Pontifícia Universidade Católica do Paraná - PUCPR
Programa de Pós Graduação em Informática Aplicada - PPGIA
Centro de Ciências Exatas e de Tecnologia - CCET

LEONARDO REIS NUNES

# CONTROLE DE CONCORRÊNCIA PARA JOGOS ELETRÔNICOS BASEADOS EM MÁQUINAS VIRTUAIS

Curitiba
2005

Pontifícia Universidade Católica do Paraná - PUCPR
Programa de Pós Graduação em Informática Aplicada - PPGIA
Centro de Ciências Exatas e de Tecnologia - CCET

LEONARDO REIS NUNES

# CONTROLE DE CONCORRÊNCIA PARA JOGOS ELETRÔNICOS BASEADOS EM MÁQUINAS VIRTUAIS

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná, como parte dos requisitos para obtenção do título de Mestre em Informática Aplicada.

Área de concentração: Sistemas Distribuídos

Orientador: Prof. Dr. Alcides Calsavara

Curitiba
2005

Nunes, Leonardo Reis.
Controle de Concorrência para Jogos Eletrônicos baseados em Máquinas Virtuais. Curitiba, 2005. 171 Páginas

Dissertação (Mestrado) – Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada.

1. Assincronismo. 2. Concorrência. 3. Máquinas Virtuais. 4. Jogos Eletrônicos.

I.Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática Aplicada.

# Agradecimentos

Agradeço ao meu orientador, Alcides, pelo incentivo, apoio, dedicação e extrema ajuda que prestou no desenvolvimento deste trabalho. Nossas reuniões geraram um bom fruto!

Agradeço minha esposa, Rosana, pela compreensão e apoio, e ao meu filho, Felipe, por entender que o período de pouca atenção e tempo por parte de seu pai era passageiro.

Agradeço ao amigo e sócio, Matias, pelas conversas sempre instrutivas e pelas dicas sobre desenvolvimento de jogos.

Agradeço aos amigos e colegas de mestrado, Aron, Nodari, Akatsu, Agnaldo, Juarez e Kolb pelo incentivo e apoio.

Agradeço a minha família e amigos, pelo apoio que sempre me deram.

Agradeço a todos os que não estão nesta lista e que de alguma forma contribuíram para que eu chegasse até aqui.

# Sumário

# Lista de figuras

# Lista de tabelas

# Abstract

The domain of electronic games has shown an important growth in the last years. Building an electronic game is a complex task and requires a proper development toolset, including a runtime support and a programming language. Both the engine and the middleware normally employed in the development of an electronic game should meet some basic requirements, such as portability, multilanguage, asynchronism and concurrency control. Moreover, an ideal concurrency model must provide fine control, determinism and a programming interface oriented to game development. However, present concurrent languages and systems do not fulfill all those requirements in an integrated fashion and, in general, they do not provide a simple and safe programming interface. This dissertation presents a concurrency model for the Virtuosi system, an object-oriented middleware based on a collection of cooperating virtual machines, and shows its usability in the context of the game development area. The proposed model includes – in a coherent and homogeneous fashion – the main concurrency mechanisms found in the literature and in modern systems and languages. It makes it possible to use the mechanisms for asynchronism and concurrency control either implicitly or explicitly, thus providing a high degree of flexibility. It can be extended by users to include new styles of timers to fulfill electronic games specific needs. Finally, the model is formalized and mapped into a programming language, thus permitting more robust code with respect to concurrency control.

**Keywords**: concurrency control, asynchronism, electronic games, virtual machines, middleware.

# Resumo

A área de jogos eletrônicos obteve um crescimento muito grande nos últimos anos. A tarefa de construção de um jogo eletrônico, por se tratar de uma tarefa complexa, necessita de um suporte adequado nos diversos níveis relacionados com o desenvolvimento, que vão desde a arquitetura de execução até as linguagens de programação utilizadas. Tanto a *engine* quanto o *middleware* utilizados no desenvolvimento de um jogo normalmente devem respeitar alguns requisitos básicos: portabilidade, multilinguagem, assincronismo e controle de concorrência. Um modelo de concorrência ideal para um jogo eletrônico requer: controle, determinismo e uma interface de programação voltada ao desenvolvimento de jogos. Entretanto os sistemas e linguagens concorrentes atuais não atendem a todos estes requisitos de forma integrada e, de forma geral, não oferecem uma interface de uso simples e seguro. Esta dissertação apresenta um modelo de concorrência para a Virtuosi, um *middleware* orientado a objeto e baseado em máquinas virtuais cooperantes, e mostra sua usabilidade no contexto da área de desenvolvimento de jogos eletrônicos. O modelo contempla de forma coerente e homogênea os principais mecanismos de concorrência documentados na literatura e disponíveis nos sistemas e linguagens modernos. Possibilita o uso dos mecanismos de assincronismo e de controle de concorrência de forma implícita ou explícita, provendo alto grau de flexibilidade. Permite a adição de formas de temporização adequadas ao domínio de jogos eletrônicos. Finalmente, o modelo está formalizado e representado na forma de linguagem de programação, o que possibilita a escrita de código mais robusto com relação ao controle de concorrência.

**Palavras-chaves**: concorrência, assincronismo, jogos eletrônicos, máquinas virtuais, *middleware*

# 1 Introdução

A área de jogos eletrônicos obteve um crescimento muito grande nos últimos anos. Além do enorme crescimento financeiro desta indústria no cenário mundial, que em 2004 chegou à marca de 7.3 bilhões de dólares somente nos Estados Unidos [ESA, 2005], o crescimento em produção cientifica também foi notável. Os jogos eletrônicos impulsionam diretamente pesquisas em diversas áreas do conhecimento, principalmente nas ciências da computação, onde possibilitam avanços significativos no desenvolvimento de hardware e software.

Considerando estes avanços obtidos nos últimos anos, conforme apresentado em [BLOW, 2004], pode-se afirmar que o desenvolvimento de um jogo eletrônico tornou-se uma tarefa extremamente complexa, pois utiliza os mais avançados recursos e algoritmos das áreas de computação gráfica, inteligência artificial, sistemas de tempo real, redes de computadores, produção de áudio e vídeo, simulação, persistência, sistemas distribuídos e linguagens de programação.

Uma vertente em especial da área de jogos é a chamada MMOG (*Massive Multiplayer Online Game)*, na qual um jogo pode ser considerado como uma espécie de mundo virtual onde milhares de jogadores interagem com o mundo virtual e entre si, simultaneamente. Este tipo de aplicação, um MMOG, é um exemplo claro de um sistema distribuído de larga escala e extremamente complexo. A figura 1.1 (explicada em detalhes em [BLOW, 2004]) mostra os diversos módulos e áreas de conhecimento necessários na construção de um jogo do tipo MMOG. Pode-se observar o alto grau de complexidade do desenvolvimento de um jogo eletrônico.

A tarefa de construção de um jogo eletrônico, por se tratar de uma tarefa complexa, necessita de um suporte adequado nos diversos níveis relacionados com o desenvolvimento, que vão desde a arquitetura de execução até as linguagens de programação utilizadas.

Atualmente é possível encontrar três níveis de código no desenvolvimento de jogos eletrônicos, são eles: *códigos de script*, *código de jogo* e *código da engine*. Código de script e código de jogo, em geral, são referentes a lógicas de alto nível relacionadas com definições específicas do próprio jogo. Já o código da *engine* trata questões de domínio específico, relacionadas sempre com os mecanismos básicos de funcionamento do jogo.

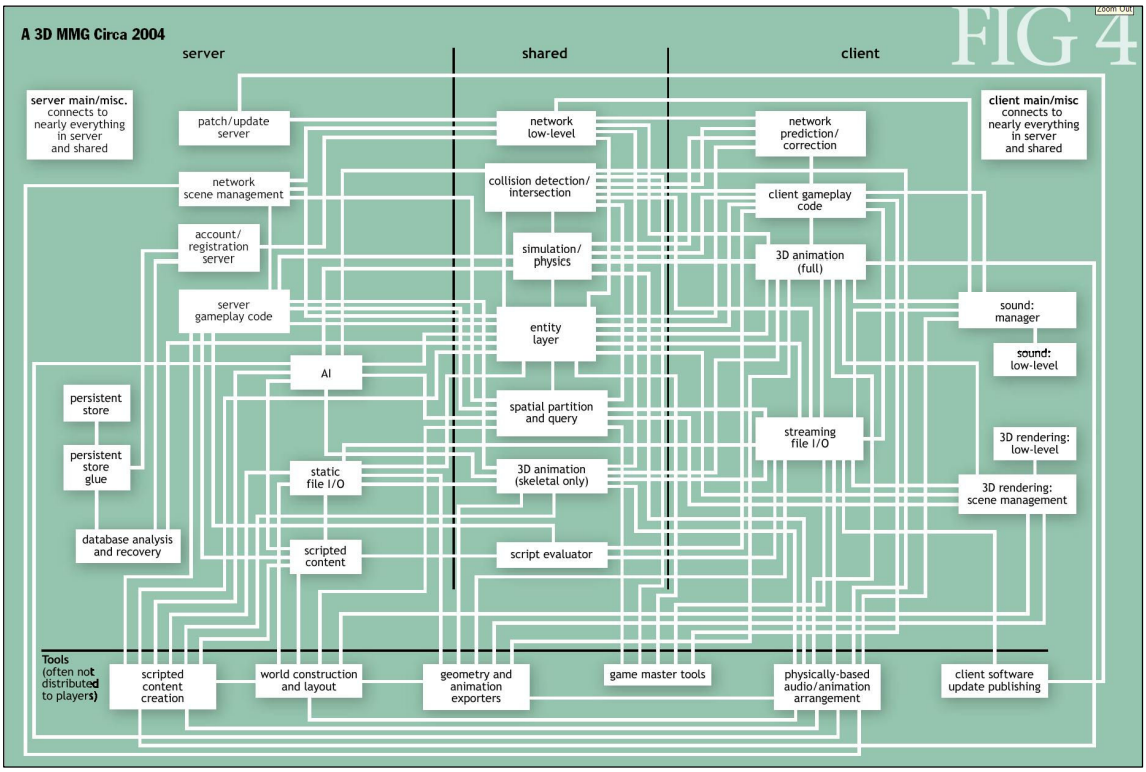


**Figura 1. 1 - *Overview* de um jogo MMOG [BLOW, 2004]**

Normalmente, em jogos para *um único usuário*[1], a *engine* é responsável por toda a matemática, física, áudio, simulação, *renderização*[2], suporte a scripts e I/O de um jogo. Já em jogos para *múltiplos usuários*[3], a *engine* pode ser acoplada a um *middleware*, e oferecer também o acesso a objetos distribuídos e serviços diversos como, por exemplo, um serviço de transação.

Tanto a *engine* quanto o *middleware* utilizados no desenvolvimento de um jogo normalmente devem respeitar alguns requisitos básicos:

- Portabilidade: Muitos jogos são desenvolvidos para serem executados em sistemas operacionais heterogêneos como Windows e Linux, e até mesmo em

[1] Do inglês, *single-player*
[2] Do inglês, *rendering*
[3] Do inglês, *multi-player*

plataformas heterogêneas, como PCs e consoles como o Playstation 2 da Sony ou o Xbox da Microsoft. Por conta da grande heterogeneidade desta indústria, a portabilidade pode ser considerada um fator determinante para o alcance dos diversos mercados existentes.

- Multilinguagem: No desenvolvimento de um jogo, normalmente ocorre de a lógica de jogo atuar sobre a *engine*/*middleware*. Neste cenário, todo controle de objetos do jogo tais como descrição do mundo virtual, personagens, mapas, objetos em geral pode ser realizado utilizando uma segunda linguagem, que não a linguagem hospedeira da *engine*/*middleware*, mas sim uma linguagem de script ou uma linguagem de mais alto nível. Uma outra motivação para a multilinguagem é que normalmente uma equipe de desenvolvimento de jogo é grande e conta com as mais diversas mentes, sendo que nem todos podem ser considerados exímios programadores ou conhecedores da linguagem na qual a *engine*/*middleware* foram desenvolvidos. Uma linguagem de mais alto nível pode aumentar em muito a curva de aprendizado de uma equipe de desenvolvimento de jogos, além de possibilitar um ganho de velocidade no desenvolvimento (produtividade).
- Suporte a *Threading*: Uma *engine* ou um *middleware* resolve diversos problemas e oferece diversos serviços. Em geral, possui muitos módulos que resolvem problemas específicos e os mesmos interagem entre si. Cada módulo tem suas próprias características e é comum a necessidade de operações assíncronas. Tais operações acontecem por meio de *Threads*. Exemplos de problemas que podem ser resolvidos com a utilização de *threads* podem ser: O recebimento e gerenciamento de conexões de rede, o cálculo de estatísticas para o módulo de inteligência artificial, a persistência de estado de jogo, simulação, renderização e muitos outros.

O requisito de suporte a múltiplas *threads* em um sistema portável e multilinguagem é um requisito difícil de obter. Isso se deve ao fato de que cada sistema operacional/arquitetura assim como cada linguagem de programação pode oferecer seu próprio modelo de *threading*, com suas próprias peculiaridades. Deve-se lembrar também que os modelos de concorrência comumente utilizados são modelos genéricos que visam atender as necessidades de

concorrência da grande maioria das aplicações, e não deste nicho específico que é esta área de jogos eletrônicos. O controle de concorrência ideal para um jogo eletrônico requer mais que este padrão normalmente oferecido, e podem-se evidenciar os seguintes requisitos para um modelo de concorrência adequado ao desenvolvimento de jogos:

- Controle: Diversos são os usos de *threads* nos módulos de uma *engine* de um jogo. Alguns destes usos necessitam de um controle total da execução, neste caso, o modelo de controle adequado é o modelo denominado não preemptivo, ou cooperativo [WIKIPEDIA, 1]. Porém, para alguns tipos de tarefas o modelo mais adequado de controle seria o modelo preemptivo [WIKIPEDIA, 2]. Em ambos os usos, os modelos devem também possibilitar um controle baseado em prioridades.
- Determinismo: A grande maioria dos modelos de *threading* não segue um modelo determinista. Em um modelo determinista, quando uma *thread* está em contenção, ou seja, aguardando alguma condição ou evento para retomada de sua execução, a ordem de retomada deve ser uma ordem definida previsível, e não uma ordem aleatória. Um exemplo de um modelo determinístico para retomada de execução seria um modelo onde as *threads* que aguardam em contenção retomam sua execução através de uma ordem definida, seja temporal (na mesma ordem que entraram na fila de contenção) ou por prioridades.
- API voltada ao desenvolvimento de jogos: Grande parte dos jogos eletrônicos precisam de alto desempenho (existem tipos específicos de jogos que não possuem esta necessidade), portanto o controle de concorrência precisa ocorrer através de mecanismos que ofereçam pouco impacto no desempenho da aplicação. Mecanismos como *travas*[4] leves, que possuem suporte direto no hardware, são exemplos deste tipo de mecanismo. Um outro exemplo de API voltada a jogos poderia ser a utilização de mecanismos de temporização baseados na *taxa de atualização de vídeo*[5] do jogo, e não em segundos ou milisegundos. Desta forma poderíamos fazer com que uma *thread* aguardasse por 10 quadros do jogo, ou até mesmo por uma colisão de entidades do jogo.

[4] Do inglês, *locks*
[5] Do inglês, *frame-rate*

> Este tipo de API, voltada às necessidades de desenvolvimento de um jogo, poderia tornar a construção de um jogo mais simples e fácil.

Um fato muito interessante relacionado com a área de jogos eletrônicos é que esta área possui um apelo pedagógico muito forte. Como visto em [GUZDIAL e SOLOWAY, 2002] e [BECKER, 2001], foi observado que atualmente nas universidades o interesse por parte dos alunos de Ciência da Computação aumenta consideravelmente quando ocorrem disciplinas associadas ao desenvolvimento de jogos eletrônicos e conteúdos multimídia em geral. Isto acontece devido à união de ensino com entretenimento, e com isto o aprendizado se torna efetivamente mais eficiente.

## *1.1 Justificativa*

A Virtuosi [CALSAVARA, 2000] é uma arquitetura de execução distribuída baseada em máquinas virtuais, orientação a objeto e reflexão computacional. Proporciona um modelo de desenvolvimento e execução totalmente orientado a objeto, possibilitando simplicidade no desenvolvimento. A execução nesta arquitetura ocorre através de máquinas virtuais, o que acarreta em portabilidade para as aplicações, e o código executado é baseado em um metamodelo comum, o que possibilita a utilização de múltiplas linguagens no desenvolvimento. A Virtuosi também provê distribuição transparente em sua arquitetura, e um de seus principais objetivos é a utilização no ensino.

Atualmente, a Virtuosi não suporta a execução de múltiplas tarefas, ou seja, não suporta assincronismo, e consequentemente não possui um mecanismo de sincronização de atividades.

Esta dissertação tem como objetivo geral o desenvolvimento do modelo de concorrência para a arquitetura Virtuosi. Para verificar a usabilidade do modelo de concorrência desenvolvido será utilizado o contexto da área de desenvolvimento de jogos eletrônicos. Esta abordagem é justificada devido ao alinhamento dos objetivos e áreas de atuação do projeto Virtuosi com as necessidades ou requisitos de concorrência na área jogos eletrônicos.

Pode-se considerar que o desenvolvimento de jogos eletrônicos está inserido em um nicho de aplicação que possui os problemas dos quais a Virtuosi se propõe a resolver, e com a aplicação da Virtuosi no desenvolvimento de jogos, a mesma ganha um apelo ainda maior em sua perspectiva pedagógica.

## 1.2 Objetivos específicos desta dissertação

Esta dissertação tem como objetivo a construção de um modelo de concorrência para a Virtuosi e que o mesmo seja demonstrado no contexto das necessidades de concorrência da área de jogos eletrônicos, tanto na parte de infra-estrutura (*engines*), quanto na parte de lógica de jogo. Para isto, podem-se delinear os seguintes objetivos:

- Projetar um modelo de concorrência na Virtuosi que ofereça o suporte a assincronismo considerando as necessidades da área de jogos eletrônicos. Tal modelo deve ser voltado para execução local, ou seja, execução em uma única máquina virtual Virtuosi.
- Projetar um mecanismo de sincronização local também voltado para as necessidades da área de jogos. O mecanismo desenvolvido deve possibilitar exclusão mútua e sincronização condicional.
- Definir como será o suporte na arquitetura Virtuosi para o modelo de concorrência desenvolvido.
- Definir como será o suporte ao modelo de concorrência desenvolvido nos níveis de linguagem de programação e API.
- Validar o modelo desenvolvido através de exemplificações de uso em um jogo eletrônico.

Não faz parte do escopo desta dissertação o desenvolvimento de um mecanismo de controle de transações nem o controle de concorrência para sistemas distribuídos.

Também não será implementada uma *engine* para jogos eletrônicos nem um jogo eletrônico em si, somente serão definidos recursos de concorrência necessários para sua implementação.

## 1.3 Estrutura da dissertação

O restante desta dissertação está dividido da seguinte forma:

- Capítulo 2 – Sistemas concorrentes. Apresenta um resumo do funcionamento dos principais sistemas concorrentes que serviram de base para a concepção deste trabalho de pesquisa e também uma revisão literária de conceitos da área de concorrência e sistemas distribuídos necessários neste trabalho.
- Capítulo 3 – Virtuosi. Apresenta as justificativas da utilização da Virtuosi no desenvolvimento de jogos eletrônicos.
- Capítulo 4 – Assincronismo. Apresenta como foi definido o uso de assincronismo neste trabalho, quais os mecanismos utilizados, qual o suporte oferecido na linguagem e biblioteca padrão de classes, e como o mesmo é integrado na arquitetura Virtuosi.
- Capítulo 5 – Controle de Concorrência. Descreve os mecanismos para controle de concorrência desenvolvidos e suas particularidades. Exemplifica a utilização dos mecanismos e demonstra como os mesmos foram definidos na arquitetura Virtuosi.
- Capítulo 6 – Discute o uso deste trabalho na área de desenvolvimento de jogos eletrônicos.
- Capítulo 7 – Conclusão. Apresenta os resultados obtidos com este trabalho, sua contribuição científica e também discute possíveis trabalhos futuros.
- Apêndice A – Comparação com Java. Compara implementações de funcionalidades desenvolvidas com respectivas versões implementadas em Java.
- Apêndice B – Apresenta a formalização das construções sintáticas inseridas na linguagem *Aram* para o suporte à concorrência oferecido na linguagem de programação.
- Anexo A – O metamodelo Virtuosi. Apresenta de forma detalhada o metamodelo Virtuosi.
- Anexo B – A linguagem *Aram*. Apresenta em detalhes a linguagem *Aram.*

# 2 Sistemas concorrentes

Este capítulo apresenta um resumo do funcionamento dos principais sistemas concorrentes que serviram de base para a concepção deste trabalho de pesquisa e também uma revisão literária de conceitos da área de concorrência e sistemas distribuídos necessários neste trabalho.

## *2.1 Concorrência*

Segundo Andrews [ANDREWS, 2000], "é denominado programa concorrente um programa que possui dois ou mais processos trabalhando juntos na execução de uma determinada tarefa. Cada processo é um programa seqüencial, ou seja, uma seqüência de instruções executadas uma após a outra". Um programa seqüencial possui uma única linha de execução e um programa concorrente possui múltiplas linhas de execução.

Podemos classificar um sistema concorrente quanto à localização dos processos que compõem o sistema: Em sistemas *multithreaded* temos múltiplas linhas de execução ou *threads* na mesma máquina, até mesmo múltiplas linhas de execução no mesmo processo do sistema operacional. Em sistemas *distribuídos* temos múltiplas linhas de execução em máquinas distintas. Em sistemas paralelos temos ambas as ocorrências.

A única forma de *threads*, ou processos, resolverem em conjunto uma tarefa é ocorrendo comunicação entre os mesmos. A forma de comunicação para a execução de uma tarefa também depende da localização. Em sistemas distribuídos temos comunicação através de passagem de mensagens. Em sistemas *multithreaded* a comunicação também pode ocorrer utilizando passagem de mensagens, mas a forma mais utilizada é comunicação através de memória compartilhada.

Devido à comunicação, independente da forma, os vários processos devem sincronizar entre si para resolver adequadamente o problema. Existem duas formas de sincronização: *exclusão mútua*, que significa que dois processos nunca executam uma *região crítica* [DIJKSTRA, 1965] ao mesmo tempo; E *sincronização condicional*, que é o ato de processos aguardarem até que uma condição se torne verdadeira.

## 2.2 Sistemas multithreaded

Como descrito anteriormente, em um sistema *multithreaded* temos diversos processos ou *threads* com a finalidade de resolver um problema computacional sendo executados na mesma máquina. O número de *threads* geralmente é maior que o número de processadores na máquina e a execução de cada *thread* ocorre através da divisão do tempo de uso/processamento de cada processador.

A principal forma de comunicação em sistemas *multithreaded* acontece através de memória compartilhada, mais especificamente através de variáveis compartilhadas. Os principais mecanismos para sincronização de *threads* utilizando variáveis compartilhadas são os seguintes:

- Travas e Barreiras: Uma trava é um mecanismo para obter exclusão mútua. Serve para barrar acessos a regiões críticas. Uma barreira é um ponto de sincronização que todos os processos envolvidos em uma computação devem atingir antes de continuar sua execução.
- Semáforos: O primeiro mecanismo de sincronização entre processos desenvolvido foi o semáforo [DIJKSTRA, 1968], e este mecanismo ainda é um dos mais importantes mecanismos de sincronização utilizados atualmente. Um semáforo pode ser implementado utilizando uma técnica de *espera ocupada*[6] ou utilizando suporte nativo do *kernel* do sistema operacional. Um semáforo é um mecanismo que possibilita duas operações atômicas (indivisíveis) que são P (*wait* ou *down*) e V (*signal* ou *up*). Quando um processo invoca a operação P, o semáforo verifica seu contador, se for um valor positivo ele realiza o decremento do contador e libera a execução do processo. Caso o valor do contador seja zero ou negativo, o processo fica bloqueado. Quando um processo invoca a operação V, o semáforo incrementa o contador, caso o contador atinja um valor positivo, ele libera algum processo que esteja aguardando a execução. Semáforos possibilitam a implementação de exclusão mútua de forma fácil, e podem ser utilizados para resolver quaisquer problemas de sincronização.

[6] Do inglês, *busy-waiting*

- Monitores: Monitores são mecanismos de sincronização mais avançados que semáforos. Um monitor encapsula a representação de um tipo de dado abstrato e possibilita um conjunto de operações para sincronização. Assim como o semáforo, um monitor pode ser utilizado para resolver exclusão mútua e também sincronização condicional. Um monitor normalmente resolve exclusão mútua implicitamente, mas a resolução explícita também pode ocorrer, neste caso um monitor pode utilizar o conceito de *lock* e *unlock* em recursos compartilhados para garantir a exclusão mútua. Já a resolução de sincronização condicional é melhor que ocorra explicitamente, pois diferentes programas requerem diferentes condições de sincronização. A forma utilizada por monitores para possibilitar sincronização condicional é a disponibilização de *variáveis condicionais*, que são utilizadas para atrasar a execução de um processo até que uma condição booleana torne-se verdadeira. Um monitor pode disponibilizar uma série de operações, que devem ser mutuamente exclusivas, e pode também oferecer métodos para suspensão e sinalização de condições de sincronização, como por exemplo, um método chamado *await* e outro método chamado *signal*, que servem para suspender a execução e sinalizar retomada da execução respectivamente. Quando um processo executa o *signal*, o mesmo é executado em um monitor, e o processo possui o controle da trava implícita do monitor. Isto leva a um dilema, pois se o *signal* acorda outro processo, os dois processos (o processo que executou o *signal* e o processo que foi acordado) podem executar neste instante, mas por conta da exclusão mútua, somente um pode possuir a trava associada ao monitor. Com isso, duas formas possíveis de sinalização podem ocorrer, são elas: *Signal and Continue (SC)* e *Signal and Wait (SW)*. Na primeira forma o processo que efetuou o *signal* continua sua execução, e o processo que foi acordado aguarda até a disponibilização da trava do monitor para continuar sua execução. Esta forma de sinalização pode ser considerada não preemptiva. Na segunda forma (SW) o processo que efetuou o *signal* interrompe sua execução para retomada em um momento posterior, disponibilizando assim a trava do monitor para o processo que foi acordado. Esta forma de sinalização pode ser considerada preemptiva. A figura 2.1 apresenta um diagrama de estados demonstrando

sincronização com monitores. Exemplos de utilização de monitores serão apresentados posteriormente neste trabalho.

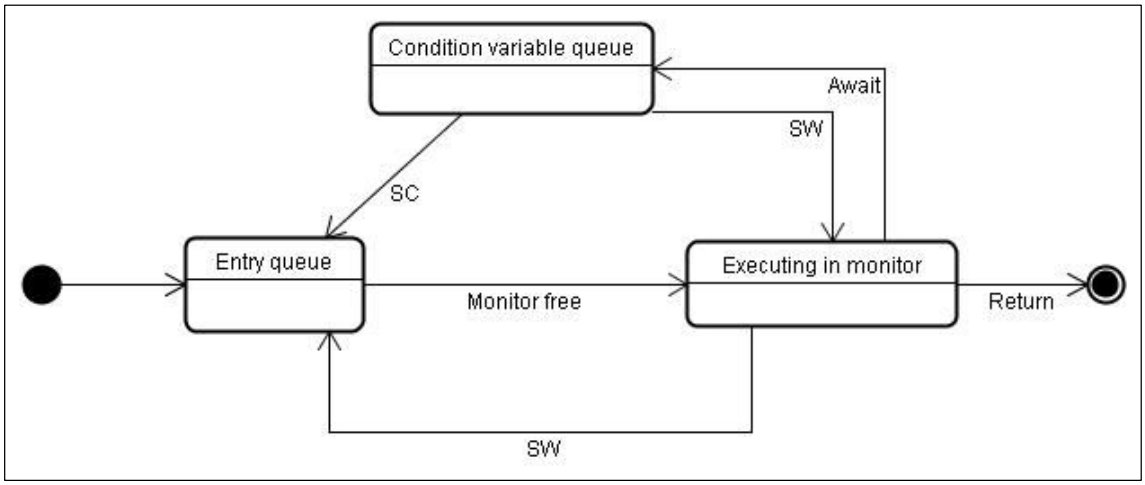


**Figura 2. 1 - Diagrama de estados - Sincronização com monitores**

## 2.3 Sistemas distribuídos

Ao contrário dos sistemas *multithreaded*, em sistemas distribuídos ocorrem processos sendo executados em máquinas distintas. Com isto, não há a possibilidade de comunicação utilizando-se de memória física compartilhada. Como mencionado anteriormente, a principal forma de comunicação em sistemas distribuídos acontece através de passagem de mensagens entre os processos. Neste caso, a sincronização é realizada através da comunicação entre os processos.

Dentre os mecanismos de passagem de mensagem, podemos destacar o mecanismo denominado *Remote Procedure Call* (*RPC*), que é a invocação de procedimentos ou funções de processos sendo realizadas por outros processos, localizados numa máquina distinta do processo destino. E também, o mecanismo denominado *Rendezvous*, que é uma forma de sincronização na qual um processo invoca uma operação em um segundo processo e aguarda uma invocação retorno (*callback*) por parte deste segundo processo. Em sistemas distribuídos, a comunicação entre os processos se dá através de uma interface de rede.

## 2.4 A linguagem SR

A primeira versão da linguagem SR[7] foi desenvolvida em 1981 por Andrews e já oferecia suporte a alguns dos conceitos de concorrência apresentados nesta dissertação, tais como invocações assíncronas e *rendezvous* [ANDREWS, 1981] e [ANDREWS et al., 1988]. Conforme visto em [OLSSON et al., 1992], a linguagem evoluiu para oferecer suporte a diversos mecanismos de sincronização incluindo os descritos na seção 2.2.

O modelo de execução de um programa SR é baseado em máquinas virtuais. Cada máquina virtual pode executar uma ou mais linhas de execução, denominadas processos. A execução em SR é baseada em operações (*op*), que podem ser invocadas de forma síncrona (*call*) ou assíncrona (*send*). Operações podem ser providas de duas formas, através de um processo (*proc*) ou uma *indicação de entrada*[8] (*in*). A tabela 2.1 demonstra as possibilidades de operações em SR.

| Invoke | Service | Effect |
|---|---|---|
| `call` | `proc` | (possibly remote) procedure call |
| `call` | `in` | rendezvous |
| `send` | `proc` | dynamic process creation |
| `send` | `in` | asynchronous message passing |

**Tabela 2. 1 - Possibilidades de operações em SR. [OLSSON et al., 1992].**

SR ainda provê diversos mecanismos que são abreviações para usos comuns de operações. Por exemplo, a declaração *process* é uma abreviação para uma *op* provida através de um *proc*. Uma instância do processo é criada por um *send* implícito.

Duas abreviações bastante utilizadas são os semáforos e a palavra reservada *receive*. A palavra *receive* é uma abreviação para um *in* que prove uma única operação que apenas armazena os argumentos em variáveis locais. Um semáforo (*sem*) é uma abreviação para duas operações sem argumentos que são *P* e *V*, onde *P* é um caso especial de um *receive* e *V* é um caso especial de um *send*.

Finalmente, SR disponibiliza ainda mais algumas construções úteis para programação concorrente, como a construção *reply* que é uma variação de um retorno na qual um valor é

[7] Do inglês, *Synchronizing Resources*
[8] Do inglês, *Input Statement*

retornado para o processo chamador, porém o processo chamado continua sua execução. Um outro exemplo é a construção denominada *co,* que serve para invocação de operações de forma paralela.

## 2.5 Concorrência em sistemas orientados a objeto

Conforme visto em [BRIOT, GUERRAOUI, LOHR, 1998], tipicamente o desenvolvimento de sistemas concorrentes orientados a objeto ocorre utilizando três abordagens específicas, que podem ser utilizadas complementarmente. São elas: abordagem utilizando bibliotecas, abordagem integrativa, e finalmente abordagem reflexiva.

A primeira abordagem, utilizando bibliotecas, consiste em os elementos de programação concorrente serem inseridos através de bibliotecas de classes que representam estes conceitos. A idéia básica é aplicar elementos de orientação a objetos aos elementos de programação concorrente, possibilitando alto nível de abstração e encapsulamento, e possibilitando também herança, como uma forma de estruturação do projeto do sistema concorrente. Nesta abordagem, a definição de classes para construção de elementos de sincronização pode ser simples, como por exemplo, uma classe Semáforo. Porém, a abstração de uma *thread* no formato de classe pode não ser tão óbvia. Ao menos três modos de utilização de uma classe *Thread* são possíveis: 1 – Uma *thread* é uma instância de alguma subclasse de alguma classe *Thread*, e sua atividade é descrita por um "construtor" desta subclasse. 2 – Uma *thread* é uma instância de alguma subclasse de alguma classe *Thread* e sua atividade é descrita através da redefinição de algum método em especial na subclasse. 3 – Uma *thread* é uma instância de alguma classe *Thread* e sua atividade é descrita através de uma função passada como parâmetro do construtor ou de algum método em especial. O principal suporte que a abordagem utilizando bibliotecas oferece é a possibilidade de projetar e implementar programação concorrente utilizando abstrações adequadas de baixo nível e possibilitando que abstrações de mais alto nível sejam desenvolvidas utilizando estas abstrações de mais baixo nível. Desta forma, a utilização desta abordagem permite um ganho de flexibilidade e reduz a complexidade para os desenvolvedores de programas concorrentes, pois as questões relativas à programação concorrente estão estruturadas em bibliotecas de classes.

A segunda abordagem, utilizando integração de conceitos, ao invés de apresentar os elementos de programação concorrente de forma ortogonal, tem como objetivo realizar a junção ou integração destes elementos com a linguagem/modelo de programação, oferecendo ao programador um modelo de objetos concorrentes unificado. A abordagem utilizando bibliotecas ajuda a estruturar conceitos como *threads*, semáforos, e demais conceitos, mas mantém estes conceitos disjuntos. A abordagem utilizando integração visa à integração destes conceitos, uniformizando o modelo de programação e conseqüentemente facilitando o trabalho dos programadores de aplicações.

A integração de conceitos de orientação a objeto com conceitos de programação concorrente pode acontecer em três níveis, a saber: 1 – A integração do conceito de objeto e do conceito de processo, *thread* ou atividade leva ao conceito de um *objeto ativo*. 2 – Em um segundo nível tem-se a integração do conceito de sincronização com um *objeto ativo*, possibilitando um conceito de *objeto sincronizado*. 3 – Um terceiro nível considera um objeto como a unidade de distribuição, levando à noção de *objeto distribuído*. Ou seja, objetos são vistos como entidades que podem ser distribuídas e replicadas entre vários processos, e a invocação de métodos destes objetos é uma forma transparente de invocação de atividades locais ou remotas.

Pode-se concluir que a abordagem utilizando integração de conceitos provê um modelo uniformizado de programação, de tal forma que elementos de programação orientada a objeto e programação concorrente não sejam disjuntos. É uma abordagem complementar a abordagem utilizando bibliotecas.

A terceira abordagem, utilizando *reflexão computacional*, tenta combinar a flexibilidade da abordagem utilizando bibliotecas com a transparência da abordagem utilizando integração de conceitos. O uso de reflexão computacional permite a construção de um *framework* geral para customizar a forma de concorrência, protocolos e aspectos da distribuição de objetos. Isto se dá através de metabibliotecas [MAES, 1987] [WATANABE e YONEZAWA, 1989] ligadas à linguagem ou bibliotecas em questão.

Este trabalho utilizará as abordagens de biblioteca e integrativa, abrangendo o primeiro e segundo nível, e possibilitando uma base para um futuro desenvolvimento do terceiro nível.

Diversas linguagens de programação e correspondentes arquiteturas de execução utilizam as abordagens mencionadas. Considerando o escopo deste trabalho, sistemas

orientados a objeto e máquinas virtuais, destacam-se algumas arquiteturas de execução, tais como Java, .NET e Python. Além destas arquiteturas, pode-se também destacar os modelos de CORBA e Ice, por apresentarem um modelo de concorrência apropriado para sistemas distribuídos.

## 2.6 Java

Uma *máquina virtual Java* (*JVM*[9]) [GOSLIN et al., 2005] pode suportar várias *threads* executando ao mesmo tempo. Estas *threads* executam códigos que operam em valores e objetos que estão em uma memória compartilhada. Em Java, *threads* podem ser suportadas em múltiplos processadores, dividindo processamento em um único processador ou dividindo processamento de vários processadores.

Em Java, a forma de sincronização nativa é realizada através de monitores (uma versão específica do conceito de monitores apresentado em [ANDREWS, 2000]), para isto a palavra reservada *synchronized* foi inserida na linguagem. Esta palavra serve para definir uma região crítica. O comportamento de um monitor em Java pode ser explicado em termos de travas associadas aos objetos e regiões críticas. Cada objeto em Java pode ter um monitor associado.

Com a utilização da palavra reservada *synchronized*, a *thread* em execução computa a referência para um objeto destino na sincronização. Então tenta obter uma trava para o monitor do objeto, e não prossegue até que consiga a trava. Ao conseguir a trava, a *thread* executa o bloco de código sincronizado e, ao final da execução, a trava do monitor é liberada. Por conveniência, um método inteiro pode ser sincronizado em Java. Caso seja um método de instância, a trava acontece no monitor da instância em questão. Caso seja um método de classe (*static*), a trava acontece no monitor do objeto que representa a classe em questão. Ao fim da execução do método, a trava é liberada.

Durante a execução de uma *thread*, uma série de ações é executada seqüencialmente. Uma *thread* pode utilizar o valor de uma variável ou atribuir um novo valor a uma variável. Caso duas *threads* utilizem a mesma variável, obtém-se o que é denominado *timing-dependent results*, ou seja, resultados dependentes da ordem de execução das *threads*. Para amenizar o problema de concorrência, mecanismos de sincronização podem ser utilizados. No

[9] Do inglês, *Java Virtual Machine*

caso de utilização de sincronização, a *thread* realiza uma trava no monitor de um determinado objeto antes do seu uso e atualiza a memória principal após o uso, e finalmente libera a trava do monitor do objeto.

Alguns métodos definidos na classe *java.lang.Object* são disponibilizados para transferência de controle de execução pelas *threads*. Os métodos são:

- *wait*(): Suspende a *thread* corrente que adquiriu a trava até que outra *thread* notifique que a *thread* corrente ou qualquer outra *thread* suspensa deva retomar sua execução.
- *notify*(): Notifica alguma *thread* que está suspensa através do *wait* para que ela retome sua execução.
- *notifyAll*(): Notifica todas as *threads* que estão suspensas através do *wait* para que retomem sua execução. Porém, neste caso, somente uma *thread* por vez conseguirá adquirir a trava novamente.

Em Java, uma classe da biblioteca denominada *java.lang.Thread* encapsula uma *thread*. Esta classe oferece uma série de métodos, como por exemplo, um método para iniciar a execução de uma *thread*, denominado *start*, e também um método que define o comportamento da *thread*, denominado *run*. A classe *Thread* define também uma série de atributos, como por exemplo, a *flag daemon*, que significa que se não houver outras *threads* não *daemons* em execução, a execução da máquina virtual pode ser interrompida.

Recentemente, na versão 1.5 de Java, novas abstrações para o suporte a concorrência foram introduzidas, dentre elas podemos destacar o suporte a travas, um novo modelo de uso de condições de sincronização, na qual a condição é ligada a uma trava, e também operações atômicas.

### 2.6.1 Modelo de memória

Uma variável representa uma área de memória onde um componente de programa pode ser armazenado (variáveis de classe e instância, e também *arrays*). Variáveis são mantidas na *memória principal* (*main memory)*, que é compartilhada entre todas as *threads*.

A memória principal mantém todas as variáveis e também todos os *locks* destas variáveis. Cada *thread* possui uma cópia local da memória principal, denominada *working memory*, na qual armazena as variáveis que lê e escreve. Ou seja, cada *thread* mantém na *working memory* as variáveis que consultar, criar, ou alterar. Cada *thread* possui também seu próprio contador de programa *program counter* e *frame stack*.

A figura 2.2 apresenta as iterações entre *thread*, *working memory* e *main memory*.

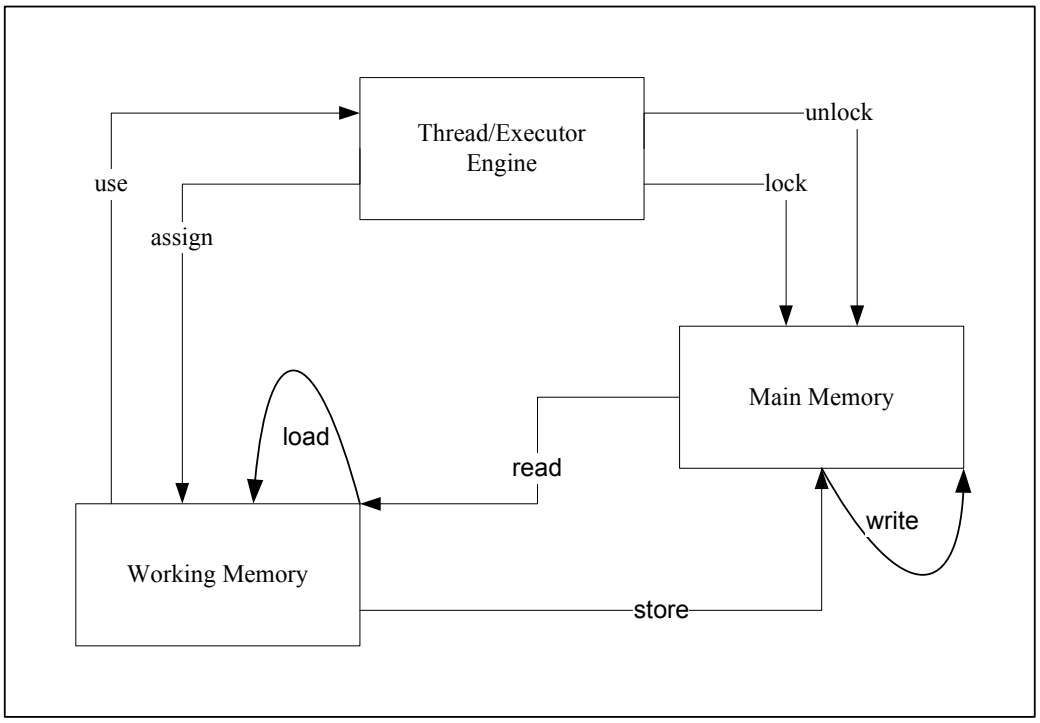


**Figura 2. 2 – Modelo de memória da JVM**

Os seguintes verbos (em inglês) definirão possíveis ações que uma *thread* pode realizar ou causar a execução (atividades atômicas, ou seja, indivisíveis):

- *use*: Uma *thread* pode consultar o valor de uma variável. Uma *thread* somente consulta valores em sua *working memory*.
- *assign*: Uma *thread* pode atribuir um valor a uma variável. Válido também somente para a *working memory* da *thread*.

- *lock* e *unlock*: Travar o acesso a uma variável e liberar o acesso respectivamente. Esta é uma operação que envolve a *thread*, o ambiente de execução e a área de memória principal. Quando uma trava é adquirida por uma *thread*, todas as variáveis da *working memory* são atualizadas. Antes de a trava ser liberada, todas as variáveis são atualizadas na memória principal. Somente uma única *thread* por vez pode possuir uma trava de um monitor.

- *read*: Este comando é executado para que os valores sejam copiados da memória principal para uma *working memory*.
- *load*: Para cada *read* executado na memória principal, um *load* é executado na *working memory* para carregar o valor lido.

- *store*: Este comando é executado pela *working memory* para que os valores sejam copiados da *working memory* para a memória principal.
- *write*: Para cada *store* executado pela *working memory* um *write* é executado na memória principal para armazenar o valor na memória principal.

Resumidamente temos: Um programa em uma *thread* pode executar um *use* ou *assign*, que encadearão a execução dos comandos *read* e *load*, *store* e *write*, respectivamente. O mesmo também pode realizar um *lock* ou *unlock*, que farão com que a *thread* adquira ou libere uma trava na memória principal.

Durante a execução de todos estes comandos pode ocorrer um tempo de espera que pode variar de transação para transação. A especificação da JVM permite certa liberdade para as implementações reordenarem os comandos. Estas reordenações podem acontecer em tempo de compilação ou em tempo de execução, sempre através de otimizações de código. Por exemplo, *reads* e *writes* podem ser invertidos. A figura 2.3 apresenta um exemplo de código Java utilizado para exemplificar concorrência, e a figura 2.4 apresenta um diagrama ilustrando as instruções no modelo de memória compartilhado da JVM com a execução concorrente do código da figura 2.3, com a utilização de duas *threads*, e que pode levar a problemas de sincronização.

```
class ExemploConcorrencia {
    int a=1, b=2;
    void x() {
        a=b;
    }
    void y() {
        b=a;
    }
}
```

**Figura 2. 3 - Exemplo de concorrência - Código Java**

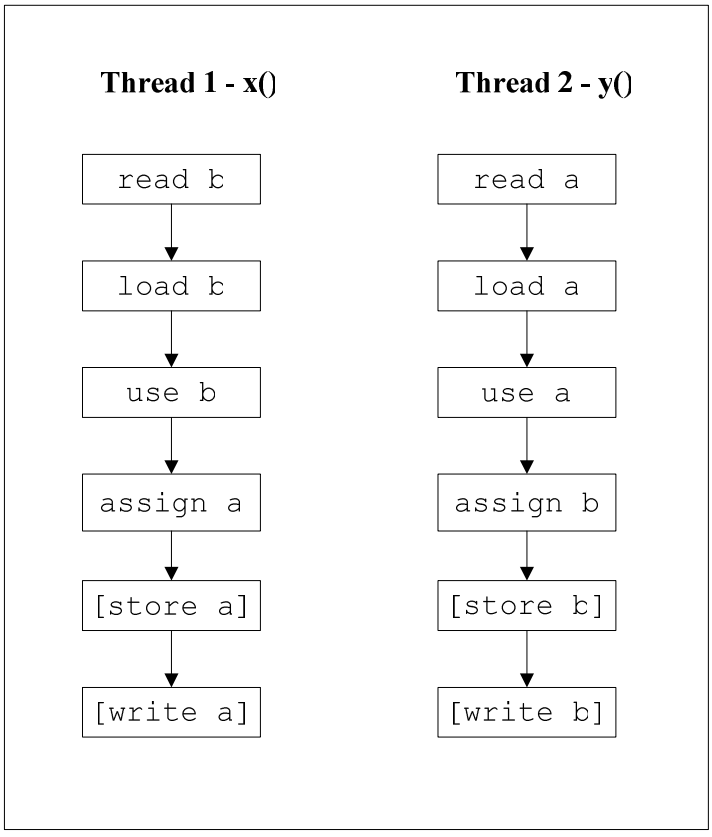


**Figura 2. 4 - Exemplo de concorrência - Instruções JVM**

A JVM possui dois modos de implementação de *Threads*. O primeiro é denominado *green threads*, que é a implementação de *threads* sem o suporte do sistema operacional. Neste modo, a própria JVM é responsável pelo escalonamento das *threads*. O segundo modo é denominado *native threads*, nesta modalidade, a JVM utiliza suporte do sistema operacional e delega o escalonamento ao mesmo.

## 2.7 JR

A linguagem JR [KEEN et al., 2004] é uma extensão da linguagem Java, baseada na linguagem SR [ANDREWS, 1981]. Programas JR são escritos em uma versão modificada de Java e depois traduzidos para Java padrão.

JR provê a criação dinâmica de máquinas virtuais remotas, criação dinâmica de objetos remotos, invocação remota de métodos, comunicação assíncrona, *rendezvous* e criação dinâmica de processos. Em JR, os métodos de Java são utilizados como os *procs* de SR. Todas as operações são baseadas em uma classe básica denominada *Op* (seguindo a nomenclatura de SR). Esta classe *Op* provê métodos para invocação síncrona e assíncrona (*call* e *send* em SR), além de prover serviço do tipo *in*, possibilitando *rendezvous*. As duas formas possíveis de invocação, via método ou *in*, deram origem a duas subclasses da classe *Op*, denominadas *ProcOp* e *InOp*, respectivamente. A declaração de uma *InOp* define implicitamente uma implementação que consiste em uma fila de invocação. As invocações para uma *InOp* aguardam em uma fila de invocação até que a *InOp* possa tratar a invocação.

JR define também um sofisticado modelo de herança para as funcionalidades relativas a controle de concorrência, porém, não será abordado neste trabalho devido ao fato de que herança não faz parte do escopo do mesmo.

Pode-se concluir que a principal contribuição da linguagem JR é prover grande parte das funcionalidades relacionadas à concorrência encontradas em SR (as principais construções não providas são as construções *co* e *reply*) de uma maneira orientada a objeto.

## 2.8 .NET

Outro ambiente de execução concorrente baseado em máquinas virtuais é o *.NET* [STÄRK e BÖRGER, 2004], que em muitos aspectos é similar ao Java, inclusive na sua linguagem principal, a linguagem C# [MICROSOFT, 2001].

No *.NET framework*, as funcionalidades referentes a *thread* e controle de concorrência podem ser encontradas no *namespace System.Threading*. Neste *namespace* encontra-se a classe *Thread* que representa uma *thread* em .NET, e também a classe Monitor, que representa a abstração de um monitor, já vista anteriormente. Neste *namespace* encontra-se também uma série de abstrações de alto nível para facilitar a sincronização de *threads*, como por exemplo, o método *Thread.MemoryBarrier()* (em Java, algumas abstrações foram implementadas recentemente e estão disponíveis a partir da versão 1.5). Encontra-se também neste *namespace* a definição do tipo *ThreadStart*, que denota uma função com zero argumentos e retorno *void*, que deve ser passado no construtor de uma *Thread*. Esta função deve implementar o comportamento da *thread* em execução. O número de estados de uma *thread* em .NET é maior que em Java e as variações de estados de uma *thread* são bem mais complexas também. A figura 2.5 apresenta os estados de uma *thread* em .NET e também alguns eventos que levam a mudança de estados.

Algumas *flags* são utilizadas no estado de uma *thread*, uma dessas *flags* indica se a *thread* se encontra no estado *Background*, que significa que se não houver outras *threads* em estados normais, a execução pode ser interrompida. Este conceito é similar a flag *daemon* encontrada em Java.

Uma outra diferença entre *threads* em .NET e Java é que a classe *Thread* em .NET é uma classe que não pode ser especializada por outra (utiliza-se o termo *sealed* para isto).

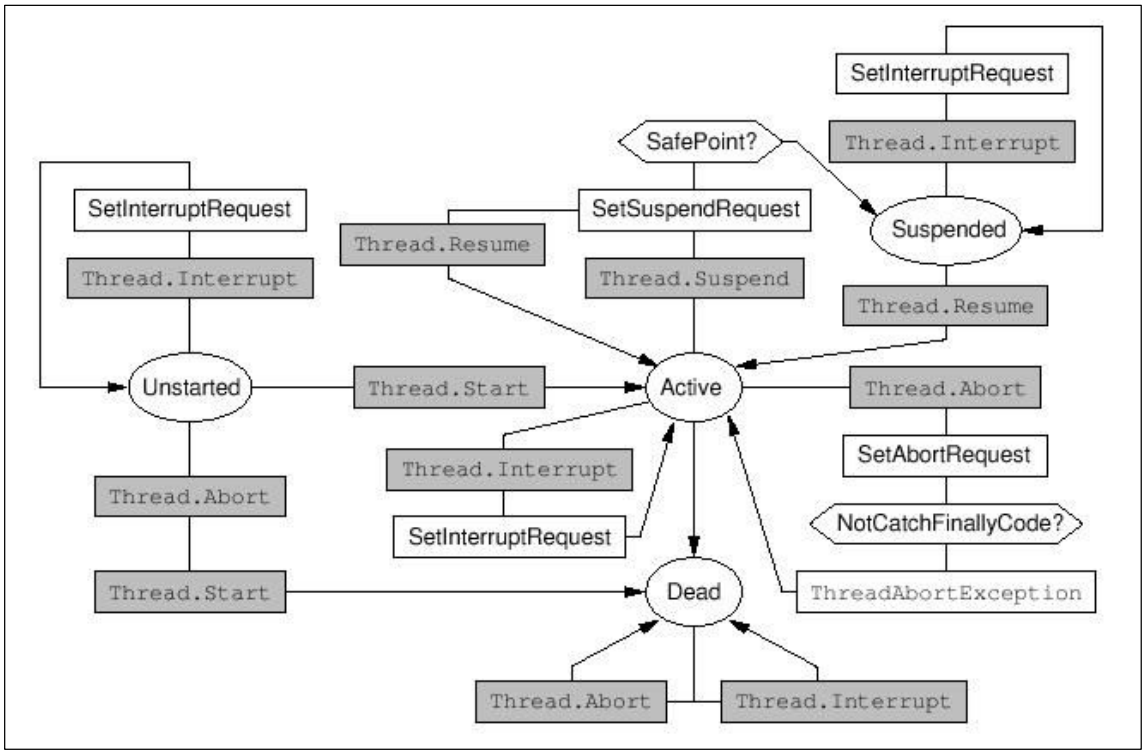


**Figura 2. 5 - Estados e eventos de uma *thread* em .NET**

A palavra reservada para sincronização utilizando monitores em .NET é o *lock*. É similar ao *synchronized* encontrado em Java. Um *lock* pode ser reduzido à invocação de duas funções, que são *Monitor.Enter* e *Monitor.Exit*. Ao contrário de Java, estes elementos podem ser invocados separadamente, o que permite o desenvolvimento de código mal formado por parte do programador (ex: *Monitor.Enter* sem o respectivo *Monitor.Exit*). A figura 2.6 apresenta a redução da palavra reservada *lock* para chamadas da API.

```
lock (exp) stm  ⟹  { object o = exp;
                     Monitor.Enter(o);
                     try { stm }
                     finally { Monitor.Exit(o); }
```

**Figura 2. 6 - Redução de um lock para chamadas da API**

O construtor da classe *Monitor* é privado, o que significa que classes de aplicação não podem instanciar diretamente um monitor. Assim como em Java, monitores são associados automaticamente aos objetos. Um monitor possui os métodos *Wait*, *Pulse* e *PulseAll*, que são similares aos métodos *wait*, *notify* e *notifyAll* encontrados na classe *Object* de Java.

Os métodos da classe Monitor são *static*, e são utilizados para adquirir e liberar travas em monitores. Por exemplo, o método *Monitor.Enter* é utilizado para adquirir uma trava. Se a *thread* corrente já possuir a trava, um contador de travas é incrementado. Se a trava estiver livre e a fila de leitura do monitor (*readyQueue*) estiver vazia, a *thread* obtém a trava. Caso contrário, a *thread* tem seu estado alterado para *Syncing* e a *thread* é adicionada na *readyQueue* do monitor. O método *Monitor.Wait* insere a *thread* corrente na fila espera (*waitQueue*) do monitor, e temporariamente libera a trava do monitor. O método *Monitor.Pulse* move o primeiro elemento da *waitQueue* do monitor para a *readyQueue*. E finalmente o método *Monitor.PulseAll* move todos os elementos da *waitQueue* para a *readyQueue*.

A utilização destas filas de espera (*waitQueue*) e obtenção de acesso (*readyQueue*) em .NET possibilita que a sincronização condicional ocorra de maneira determinista, o que não ocorre em Java, pois as listas de espera e de leitura do monitor não seguem o padrão de uma fila (FIFO[10]), elas utilizam uma ordem qualquer (arbitrária).

### 2.8.1 Modelo de memória

Algumas diferenças ocorrem no modelo de memória entre o .NET e o Java. No .NET o modelo de memória é bem mais restritivo e algumas otimizações possíveis em Java não são permitidas. A seguir uma lista de restrições que regem o modelo de memória do .NET e são permitidas em Java:

- *reads* e *writes* oriundos da mesma *thread* não podem ser reordenados;
- Um *read* não pode ser movido para antes de um *lock*;
- Um *write* não pode ser movido para depois de um *lock release*;

[10] Do inglês, *First In, First Out*

## 2.9 Python

A linguagem Python [MATLOFF and HSU, 2005], assim com Java ou .NET, também tem sua execução baseada em uma máquina virtual. O gerenciamento de *threads* em Python segue o modelo preemptivo, porém não há uma delegação para o sistema operacional, e a própria máquina virtual controla a execução. O controle se dá através da contagem do número de instruções (*byte code*) executadas, e quando uma *thread* atinge certo número de instruções a mesma é interrompida, cedendo a execução para uma *thread* que aguardava a execução. O número padrão de instruções para interrupção é 10, mas este número pode ser alterado através do método *sys.setcheckinterval()*.

Em Python, as *threads* estão disponíveis em dois módulos, *thread.py* e *threading.py*. O primeiro oferece um acesso mais simples, o segundo mais avançado.

O módulo *thread.py* possibilita a criação de *threads* através da chamada *thread.start_new_thread( method, (args,))*. Esta forma de criação de thread já indica qual o método a ser invocado no início de execução da *thread* e quais os argumentos devem ser passados para execução. O principal mecanismo para controle de concorrência em Python é a trava. A forma de criação de uma trava no módulo *thread.py* se dá através da chamada *thread.allocate_lock()* que retorna uma variável do tipo *lock*, que responde aos métodos *acquire()* e *release()*. O conceito de uma *thread* neste módulo não segue o padrão orientado a objeto, mas sim estruturado.

O módulo *threading.py* define uma versão orientada a objeto para *threads*, que é a classe *Thread.* A classe *Thread*, assim como em Java, disponibiliza métodos como *run()*, *start()* e *join()*.

Python possibilita também o controle de concorrência através de variáveis condicionais, disponíveis através das classes *threading.Condition* e *threading.Event*. A primeira (*Condition*) precisa ser utilizada em conjunto com travas e oferece os métodos *wait(), notify()* e *notifyAll()*. A segunda (*Event*) encapsula a utilização das travas fornecendo um serviço de mais alto nível através dos métodos *wait()* (aguarda um evento), *set()* (notifica ocorrência do evento), e por último *clear()* (marca o evento como não ocorrido novamente, pois após a invocação do *set()*, todo *wait()* retorna imediatamente).

Python também oferece em sua biblioteca classes como *threadin.Semaphore* e *threading.Timer*.

## 2.10 Serviço de concorrência em CORBA

Além destes três importantes exemplos de arquiteturas concorrentes (Java, .NET e Python) outro importante exemplo é o modelo do serviço de concorrência definido para CORBA [CORBA, 2000]. O objetivo do serviço de concorrência em CORBA é mediar/coordenar o acesso concorrente a um objeto de forma que o objeto consiga ser acessado concorrentemente de forma consistente.

Este serviço dispõe também de um mecanismo de travas interessante, que pode ser usado juntamente com um serviço de transação (a trava é liberada no *commit* ou *rollback* da transação). Cada trava é associada com um recurso (um objeto) e um cliente.

Existem também algumas formas de definição de travas, que correspondem a diferentes categorias de acesso, são elas: *trava para leitura* (*R*), *trava para escrita* (*W*), e *trava para atualização* (*U*). O serviço de concorrência de CORBA possibilita certa granularidade na utilização das travas, oferecendo mais duas categorias de travas que são: *intenção de leitura* (*IR*) e *intenção de escrita* (*IW*). Estas se aplicam para objetos complexos, isto é, containeres de outros objetos para os quais se podem adquirir travas de leitura e escrita.

A tabela 2.1 apresenta as possíveis travas no serviço de concorrência de CORBA, suas inter-relações e os conflitos entre as travas. Os asteriscos (*) indicam o conflito entre duas travas.

O serviço de concorrência possibilita também a definição de um conjunto de travas e define as diversas interfaces utilizadas para o controle de concorrência, como por exemplo, a interface *CosConcurrencyControl*, *LockCoordinator*, além de diversas outras.

Lock Compatibility

| Granted Mode | Requested Mode | | | | |
|---|---|---|---|---|---|
| | *IR* | R | U | IW | W |
| Intention Read (IR) | | | | | * |
| Read (R) | | | | * | * |
| Upgrade (U) | | | * | * | * |
| Intention Write (IW) | | * | * | | * |
| Write (W) | * | * | * | * | * |

**Tabela 2. 2 - Compatibilidade entre travas no serviço de concorrência de CORBA. [CORBA, 2000].**

## 2.11 Ice

Ice (*Internet Communications Engine*) [HENNING, 2004] e [HENNING e SPRUIELL, 2005] surgiu como uma alternativa ao CORBA, e seu objetivo, segundo seus autores é: "ser um *middleware* tão poderoso quanto CORBA, porém sem cometer os mesmos erros de design cometidos em CORBA".

Seguindo esta linha, Ice também oferece suporte à concorrência, porém com algumas adições ao suporte oferecido por CORBA.

O controle de concorrência também suporta sincronização condicional e exclusão mútua. Sincronização condicional é obtida através de monitores, seguindo uma linha similar à encontrada em Java (utilizando-se dos métodos *wait(), notify() e notifyAll()* ). Exclusão mútua é obtida através de semáforos, cuja API pode ser vista no código apresentado na figura 2.7.

```
namespace IceUtil {
    class Mutex {
        public:
            Mutex();
            ~Mutex();
            void lock() const;
            bool tryLock() const;
            void unlock() const;

            typedef LockT<Mutex> Lock;
            typedef TryLockT<Mutex> TryLock;
    };

    struct StaticMutex {
        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<StaticMutex> Lock;
        typedef TryLockT<StaticMutex> TryLock;
    };
}
```

**Figura 2. 7 - Exclusão mútua em Ice**

Uma novidade em relação ao CORBA no suporte à concorrência oferecido por Ice são travas temporais, apresentadas na figura 2.8. Com travas temporais é possível aguardar um tempo máximo para obtenção de uma trava, seja ela de leitura, escrita ou atualização.

```
namespace IceUtil {
    class RWRecMutex {
    public:
        void readLock() const;
        bool tryReadLock() const;
        bool timedReadLock(const Time &) const;

        void writeLock() const;
        bool tryWriteLock() const;
        bool timedWriteLock(const Time &) const;

        void unlock() const;

        void upgrade() const;
        bool timedUpgrade(const Time &) const;

        typedef RLockT<RWRecMutex> RLock;
        typedef TryRLockT<RWRecMutex> TryRLock;
        typedef WLockT<RWRecMutex> WLock;
        typedef TryWLockT<RWRecMutex> TryWLock;
    };
}
```

**Figura 2. 8 - Travas temporais em Ice**

Uma outra novidade em relação à CORBA é que além de invocação assíncrona de mensagens (AMI[11]), Ice também oferece o despacho assíncrono de mensagens (AMD[12]). AMD é o equivalente ao AMI, porém somente no lado servidor, ou seja, uma mensagem chega ao servidor de forma não bloqueante liberando a *thread* de despacho para o Ice. Ao final do processamento, o servidor invoca uma função de *callback*. Somente então o resultado é enviado ao cliente. A execução através de AMD e AMI é transparente para a aplicação servidora e aplicação cliente.

## 2.12 Lua

Lua [IERUSALIMSCHY, FIGUEIREDO, CELES, 2005] é uma linguagem de script extensível, que oferece suporte à programação procedural e facilidades para descrição de dados. Sua sintaxe é semelhante à de Pascal. Foi projetada para ser acoplada (Lua é uma biblioteca implementada em C) a programas maiores que precisem ler e executar programas

[11] Do inglês, *Asynchronous Message Invocation.*
[12] Do inglês, *Asynchronous Message Dispatch.*

escritos pelos usuários. A linguagem Lua é bastante utilizada como linguagem de script no desenvolvimento de jogos eletrônicos.

Lua oferece suporte à concorrência através de suas co-rotinas, que são espécies de *threads* cooperativas. Somente três funções são disponibilizadas para o controle das co-rotinas, são elas: *coroutine.create()*, que serve para criar uma co-rotina. *coroutine.resume()*, que serve para iniciar a execução de uma co-rotina e finalmente *coroutine.yield()* que serve para suspensão da execução da co-rotina. Uma co-rotina somente interrompe sua execução quando a função *yield()* é invocada. A linguagem Lua não fornece mecanismos para controle de concorrência.

# 3 Virtuosi

A Virtuosi [Calsavara, 2000] é uma arquitetura de execução orientada a objeto baseada em máquinas virtuais distribuídas, voltada a ensino. Este capítulo apresenta as justificativas da utilização da Virtuosi no desenvolvimento de jogos eletrônicos.

Os exemplos de código utilizados para demonstrar as funcionalidades desenvolvidas neste trabalho serão implementados na linguagem *Aram* [BORGES, 2004]. *Aram* é uma linguagem com sintaxe Java *like*, criada para representar as possibilidades do metamodelo Virtuosi. O metamodelo da Virtuosi define classes e associações que formalizam os elementos encontrados em uma linguagem de programação orientada a objeto que ofereça suporte aos conceitos e elementos da arquitetura Virtuosi. O Apêndice B apresenta a formalização das construções sintáticas da linguagem Aram, incluindo as construções para suporte à concorrência desenvolvidas neste trabalho.

Um documento detalhado do metamodelo Virtuosi é apresentado no Anexo A. O Anexo B apresenta em detalhes a linguagem *Aram*. Alguns trechos de código serão implementados na linguagem Java, mas somente para comparação com a implementação em *Aram*, e neste caso, uma indicação da utilização da linguagem Java será inserida na legenda do exemplo.

Os princípios que fazem parte da concepção da Virtuosi podem ser empregados de maneira que agreguem robustez, simplicidade, controle e flexibilidade ao desenvolvimento de jogos eletrônicos.

Dentre estes princípios, destacam-se:

## *3.1 Orientação a objeto*

O paradigma de orientação a objeto é provavelmente o paradigma de programação mais utilizado no meio industrial e acadêmico nestes últimos anos. Muitas das linguagens comumente utilizadas atualmente seguem este paradigma. Como exemplo, podemos evidenciar Java [GOSLING et al., 2005], C++ [STROUSTRUP, 1986], C# [MICROSOFT, 2001], Eiffel [MEYER, 1992] e Smalltalk [GOLDBERG e ROBSON, 1983]. O próprio ensino de programação muitas vezes ocorre através de uma dessas linguagens.

A ampla utilização deste paradigma tras benefícios tais como a ocorrência de muitos trabalhos e pesquisas que auxiliam no objetivo de facilitar e simplificar o desenvolvimento utilizando orientação a objeto. Destes trabalhos, devemos evidenciar as obras [MEYER, 1997], [GAMMA et al., 1995] e [FOWLER, 1999], que apresentam um apanhado completo sobre desenvolvimento de software orientado a objeto, soluções de problemas comuns e simplicidade no desenvolvimento.

Devido a grande gama de trabalhos e pesquisas nesta área, além da ampla disseminação deste paradigma entre os profissionais, pode-se afirmar que o uso de orientação a objeto no desenvolvimento de jogos eletrônicos pode simplificar e facilitar o desenvolvimento.

## 3.2 Portabilidade

A utilização de máquinas virtuais no desenvolvimento de software provê uma série de benefícios para as aplicações, dentre estes benefícios destacam-se a portabilidade e controle, pois os sistemas executam sobre uma arquitetura controlada que encapsula as peculiaridades de hardware e sistemas operacionais.

## 3.3 Suporte a múltiplas linguagens

A utilização de um metamodelo que define as operações possíveis em uma arquitetura possibilita a construção de diversos subconjuntos de linguagens que atuam neste metamodelo. Com isto, linguagens de alto e baixo nível podem ser desenvolvidas sobre uma mesma base, e podem ser utilizadas por diferentes tipos de profissionais.

Esta abordagem tras benefícios diversos como, por exemplo, a coesão no relacionamento entre peças de software desenvolvidas nestas diferentes linguagens (linguagens que atuam sobre o mesmo metamodelo), e facilidade de disseminação, pois as linguagens desenvolvidas nada mais são do que visões com diferentes níveis de complexidade do mesmo metamodelo.

## 3.4 Sistemas distribuídos

Um dos objetivos da Virtuosi é distribuição transparente, onde um objeto de uma máquina virtual pode invocar um método de outro objeto em outra máquina virtual de forma transparente. Esta possibilidade de distribuição transparente pode facilitar significativamente o desenvolvimento de jogos eletrônicos para múltiplos usuários, como por exemplo, jogos do tipo MMOGs.

# 4 Assincronismo

Na arquitetura Virtuosi, pode-se definir como *atividade* a execução de um construtor de uma classe ou a execução de um método de um objeto. Conforme apresentado em [CALSAVARA, 2000], a Virtuosi possui três modos de invocação de atividades, que são: *atividade síncrona*, *atividade assíncrona sem notificação de término* e finalmente *atividade assíncrona com notificação de término*. A primeira, *atividade síncrona,* foi inserida no metamodelo da Virtuosi em [NUNES e CALSAVARA, 2001] e posteriormente implementada em [KOLB, 2004]. Porém atividades assíncronas não haviam sido definidas até o presente trabalho.

Este capítulo apresenta como foi definido o uso de assincronismo na Virtuosi, quais os mecanismos utilizados e qual o suporte oferecido na linguagem e biblioteca padrão de classes.

## *4.1 Threads na Virtuosi*

Uma *thread* na Virtuosi segue o modelo comum de *threads* que executam operações seguindo a seqüência de instruções de um método ou construtor, porém, não através de uma pilha de instruções e um *program counter*, como em Java e .NET. A implementação na Virtuosi segue o pattern *Visitor* [GAMMA et al., 1995], onde a *thread* percorre o grafo de *statements* de um método ou construtor para executá-los.

Múltiplas *threads* podem executar de forma concorrente na Virtuosi. A forma de controle de execução definida foi a forma não preemptiva, ou cooperativa [WIKIPEDIA, 1], onde cada *thread* executa até seu bloqueio, e somente então a próxima *thread* inicia sua execução. A escolha da próxima *thread* a executar é realizada considerando as prioridades das *threads* que aguardam para execução (cada *thread* possui uma prioridade associada). Desta forma, a próxima *thread* a executar será a *thread* pronta para execução de maior prioridade, ou, em caso de empate na prioridade, a próxima *thread* a executar será a que está aguardando por mais tempo para execução. O *scheduler* é implementado com filas baseadas nas prioridades das *threads*.

A opção de um *scheduler* cooperativo, sem delegação para o *scheduler* do sistema operacional, se deu pelo motivo de que um *scheduler* próprio, cooperativo, possibilitaria o

total controle da execução, o que é uma necessidade para a construção de jogos eletrônicos. A delegação para o sistema operacional traria benefícios em arquiteturas paralelas, mas não foi tratada neste trabalho.

O modelo de memória definido neste trabalho, ao contrário do modelo de memória de Java, é um modelo simples, que não utiliza o recurso de cópia local de memória para cada *thread*. O modelo de memória utilizado segue o padrão definido em [CESAR FILHO, 2004], onde cada *thread* manipula diretamente a tabela de objetos da máquina virtual.

## 4.2 Estrutura de threads na Virtuosi

Uma *thread* na Virtuosi pode ser marcada como *background*, mesma semântica de *Background* em .NET e também *daemon* em Java.

As operações de uma *thread* podem ser: *start()*, *abort()*, *suspend()*, *resume()* e *yield*(), que servem para iniciar, abortar, suspender, retomar a execução e conceder a execução para a próxima *thread*. A figura 4.1 apresenta os *statements* inseridos no metamodelo para o suporte ao controle de execução de uma *thread*.

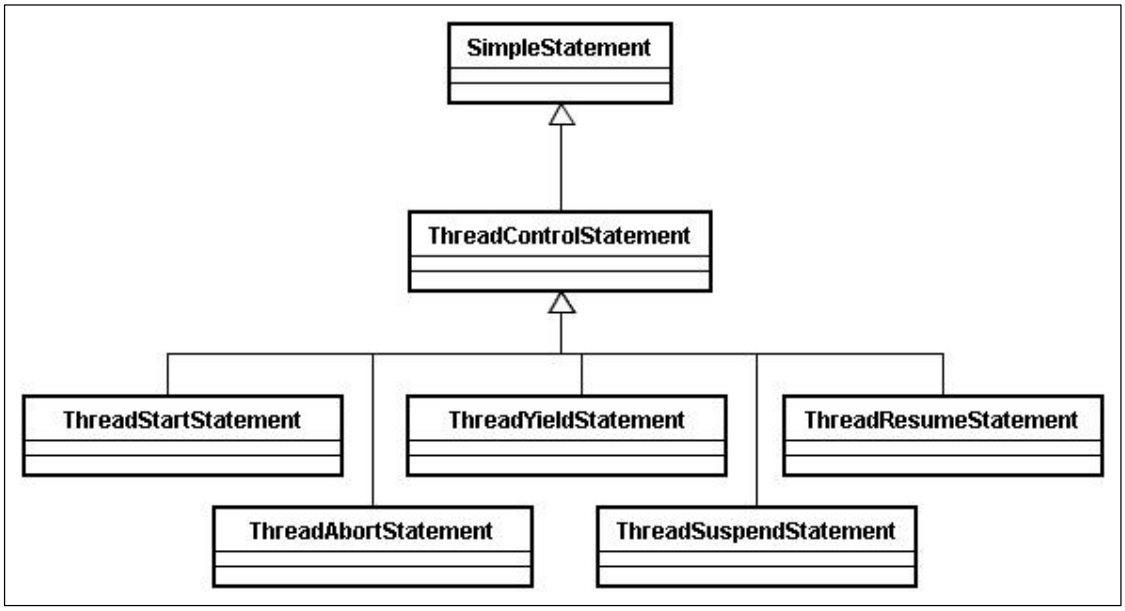


**Figura 4. 1 - *Statements* para controle de execução de uma *thread***

Os *statements ThreadStartStatement* e *ThreadResumeStatement* tornam a *thread* elegível para execução pelo *scheduler*. Os *statements ThreadAbortStatement* e *ThreadSuspendStatement* fazem com que a *thread* não possa ser elegível para execução pelo *scheduler*. O *statement ThreadYieldStatement* faz com que a *thread* em execução vá para o final da fila das *threads* de mesma prioridade, concedendo a execução para a próxima *thread* eleita pelo *scheduler*.

A figura 4.2 apresenta parte de uma classe de biblioteca (API padrão) que encapsula o controle de uma *thread*.

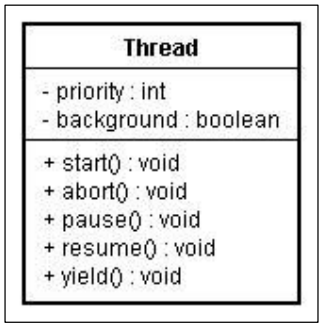


**Figura 4. 2 - Exemplo de classe de biblioteca – Funcionalidades de uma *Thread***

Uma *thread* na arquitetura Virtuosi deve sempre ter como alvo a invocação de um método ou construtor. A figura 4.3 apresenta um exemplo de construtores disponíveis na classe de biblioteca que encapsula uma *thread*.

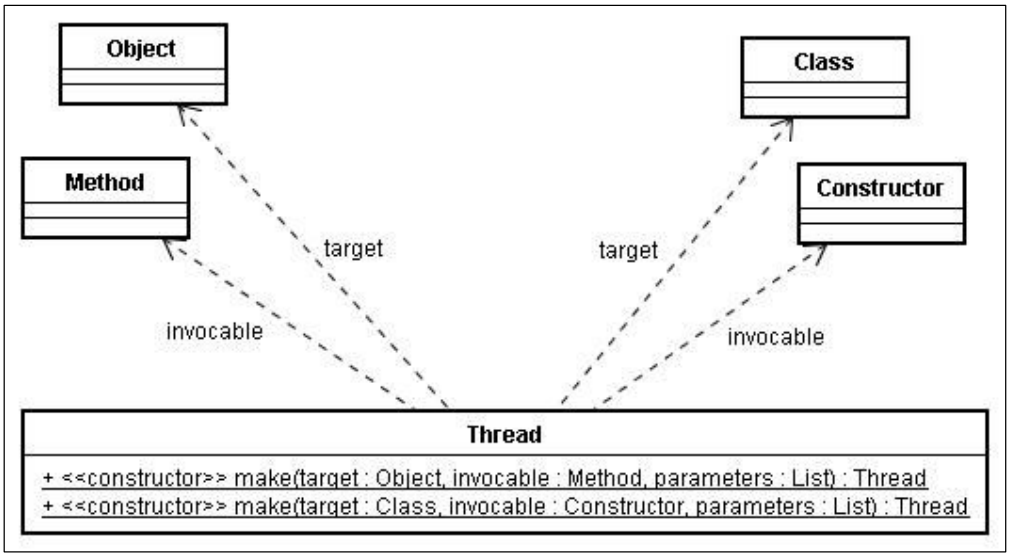


**Figura 4. 3 - Exemplo de classe de biblioteca - Criação de uma *Thread***

O início da execução de uma *thread* (operação *start()*) faz com que o método ou construtor recebido na construção da *thread* seja invocado, conforme discutido a seguir, na seção 4.3.

## *4.3 Invocações assíncronas*

O mecanismo desenvolvido para possibilitar o assincronismo, ou seja, criação de *threads,* é baseado na utilização de um novo *statement* para invocação de construtores ou métodos, denominado *AsyncInvocationStatement*. Existem dois tipos de invocações assíncronas, que são os *statements SimpleAsync* (sem notificação de término) e *RendezvousAsync*[13] (com notificação de término). Ambos os *statements* podem ter como alvo uma invocação de um método *void*. O *statement RendezvousAsync* pode ainda ter como alvo a atribuição do resultado da invocação de um construtor ou um método que possui retorno. Os *statements* desenvolvidos podem ser visualizados na figura 4.4.

[13] O uso de *RendezvousAsync* será tratado com maiores detalhes na seção 4.4.

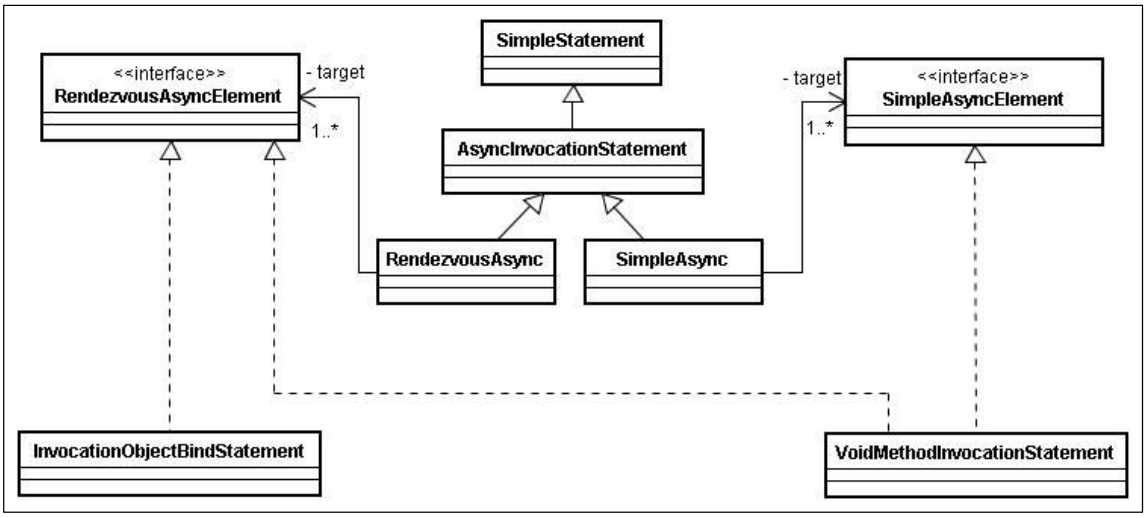


**Figura 4. 4 - *Statements* para invocação assíncrona**

A decisão de assincronismo numa invocação é tomada pelo cliente, ou invocador, porém o lado servidor, ou invocado, pode forçar a forma de invocação, através da utilização dos modificadores de acesso *async* e *sync*, utilizados na definição de um método ou construtor. A figura 4.5 mostra um exemplo de código onde o servidor força o tipo de invocação (síncrona ou assíncrona).

```
1
2   class ServerDefinedInvocationMannerExampleClass {
3
4       sync method void someMethod1() {
5           //This method must be invocated in a synchronous manner.
6       }
7       async method void someMethod2() {
8           //This method must be invocated in an asynchronous manner.
9       }
10  }
11
```

**Figura 4. 5 - Exemplo de definição de forma de invocação no servidor**

Caso o servidor defina um método como *sync*, um cliente não poderá invocar o método de forma assíncrona. Caso o servidor defina um método como *async*, um cliente não poderá invocar o método de forma síncrona. Caso o servidor não defina o modificador de acesso, o cliente pode realizar a invocação de ambas as formas. A tabela 4.1 mostra as possibilidades relacionadas à decisão de assincronismo.

| Client / Server | ***sync*** | ***async*** | **not specified** |
|---|---|---|---|
| ***sync*** | *sync* | not allowed | *sync* |
| ***async*** | not allowed | *async* | not allowed |
| **not specified** | *sync* | *async* | *sync* |

**Tabela 4. 1 - Compatibilidade na decisão de assincronismo**

O uso de assincronismo nas aplicações pode ocorrer de duas formas, assincronismo explícito ou assincronismo implícito.

1. Assincronismo explícito: Nesta forma, o cliente de uma invocação deve utilizar a classe *Thread* da biblioteca padrão. Na forma explícita, o cliente pode manipular livremente a *thread* que representa a atividade, através dos métodos oferecidos pela classe *Thread*. São exemplos de manipulações: Alteração de prioridade da *thread*, interrupção temporária ou definitiva da *thread*, retomada de execução no caso de interrupção temporária, alteração da propriedade *background* de uma *thread*, ou até mesmo aguardar o término de execução da *thread*. A figura 4.6 apresenta um exemplo de código na linguagem *Aram* com a utilização de assincronismo explícito pelo uso da biblioteca padrão da arquitetura Virtuosi. Na linha 12 ocorre a criação do objeto que encapsula a *thread*, porém a execução assíncrona somente ocorre na linha 14, com o comando de início de execução da *thread*. O método *start()* irá fazer com que o método *someMethod()* do objeto *targetObject* seja invocado. Nas linhas 16 e 17 ocorrem algumas das manipulações possíveis em uma *thread*.

```
1
2   class Client {
3
4       method void explicitAsync() {
5
6           //...
7
8           SomeClass targetObject = SomeClass.make();
9           Method targetMethod = targetObject.getClass().getMethod("someMethod");
10          List parameters = List.makeEmpty();
11
12          Thread t = Thread.make(targetObject, targetMethod, parameters);
13
14          t.start();
15
16          t.decreasePriority();
17          t.abort();
18
19          //...
20      }
21  }
22
```

**Figura 4. 6 - Exemplo de assincronismo explícito**

2. Assincronismo implícito: Nesta forma, o cliente de uma invocação assíncrona não possui nenhum controle sobre a *thread* criada para execução do método ou construtor invocado. A única possibilidade de interação com a *thread* é a sincronização através de *Rendezvous*[14]. A figura 4.7 apresenta um exemplo de código na linguagem *Aram* com a utilização de assincronismo implícito (uso da palavra reservada *async*). Neste exemplo, uma segunda *thread* é iniciada a partir da linha 9, para execução do método *someVoidMethod()*.

[14] *Rendezvous* será apresentado na seção 4.4.

```
1
2   class Client {
3
4       method void implicitAsync() {
5           //...
6
7           SomeClass targetObject = SomeClass.make();
8
9           async( targetObject.someVoidMethod() );
10
11          //...
12      }
13  }
14
```

**Figura 4. 7 - Exemplo de assincronismo implícito**

A utilização de assincronismo de forma implícita, por possuir suporte na linguagem de programação, é mais simples e fácil para os programadores, porém a forma explícita, que possui suporte na API da biblioteca padrão, permite um controle maior da execução.

## *4.4 Rendezvous*

*Rendezvous,* ou encontro, é um mecanismo para controle de execução de *threads*. Define um ponto de encontro entre duas *threads*. Seu funcionamento ocorre através do bloqueio da *thread* interessada no encontro, até a finalização de uma *thread* qualquer em execução (*thread* na qual a *thread* bloqueada possui interesse).

O suporte a este mecanismo se dá através do *statement RendezvousAsync*, demonstrado na figura 4.8. O *RendezvousAsync*, assim como o *SimpleAsync*[15], é um *AsyncInvocationStatement*, ou seja, um *statement* para invocação assíncrona. Porém o *RendezvousAsync* pode possuir um bloco de código associado (*CompoundStatement*), e também pode ter como elemento, além de um método *void*, um método que possui retorno ou até mesmo um construtor.

[15] Apresentado na figura 4.4.

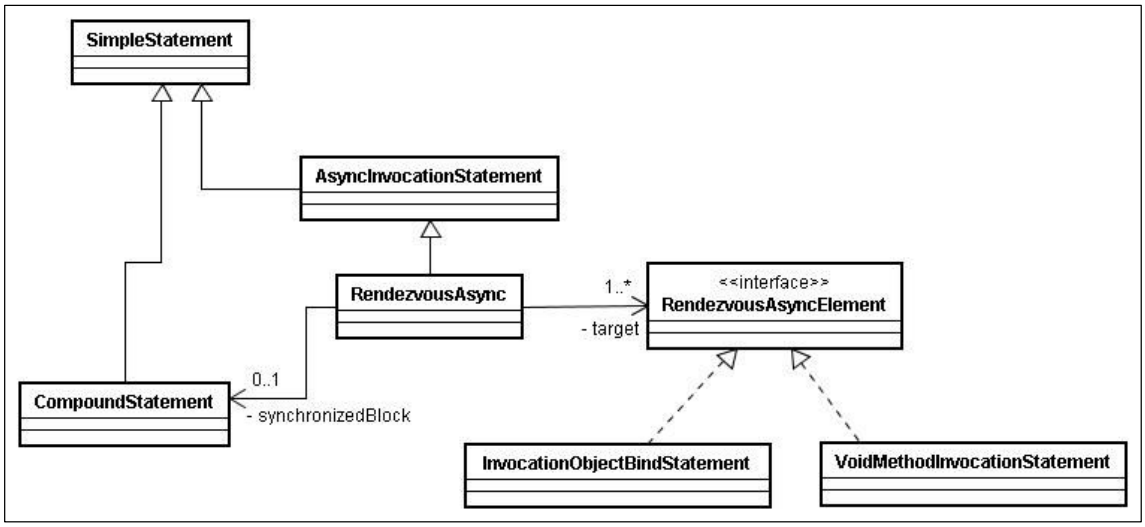


**Figura 4. 8 - Suporte a *Rendezvous***

O uso do mecanismo *Rendezvous* pode ocorrer através da linguagem de programação ou também através da biblioteca de classes padrão. O uso através da linguagem de programação é exemplificado na figura 4.9. Neste exemplo ocorre uma invocação de um construtor de forma assíncrona na linha 8, desta forma, uma nova *thread* é criada para executar o construtor da classe *SomeClass*. A *thread* original continua sua execução no bloco sincronizado, nas linhas 9 até 11. A linha 12 (fim de bloco) determina o ponto de encontro entre as duas *threads*. É importante ressaltar que o uso do objeto *someObject* não é permitido até que ocorra o encontro. A partir da linha 13, o mesmo pode ser utilizado normalmente.

A utilização do *Rendezvous* na linguagem é caracterizada pela associação de um bloco {} ao uso da palavra reservada *async*. O código contido no bloco é executado antes do encontro.

```
1
2   class RendezvousInTheLanguageExampleClass {
3
4       method void someMethod() {
5
6           SomeClass someObject;
7
8           async( someObject = SomeClass.make(); ) {
9               // Do something until the "someObject" gets ready.
10              // Here the use of "someObject" is not allowed.
11              //...
12          }
13
14          //Now I can use the "someObject".
15          someObject.doSomething();
16      }
17  }
18
```

**Figura 4. 9 - Suporte a *Rendezvous* na linguagem**

O uso do *Rendezvous* através da biblioteca de classes (API padrão) ocorre por meio do método *join()* disponível na classe *Thread*. A invocação do mesmo bloqueia a *thread* invocadora até o final da execução da *thread* invocada. O retorno do método *join()* é o retorno do construtor ou método associado na criação da *thread.* A figura 4.10 apresenta um exemplo de uso de *Rendezvous* através da biblioteca de classes. A *thread t* é iniciada na linha 12, através do método *start()*. A *thread* original executa concorrentemente a *t* até a linha 17, onde, através do método *join()*, bloqueia até o término da execução de *t*.

No uso de *Rendezvous* através da biblioteca de classes não há garantias de não utilização da referência para a variável antes do encontro, no exemplo demonstrado na figura 4.10, caso a variável *someObject* fosse utilizada antes da linha 17, que é onde ocorre o encontro, um erro por referência nula poderia ocorrer. Este erro não ocorre no uso de *Rendezvous* por meio da linguagem de programação.

```
1
2   class RendezvousInTheAPIExampleClass {
3
4       method void someMethod() {
5
6           SomeClass someObject;
7
8           Constructor constructor = SomeClass.getConstructor("make");
9           List parameters = List.makeEmpty();
10
11          Thread t = Thread.make(SomeClass.getClass(), constructor, parameters);
12          t.start();
13
14          // Do something until the "someObject" gets ready.
15          // Here, "someObject" is null.
16
17          someObject = t.join(); //It blocks until the make constructor finishes
18
19          //Here, "someObject" is ready.
20          someObject.doSomething();
21      }
22  }
23
```

**Figura 4. 10 - Suporte a *Rendezvous* na API**

## 4.5 Considerações sobre o modelo de assincronismo desenvolvido

A utilização de assincronismo de forma explícita permite maior flexibilidade e controle que a utilização de forma implícita. Porém, a possibilidade de assincronismo de forma implícita simplifica significativamente o desenvolvimento de sistemas concorrentes, além de reduzir as chances de falhas causadas por erro de programação. Um exemplo desta simplicidade pode ser observado no uso implícito de *Rendezvous* para obtenção do retorno de invocações assíncronas. O assincronismo implícito pode ser considerado uma vantagem do modelo de concorrência desenvolvido sobre os modelos de concorrência de outras arquiteturas tais como Java e .NET.

Uma outra vantagem deste modelo sobre o modelo de Java e .NET é que este modelo permite maior segurança, pois o servidor de uma invocação pode forçar o modo de invocação (síncrono ou assíncrono) mais apropriado, impedindo erros de concorrência no cliente (invocador) causados por uso de um modo de invocação inapropriado.

# 5 Controle de concorrência

O assincronismo possibilita com que inúmeras *threads* executem de forma concorrente. Estas *threads* podem executar para resolver problemas distintos, ou até mesmo problemas em comum. Quando *threads* acessam recursos em comum, ou seja, compartilham recursos, ou de alguma forma precisam se comunicar para um controle de sua execução, surge a necessidade de mecanismos para o controle de concorrência entra as *threads*. Tais mecanismos devem garantir o sincronismo na execução das *threads*, e devem também garantir que os recursos compartilhados sejam acessados de forma correta, de forma sincronizada.

Este capítulo apresenta os mecanismos para controle de concorrência desenvolvidos e suas particularidades. Exemplifica a utilização dos mecanismos e demonstra como os mesmos foram definidos na arquitetura Virtuosi.

## *5.1 Travas*

Os mecanismos para controle de acesso a recursos compartilhados desenvolvidos neste trabalho são baseados em travas. Dois níveis de granularidade são suportados, travas para escrita e travas para leitura. As travas para escrita permitem que somente uma *thread* por vez consiga acesso a trava, com isto todas as outras *threads* que tentarem obter uma trava para escrita ou leitura ficarão bloqueadas em uma fila, chamada *EntryQueue,* aguardando a liberação da trava. As travas para escrita possibilitam exclusão mútua. Travas para leitura permitem que várias *threads* possam obter uma trava para leitura, e enquanto as mesmas não liberam a trava, nenhuma outra *thread* consegue obter a trava para escrita (aguardam na *EntryQueue*).

### 5.1.1 Classes concorrentes

Travas são associadas a objetos de classes de um tipo em especial, denominados *Classes Concorrentes*. Quando uma classe é marcada como concorrente, a mesma herda métodos que permitem o uso de travas.

Dois tipos de classes concorrentes foram desenvolvidos: *Monitor class* e *Lockable class*.

- *Monitor class*: Uma classe do tipo *Monitor* possui todos os seus métodos sincronizados através de uma trava para escrita associada às instâncias da classe. Isto faz com que somente uma *thread* por vez possa invocar métodos de objetos que são instâncias de classes do tipo *Monitor*.
- *Lockable class*: Uma classe do tipo *Lockable* é uma classe que pode ser alvo de uma trava para escrita ou leitura. Todo acesso a objetos desta classe somente é permitido por *threads* que tenham adquirido a trava associada à instância, para escrita ou leitura. Acessos por *threads* que não adquiram a trava associada à instância não são permitidos.

Pode-se considerar uma *Classe Concorrente* como uma classe que foi projetada para que suas instancias sejam acessadas por múltiplas *threads*, ou seja, uma classe cujas instancias possam ser acessadas de forma sincronizada.

### 5.1.2 Uso de travas

O uso de travas pode ocorrer através da linguagem de programação ou através da biblioteca padrão de classes. São possíveis quatro formas de uso de travas, são eles: uso através de *Monitor class*, uso através de qualificador de método, uso através de qualificador de bloco e finalmente uso explícito através da aquisição e liberação da trava.

- Uso através de *Monitor class*: Esta forma de uso de uma trava ocorre em todas as classes do tipo *Monitor*. O tipo de trava utilizado é implícito: trava para escrita. A obtenção e liberação da trava ocorrem automaticamente no início e fim dos métodos de uma classe *Monitor*. O alvo da trava sempre é a instância da classe *Monitor* em questão. E o modo de sincronização permite somente sincronização do tipo *único leitor, único escritor*[16]. O uso de travas através de uma classe *Monitor* não permite métodos ou blocos não sincronizados na classe. A figura 5.1 apresenta um exemplo de uso de trava através de uma classe *Monitor*. Uma classe é marcada como uma classe do tipo *Monitor* através da palavra reservada *monitor* como qualificador da classe.

```
1
2   monitor class MonitorExampleClass {
3
4       String someAtribute;
5       constructor make() {
6           this.name = String.make("MonitorExample");
7       }
8
9       //All methods are writeLocked.
10
11      method void someMethod1() {
12          //This is a writeLocked method.
13      }
14
15      method String getSomeAtribute() {
16
17          //This is a writeLocked method.
18          return this.someAtribute;
19      }
20  }
21
```

**Figura 5. 1 - Exemplo de *Monitor class***

- Uso através de qualificador de método: Esta forma de uso de uma trava ocorre quando um método é marcado através de um qualificador que pode ser *readLocked* (trava para leitura) ou *writeLocked* (trava para escrita). O tipo de

[16] Do ingles, *single reader, single writer*

trava utilizado é explícito, travas para leitura ou escrita. A obtenção e liberação da trava ocorrem automaticamente no início e fim dos métodos marcados. O alvo da trava sempre é a instância da classe que define o método em questão. E o modo de sincronização permite sincronização do tipo *múltiplos leitores, único escritor*[17]. Travas para escrita possibilitam sincronização do tipo exclusão mútua, enquanto travas para leitura permitem sincronização do tipo múltiplos leitores, único escritor. Em Java, o modificador de método possível é o modificador *synchronized*, que permite somente exclusão mútua. O uso de travas através de um qualificador de método não permite trechos não sincronizados no método. A figura 5.2 apresenta um exemplo de uso de travas através de qualificadores de métodos. Na linha 6 é mostrado um método que utiliza uma trava para escrita, na linha 9 é mostrado um método que utiliza uma trava para leitura. Na utilização de travas através de qualificadores de métodos, as travas sempre são relacionadas com as instâncias da classe, no caso do exemplo da figura 5.2, instâncias da classe *MethodQualifierLockExampleClass*.

```
1
2   lockable class MethodQualifierLockExampleClass {
3
4       //...
5
6       writeLocked method void someMethod1() {
7           //This is a writeLocked method.
8       }
9       readLocked method void someMethod2() {
10          //This is a readLocked method.
11      }
12  }
13
```

**Figura 5. 2 - Exemplo de uso de travas através de qualificadores de métodos**

- Uso através de qualificador de bloco: Esta forma de uso de uma trava ocorre quando um bloco de código é marcado através de um qualificador que pode ser

[17] Do inglês, *multiple readers, single writer*

*readLocked* (trava para leitura) ou *writeLocked* (trava para escrita). O tipo de trava utilizado é explícito, travas para leitura ou escrita. A obtenção e liberação da trava ocorrem automaticamente no início e fim do bloco. O alvo da trava pode ser qualquer instância de classes do tipo *Lockable*. O modo de sincronização permite sincronização do tipo *múltiplos leitores, único escritor*. O uso de travas através de um qualificador de bloco permite trechos não sincronizados no mesmo método, e também não impede que outros métodos da classe não possuam sincronização. A figura 5.3 apresenta um exemplo de uso de travas através de qualificadores de blocos.

```
1
2   class BlockQualifierLockExampleClass {
3
4       //...
5
6       method void someMethod1() {
7
8           //Some non-synchronized code.
9
10          writeLocked(anyLockableObject) {
11              //This is a writeLocked code.
12          }
13
14          //Some non-synchronized code.
15
16          readLocked(anyLockableObject) {
17              //This is a readLocked code.
18          }
19
20          //Some non-synchronized code.
21      }
22
23      method void someMethod2() {
24          //This is a non-synchronized method.
25      }
26  }
27
```

**Figura 5. 3 - Exemplo de uso de travas através de qualificadores de blocos**

Neste exemplo da figura 5.3, é mostrada uma classe que possui trechos não sincronizados e trechos sincronizados dentro do mesmo método. Na linha 10 é realizada uma sincronização para escrita tendo como alvo o objeto *anyLockableObject*, é importante ressaltar que neste caso, o alvo da trava não é *this*, pois pode ser qualquer objeto do tipo *Lockable* (inclusive *this* se a classe em questão for do tipo *Lockable*). Na linha 11 e 12 ocorre o trecho sincronizado para escrita. Na linha 14 ocorre novamente código não sincronizado. Na linha 16, é realizada uma sincronização para leitura novamente no objeto *anyLockableObject*. Na linha 17 e 18 ocorre código sincronizado para leitura e finalmente, na linha 20 ocorre novamente código não sincronizado. Na linha 23 é mostrado um segundo método não sincronizado.

- Uso explícito através da aquisição e liberação da trava: Esta forma de uso de uma trava ocorre através da chamada dos métodos *readLock()* (trava para leitura) e *writeLock()* (trava para escrita). O tipo de trava utilizado é explícito, travas para leitura ou escrita. A liberação da trava ocorre através da chamada do método *release()* para uma referência de *Lock* obtida, mas não é garantida automaticamente, pois depende do programador. O alvo da trava pode ser qualquer instância de classes do tipo *Lockable*. O modo de sincronização permite sincronização do tipo *múltiplos leitores, único escritor*. O uso de travas através da aquisição e liberação permite trechos não sincronizados no mesmo método, e também não impede que outros métodos da classe não possuam sincronização. A figura 5.4 apresenta um exemplo de uso de travas através da aquisição e liberação explícita das travas. Neste exemplo da figura 5.4, é mostrado uma classe que possui trechos não sincronizados e trechos sincronizados dentro do mesmo método. Na linha 10 o método *writeLock()* é invocado, fazendo com que a *thread* corrente adquira a trava do objeto *anyLockableObject* para escrita. O código da linha 12, portanto, é um código sincronizado. Na linha 14 a trava é liberada, conseqüentemente o código da linha 16 é código não sincronizado.

```
1
2   class ExplicitLockExampleClass {
3
4       //...
5
6       method void someMethod1() {
7
8           //Some non-synchronized code.
9
10          Lock lock1 = anyLockableObject.writeLock();
11
12          //This is a writeLocked code.
13
14          lock1.release();
15
16          //Some non-synchronized code.
17
18          Lock lock2 = anyLockableObject.readLock();
19
20          //This is a readLocked code.
21
22          lock2.release();
23
24          //Some non-synchronized code.
25      }
26
27      method void someMethod2() {
28          //This is a non-synchronized method.
29      }
30  }
31
```

**Figura 5. 4 - Exemplo de uso de travas através de aquisição e liberação explícita**

Na linha 18 ocorre novamente a obtenção da trava associada ao objeto *anyLockableObject*, desta vez para leitura. O código da linha 20 consequentemente é sincronizado. Na linha 22 ocorre a liberação da trava, e com isto o código da linha 24 é não sincronizado. Na linha 27 é mostrado um método não sincronizado. O uso explícito de uma trava permite ainda mais uma funcionalidade, que é a tentativa não bloqueante de obtenção de uma trava. Isto ocorre através do método *tryReadLock()* e *tryWriteLock()*, que caso consiga adquirir a trava retorna uma referência para a mesma e caso não consiga retorna uma referência nula. Os métodos *tryReadLock()* e

*tryWriteLock()* são necessários para tentativas de sincronização com baixo custo.

A tabela 5.1 apresenta um resumo quanto ao uso de travas, o momento de liberação e o tipo de trava possível.

| Lock usage | Lock release | Lock type |
|---|---|---|
| **monitor class** | **block end** | implicit |
| **method qualifier** | **block end** | explicit |
| **block qualifier** | **block end** | explicit |
| **lock acquire/release** | **anywhere** | explicit |

**Tabela 5. 1 - Uso de travas - Liberação e Tipo**

A tabela 5.2 apresenta um resumo quanto ao uso de travas e o modo de sincronização e o alvo possível de uma sincronização. SRSW é uma abreviação para *single reader, single writer*, e MRSW é uma abreviação para *multiple readers, single writer*.

| Lock usage | Synchronization mode | Synchronization target |
|---|---|---|
| **monitor class** | SRSW | this (Monitor instance) |
| **method qualifier** | MRSW | this (Lockable instance) |
| **block qualifier** | MRSW | any (Lockable instance) |
| **lock acquire/release** | MRSW | any (Lockable instance) |

**Tabela 5. 2 - Uso de travas - Modo de sincronização e Alvo**

A tabela 5.3 apresenta um resumo quanto ao uso de travas e permissão de trechos não sincronizados na própria classe e no próprio método.

| Lock usage | Non-synchronized methods | Non-synchronized blocks |
|---|---|---|
| **monitor class** | not allowed | not allowed |
| **method qualifier** | not allowed | not allowed |
| **block qualifier** | allowed | allowed |
| **lock acquire/release** | allowed | allowed |

**Tabela 5. 3 - Uso de travas - Permissão de trechos não sincronizados**

O suporte à travas no metamodelo da Virtuosi foi possível devido à inserção da classe *LockReference*, como um novo tipo de referência. Novos *statements* também foram criados para possibilitar as operações de aquisição, tentativa de aquisição e liberação de uma trava. A figura 5.5 demonstra as inclusões no metamodelo para suporte a travas.

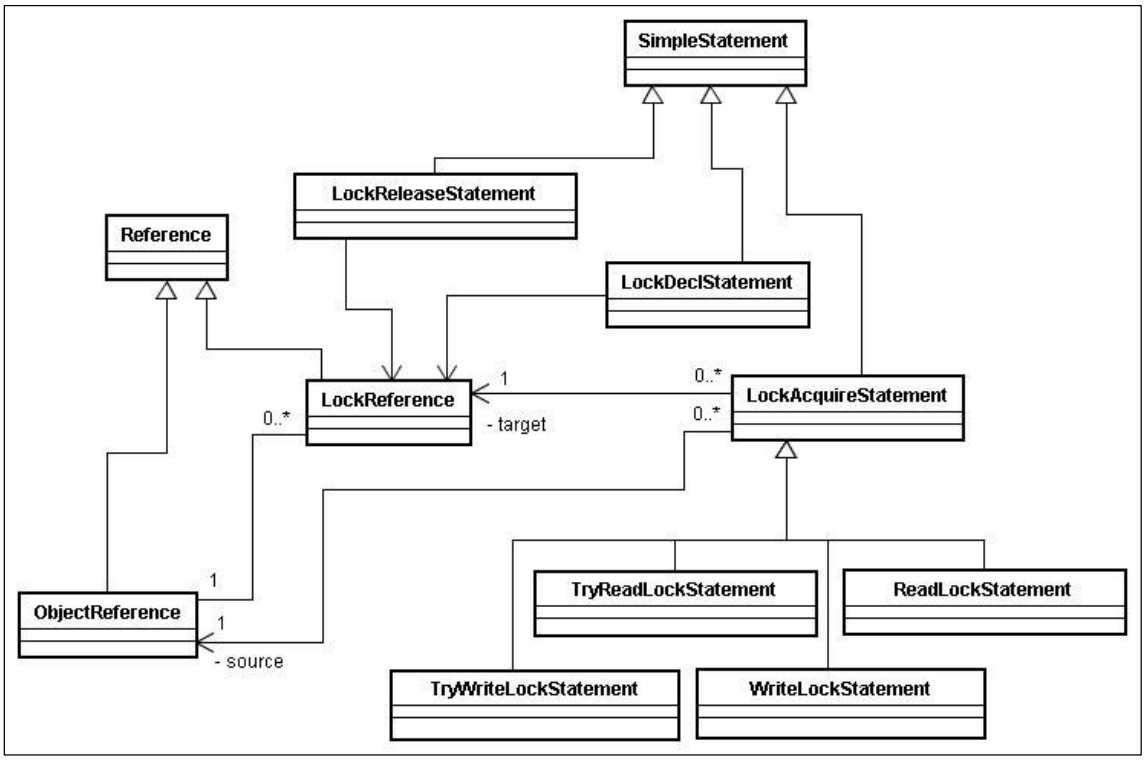


**Figura 5. 5 - Metamodelo - Suporte a travas**

Os *statements* de aquisição de trava possuem uma superclasse em comum: a classe *LockAcquireStatement*. Toda aquisição de uma trava tem como fonte uma referência para um objeto de uma classe concorrente e como destino uma referência para uma *LockReference*. Foi inserido também um *statement* para declaração de uma *LockReference*.

## 5.2 Condições de sincronização

Quando uma classe é marcada como concorrente, além de herdar funcionalidades relacionadas com o uso de travas, a classe herda também funcionalidades relacionadas com o uso de condições de sincronização. Os métodos disponibilizados para condições de sincronização são: *await()*, *signalAndContinue()*, *signalAllAndContinue(), signalAndWait()*, e *signalAllAndWait()*.

- *await()*: O método *await()* faz com que a *thread* invocadora deste método interrompa sua execução e aguarde em uma fila a notificação de satisfação da condição. A interrupção da *thread* faz com que a mesma libere as travas que possui, dando oportunidade para outras *threads* adquirirem as travas.
- *signalAndContinue()*: O método *signalAndContinue()* faz com que a primeira *thread* da fila de espera da condição possa retomar sua execução, movendo a *thread* da fila de espera da condição para a fila de *threads* prontas para executar do *scheduler*, e inserindo a mesma nas filas de obtenção (*EntryQueue*) de cada trava associada a *thread* notificada. Porém a *thread* que notificou a satisfação da condição continuará a execução até sua finalização ou um possível bloqueio. Caso a mesma possua alguma trava na qual a *thread* notificada possui interesse, a *thread* notificada terá que aguardar a liberação da trava para prosseguir.
- *signalAllAndContinue()*: O método *signalAllAndContinue()* tem a mesma semântica do método *signalAndContinue()*, porém ao invés de fazer com que somente a primeira *thread* da fila de espera da condição possa retomar a execução, faz com que todas as *threads* da fila de espera da condição possam retomar a execução.
- *signalAndWait()*: O método *signalAndWait()* faz com que a *thread* notificante libere suas travas e interrompa sua execução temporariamente, dando condições para que a primeira *thread* da fila de espera da condição possa retomar sua execução, movendo a *thread* da fila de espera da condição para a fila de *threads* prontas para executar do *scheduler*, e inserindo a mesma nas filas de obtenção prioritária (*ReadyQueue*) de cada trava associada à *thread* notificada. Quando uma *thread* entra na fila de obtenção prioritária

(*ReadyQueue*) de uma trava, nenhuma nova *thread* além da primeira *thread* da *ReadyQueue* pode obter a trava em questão. Caso a *thread* notificada necessite acesso de escrita e outras *threads* já possuam acesso de leitura, a *thread* notificada aguarda até a liberação da trava pelas *threads* leitoras.

- *signalAllAndWait()*: O método *signalAllAndWait()* tem a mesma semântica do método *signalAndWait()*, porém ao invés de fazer com que somente a primeira *thread* da fila de espera da condição retome a execução, faz com que todas as *threads* da fila de espera da condição retomem a execução.

A figura 5.6 mostra os possíveis estados de uma *thread* quanto ao uso de travas e condições de sincronização.

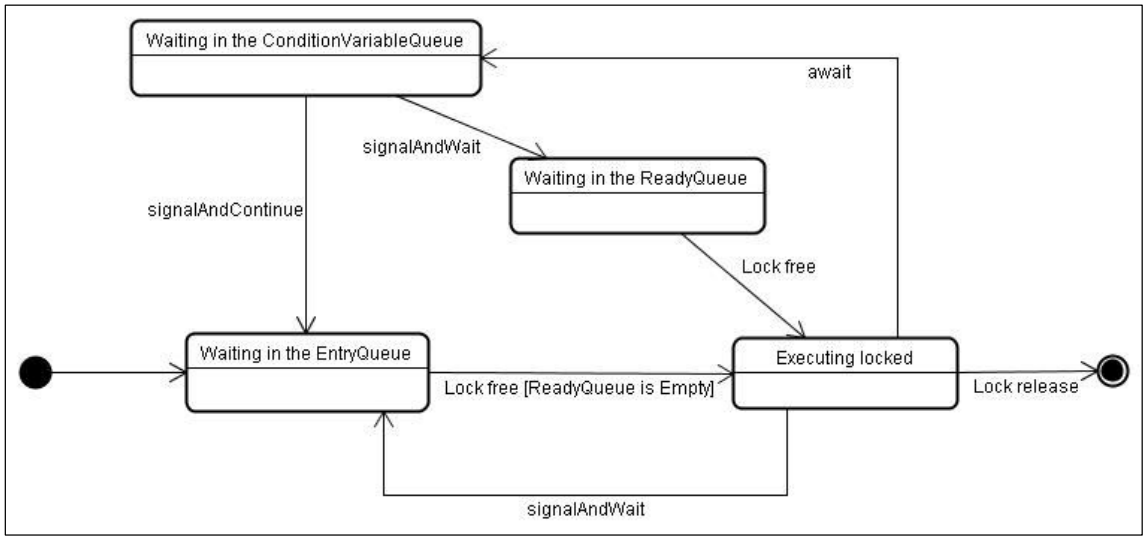


**Figura 5. 6 - Estados de uma *thread* no uso de travas e condições de sincronização**

Uma outra forma de compreender o relacionamento entre *threads*, travas e condições é mostrada nas figuras 5.7, que considera o estado de uma trava com relação à sua granularidade (escrita) e os eventos possíveis gerados por *threads* em execução.

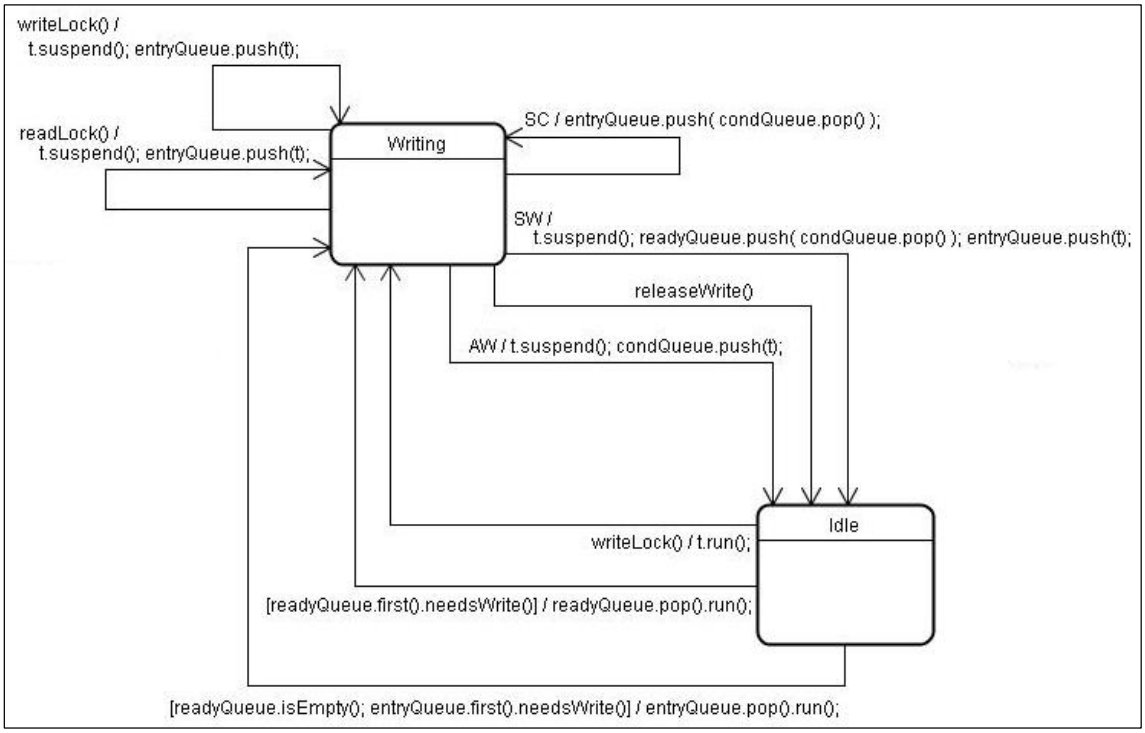


**Figura 5. 7 - Estados de uma trava com relação à granularidade - *Writing***

Na figura 5.7, *t* é a *thread* geradora do evento, *AW* significa uma invocação do método *await()* de alguma condição, *SW* significa uma invocação do método *signalAndWait()* de alguma condição, e finalmente, *SC* significa a invocação do método *signalAndContinue()* de alguma condição.

A figura 5.8 apresentada a seguir considera o estado de uma trava com relação a sua granularidade (leitura) e os eventos possíveis gerados por *threads* em execução. A figura 5.8 utiliza a mesma notação da figura 5.7.

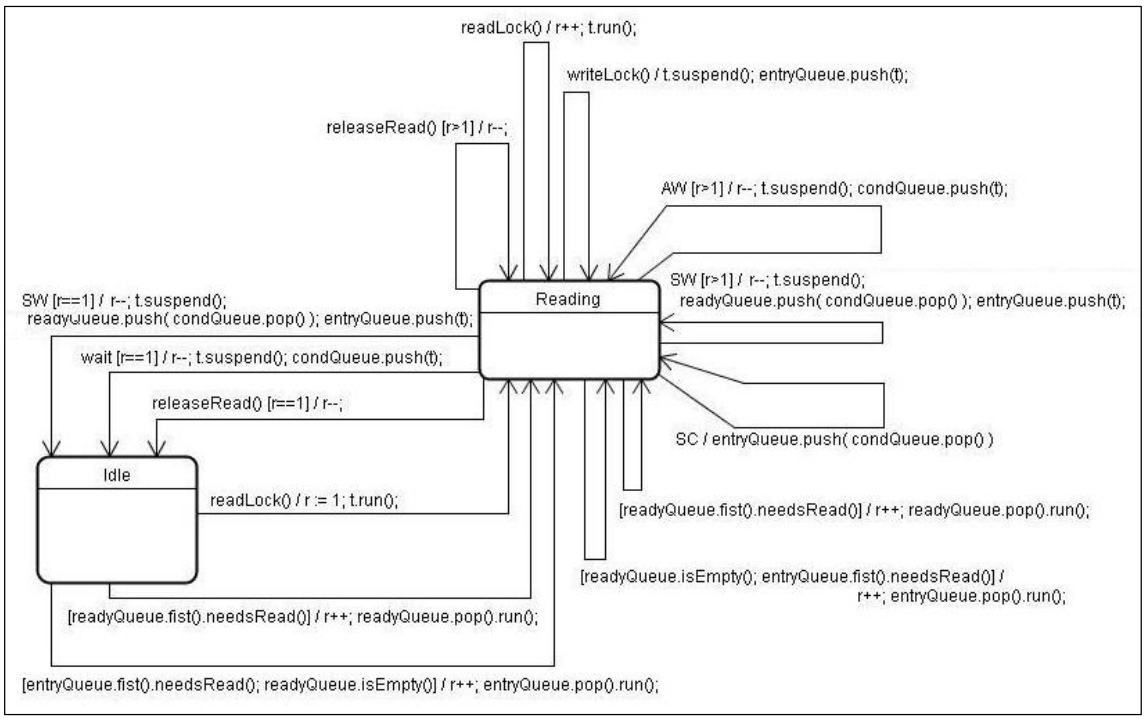


**Figura 5. 8 - Estados de uma trava com relação à granularidade - *Reading***

O suporte a condições de sincronização no metamodelo da Virtuosi foi possível devido à inserção da classe *ConditionReference*, como um novo tipo de referência. Novos *statements* também foram criados para possibilitar as operações de declaração, criação, atribuição, espera e notificação. A figura 5.9 demonstra as inclusões no metamodelo para suporte a condições de sincronização.

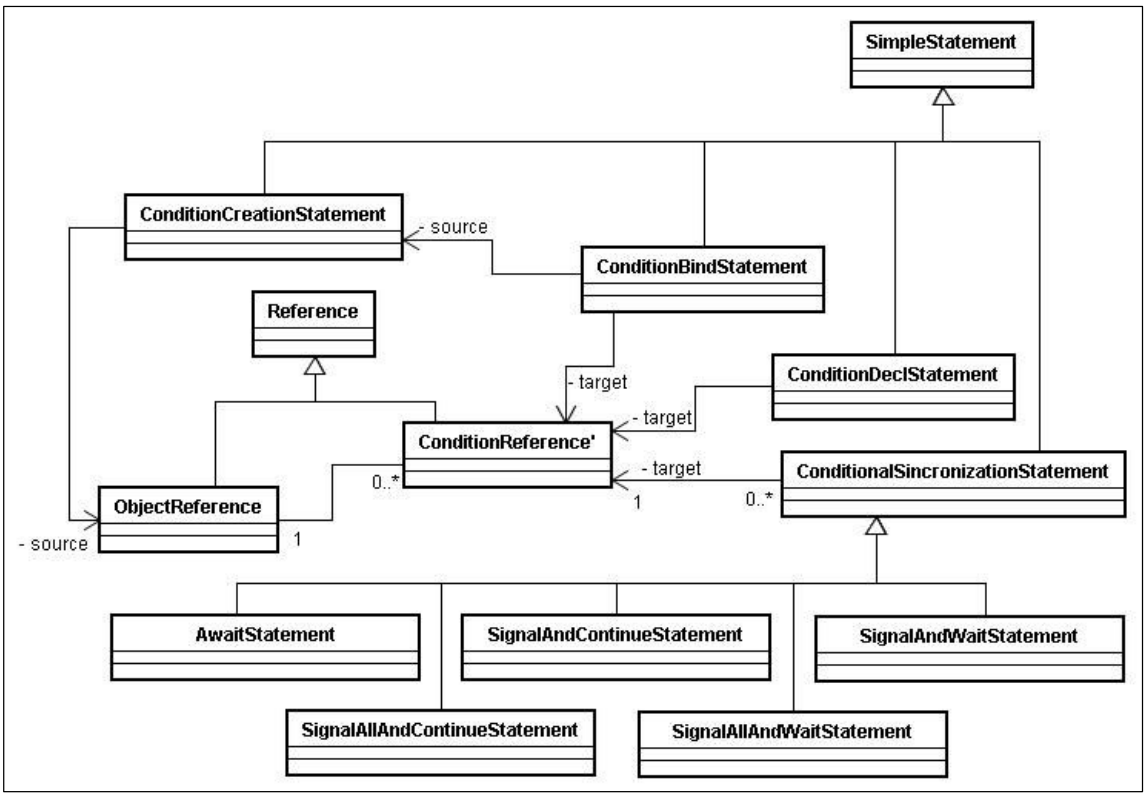


**Figura 5. 9 - Metamodelo - Suporte a condições de sincronização**

## *5.3 Dispositivos de controle e sincronização*

Com a utilização de travas e condições de sincronização, diversos outros mecanismos de controle de concorrência podem ser desenvolvidos. Para ilustrar esta possibilidade, serão apresentadas a seguir duas implementações de semáforos, quatro versões de implementação de uma fila bloqueante, cada versão utilizando uma das quatro abordagens de uso de travas desenvolvidas, e finalmente um mecanismo de temporização que pode ser agregado ao modelo de controle de execução e controle de concorrência.

### 5.3.1 Semaphore

A classe *Semaphore* possibilita o acesso compartilhado de *N threads* a um recurso. A figura 5.10 apresenta um exemplo de semáforo implementado utilizando uma classe concorrente do tipo *Monitor* e uma condição de sincronização.

```
1
2   monitor class Semaphore {
3
4       Integer permits;
5       Condition available;
6
7       constructor make(Integer initialPermits) export all {
8
9           this.permits = initialPermits;
10          this.available = this.newCondition();
11      }
12
13      method void up() export all {
14
15          this.permits.inc();
16          this.available.signalAndContinue();
17      }
18
19      method void down() export all {
20
21          if(this.permits.lessThan(1)) {
22              this.available.await();
23          }
24          this.permits.dec();
25      }
26  }
27
```

**Figura 5. 10 - Implementação da classe *Semaphore***

Na criação de um *Semaphore* (linha 7) é passado como parâmetro o número de *threads* que podem acessar o semáforo ao mesmo tempo. Todos os métodos da classe são mutuamente exclusivos, devido à utilização de uma classe do tipo *Monitor*. Na linha 13 é descrito o método de sinalização do semáforo, cada invocação deste método incrementa o número permissões de utilização, ou seja, o número de *threads* que ainda poderão utilizar simultaneamente o semáforo, além de notificar possíveis *threads* que estejam aguardando para execução (linha 16).

Na linha 19 é descrito o método de requisição de utilização. Caso o número de permissões seja menor que 1 (um), é utilizado uma condição de sincronização para aguardar a notificação de disponibilidade (linha 22). Assim que alguma *thread* notificar disponibilidade através do método *up()*, a *thread* prossegue sua execução executando o decremento do número de permissões restantes.

### 5.3.2 Mutex

Um *Mutex* é um tipo especial de semáforo no qual somente uma *thread* por vez pode executar. A figura 5.11 apresenta a implementação de um *Mutex* também com a utilização de uma classe do tipo *Monitor* e uma condição de sincronização.

```
1
2   monitor class Mutex {
3
4       Boolean isInUSe;
5       Condition available;
6
7       constructor make() export all {
8           this.isInUse = Boolean.make(false);
9           this.available = this.newCondition();
10      }
11
12      method void up() export all {
13
14          this.isInUse.setFalse();
15          this.available.signalAndContinue();
16      }
17
18      method void down() export all {
19
20          if(this.isInUse) {
21              this.available.await();
22          }
23          this.isInUse.setTrue();
24      }
25  }
26
```

**Figura 5. 11 - Classe *Mutex***

A diferença na implementação da classe *Mutex* e da classe *Semaphore* está na validação de disponibilidade, que na classe *Semaphore* utiliza-se um contador e na classe *Mutex* utiliza-se uma variável booleana (linha 20).

### 5.3.3 BlockingQueue

A implementação de uma fila de eventos bloqueante foi escolhida para ilustrar os quatro modos de uso de travas desenvolvidos. Nesta seção serão apresentadas quatro implementações que possuem a mesma semântica, variando apenas na forma de uso de travas.

A fila bloqueante é uma classe que possui um método *push()* utilizado para adicionar evento na fila, e um método *pop()*, utilizado para remover um evento da fila. Caso a fila não possua elementos, a invocação do método *pop()* deve bloquear a *thread* invocadora até a que um elemento esteja disponível, ou seja, alguma outra *thread* invocou o método *push()*.

A figura 5.12 apresenta a primeira versão, utilizando uma classe do tipo *Monitor*.

```
1
2   monitor class BlockingQueue {
3
4       List listOfEvents;
5       Condition available;
6
7       constructor make() export all {
8           this.listOfEvents = List.make();
9           this.available = this.newCondition();
10      }
11
12      method void push(Event event) export all {
13
14          this.listOfEvents.add(event);
15          this.available.signalAndContinue();
16      }
17
18      method Event pop() export all {
19
20          if(this.listOfEvents.isEmpty()) {
21              this.available.await();
22          }
23          Event event = this.listOfEvents.getFirst();
24          this.listOfEvents.removeFirst();
25          return event;
26      }
27  }
28
```

**Figura 5. 12 - *BlockingQueue* – Classe do tipo *Monitor***

A implementação da fila bloqueante utiliza-se de uma condição para indicar a disponibilidade de elementos na fila. Cada vez que um elemento é inserido, uma das *threads* que aguardam um novo elemento é notificada (linha 15). Quando uma *thread* invoca o método *pop()* para adquirir um elemento da fila, é realizado um teste verificar a disponibilidade, caso não haja um elemento disponível, a *thread* é bloqueada (linha 21) e só retoma sua execução quando alguma outra *thread* inserir um novo elemento na fila. Por se tratar de uma classe do tipo *Monitor*, os métodos *push()* e *pop()* são mutuamente exclusivos (quando aplicados sobre a mesma instância de objeto do tipo *Monitor*) .

A figura 5.13 apresenta a mesma classe implementada com qualificadores de método ao invés de uma classe do tipo *Monitor*.

```
1
2   lockable class BlockingQueue {
3
4       List listOfEvents;
5       Condition available;
6
7       constructor make() export all {
8           this.listOfEvents = List.make();
9           this.available = this.newCondition();
10      }
11
12      writeLocked method void push(Event event) export all {
13
14          this.listOfEvents.add(event);
15          this.available.signalAndContinue();
16      }
17
18      writeLocked method Event pop() export all {
19
20          if(this.listOfEvents.isEmpty()) {
21              this.available.await();
22          }
23          Event event = this.listOfEvents.getFirst();
24          this.listOfEvents.removeFirst();
25          return event;
26      }
27  }
28
```

**Figura 5. 13 - *BlockingQueue* - Qualificador de método**

A diferença no uso de travas entre a primeira implementação (figura 5.12) e esta implementação (figura 5.13) é visível na definição do tipo da classe, que é uma classe do tipo *Lockable* ao invés de uma classe *Monitor* e com isto, todos os métodos precisaram ser marcados como métodos de acesso exclusivo (*writeLocked*), ou seja, os mesmos adquirem travas para escrita e tem como alvo a própria instância da fila bloqueante.

A figura 5.14 apresenta a versão utilizando qualificadores de bloco ao invés de qualificadores de método.

```
1
2   lockable class BlockingQueue {
3
4       List listOfEvents;
5       Condition available;
6
7       constructor make() export all {
8           this.listOfEvents = List.make();
9           this.available = this.newCondition();
10      }
11
12      method void push(Event event) export all {
13
14          writeLocked(this) {
15              this.listOfEvents.add(event);
16              this.available.signalAndContinue();
17          }
18      }
19
20      method Event pop() export all {
21
22          writeLocked(this) {
23              if(this.listOfEvents.isEmpty()) {
24                  this.available.await();
25              }
26              Event event = this.listOfEvents.getFirst();
27              this.listOfEvents.removeFirst();
28              return event;
29          }
30      }
31  }
32
```

**Figura 5. 14 - BlockingQueue - Qualificador de bloco**

A diferença no uso de travas entre a versão com qualificador de método e a versão com qualificador de bloco é que na versão com qualificador de bloco, o alvo da sincronização

deve ser explícito (*this* no caso do exemplo - linhas 14 e 22), e também, na versão com qualificador de bloco, somente parte do método é sincronizado, e não todo o método.

Finalmente, a figura 5.15 apresenta a versão que utiliza aquisição (linhas 14 e 22) e liberação (linhas 17 e 28) explícita de travas.

```
1
2   lockable class BlockingQueue {
3
4       List listOfEvents;
5       Condition available;
6
7       constructor make() export all {
8           this.listOfEvents = List.make();
9           this.available = this.newCondition();
10      }
11
12      method void push(Event event) export all {
13
14          Lock lock = this.writeLock();
15          this.listOfEvents.add(event);
16          this.available.signalAndContinue();
17          lock.release();
18      }
19
20      method Event pop() export all {
21
22          Lock lock = this.writeLock();
23          if(this.listOfEvents.isEmpty()) {
24              this.available.await();
25          }
26          Event event = this.listOfEvents.getFirst();
27          this.listOfEvents.removeFirst();
28          lock.release();
29          return event;
30      }
31  }
32
```

**Figura 5. 15 - *BlockingQueue* - Aquisição e liberação explícita**

A diferença entre a versão com modificador de bloco e a versão com aquisição e liberação explícita da trava é que a versão com liberação explícita, a liberação da trava fica a cargo do programador, e não ocorre automaticamente como nas versões anteriores.

Todas as quatro implementações demonstradas possuem o mesmo comportamento, diferem apenas no modo de utilização das travas.

## 5.4 Mecanismos de temporização

Uma necessidade comum em sistemas concorrentes é a possibilidade de controles temporais associados aos mecanismos de controle de concorrência, isto é, realizar operações de concorrência baseadas em tempo real. O presente trabalho define uma forma extensível de inserção de controle temporal associado aos mecanismos de controle de concorrência desenvolvidos. O mecanismo é baseado num novo tipo de referência, denominado *TimerReference*, que é uma referência para um temporizador. Para cada tipo de *TimerReference* deve haver um correspondente *statement* desenvolvido na máquina virtual que encapsule a criação (dado uma referência para um literal) de um mecanismo de controle temporal para o tipo de *TimerReference* em questão. Por exemplo, um *MilisecondsTimerReference* precisa necessariamente de um *statement*, desenvolvido na máquina virtual, que possa criar um mecanismo de controle temporal baseado em milisegundos. A figura 5.16 apresenta um exemplo com dois tipos de *TimerReference* (um baseado em milisegundos e outro baseado em nanosegundos), com seus respectivos *statements* de suporte à criação na máquina virtual.

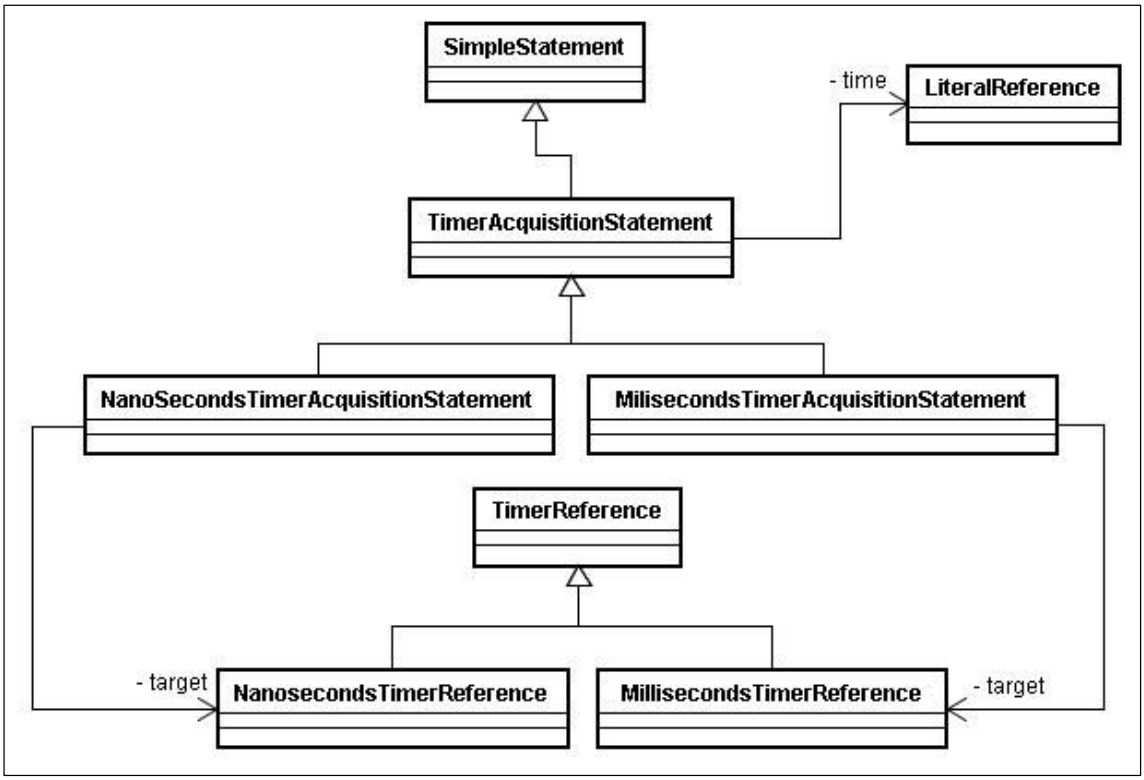


**Figura 5. 16 – Metamodelo - Suporte a controle temporal**

Este novo tipo de referência pode ser utilizado em três situações: Suspensão temporal de uma *thread*, tempo máximo de obtenção de uma trava e finalmente tempo máximo de espera por uma condição de sincronização.

1. Utilização na suspensão temporal de uma *thread*: Nesta forma de uso, uma *TimerReference* e um valor temporal na forma literal devem ser utilizados em conjunto com o *statement ThreadSuspendStatement*. A figura 5.17 apresenta um exemplo de código fonte com o uso de suspensão temporal de uma *thread*. Neste exemplo, a suspensão seria encapsulada pelo método *sleep()*, na linha 16. A invocação faz com que a *thread* corrente interrompa sua execução por 10 milisegundos.

```
1
2   class ThreadSuspendExample {
3
4       method void example() {
5
6           //...
7
8           SomeClass targetObject = SomeClass.make();
9           Method targetMethod = targetObject.getClass().getMethod("someMethod");
10          List parameters = List.makeEmpty();
11
12          Thread t = Thread.make(targetObject, targetMethod, parameters);
13
14          t.start();
15
16          t.sleep(Timer.makeMilisecondsTimer(10));
17
18          //...
19      }
20  }
```

**Figura 5. 17 - Exemplo de suspensão temporal de uma *thread***

A figura 5.18 apresenta o suporte necessário no metamodelo para a suspensão temporal de uma *thread*. A figura 5.19 apresenta também a classe de biblioteca *Thread*, com o respectivo método *sleep()* que encapsula o uso dos *statements* para suspensão temporal.

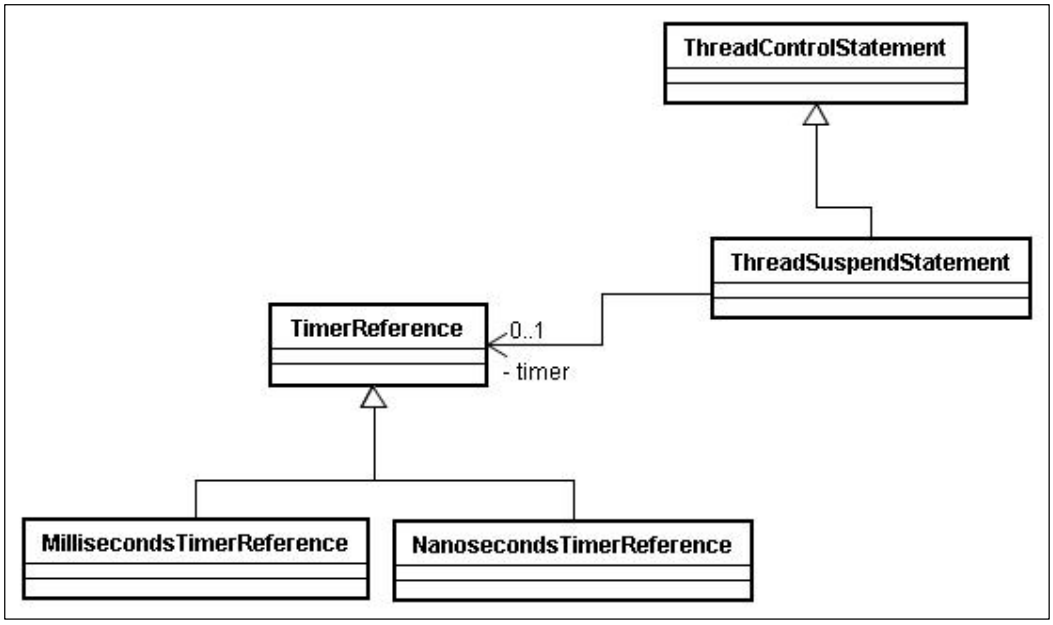


**Figura 5. 18 - Suspensão temporal de uma *Thread***

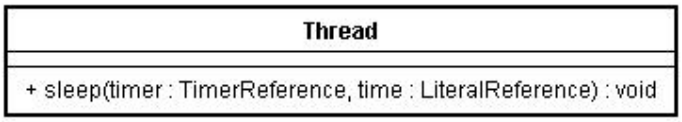


**Figura 5. 19 - Exemplo de classe de biblioteca para suspensão temporal de uma thread**

2. Tempo máximo de obtenção de uma trava: Uma segunda possibilidade de utilização de controle temporal é na obtenção de uma trava. Nesta forma de uso, uma *TimerReference* e um valor temporal na forma literal devem ser utilizados em conjunto com os *statements TryWriteLockStatement* e *TryReadLockStatement*. A figura 5.20 apresenta um exemplo de código fonte com o uso de obtenção de uma trava com um tempo máximo de espera. A tentativa de obtenção da trava acontece na linha 6, e caso a trava não fique disponível em 100 milisegundos, o método *tryWriteLock()* retorna uma referencia nula.

```
1
2   class LockTimedAcquisitionExampleClass {
3
4       method void example() {
5
6           Lock lock = anyLockableObject.tryWriteLock(Timer.makeMilisecondsTimer(100));
7
8           if(lock==null) {
9               //Non-synchronized code.
10          } else {
11              //WriteLocked code.
12              lock.release();
13          }
14      }
15
16  }
17
```

**Figura 5. 20 - Exemplo de aquisição de travas com tempo máximo de espera**

A figura 5.21 apresenta o suporte necessário no metamodelo para a aquisição de travas com tempo máximo de espera.

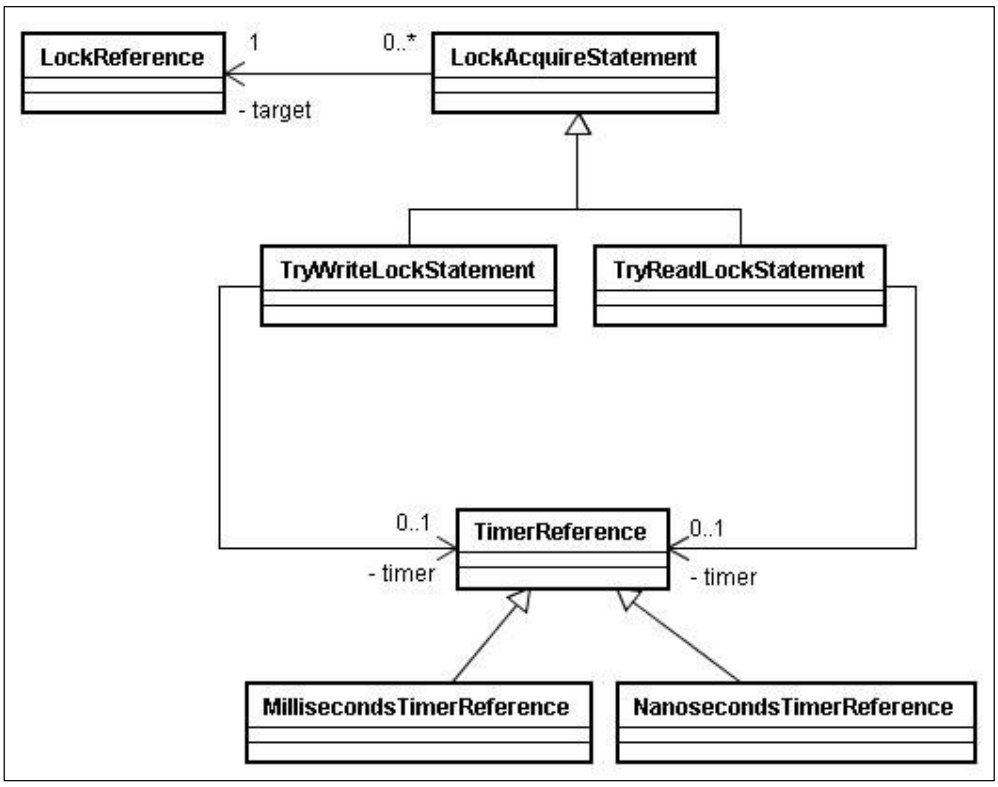


**Figura 5. 21 - Metamodelo - Aquisição de travas com tempo máximo de espera**

3. Tempo máximo de espera por uma condição de sincronização: A terceira possibilidade de utilização de controle temporal é na espera por satisfação de uma condição de sincronização. Nesta forma de uso, uma *TimerReference* e um valor temporal na forma literal devem ser utilizados em conjunto com o *statement AwaitStatement*. Caso expire o tempo máximo de espera, a *thread* que aguardava na condição é acordada como se alguma outra *thread* tivesse sinalizado através do método *signalAndContinue()*, porém, a condição fica marcada como expirada para a *thread*. A figura 5.22 apresenta um exemplo de código fonte com o uso de condição de sincronização com tempo máximo de espera. Na linha 12, a *thread* corrente é interrompida até que alguma outra *thread* notifique a satisfação da condição ou 5000 milisegundos se passem desde a invocação do método *await()*, o que ocorrer primeiro. Na linha 14, um teste é realizado para descobrir se a condição foi satisfeita ou o tempo de espera expirou.

```
1
2   lockable class TimedConditionExampleClass {
3
4       Condition cond;
5
6       constructor make() {
7           this.cond = this.newCondition();
8       }
9
10      method void example() {
11
12          this.cond.await(Timer.makeMilisecondsTimer(5000));
13
14          if(this.cond.isExpired()) {
15              //Nobody signaled the condition.
16          } else {
17              //Someone signaled the condition.
18          }
19      }
20
21      //...
22  }
23
```

**Figura 5. 22 - Exemplo de condições de sincronização com tempo máximo de espera**

A figura 5.23 apresenta o suporte necessário no metamodelo para condições de sincronização com tempo máximo de espera.

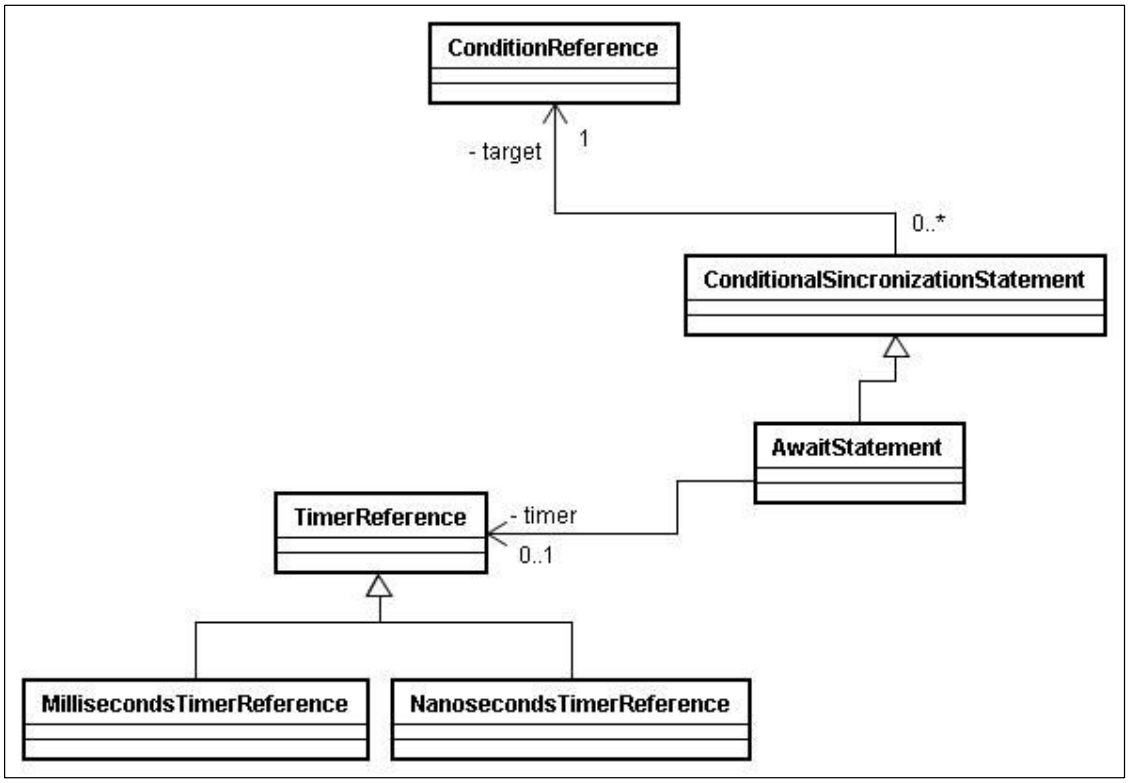


**Figura 5. 23 - Metamodelo - Condições de sincronização com tempo máximo de espera**

## *5.5 Considerações sobre os mecanismos de controle de concorrência desenvolvidos*

O modelo desenvolvido é abrangente pois contempla de forma coerente e homogênea os principais mecanismos de concorrência documentados na literatura e disponíveis nos sistemas e linguagens modernos.

Permite o uso de travas desde o mais baixo nível (aquisição e liberação explícita), com maior flexibilidade no uso, porém menos seguro, até o mais alto nível (classe do tipo

*Monitor*), onde os controles são implícitos e o uso é mais restrito, porém mais seguro. Um outro exemplo de flexibilidade é a possibilidade de extensão do modelo de controle de temporização para utilização com os controles de concorrência desenvolvidos.

O modelo desenvolvido introduz também o conceito de classe *Lockable*, na qual todo método deve ser especificado como *readLocked* ou *writeLocked*. Este recurso evita que objetos de classes que não tenham sido projetadas para uso concorrente sejam indevidamente travados e incorretamente acessados, possibilitando assim, a construção de sistemas concorrentes mais robustos.

O código de uma classe *Lockable* é ao mesmo tempo mais robusto e mais flexível que o código equivalente escrito na linguagem Java (utilizando-se da palavra reservada *synchronized*), pois em Java não há garantia de que um acesso deve ser obrigatoriamente sincronizado, como criticado em [HANSEN, 1999], e também Java não possibilita a sincronização do tipo múltiplos leitores e um único escritor com o *synchronized*, somente permite exclusão mútua.

# 6 Concorrência para jogos na Virtuosi

Este capítulo discute a adequação do trabalho desenvolvido às necessidades de desenvolvimento na área de jogos eletrônicos, alem de apresentar funcionalidades que podem simplificar o desenvolvimento de um jogo propriamente dito.

## *6.1 Controle da execução*

Um jogo eletrônico é um sistema complexo, alem disso, muitos dos jogos eletrônicos possuem características intrínsecas de sistemas de tempo real, e devido a estes fatos, o controle da execução das *threads* torna-se um requisito de grande importância para o desenvolvimento de um jogo eletrônico.

O suporte a assincronismo e controle de concorrência desenvolvidos permitem um controle de execução amplo por parte dos programadores de aplicações. Diversas decisões de projeto foram tomadas levando em consideração o requisito de controle de execução. Dentre estas decisões pode-se evidenciar:

- Modelo de *scheduling* cooperativo: O modelo de *scheduling* escolhido foi o modelo cooperativo (não preemptivo). Tal modelo permite que o controle de execução seja maior por parte do programador, pois a preempção de uma *thread* somente ocorre em situações de bloqueio ou por explícita decisão do programador, através do método de controle *yield()* ou *sleep()*, ambos da classe *Thread*.
- Suporte a travas leves: Travas leves são travas não bloqueantes, disponibilizadas através dos métodos *tryReadLock()* e *tryWriteLock()*. Possibilitam maior controle da execução, pois permitem que uma *thread* possa executar uma ação alternativa no caso de não aquisição de uma trava, ao invés de ficar bloqueada indefinidamente aguardando a liberação da mesma. O suporte desenvolvido também possibilita a tentativa de obtenção de travas com prazo máximo de espera.

- Uso de sinalização de condições de sincronização de forma preemptiva e não preemptiva: O mecanismo de sinalização de condições desenvolvido permite um controle maior do programador, pois permite a sinalização de forma preemptiva ou não, através dos métodos *signalAndWait()* e *signalAndContinue(),* respectivamente. O método *signalAndWait()* é preemptivo pois a *thread* notificadora libera suas travas e passa o controle da execução para a *thread* notificada, já o método *signalAndContinue()* é não preemptivo pois a *thread* notificadora continua sua execução e mantém suas travas adquiridas. Fica a critério do desenvolvedor da aplicação definir qual a melhor forma de notificação apropriada para o problema a ser resolvido.
- Controle por temporização: O mecanismo de controle desenvolvido permite controles temporais comuns em outras arquiteturas, como suspensão temporária da execução de uma *thread* (através do método *sleep()*, por exemplo), tentativa de obtenção de travas com prazo máximo de espera, e espera por uma condição de sincronização com prazo máximo de espera.
- Possibilidade de extensão: Uma grande vantagem do mecanismo desenvolvido com relação a outros mecanismos de controle é que o mesmo permite extensões, possibilitando a utilização de outras unidades de tempo, incluindo unidades de tempo discretas. Um exemplo desta vantagem é demonstrado na seção 6.3.

## 6.2 Determinismo

Os mesmos motivos que fazem o requisito *controle* importante também se aplicam ao requisito *determinismo*. Além destes requisitos, a comum necessidade de execução em plataformas heterogêneas reforça a necessidade de um ambiente na qual a execução seja controlada e determinística.

Pode-se afirmar que o suporte a concorrência desenvolvido é determinístico pelos seguintes fatos:

1. Ordenação nas t*hreads* em contenção: A escolha da próxima *thread* a executar é realizada através de uma fila de prioridades, onde o critério de desempate para *threads* de mesma prioridade é o tempo de espera para execução. Com

isto, uma *thread* que recém entrou na fila somente será eleita para execução após todas as *threads* de mesma prioridade que já estejam aguardando.

2. Ordenação na obtenção de travas: Quando uma *thread* tenta adquirir uma trava, ela entra numa fila de aquisição associada à trava (*entryQueue*), e caso uma segunda *thread* tente obter a mesma trava, será inserida ao final da mesma fila. Portanto uma *thread* nunca obterá uma trava antes das *threads* que já estejam na fila aguardando pela trava. É importante ressaltar que existe também uma outra fila de obtenção de travas que possui maior prioridade sobre a *entryQueue*, que é a *readyQueue*. A *readyQueue* é utilizada para *threads* que já obtiveram a trava em algum momento no passado, porém liberaram a trava para aguardar por uma condição de sincronização qualquer. Caso a condição seja satisfeita e notificada através do método *signalAndWait()*, a *thread* entra na *readyQueue*, e não na *entryQueue*, e com isto, obtém a trava antes das *threads* que aguardam na *entryQueue*.
3. Ordenação no uso de condições: Quando uma *thread* aguarda por uma condição de sincronização através do método *await()*, ela entra em uma fila de espera associada à condição. Novas *threads* que venham a aguardar a mesma condição entrarão no final da mesma fila.

## 6.3 API voltada a jogos eletrônicos

A possibilidade de extensão do modelo desenvolvido permite o desenvolvimento de funcionalidades interessantes para o desenvolvimento de jogos eletrônicos. Um possível exemplo de tal funcionalidade pode ser a construção de uma API para controle temporal que tem como base o número de atualizações de quadros de vídeo. Com isto, um programador de jogo poderia fazer facilmente com que uma *thread* aguardasse por 15 quadros de jogo, ou então, esperasse por uma condição por no máximo 20 quadros de jogo, e caso a condição não fosse satisfeita a tempo, retomasse a execução a tempo de manter uma boa qualidade na taxa de atualização de vídeo. A figura 6.1 apresenta o suporte no metamodelo para um timer baseado no número de atualizações de quadros de vídeo. A implementação deste timer deve

ser ligada diretamente com implementações de objetos de fronteira[18] responsáveis por vídeo. A cada quadro desenhado, o timer deve ser notificado.

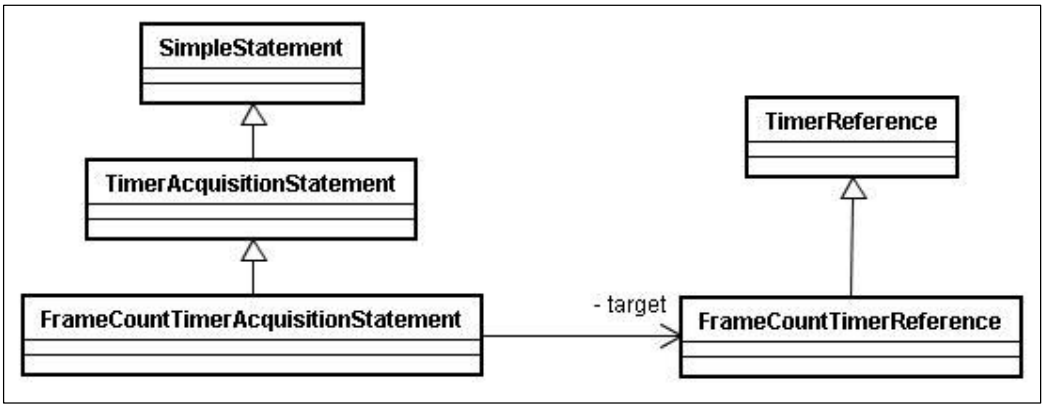


**Figura 6. 1 - Suporte a controle temporal baseado na atualização de quadros de vídeo**

Com este novo tipo de *timer* construções para verificações de tempo expirado, que seriam trabalhosas em outras arquiteturas, tornam-se triviais. Isto é demonstrado na figura 6.2. A figura mostra um possível cliente de um jogo eletrônico que depende de eventos de um servidor ou de outros clientes. Para tornar a experiência de jogo melhor, uma prática comum em jogos multiplayer no caso de os eventos não chegarem a tempo, é o jogo simular os eventos não recebidos. Na linha 15, pode-se observar o uso de sincronização condicional com tempo máximo de espera. Na linha 17 é realizado um teste para verificar se os eventos esperados chegaram a tempo.

Funcionalidades como a apresentada no exemplo da figura 6.2, se possíveis de serem implementadas de maneira simples como demonstrada, podem trazer grandes benefícios no desenvolvimento de jogos eletrônicos, pois simplificam significantemente o desenvolvimento.

[18] Objetos que fazem a interface com o hardware

```
1
2   monitor class FrameCountTimerExampleClass {
3
4       Condition eventsReceivedFromServer;
5
6       constructor make() {
7           this.eventsReceivedFromServer = this.newCondition();
8       }
9       //...
10
11      method void gameLoop() {
12
13          //Some game computation ...
14
15          this.eventsReceivedFromServer.await(Timer.makeFrameCountTimer(5));
16
17          if(this.eventsReceivedFromServer.isExpired()) {
18              this.simulateEvents(); //When they are predictable
19          } else {
20              this.treatRealEvents();
21          }
22
23          //Some game computation ...
24      }
25
26      method void simulateEvents() {
27          //The server did not respond in time.
28          //I will simulate the responses for a more realistic gaming experience.
29          //...
30      }
31      method void treatRealEvents() {
32          //The server respond in time.
33          //Let's check if the simulated events were correct.
34          //Let's treat the real events.
35          //...
36      }
37      //...
38  }
39
```

**Figura 6. 2 - Exemplo de tempo de espera baseado no número de quadros de jogo**

## 6.4 Exemplos de utilização de concorrência na área de jogos eletrônicos

Diversos podem ser os usos de concorrência em um jogo eletrônico. Os usos vão desde a parte de infra-estrutura do jogo (*engine*) (um exemplo é a subárea que trata *networking*, na qual a utilização de concorrência é praticamente inevitável pois depende de eventos assíncronos externos ao jogo). Outra possibilidade pode ser a utilização de concorrência na própria lógica de jogo.

Nesta seção serão exemplificadas algumas possíveis utilizações do modelo de concorrência desenvolvido neste trabalho.

A figura 6.3 exibe a implementação da lógica de jogo de um jogo chamado Jackpot, um jogo de sorte/azar que consiste em o usuário tentar obter a combinação de três números iguais.

```
1
2   class JackpotGame {
3
4       composition NumberGenerator seed1 = NumberGenerator.make();
5       composition NumberGenerator seed2 = NumberGenerator.make();
6       composition NumberGenerator seed3 = NumberGenerator.make();
7
8       constructor initGame() {
9           async(this.seed1.start());
10          async(this.seed2.start());
11          async(this.seed3.start());
12      }
13
14      method Boolean hit() {
15          async(Integer result1 = this.seed1.getResult()) {
16              async(Integer result2 = this.seed2.getResult()) {
17                  async(Integer result3 = this.seed3.getResult()) {
18                  }
19              }
20          }
21          return (result1 == result2 && result2 == result3);
22      }
23  }
```

**Figura 6. 3 - Exemplo de jogo - Jackpot**

A implementação do jogo faz uso de três geradores de números, sendo que os mesmos são inicializados de forma assíncrona nas linhas 9, 10 e 11. Quando o usuário decide tentar a sorte, o método *hit()* da linha 14 é invocado. O mesmo aguarda a finalização dos três geradores através de *rendezvous* e ao final retorna se os três resultados obtidos são iguais.

A figura 6.4 apresenta a implementação do gerador de números. O método *start* na linha 8 é um *loop* que fica em execução até que a variável *mustStop* torne-se verdadeira. Para inserir um indeterminismo no resultado, é utilizado um número aleatório para que a *thread* interrompa sua execução por alguns milisegundos (linha 16). Antes do retorno do método, o mesmo notifica que o resultado já pode ser utilizado (linha 18).

```
1
2   lockable class NumberGenerator {
3
4       Integer result;
5       Boolean mustStop;
6       Condition resultOk;
7
8       constructor make() {
9           this.result = Integer.make(0);
10          this.mustStop = Boolean.make("false");
11          this.resultOk = this.newCondition();
12      }
13
14      async writeLocked method void start() {
15          while(!mustStop) {
16              result.increment();
17
18              if(result >50) {
19                  result = Integer.make(0);
20              }
21
22              Thread.current().sleep(Ramdon.getIntegerBetween(1,50);
23          }
24          resultOk.signalAndContinue();
25      }
26
27      writeLocked method Integer getResult() {
28          this.mustStop = Boolean.make("true");
29          this.resultOk.wait();
30          return this.result;
31      }
32  }
```

**Figura 6. 4 - Gerador de números para Jackpot**

O método *getResult()*, na linha 21 indica que o gerador deve parar sua execução e aguarda através de uma condição de sincronização a finalização do resultado (linha 23), para então retornar à execução (linha 24).

O exemplo a seguir, figura 6.5, demonstra uma possível utilização do modelo desenvolvido para controlar eventos em um jogo multiplayer.

```
1
2   class GameServer {
3
4       EventManager eventManager;
5
6       constructor init() {
7           this.eventManager := EventManager.make();
8           this.gameLoop();
9       }
10
11      method void gameLoop() {
12          while(true) {
13              Event eventReceived = this.eventManager.nextEvent();
14              Event eventResult = this.treatEvent(eventReceived);
15              this.eventManager.dispatchEvent(eventResult);
16              // ...
17          }
18      }
19
20      method Event treatEvent(Event eventReceived) {
21          //Treat and validate the event received.
22          //Generate a response event and return
23      }
24
25  }
26
```

**Figura 6. 5 - *Loop* de um jogo multiplayer para tratar eventos de usuários**

O servidor do jogo possui um gerenciador de eventos (classe *EventManager*). O loop para tratamento de eventos consiste em recuperar um evento, tratá-lo e gerar um evento resultado para o cliente do jogo. Para o servidor, toda o tratamento de sincronização fica transparente pois este tratamento é encapsulado no gerenciador de eventos, exibido a seguir na figura 6.6.

```
1
2   lockable class EventManager {
3
4       BlockingQueue eventQueue;
5       Condition full;
6       Integer actualSize;
7       Integer allowedSize;
8       EventDispatcher dispatcher;
9
10      constructor make() {
11          eventQueue = BlockingQueue.make();
12          full = eventQueue.newCondition();
13          actualSize = Integer.make(0);
14          allowedSize = Integer.make(15);
15          dispatcher = EventDispatcher.make();
16      }
17
18      method Event nextEvent() {
19          writeLocked(this.eventQueue) {
20              Event event = this.eventQueue.pop();
21              this.full.signalAndContinue();
22              return event;
23          }
24      }
25
26      method void dispatchEvent(Event event) {
27          async { this.dispatcher.dispatchEvent(event) };
28      }
29
30      method void postEvent(Event event) {
31          writeLocked(this.eventQueue) {
32              if(actualSize.greaterOrEqual(allowedSize) {
33                  full.await();
34              }
35              this.eventQueue.push(event);
36          }
37      }
38  }
39
```

**Figura 6. 6 - Gerenciador de eventos**

O gerenciador de eventos disponibiliza o método *nextEvent()* (linha 18) que remove um evento da lista de eventos recebidos para que o mesmo seja tratado. Isso faz com que o gerenciador notifique que a fila de eventos não está cheia.

Além disso, despacha eventos de retorno encapsulando o assincronismo necessário nesta operação (linha 27).

O gerenciador de eventos também disponibiliza um método para recebimento de novos eventos por parte dos clientes (método *postEvent()* na linha 30). Este método sincroniza escrita na fila de eventos recebidos (linha 31) e bloqueia a execução caso a fila esteja cheia (linha 33).

A figura 6.7 demonstra a interface da classe *EventDispatcher* utilizada no gerenciador de eventos. O método para despacho de eventos força que seja tratado o assincronismo referente a esta operação (linha 4).

```
1
2   class EventDispatcher {
3
4       async method void dispatchEvent(Event event) {
5           //Dispatch the event to its destination (one of the game clients).
6       }
7
8   }
```

**Figura 6. 7 - Interface da classe *EventDispatcher***

Finalmente, como mostrado neste capítulo, pode-se afirmar que o modelo de concorrência desenvolvido é adequado para o desenvolvimento de jogos eletrônicos, pois supre de forma coesa e abrangente requisitos necessários para a utilização de concorrência no desenvolvimento de jogos, como por exemplo: controle de execução, determinismo, API voltada a jogos e simplicidade de utilização.

# 7 Conclusão

A principal contribuição cientifica deste trabalho foi a definição de um modelo de concorrência abrangente, robusto, flexível, extensível e simples para a Virtuosi. Este modelo é um modelo mais adequado para as necessidades de concorrência da área de jogos eletrônicos que os modelos oferecidos pelos sistemas atualmente utilizados, incluindo linguagens de script, como Lua, linguagens específicas para concorrência, como SR e JR, sistemas de grande escala, como Java, .NET e Python, sistemas que implementam o modelo proposto pelo padrão CORBA, e também variações desse padrão que se propõem a ser mais avançadas, tal como Ice. Essa melhor adequação verifica-se pelos seguintes aspectos:

- Abrangência: O modelo contempla de forma coerente e homogênea os principais mecanismos de concorrência documentados na literatura e disponíveis nos sistemas e linguagens modernos. Inclusive, foram incorporadas duas características que não são normalmente encontradas em outros sistemas, nem mesmo em linguagens específicas para concorrência, como SR[19] e JR:
  - Sinalização preemptiva (SW) e não preemptiva (SC).
  - Travas para leitura e travas para escrita.
- Robustez: O modelo exige a especificação explícita de classes concorrentes, utilizando um dos seguintes tipos:
  - *Monitor*: Implementação clássica do conceito de monitor (*single reader*, *single writer*).
  - *Lockable*: Classes que podem sofrer acesso concorrente, com um controle mais fino (*multiple readers*, *single writer*).

  Instâncias de classes não especificadas dessa forma não podem ser alvos de travas. Esta garantia de utilização no controle de concorrência somente classes que foram projetadas para tal é uma vantagem deste modelo sobre os outros. Este ponto é abordado por Hansen em [HANSEN, 1999].
- Flexibilidade: Uso dos mecanismos de assincronismo e de controle de concorrência de forma implícita ou explícita, possibilitando menor ou maior

[19] SR possui construções não implementadas no modelo desenvolvido, com exemplo as construções *co* e *reply*.

grau de flexibilidade, respectivamente. Esta possibilidade não é apresentada de forma tão completa nos outros modelos.

- Extensibilidade: O modelo contempla a adição de novas formas de temporização, por exemplo, o mecanismo de controle baseado na taxa de atualização de vídeo.
- Simplicidade de uso: Os recursos do modelo estão formalizados no metamodelo Virtuosi, o que permite a sua representação na forma de linguagem de programação, além disso, as construções sintáticas desenvolvidas seguem alguns dos padrões comumente utilizados nas diversas linguagens concorrentes estudadas.

A constatação dos benefícios decorrentes do modelo desenvolvido com este trabalho de pesquisa, mesmo sendo um trabalho inicial, motiva a sua continuidade em diversas direções tais como:

- Finalização da implementação do modelo de concorrência na máquina virtual Virtuosi. Estima-se que serão necessários aproximadamente três meses para a conclusão desta tarefa.
- A implementação de uma *engine* de jogos eletrônicos.
- A investigação da viabilidade de adição de um scheduler preemptivo ao modelo cooperativo desenvolvido. Um exemplo seria a utilização de uma política cooperativa para *threads* de mesma prioridade e preemptiva para *threads* de diferentes prioridades. Outra possibilidade seria utilizar a forma de preempção utilizada em JR, baseada em instruções executadas.
- Pesquisar a necessidade de aumento da granularidade das travas desenvolvidas, e com isto uma possível introdução do modelo de *Intention locks* encontrado em CORBA.
- O desenvolvimento de um mecanismo de detecção de *deadlocks*.
- A definição de um modelo de herança para classes concorrentes.
- O desenvolvimento de um mecanismo de transação local e transação distribuída.
- Estudo de um modelo que permita a composição de *Timers* para a construção de mecanismos de temporização mais complexos.

Portanto, o presente trabalho prestou uma contribuição para o desenvolvimento de um modelo de concorrência na Virtuosi, e com isto abriu uma série de possibilidades de pesquisas e avanços nesta área.

# Apêndice A – Comparação com Java

# Apêndice A - Comparação com Java

Este apêndice mostra algumas das funcionalidades desenvolvidas neste trabalho comparando implementações concorrentes na arquitetura Virtuosi com implementações na arquitetura Java.

A escolha da arquitetura Java para as comparações deve-se ao fato da mesma oferecer um bom suporte a concorrência e também da linguagem *Aram* possuir uma sintaxe *Java like*, o que facilita a comparação.

## *A.1 - Assincronismo*

A figura A.1 apresenta um exemplo de assincronismo na Virtuosi.

```
 1
 2   class Client {
 3
 4       method void implicitAsync() {
 5           //...
 6
 7           SomeClass targetObject = SomeClass.make();
 8
 9           async( targetObject.someVoidMethod() );
10
11           //...
12       }
13   }
14
```

**Figura A. 1 - Exemplo de assincronismo na Virtuosi**

A figura A.2 apresenta um exemplo de assincronismo em Java.

```
1
2   class TempClass implements Runnable {
3       public void run() {
4           SomeClass targetObject = new SomeClass();
5           targetObject.someMethod();
6       }
7   }
8
9   public class Client {
10
11      void async() {
12          //...
13
14          TempClass runnable = new TemClass();
15          Thread t = new Thread(runnable);
16
17          t.start();
18
19          //...
20      }
21  }
22
```

**Figura A. 2 - Exemplo de assincronismo em Java**

## *A.2 - Rendezvous*

A figura A.3 apresenta um exemplo de *Rendezvous* na Virtuosi.

```
1
2   class RendezvousInTheLanguageExampleClass {
3
4       method void someMethod() {
5
6           SomeClass someObject;
7
8           async( someObject = SomeClass.make(); ) {
9               // Do something until the "someObject" gets ready.
10              // Here the use of "someObject" is not allowed.
11              //...
12          }
13
14          //Now I can use the "someObject".
15          someObject.doSomething();
16      }
17  }
18
```

**Figura A. 3 - Exemplo de *Rendezvous* na Virtuosi**

A figura A.4 apresenta um exemplo de *Rendezvous* em Java.

```
1
2   class TempClass implements Runnable {
3
4       public SomeClass someObject;
5       public void run() {
6           this.someObject = new SomeClass();
7       }
8   }
9
10  public class RendezvousInJava {
11
12      void async() {
13          SomeClass someObject;
14
15          TempClass runnable = new TemClass();
16          Thread t = new Thread(runnable);
17
18          t.start();
19
20          // Do something until the "someObject" gets ready.
21          // Here the use of "someObject" will cause a NullPointerException.
22          //...
23
24          t.join();
25          someObject = runnable.someObject;
26          someObject.doSomething();
27      }
28  }
29
```

**Figura A. 4 - Exemplo de *Rendezvous* em Java**

## *A.3 - Controle de concorrência com granularidade de leitura e escrita*

A figura A.5 apresenta um exemplo de controle de concorrência na Virtuosi, com a utilização de acesso de leitura e escrita.

```
1
2   lockable class MethodQualifierLockExampleClass {
3
4       //...
5
6       writeLocked method void someMethod1() {
7           //This is a writeLocked method.
8       }
9       readLocked method void someMethod2() {
10          //This is a readLocked method.
11      }
12  }
13
```

**Figura A. 5 - Exemplo de controle com leitura e escrita na Virtuosi**

A figura A.6 apresenta um exemplo de controle de concorrência com a utilização de acesso de leitura e escrita em Java.

```
1
2   class ReadWriteLockExampleClass {
3
4       //...
5
6       void someMethod1() {
7           ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
8           Locl l = rwLock.WriteLock;
9           l.lock();
10          //This is a writeLocked method.
11          l.release();
12      }
13
14      void someMethod2() {
15          ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
16          Locl l = rwLock.ReadLock;
17          l.lock();
18          //This is a readLocked method.
19          l.release();
20      }
21  }
22
```

**Figura A. 6 - Exemplo de controle com leitura e escrita em Java**

## *A.4 - Monitor*

A figura A.7 apresenta um exemplo de monitor na Virtuosi.

```
1
2   monitor class MonitorExampleClass {
3
4       constructor make() {
5       }
6
7       method void someMethod1() {
8           //This is a writeLocked method.
9       }
10
11      method String getSomeAtribute() {
12          //This is a writeLocked method.
13          return this.someAtribute;
14      }
15  }
16
```

**Figura A. 7 - Exemplo de classe concorrente do tipo Monitor**

A figura A.8 apresenta uma versão de uma classe monitor em Java.

```
1
2   class MonitorExampleClass {
3
4       MonitorExampleClass() {
5       }
6
7       synchronized someMethod1() {
8           //This is a writeLocked method.
9       }
10
11      synchronized String getSomeAtribute() {
12          //This is a writeLocked method.
13          return this.someAtribute;
14      }
15  }
16
```

**Figura A. 8 - Exemplo de classe concorrente do tipo Monitor em Java**

## *A.5 - Considerações sobre as comparações*

É visível nos exemplos apresentados que o suporte a concorrência desenvolvido neste trabalho proporciona maior simplicidade e robustez no desenvolvimento de aplicações concorrentes. Deve ser ressaltado ainda que diversas funcionalidades não foram comparadas com uma versão em Java pelo fato de Java não oferecer suporte para as mesmas, como exemplo pode-se listar as seguintes funcionalidades: definição de assincronismo no lado servidor, temporização baseada em unidades discretas e sinalização preemptiva de condição de sincronização.

# Apêndice B – Formalização das construções sintáticas da linguagem Aram

# 8 Referências bibliográficas

[ANDREWS, 1981] - ANDREWS, Gregory R. (1981). Synchronizing Resources. ACM Trans. Program. Lang. Syst. vol.3, n.4 p.405-430, Oct. 1981 DOI= http://doi.acm.org/10.1145/357146.357149.

[ANDREWS et al., 1988] - ANDREWS Gregory R.; et al. (1988). An overview of the SR language and implementation. ACM Trans. Program. Lang. Syst. vol.10, n.1, p.51-86, Jan. 1988. DOI= http://doi.acm.org/10.1145/42192.42324.

[ANDREWS, 2000] - ANDREWS, Gregory R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley, 2000. 664 p.

[BECKER, 2001] - BECKER, Katrin (2001). Teaching with games: the Minesweeper and Asteroids experience. J. Comput. Small Coll. USA, Consortium for Computing Sciences in Colleges. vol.17, n.2, 23-33, Dez. 2001.

[BLOW, 2004] - BLOW, Jonathan (2004). Game Development: Harder Than You Think. Queue. NY, USA. ACM Press. vol.1, n.10, p.28-37, Fev. 2004. DOI= http://doi.acm.org/10.1145/971564.971590.

[BORGES, 2004] BORGES, Aron (2004). Uma linguagem de programação orientada a objetos mínima. Projeto final II, Pontifícia Universidade Católica do Paraná, Curitiba, Brasil. 50p. in Portuguese.

[BRIOT, GUERRAOUI, LOHR, 1998] - BRIOT, Jean Pierre; GUERRAOUI, Rachid e LOHR, Klaus Peter (1998). Concurrency and distribution in object-oriented programming. NY, USA. ACM Comput. Surv. vol.30, n.3, p.291-329. Set. 1998. DOI= http://doi.acm.org/10.1145/292469.292470

[CALSAVARA, 2000] - CALSAVARA, A. (2000). Virtuosi: Máquinas virtuais para objetos distribuídos. Technical report approved on internal examination for career ascencion, Pontifícia Universidade Católica do Paraná, Curitiba, Brasil. 99 p. in Portuguese.

[CESAR FILHO, 2004] CESAR FILHO, Juarez da Costa (2004). Mecanismo de mobilidade de objetos para a virtuosi. Master's thesis, Pontifícia Universidade Católica do Paraná, Curitiba, Brasil. 163p. in Portuguese.

[CORBA, 2000] CORBA - Concurrency Service Specification, versão 1.0. Disponível em: http://www.omg.org/technology/documents/formal/concurrency_service.htm. Acesso em: 31 de Julho de 2005.

[DIJKSTRA, 1965] - DIJKSTRA, Edsger. W. (1965). Solution of a problem in concurrent programming control. Communications of the ACM. vol.8, n.9. Set. 1965.

[DIJKSTRA , 1968] - DIJKSTRA, Edsger. W. (1968). The structure of the “THE” multiprogramming system. Communications of the ACM. vol.11, n.5. p.341-46. Maio 1968

[ESA, 2005] - THE ENTERTAINMENT SOFTWERE ASSOCIATION - ESA (2005). 2005 ESSENTIAL FACTS ABOUT THE COMPUTER AND VIDEO GAME INDUSTRY. Disponível em: http://www.theesa.com/files/2005EssentialFacts.pdf. Acesso em: 31 de Julho de 2005.

[FOWLER, 1999] - FOWLER, Martin (1999). Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc. 1999.

[GAMMA et al., 1995] - GAMMA, Erich; et al. (1995). Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. 1995.

[GOLDBERG e ROBSON, 1983] - GOLDBERG, A. e ROBSON, D. (1983). Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc. 1983.

[GOSLING et al., 2005] - GOSLING, James; et al. (2005). The Java Language Specification. Third Edition. Disponível em: http://java.sun.com/docs/books/jls/third_edition/html/memory.html. Acesso em: 31 de Julho de 2005.

[GUZDIAL and SOLOWAY, 2002] - Guzdial, M. and Soloway, E. (2002). Teaching the Nintendo generation to program. Communications of ACM. NY, USA. ACM Press. vol.45, n.4, p.17-21. Abr. 2002. DOI= http://doi.acm.org/10.1145/505248.505261

[HANSEN, 1999] - HANSEN, Per Brinch (1999). Java's insecure parallelism. SIGPLAN Not. 34,4 (Abr. 1999), 38-45. DOI=http://doi.acm.org/10.1145/312009.312034

[HENNING, 2004] - HENNING, M. (2004). A New Approach to Object-Oriented Middleware. IEEE Internet Computing. NJ, USA. vol.8, n.1, p66-75. Jan. 2004. DOI= http://dx.doi.org/10.1109/MIC.2004.1260706

[HENNING e SPRUIELL, 2005] - HENNING, Michi e SPRUIELL, Mark (2005). Distributed Programming with Ice. ZeroC, Inc. rev.2.1.2,28 1464p. Disponível em: http://www.zeroc.com/download/Ice/2.1/Ice-2.1.2.pdf. Acesso em: 31 de Julho de 2005.

[IERUSALIMSCHY, FIGUEIREDO, CELES, 2005] - IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique de; CELES, Waldemar (2005). The Implementation of Lua 5.0 - Proceedings of SBLP 2005, pp. 63-75. 2005.

[KEEN et al., 2004] - KEEN, Aaron W.; et al. (2004). JR: Flexible distributed programming in an extended Java. ACM Trans. Program. Lang. Syst. (TOPLAS), vol.26, n.3, p.578-608. May. 2004. DOI= http://doi.acm.org/10.1145/982158.982162

[KOLB, 2004] - KOLB, Carlos José (2004). Um Sistema de Execução para Software Orientado a Objeto Baseado em Árvores de Programa. Master's thesis, Pontifícia Universidade Católica do Paraná, Curitiba, Brasil. 203p. in Portuguese.

[MAES, 1987] - MAES, P. (1987). Concepts and experiments in computational reflection. In Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications. N. Meyrowitz, Ed. OOPSLA '87. ACM Press, NY, USA. p.147-155. DOI= http://doi.acm.org/10.1145/38765.38821

[MATLOFF and HSU, 2005] - MATLOFF, Norman and HSU, Francis (2005). Introduction to Threads Programming with Python. University of California. Disponível em: http://heather.cs.ucdavis.edu/~matloff/Python/PyThreads.pdf. Acesso em: 19 de Novembro de 2005.

[MEYER, 1992] - MEYER, Bertrand (1992). Eiffel: the language. Prentice-Hall, Inc. 1992.

[MEYER, 1997] - MEYER, Bertrand (1997). Object-Oriented Software Construction. Prentice Hall PTR, second edition. 1997.

[MICROSOFT, 2001] - MICROSOFT CORPORATION (2001). Microsoft C# Language Specifications. Microsoft Press. 2001.

[NUNES e CALSAVARA, 2001] NUNES, Leonardo R. e CALSAVARA, Alcides (2001). Estudos sobre a concepção de uma linguagem de programação reflexiva e correspondente ambiente de execução. In: Simpósio Brasileiro de Linguagens de Programação – SBLP. 2001. Curitiba. P.C193-C204. in Portuguese.

[OLSSON et al., 1992] - OLSSON Ronald A.; et al. (1992). SR: A Language for Parallel and Distributed Programming. TR 92-09 - Department of Computer Science - The University of Arizona. Mar. 1992. Disponível em: http://www.cs.arizona.edu/sr/language.pdf Acesso em: 19 de Novembro de 2005.

[STÄRK e BÖRGER, 2004] STÄRK, Robert F. e BÖRGER. Egon (2004). An ASM specification of C# threads and the .NET memory model. In: B. Thalheim and W. Zimmermann, editores, Abstract State Machines, 11th International Workshop, ASM 04, Halle-Wittenberg, Germany. 2004.

[STROUSTRUP, 1986] - STROUSTRUP, B. 1986 The C++ programming language. Addison-Wesley Longman Publishing Co., Inc. 1986.

[WATANABE e YONEZAWA, 1989] - WATANABE, T. and YONEZAWA, A. (1989). Reflective computation in object-oriented concurrent systems and its applications. In Proceedings of the 5th international Workshop on Software Specification and Design. IWSSD '89. ACM Press, NY, USA. p.56-58. DOI= http://doi.acm.org/10.1145/75199.75209

[WATT, 1991] - WATT, D. A. (1991). Programming Language Syntaxe and Semantics. University of Glasgow, UK.

[WIKIPEDIA, 1] - Co-operative multitasking. in: Wikipedia, the free encyclopedia. Disponível em: http://en.wikipedia.org/wiki/Co-operative_multitasking. Acesso em: 31 de Julho de 2005.

[WIKIPEDIA, 2] - Pre-emptive multitasking. in: Wikipedia, the free encyclopedia. Disponível em: http://en.wikipedia.org/wiki/Pre-emptive_multitasking. Acesso em: 31 de Julho de 2005.

# Anexo A – O Metamodelo Virtuosi

# Anexo B – A linguagem Aram