o `garfo` no log da transação, uma nova `LogEntry` será adicionada, onde o atributo `oldValue` irá conter o valor atual do `garfo` e o atributo `newValue` recebe o valor vindo do método `setGarfo`.

O método `getGarfo` primeiro verifica se existe uma `LogEntry` para o `garfo` no log. Se existe, o método retorna o valor do atributo `newValue` que está no log, caso contrário, ele adiciona uma nova `LogEntry` no log, onde tanto o atributo `oldValue` quanto o atributo `newValue` irão conter o valor corrente do `garfo`.

5.2.4 O método `retry`

O método `retry` possui uma implementação simples:

```
public static STM<Void> retry() {
  return new STM<Void>( { Log l =>
    new STMResult(new Void(),l,STMRTS.RETRY)});
}
```

Este método simplesmente pára a transação corrente mudando o estado da transação para `STMRTS.RETRY`. Quando uma transação retorna `STMRTS.RETRY`, a transação é abortada (a implementação pode ser vista nos métodos `bind` e `then` na seção 5.2.2).

O método `atomic` é responsável por re-executar transações abortadas (Seção 5.2.6).

5.2.5 O método `orElse`

O método `orElse` é usado para combinar transações como alternativas. Se a primeira alternativa chama `retry`, todas as alterações feitas por ela devem ser invisíveis enquanto a segunda alternativa está sendo executada.

O método `orElse` é implementado de maneira similar ao método `orElse` na linguagem STM Haskell. Cada alternativa é executada usando um novo log chamado de *nested log*. Todas as escritas nos atributos são gravadas no *nested log* enquanto leituras devem consultar tanto o *nested log* quanto o log da transação. Se uma das alternativas terminar sem chamar `retry`, o *nested log* deve ser unido com o log da transação e o `STMResult` retorna `ACTIVE`, já contendo o log após a união dos dois. Se a primeira alternativa chama `retry`, então a segunda alternativa é executada usando um novo *nested log*. Se a segunda alternativa também chamar `retry`, todos os três logs são unidos e um `STMResult` contendo o log unido e o estado `RETRY` é retornado. Todos os logs devem ser unidos para que a transação abortada seja executada novamente quando um dos atributos acessados por essa transação seja modificado (Seção 5.2.6).

5.2.6 O método `atomic`

Uma ação STM pode ser executada chamando o método `atomic`:

```
public static <A> A atomic (STM<A> t)
```

O método `atomic` recebe como argumento uma transação e executa ela atomicamente, respeitando outras chamadas de atomic que estão sendo executadas concorrentemente.

Para executar uma ação transacional, o método `atomic` gera um novo log que é usado para invocar a transação:

```
Log l = STMRTS.generateNewLog();
STMResult<A> r = t.stm.invoke(l);
```

Como descrito anteriormente, uma transação invocada vai retornar um STMResult contendo o resultado da execução da transação, um log e o estado da transação que pode

ser ACTIVE ou RETRY. Um log contém, além das LogEntrys dos atributos, um bloqueio global que é usado pelo método atomic para atualizar as estruturas de dados compartilhadas enquanto efetiva uma transação (*commit*).

Se o resultado da transação executada é RETRY, o método atomic irá adquirir o bloqueio global e validar o log da transação. Se a validação do log for válida, a transação somente será executada quando algum objeto acessado por essa transação for modificado por outra transação. Se a validação foi inválida, a transação será reexecutada automaticamente porque alguma outra transação já modificou algum objeto que foi usado por essa transação. Para validar um log, é verificado em cada entrada do log o atributo oldValue, para ver se ele é igual ao valor corrente do objeto. O valor corrente do objeto é acessado através de um PBox, criado para cada objeto transacional.

Quando uma transação chama RETRY e seu log é válido, a transação deve ficar bloqueada até que um dos objetos usado por ela seja modificado. Para fazer isso, o método atomic cria um SyncObject para a transação.

Um SyncObject possui dois métodos:

```
public void block();
public void unblock();
```

Quando uma thread chama o método block de um SyncObject, a thread vai bloquear até que alguma outra thread chame o método unblock do mesmo objeto. Todos os PBox também contém uma lista de SyncObjects, onde os SyncObjects das threads que estão esperando por modificação nesse objeto serão inseridas. Então, sempre que uma thread é bloqueada, um SyncObject para essa thread deve ser inserido na lista de todos os PBox que a transacao acessou. Quando uma transação é efetivada, ela também percorre a lista acordando todas as threads esperando por modificacao no objeto. Dessa forma, quando a transação é acordada, significa que um dos TObjects presentes no log foi atualizado e a transação pode ser executada novamente com um novo log.

Se o resultado da transação executada é ACTIVE, o método adquire o bloqueio global e valida o log da transação. Se válido, a transação é efetivada modificando todos os Tobjects que estavam no log.

5.3 Implementação baseada no algoritmo TL2

Nesta seção é apresentada a segunda implementação do sistema transacional da linguagem CMTJava, baseada no algoritmo TL2 (*Transactional Locking II*).

5.3.1 Algoritmo TL2

O algoritmo TL2 (DICE; SHALEV; SHAVIT, 2006) é um algoritmo que combina o uso de um relógio global, para controle de versão dos dados, com uma técnica de aquisição do bloqueio dos objetos em tempo de efetivação.

A vantagem do algoritmo TL2 é que as transações sempre trabalham com um estado de memória consistente. Muitos dos sistemas transacionais existentes permitem que suas transações operem com um estado de memória inconsistente, isto é, as transações fazem leituras inválidas e seguem executando sem serem abortadas. Essas leituras inconsistentes podem causar efeitos inesperados nas execuções das transações. Um exemplo

disso é uma transação entrar em um loop infinito por causa de uma leitura inválida. Alguns sistemas transacionais utilizam técnicas para resolver esse problema, testando a cada passo do loop o estado de memória da transação para verificar se é validado ou não. Caso seja inválido, a transação será abortada. Com o algoritmo TL2 isso não ocorre, pois ele garante que as transações sempre vão operar com um estado de memória consistente.

Esse algoritmo é implementado usando uma detecção de conflitos tardia, de forma a ler os objetos e verificar se houve ou não conflitos ao fim da execução da transação. Para fazer a detecção de conflitos é usado um relógio global, um contador compartilhado pelas transações e incrementado sempre que uma transação é efetivada. Quando uma transação começa a executar, ela guarda o valor atual do relógio global em uma variável local chamada *read stamp*.

Cada atributo de um `TObject` deve possuir os seguintes atributos:

- *stamp*: contém o *read stamp* da última transação que escreveu nesse objeto;
- versão: contém a última versão do objeto;
- *lock*: contém o *lock* do objeto;

A transação "virtualmente" executa uma seqüência de leituras e escritas nos objetos. Virtualmente significa que nenhum objeto é realmente modificado. Ao invés disso, a transação usa um log de leitura para guardar o valor dos objetos lidos e um log de escrita para guardar os objetos que estão sendo modificados, incluindo o novo valor desses objetos.

Quando uma transação chama o método `get()` de um atributo de um `TObject`, o método primeiro verifica se esse atributo está no log de escrita da transação. Se sim, será retornado o que está no log. Se não, o método verifica se alguma outra transação adquiriu o `lock` desse atributo. Se alguma transação adquiriu o `lock` desse atributo é porque a transação está sendo efetivada e um conflito é detectado, abortando a transação que chamou o método `get()`. Se o `lock` do atributo está liberado, o método adiciona o atributo no log de leitura e o valor dele é retornado.

O método `set()` funciona de forma similar. Se o atributo não está no log de escrita, ele será inserido no log com o seu novo valor e esse valor é retornado.

De forma a sempre manter a memória consistente, o sistema checa se cada operação de leitura é válida. Essa tarefa é feita da seguinte forma: (1) para cada operação de leitura, o sistema compara o *stamp* do atributo com o *read stamp* da transação; (2) antes de uma transação ser efetivada o *stamp* dos atributos que estão no log de leitura da transação são novamente comparados com o *read stamp* da transação. Se em qualquer um deles o *stamp* do atributo for maior que o *read stamp* da transação é porque uma operação de escrita foi realizada no atributo (alterando seu *stamp*) durante a execução da transação, o que deixa a transação com um estado inconsistente, fazendo com que ela seja abortada.

Quando uma transação vai ser efetivada a primeira coisa que ela deve fazer é adquirir os *locks* dos atributos que estão no seu log de escrita. Se ela conseguir adquirir todos os locks, ela incrementa o relógio global usando uma operação CAS (*Compare-And-Swap*). Isso é necessário para saber o tempo exato em que a transação está sendo efetivada. Depois disso, a transação verifica se nenhuma transação adquiriu o `lock` de algum atributo que está em seu log de leitura e testa se o *stamp* dos atributos não é maior que o *read stamp* da transação. Se essa validação for realizada com sucesso, a transação

pode então ser efetivada definitivamente. A transação então, atualiza o *stamp* dos atributos que estão no seu log de escrita com o valor atual do relógio global, que ela mesma incrementou, e libera o *lock* deles.

Se ocorrer erro em algum dos passos acima, a transação é abortada, seus logs de escrita e leitura são descartados e os locks dos atributos são liberados (no caso de já terem sido adquiridos pela transação).

5.3.2 A mônada STM

Assim como na primeira implementação, ver Seção 5.2.2, a mônada para ações STM é implementada através de uma mônada de passagem de estados, mas nessa implementação esse estado é uma transação. A classe STM é implementada como segue:

```
class STM<A> {
  public { Trans => TResult } stm;
  public STM ({ Trans => TResult } stm) {
    this.stm = stm;
  }
}
```

Uma transação `STM<A>` é uma função que recebe um `Trans` como argumento e retorna um `TResult`. O `Trans` representa o estado corrente da transação durante sua execução e `TResult` descreve o novo estado da transação depois de sua execução.

A classe TResult possui três atributos:

```
class TResult <A> {
  A result;
  Trans newTrans;
  int state;
  (...)
}
```

Antes a classe `STMResult` possuia um log que representava o estado da transação. O novo `TResult` possui um atributo do tipo `Trans` que repesenta o estado da transação. Esse atributo contém tanto o log de escrita quanto o log de leitura.

O método `bind`, usado para compor ações transacionais, terá a seguinte implementação:

```
public static <A,B> STM<B> bind (STM<A> t, {A=>STM<B>} f){
  return new STM<B> ( {Trans t1 =>
    TResult<A> r1 = t.stm.invoke(t1);
    TResult<B> r2;
    if (r1.state == STMRTS.ACTIVE) {
      STM<B> r3 = f.invoke(r1.result);
      r2 = r3.stm.invoke(r1.newTrans);
    } else {
      r2 = new TResult(null, r1.newTrans, r1.state);
    }
    r2
  } );
}
```

Nessa nova implementação, o método `bind` executa uma ação `STM<A> t` passando uma transação (`t1`) como argumento, e não mais passando um log.

O método `then` é implementado de uma forma similar.

5.3.3 Compilando TObjects

Uma classe como a classe `Garfo` descrita na seção 4.1.1 é compilada para a classe descrita na Figura 5.3.

```
class Garfo implements TObject {

  private volatile Boolean garfo = true;
  FieldInfo<Boolean> garfoFieldInfo =
             new FieldInfo<Boolean> ({Boolean b => garfo = b;});

  public STM<Void> setGarfo (Boolean b) {
    return new STM<Void>({Trans t =>
      TResult r = null;
      if (garfoFieldInfo.lock.isLocked()) {
        r = new TResult(null, t, STMRTS.ABORTED);
      } else {
        t.writeSet.put(garfoFieldInfo,b);
        r = new TResult(new Void(), t, STMRTS.ACTIVE);
      }
      r
    });
  }

  public STM<Boolean> getGarfo() {
    return new STM<Boolean> ({Trans t =>
      TResult r = null;
      Boolean result = (Boolean)t.writeSet.get(garfoFieldInfo);
      if( result == null) {
        if (garfoFieldInfo.lock.isLocked()) {
          r = new TResult(null, t, STMRTS.ABORTED);
        } else {
          t.readSet.addElement(garfoFieldInfo);
          result = garfo;
          r = new TResult(result, t, STMRTS.ACTIVE);
        }
      } else {
        r = new TResult(result, t, STMRTS.ACTIVE);
      }
      if (garfoFieldInfo.stamp > t.readStamp) {
        r = new TResult(null, t, STMRTS.ABORTED);
      }
      r
    });
  }
}
```

Figura 5.3: Classe Garfo depois de ser compilada

Nessa implementação todo `TObject` possui um `FieldInfo` para cada um de seus atributos. Um `FieldInfo` possui, além de uma função para atualizar o valor do atributo, o `stamp` e o `lock` do atributo.

O estado de uma transação agora é definido por uma classe `Trans`:

```
public class Trans {
  public volatile long readStamp;
  public WriteSet writeSet;
  public ReadSet readSet;
```

```
   (...)
}
```

O primeiro atributo da classe `Trans` é o `readStamp`. Quando a transação inicia, o valor do relógio global é inserido no *readStamp* da transação para saber o tempo exato em que a transação começou a executar. O segundo atributo, `writeSet`, representa o log de escrita da transação. Enquanto o terceiro atributo, `readSet`, representa o log de leitura da transação.

Quando uma transação chama o método `get()` de um atributo, o método primeiro verifica se esse objeto está no `writeSet`. Se sim, o valor a ser retornado é o valor do `writeSet`. Se não, o método verifica se o atributo está locado por outra transação. Não estando locado, o atributo será inserido no `readSet` e seu valor corrente será retornado. Caso ele esteja locado, um conflito é detectado e a transação será abortada.

O método `set()` funciona de forma similar. Se o objeto não está no `writeSet`, ele será inserido com o seu novo valor e esse valor é retornado.

5.3.4 O método `atomic`

Da mesma forma da implementação anterior, o método `atomic` recebe como argumento uma transação e executa ela atomicamente. Para executar uma ação transacional, o método `atomic` gera um novo objeto do tipo `Trans` que é usado para invocar a transação:

```
Trans t = new Trans();
TResult<A> r = c.stm.invoke(t);
```

Uma transação invocada vai retornar um `TResult` contendo o resultado da execução da transação. Se ao fim da execução da transação seu *status* é `ACTIVE`, ela será efetivada. Para isso, o método `atomic` tenta adquirir os *locks* de todos os objetos que estão no `writeSet` da transação. Depois de adquirir todos os *locks*, o relógio global é incrementado e o `readSet` da transação será validado, para verificar se nenhum objeto está bloqueado por outra transação ou se teve seu valor alterado depois da transação ter lido esse objeto. Se o `readSet` for validado com sucesso a transação é efetivada, para isso, ela percorre todos os objetos do seu `writeSet` para atualizar o `stamp` dos objetos com o novo valor do relógio global, atualiza o o valor dele na memória e acorda todas as threads que estavam aguardando modificações nesses objetos. Depois de atualizar todos os objetos, o *lock* dos mesmos é liberado. Se ocorrer um erro durante o processo da efetivação, a transação será abortada.

Se o *status* da transação exectutada é `RETRY`, o método `atomic` irá validar o `readSet` e o `writeSet` da transação antes de reexecutá-la. Para validá-los, é verificado se todos os atributos possuem seus `locks` liberados e se o *stamp* dos atributos é menor que o *readstamp* da transação. Se sim, é porque nenhum atributo sofreu alteração. Então a transação fica aguardando nesses objetos para que quando houver uma modificação neles, a transação seja reexecutada. Se não, a transação é reexecutada automaticamente.

Para aguardar modificações nos objetos, dentro de cada `FieldInfo` existe uma lista com as threads que estão esperando. Sempre que uma transação tenta modificar essa lista de espera ela precisa adquirir o `lock` do mesmo, para que não conflite com outras transações que também podem tentar modificar essa lista ao mesmo tempo.

5.4 Comparação entre as implementações

Com o objetivo de comparar o desempenho dos dois sistemas transacionais desenvolvidos para a linguagem CMTJava, foi desenvolvido um experimento para testá-los. Esta seção apresenta o experimento desenvolvido, bem como uma análise do desempenho dos sistemas transacionais.

5.4.1 Experimento

O experimento desenvolvido é um programa que faz algumas operações em uma lista encadeada e conta quantas transações foram efetivadas em um intervalo de tempo.

As threads criadas ficam executando algumas ações de forma aleatória. Três ações podem ser executadas: inserir um elemento no fim da lista, retirar um elemento da lista ou procurar um elemento na lista. A ação de procurar um elemento na lista tem 50% de chance de ser executada, enquanto as operações de inserir e retirar possuem apenas 25% cada.

O experimento realizado conta o número de operações (inserir, retirar, procurar) realizadas em uma lista encadeada durante 20 segundos. Quanto maior o número de operações realizadas, melhor será o desempenho da execução.

A lista encadeada utilizada no experimento é uma lista que possui um ponteiro para o primeiro e para o último elemento e cada elemente aponta para o próximo. Sempre que um elemento é inserido, sua inserção é no fim da lista.

Foram rodados três programas. O primeiro, escrito na linguagem Java utilizando uma sincronização baseada em bloqueios. Essa implementação pode ser vista no Anexo A. O segundo, escrito na linguagem CMTJava usando o sistema transacional baseado na linguagem STM Haskell. O terceiro, escrito também na linguagem CMTJava, mas usando o sistema transacional baseado no algoritmo TL2. O programa escrito na linguagem CMTJava pode ser visto no Anexo B e a tradução para o sistema transacional baseado no algoritmo TL2 pode ser visto no Anexo C.

5.4.2 Ambiente de avaliação

Para rodar o experimento das listas encadeadas foi utilizada uma máquina AMD Turion 64 X2 1.80GHz com 2GB de memória RAM. Essa é uma máquina multicore com dois processadores. Para cada um dos três programas foram realizados dois experimentos, o primeiro usando uma thread e o segundo usando duas threads. Cada experimento foi executado três vezes e o resultado final foi uma média das três execuções.

5.4.3 Resultados

No primeiro experimento usando apenas uma thread e consequentemente apenas um processador da máquina, a segunda versão da CMTJava (baseada no algoritmo TL2) conseguiu realizar 80% mais operações que a primeira versão (baseada na STM Haskell), mas comparado com o programa escrito em Java puro ela perde. O programa em Java puro realiza em torno de 150% mais operações que a segunda versão da CMTJava.

No experimento usando duas threads que são executadas em paralelo pelo processador multicore, o número de operações realizadas pelo programa escrito em Java puro cai pela metade, enquanto que tanto a segunda versão da CMTJava, quanto a primeira, tem um incremento de 20% no número de operações realizadas no intervalo de 20 segundos.

Mesmo o desempenho do programa escrito em Java puro caindo pela metade, ele ainda é mais rápido que a CMTJava, realizando em torno de 70% mais operações.

A linguagem STM Haskell, quando testada com o mesmo tipo de lista encadeada (SULZMANN; LAM; MARLOW, 2009) também apresentou desempenho bem abaixo do desempenho de um mesmo programa escrito com uma sincronização baseada em bloqueios.

Um dos fatores que contribui para um desempenho abaixo do desempenho usando bloqueios é que existe todo um mecanismo transacional que é executado para dar suporte as transações, além disso, a lista encadeada usada neste experimento é uma lista 100% igual a uma lista encadeada de uma implementação sequencial feita em Java, o que difere é que na CMTJava as variáveis são transformadas em TObjects para serem acessadas dentro das transações, como pode ser visto no Anexo C. Em (SULZMANN; LAM; MARLOW, 2009), são apresentadas algumas otimizações na implementação de listas encadeadas usando STM Haskell. Essas otimizações levam em conta o funcionamento do sistema transacional e aumentam o desempenho das listas, tornando-as mais rápidas do que listas implementadas usando bloqueios, com a desvantagem de tornar a implementação das listas mais complicada.

Outro fator que deve ser levado em consideração é que a implementação do algoritmo TL2 no sistema transacional da CMTJava é uma implementação simples. Existem algumas otimizações para esse algoritmo que não foram aplicadas na CMTJava. Uma das otimizações é em cima do relógio global. Quando várias threads tentam atualizar o relógio global utilizando uma operação de CAS, um gargalo é introduzido no sistema.

5.5 Observações finais

Este capítulo apresentou as duas implementações da linguagem CMTJava. Além dos detalhes das implementações, também foi apresentado alguns resultados preliminares da execução de um experimento para comparar os dois sistemas transacionais desenvolvidos.

O capítulo seguinte é reservado para as conclusões do trabalho. Esse capítulo inclui, além das conclusões, as perspectivas de continuidade do trabalho e algumas publicações geradas contendo os principais resultados alcançados.

6 CONCLUSÃO

6.1 Conclusões

O mecanismo mais usado hoje para fazer sincronização de acesso em áreas compartilhadas de memória são os bloqueios (*locks*). Mas, sincronizações baseadas em bloqueios possuem algumas armadilhas que dificultam a programação e são propensas a erros (PEYTON JONES, 2007; HERLIHY; MOSS, 1993; ADL-TABATABAI; KOZYRAKIS; SAHA, 2006).

Memória Transacionais surgiram como uma alternativa aos métodos de sincronização baseados em bloqueios, usando uma nova abstração para programação concorrente baseada na idéia de transações. Ela traz para a programação concorrente os conceitos de controle de concorrência usados a décadas pela comunidade de banco de dados. Todo o controle de acesso à memória compartilhada é feito automaticamente pelo sistema transacional, o que facilita a programação concorrente porque o programador não precisa se preocupar em garantir a sincronização como nas abordagens baseadas em bloqueios.

Este trabalho buscou apresentar uma nova linguagem de domínio específico para programação de memórias transacionais em Java, denominada CMTJava, que visa facilitar a programação de máquinas multi-core. CMTJava (DU BOIS; ECHEVARRIA, 2009) foi criada adaptando conceitos da linguagem funcional STM Haskell (HARRIS et al., 2008) para um contexto orientado a objetos.

CMTJava pode ter seu sistema transacional implementado utilizando algoritmos para memórias transacionais. Este trabalho apresentou duas formas de implementar o sistema transacional da CMTJava utilizando dois algoritmos diferentes. Um baseado no sistema transacional da linguagem STM Haskell e outro baseado no algoritmo TL2.

Para comparar o desempenho dos dois sistemas desenvolvidos, foi utilizado um experimento que conta o número de operações (inserir, procurar e retirar) realizadas em uma lista encadeada durante 20 segundos. Nesse experimento a segunda versão da CMTJava, baseada no algoritmo TL2, se mostrou mais eficiente que a primeira, baseada na STM Haskell.

6.2 Continuidade do trabalho

A linguagem CMTJava foi desenvolvida tendo como inspiração a linguagem STM Haskell, com o objetivo de adptar as idéias da STM Haskell para um contexto orientado a objetos. Hoje a CMTJava conta com duas implementações para o seu sistema transacional, uma baseada no sistema transacional da própria STM Haskell e outra baseada no

algoritmo TL2.

Para dar continuidade ao trabalho, deve-se criar um programa que realize as traduções da linguagem CMTJava para Java puro + *closures*, conforme as regras de tradução descritas na Seção 5.1.3.

Essa etapa foi iniciada durante o trabalho, para isso foi escolhida uma ferramenta, chamada Java-Front (JAVA FRONT, 2009), que foi usada para fazer essas traduções (de CMTJava para Java puro + *closures*). Java-Front foi desenvolvida especificamente para realizar traduções em códigos Java, por isso se mostrou interessante para o projeto. Essa etapa foi iniciada, alguns testes com essa ferramenta foram feitos, mas não foi possível concluir todas as traduções.

Basicamente as tradução que devem ser feitas podem ser divididas em duas etapas:

- A primeira etapa consiste em adicionar os métodos de `get` e `set` para todos os atributos das classes que implementam a interface `TObject`;
- A segunda etapa consiste em traduzir os blocos `STM` para chamadas de `bind` e `then` descritas conforme as regras de tradução apresentadas na Seção 5.1.3.

Outro trabalho futuro seria a implementação das otimizações do algoritmo TL2. Uma das otimizações que podem ser feitas é em cima do relógio global que atualmente inclui um gargalo no sistema já que várias transações podem tentar modificá-lo ao mesmo tempo utilizando uma operação CAS.

6.3 Artigos publicados

No decorrer do desenvolvimento deste trabalho, algumas publicações geradas contendo os principais resultados alcançados foram divulgadas através dos trabalhos a seguir:

- ERAD 2009: ECHEVARRIA, Marcos. Uma Linguagem de Domínio Específico para Programação de Memórias Transacionais em Java. In: 9a Escola Regional de Alto Desempenho, 2009, Caxias do Sul-RS. ERAD 2009 - Fórum de Pós-Graduação da 9a Escola Regional de Alto Desempenho, 2009.
- DSL 2009: DU BOIS, André; ECHEVARRIA, Marcos. A Domain Specific Language for Composable Memory Transactions in Java. In: DSL, 2009. Anais. . . Springer, 2009. p.170-186. (Lecture Notes in Computer Science, v.5658).
- LTPD 2009: ECHEVARRIA, Marcos. Um Sistema de Versionamento Adiantado para a Linguagem CMTJava. In: Languages and Tools for Parallel and Distributed Programming, 2009, Gramado-RS.
- ERAD 2010: ECHEVARRIA, Marcos. Melhorando o Desempenho da CMTJava com Versionamento de Dados Adiantado In: 9a Escola Regional de Alto Desempenho, 2010, Passo Fundo-RS. ERAD 2010 - Fórum de Pós-Graduação da 9a Escola Regional de Alto Desempenho, 2010.

REFERÊNCIAS

ADL-TABATABAI, A.-R.; KOZYRAKIS, C.; SAHA, B. Unlocking concurrency. **ACM Queue**, [S.l.], v.4, n.10, p.24–33, 2006.

ALL ABOUT MONADS. WWW page, http://www.haskell.org/all_about_monads/html/index.html.

ANANIAN, C. S.; ASANOVIC, K.; KUSZMAUL, B. C.; LEISERSON, C. E.; LIE, S. Unbounded Transactional Memory. In: INTERNATIONAL CONFERENCE ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE (HPCA-11 2005) (11TH HPCA'05), 11., 2005, San Francisco, CA, USA. **Proceedings...** IEEE Computer Society, 2005. p.316–327.

ANDREWS, G. R. **Concurrent Programming — Principles and Practice**. Menlo Park, CA: Benjamin/Cummings, 1991.

CARLSTROM, B. D.; MCDONALD, A.; CHAFI, H.; CHUNG, J.; MINH, C. C.; KOZYRAKIS, C.; OLUKOTUN, K. The ATOMOS transactional programming language. **ACM SIGPLAN Notices**, [S.l.], v.41, n.6, p.1–13, June 2006.

DAMRON, P.; FEDOROVA, A.; LEV, Y.; LUCHANGCO, V.; MOIR, M.; NUSSBAUM, D. Hybrid transactional memory. In: ASPLOS, 2006. **Anais...** ACM, 2006. p.336–346.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional Locking II. In: DOLEV, S. (Ed.). **Distributed Computing (20th DISC'06)**. Stockholm: Springer-Verlag (New York), 2006. p.194–208. (Lecture Notes in Computer Science (LNCS), v.4167).

DU BOIS, A. Memórias Transacionais e Troca de Mensagens: Duas alternativas para a programação de Máquinas Multi-Core. In: SBC, P. A. (Ed.). **Escola Regional de Alto Desempenho**. 8.ed. [S.l.]: ERAD, 2008. p.43–76.

DU BOIS, A. R.; ECHEVARRIA, M. A Domain Specific Language for Composable Memory Transactions in Java. In: DSL, 2009. **Anais...** Springer, 2009. p.170–186. (Lecture Notes in Computer Science, v.5658).

HAMMOND, L.; WONG, V.; CHEN, M. K.; CARLSTROM, B. D.; DAVIS, J. D.; HERTZBERG, B.; PRABHU, M. K.; WIJAYA, H.; KOZYRAKIS, C.; OLUKOTUN, K. Transactional Memory Coherence and Consistency. In: ISCA, 2004. **Anais...** IEEE Computer Society, 2004. p.102–113.

HARRIS, T.; FRASER, K. Language support for lightweight transactions. **ACM SIGPLAN Notices**, [S.l.], v.38, n.11, p.388–402, Nov. 2003.

HARRIS, T.; MARLOW, S.; Peyton Jones, S. Haskell on a Shared-Memory Multiprocessor. In: HASKELL '05: PROCEEDINGS OF THE 2005 ACM SIGPLAN WORKSHOP ON HASKELL, 2005. **Anais...** ACM Press, 2005. p.49–61.

HARRIS, T.; MARLOW, S.; PEYTON JONES, S.; HERLIHY, M. Composable memory transactions. **Commun. ACM**, [S.l.], v.51, n.8, p.91–100, 2008.

HARRIS, T.; PLESKO, M.; SHINNAR, A.; TARDITI, D. Optimizing memory transactions. **ACM SIGPLAN Notices**, [S.l.], v.41, n.6, p.14–25, June 2006.

HERLIHY; LUCHANGCO; MOIR; SCHERER. Software Transactional Memory for Dynamic-sized Data Structures. In: PODC: 22TH ACM SIGACT-SIGOPS SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2003. **Anais...** [S.l.: s.n.], 2003.

HERLIHY; LUCHANGCO; MOIR; SCHERER. Software Transactional Memory for Dynamic-sized Data Structures. In: PODC: 22TH ACM SIGACT-SIGOPS SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2003. **Anais...** [S.l.: s.n.], 2003.

HERLIHY, M.; MOSS, J. E. B. Transactional Memory: Architectural Support for Lock-Free Data Structures. In: ISCA, 1993. **Anais...** [S.l.: s.n.], 1993. p.289–300.

HUCH, F.; KUPKE, F. A High-Level Implementation of Composable Memory Transactions in Concurrent Haskell. In: IFL, 2005. **Anais...** [S.l.: s.n.], 2005. p.124–141.

HUTTON, G.; MEIJER, E. Monadic Parsing in Haskell. **J. Funct. Program**, [S.l.], v.8, n.4, p.437–444, 1998.

JAVA CLOSURES. WWW page, http://www.javac.info/.

JAVA FRONT. WWW page, http://strategoxt.org/Stratego/JavaFront.

KUMAR, S.; CHU, M.; HUGHES, C. J.; KUNDU, P.; NGUYEN, A. Hybrid transactional memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (11TH PPOPP'2006), ACM SIGPLAN NOTICES, 2006, New York, New York, USA. **Proceedings...** ACM SIGPLAN 2006, 2006. p.209–220. Published as Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (11th PPOPP'2006), ACM SIGPLAN Notices, volume 41, number 3.

LARUS, J.; RAJWAR, R. **Transactional Memory**. [S.l.]: Morgan & Claypool, 2006.

LEE. The Problem with Threads. **COMPUTER: IEEE Computer**, [S.l.], v.39, 2006.

MCDONALD, A.; CHUNG, J.; CHAFI, H.; MINH, C. C.; CARLSTROM, B. D.; HAMMOND, L.; KOZYRAKIS, C.; OLUKOTUN, K. Characterization of TCC on Chip-Multiprocessors. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURE AND COMPILATION TECHNIQUES (14TH PACT'05), 14., 2005, Saint Louis, MO, USA. **Proceedings...** IEEE Computer Society, 2005. p.63–74.

PETERSON, J.; HUDAK, P.; ELLIOTT, C. Lambda in Motion: Controlling Robots with Haskell. **Lecture Notes in Computer Science**, [S.l.], v.1551, p.91–105, 1999.

PEYTON JONES, S. Special Issue: Haskell 98 Language and Libraries. **Journal of Functional Programming**, [S.l.], v.13, Jan. 2003.

PEYTON JONES, S. Beautiful Concurrency. In: ORAM, A.; WILSON, G. (Ed.). **Beautiful Code**. Sebastopol, CA 95472: O'Reilly & Associates, Inc., 2007. p.385–406. ch. 24.

RAJWAR, R.; GOODMAN, J. A. Transactional Execution: Toward Reliable, High-Performance Multithreading. **IEEE Micro**, [S.l.], v.23, n.6, p.117–125, 2003.

RAJWAR, R.; GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In: ASPLOS, 2002. **Anais...** [S.l.: s.n.], 2002. p.5–17.

RAJWAR, R.; HERLIHY, M.; LAI, K. K. Virtualizing Transactional Memory. In: ANN. INTL SYMP. ON COMPUTER ARCHITECTURE (32TH ISCA'05), ACM COMPUTER ARCHITECTURE NEWS (CAN), 32., 2005, Madison, Wisconsin, USA. **Proceedings...** ACM SIGARCH / IEEE Computer Society, 2005. p.494–505. Published as Proc. 32th Ann. Intl Symp. on Computer Architecture (32th ISCA'05), ACM Computer Architecture News (CAN), volume 33.

SHAVIT, N.; TOUITOU, D. Software Transactional Memory. In: PODC, 1995. **Anais...** [S.l.: s.n.], 1995. p.204–213.

SULZMANN, M.; LAM, E. S. L.; MARLOW, S. Comparing the performance of concurrent linked-list implementations in Haskell. In: DAMP, 2009. **Anais...** ACM, 2009. p.37–46.

TANENBAUM, A. S.; WOODHULL, A. S. **Operating systems**: design and implementation. 3.ed. pub-PEARSON-PH:adr: Pearson Prentice Hall, 2006.

ANEXO A LISTA ENCADEADA IMPLEMENTADA EM JAVA USANDO SINCRONIZAÇÃO BASEADA EM BLOQUEIOS

```
import java.io.*;
import java.util.*;

class ListJava {
  public static void main (String args[])throws Exception {
    List l = new List();
    int commits = 0;
    int nthreads = Integer.parseInt(args[0]);
    for(int i=0;i<=50;i++) {
      v = (Integer) (int)(300*Math.random()+1);
      l.addToTail(v);
    }
    ThreadRandom vet[] = new ThreadRandom[200];
    for(int i = 0;i<nthreads; i++) {
      vet[i] = new ThreadRandom(l);
      vet[i].start();
    }
    Thread.sleep(20000);
    for(int i = 0;i<nthreads; i++) {
      vet[i].cond = false;
    }
    for(int i = 0;i<nthreads; i++) {
      vet[i].join();
    }
    for(int i = 0;i<nthreads; i++) {
      commits = commits + vet[i].commits;
    }
    System.out.println("Numero de Commits: "+ commits);
  }
}

class ThreadRandom extends Thread {
```

```
  public int commits=0;
  volatile boolean cond = true;
  private List l;
  ThreadRandom(List l) {
    this.l = l;
  }
  public void run() {
    Integer v; int operacao;
    while(cond) {
      operacao = 1 + (int) (Math.random()*4);
      v = (Integer) (int)(200*Math.random()+1);
      if(operacao ==1) {
        l.addToTail(v);
        commits++;
      } else {
        if(operacao==2) {
          l.delete(v);
          commits++;
        } else {
          l.find(v);
          commits++;
        }
      }
    }
  }
}

class List {
  private Node headList;
  private Node tailList;
  private void setTailList(Node n){
    this.tailList = n;
  }
  public List(){
    Node n = new Node(null,null);
    headList = new Node(null, n);
    this.setTailList(n);
  }
  public synchronized void addToTail(Integer n){
    Node a = new Node(null, null);
    tailList.setNext(a);
    tailList.setVal(n);
    tailList=a;
  }
  public synchronized boolean find(Integer i){
    return find2(headList.getNext(), i);
  }
  private synchronized boolean find2(Node curNode, Integer i){
    if(curNode.getVal()==null) return false;
    if(curNode.getVal()==i) return true;
```

```
    return find2(curNode.getNext(), i);
  }
  public synchronized boolean delete(Integer i){
    return delete2(headList, i);
  }
  private synchronized boolean delete2
                      (Node prevNode, Integer i){
    Node curNode = prevNode.getNext();
    if(curNode==null) return false;
    if(curNode.getVal() == null) return false;
    if(!curNode.getVal().equals(i)) {
      return delete2(curNode, i);
    } else{
      prevNode.setNext(curNode.getNext());
      return true;
    }
  }
}

class Node{
  private Integer val;
  private Node next;
  public Node(Integer val, Node next){
    this.val=val;
    this.next=next;
  }
  public Integer getVal() {
    return this.val;
  }
  public Node getNext(){
    return this.next;
  }
  public void setVal(Integer val){
    this.val = val;
  }
  public void setNext(Node n){
    this.next = n;
  }
}
```

ANEXO B LISTA ENCADEADA IMPLEMENTADA NA CMTJAVA

```
import stm.*;

class ListCMT {
  public static void main (String args[])throws Exception {
    List l = new List();
    int commits = 0;
    int nthreads = Integer.parseInt(args[0]);
    Integer v;
    for(int i=0;i<=50;i++) {
      v = (Integer) (int)(300*Math.random()+1);
      STMRTS.atomic(l.addToTail(v));
    }
    ThreadRandom vet[] = new ThreadRandom[200];
    for(int i = 0;i<nthreads; i++) {
      vet[i] = new ThreadRandom(l);
      vet[i].start();
    }
    Thread.sleep(20000);
    for(int i = 0;i<nthreads; i++) {
      vet[i].cond = false;
    }
    for(int i = 0;i<nthreads; i++) {
      vet[i].join();
    }
    for(int i = 0;i<nthreads; i++) {
      commits = commits + vet[i].commits;
    }
    System.out.println("Numero de Commits: "+ commits);
  }
}

class ThreadRandom extends Thread {
  public int commits=0;
  volatile boolean cond = true;
  private List l;
  ThreadRandom(List l) {
```

```
    this.l = l;
  }
  public void run() {
    Integer v; int operacao;
    while(cond) {
      operacao = 1 + (int) (Math.random()*4);
      v = (Integer) (int)(200*Math.random()+1);
      if(operacao ==1) {
        STMRTS.atomic(l.addToTail(v));
        commits++;
      } else {
        if(operacao==2) {
          STMRTS.atomic(l.delete(v));
          commits++;
        } else {
          STMRTS.atomic(l.find(v));
          commits++;
        }
      }
    }
  }
}

class List extends TObject{
  private Node headList;
  private Node tailList;
  public List(){
    Node n = new Node(null,null);
    headList = new Node(null, n);
    tailList = n;
  }
  public STM<Void> AddToTail(Integer n) {
    STM<Boolean> r = STM{
      Node a <- STMRTS.stmReturn (new Node(null,null));
      tailList.setNext(a);
      tailList.setVal(n);
      this.setTailList(a);
    }
    return r;
  }
  public STM<Boolean> find(Integer i) {
    STM<Boolean> r = STM{
      Node head <- headList.getNext();
      find2(head, i);
    }
    return r;
  }
  public STM<Boolean> find2(Node curNode, Integer i) {
    STM<Boolean> r = STM{
      Integer cv <- curNode.getVal();
```

```
      if(cv == null) {
        STMRTS.stmRetun(false);
      } else {
        if(cv.equals(i)) {
          STMRTS.stmReturn(true);
        } else {
          Node next = curNode.getNext();
          find2(next, i);
        }
      }
    }
    return r;
  }
  public STM<Boolean> delete(Integer v) {
    STM<Boolean> r = STM{
      delete2(headList, i);
    }
    return r;
  }
  public STM<Boolean> delete2(Integer v) {
    STM<Boolean> r = STM{
      Node curNode <- prevNode.getNext();
      if(curNode==null) {
        STMRTS.stmReturn(false);
      } else {
        Integer cv <- curNode.getVal();
        if(cv==null) {
          STMRTS.stmReturn(false);
        } else {
          if(!cv.equals(i)) {
            delete2(curNode, i));
          } else {
            Node next <- curNode.getNext();
            prevNode.setNext(next);
            STMRTS.retReturn(true);
          }
        }
      }
    }
    return r;
  }
}

class Node extends TObject {
  private Integer val;
  private Node next;
  public Node(Integer val, Node next) {
    this.val=val;
    this.next=next;
  }
```

```
}
```

ANEXO C LISTA ENCADEADA IMPLEMENTADA NA CMTJAVA, TRADUZIDA PARA O SISTEMA TRANSACIONAL BASEADO NO ALGORITMO TL2

```
import stm.*;

class ListCMT {
  public static void main (String args[])throws Exception {
    List l = new List();
    int commits = 0;
    int nthreads = Integer.parseInt(args[0]);
    Integer v;
    for(int i=0;i<=50;i++) {
      v = (Integer) (int)(300*Math.random()+1);
      STMRTS.atomic(l.addToTail(v));
    }
    ThreadRandom vet[] = new ThreadRandom[200];
    for(int i = 0;i<nthreads; i++) {
      vet[i] = new ThreadRandom(l);
      vet[i].start();
    }
    Thread.sleep(20000);
    for(int i = 0;i<nthreads; i++) {
      vet[i].cond = false;
    }
    for(int i = 0;i<nthreads; i++) {
      vet[i].join();
    }
    for(int i = 0;i<nthreads; i++) {
      commits = commits + vet[i].commits;
    }
    System.out.println("Numero de Commits: "+ commits);
  }
}

class ThreadRandom extends Thread {
```

```
  public int commits=0;
  volatile boolean cond = true;
  private List l;
  ThreadRandom(List l) {
    this.l = l;
  }
  public void run() {
    Integer v; int operacao;
    while(cond) {
      operacao = 1 + (int) (Math.random()*4);
      v = (Integer) (int)(200*Math.random()+1);
      if(operacao ==1) {
        STMRTS.atomic(l.addToTail(v));
        commits++;
      } else {
        if(operacao==2) {
          STMRTS.atomic(l.delete(v));
          commits++;
        } else {
          STMRTS.atomic(l.find(v));
          commits++;
        }
      }
    }
  }
}

class List{
  private Node headList;
  private Node tailList;
  FieldInfo<Node> headListFieldInfo;
  FieldInfo<Node> tailListFieldInfo;
  List() {
    Node n = new Node(null,null);
    headList = new Node(null, n);
    tailList = n;
    headListFieldInfo = new FieldInfo<Node>
                        ({Node a => headList = a; });
    tailListFieldInfo = new FieldInfo<Node>
                        ({Node a => tailList = a; });
  }
  public STM<stm.Void> setHeadList (Node n) {
    return new STM<stm.Void>({Trans t =>
      TResult r = null;
      if (headListFieldInfo.lock.isLocked()) {
        r = new TResult(null, t, STMRTS.ABORTED);
      }else{
        t.writeSet.put(headListFieldInfo, n);
        r = new TResult(new stm.Void(), t, STMRTS.ACTIVE);
      }
```

```
    r
  });
}
public STM<Node> getHeadList() {
  return new STM<Node> ({Trans t =>
    TResult r = null;
    Node result = (Node)t.writeSet.get(headListFieldInfo);
    if(result == null) {
      if (headListFieldInfo.lock.isLocked()) {
        r = new TResult(null, t, STMRTS.ABORTED);
      } else {
        t.readSet.addElement(headListFieldInfo);
        result = headList;
        r = new TResult(result, t, STMRTS.ACTIVE);
      }
    } else {
      r = new TResult(result, t, STMRTS.ACTIVE);
    }
    if (!headListFieldInfo.validate(t.readStamp)){
      r = new TResult(null, t, STMRTS.ABORTED);
    }
    r
  });
}
public STM<stm.Void> setTailList (Node n) {
  return new STM<stm.Void>({Trans t =>
    TResult r = null;
    if (tailListFieldInfo.lock.isLocked()) {
      r = new TResult(null, t, STMRTS.ABORTED);
    } else {
      t.writeSet.put(tailListFieldInfo,n);
      r = new TResult(new stm.Void(), t, STMRTS.ACTIVE);
    }
    r
  });
}
public STM<stm.Void> addToTail(Integer n) {
  STM<stm.Void> r = STMRTS.bind(STMRTS.ret
    (new Node( null,null)), { Node a =>
    STMRTS.then (tailList.setNext(a),
    STMRTS.then (tailList.setVal(n), this.setTailList(a)))
  });
  return r;
}
public STM<Boolean> find(Integer i) {
  STM<Boolean> r = STMRTS.bind(
      headList.getNext(), {Node head => find2(head, i)});
  return r;
}
public STM<Boolean> find2(Node curNode, Integer i) {
```

```
    STM<Boolean> r = STMRTS.bind(curNode.getVal(),
      {Integer cv => (cv == null)? STMRTS.ret(false) :
      ((cv.equals(i))? STMRTS.ret(true) :
      STMRTS.bind(curNode.getNext(),
      {Node next => (find2(next, i))})
    )});
    return r;
  }
  public STM<Boolean> delete(Integer i) {
    STM<Boolean> r = delete2(headList, i);
    return r;
  }
  public STM<Boolean> delete2(Node prevNode, Integer i) {
    STM<Boolean> r = STMRTS.bind(prevNode.getNext(),
      {Node curNode => (curNode==null)? STMRTS.ret(false) :
      STMRTS.bind( curNode.getVal(), {Integer cv =>
      (cv == null) ? STMRTS.ret(false) :
      (!cv.equals(i)) ? delete2(curNode, i) :
      STMRTS.bind( curNode.getNext(), {Node next =>
      STMRTS.then(prevNode.setNext(next),
      STMRTS.ret(true))})})
    });
    return r;
  }
}

class Node{
  private Integer val;
  private Node next;
  FieldInfo<Integer> valFieldInfo;
  FieldInfo<Node> nextFieldInfo;
  Node(Integer v, Node n) {
    this.val=v;
    this.next=n;
    valFieldInfo = new FieldInfo<Integer>
                   ({Integer i => val = i;});
    nextFieldInfo = new FieldInfo<Node>
                    ({Node x => next = x;});
  }
  public STM<stm.Void> setVal (Integer i) {
    return new STM<stm.Void>({Trans t =>
      TResult r = null;
      if (valFieldInfo.lock.isLocked()) {
        r = new TResult(null, t, STMRTS.ABORTED);
      } else {
        t.writeSet.put(valFieldInfo,i);
        r = new TResult(new stm.Void(), t, STMRTS.ACTIVE);
      }
      r
    });
```

```
}
public STM<stm.Void> setNext (Node n) {
  return new STM<stm.Void>({Trans t =>
    TResult r = null;
    if (nextFieldInfo.lock.isLocked()) {
      r = new TResult(null, t, STMRTS.ABORTED);
    } else {
      t.writeSet.put(nextFieldInfo,n);
      r = new TResult(new stm.Void(), t, STMRTS.ACTIVE);
    }
    r
  });
}
public STM<Node> getNext() {
  return new STM<Node> ({Trans t =>
    TResult r = null;
    Node result = (Node)t.writeSet.get(nextFieldInfo);
    if( result == null) {
      if (nextFieldInfo.lock.isLocked()) {
        r = new TResult(null, t, STMRTS.ABORTED);
      } else {
        t.readSet.addElement(nextFieldInfo);
        result = next;
        r = new TResult(result, t, STMRTS.ACTIVE);
      }
    } else {
      r = new TResult(result, t, STMRTS.ACTIVE);
    }
    if (!nextFieldInfo.validate(t.readStamp)){
      r = new TResult(null, t, STMRTS.ABORTED);
    }
    r
  });
}
public STM<Integer> getVal() {
  return new STM<Integer> ({Trans t =>
    TResult r = null;
    Integer result = (Integer)t.writeSet.get(valFieldInfo);
    if( result == null) {
      if (valFieldInfo.lock.isLocked()) {
        r = new TResult(null, t, STMRTS.ABORTED);
      } else {
        t.readSet.addElement(valFieldInfo);
        result = val;
        r = new TResult(result, t, STMRTS.ACTIVE);
      }
    } else {
      r = new TResult(result, t, STMRTS.ACTIVE);
    }
    if (!valFieldInfo.validate(t.readStamp)){
```

```
        r = new TResult(null, t, STMRTS.ABORTED);
      }
      r
    });
  }
}
```