UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Estudo e Implementação do Modelo Reflexivo Tempo Real RTR sobre a Linguagem Java

Dissertação submetida à Universidade Federal de Santa Catarina
para obtenção do grau de
Mestre em Engenharia Elétrica

Danielle Nishida

Florianópolis, 16 de Dezembro de 1996.

Estudo e Implementação do Modelo Reflexivo Tempo Real RTR

sobre a Linguagem Java

Danielle Nishida

Esta dissertação foi julgada para obtenção do título de
Mestre em Engenharia
especialidade **Engenharia Elétrica,**
área de concentração **Controle, Automação e Informática Industrial,**
e aprovada em sua forma final pelo Curso de Pós-Graduação.

Prof. Joni da Silva Fraga, Dr.
Orientador

Prof. Jean-Marie Farines, Dr. Ing.
Co-Orientador

Prof. Adroaldo Raizer, Dr.
Coordenador

Banca Examinadora:

Prof. Carlos Alberto Maziero, Dr.

Prof. Olinto José Varela Furtado, M Sc.

Roberto Willrich, Dr.

Ao meu pai, Lintaro Nishida,
por seu carinho, apoio e,
principalmente, por seu exemplo.

AGRADECIMENTOS

Aos Profs. Joni da Silva Fraga e Jean-Marie Farines pela confiança, dedicação e orientação.

Aos membros da banca examinadora, pelas críticas, sugestões e pelo incentivo, em especial ao Prof. Olinto Varela Furtado pela amizade, dedicação e empenho, sem os quais este trabalho não teria se realizado.

A CAPES pelo suporte financeiro.

Agradeço ao Prof. Ubiratan Holanda Bezerra pela confiança e por ter despertado em mim o gosto pela pesquisa.

Ao Sr. Edvaldo Machado, D. Dair, Júnior e família, pelo carinho e amizade a mim dedicados durante estes anos de mestrado.

A todos os colegas do LCMI e aos amigos Frank Siqueira e Lau Cheuk Lung pelo apoio e pelas sugestões em relação ao trabalho. Especialmente, a André Leal, Fernanda Paiva dos Santos, Marcelo Maciel, Max Mauro Santos e Ricardo Martins pela amizade e paciência.

Agradeço aos meus pais Lintaro e Terezinha Nishida, e irmãs Waleska e Alexsandra Nishida pelo amor, apoio e incentivo que me possibilitaram concluir este trabalho. Ao meu namorado Emerson Raposo, pelo carinho, paciência e companheirismo em todos os momentos, inclusive nos mais difíceis.

A Deus e ao meu anjo da guarda, por tudo.

RESUMO

Este trabalho situa-se na área dos Sistemas Tempo Real e baseia-se no Modelo Reflexivo para Tempo Real RTR [Fur95].

O modelo RTR estabelece uma filosofia para o desenvolvimento de aplicações tempo real, facilitando a implementação de ferramentas de desenvolvimento e de aplicações sujeitas a restrições temporais. Com este intuito, o modelo RTR alia os paradigmas de orientação a objetos e reflexão computacional, incorporando, desta forma, características tais como facilidade de manutenção, capacidade de reutilização e um alto nível de organização interna do sistema.

O principal objetivo deste trabalho é validar o Modelo RTR. Neste sentido, é feito um estudo detalhado do mesmo a fim de mapeá-lo sobre a linguagem de programação Java. A seguir, através da implementação de um protótipo do modelo sobre a referida linguagem, são realizados vários testes envolvendo aplicações com restrições temporais e de sincronização, procurando-se verificar aspectos como a coerência lógica e temporal, flexibilidade e a capacidade do modelo em expressar restrições temporais.

ABSTRACT

This work is focused on the area of real time systems and is based on the Real-Time Reflective Model - RTR [Fur95].

The RTR model establish a philosophy for development of real-time applications, making easy the implementation of development tools and applications with timing constraints. Following this idea, the RTR model join the object-orientation and reflective computation paradigms, adding some caracteristics like maintainability, reusability and high level internal organization of the system.

The aim of this work is the validation of the RTR model. By this way, a detailed study of model is done with the aim of mapping it onto the Java programming language. After it, through the prototype implementation of the model over the Java language, several tests are done, including applications with synchronization and timing constraints, trying to verify aspects like logical and temporal coherence, flexibility and the model capability to express timing constraints.

ÍNDICE

ÍNDICE DE FIGURAS

Capítulo 1

Introdução

Diferentes dos sistemas computacionais de propósito geral, os Sistemas Tempo Real devem suportar restrições temporais impostas pelo ambiente. Com o número crescente destas aplicações, cada vez maiores e mais complexas, é necessário que pesquisas específicas sejam dedicadas a este domínio.

Assim, muitos trabalhos vêm sendo realizados na área de Sistemas Tempo Real, a fim de desenvolver tecnologias que sejam capazes de tratar de forma coerente questões como a correção temporal. Com este fim, têm sido propostas arquiteturas, linguagens de programação, sistemas operacionais e metodologias de desenvolvimento.

Este trabalho baseia-se no Modelo Reflexivo para Tempo Real (Modelo RTR) [Fur95]. Desenvolvido no Laboratório de Controle e MicroInformática (LCMI), o Modelo RTR tem como objetivo estabelecer uma filosofia para o desenvolvimento de aplicações voltadas para o domínio tempo real, procurando reduzir problemas como o gerenciamento de complexidade e a falta de flexibilidade na representação de aspectos temporais. Com este intuito, o modelo alia os paradigmas de orientação a objetos e reflexão computacional.

A orientação a objetos confere ao modelo características como modularidade, capacidade de reutilização e facilidade de manutenção. A reflexão computacional, pór sua vez, possibilita a uma aplicação o controle sobre seu próprio comportamento, através da separação entre suas atividades funcionais e de gerenciamento, contribuindo diretamente

para uma maior organização interna do sistema e para o aumento da modularidade e flexibilidade.

Devido as suas características, o modelo mostra-se bastante adequado à programação de aplicações tempo real que sigam a abordagem *best effort*. Dentre estas aplicações, encontram-se os sistemas multimídia [Fra95].

Baseando-se nestas premissas, os objetivos deste trabalho são:

- O estudo do Modelo RTR, a fim de identificar suas principais características e sua dinâmica de funcionamento;

- O mapeamento do Modelo RTR sobre a linguagem de programação Java, com o objetivo de identificar as potencialidades da linguagem para a representação do modelo;

- A implementação do Modelo RTR na linguagem Java, a fim de validar o modelo através do teste de algumas de suas funcionalidades.

A linguagem Java, um produto Sun MicroSystems Inc., é uma linguagem de programação orientada a objetos, de propósito geral, para distribuição em redes heterogêneas. Um programa escrito em Java é compilado para um arquivo em código binário que pode executar em qualquer máquina na qual a plataforma Java esteja presente. Além disso, características como simplicidade, robustez e concorrência incluíram Java entre os objetos de estudo desta dissertação.

Os resultados deste trabalho visam contribuir para a avaliação do Modelo RTR, fornecendo parâmetros para a especificação da Linguagem Java/RTR [Fur96]. Esta linguagem é uma extensão de Java, realizada a partir da filosofia de programação ditada pelo Modelo RTR.

Os capítulos que compõem esta dissertação estão estruturados da seguinte maneira:

Capítulo 2: Este capítulo aborda o paradigma da reflexão computacional, apresentando como exemplo, a linguagem Open C++ [Chi93a] - que implementa um protocolo de meta-objetos; em seguida, é feito um breve resumo a respeito dos sistemas tempo real, apresentando suas definições básicas. Por fim, são apresentados os modelos para programação de aplicações tempo real RTC++ [Ish90], RTO.k [Kim94], DRO [Tak92], RT Java [Nil95] e R2 [Hon94]. As informações apresentadas neste capítulo objetivam definir o contexto no qual este trabalho foi desenvolvido.

Capítulo 3: Apresenta o Modelo RTR, suas principais características, estrutura e dinâmica de funcionamento.

Capítulo 4: Apresenta as características básicas da linguagem Java e o mapeamento do Modelo RTR sobre a mesma, detalhando a dinâmica resultante deste mapeamento.

Capítulo 5: Apresenta o protótipo implementado, fazendo uma descrição a respeito de soluções adotadas para problemas específicos de implementação. Este capítulo mostra, ainda, os testes realizados sobre este protótipo, além da especificação do mesmo.

Capítulo 6: Este capítulo reúne as conclusões a respeito do trabalho de dissertação desenvolvido, além de apresentar sugestões para a continuidade das pesquisas neste contexto.

Capítulo 2

Contexto de Trabalho

Este capítulo tem como objetivo delinear o contexto sobre o qual este trabalho foi estruturado, definindo conceitos e técnicas que nortearam seu desenvolvimento. *A priori,* apresentaremos o paradigma da reflexão computacional, citando como exemplo a linguagem Open C++, que permite a implementação de aplicações segundo a técnica reflexiva. Em seguida, faremos uma breve introdução aos Sistemas de Tempo Real e analisaremos alguns dos modelos de programação propostos na literatura para o tratamento de problemas específicos a este domínio.

2.1. Reflexão Computacional

Entende-se por reflexão computacional a atividade realizada por um sistema computacional quando esse controla e atua sobre si mesmo [Mae87]. Para que isso seja possível, os sistemas reflexivos devem possuir um conjunto de estruturas que representem seus próprios aspectos, tanto estruturais quanto computacionais. Este conjunto é denominado de auto-representação do sistema.

A técnica de reflexão computacional vem sendo largamente utilizada com o intuito de obter-se um maior grau de modularidade em aplicações, imprimindo aos sistemas computacionais maior capacidade de gerenciamento, maior legibilidade e facilidade de manutenção. Alguns exemplos da aplicação de técnicas reflexivas são encontrados nas áreas de tolerância a faltas [Fab95], concorrência [Yon88] e tempo real [Hon94].

Um sistema computacional com arquitetura reflexiva, segundo a abordagem de meta-objetos [Mae87] deve ser constituído por um nível base e um nível meta. O nível base deve ser responsável pela resolução de problemas pertencentes a um domínio externo, enquanto o nível meta deve ser responsável pelo gerenciamento do nível base. Desta forma, podem ser separadas as atividades funcionais e não-funcionais do sistema em questão.

Parte-se do princípio que, para cada objeto x pertencente ao sistema, é associado um meta-objeto ^x, que possui a auto representação de x. Como o objeto x deve estar conectado a ^x de forma causal, podemos admitir que quaisquer mudanças ocorridas em ^x deverão ser refletidas em x. Assim, podemos controlar os procedimentos de x através de manipulações em ^x.

De forma simplificada, pode-se dizer que uma chamada a um método de x será desviada ao seu meta-objeto correspondente ^x, que será responsável por todos os procedimentos relacionados ao gerenciamento da execução do método solicitado. O método do objeto x será ativado, segundo este gerenciamento, a partir de ^x.

Quando o paradigma da reflexão computacional é adotado na implementação de um sistema, deve ser possível ao usuário utilizar o controle a nível meta ou, simplesmente, utilizar os objetos-base independentes de seus meta-objetos, se isto for conveniente.

A reflexão computacional, porém, não auxilia na resolução de problemas pertencentes a um domínio externo. Seu propósito é contribuir para uma maior organização interna do sistema, garantindo o funcionamento desejado dos objetos a nível base.

Além de ser utilizada na implementação de sistemas para atribuir-lhes maior capacidade de gerenciamento, flexibilidade e modularidade, a reflexão computacional é utilizada na extensão de linguagens de programação. De modo geral, elas são desenvolvidas com procedimentos bem definidos, o que as torna, por vezes, inflexíveis e impróprias às necessidades dos programadores.

Com a disseminação da técnica de reflexão computacional, alguns protocolos de meta-objeto têm sido propostos para linguagens de programação. O objetivo destes protocolos é permitir aos usuários modificar os procedimentos e a implementação destas linguagens, a fim de que elas possam ser adequadas as suas necessidades específicas.

Uma linguagem baseada num protocolo de meta-objetos é, por si só, um programa orientado a objetos. Assim, características como flexibilidade e capacidade de manutenção são também atribuídas à linguagem.

Em sistemas tempo real, a adoção da técnica de reflexão computacional é adequada por possibilitar extensões de linguagem que permitam a especificação de novas restrições temporais, segundo uma aplicação específica, além de contribuir para a flexibilidade do sistema permitindo a alteração do algoritmo de escalonamento quando necessário e possibilitar mudanças de procedimento quando o sistema detecta que alguma tarefa não pode cumprir suas restrições temporais.

A seção seguinte abordará um exemplo da aplicação da técnica reflexiva na extensão de linguagens de programação.

2.1.1. Open C++

Open C++ [Chi93a] é uma variação da linguagem de programação C++, contendo um protocolo de meta-objetos (MOP). Como visto no item 2.1, protocolos de meta-objetos permitem que extensões sejam feitas às linguagens que os suportam, sem que seja necessário, para isso, reconstruir seu compilador. Os programadores podem, então, adequar a linguagem de programação as suas necessidades específicas.

No Open C++, tanto as chamadas a métodos quanto o acesso às variáveis são extensíveis. A reflexão é individual, ou seja, cada objeto-base possui um meta-objeto correspondente. Quando uma chamada a um método de um objeto-base é realizada, um método específico de seu meta-objeto é invocado, e não a implementação padrão que deveria

ter sido compilada. O meta-objeto é executado e se encarrega da invocação do método do objeto-base.

O usuário poderá modificar a implementação de uma chamada a método implementando outros meta-objetos e substituindo-os. No Open C++, os detalhes inerentes à implementação do protocolo de meta-objetos são ocultados do usuário, de modo a tornar a extensão de uma chamada a método simples e acessível.

Em Open C++, podem ser criados objetos reflexivos e não reflexivos. Os objetos reflexivos são controlados por meta-objetos, os não reflexivos não têm meta-objetos e são executados como objetos normais de C++.

Os objetos de uma classe reflexiva podem ou não ser reflexivos, conforme selecionado pelo usuário, um objeto declarado como um objeto não-reflexivo não poderá tornar-se reflexivo posteriormente. Da mesma forma, os métodos e as variáveis de um objeto reflexivo só serão controlados por um meta-objeto se forem declarados como reflexivos, caso contrário, serão tratados de forma convencional.

Diretivas especiais são adicionadas ao pré-processador para serem utilizadas na criação de objetos, métodos e variáveis reflexivas. Para criar um objeto reflexivo deve-se declarar sua classe da seguinte forma:

//MOP reflect class X: M;

A diretiva //MOP indica que uma instância da classe X pode ser reflexiva e que seu meta-objeto será uma instância da classe M. Quando o compilador lê esta linha de código, ele cria, automaticamente, uma classe denominada refl_X. A classe refl_X é uma subclasse de X.

Se o usuário desejar criar um objeto reflexivo, deve criar uma instância da classe refl_X; caso deseje criar um objeto comum, deve instanciar a classe X.

Como a declaração de uma classe reflexiva é iniciada com //MOP - e barras duplas são indicadores de comentários em C++ - os códigos produzidos para o pré-processador Open C++ poderão ainda ser reutilizados e compilados por um compilador C++.

Para declarar métodos ou variáveis reflexivas é utilizada a diretiva

```
//MOP reflect:
    int método1();
    ... // outros métodos reflexivos
```

Ou, ainda, os métodos reflexivos podem ser declarados com um *nome de categoria*:

```
//MOP reflect(nome_de_categoria):
```

O nome de categoria é utilizado para que o meta-objeto reconheça o método reflexivo como pertencendo a uma categoria especificada, podendo, assim, adotar os procedimentos apropriados àquela categoria, chamando o "meta-método" correspondente.

Os meta-objetos criados devem ser herdeiros da classe **MetaObj**. A classe MetaObj define um conjunto de métodos que servirá de suporte às extensões de linguagens.

Um desses métodos, denominado *Meta_MethodCall()*, é invocado sempre que uma chamada a um método reflexivo é realizada. Sua sintaxe é a seguinte:

```
void Meta_MethodCall(Id method, Id category, ArgPac& args, ArgPac& reply);
```

onde:

method: identificador do método do objeto-base chamado;

category: identificador da categoria do método chamado;

args: referência para um objeto contendo os parâmetros do método chamado;

reply: referência para uma variável na qual o valor de retorno do método chamado é armazenado.

Outro método da classe MetaObj, denominado *Meta_HandleMethodCall()*, é utilizado para executar um método reflexivo do objeto base. Sua sintaxe é:

void Meta_HandleMethodCall(Id method, ArgPac& args, ArgPac& reply);

onde:

method: identificador do método do objeto-base a ser executado;

args: referência para um objeto contendo os parâmetros do método chamado;

reply: contém o valor de retorno do método solicitado.

Somente com os dois métodos anteriormente citados já é possível realizar uma chamada a um método reflexivo, o que será ilustrado na Figura 2.1.

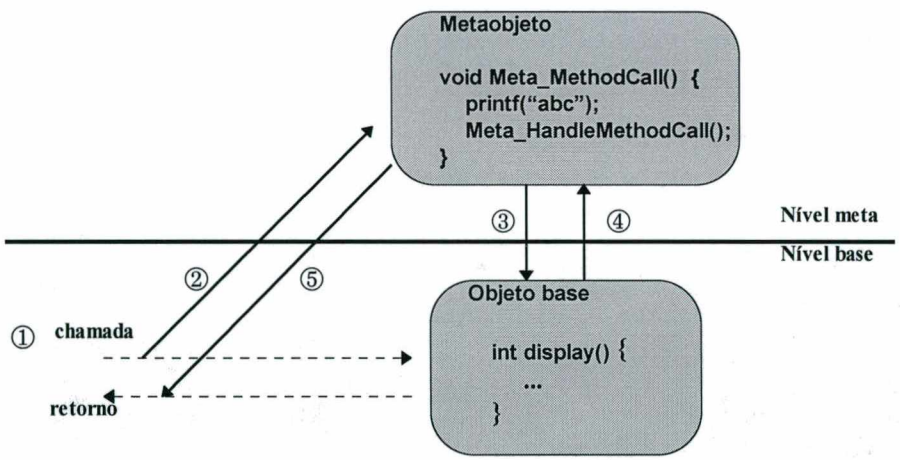


Figura 2.1. Dinâmica de uma chamada a um método reflexivo.

Na Figura 2.1, uma chamada ao método *display()* do objeto-base (①) é desviada ao meta-objeto correspodente (②), sendo invocado o método *Meta_MethodCall()*. O método *Meta_MethodCall()* define que a mensagem "abc" deve ser exibida na tela antes da execução do método *Meta_HandleMethodCall()*. O método *Meta_HandleMethodCall()*, por sua vez,

invoca o método *display()* (③ e ④). Após a execução de *display(),* o controle retorna ao objeto que originou a chamada (⑤).

Os argumentos do método do objeto-base solicitado devem ser armazenados num objeto da classe ArgPac, como citado anteriormente. Alguns tipos de dados como int, float e ponteiros, poderão ser passados como parâmetro diretamente, sendo suportados pelo compilador. Outros tipos de dados, como *structs*, por exemplo, devem ser manipulados de maneira particular, pois não são aceitos pelo compilador do Open C++. Um conjunto de métodos são disponíveis na classe ArgPac para manipulação dos tipos de dados definidos pelo usuário.

O Open C++ provê aos usuários uma *framework* para auxiliar a implementação de aplicações distribuídas, denominada *Comunidade de Objetos.* A Comunidade de Objetos tem como principal objetivo auxiliar o gerenciamento de um grupo de objetos distribuídos, de forma que múltiplos processos possam ser manipulados.

Um meta-objeto pertencente à Comunidade de Objetos pode controlar a concorrência interna na execução dos métodos de seu objeto-base, podendo, inclusive, executar de forma concorrente múltiplos métodos destes objeto.

Porém, os métodos de um meta-objeto só podem ser executados seqüencialmente. Para que haja concorrência no nível meta, seria necessária a adoção de um segundo nível meta, um "meta-meta-objeto" para controlar a concorrência do meta-objeto.

O *overhead* causado pela utilização de um protocolo de meta-objetos foi medido para o Open C++ [Chi93a]. Os valores foram obtidos sobre um sistema composto de uma SPARC Station 2, sistema operacional SunOS 4.1.1 e compilador Sun C++ 2.1.

Os resultados mostram que uma chamada a um método reflexivo é de 6 a 8 vezes mais lenta que uma chamada a um método virtual, o que foi considerado aceitável, principalmente em aplicações distribuídas. Ainda segundo Chiba, em [Chi93a], se em uma

determinada aplicação, o *overhead* no desempenho for menor que um fator de 10, a utilização da técnica reflexiva é indicada.

2.1.1.1. Considerações sobre a linguagem Open C++

A princípio, a linguagem Open C++ seria adotada para o desenvolvimento deste trabalho por possuir uma série de características adequadas, como o suporte a reflexão computacional e à programação distribuída. Porém, alguns aspectos referentes à concorrência inviabilizaram sua utilização. Isto porque o Open C++ permite a concorrência a nível base, mas não a nível meta, que era o desejado. Para obter concorrência a nível meta seria necessário definir um segundo nível meta (um meta-meta-objeto), o que descaracterizaria a estrutura do modelo adotado (RTR). Assim, embora a linguagem Open C++ não tenha sido adotada, seu estudo consta deste trabalho por contribuir como um exemplo da aplicação da técnica reflexiva.

2.2. Sistemas de Tempo Real

De modo formal, um sistema de tempo real é um sistema para o qual é requerida uma reação a estímulos oriundos do ambiente (incluindo o passar do tempo físico) dentro de intervalos de tempo impostos por este ambiente [Aud90]. São inúmeros os exemplos de aplicações de tempo real e, dentre estas, podemos citar os sistemas de controle de tráfego aéreo, controle de processos, sistemas multimídia, entre outros.

Nos sistemas de tempo real, alguns aspectos particulares têm que ser considerados. Um processamento ativado por um estímulo do ambiente deve se completar dentro de prazos (*deadlines*) especificados, senão, considera-se uma falha do sistema. Assim, pode-se afirmar que, além de correção lógica, um sistema de tempo real deve apresentar correção temporal (*timeliness*).

Para classificar os sistemas de tempo real, alguns dos critérios adotados por Audsley [Aud90], Berryman [Ber93], Farines [Far93] e Kopetz [Kop92a] podem ser utilizados. A seguir, são apresentados alguns deles:

• **Criticalidade** ➪ refere-se às consequências de uma falha no sistema. Estas falhas podem ser *benignas* ou *catastróficas*. As consequências de uma falha benigna consistem basicamente na perda dos benefícios oferecidos pelo sistema quando em operação normal. Já as consequências de uma falha catastrófica são de magnitude muito superior aos benefícios oferecidos pelo sistema, podendo implicar até em perdas humanas. Assim, os sistemas de tempo real podem ser críticos, quando existe a possibilidade de pelo menos uma falha catastrófica, ou não críticos, quando todas as falhas possíveis são benignas. Se num sistema de tempo real crítico, uma falha catastrófica pode ocorrer devido a falhas temporais, este sistema é chamado um sistema de tempo real *hard*.

• **Modos de Demanda** ➪ refere-se à carga externa a qual o sistema de tempo real deve responder. Esta carga pode ser uniforme ou não uniforme. Quando a carga é não uniforme, ela pode variar de um valor mínimo a um valor de pico, que deve ser identificado a fim de se estimar os recursos necessários do sistema para suportá-lo.

• **Modos de Resposta** ➪ refere-se ao grau de funcionalidade do sistema para evitar uma catástrofe. Dentre os vários modos de resposta apresentados pelos sistemas de tempo real, podemos ressaltar o modo de funcionalidade plena ou silêncio, num extremo - indicando que qualquer um dos estados evitará uma catástrofe -, ou o de funcionalidade plena - como único estado aceitável -, noutro extremo. Outros modos intermediários podem ser estabelecidos, fornecendo modos de funcionamento degradados ou alternativos.

• **Previsibilidade** ➪ refere-se à capacidade de se conhecer o comportamento temporal de um sistema de tempo real antes de sua execução. Segundo [Sta88], a previsibilidade é uma das mais importantes características de um sistema de tempo real. Um sistema previsível pode apresentar uma antecipação determinista (afirmar que todos os

deadlines serão cumpridos) ou uma antecipação probabilista (qual a probabilidade de que os *deadlines* sejam cumpridos) [Fra94].

Os processos (ou tarefas) dependentes do tempo, são executados pelo processador do sistema em uma determinada ordem, ditada por uma política de escalonamento. O objetivo do escalonamento é conseguir que todos os recursos estejam disponíveis para que os processos sejam executados satisfazendo as suas restrições temporais. Para isso, um escalonador é responsável pela construção de uma escala de execução, que indica a ordem de ocupação do processador por um conjunto de tarefas.

Os algoritmos de escalonamento podem ser classificados em [Kop92a]:

Escalonamento para tempo real *soft* x Escalonamento para tempo real *hard* ➪ no contexto dos sistemas tempo real *soft* é tolerado que, em circunstâncias adversas, nem todas as tarefas de tempo real sejam terminadas. Já em sistemas tempo real *hard*, os *deadlines* de todas as tarefas críticas devem ser garantidos antecipadamente a sua execução.

Escalonamento estático x Escalonamento dinâmico ➪ o escalonamento é dito estático quando a escala de execução é calculada *off line,* ou seja, previamente à execução do sistema. Já no escalonamento dinâmico, a escala é calculada em tempo de execução, com base nas requisições correntes.

Escalonamento preemptivo x Escalonamento não preemptivo ➪ no caso do escalonamento preemptivo, a execução de uma tarefa pode ser interrompida por outra tarefa de maior urgência. Se as tarefas em execução não podem ser suspensas, o escalonamento é dito não preemptivo.

Escalonamento centralizado x Escalonamento distribuído ➪ Num sistema distribuído, o escalonamento centralizado ocorre quando todas as decisões de escalonamento são realizadas num único nó da rede. No entanto, se as decisões de escalonamento são feitas de forma cooperativa através de algoritmos distribuídos, tem-se o escalonamento distribuído.

2.3. Modelos para Programação de Sistemas Tempo Real

Alguns modelos têm sido propostos a fim de direcionar a implementação de ferramentas para o tratamento de aplicações de tempo real, ditando a filosofia de programação a ser adotada e a dinâmica desejada do sistema a ser implementado. A seguir, faremos uma análise dos modelos RTC++, RTO.k, DRO e RT Java.

2.3.1. RTC++

RTC++ (*Real Time C++ model*) é uma extensão da linguagem de programação C++ para aplicações de tempo real. O modelo proposto por RTC++ introduz o conceito de objetos ativos, que diferem dos objetos normais de C++ porque podem possuir múltiplos fluxos de execução (*threads*). Os objetos ativos, se definidos com restrições temporais, são denominados objetos de tempo real.

As *threads* dos objetos ativos são denominadas *threads* **membro** *(member)* e podem ser de dois tipos: **mestre** (*master)* e **escravo** (*slave)*. As *threads* "mestre" são responsáveis pela execução de métodos periódicos e as *threads* "escravo" são responsáveis pela execução dos demais métodos do objeto. O mecanismo de herança de prioridades é utilizado para evitar o problema de inversão de prioridades.

Para controlar a concorrência, RTC++ utiliza-se de regiões críticas, implementadas através de expressões de guarda. Este suporte é oferecido, na linguagem RTC++, por uma classe especial chamada Active Entity, a partir da qual pode ser definida uma classe *Region* com os mecanismos para controlar o acesso à região crítica.

As restrições temporais podem ser traduzidas através de algumas expressões que devem ser utilizadas a nível de declaração. Estas expressões, na linguagem RTC++ são:

within ➪ duração da execução;

at ⇨ restrição de início de execução;

before ⇨ restrição de fim de execução.

A especificação das restrições temporais, com base em expressões pré-definidas imprime uma certa rigidez ao modelo.

Os métodos periódicos podem ser especificados através da declaração *cycle*, da seguinte maneira:

```
cycle (starttime; endtime; period; deadline) {
        ... ;
}
```

Para o tratamento de exceções, a linguagem RTC++ dispõe da palavra reservada *except*. Os segmentos de código iniciados por *except* são denominados blocos manipuladores de exceção e sucedem blocos *do*, *within*, *cycle* ou *region*. Uma declaração *within*, por exemplo, pode ser seguida de um bloco manipulador de exceções como:

```
within(time) {
        ... ;
} except {
        ... ;
}
```

A comunicação suportada por RTC++ é síncrona e provê dois tipos de mensagens de resposta: *reply* e *return*. A semântica da declaração *return* consiste no envio de uma resposta ao chamador e a finalização da função. A declaração *reply*, por outro lado, consiste no envio de uma resposta ao chamador e a execução de declarações subsequentes, ao invés do término da função.

RTC++ não suporta comunicação assíncrona, o que restringe, de certa forma, a dinâmica muitas vezes necessária à definição de aplicações tempo real.

2.3.2. RTO.k

RTO.k *(Real Time Object model)* [Kim94] é um modelo de objetos tempo real que se caracteriza pela manipulação precisa do comportamento temporal do sistema modelado e por possuir um alto grau de abstração em sua definição.

O modelo é independente da linguagem utilizada na especificação de seus objetos bem como da forma como o mecanismo de herança é oferecido; além disso, RTO.k é independente do protocolo de mensagens utilizado para troca de informações entre objetos.

Algumas diretrizes básicas são definidas na concepção de RTO.k: para cada execução de um método de um objeto do modelo é imposto um *deadline*; alguns métodos utilizam um relógio tempo real que é responsável por disparar a execução de métodos que são chamados métodos *time-triggered* (TT); finalmente, em relação aos dados contidos em objetos RTO.k, pode-se afirmar que os mesmos tornam-se inválidos depois que um intervalo de tempo máximo é espirado.

Uma característica interessante do RTO.k é a separação entre os *métodos time-triggered*, também chamados métodos espontâneos (SpM) dos métodos *message-triggered* (MT), também chamados métodos de serviço (SvM).

A concorrência na execução dos métodos de um objeto RTO.k pode ser das três formas seguintes:

(1) Concorrência entre métodos -TT
(2) Concorrência entre métodos -MT
(3) Concorrência entre métodos -TT e -MT.

No caso (1) a concorrência é naturalmente estabelecida. Nos casos (2) e (3), o espaço de dados do objeto (ODS) deve ser explicitamente declarado para cada método.

ODS é uma unidade de armazenamento atômico especificada pelo projetista, composta por definições de tipo e declaração de variáveis. Os métodos devem especificar os ODS's que irão utilizar e a forma como isto será feito (*read-only* ou *read-write*). A concorrência será permitida sempre que não houver conflitos entre os dados.

A interação entre os objetos é feita por chamadas dos objetos do cliente para métodos de serviço dos objetos do servidor. São aceitas em RTO.k as chamadas síncronas e assíncronas.

O nível de abstração apresentado por RTO.k, a definição básica de sua estrutura e a total independência de uma linguagem de programação ou suporte, atribuem ao modelo um alto grau de flexibilidade, permitindo que o mesmo possa ser utilizado na definição de modelos mais específicos.

Diferente da maioria dos modelos para tempo real, o modelo RTO.k não propõe nenhum mecanismo para manipulação de erros, o que o torna, de certa forma, deficiente na representação de problemas mais realísticos pertencentes ao domínio tempo real.

2.3.3. DRO

DRO (*Distributed Real Time Object model)* [Tak92] é um modelo de objetos tempo real distribuído que pode ser caracterizado por exercer as propriedades de melhor esforço (*best effort)* e mínimo sofrimento (*least suffering)* na tentativa de satisfazer as especificações temporais das tarefas a ele associadas.

Em DRO, os objetos possuem uma única *thread* (*single thread*). Um servidor armazena as mensagens recebidas e a cada fim de execução de um método, ele seleciona e inicia outro pedido em condições de execução.

A comunicação entre objetos tempo real distribuídos é baseada em invocações polimórficas temporais por parte do cliente e métodos com restrição temporal por parte do

servidor. No caso de polimorfismo temporal, restrições temporais são atribuídas a procedimentos distintos; o procedimento a ser adotado é selecionado em tempo de execução, sendo escolhido aquele que puder preencher da melhor forma o tempo disponível para execução.

O modelo DRO prevê suporte a métodos com execução periódica e a especificação do tempo de execução dos métodos dos objetos tempo real.

Os protocolos de comunicação podem ser construídos pelo usuário através das primitivas assíncronas *send/receive.* Um protocolo pode ser definido através de um diagrama de transição de estados. Os estados correspondem à abstrações das condições de execução e às ações correspondem ao envio/recepção de mensagens. A flexibilidade na definição de um protocolo de comunicação é uma característica particular e essencial de DRO.

O modelo DRO adota o paradigma da reflexão computacional (ítem 2.3), estabelecendo dois níveis de objetos, base e meta. Os objetos-base implementam as funções da aplicação e os meta-objetos são responsáveis pelo gerenciamento dos objetos-base. Os dois meta-objetos definidos em DRO são:

- **ProtocolMeta:** responsável pela definição e controle dos protocolos;
- **AbstractStateMeta:** responsável pelo gerenciamento dos pedidos em condição de execução.

Este modelo serviu de base para a especificação de DROL, uma extensão da linguagem C++ com a capacidade de descrever aplicações tempo real distribuídas.

2.3.4. RT Java

RT Java (*Real Time* Java) [Nil95] é uma linguagem para programação de aplicações tempo real, estendida a partir da linguagem Java e baseada num modelo de execução que

tem como objetivo principal facilitar a especificação de tarefas com restrições temporais, preservando as características de simplicidade encontradas em Java.

A arquitetura do Modelo RT Java prevê a existência de atividades tempo real, constituídas pela união de um gerenciador de configuração, um administrador e objetos tempo real.

Assume-se que os métodos dos objetos tempo real executem no início de seus períodos. Porém, na realidade, os mesmos podem executar a qualquer instante dentro destes períodos, contanto que terminem antes da finalização dos mesmos. A prioridade destes objetos deve ser sempre maior que a prioridade das demais tarefas do sistema. Os algoritmos de escalonamento disponíveis são o *rate monotonic* e o *cíclico estático.*

Os objetos tempo real que compõem o modelo podem ser cíclicos, esporádicos, dinâmicos ou contínuos e devem ser criados a partir de uma classe denominada Task. Os objetos tempo real **cíclicos** constituem-se de uma única *thread* que deverá conter: um segmento de código inicial, com tempo de execução limitado pelo pior caso; um manipulador de exceções; um ou mais segmentos de código com tempo de execução limitado e a frequência de execução.

Os objetos **esporádicos** constituem-se de uma *thread,* composta por um segmento de início, com tempo de execução de pior caso; um manipulador de exceções; um ou mais segmentos de código com tempo de execução limitado e a frequência de execução de pior caso. Estas tarefas poderão ser disparadas por interrupções ou pela detecção de uma condição particular.

Os objetos tempo real **dinâmicos** são constituídos por uma *thread* composta por um segmento de início, com tempo de execução de pior caso; um manipulador de exceções de finalização; um ou mais segmentos de código com tempo de execução limitado. O executivo tempo real é responsável por interromper esta *thread* quando o tempo restante disponível para que ela se execute iguala o tempo requerido para executar o método de finalização.

Os objetos tempo real **contínuos** são constituídos por uma *thread* e diferem dos objetos cíclicos por terem sua *thread* reassumida, e não reiniciada a cada período de execução.

O gerenciador de configuração é responsável por ajustar a atividade tempo real ao ambiente computacional local, determinando quais métodos serão interpretados, calculando os tempos de execução das tarefas e determinando os requisitos de memória necessários.

A função do administrador é a negociação de recursos com o executivo tempo real. O administrador comunica ao executivo quais os recursos necessários ao cumprimento das restrições temporais, baseado na análise de configuração feita pelo gerenciador. Estes requisitos temporais são expressos ao executivo tempo real em termos da freqüência de execução, do tempo mínimo de execução, e do tempo de execução desejado para cada tarefa que compõe a atividade tempo real.

O executivo tempo real é responsável pela alocação de recursos, decidindo qual o tempo de processador e o montante de memória que serão cedidos a cada atividade.

Em relação à flexibilidade, algumas restrições são encontradas, como a dificuldade na especificação de um algoritmo de escalonamento alternativo. Além disso, o modelo proposto é fortemente vinculado à linguagem suporte, sendo muito difícil o seu mapeamento sobre outras linguagens de programação.

2.3.5. O Modelo R2

O modelo R2 [Hon94] é um modelo computacional concorrente baseado em objetos, que adota o esquema de reflexão individual. Sua arquitetura é proposta para a programação de sistemas de tempo real e foi implementada sobre a linguagem ABCL/R2 [Mat91].

Basicamente, o modelo R2 é composto por um conjunto de objetos básicos que são controlados por seus respectivos meta-objetos. Estes objetos executam de forma concorrente,

não existindo concorrência interna (entre métodos de um mesmo objeto). A um conjunto de métodos de um objeto é dado o nome de *script* e um método é denominado um *segmento de script.*

Os procedimentos básicos de um objeto são:

1. receber uma mensagem;
2. selecionar um *segmento de script* e
3. executar o *segmento de script.*

Estes procedimentos são executados sequencialmente. A comunicação entre objetos é baseada no mecanismo de passagem de mensagens.

A arquitetura R2 é constituída de objetos no nível base, um meta-objeto e um meta-objeto tempo real para cada objeto-base, um objeto escalonador e um objeto relógio comuns a todos os objetos. A Figura 2.2 ilustra a arquitetura R2, onde são identificados todos os componentes do modelo.

Objetos-base: nos objetos-base os métodos são especificados contendo restrições temporais. Estas especificações são passadas como mensagens para o meta-objeto correspondente a este objeto-base. Quando o meta-objeto recebe a notificação da mensagem, ele a transfere para seu meta-objeto tempo real, responsável pela satisfação das restrições temporais.

Meta-objetos padrão: os meta-objetos padrão são responsáveis por estabelecer a comunicação entre os objetos-base e seus respectivos meta-objetos tempo real. Suas atividades incluem o controle da concorrência e a manipulação de mensagens a fim de facilitar esta comunicação.

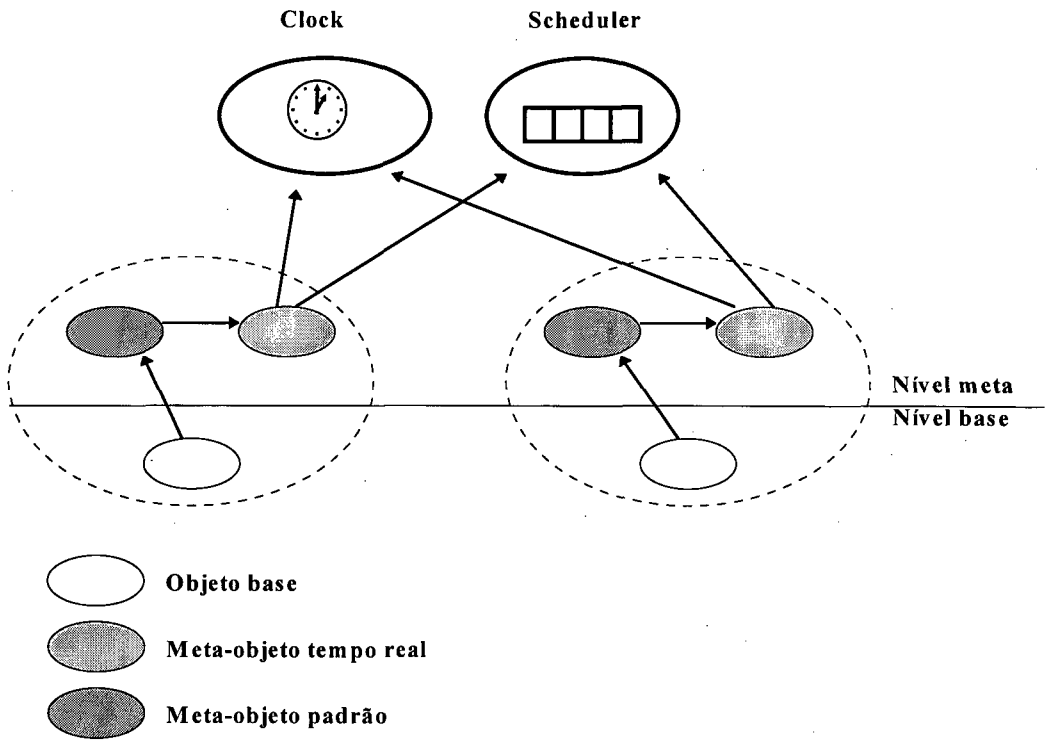


Figura 2.2: Arquitetura do Modelo R2

Meta-objeto tempo real: os meta-objetos tempo real são responsáveis pelo tratamento das restrições temporais associadas aos métodos. Sua implementação é totalmente dependente da aplicação, devendo o mesmo estar apto a suportar as restrições temporais associadas a esta. O meta-objeto tempo real deverá interagir com os meta-objetos *alarm-clock* e *scheduler* a fim de promover o melhor esforço (*best effort*) para satisfazer as restrições temporais e o mínimo sofrimento (*least suffering*) caso as mesmas não possam ser satisfeitas.

Meta-objeto alarm-clock: o meta-objeto *alarm-clock* é responsável por prover as funções de relógio do sistema. Suas atividades básicas consistem em retornar o valor do tempo corrente e atuar como um alarme, sinalizando um determinado instante de tempo.

Meta-objeto Scheduler: o meta-objeto *scheduler* determina a ordem de execução de segmentos de *script*. Estes segmentos devem executar de forma concorrente a fim de se obter maior utilização dos recursos disponíveis no sistema. O meta-objeto *Scheduler* permite

a mudança do algoritmo de escalonamento, já que este procedimento pode contribuir para a satisfação das restrições temporais.

A arquitetura proposta no modelo R2 apresenta como principal característica, a adoção da programação reflexiva. Este procedimento confere ao modelo um alto grau de flexibilidade, tanto na especificação de restrições temporais quanto na mudança do algoritmo de escalonamento.

Por não garantir o cumprimento das restrições temporais associadas às tarefas (não realiza análise de escalonabilidade), o modelo R2 é indicado para o tratamento de aplicações de tempo real *soft*.

2.4. Sumário

Este capítulo definiu o contexto a partir do qual este trabalho foi realizado. Foram apresentadas algumas definições básicas a respeito de Sistemas Tempo Real e as classificações mais adotadas, além da classificação de algoritmos de escalonamento proposta por Stankovic [Sta94]. Alguns modelos de programação de aplicações tempo real foram analisados, procurando-se salientar dos mesmos as características mais importantes. Os modelos abordados foram RTC++, RTO.k, DRO e RT Java. A seguir, foram apresentados alguns conceitos básicos a respeito do paradigma da reflexão computacional e a linguagem de programação Open C++. Finalmente, foi apresentado o Modelo R2, que incorpora características de reflexão e é dedicado a programação de aplicações tempo real.

Capítulo 3

Modelo Reflexivo para Tempo Real

Este capítulo tem como objetivo descrever o Modelo Reflexivo para Tempo Real [Fur95], no qual este trabalho será baseado. Assim, estudaremos as principais características do modelo, sua estrutura e dinâmica de funcionamento e faremos uma análise detalhada de cada um de seus componentes.

3.1. Características Gerais

Desenvolvido no Laboratório de Controle e MicroInformática (LCMI), o Modelo Reflexivo para Tempo Real estabelece uma filosofia de programação a ser adotada na implementação de ferramentas de desenvolvimento e em aplicações para tempo real [Fur95].

Definido segundo os paradigmas da orientação a objetos e reflexão computacional, este modelo reúne uma série de características que o tornam extremamente adequado aos fins para os quais se propõe.

A orientação a objetos lhe confere um alto grau de modularidade, reusabilidade e facilidade de manutenção, além de possibilitar o tratamento adequado de questões como concorrência e sincronização. Além disso, a adoção da orientação a objetos torna viável a aplicação da técnica de reflexão computacional segundo a abordagem de meta-objetos.

Neste modelo, é adotado o paradigma da reflexão individual, onde cada objeto-base possui um meta-objeto associado. Os objetos-base são responsáveis pela implementação das

atividades funcionais do modelo e os meta-objetos, responsáveis pelas atividades não-funcionais; desta forma, estas questões podem ser tratadas separadamente.

As atividades funcionais dizem respeito aos algoritmos da aplicação, cujos códigos devem ser implementados através de objetos-base; já as atividades não-funcionais, dizem respeito ao controle (gerenciamento) efetuado sobre o processamento da aplicação, implementado através dos meta-objetos. No caso do Modelo RTR, as atividades de controle correspondem a questões como restrições temporais, exceções temporais e sincronização, entre outras, impostas durante a execução dos algoritmos da aplicação.

As restrições temporais inerentes a uma aplicação tempo real devem ser expressas como restrições associadas a execução de métodos dos objetos-base. Estas restrições, como citado anteriormente, serão analisadas e tratadas a nível meta sendo, inclusive, implementadas pelo meta-objeto correspondente a este objeto-base. O usuário poderá dispor de uma classe pré-definida onde encontrará métodos implementando restrições temporais mais usuais, podendo, ainda, redefinir outras conforme suas necessidades. Esta flexibilidade deverá ser encontrada, também, no que se refere ao algoritmo de escalonamento adotado, podendo o mesmo ser escolhido pelo usuário dentre aqueles disponíveis pelo modelo ou implementados particularmente, no sentido de atender da melhor maneira a uma aplicação específica.

Da forma como foi especificado, o Modelo RTR é adequado a aplicações que se enquadrem no contexto *soft real time*, visto que as restrições temporais associadas a uma determinada aplicação não têm a garantia de ser atendidas, ainda que as violações destas restrições sejam certamente detectadas e os manipuladores de exceções correspondentes sejam ativados, implementando-se, assim, uma política *best effort*. A adequação do modelo ao contexto *hard real time* dependerá de alterações no sentido de garantir o cumprimento das restrições temporais, o que está estreitamente relacionado a possibilidade de se realizar "*worst case analysis*" envolvendo os dois níveis: nível meta e nível base.

O modelo deverá possibilitar a concorrência a nível de meta-objetos, a fim de que vários pedidos de ativação de métodos da aplicação possam ser tratados simultaneamente. Porém, a nível base, deve ser garantida a exclusão mútua na execução de métodos de um mesmo objeto, função esta que deverá ser gerenciada pelo meta-objeto correspondente ao objeto-base em questão.

3.2. Estrutura do Modelo RTR

O Modelo RTR é composto por objetos-base e seus respectivos meta-objetos (que serão chamados meta-objetos Manager), por um meta-objeto Clock e um meta-objeto Scheduler.

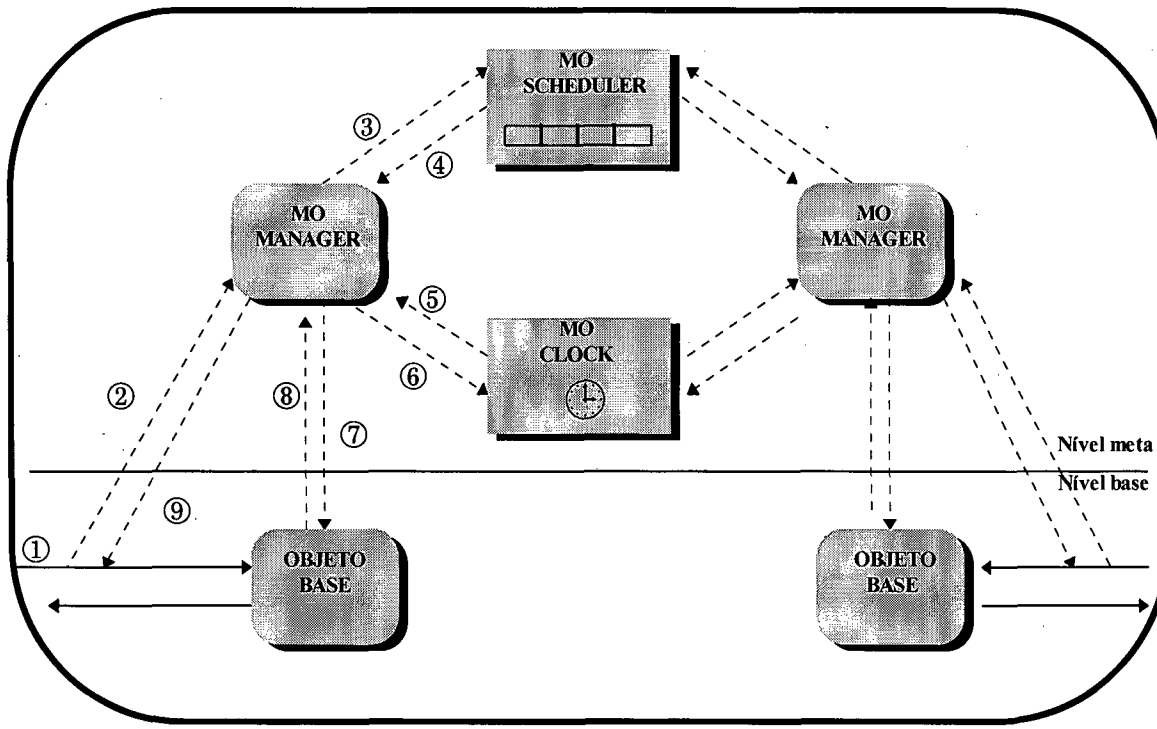


Figura 3.1: Estrutura do Modelo Reflexivo para Tempo Real.

Com o auxílio da Figura 3.1, podemos acompanhar a descrição da dinâmica esperada do modelo, que tem início quando um dos objetos-base da aplicação recebe uma solicitação para ativação de um de seus métodos ①. Esta solicitação deve ser desviada ao meta-objeto correspondente ②, para que sejam tratadas as questões temporais e de

sincronização. O meta-objeto Manager deverá interagir com os meta-objetos Scheduler (③ e ④) e Clock (⑤ e ⑥) a fim de que as restrições temporais associadas aos métodos sejam processadas. Caso estas restrições sejam cumpridas, o meta-objeto Manager sinaliza ao seu objeto-base a execução do método solicitado ⑦. Após a execução do método no objeto-base, o controle retorna ao meta-objeto Manager ⑧, que retorna os resultados ao objeto que originou a chamada ⑨.

O modelo pode ser implementado de forma distribuída, sendo que os pares objeto-base/ meta-objeto devem estar presentes em um mesmo nodo da rede. A restrições temporais são consideradas a nível local e, além dos pares objeto-base/meta-objeto Manager deverão estar presentes um meta-objeto Scheduler e um meta-objeto Clock em cada nodo considerado do sistema distribuído.

As seções seguintes descreverão os objetos (objeto-base e meta-objetos) que compõem o modelo. A figura 3.2 ilustra as interações entre os diferentes objetos.

3.2.1. Os objetos-base

Os objetos-base do modelo serão semelhantes aos objetos convencionais, com a diferença que são associados à restrições temporais e manipuladores de exceções nas declarações de seus métodos.

A cada objeto-base corresponde um meta-objeto Manager e esta ligação deverá ser feita automaticamente à criação do objeto-base. O meta-objeto Manager deverá possuir o mesmo nome do objeto-base, precedido da palavra "Meta".

Um objeto-base deverá constituir-se, basicamente, de três seções: seção de definição de tipos de restrições temporais, seção de declaração de dados e seção de declaração de métodos.

Na ***seção de definição de tipos de restrições temporais*** poderão ser introduzidos pelo usuário novos tipos de restrições, em adição aos básicos oferecidos pelo modelo. Os novos tipos poderão, inclusive, utilizar os preexistentes em sua implementação. De forma geral, a declaração de um novo tipo de restrição temporal pode ser:

RT-TYPE id-tipo-restrição = definição-do-tipo-de-restrição

A seguir, um exemplo da declaração de um novo tipo de restrição temporal:

RT-TYPE sporadic = (deadline, intervalo_min_entre_ativações, tempo_max_execução);

Na ***seção de declaração de dados***, deverão ser declaradas as variáveis e constantes do objeto-base, como feito habitualmente na declaração de um objeto comum.

Na ***seção de declaração de métodos*** deverão ser declarados os métodos do objeto-base, sendo que as restrições temporais e exceções serão associadas aos mesmos nessas declarações. A declaração de métodos poderá ser da seguinte forma:

<tipo> <id-método> (<parâmetros>), <restrição-temporal>, <id-exceção>

Por exemplo, a declaração de um método associado a restrições temporais poderá ser:

tipo alarme(...), aperiodic(D, TME=10), shutdown();

Neste exemplo, alarme(...) é um método que foi declarado como possuindo execução aperiódica, onde D é o valor de *deadline* dentro do qual ele deverá ser executado e TME é o tempo máximo estimado de sua execução, igual a 10 milisegundos. O método shutdown() corresponde ao método que deverá ser ativado, caso o método alarme(...) não consiga executar dentro do *deadline* especificado.

Os valores das restrições temporais são atribuídos ou na criação dos objetos-base ou fornecidos quando das chamadas aos métodos.

3.2.2. Os meta-objetos Manager (MOM)

Como definido anteriormente, os meta-objetos Manager serão responsáveis pelas questões referentes ao gerenciamento de seus respectivos objetos-base. As atividades compreendidas neste gerenciamento serão detalhadas ao analisarmos a estrutura destes meta-objetos.

A estrutura geral dos meta-objetos Manager compreende as seguintes seções: seção de gerenciamento, seção de sincronização, seção de restrições temporais e a seção de exceções.

A ***seção de gerenciamento*** deverá ser responsável pelo recebimento dos pedidos de ativação de métodos dos objetos-base (Fig. 3.2., ❶), verificando sua restrição temporal e encaminhando-os a seção de restrições temporais, mais especificamente, ao método que implementa a restrição temporal em questão (Fig.3.2, ❷). Se o método não possuir restrições temporais associadas, deverá ser tratado por um método específico da seção de gerenciamento.

A seção de gerenciamento deverá, ainda, ser responsável pela garantia da exclusão mútua na execução dos métodos dos objetos-base. Este controle se dará através de uma variável que identificará o estado deste objeto como "ativo" - quando um de seus métodos estiver sendo executado - ou "dormindo" - caso contrário. Os pedidos que tentarem executar enquanto o estado do objeto-base for "ativo" serão colocados em uma fila, denominada Fila de Pendências. Quando o pedido sendo processado for concluído, os pedidos colocados na Fila de Pendências serão retirados um a um, segundo a política a ser definida na implementação do modelo; se, por outro lado, a Fila de Pendências estiver vazia, o estado do objeto-base volta a ser "dormindo" novamente. Os pedidos que tentarem executar quando o estado do objeto-base for "dormindo" serão liberados somente se seu estado de sincronização assim o permitir.

Se existirem restrições de sincronização envolvendo a execução de métodos de um objeto-base, deverá ser implementada uma ***seção de sincronização***. Nesta seção serão declaradas as restrições de sincronização, conforme o mecanismo adotado para este fim, que poderá ser, por exemplo, os mecanismos de *path expression* ou *set enables*. Estas restrições estabelecem relações de precedência entre métodos do objeto-base.

A ***seção de restrições temporais*** deverá compreender as implementações das restrições temporais envolvidas na aplicação. Para cada restrição especificada nas declarações dos métodos de um objeto-base deverá haver uma implementação correspondente na seção de restrições temporais do respectivo meta-objeto Manager.

Os métodos que implementam as restrições temporais serão responsáveis pela solicitação de escalonamento do pedido de ativação de um método do objeto-base ao meta-objeto Scheduler (Fig. 3.2, ❸). Após retornar do escalonador, deverão ser verificadas as condições temporais para execução do método, além de suas condições de concorrência e sincronização (Fig. 3.2, ❹). Caso as restrições sejam satisfeitas, o meta-objeto Manager poderá liberar a execução do método solicitado (Fig. 3.2, ❺). Por outro lado, se as restrições temporais forem violadas, o método de exceção associado ao método do objeto-base deve ser ativado (Fig. 3.2, ❻). Opcionalmente, poderá ser adotado um mecanismo para salvar o estado do objeto-base, com o objetivo de se restaurar este estado caso a execução de um método deste objeto termine fora das condições temporais especificadas. Ainda através do método que implementa a restrição temporal, poderá ser realizada a interação com o meta-objeto Clock, a fim de programar ativações futuras de métodos e monitorar a passagem do tempo (Fig. 3.2, ❼).

A ***seção de exceções*** deverá conter os métodos que implementam os manipuladores de exceção associados aos métodos dos objetos-base. Os métodos de exceção deverão ser ativados quando da violação das restrições temporais especificadas. Para cada exceção declarada nos objetos-base deverá haver uma implementação correspondente no meta-objeto Manager.

3.2.3. O meta-objeto Scheduler (MOS)

O meta-objeto Scheduler será responsável pelo escalonamento dos pedidos de ativação de métodos dos objetos-base. O processo de escalonamento consistirá em receber os pedidos de ativação, ordená-los segundo a política de escalonamento especificada e, por fim, liberá-los.

O processo de escalonamento deverá atender pedidos originados a partir de um ou mais meta-objetos Managers pertencentes ao mesmo nodo. Para acomodar estes pedidos, o meta-objeto Scheduler deverá dispor de uma fila denominada Fila de Escalonamento.

Deve ser possível ao usuário do modelo escolher entre algoritmos de escalonamento disponíveis ou acrescentar outros que mais se adequem a sua aplicação, procedimento este que deverá ser realizado com alterações a nível meta, mais especificamente no meta-objeto Scheduler.

O meta-objeto Scheduler deverá dispor, ainda, de uma variável para indicar seu estado. Esta variável será igual a "dormindo," caso o meta-objeto Scheduler esteja em desuso e a Fila de Escalonamento esteja vazia; e será igual a "ativo," caso algum pedido de ativação esteja sendo escalonado, ou já estiver ordenado na Fila de Escalonamento.

Se outro pedido de escalonamento for solicitado enquanto o estado do MOS for "ativo", este será inserido na Fila de Escalonamento, ordenado segundo a política especificada. A *thread* correspondente deverá ser suspensa e o meta-objeto Clock programado para monitorar o *deadline* deste pedido, enquanto na Fila de Escalonamento.

Quando a execução do método solicitado for finalizada, ou esta execução for suspensa por questões de concorrência ou sincronização, o MOS será avisado. Neste caso, o estado do MOS passa a ser "dormindo," caso a Fila de Escalonamento esteja vazia ou, caso contrário, o próximo pedido na Fila de Escalonamento será reassumido.

3.2.4. O meta-objeto Clock (MOC)

O meta-objeto Clock deverá ser uma representação do relógio do sistema. As atividades desempenhadas por ele devem consistir, basicamente, na programação de ativações futuras e na monitoria do tempo destas ativações.

Assim, sempre que uma ativação futura de um método de um determinado objeto-base tenha que ser programada, isto será feito através do meta-objeto Clock. O armazenamento dos pedidos de ativação futura deverá ser feito em uma fila denominada Fila de Pedidos Futuros. Para que esta ativação seja efetivada no momento apropriado, o MOC será responsável pelo monitoramento da Fila de Pedidos Futuros, em intervalos de tempo regulares, a fim de iniciar os pedidos cujo instante de ativação tenha chegado. O MOC deverá possibilitar, ainda, o cancelamento dos pedidos de ativação futura, que deverão ser retirados da fila.

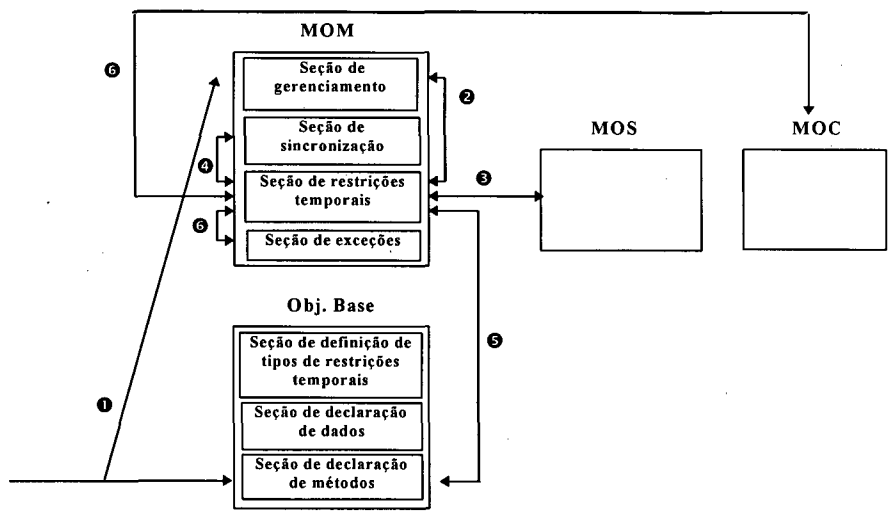


Figura 3.2. Interação entre os objetos/meta-objetos que compõem o modelo e suas respectivas seções

3.3. Sumário

Este capítulo se propôs a apresentar uma descrição detalhada do Modelo Reflexivo Tempo Real (RTR), que é a base para o desenvolvimento do nosso trabalho.

Nessa descrição, os principais conceitos e paradigmas que servem de fundamento ao modelo são identificados. Uma descrição da forma na qual o modelo é estruturado, através da descrição de seus objetos e meta-objetos é apresentada. E, por fim, uma descrição detalhada da estrutura de seus componentes - objetos-base, meta-objetos - foi realizada a fim de identificar as funções que deverão ser desempenhadas por cada um desses componentes.

Capítulo 4

Mapeamento do Modelo Reflexivo Tempo Real sobre a Linguagem Java

Este capítulo tem por objetivo descrever o mapeamento do Modelo Reflexivo Tempo Real sobre a linguagem Java. Para isso, é feita uma introdução à linguagem, onde são abordadas algumas de suas principais características. A seguir, descrevemos o mapeamento do modelo sobre a linguagem e a dinâmica de funcionamento resultante deste mapeamento.

4.1. A linguagem Java

Java é uma linguagem de programação orientada a objetos, desenvolvida pela Sun Microsystems que, além de sua semelhança com a linguagem C++, pode ser caracterizada pela portabilidade, robustez, além do suporte à programação distribuída e amplos recursos para implementação de aplicações multimídia. Seu compilador, o Javac, foi desenvolvido na própria linguagem Java e seu sistema de execução escrito em ANSI C, seguindo o padrão de portabilidade POSIX (Portable Operating System Interface).

Em Java, podem ser implementadas aplicações independentes (*stand alone applications*) ou "*applets*". As aplicações independentes são executadas diretamente pelo interpretador, possuindo um método "*main*" para configuração de um contexto de execução, seguindo os mesmos princípios de qualquer aplicação desenvolvida em C++. "*Applets*", por outro lado, são programas que podem ser referenciados por um documento HTML, podendo, desta forma, ser inseridos em páginas *web* para serem distribuídos através de redes como a

Internet. Eles são carregados e executam com o auxílio de um "*browser*" como o Netscape Navigator, por exemplo, ou o "*appletviewer*"[1].

A seguir, descreveremos algumas das principais características da linguagem Java:

4.1.1. Simplicidade e familiaridade

Um dos principais objetivos da Sun Microsystems foi fazer com que a linguagem Java fosse familiar à grande maioria dos programadores. Assim, muitas das construções de Java são semelhantes às de C e C++. Por outro lado, algumas das características de C++ consideradas ambíguas ou mesmo fora do contexto da programação orientada a objetos foram excluídas. Alguns aspectos importantes que contribuem para a maior simplicidade de Java são citados abaixo:

- O gerenciamento de memória e a coleta automática de "lixo" são exemplos de recursos que simplificam a programação em Java. Todas as vezes que alocamos memória para um objeto, o ambiente, em tempo de execução, armazena uma série de referências para este objeto. Quando não possuir mais referências, ou seja, quando estiver em desuso, este objeto está passível de ser coletado e a memória liberada. Desta forma, o programador não é mais responsável pelo gerenciamento de memória, o que diminui em muito a ocorrência de erros de programação.

- Em Java não são usados arquivos de cabeçalho porque não existem *#define*, *#typedef* nem diretivas de pré-processador. Todas as informações a respeito de uma classe são contidas num único arquivo, o arquivo onde a classe é declarada; assim, entender o código de outros programadores torna-se muito mais simples. A ausência de arquivos de cabeçalho é possível porque o compilador Java gera um código binário contendo todas as informações sobre as classes declaradas.

[1]"browser" fornecido pelo Java Development Kit (JDK) para testar os "applets" compilados.

• Em Java também não são usados tipos de dados como estruturas e uniões. Para substituí-los, são criadas classes com as variáveis de instância desejadas.

• Em Java, *arrays* são objetos que podem ser manipulados por índices e por um conjunto de métodos disponíveis, não existem estruturas, e objetos são passados por valor, e não por referência. Assim, *a priori*, a ausência de apontadores em Java não traz impecilhos, a não ser que seja necessário identificar métodos em tempo de execução. Esta é uma função que ainda não é suportada pela linguagem em sua versão 1.02. A não ser por este motivo, a ausência de ponteiros traz muitos benefícios, já que eles constituem uma das maiores fontes de erro na maioria dos programas onde são utilizados.

4.1.2. Java é orientada a objetos

O fato de Java ser totalmente orientada a objetos implica em dizer que tudo o que se deseja fazer num programa Java deve ser feito através de chamadas a métodos de um objeto. Até mesmo o método *main()* deve pertencer a uma classe que, por sua vez, deve herdar da classe primitiva *java.lang.Object*, a classe base para toda hierarquia de classes Java.

Baseada nos conceitos de outras linguagens orientadas a objetos como Eiffel, SmallTalk, Objective C e C++, Java incorporou, ainda, aspectos como encapsulamento, polimorfismo e herança.

4.1.3. Java é uma linguagem interpretada

Um programa escrito na linguagem Java, quando compilado, gera um código intermediário e não um código de máquina como ocorre na maioria das linguagens. Esse código intermediário é independente de plataformas de *hardware* e *software*. Desta forma, uma aplicação em Java pode executar, sem qualquer alteração prévia, sobre qualquer plataforma, bastando que haja para a mesma um interpretador desse código intermediário e o sistema de execução Java disponíveis.

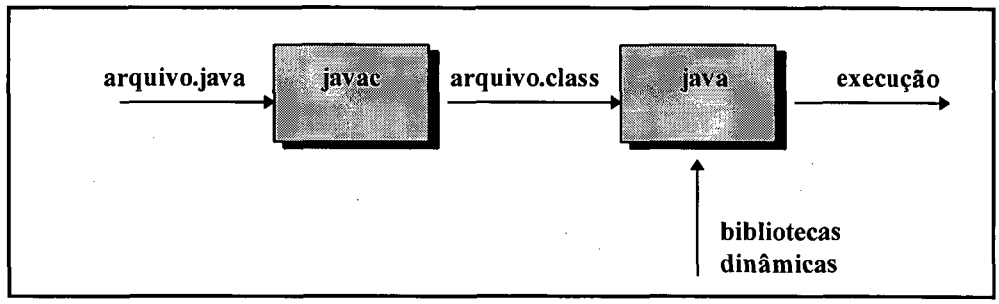


Figura 4.1: Ciclo de desenvolvimento de uma aplicação em Java.

A união do interpretador com o sistema de execução constitui a chamada Máquina Virtual de Java.

A arquitetura neutra, que independe da plataforma de execução, resulta em grande portabilidade para as aplicações, uma característica essencial aos sistemas computacionais atuais.

Outro aspecto importante de Java é a robustez. Além de verificar erros durante a compilação com bastante rigidez, a linguagem faz uma série de verificações em tempo de execução para garantir consistência. Além disso, o fato de não possuir ponteiros impede que dados sejam corrompidos por erros de endereçamento de memória e, como consequência, temos aplicações mais seguras e confiáveis. Ainda para garantir maior segurança, Java implementa um mecanismo de verificação do código de bytes. Este procedimento tem como objetivo evitar que códigos que não foram submetidos a um compilador Java tentem ser executados.

4.1.4. Exceções

A linguagem Java oferece um mecanismo de tratamento de erros através de exceções. Exceções são situações que não deveriam ocorrer em casos de execução normal de um programa, ou, como definido em [Sun95b], exceções são eventos que ocorrem durante a execução de um programa que rompem o fluxo normal de instruções. Por exemplo, podemos

passar como parâmetro de um método, um índice fora dos limites de um vetor. Neste caso, o método deverá detectar o erro e sinalizar a exceção correspondente.

4.1.5. Java suporta *multithreading*

Uma *thread* é um fluxo sequencial de controle dentro de um processo. Assim, um sistema *multithreading* é um sistema no qual vários fluxos de controle coexistem e concorrem pelo processador. Um exemplo bastante atual de sua utilização são as aplicações multimídia, onde é necessário, entre outros, o tratamento concorrente de som e imagem.

Java oferece os recursos de *multithreading* a nível de linguagem, possibilitando a criação e manipulação de *threads* de forma bastante acessível ao programador. Implementados pela classe *java.lang.Thread*, um conjunto de métodos permite que o programador inicie uma *thread*, pare, suspenda, reassuma, entre outras possibilidades.

A seguir, descreveremos alguns atributos que melhor definirão os serviços de *multithreading* oferecidos por Java [Sun95b]:

4.1.5.1. Corpo de uma *Thread*:

Toda *thread* deve possuir um método *run()*. Este método deverá conter todas as instruções a serem executadas pela mesma. Assim, quando uma *thread* é iniciada, o sistema de execução faz uma chamada ao método *run()*, que equivale ao corpo da *thread*.

Uma classe pode implementar o método *run()* sendo herdeira da classe *java.lang.Thread* ou implementando a interface *Runnable*. Interfaces declaram tipos, consistindo de um conjunto de métodos e constantes, sem especificar sua implementação. Uma classe pode ser declarada com uma ou mais interfaces, podendo desta forma implementar todos os métodos especificados pelas mesmas. A múltipla herança de interfaces permite que objetos suportem procedimentos comuns sem compartilhar sua implementação. A interface *Runnable* provê o método *run()* para a implementação de *Threads*. De modo

geral, deve-se optar pela interface *Runnable* quando a classe em questão já herda de alguma outra classe e por conseguinte não poderá herdar também da classe *java.lang.Thread* (Java não suporta múltipla herança!).

4.1.5.2. Ciclo de vida de uma *thread*:

Uma *thread* possui quatro estados distintos nos quais pode encontrar-se após ter sido criada:

• ***New thread*:** a *thread* foi criada mas não foi iniciada. Neste estado podem ser feitas chamadas aos métodos *start()* - para iniciar a *thread* - e *stop()* - para finalizá-la. Qualquer outro método da classe *java.lang.Thread* chamado neste estado sinalizará uma exceção do tipo *IllegalThreadStateException*.

• ***Runnable*:** a *thread* foi iniciada. Este estado é chamado *runnable*, e não *running*, porque em sistemas com um único processador a *thread* pode não estar realmente executando, mas sim esperando em uma fila, ordenada segundo o escalonador do sistema.

• ***Not Runnable*:** neste estado, mesmo que o processador se torne disponível, a *thread* não executará. Quatro eventos podem levar uma *thread* ao estado *não executável* e, para cada um deles há um procedimento dual que a tornará executável novamente. São eles:

EVENTO	PROCEDIMENTO DE RETORNO
suspend()	resume()
sleep()	Fim do número de milisegundos especificado
wait()	A variável esperada é liberada por notify() ou notifyAll()
bloqueio de I/O	É realizado o comando de I/O especificado

Figura 4.2: Eventos que levam uma thread ao estado *not runnable* e seus respectivos procedimentos de retorno ao estado *runnable*.

• ***Dead*:** quando o método *run()* de uma *thread* termina sua execução, a *thread* "morre". Por outro lado, se o método *run()* possuir um *loop* infinito ou, por qualquer outro motivo desejarmos matar a *thread*, podemos utilizar o método *stop()*.

Para cada um dos estados de uma *thread* citados anteriormente existe um conjunto de métodos que pode ser utilizado adequadamente. Porém, existem procedimentos que não podem ser executados em determinados estados, como, por exemplo, tentar suspender uma *thread* que já se encontra em estado *not runnable*. Caso isto seja feito, o sistema exibirá uma exceção do tipo *IllegalThreadStateException*.

4.1.5.3. Prioridades:

Todas as *threads* criadas em Java possuem um valor de prioridade que pode variar de 1 a 10 (representados pelas constantes MIN_PRIORITY e MAX_PRIORITY, respectivamente). Esse valor é herdado da *thread* criadora e pode ser verificado e alterado por métodos disponíveis na classe *java.lang.Thread*. O sistema Java utiliza um algoritmo de escalonamento baseado em prioridades para determinar a ordem de execução das *threads*; além disso, as mesmas são preemptivas e podem ser interrompidas a qualquer instante por outra de maior prioridade. Quando as *threads* da aplicação possuírem a mesma prioridade, o algoritmo de escolha utilizado pelo sistema Java é o *round-robin*.

4.1.5.4. *Threads* Daemon:

Uma *thread* pode ser explicitamente definida como *daemon* quando sua função é de prestação de serviços a outras *threads* do mesmo processo. As *threads daemon* existem em função das *threads* "comuns" (não *daemon*). Desta forma, quando as *threads* comuns terminarem sua execução, as *threads daemon* serão finalizadas automaticamente pelo sistema.

4.1.5.5. Grupos de Threads:

Em Java podem ser formados grupos de *threads*, afim de que todas as *threads* pertencentes ao mesmo grupo possam ser manipuladas de uma só vez, referenciando-se, para isso, o nome do grupo. Quando uma *thread* é criada sem que um grupo seja explicitado, ela fará parte do grupo da *thread* que a criou. Se nenhuma *thread* no sistema foi explicitamente criada dentro de um grupo, todas as *threads* neste sistema farão parte do grupo chamado "*main*" ou, no caso de um *applet*, o nome do grupo dependerá do *browser* sendo utilizado.

4.1.5.6. Sincronização:

Em programas *multithreading* que compartilham dados ou cujas *threads* dependem dos estados umas das outras, é importante que se utilizem recursos de sincronização para garantir a consistência dos dados compartilhados.

Java utiliza-se de monitores para controlar o acesso a estes dados. Quando uma *thread* detém o monitor, as outras ficam impedidas de acessar ou alterar a variável associada a ele. As regiões críticas nas quais estas variáveis são passíveis de ser alteradas são identificadas pela palavra reservada *synchronized*. Assim, duas *threads* não podem acessar regiões sincronizadas de um mesmo objeto simultaneamente. Essas regiões podem ser métodos ou segmentos menores de código.

4.2. Considerações a respeito da linguagem Java

O presente trabalho prevê a utilização da linguagem Java em aplicações de domínio tempo real. Assim, é interessante que sejam identificadas na linguagem as características que possam, de alguma forma, ser consideradas inadequadas a implementação destas aplicações.

No que se refere à sincronização, a linguagem Java utiliza-se de monitores para controlar o acesso aos dados compartilhados por diferentes *threads* (4.1.5.6). Desta forma,

poderão ocorrer inversões de prioridade na execução das *threads* do sistema. Inversões de prioridade ocorrem quando uma *thread* adentra uma região crítica e não pode ser preemptada por uma segunda *thread*, com prioridade maior que a sua , que tente acessar a mesma região.

O *garbage collector* (4.1.1) é uma funcionalidade da linguagem Java que pode inserir algum grau de indeterminismo às aplicações tempo real. Isto porque os atrasos provocados por sua execução são valores que não podem ser previstos. Entretanto, o *garbage collector* executa em uma *thread* de baixa prioridade e , portanto, poderia ser escalonado junto às *threads* da aplicação. Em outras palavras, ele só atuaria quando o sistema estivesse inativo, não comprometendo a atividade das demais *threads*.

4.3. Mapeamento do Modelo RTR sobre a Linguagem Java: RTR Java

Nesta seção, descreveremos como os recursos da linguagem Java foram utilizados para representar o Modelo Reflexivo Tempo Real e como as necessidades impostas pelo modelo - descritas no Capítulo 3 - foram satisfeitas.

A princípio, foram definidos objetos-base e meta-objetos como objetos de Java, sendo que a reflexão computacional é alcançada através da arquitetura implementada. A concorrência desejada no nível meta foi alcançada com a utilização de *multithreading*. No nível base, a exclusão mútua na execução de métodos-base é garantida pelo gerenciamento feito no meta-objeto.

O meta-objeto Scheduler possuirá uma fila denominada Fila de Escalonamento, responsável pela acomodação dos pedidos de execução de métodos de objetos-base, ordenados segundo a política de escalonamento adotada. O meta-objeto Clock, possuirá uma fila denominada Fila de Pedidos Futuros, que armazenará os pedidos de ativação futura de métodos, sendo utilizada pelas restrições temporais **periodic** e **start_at**. Esta fila deverá ser

ordenada, também, segundo a política de escalonamento adotada, o que agilizará o procedimento de procura das *threads* que devem ser disparadas pelo relógio e que estarão armazenadas na Fila de Pedidos Futuros. Essas duas filas serão utilizadas pelos diversos pares objeto-base/meta-objeto existentes na aplicação.

Cada meta-objeto possuirá também uma fila para armazenar os pedidos de execução de métodos que não estiverem aptos a executar devido a concorrência e restrições de sincronização. Esta fila é denominada Fila de Pendências e será uma fila FIFO (*first in, first out*).

Para garantir a consistência de variáveis compartilhadas, como as filas de Escalonamento e de Pedidos Futuros, além de outras, os métodos que manipulam estas variáveis devem ser declarados como sincronizados, utilizando-se a palavra reservada *synchronized*, da linguagem Java (ítem 4.1.5.6).

No item a seguir, descreveremos com mais detalhes como foi inserida a concorrência no nível meta.

4.3.1. Concorrência no nível meta

A utilização de *threads* de controle permitirá a concorrência a nível de meta-objetos. Desta forma, vários pedidos de execução de métodos de objetos-base poderão ser recebidos e tratados de forma simultânea.

Como definido no item 4.1.5, *threads* são fluxos seqüenciais de controle dentro de um processo. No caso do mapeamento do Modelo RTR, adotamos a concorrência passiva [Lea95], onde objetos são encarados como agentes passivos, que são percorridos por fluxos de controle, através de chamadas a métodos.

O suporte a multithreading com mecanismos de preempção oferecidos por Java, nos permite garantir que, a qualquer instante, a *thread* em execução será aquela de maior prioridade no sistema (em estado *runnable*).

O mapeamento define quatro tipos de *threads* que serão utilizadas para viabilizar a dinâmica desejada do sistema. Estes quatro tipos de *threads* são descritos a seguir:

4.3.1.1. *threads* principais

As ***threads* principais** são originadas a partir da solicitação de execução de um método de um objeto-base. Nestas *threads* são executados todos os procedimentos relativos ao escalonamento destes métodos, como a verificação da restrição temporal associada, a interação da seção de gerenciamento com as seções de sincronização e de restrições temporais e a interação do meta-objeto Manager com os meta-objetos *Scheduler* e *Clock*; além disso, a execução do método do objeto-base solicitado também é realizada em uma ***thread* principal**. As ***threads* principais** podem ser originadas a partir de chamadas síncronas ou assíncronas.

4.3.1.2. *thread* Clock

A ***thread* Clock** deverá atuar como o relógio do sistema. Criada a partir do construtor da classe *Clock*, ela deverá ser responsável pelo controle do tempo de ativação de métodos programados para serem ativados em instantes de tempo futuros. Assim, ela verificará a fila de pedidos de ativação futura e liberará aqueles pedidos cujo tempo de ativação é menor ou igual ao tempo atual no instante da verificação. A seguir, a *thread Clock* deverá "dormir" por um período de tempo determinado em função da aplicação, permanecendo no estado ***not runnable*** durante este período e permitindo que as outras *threads* do sistema com menor prioridade possam executar. A prioridade da *thread Clock* será máxima (**MAX_PRIORITY**) para permitir que ela interrompa a execução de outras *threads* quando expirar o tempo especificado.

Como sua função é a de prestar serviços ao modelo, a ***thread* Clock** será explicitamente definida como uma *thread* ***daemon*** (ítem 4.1.5.4). Assim, quando não houver mais nenhum pedido de execução de métodos do objeto-base pendentes, a ***thread* Clock** finalizará por si só.

4.3.1.3. *threads* periodic

As *threads Periodic* são criadas para atender a restrição temporal **periodic**. Um método com restrição temporal do tipo **periodic** deve executar uma vez a cada período de tempo especificado. Cada uma destas execuções deverá ser realizada em uma *thread*. Assim, após executar o método do objeto-base, a *thread* corrente programa a ativação futura deste método para o próximo período em uma nova *thread*. A seguir, a *thread* corrente "morre"; a nova *thread* deverá ser ativada pela *thread* Clock no instante para a qual foi programada.

4.3.1.4. *thread* Libera Próximo Pedido

Todas as vezes em que um método do objeto-base acaba sua execução, é chamado um método do meta-objeto Scheduler para liberar o próximo pedido na Fila de Escalonamento. Este método é denominado *Libera_Próximo_Pedido()* e deverá executar em uma *thread* própria, com o único objetivo de liberar a *thread* chamadora.

4.3.2. Atribuição de Prioridades às *threads*

O uso de *multithreading* cria um contexto de execução muito mais amplo que a utilização de um único fluxo de controle. Além das preocupações com questões de sincronização e exclusão mútua, para que as aplicações executem corretamente é necessário que se dê especial atenção ao escalonamento das *threads* pelo sistema de execução de Java.

Para garantir a dinâmica desejada do Modelo Reflexivo Tempo Real, foi implementado um esquema de prioridades para as *threads* do sistema. Este esquema será descrito a seguir:

• a ***thread*** **Clock** terá prioridade máxima (MAX_PRIORITY), já que o relógio deverá poder preemptar todas as outras *threads* do sistema, quando necessário;

• as ***threads*** **principais** realizarão os pedidos de execução de métodos dos objetos-base com prioridades iguais a prioridade *default* 5 (NORM_PRIORITY) e permanecerão com esta prioridade a não ser quando:

◆ o meta-objeto Scheduler for acessado; neste caso, a prioridade das ***threads*** **principais** cai para 1 (MIN_PRIORITY), de modo a permitir que os pedidos de escalonamento cheguem ao meta-objeto Scheduler e possam ser escalonados junto com outros pedidos. Se esta prioridade não fosse diminuída para 1, os pedidos que chegassem ao meta-objeto Manager seguiriam todos os procedimentos de escalonamento, sem permitir que outros pedidos acessassem o meta-objeto Scheduler.

◆ o fluxo de controle retornar do meta-objeto Scheduler com permissão para execução do método do objeto-base. Neste caso, a prioridade da ***thread*** **principal** sobe para 9, já que, um pedido que passou pelo processo de escalonamento e foi liberado para executar, não pode ser preemptado por qualquer outro pedido. Assim sendo, a prioridade da ***thread*** **principal** torna-se menor apenas que a da ***thread*** **Clock**.

• as ***threads*** **periodic** terão prioridades padrão, iguais a cinco (NORM_PRIORITY), já que atuarão como ***threads*** **principais** depois de sua ativação pelo meta-objeto Clock;

• a ***thread*** **Libera Próximo Pedido** também possuirá prioridade normal, pois sua única função é a de liberar a *thread* chamadora.

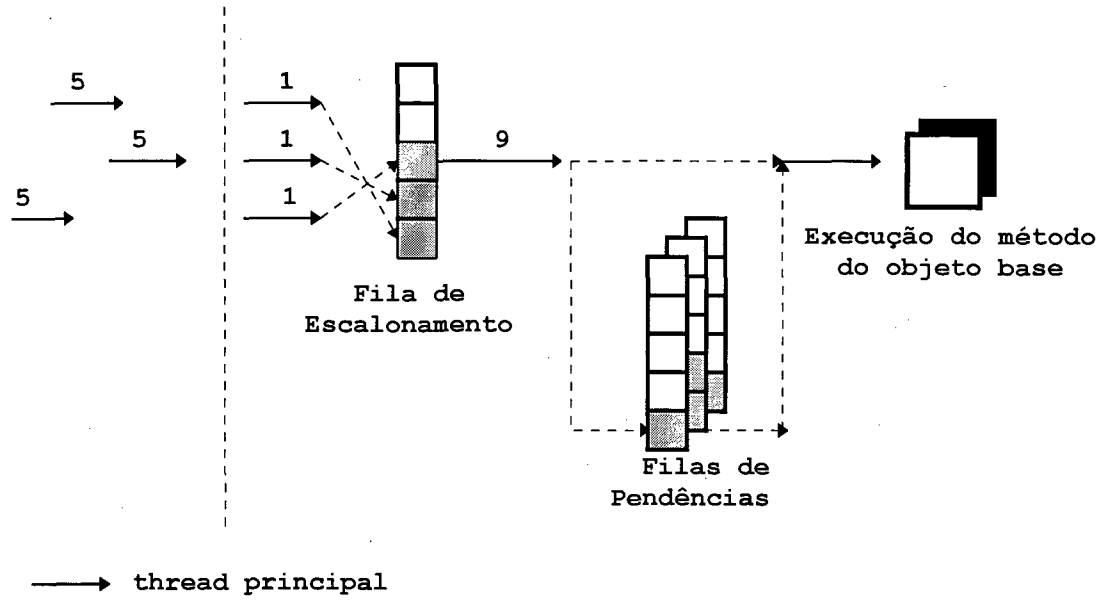


Figura 4.3: Fluxo de execução do modelo RTR Java devido a atribuição de prioridades às threads

4.3.3. Dinâmica resultante do mapeamento

Em uma aplicação, cada chamada a um método de um objeto-base será feita em uma *thread* principal. A partir daí, se desenvolverá a dinâmica de uma chamada a método realizada através do Modelo RTR Java. Assim, com o auxílio da Figura 4.3, analisaremos todos os procedimentos que serão executados nesta *thread* e, a partir dela, toda a dinâmica do sistema.

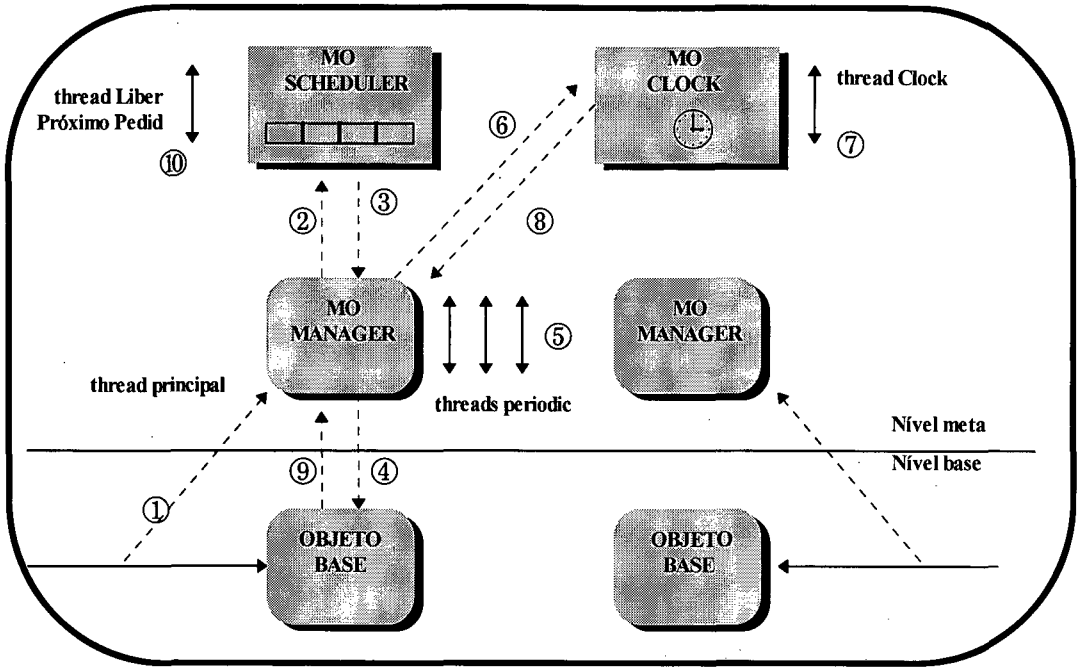


Figura 4.4: Mapeamento do Modelo Reflexivo Tempo Real sobre a linguagem Java.

• Inicialmente, o pedido de execução do método do objeto-base é desviado ao meta-objeto correspondente ①, onde sua restrição temporal é verificada e é realizada a interação da seção de gerenciamento com a de restrições temporais;

• o método da seção de restrições temporais correspondente solicita o escalonamento do método do objeto-base ao meta-objeto Scheduler ②. Antes disso, é atribuída a *thread* principal a prioridade mínima;

• se a Fila de Escalonamento estiver vazia e nenhum outro método do objeto-base estiver sendo executado, o fluxo de controle retorna ao meta-objeto Manager com a permissão para a execução do método solicitado ③; neste caso, a prioridade da ***thread* principal** é aumentada para 9. Por outro lado, se algum método do objeto-base estiver executando neste momento, o pedido solicitado é colocado na Fila de Escalonamento, ordenado segundo a política especificada. Isso deverá ocorrer quando a *thread* executando o método solicitado passar para o estado *not runnable* - por exemplo, se for chamado o método *sleep*). O controle, neste caso, deve retornar ao meta-objeto Manager ③ com a

indicação para suspender a *thread* principal. Esta permanecerá suspensa até que seja a primeira na Fila de Escalonamento - aquela com menor *deadline* - e outra *thread* principal, em execução, sinalizar ao meta-objeto Scheduler a liberação do próximo pedido na Fila de Escalonamento.

• caso a restrição temporal associada ao método do objetos-base seja do tipo periodic, por exemplo, ao retornar do meta-objeto Scheduler com permissão para a execução do mesmo, a *thread* principal verifica novamente se existem condições temporais para execução. Se existirem, as condições de sincronização são verificadas. Se o método possuir um estado de sincronização que lhe permita executar, a *thread* principal retorna à seção de gerenciamento e verifica se ainda há tempo para a execução do método. Se houver, o método do objeto-base é executado ⑫, caso contrário, um método de exceção é sinalizado; após qualquer um desses casos, a seção de gerenciamento do meta-objeto será avisada que o próximo pedido de execução pode ser liberado, indicando o fim de execução deste pedido.

• ainda supondo uma restrição temporal do tipo periodic, após a execução do método do objeto-base, deverá ser feita a programação de sua ativação para o próximo período. Neste caso, é criada uma nova *thread*, que chamamos de ***thread* periodic** ⑤ (item 4.3.1.3). Esta *thread* será colocada na Fila de Pedidos Futuros ⑥ até que, no momento determinado (início do próximo período), o relógio a retire da fila ⑦ e sinalize ao meta-objeto Manager sua iniciação ⑧. Esta nova *thread* deverá chamar o método *periodic()* da seção de restrições temporais, para que a execução no novo período tenha início, configurando, desta forma, a execução cíclica do método.

• se o método solicitado não possuir restrições temporais, deverá, mesmo assim, ser submetido à seção de gerenciamento do meta-objeto, e tratado pelo método PedidoSRT(...) - Pedido sem restrição temporal. Para fins de escalonamento, este pedido possuirá sempre um valor de *deadline* "infinito", que corresponderá ao maior valor possível para os inteiros *long*, em Java.

• após a execução do método do objeto-base, o fluxo de controle retorna ao meta-objeto Manager ⑨ onde é sinalizada ao meta-objeto Scheduler a liberação do próximo pedido na Fila de Escalonamento, através de uma chamada ao método Libera_Proximo_Pedido(). Para executar LiberaProximoPedido() é criada uma nova *thread* ⑩, com a finalidade de realizar uma chamada assíncrona a este método.

4.4. Sumário

Este capítulo abordou as principais características da linguagem Java, ressaltando aquelas mais relevantes a nossa aplicação, como o suporte a *multithreading* a nível de linguagem. A seguir, descrevemos o mapeamento do Modelo RTR sobre a plataforma Java de execução. A dinâmica deste mapeamento foi estudada detalhadamente, os tipos de *threads* e a atribuição de prioridades a estas foram definidos a fim de demonstrar que os requisitos exigidos pelo Modelo RTR podem ser satisfeitos através do Modelo RTR Java.

Capítulo 5

Validação do Modelo RTR/Java

Este capítulo tem como objetivo abordar aspectos específicos da implementação do Modelo Tempo Real e descrever algumas soluções adotadas para alcançar a dinâmica de funcionamento desejada. Além disso, são descritos alguns testes realizados sobre a implementação final, que objetivam verificar as potencialidades do modelo sob aspectos funcionais e de expressividade na representação de restrições temporais e manipuladores de exceções. Ainda neste capítulo, é feita uma breve especificação do modelo, onde informações que visam tornar acessível a sua utilização são fornecidas.

5.1. Aspectos referentes à implementação

O Modelo RTR foi implementado na linguagem Java, versão 1.02 beta, sobre a plataforma de execução Solaris © Sun Microsystems, versão 2.3. A implementação foi testada e executa sem quaisquer alterações sobre o sistema operacional Windows 95.

Este programa foi implementado como uma aplicação Java (*stand alone application*) e, portanto, possui um método *main()* responsável pela configuração de um contexto de execução que será analisado posteriormente.

Os desvios das chamadas aos objetos-base para seus respectivos meta-objetos, inerentes à arquitetura reflexiva, foram feitos através de chamadas a métodos dos meta-objetos, visto que a liguagem Java não dispõe de recursos para reflexão em sua versão 1.02. Considerando-se que uma chamada a método feita em Java (utilizando-se computadores

pessoais de alto desempenho ou *workstations*) requer aproximadamente 1.7 mseg [Sun95c], pode-se considerar o tempo de desvio para o nível meta desprezível.

Nesta seção descreveremos alguns aspectos relativos à implementação.

5.1.1. Executando um procedimento em uma *thread* específica

A utilização de recursos de *multithreading* é adequada a várias aplicações, inclusive às aplicações de tempo real, pois possibilita um melhor aproveitamento dos recursos disponíveis no sistema.

No caso da implementação do Modelo RTR, a criação de *threads* seguiu um padrão de desenvolvimento, fazendo com que todas as classes que utilizassem novas *threads*, implementassem a interface *Runnable*.

A classe Clock, por exemplo, necessita que a Fila de Pedidos Futuros seja verificada em intervalos de tempo regulares, simultaneamente as outras atividades do modelo. Para imprimir a concorrência desejada, esta função é executada em uma *thread* própria. Assim, foi criada a ***thread supervisora***, cujo corpo faz uma chamada ao método *EfetuaAtivação()*, responsável pela verificação da fila. A Figura 5.1 ilustra como isto foi feito.

É importante ressaltar que a definição do método *run()* não suporta parâmetros. Isto obriga a criar métodos para fixar atributos de classe que sejam acessíveis ao corpo da *thread*.

Em todas as outras situações onde foi necessária a criação de *threads*, o procedimento utilizado foi análogo ao descrito anteriormente.

```
class Clock implements Runnable {
      Thread supervisora;
            public void ativaclock() {
                  supervisora = new Thread();
                  supervisora.start();
            }
            public void run() {
                  while(supervisora != null) do {
                        EfetuaAtivacao();
                        try {
                              superv.sleep(10);
                        }catch(InterruptedException e) {}
                  }
            }
            ...
}
```

Figura 5.1. Criação de uma thread para execução de um método específico.

Quando uma *thread* é criada, após sua inicialização pelo método *start()* têm-se, automaticamente, a continuação da *thread* chamadora. Assim, pode-se fazer chamadas assíncronas em Java colocando o método que se deseja chamar assincronamente, dentro do corpo da *thread*. No caso desta implementação, uma classe denominada *Intermediária* foi criada para realizar as funções referentes a uma chamada assíncrona.

5.1.2. O algoritmo de escalonamento

O algoritmo de escalonamento implementado foi o *Earliest Deadline First* [Bur90] que ordena as tarefas segundo seus *deadlines*, de modo que as tarefas com menor *deadline* sejam executadas primeiro.

No caso do Modelo RTR, os pedidos que chegam ao meta-objeto Scheduler são ordenados no instante de sua inserção na Fila de Escalonamento.

Para prover os serviços de manipulação de filas, foi construída uma classe auxiliar chamada classe Fila, com métodos de inserção de objetos, remoção, procura, etc. As filas utilizadas na implementação - Fila de Pedidos Futuros, Fila de Escalonamento e Fila de Pendências - são manipuladas através destes métodos.

5.1.3. Tratamento de chamadas a métodos aninhados

Foram denominados métodos aninhados aqueles métodos cuja execução será controlada pelo meta-objeto *Manager* e que são chamados a partir de outro método com execução também controlada por um meta-objeto. A Figura 5.2 ilustra uma chamada de met1() ao um método aninhado met2().

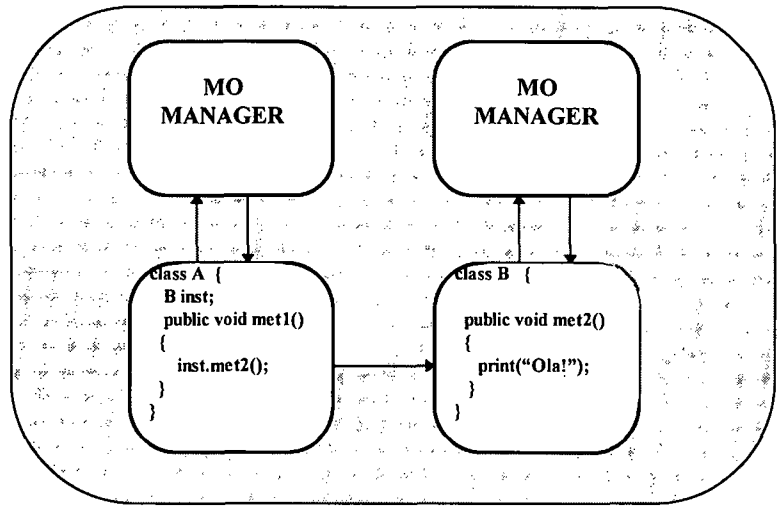


Figura 5.2. Exemplo de uma chamada a um método aninhado

Os pedidos de execução de métodos aninhados deverão passar pelo meta-objeto Scheduler e, sendo liberados, não podem ser bloqueados por questões de sincronização, sob pena de provocarem possíveis *deadlocks*.

As chamadas aninhadas devem ser especialmente tratadas. A nível de meta-objeto Scheduler, é utilizada uma fila que deverá conter os nomes das *threads* já liberadas pelo escalonador. Se o método em questão for solicitado em uma *thread* que já foi liberada (se for um método aninhado), é forçada a liberação da primeira *thread* na Fila de Escalonamento,

que poderá ser a chamada corrente ou uma mais prioritária que se encontre na fila. Isso é feito tanto para chamadas de métodos de um mesmo objeto quanto para chamadas aninhadas de objetos diferentes.

Na seção de gerenciamento de cada meta-objeto *Manager* é introduzido um contador de chamadas aninhadas. Este contador tem como objetivo evitar que um método aninhado, quando termine de executar, libere o próximo pedido na Fila de Escalonamento, ao invés de retomar o controle ao método chamador. Assim, sempre que o contador registrar chamadas aninhadas a uma determinada *thread*, não é liberado o próximo pedido na Fila de Escalonamento.

É importante observar que os métodos aninhados têm que apresentar algum tipo de restrição temporal. Se um método aperiódico que foi liberado para execução, segundo seu *deadline,* fizer uma chamada a um método aninhado sem restrição temporal, a execução do método chamador ficará comprometida e o resultado será inconsistente. Além disso, os *deadlines* dos métodos aninhados devem ser contados como tempo de execução de seus métodos chamadores.

5.1.4. Restrições Temporais

Os métodos que implementam as restrições temporais, são responsáveis por interagir com o meta-objeto Scheduler a fim de solicitar o escalonamento do método do objeto-base. Além disso, devem verificar questões de sincronização e concorrência, e interagir com a seção de exceções temporais, caso o método do objeto-base não esteja apto a executar.

Foram implementados três tipos de restrições temporais: aperiodic, periodic e start_at (restrição de início de execução), que estarão incorporadas a implementação como opções para o usuário. Diferente de outros modelos para tempo real como RTC++ [Ish90], onde as restrições temporais são expressas através de declarações como *within*, *at* e *before*, no Modelo RTR o usuário poderá dispor de maior flexibilidade, podendo implementar as

restrições que forem mais adequadas a sua aplicação. Poderão, inclusive, ser utilizadas as restrições temporais disponíveis, na novas restrições implementadas.

As restrições anteriormente citadas encontram-se descritas nas Figuras 5.3, 5.4 e 5.5.

5.1.4.1. Restrição temporal aperiodic

Os métodos com restrição temporal aperiodic devem ser executados dentro de seus respectivos *deadlines*. Assim, o método Aperiodic(...), solicita o escalonamento do método do objeto-base e verifica suas condições de sincronização, além das condições temporais para que o mesmo seja executado.

```
public void Aperiodic(parâmetros)  {
        // Se o pedido não for liberado pelo escalonador,
        // suspende a thread corrente
        if(!sched.Escalona(metodo solicitado)) then
                (id_thread).suspende();
        // Se ainda há tempo para a execução...
        if(deadline > (tempo_atual + tme)) then ...{
                // Se não puder liberar o pedido por questões de
                // sincronização, suspende a thread corrente.
                if(!LiberaPedidodeAtivacao()) then
                        (id_thread).suspende();
                // Se ainda há tempo para execução do método...
                if(deadline > (tempo_atual + tme)) then
                        exec(metodo); // Executa o método solicitado
                else
                        excecao(id_excecao);  // Sinaliza exceção
                FimdeExecucao();
        }
        eise  {
                excecao(id_excecao);
                LiberaProximoPedido();
        }
}
```

Figura 5.3. Restrição temporal aperiodic

5.1.4.2. **Restrição temporal periodic**

Um método com restrição temporal periodic deve executar uma vez a cada período de tempo previamente especificado. O método é executado e, em seguida, é programada sua ativação para o próximo período. E, assim, sucessivamente até que o instante de fim de execução especificado seja alcançado.

```
public void Periodic(parâmetros)  {
        // Se o pedido não for liberado pelo escalonador, suspende esta thread
        if(!sched.Escalona(método solicitado)) then
                (id_thread).suspende();
        // Se há tempo para execução do método solicitado...
        if((inicio + periodo) > (tempo_atual + tme)) then  {
                // Se não puder liberar o pedido por questões de
                // sincronização, suspende a thread corrente
                if(!LiberaPedidodeAtivacao(método solicitado))then
                        (id_thread).suspende();
                // Se ainda há tempo para execução...
                if((inicio + periodo)>(tempo_atual + tme)) then {
                        exec(metodo); // Executa o método solicitado
                        inicio = inicio + periodo;
                        deadline = inicio + periodo;
                        if(inicio < fim) then        {
                                // Programa ativação do próx. período
                                relogio.ProgramaAtivacao(newthread);
                        }
                }
                else
                        excecao(id_excecao);  // Sinaliza exceção
        FimdeExecucao();
        }
        else    {
                excecao(id_excecao);
                LiberaProximoPedido();
        }
}
```

Figura 5.4. Restrição temporal periodic

O pseudo-código na Figura 5.4, ilustra como foi implementada a restrição temporal periodic:

5.1.4.3. **Restrição temporal start_at**

Quando a restrição temporal em questão for **start_at**, o método é executado uma única vez, mas com início de execução programado para um instante de tempo futuro. A princípio, é verificado se o tempo de início de execução do método é maior que o tempo no instante de verificação; se for, sua ativação é programada para o tempo especificado e a *thread* corrente é suspensa. Caso contrário, o tratamento é análogo a um método com restrição aperiodic.

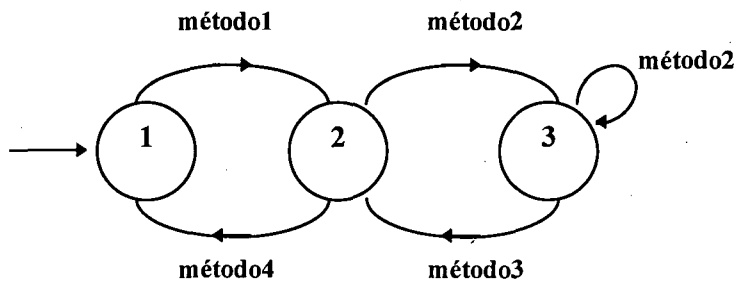
```
public void Start_at(parâmetros)  {
        //Se o tempo de início solicitado for maior que o tempo atual, a ativação
        //é programada para o tempo especificado e a thread suspensa
        if(inicio > tempo_atual)  then {
                relogio.ProgramaAtivacao(método solicitado);
                (id_thread).suspende();
        }
        if(!sched.Escalona(método solicitado)) then
                (id_thread).suspende();
        if(deadline > (tempo_atual + tme)) then {
                if(!LiberaPedidodeAtivacao(método solicitado))
                        (id_thread).suspende();
                if(deadline > (tempo_atual + tme)) then
                        exec(id_metodo);  //Executa o método solicitado
                else
                        excecao(id_excecao);  // Sinaliza exceção
                FimdeExecucao();
        }
        else    {
                excecao(id_excecao);
                LiberaProximoPedido();
        }
}
```

Figura 5.5. Restrição temporal start_at.

5.1.5. A seção de sincronização

As execuções dos métodos de um objeto-base podem estar sujeitas a restrições de sincronização. Por exemplo, podemos estipular que a execução de um método A deverá ocorrer somente após a execução de um método B.

As restrições de sincronização são expressas através de autômatos finitos, que são representados por matrizes. Uma variável é utilizada para indicar o estado de sincronização do método solicitado. A Figura 5.6 mostra um exemplo de um autômato de sincronização e sua matriz de estados.



	método1	método2	método3	método4
estado1	2	-1	-1	-1
estado2	-1	3	-1	1
estado3	-1	3	2	-1

Figura 5.6: Autômato de sincronização e sua matriz de estados

Na matriz da Figura 5.6, os valores iguais a -1 representam um estado de sincronização ilegal.

Três métodos são responsáveis pela sincronização na execução dos métodos de um objeto-base e, para isso, verificam a matriz de estados que expressa estas restrições. Os métodos da seção de sincronização são:

- **`public boolean verificaSincronização(string metodo)`** ➪ responsável pela verificação das condições de sincronização do método solicitado. Retorna *true* se o método puder executar e *false*, caso contrário.

- **`public void atualizaSincronização(string metodo)`** ➪ atualiza os valores da variável estado_de_sincronização após a execução de um método do objeto-base.

- **`public int proximo()`** ➪ procura o próximo método na Fila de Pendências com estado de sincronização "ok". Retorna a posição do método na fila.

5.2. Testes e Resultados

Alguns testes foram realizados para verificação das funcionalidades do Modelo RTR. Estes testes são divididos em três classes: testes de funcionalidade, testes de expressividade e testes dos mecanismos de tratamento de exceção.

Uma classe que provê métodos para animação gráfica foi implementada como aplicação. Esta classe está disponível no Anexo A e deve possibilitar a exibição de um *frame* para suportar a animação, bem como a exibição de um único quadro de imagem ou de trechos de animação na tela.

Os arquivos de imagem suportados pela linguagem Java em sua versão 1.02 podem ser de formatos .gif ou .jpg. No caso desta implementação, foram utilizados arquivos no formato .gif.

Os testes foram realizados na classe Real Time do sistema operacional Solaris (Anexo B), que possui maior prioridade em relação aos processos Daemons do sistema e aos processos UNIX comuns.

A seguir, descreveremos alguns dos testes realizados sobre o Modelo RTR:

5.2.1. Testes de funcionalidade do Modelo

Os testes funcionais foram realizados para verificar aspectos básicos da implementação do modelo, além de sua correção lógica e temporal.

- **Teste funcional do mecanismo de escalonamento**

Este teste verifica se o esquema de prioridades implementado permite que os pedidos de execução de métodos sejam escalonados pelo sistema.

Segundo descrito no Capítulo 4, item 4.2.2, a prioridade das *threads* que chegam ao meta-objeto Scheduler são diminuídas para 1 (MIN_PRIORTY) permitindo que os pedidos lançados em instantes próximos cheguem juntos ao escalonador.

Cinco métodos aperiódicos são chamados em ordem aleatória de seus *deadlines*. As ações realizadas pelos métodos limitam-se a exibição de mensagens na tela do tipo "Executando método x". Uma mensagem também é exibida quando a Fila de Escalonamento é utilizada para armazenar os pedidos ordenados.

Os métodos com seus respectivos *deadlines* são solicitados na seguinte ordem:

1- metodo1, *deadline* = 100 ms;
2- metodo2, *deadline* = 90 ms;
3- metodo3, *deadline* = 70 ms;

4- metodo4, *deadline* = 80 ms;

5- metodo5, *deadline* = 60 ms;

Um dos resultados obtidos é mostrado a seguir:

**** Executando metodo2!!!*
Vou inserir na FiladeEscalonamento o metodo1
Vou inserir na FiladeEscalonamento o metodo5
Vou inserir na FiladeEscalonamento o metodo3
Vou inserir na FiladeEscalonamento o metodo4
Vou reassumir o metodo5
**** Executando metodo5!!!*
Vou reassumir o metodo3
**** Executando metodo3!!!*
Vou reassumir o metodo4
**** Executando metodo4!!!*
Vou reassumir o metodo1
** Sinalizando exceção para o metodo1*

É importante notar que os métodos com menor *deadline* foram reassumidos e executaram primeiro, conforme o esperado.

- **Teste dos mecanismos de sincronização**

Para testar os mecanismos de sincronização do Modelo RTR, foi implementado o problema Produtor x Consumidor. Neste problema, o produtor é responsável por dispor de determinado artigo e por depositá-lo num *buffer* limitado; o consumidor, por sua vez, retira os artigos deste *buffer*. As restrições de sincronização consistem em respeitar os limites do *buffer*, não permitindo que o produtor insira mais artigos do que é suportado, nem permitindo que o consumidor os retire se o *buffer* estiver vazio.

A Figura 5.7 ilustra o mapeamento do problema Produtor x Consumidor sobre o Modelo RTR. Tem-se três objetos-base: produtor, consumidor e *buffer*. A classe Produtor possui um método denominado produz(), que faz uma chamada ao método insere(), da classe Buffer. A classe Consumidor, por sua vez, possui um método denominado consome(), que chamà retira(), também da classe Buffer. As questões de sincronização em relação ao objeto *buffer* têm seu gerenciamento feito pelo meta-objeto *meta_buffer*. Neste caso, as ações do produtor e do consumidor não possuem restrições temporais (pedido_SRT). As restrições de sincronização são armazenadas numa matriz (variável autômato_finito).

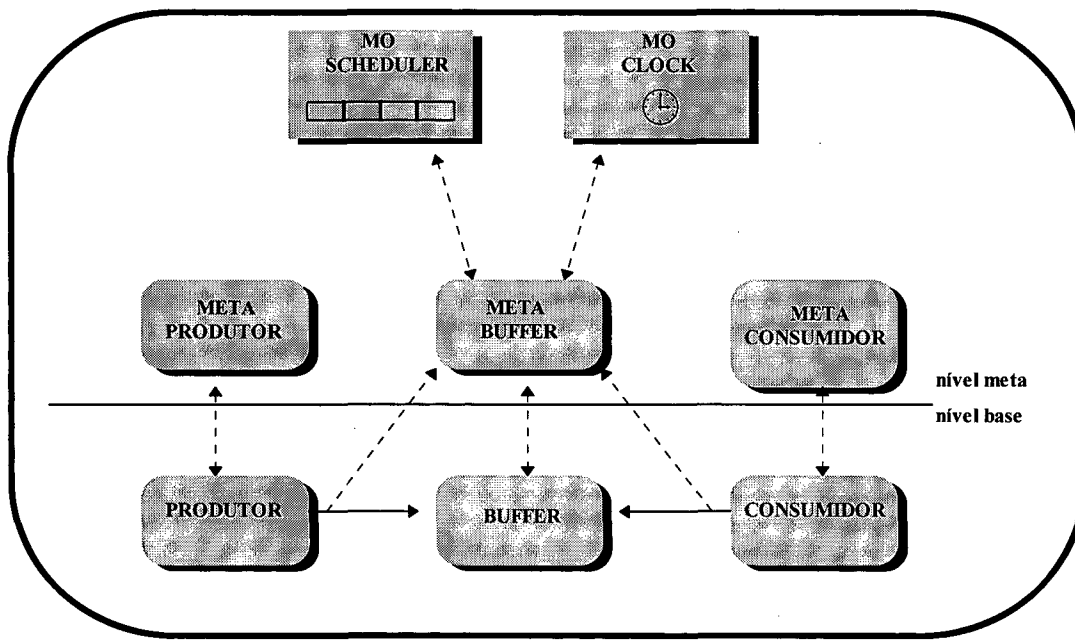
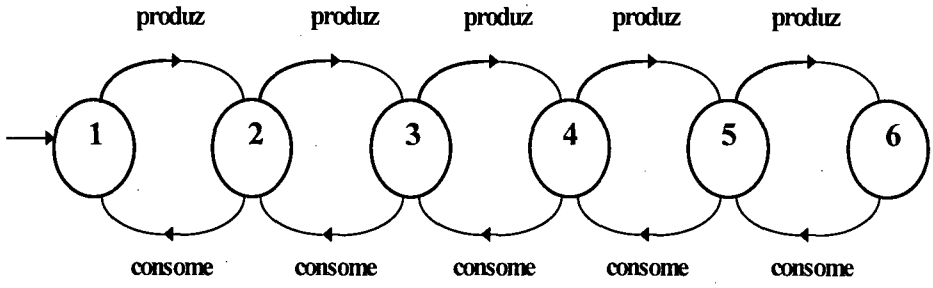


Figura 5.7. Mapeamento do problema Produtor x Consumidor sobre o Modelo RTR

O autômato que expressa as restrições de sincronização sobre o objeto *buffer* e a matriz de estados resultante são mostrados na Figura 5.8. Os valores iguais a -1 representam estados de sincronização ilegais. Neste exemplo, o *buffer* é limitado em cinco posições, assim, são seis os estados que compõem o autômato; as transições representam as ações de produzir e consumir.



	estado 1	estado 2	estado 3	estado 4	estado 5	estado 6
produz	2	3	4	5	6	-1
consome	-1	1	2	3	4	5

Figura 5.8. Autômato das restrições de sincronização do problema produtor x consumidor e matriz de estados correspondente.

Os pseudo-códigos das classes Produtor, Consumidor e Buffer são mostrados nas Figuras 5.9, 5.10 e 5.11.

```
class Produtor {

        public void produz() {
                Intermediaria inst1 = new Intermediaria();
                // Faz uma chamada assíncrona ao método
                // insere
                inst1.assinc(manag,' insere', 'pedidoSRT');
        }
}
```

Figura 5.9. Pseudo-código da classe Produtor

```
class Consumidor {

        public void consome() {
    Intermediaria inst2 = new Intermediaria();
        // Faz uma chamada assíncrona ao método
        // retira(), da classe Buffer
        inst2.assinc(manag, 'retira', 'pedidoSRT');
        }
}
```

Figura 5.10. Pseudo-código da classe Consumidor.

```
import java.util.Vector;

class Buffer {
        Vector buff = new Vector(5);

                public void insere(Nodo n) {
                        buff.adiciona(new Character('x'));
                        exibe("Produtor: "+ bufft);
                }

                public void retira(Nodo n) {
                        buff.retira(new Character('x'));
                        exibe("Consumidor: "+ buff);
                }
}
```

Figura 5.11. Pseudo-código da classe Buffer.

A matriz de sincronização, mostrada na Figura 5.8, será representada pela variável autômato_finito como:

static int automato_finito[][] = {{1, -1}, {2, 0}, {3, 1}, {4, 2}, {5, 3}, {-1, 4}};

Um dos possíveis resultados desta aplicação, supondo-se que os artigos produzidos/consumidos são iguais a "x" e o *buffer* limitado por colchetes, é:

Produz - [x]

Produz - [x x]

Consome - [x]

Consome - []

Consumidor aguardando...

Produz - [x]

Consumidor liberado!

Consome - []

5.2.2. Testes da expressividade do Modelo

Os testes de expressividade mostram a eficiência do modelo em mapear situações comumente apresentadas no domínio tempo real.

- **Testes da restrição temporal aperiodic**

A restrição temporal **aperiodic** foi testada na implementação dos *frames* para suportar as animações gráficas. Estes *frames* são implementados pela classe *java.awt.Frame* e constituem-se de uma janela gráfica, que segue o padrão de interface do sistema suporte. O objeto de animação criado foi chamado de forma síncrona a partir do método main().

Outro teste com métodos de restrição temporal **aperiodic** foi citado no item 5.2.1, neste caso, foram feitas cinco chamadas a métodos aperiódicos para demonstrar o funcionamento do esquema de prioridades adotado.

A Figura 5.12 mostra o *frame* criado a partir de uma chamada aperiódica e um dos quadros de imagem utilizados.

Figura 5.12: Frame de animação e um dos quadros de imagem utilizados.

A exibição de um quadro de imagem na tela, através do Modelo RTR, é realizado em aproximadamente 20 ms, quando executando na classe Real Time Solaris. Levando-se em conta que uma animação deve apresentar em torno de 24 quadros por segundo -

aproximadamente um quadro a cada 40 ms - para uma animação de boa qualidade [Sun95b], o resultado obtido é considerado satisfatório e leva a crer que a dinâmica envolvida no modelo não causa um *overhead* tal que inviabilize a utilização de aplicações como estas.

- **Testes da restrição temporal periodic**

Para testar a restrição temporal periodic, criamos dois objetos de animação. Cada um deles executa um método que tem como função exibir um quadro de imagem na tela. Este método é chamado de forma periódica e, em cada período, exibe um quadro na sequência de animação. Os dois objetos executam simultaneamente.

Este teste foi realizado com métodos **periodic** com ativação **event trigger** e **time trigger**. Nos dois casos, os resultados são semelhantes e mostram dois objetos de animação executando simultaneamente.

Uma variação da restrição temporal **periodic** que prevê um início de execução programado, foi implementado e testado sob o nome de **periodic_at**. Neste caso, é solicitada a execução periódica do método para um tempo futuro, a restrição temporal verifica qual o instante indicado para ativação do método e programa sua ativação. O relógio do sistema ativa o método do objeto-base, reassumindo a *thread* correspondente, no instante para o qual foi programada.

Um terceiro teste realizado com métodos chamados de forma periódica, visa verificar o funcionamento do sistema em situações de carga, além de verificar o funcionamento do método de restrição periódica em relação a finalização da execução. Para isso, são criados nove objetos de animação que executam simultaneamente e exibem, como anteriormente, um quadro de imagem a cada período de execução. Aos três primeiros objetos de animação, é atribuído um tempo para finalização t1, aos três objetos seguintes, é atribuído um tempo t2 e aos três últimos, um tempo t3, sendo que $t1 < t2 < t3$. Podemos verificar que, conforme os objetos finalizam sua execução, os objetos restantes adquirem um melhor desempenho da animação, como era esperado.

- **Teste da restrição temporal start_at**

Para verificar a implementação da restrição temporal start_at, foram criados dois objetos de animação. O primeiro deles executa uma animação de forma periódica; o segundo, executa três métodos com restrição temporal start_at que exibem o quadro 1 no instante igual a 5 seg, o quadro 7 no instante igual a 10 seg, e o quadro 15 no instante igual a 20 seg.

5.2.3. Testes dos mecanismos de tratamento de exceção

Para demonstrar os mecanismos de tratamento de exceção foram implementados dois testes. Em cada um deles, é dado um tratamento diferente às tarefas cujo *deadline* não pôde ser cumprido. Os objetivos destes testes são, além de verificar o funcionamento correto do modelo implementado, mostrar sua flexibilidade e a forma bastante acessível com que podemos associar os métodos de exceção à execução dos métodos dos objetos-base.

Ambos os testes são realizados com animações gráficas. Um método que exibe um quadro de imagem é chamado com restrição temporal **periodic**. Em cada período, um quadro que compõe a animação é exibido. Quando um método não consegue executar dentro do *deadline* especificado, um método de exceção é sinalizado.

No primeiro método de exceção implementado, o quadro de imagem que não consegue ser exibido é descartado e o próximo quadro na sequência da animação é programado para o próximo período. Como nos pedidos periódicos, o próximo período será inicializado pelo relógio no instante para o qual foi programado.

Como resultado, todas as vezes em que um quadro não puder ser exibido, o próximo quadro na sequência de animação será mostrado. Quanto menor o deadline estipulado para execução do método, maior a probabilidade de falhas na sequência de animação. A Figura 5.13 mostra um exemplo de execução; no caso, devido a sinalização de exceções durante a

animação, uma sequência entrecortada de imagens é exibida, sendo mostrada, por exemplo, a sequência 1, 7 e 11.

Figura 5.13. Exemplo de uma animação na qual exceções que suprimem quadros de imagem foram sinalizadas.

No segundo método de exceção testado, o quadro de imagem que não consegue ser exibido é mantido e programado para executar no próximo período. Assim, não existirão quebras na sequência de animação, mas sim um "congelamento" da imagem cujo deadline foi perdido. Este método, em relação ao primeiro, é o mais comumente usado em aplicações multimídia. A Figura 5.14 mostra um exemplo de execução, no qual, mesmo tendo sido sinalizadas exceções durante a animação, é mostrada a sequência de imagens em sua ordem correta. Assim, quanto maior o número de exceções sinalizadas entre uma execução e outra, maior o tempo em que um quadro de imagem permanecerá na tela.

Figura 5.14. Exemplo de animação na qual exceções que mantém o quadro de imagem na tela são sinalizadas.

5.3. Considerações sobre a Realização dos Testes

Os testes apresentados foram realizados para verificar as funcionalidades do Modelo RTR. Alguns aspectos do modelo puderam ser diretamente mapeados sobre a linguagem Java, o que resultou em simplicidade para o código implementado.

Os testes funcionais verificaram os aspectos básicos do modelo, como sua adequação à dinâmica proposta, envolvendo múltiplas *threads* de controle. Neste sentido, o mapeamento descrito no capítulo 4 mostrou-se eficiente ao prever e evitar problemas de sincronização e de exclusão mútua aos dados compartilhados, permitindo o escalonamento e priorizando a execução dos métodos liberados pelo escalonador de forma eficiente e coerente com os objetivos do modelo.

Estes testes permitiram demonstrar a simplicidade do modelo e sua flexibilidade na representação de restrições temporais típicas ao cenário tempo real. As restrições de periodicidade, aperiodicidade e restrição de início de execução foram implementadas com total liberdade de expressão. O caráter flexível do Modelo RTR é tornado ainda mais evidente quando as restrições implementadas são utilizadas para criar novas restrições, como foi o caso da criação de uma restrição temporal de periodicidade com restrição de início de execução (*periodic_at*) criada a partir da restrição de periodicidade com início de execução imediato.

A mesma flexibilidade apresentada na expressão de restrições temporais também está presente na expressão de métodos manipuladores de exceções. Os dois métodos implementados, apresentados na seção anterior, demonstram que o mecanismo de tratamento de erros oferece inúmeras possibilidades aos usuários. No caso da aplicação em questão, poderiam ser implementados, ainda, métodos manipuladores de exceções que abortassem o objeto cujo método não pôde alcançar seu *deadline*, por exemplo.

O teste dos mecanismos de sincronização apresentado mostrou que estabelecer relações de precedência na execução de métodos de um objeto-base é uma tarefa simples,

que consiste na definição de uma matriz a partir do autômato representando as restrições de sincronização. Estes mecanismos auxiliam em muito a programação de aplicações multimídia.

Por fim, através dos testes realizados pode-se afirmar que o Modelo RTR garante realmente, que as perdas de *deadline* dos métodos de uma aplicação serão sempre detectados e tratados da forma especificada pelo usuário.

O conjunto de funcionalidades do modelo possibilita a representação de inúmeros cenários típicos de tempo real, com grande flexibilidade, organização e eficiência. A técnica reflexiva utilizada não resultou num *overhead* tal que influenciasse a execução das aplicações, por outro lado, os benefícios desta técnica viabilizaram muitas das potencialidades do modelo.

5.4. Especificação do Modelo RTR Implementado

Esta seção objetiva viabilizar a utilização do Modelo RTR implementado em Java, através de sua especificação, onde são detalhados os procedimentos necessários ao seu funcionamento, como a geração de um contexto de execução, a implementação de reflexão e a definição do algoritmo de escalonamento.

5.4.1. Gerando o contexto de execução

Esta implementação foi realizada sob a forma de uma aplicação Java (*stand alone application*) e, portanto, possui um método main() responsável pela configuração de um contexto de execução. No caso do Modelo RTR Java, o método main() deverá ser responsável pela criação das instâncias das classes Clock, Scheduler e da meta classe auxiliar LiberaProxPedido. Os objetos-base poderão ser criados, também, a partir deste método ou a partir de outras classes base, conforme as necessidades da aplicação. A criação dos meta-objetos referentes a cada objeto-base criado deverá ser feita explicitamente. Cada

meta objeto deverá "conhecer" seu objeto-base correspondente, bem como as instâncias das Classes Clock, Scheduler e LiberaProxPedido, sendo que todos deverão ser passados como parâmetro para o construtor do meta objeto.

A Figura 5.14, ilustra um exemplo do que poderia ser o método main() do modelo.

```
public static void main() {
        Clock relogio = new Clock();
        Scheduler sched = new Scheduler(relogio);
        LiberaProxPedido lib = new LiberaProxPedido(sched);
        relogio.setsched(sched);
        ...
}
```

Figura 5.15. Configuração de um contexto de execução.

5.4.2. Reflexão no Modelo RTR implementado

A reflexão computacional no modelo implementado é feita através de chamadas ao método RecebePedido(...) dos meta-objetos. O método RecebePedido é responsável pelo recebimento dos pedidos de execução de métodos do objeto-base. Estes pedidos podem ser feitos de forma síncrona ou assíncrona.

Para solicitar a execução de um método do objeto-base de forma síncrona, deve-se fazer uma chamada ao método RecebePedido do meta-objeto correspondente àquele objeto-base. Uma instância da classe Nodo deverá englobar os atributos do método solicitado, como tipo de restrição temporal, *deadline*, método de exceção, entre outros, e será passada como parâmetro. A Figura 5.16 ilustra uma chamada síncrona a um método aperiódico.

```
Nodo no = new  Nodo();
no.setvalue(thread_corrente, meta_id, inst_ativacao, deadline,
            'metodo', tme,'exc', 'aperiodic', termo_ord);
meta_id.RecebePedido(no);
```

Figura 5.16. Exemplo de chamada síncrona.

Os parâmetros do método *setvalue(...),* na Figura 5.16, são:

thread_corrente: identificador da thread corrente;
meta_id: identificador do meta objeto correspondente;
inst_ativacao: tempo atual;
deadline: valor do deadline especificado;
metodo: identificador do método solicitado;
tme: tempo máximo de execução do método solicitado;
exc: identificador do método de exceção correspondente ao método solicitado;
aperiodic: tipo da restrição temporal
termo_ord: variável que será utilizada para ordenação da Fila de Escalonamento. Será igual ao *deadline*, se a política adotada for *Earliest Deadline First*.

O método *setvalue(...)* é sobrecarregado para cada tipo de restrição temporal em questão, e suas possíveis sintaxes são definidas na classe Nodo (anexo A).

No caso de uma chamada assíncrona, uma classe auxiliar denominada Intermediária é responsável pela criação de uma *thread* específica (*thread* principal) para acessar o nível meta. Neste caso, a classe Intermediária se encarrega de agrupar os atributos do método num objeto do tipo Nodo. O método da classe Intermediária utilizado para criar chamadas assíncronas é denominado *assinc(...)*. A Figura 5.17 ilustra a realização de uma chamada assíncrona a um método aperiódico.

```
Intermediaria interm = new Intermediaria();
interm.assinc(meta_id,deadline,tme,'metodo','exc','aperiodic');
```

Figura 5.17. Exemplo de chamada assíncrona.

Na Figura 5.17, os parâmetros do método *assinc(...)* são:

meta_id: identificador do meta objeto correspondente;
deadline: valor do deadline especificado;

metodo: identificador do método solicitado;
tme: tempo máximo de execução do método solicitado;
exc: identificador do método de exceção correspondente ao método solicitado;
aperiodic: tipo da restrição temporal

5.4.3. Alterando o algoritmo de escalonamento

Os pedidos que chegam ao meta-objeto *Scheduler* são ordenados no instante de sua inserção na Fila de Escalonamento, segundo a política de escalonamento adotada. A definição desta política, no caso de algoritmos como o *Earliest Deadline First* ou o algoritmo de Prioridades Fixas, pode ser feita através da variável termo_ord, na classe Nodo. É através desta variável que o método de inserção na Fila de Escalonamento ordenará a execução dos métodos dos objetos-base. No caso de outro algoritmo de escalonamento, o mesmo deverá ser definido no meta-objeto Scheduler.

Esta flexibilidade na alteração das características do Modelo RTR, é também resultado da arquitetura reflexiva adotada. Pode-se notar que, alterar o algoritmo de escalonamento é uma tarefa simples, bastando para isso algumas alterações a nível meta, sem comprometer a aplicação.

5.4.4. Acrescentando uma nova restrição temporal

O Modelo RTR foi implementado com três opções de métodos de restrição temporal. Porém, sua capacidade de expressar restrições temporais é bastante grande, podendo o usuário implementar aquela que lhe for conveniente.

Uma nova restrição temporal pode ser acrescentada ao modelo na seção de restrições temporais do meta-objeto desejado. Além disso, sua chamada deve ser acrescentada ao método RecebePedido(), como uma declaração *case* além das já existentes. A Figura 5.18 ilustra a alteração que deve ser feita no método RecebePedido().

```
public void RecebePedido(parâmetros) {
        switch(restrição) {
                case 'periodic': {
                                        Periodic(parâmetros);
                                        break;
                                }
                case 'aperiodic': {
                                        Aperiodic(parâmetros);
                                        break;
                                }
                case 'start_at':  {
                                        Start_at(parâmetros);
                                        break;
                                }
                case 'nova_restricao': {
                                        Nova_restricao(parâmetros);
                                        break;
                                        }
                }
        }
```

Figura 5.18. Alteração do método RecebePedido(...) ao acrescentarmos uma nova restrição temporal

Além disso, a classe Nodo (Anexo A) deve ser alterada quando forem utilizados novos atributos além daqueles já especificados. A seguir, basta que o código fonte seja novamente compilado para que a nova restrição temporal seja incorporada ao modelo.

É importante salientar que as classes implementadas podem ser utilizadas diretamente pelos usuários, ou estendidas, conforme suas necessidades.

5.5. Sumário

Neste capítulo foram abordados aspectos específicos da implementação do Modelo RTR sobre a linguagem Java, como o suporte de implementação utilizado, os procedimentos adotados na solução de problemas como o tratamento a chamadas aninhadas, entre outros.

Os testes realizados sobre o protótipo implementado foram descritos, a fim de demonstrar as capacidades do Modelo RTR na expressão de situações comuns ao domínio tempo real. Os testes foram realizados sobre uma aplicação gráfica e demonstram a flexibilidade do modelo, além da adequação da linguagem Java a aplicações dessa natureza.

Foi feita uma breve especificação do protótipo implementado, objetivando torná-lo acessível aos usuários. Foram explicitados os procedimentos para a realização de chamadas síncronas e assíncronas, para o acréscimo de restrições temporais, entre outros.

Capítulo 6

Conclusões e Perspectivas

O principal objetivo deste trabalho foi promover a validação de um modelo de programação para aplicações de tempo real, denominado Modelo RTR. Com este intuito, foi feito um estudo detalhado do modelo, a fim de que fossem identificadas suas principais características e dinâmica de funcionamento. A seguir, o modelo foi mapeado sobre a linguagem de programação Java, buscando-se, nesta etapa, identificar as potencialidades que a mesma dispunha para satisfazer as necessidades do Modelo RTR. Por fim, um protótipo foi implementado, seguindo as diretrizes básicas traçadas pelo mapeamento.

O Modelo RTR apresenta em sua estrutura características essenciais para atender ao domínio para o qual é dedicado. De maneira geral, em comparação aos outros modelos citados neste trabalho, o Modelo RTR é o que apresenta maior flexibilidade na representação de restrições temporais, sendo altamente favorecido por adotar uma arquitetura reflexiva. Com isto, questões referentes ao controle de aplicações são solucionadas de forma simples e modular. O conjunto de restrições temporais disponíveis, os algoritmos de escalonamento suportáveis e a linguagem de programação suporte não são questões impostas, mas altamente negociáveis, conforme as necessidades de uma aplicação específica.

Para o mapeamento do modelo, a linguagem Java foi escolhida por questões naturais, já que é objetivo deste trabalho que seus resultados contribuam para a especificação da linguagem Java RTR, que, por sua vez, é estendida a partir de Java. Porém, é importante salientar que Java possui características que facilitaram em muito o mapeamento do modelo. Além de orientada a objetos, Java oferece recursos de *multithreading* a nível de linguagem, o

que permitiu um alto grau de concorrência ao protótipo implementado, característica esta que constitui-se numa das premissas do Modelo RTR. Além disso, sua simplicidade também contribuiu para facilitar o mapeamento. Em relação a reflexão computacional, Java ainda não oferece, em sua versão 1.02, o suporte a implementação. Esta é uma das potencialidades que deverão estar disponíveis em Java a partir de sua versão 1.1.

O protótipo do Modelo RTR foi implementado com algumas funcionalidades básicas, suficientes para validar alguns de seus principais aspectos. Foram implementados o algoritmo de escalonamento *Earliest Deadline First* e as restrições temporais de periodicidade, aperiodicidade e de restrição de início de execução.

Os testes realizados dividiram-se em três classes: testes de funcionalidade, expressividade e de manipulação de exceções. Nos testes de funcionalidade, foram verificadas as características básicas do modelo, como a coerência da dinâmica proposta, envolvendo múltiplas *threads* de controle e as interferências *time trigger* efetuadas pelo meta-objeto Clock, além disso, foram verificados os mecanismos de sincronização na execução de métodos de um mesmo objeto, através do exemplo Produtor x Consumidor.

Os testes de expressividade tiveram o objetivo de verificar as capacidades do modelo na representação de cenários comuns ao domínio tempo real. Assim, foram implementados e testados métodos de restrição temporal de periodicidade, aperiodicidade e restrição de início de execução. A implementação destes métodos foi considerada simples, visto que o modelo não impõe nenhuma restrição neste sentido. A flexibilidade foi constatada, ainda, na criação de uma nova restrição, a partir de uma já existente: a partir da restrição de periodicidade com início imediato, foi criada a restrição de periodicidade com início de execução programado.

Os testes de manipuladores de exceções tiveram como objetivo verificar as potencialidades do modelo no tratamento de erros. Foram implementados dois métodos distintos, utilizados sobre aplicações envolvendo objetos de animação gráfica. Estes testes permitiram verificar a simplicidade na implementação de métodos para o tratamento de exceções temporais.

Como dito anteriormente, grande parte dos testes foi realizada sobre aplicações envolvendo objetos de animação gráfica. Assim, foi possível ter-se uma idéia do desempenho do modelo na programação de aplicações multimídia. Os resultados obtidos foram bastante satisfatórios, verificando-se que toda a dinâmica envolvida no Modelo RTR não dispende um montante de processamento que inviabilize aplicações deste tipo.

Desta forma, conclui-se que os objetivos deste trabalho foram alcançados, apresentando-se como perspectivas futuras de desenvolvimento, as seguintes sugestões:

. O desenvolvimento de um modelo para distribuição deste protótipo, utilizando o IDL Java, seguindo o padrão CORBA. As especificações CORBA (*Common Object Request Broker Architecture*) foram propostas pelo OMG (*Object Management Group*) e têm como objetivo possibilitar a integração de diferentes sistemas de programação baseados em objetos. O uso de padrões CORBA permite que objetos possam interagir num sistema distribuído, independente de suas linguagens de codificação, arquitetura de máquina ou sistemas operacionais. No caso, o IDL é a linguagem que permite especificar a interface com cada objeto.

. A implementação de restrições temporais e de sincronização mais específicas à aplicações multimídia com maior grau de complexidade.

. A implementação da Linguagem Java/RTR, uma extensão de Java realizada a partir da filosofia de programação ditada pelo modelo RTR. A linguagem Java/RTR tem como objetivo viabilizar a utilização do modelo, introduzindo novos tipos de classes que correspondem aos componentes dos níveis base e meta do modelo, permitindo a reflexão automática, além de outras facilidades para a implementação de aplicações sujeitas a restrições temporais.

Referências Bibliográficas

[Ait96] Aitken, G., "***Moving from C++ to Java***", Dr. Dobb's Journal, March 1996.

[Aud90] Audsley, N. et. al., "***Predictability Dependable Computing Systems***", On First Year Report, Task B, of Espirit BRA Project 3092, vol. 2, cap.2, 1990.

[Ber93] Berryman, S. J., ***"Modelling and Evaluating Time Constraints in Real-Time Systems***", Thesis, Lancaster University, March 1993.

[Bla96] Blakowski, G. and Steinmetz, R., "***A Media Synchronization Survey: Reference Model, Specification, and Case Studies***", IEEE Journal on Selected Areas in Communications, vol. 14, n. 1, January 1996, pp. 5-35.

[Bur90] Burns, A. and Audsley, N., "***Real-Time Systems Scheduling***", University of York, 1990.

[Chi93a] Chiba, S. and Masuda, T., "***Designing an Extensible Distributed Language with a Meta-Level Architecture***", Proceedings of 7th European Conference on Object-Oriented Programming (ECOOP'93), Kaiserslautern, July 1993, pp. 482-501.

[Chi93b] Chiba, S., "***Open C++ Programmer's Guide***", Technical Report 93-3, Department of Information Science, University of Tokio, 1993.

[Fab95] Fabre, J., Nicomette, V., Pérennou T., Stroud R. J. and Wu, Z., "***Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming***", Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing, Pasadena (CA), June 1995.

[Far93] Farines, J. M., "***Questões de Tempo Real nos Sistemas Informáticos e Aspectos de sua Modelização***", Universidade Federal de Santa Catarina, 1993.

[Fer94] Ferrari, A. D., "***Projeto e Implementação de um Núcleo de Tempo Real Segundo a Abordagem Síncrona***", Dissertação de Mestrado, Laboratório de Controle e MicroInformática (LCMI), Universidade Federal de Santa Catarina, 1994.

[Fra94] Fraga, J. da S., "***Sistemas Tempo Real***", Notas de aula, Laboratório de Controle e MicroInformática (LCMI), Universidade Federal de Santa Catarina, 1995.

[Fra95] Fraga, J., Farines, J. M., Furtado, O. J. V. and Siqueira, F., "***A Programming Model for Real-Time Applications in Open Distributed Systems***", 5th Workshop on Future Trends in Distributed Computing Systems, Cheju Island, Republic of Korea, August 1995.

[Fur95] Furtado, J. O., "***Um Modelo e uma Linguagem para Aplicações Tempo Real***", Exame de Qualificação ao Doutoramento, Laboratório de Controle e MicroInformática (LCMI)/ EEL/ UFSC. October, 1995.

[Fur96] Furtado, O . and Farines, J. M., "***Java/RTR - Uma Linguagem Reflexiva para Programação de Aplicações Tempo Real***", I SBPL, Belo Horizonte, September 1996.

[Hon94] Honda, Y. and Tokoro, M., "***Reflection and Time-Dependent Computing: Experiences with the R2 Architecture***", Technical Report, SONY C. S. Laboratory Inc., Tokio, July 1994.

[Ish90] Ishikawa, Y., Tokuda, H. and Mercer, C. W., "***Object-Orlented Real-Time Language Design: Constructs for Timing Constraints***", Carnegie-Mellon Technical Report CMU-CS-90-111, March 1990.

[Ish92] Ishikawa, Y., Tokuda, H. and Mercer, C. W., ***"An Object-Oriented Real-Time Programming Language***", Computer, October 1992, pp. 66-73.

[Kim93] Kim, H. and Bacellar, L. F., "***A Real-Time Object Model: A Step toward an Integrated Methodology for Engineering Complex Dependable Systems***", Proceedings of CSESAW'93, US Navy NSWC, July 1993.

[Kim94] Kim, K. H. and Kopetz, H., "***A Real-Time Object Model RTO.k and an Experimental Investigation of its Potentials***", COMPSAC'94, November 1994.

[Kop92a] Kopetz, H., "***Scheduling***", An Advanced Course on Distributed Systems. Estoril, Portugal, July 1992.

[Kop92b] Kopetz, H., "***Real-Time and Real-Time Systems***", An Advanced Course on Distributed Systems. Estoril, Portugal, July 1992.

[Lea96] Lea, D., "***Concurrent Programming in Java. Design Principles and Patterns***", Addison Wesley. October, 1996.

[Liu73] Liu, C. L. and Layland, J. W., "***Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment***", JACM, vol. 20, n.1, January 1973, pp. 46-61.

[Mae87] Maes, P., "***Concepts and Experiments in Computational Reflection***" Proceedings of OOPSLA'87, October 1987, pp. 147-155.

[Mat91] Matsuoka, S., Watanabe, T. and Yonezawa, A ., "***Hybrid group reflective architecture for object-oriented concurrent reflective programming***", in Proceedings of ECOOP, 1991.

[Nil95] Nilsen, K., "***Issues in the Design and Implementation of Real-Time Java***", Iowa State University, Ames, Iowa, 1995.

[Nil96] Nilsen, K., "***Real-Time Java - draft 1.1***", Iowa State University, Ames, Iowa, 1996.

[Ram94] Ramanathan, P. and Shin, K. G., "***Real-Time Computing; A New Discipline of Computer Science and Engineering***", Proceedings of the IEEE, vol. 82, n 1, January 1994, pp. 6-23.

[Sha89] Sha, L. et al, "***Aperiodic Task Scheduling for Hard Real-Time Systems***", The Journal of Real-Time Systems, n. 1, 1990, pp. 27-60.

[Sta88] Stankovic, J. A., "***Misconceptions About Real-Time Computing***", IEEE Computer, vol. 21, n. 10, October 1988.

[Sta94] Stankovic, J. A . and Ramamritham, K. "***Scheduling Algorithms and Operating Systems Support for Real-Time Systems***", Proceedings of the IEEE, vol. 82, n. 1, January 1994, pp. 55-67.

[Ste95] Steinmetz R., "***Analyzing the Multimedia Operating System***", IEEE MultiMedia, Spring 1995, pp. 68-83.

[Sto92] Stoyenko, A. D., "***The Evolution and State-of-Art of Real-Time Languages***", The Journal of Systemas and Software, April 1992, pp. 61-84.

[Sun95a] Sun Microsystems Inc. "***The Java Language Specification - version 1.0 beta***", Mountain View, CA, 1995.

[Sun95b] Sun Microsystems Inc. "***The Java Language Tutorial***", Mountain View, CA, 1995.

[Sun95c] Sun Microsystems Inc. "***The Java Language Environment: A White Paper***", Mountain View, CA, 1995.

[Tak92] Takashio, K. and Tokoro, M., "***DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems***", Proceedings of OOPSLA'92, 1992, pp. 276-294.

[Tak96] Takashio, K. and Tokoro, M., "***Least Suffering Strategy in Distributed Real-Time Programming Language DROL***", Real Time Systems, Kluwier Academic Publishers, Boston, 1996, pp. 41-70.

[Tan95] Tanenbaum, A. S., "***Distributed Operating Systems***", Prentice Hall, 1995.

[Whi96] White, G., "***Java Command-Line Arguments***", Dr. Dobb's Journal, February, 1996.

[Wat88] Watanabe, T. and Yonezawa, A., "***Reflection in an Object-Oriented Concurrent Language***", Proceedings of OOPSLA'88, September 1988, pp. 306-314.

[Xu93] Xu, Jia and Parnas, D. L., "***On Satisfying Timing Constraints in Hard-Real-Time Systems***", IEEE, Transactions on Software Engineering, vol. 19, no. 1 January 1993, pp. 70-84.

[Yon88] Yonezawa, A . and Watanabe, T., ***"Reflection in an Object-Oriented Concurrent Language***", Proceedings of OOPSLA'88, September 1988, pp. 306-315.

Anexo A

Este anexo contém o pseudo-código e alguns comentários a respeito das classes implementadas no presente trabalho.

1. Classe Clock (MOC)

1.1 - Variáveis

FiladePedidosFuturos - Fila onde serão inseridos os pedidos de ativação de métodos em um instante de tempo futuro.

1.2 - Métodos

public synchronized void ProgramaAtivação(Nodo n) - responsável por inserir na *FiladePedidosFuturos* os métodos cuja ativação deverá ser feita num instante de tempo futuro.

public synchronized void EfetuaAtivação() - sua função é verificar se existe na FiladePedidosFuturos algum método cujo instante de sinalização é menor ou igual ao instante atual e, se houver, sinalizá-lo. Este método executa em uma thread própria, concorrente as outras threads do sistema, durante todo o tempo de execução do mesmo.

public void ativaclock() - cria e inicializa uma nova thread cuja prioridade é igual a máxima prioridade do sistema, ou seja, 10.

public void run() - após chamar EfetuaAtivação, "dorme" por 10 ms. Tempo que permitirá as outras threads do sistema, com prioridades menores, executarem.

1.3 - Pseudo-código

```
class Clock implements Runnable {

    Fila FiladePedidosFuturos = new Fila();

    public synchronized void ProgramaAtivacao (Nodo n)
    {
        FiladePedidosFuturos.insere(n);
    }

    public synchronized void EfetuaAtivacao()
    {
        if(!FiladePedidosFuturos.vazio()) then
        {
            // Verifica se algum pedido deve ser iniciado no tempo corrente
            while(FiladePedidosFuturos.elemento(i).inicio <= tempo_atual)
            {
                switch(no_sup.restricao)
                {
                case 'periodic':
                    {
                    // Sinaliza ao MO o início do novo período de uma
```

```
                                        //execução periódica
                                        (no_sup.manag).start_thread(no_sup);
                                        FiladePedidosFuturos.cancela(i);
                                        break;
                                        }

                                case 'start_at':
                                        {
                                        // Reassume a thread suspensa.
                                        (no_sup.id_thread).reassume();
                                        FiladePedidosFuturos.cancela(i);
                                        break;
                                        }
                                }
                        }
                }
}

public void run() {
        while(supervisora != null) {
                EfetuaAtivacao();
                // A thread "dorme" por 10 ms.
                        supervisora.dorme(10);
        }
}

public void ativaclock() {
        supervisora = new Thread(this, "Clock");
        // Fixa sua prioridade para a prioridade máxima do sistema.
        supervisora.fixaPrioridade(MAX_PRIORITY);
        // Inicia a thread.
        supervisora.start();
}
```

2. Classe Scheduler (MOS)

2.1. Política de Escalonamento implementada

Earliest Deadline First (EDF).

2.2 - Variáveis

FiladeEscalonamento - onde serão inseridos os pedidos de escalonamento, ordenados segundo a política especificada.

estado - variável que indica se há alguma solicitação de escalonamento sendo tratada pelo Meta Objeto Scheduler.

2.3 - Métodos

public boolean Escalona(Nodo n) - responsável pelo recebimento dos pedidos de escalonamento. Os pedidos recebidos são inseridos na FiladeEscalonamento ordenadamente.

public synchronized void LiberaProximoPedido() - é responsável pela liberação do primeiro método na FiladeEscalonamento, ou seja, aquele com menor *deadline*. Este método executa em uma thread própria a fim de liberar a thread que originou a chamada.

2.4 - Pseudo-código

```
class Scheduler  {

        Fila FiladeEscalonamento = new Fila();
        boolean estado = false;

        public synchronized boolean Escalona(Nodo n)
        {
                if(estado == false)  then
                {
                        // Se não há nenhum pedido sendo escalonado...
                        (thread_corrente).fixaPrioridade(8);
                        estado = true;      // ... agora há.
                        return true;
                }
                else {
                        FiladeEscaionamento.insere(n);
                        return false;
                }
        }

        public synchronized void LiberaProximo()
        {
                if(FiladeEscalonamento.vazio()   then)
                        estado=false;
                else
                {
                        Thread prim = (FiladeEscalonamento.first());
                        // Retira o próximo pedido da FiladeEscalonamento (com menor
                        // deadline)
                        FiladeEscalonamento.cancela(0);
                        (thread_corrente).fixaPrioridade (9);
                        prim.reassume(); // Reassume a thread do pedido retirado da
                                        // FiladeEscalonamento.
                }
        }
}
```

3. Um exemplo da Classe Manager (MOM)

3.1. Restrições temporais implementadas

Periodicidade, Aperiodicidade e de Tempo de Início de Execução.

3.2. Variáveis

FiladePendências - Fila onde serão inseridos os métodos liberados pelo MOS cujos estados de sincronização não permitem sua execução.

AutômatoFinito - Variável contendo o autômato que representa as condições de sincronização na execução dos métodos de um Objeto Base.

estado - controla a exclusão mútua na execução dos métodos do Objeto Base.

n_ativos - controla o número de métodos com restrição temporal aninhados.

3.3 - Métodos

3.3.1. Seção de Gerenciamento

public void RecebePedido (Nodo n) - recebe os pedidos de execução de métodos; verifica o tipo de restrição temporal associado e executa uma chamada ao método da restrição correspondente, na seção de restrições temporais.

public void LiberaPedidodeAtivação (Nodo n) - é responsável pela exclusão mútua na execução dos métodos do Objeto Base. Verifica, ainda, as condições de sincronização para a execução destes métodos e o número de métodos chamados pertencentes a mesma thread (métodos aninhados).

public synchronized void FimdeExecução (Nodo n) - é chamado ao final da execução de um método do Objeto Base. Se o método executado é um método aninhado, atualiza a variável n_ativos e retorna. Senão, reassume o próximo pedido na FiladePendências, ou seja, aquele cujo estado de sincronização permite que seja executado. Se a FiladePendências estiver vazia ou se nenhum método estiver em condições de executar devido ao estado de sincronização, sinaliza ao MOS a liberação de um próximo pedido.

3.3.2. Seção de Sincronização

public boolean verificaSincronização (int método) - verifica, segundo o autômato determinado, se as condições de sincronização permitem a execução do método do Objeto Base. Retorna true para um estado de sincronização que permita a execução do método e false caso contrário.

public synchronized void atualizaSincronização (int método) - atualiza a variável estado_de_sincronização, conforme o estado de sincronização anterior e o método que acabou de executar.

public synchronized int proximo() - busca na FiladePendências o próximo método cujas condições de sincronização permitem executar. Retorna a posição na FiladePendências na qual este método se encontra.

3.3.3. Seção de Restrições Temporais

public void PedidoSRT(Nodo n) - trata os pedidos de execução de métodos do Objeto Base sem restrição temporal. Estes pedidos são escalonados com um deadline "infinito" (no caso, o maior valor aceito em Java para os inteiros do tipo long).

public void Aperiodic(Nodo n) - solicita o escalonamento do método do Objeto Base e verifica se as condições de tempo de sincronização permitem sua execução. Se o método não conseguir executar dentro do limite de tempo pré-estabelecido, é sinalizado o método de exceção temporal correspodente.

public void Periodic(Nodo n) - solicita o escalonamento do método do Objeto Base (metId) e verifica as condições de tempo e sincronização. Se essas condições forem cumpridas, é realizada uma chamada ao metId no Objeto Base. Após a execução de metId é feita a programação para ativação futura deste método no próximo período. Cada período, por sua vez, executa em uma thread própria, inicializada a seu tempo pelo relógio do sistema. Métodos relacionados:

public synchronized void start_thread(Nodo n) - inicializa a thread correspondente ao novo período de MetId.

public void run() - código de execução da thread inicializa em start_thread. Faz uma chamada a Periodic(Nodo n).

public void Start_at(Nodo n) - esta restrição solicita o escalonamento do método do Objeto Base para um instante de tempo futuro. Assim, a thread corrente é suspensa e o método é programado para ser ativado no instante desejado. O relógio do sistema é responsável por reassumir esta thread no instante especificado.

3.3.4. Seção de Exceções

public void exceção() - método sinalizado em casos de violação das restrições temporais dos métodos do Objeto Base.

3.4. Pseudo-Código

```
class Manager implements Runnable {

Clock relogio;
Scheduler sched;
Fila FiladePendencias = new Fila();
boolean ativo = false;
Base inst_base;
Thread thread_ativa;
int n_ativos=0;
int estado_de_sincronizacao=0;
static int automatoFinito[][] = {{1, -1, -1}, {-1, 2, -1}, {-1, 2, 0}};

/*** Seção de Gerenciamento ***/

public void RecebePedido(Nodo n)
// Recebe os pedidos e chama a restrição temporal correspondente
{
        switch (n.restricao)
        {
        case 1:
                {
                Periodic(n);
                break;
                }
        case 2:
                {
                Aperiod(n);
                break;
                }
        case 3:
                {
                Start_at(n);
                break;
                }
        case 4:
                {
                PedidoSRT(n);
                break;
                }
        }
}

public synchronized boolean LiberaPedidodeAtivacao(Nodo no)
{
        if(ativo == true)   then
        {
                if(thread_corrente == thread_ativa)    then
                {
                // O método é um método aninhado
                        n_ativos ++;
```

```
                    return true;
                }
                else
                {
                LiberaProximoPedido();
                FiladePendencias.insere(no);
                return false;
                }
        }
        else {
                if(verificaSincronizacao(no.metodo))  then
                {
                        n_ativos = 1;
                        thread_ativa = thread_corrente;
                        ativo = true;
                        return true;
                }
                else
                {
                        LiberaProximoPedido();
                        FiladePendencias.insere(no);
                        return false;
                }
        }
}

public synchronized void FimdeExecucao(Nodo no) {
        if(n_ativos>1)  then
                n_ativos--;
        else {
                atualizaSincronizacao(no.metodo);
                if(!FiladePendencias.vazio())  then
                {
                        if(proximo() != -1)  then
                        {
                                int x = proximo();
                                (FiladePendencias.elemento(x)).id_thread.reassume();
                                FiladePendencias.cancela(x);
                                thread_ativa = thread_corrente;
                        }
                        else {
                                ativo = false;
                                LiberaProximoPedido();
                                }
                }
                else
                {
                        ativo = false;
                        LiberaProximoPedido();
                }
        }
}

/*  *** Seção de Sincronização ***/

public boolean verificaSincronizacao(int metodo)  {
        if(automato[estado_de_sincronizacao][metodo] != -1)  then
        return true;
        else
        return false;
```

```
}

public void atualizaSincronizacao(int metodo) {
        estado_de_sincronizacao = automato[estado_de_sincronizacao][metodo];
}

public int proximo() {
int cont=0;
boolean flag = true;
while(cont < FiladePendencias.tamanho() and flag==true)
{
if(verificaSincronizacao(((Nodo)FiladePendencias.elemento(cont)).metodo))   then
        flag = false;
else
        cont ++;
}
if(flag == true)   then
        return -1;
else
        return cont;

}

/*  *** Seção de Restrições Temporais ***/

public void PedidoSRT(Nodo n)
{
(n.id_thread.fixaPrioridade (1);
if(!sched.Escalona(n)) then
thread_corrente.suspende();
if(!LiberaPedidodeAtivacao(n))  then
(n.id_thread).suspende();
switch(n.metodo) {
                        case 'metodo1': {
                        inst_base.metodo1(n.id_thread);
                        break;
                        }
                        case 'metodo2':  {
                        inst_base.metodo2(n.id_thread);
                        break;
                        }
                }
FimdeExecucao(n);
}

public void Aperiod( Nodo no)
 {
        (no.id_thread).setPriority(1);
        if(!sched.Escalona(no))   then
        (no.id_thread).suspend();
        if(no.deadline > (tempo_corrente + no.tme))  then
        {
                if(!LiberaPedidodeAtivacao(no))  then
                (no.id_thread).suspend();
                if(no.deadline > (tempo_corrente + no.tme))   then
                {
                        switch(no.metodo) {
                                case 'metodo1': {
                                inst_base.metodo1();
                                break;
                                }
```

```
                                        case 'metodo2': {
                                        inst_base.metodo2();
                                        break;
                                        }
                                }
                        }
                        else
                                excecao(no.metodo);
                        FimdeExecucao(no);
                }
                else
                {
                        excecao(no.metodo);
                        LiberaProximoPedido();
                }
}

public void Start_at( Nodo no)
{
        if(no.inicio > tempo_corrente)  then
        {
        relogio.ProgramaAtivacao(no);
        (no.id_thread).suspend();
        }
        (thread_corrente).setPriority(1);
        if(!sched.Escalona(no))  then
        (no.id_thread).suspend();
        if(no.deadline > (tempo_corrente + no.tme))  then
        {
                if(!LiberaPedidodeAtivacao(no))  then
                (no.id_thread).suspend();
                if(no.deadline > (tempo_corrente + no.tme))  then
                {
                        switch(no.metodo) {
                                case 'metodo1': {
                                inst_base.metodo1();
                                break;
                                }
                                case 'metodo2': {
                                inst_base.metodo2();
                                break;
                                }
                        }
                }
                else
                        excecao(no.metodo);
                FimdeExecucao(no);
        }
        else
        {
                excecao(no.metodo);
                LiberaProximoPedido();
        }
}

public void Periodic(Nodo no)
{
        no.recebevalor(no);
        (thread_corrente).ficaPrioridade(1);
        if(!sched.Escalona(no_esc))  then
```

```
                {
                (no.id_thread).suspend();
                }
                if((no.inicio + no.periodo) > (tempo_corrente + no.tme))    then
                {
                        if(!LiberaPedidodeAtivacao(no))   then
                        (no.id_thread).suspend();
                        if((no.inicio + no.periodo) > (tempo_corrente + no.tme)))   then
                        {
                                switch(no.metodo) {
                                        case 'metodo1': {
                                        inst_base.metodo1();
                                        break;
                                        }
                                        case 'metodo2':  {
                                        inst_base.metodo2();
                                        break;
                                        }
                                        case 'metodo3':  {
                                        inst_base.metodo3();
                                        break;
                                        }
                                }
                                novoinic = no.inicio + no.periodo;
                                novotda = novoinic + no.periodo;
                                if (novoinic < no.fim )   then
                                {
                                        Thread newthread = new Thread(this);
                                        Nodo n = new Nodo();
                                        n.recebevalor(newthread, no.manag, novoinic, no.fim, no.periodo,
no.tme, no.metodo, no.excecao, 1, novotda);
                                        relogio.ProgramaAtivacao(n);
                                }
                        }
                        else
                                excecao(no.metodo);
                        FimdeExecucao(no);
                }
                else
                {
                        excecao(no.metodo);
                        FimdeExecucao(no);
                }
        }

        /*  *** Seção de Exceções Temporais ***/

        public void exceção(int metodo) {
                System.out.println(" Exceção sinalizada para o metodo"+ metodo);
        }
```

3.5. Classe Teste

3.5.1. Métodos

public static void main() - este método é responsável pela criação de todo o contexto de execução do sistema. Aqui são instanciados o MOS, MOC, os Objetos Base e seus respectivos MOs; além disso, é inicializada a thread do MOC. Neste método o usuário poderá fazer a chamada ao método do Objeto Base desejado, através de uma chamada ao método assinc(...) da Classe Intermediária *(veja item 3.5)*.

3.5.2. Pseudo-Código

```
class Teste1 {

        public static void main(String argv[]) {

        (thread_corrente).fixaPrioridade(9);
        Clock relogio = new Clock(); // Instancia relogio
        Scheduler sched = new Scheduler(relogio);  //Instancia escalonador
        Base1 inst1 = new Base1(); //Instancia objetos base
        Base2 inst2 = new Base2();
        Base3 inst3 = new Base3();
        Manager1 manag1 = new Manager1(relogio, sched); //Instancia MOs
        Manager2 manag2 = new Manager2(relogio, sched);
        Manager3 manag3 = new Manager3(relogio, sched);
        Intermediaria primeira = new Intermediaria(); //Instancia objetos Intermediária
        Intermediaria segunda = new Intermediaria();
        Intermediaria terceira = new Intermediaria();
        relogio.setsched(sched);
        relogio.ativaclock(); // Inicializa a thread do MOC

        // Chamada ao metodo1, periodico
        primeira.assinc(manag1, 10000, 80, 10, 'metodo1', "excecao", 1);
        //Chamada ao metodo2, aperiodico
        segunda.assinc(manag2, 100, 15, 'metodo2', "excecao", 2);
        //Chamada ao metodo3, start_at
        terceira.assinc(manag1, 350, 100, 10, 'metodo3', "excecao", 3);
        }
}
```

3.6. Classe Intermediária

3.6.1. Métodos

public synchronized void assinc(MO meta, long tfim, long P, long tme, char metodo, char exceção, int flag) - transforma estes parâmetros em um objeto do tipo Nodo e faz uma chamada a RecebePedido(Nodo n) do MO correspondente. Este chamada é feita em uma nova thread, de forma assíncrona. A função assinc com os parâmetros acima, refere-se a uma chamada a método com restrição Periodic, sendo:

meta - identificador do MO;
tfim - tempo de finalização da execução;
P - período para a execução do método;
tme - tempo máximo estimado para a execução do método;

metodo - identificador do método;
exceção - identificador do método de exceção temporal;
flag - identificador do tipo de restrição temporal.

public synchronized void assinc(MO meta, long deadline, long tme, char metodo, char exceção, int flag) - O mesmo que o método anterior para uma chamada a método com restrição Aperiodic, sendo, além dos parâmetros anteriores:

deadline - Intervalo de tempo máximo dentro do qual o método deve ser executado.

public synchronized void assinc(MO meta, long inicio, long deadline, long tme, char metodo, int flag) - Idem ao anterior, para uma chamada a método com restrição Start_at, sendo, além dos parâmetros já citados:

inicio - instante de tempo futuro no qual o método do Objeto Base deve ser inicializado.

3.6.2 - Pseudo-Código

```
class Intermediaria implements Runnable {
Vector FIFO = new Vector(10,2);

//Cria a thread para os pedidos com restrição periodica
public synchronized void assinc(Manager man, long tfim, long P, long tme, char met, String exc, int
flag) {
Nodo n = new Nodo();
long tempo_ativacao = tempo_atual();
Thread recebep = new Thread(this);
recebep.fixaPrioridade(2);
FIFO.addElement(n.recebevalor(recebep, man, tempo_ativacao, tfim+tempo_ativacao, P, tme, met,
exc, flag, tempo_ativacao+P));
recebep.start();
}

//Cria a thread para os pedidos com restrição aperiodica
public synchronized void assinc(Manager man, long deadline, long tmax, char met, String excecao,
int flag) {
Nodo n = new Nodo();
long tempo_ativacao = tempo_atual();
Thread recebea = new Thread(this);
recebea.setPriority(2);
FIFO.addElement(n.recebevalor(recebea, man, tempo_ativacao , deadline+tempo_ativacao, tmax,
met, excecao, flag, tempo_ativacao+deadline));
recebea.start();
}

//Cria a thread para os pedidos com restrição start_at
public synchronized void assinc(Manager man, long inicio, long deadline, long tme, char met,
String exc, int flag) {
Nodo n = new Nodo();
long tempo_ativacao = tempo_atual();
```

```
        Thread recebep = new Thread(this);
        recebep.fixaPrioridade(2);
        FIFO.addElement(n.recebevalor(recebep, man, tempo_ativacao, inicio+tempo_ativacao+deadline, P,
tme, met, exc, flag, tempo_ativacao+deadline));
        recebep.start();
        }

        //Cria a thread para os pedidos sem restrição temporal
        public synchronized void assinc(Manager man, char met, int flag) {
        Nodo n = new Nodo();
        long tempo_ativacao =tempo_atual();
        Thread recebep = new Thread(this);
        recebep.fixaPrioridade(2);
        FIFO.addElement(n.recebevalor(recebep, man, met, flag));
        recebep.start();
        }

        //Chama EnviaPedido e "dorme" por 10 ms
        public void run() {
        EnviaPedido();
         (thread_corrente).dorme(10);
        }

        //Chama RecebePedido
        public synchronized void EnviaPedido() {
        (manag).RecebePedido(FIFO.first());
        }
}
```

3.7. Um exemplo de Classe Base

A classe apresentada a seguir é referente aos testes realizados com animações gráficas, descritos a partir do item 5.2.2.

3.7.1. Código

```
import java.awt.*;

public class BaseTeste extends Frame {
        int frameNumber;
        Image images[];
        Dimension offDimension;
        Image offImage;
        Graphics offGraphics;
        MediaTracker tracker;
        int x = 0; int y = 0; int frameNumberMax = 0; int begin = 0;      int end = 0;

//Exibe uma animação no vídeo
public void anima( int ptx, int pty, int delay, int bg, int ed, int quadros  ) {
        long startTime = tempo_atual;
        int i = 0;
        x =  ptx;
        y = pty;

        delay = (delay > 0 ) ? ( 1000 / delay ) : 100;
        begin  = bg-1;
```

```
        end = ed-1;
        frameNumber = begin;

        while ( i< quadros) {
                repaint();
                        startTime += delay;
                        Thread.sleep( Math.max( 0, startTime - tempo_atual()) );
                } catch ( InterruptedException e ) {
                        break;
                }
                frameNumber++;
                i++;
        }
}

//Exibe uma imagem no vídeo
public void imagem( int ptx, int pty, int quadro  ) {
        x =  ptx;
        y = pty;
        frameNumber = quadro;
        repaint();
}

public void paint( Graphics g ) {
        if( offImage != null ) {
                System.out.println("Paint "+frameNumber);
                g.drawImage( offImage, 0, 0, this );
        }
 }

public void update( Graphics g ) {
        Dimension d = size();
        if ( ( offGraphics == null )  || ( d.width != offDimension.width )
            || ( d.height != offDimension.height ) ) {
                offDimension = d;
                offImage = createImage( d.width, d.height );
                offGraphics = offImage.getGraphics();
        }
        offGraphics.setColor( getBackground() );
        offGraphics.fillRect( 0, 0, d.width, d.height );
        offGraphics.setColor( Color.black );
        if ( tracker.statusID( 0,true ) == MediaTracker.COMPLETE ) {
                offGraphics.drawImage( images[frameNumber ], x, y, this );
        }
        g.drawImage( offImage, 0, 0, this );
}

public void init( int frmax ) {
        frameNumberMax = frmax;
        images = new Image[frameNumberMax];
        tracker = new MediaTracker( this );
        for( int i = 1; i <= frameNumberMax; i++ ) {
                images[i-1] = Toolkit.getDefaultToolkit().getImage( "t"+i+".gif" );
                tracker.addImage( images[i-1], 0 );
        }
}

public void metodo1() {
        System.out.println("metodo1!!!!");
        init( 17 );
        move(200,200);
        resize( 200, 200 );
```

```
        show();
        anima( 15, 15, 50,  1, 17 , 17);
}

public void metodo2(){
        System.out.println("metodo2 !!!!");
        imagem(50,50,frame);
        }
}
```

Anexo B

A Classe Real-Time do sistema operacional Solaris

No sistema operacional Solaris, os processos são escalonados segundo o conceito de classes de escalonamento, no qual cada classe possui uma política de escalonamento particular.

As classes de escalonamento definidas pelo sistema possuem uma hierarquia baseada em prioridades e são mostradas na Fig. B1:

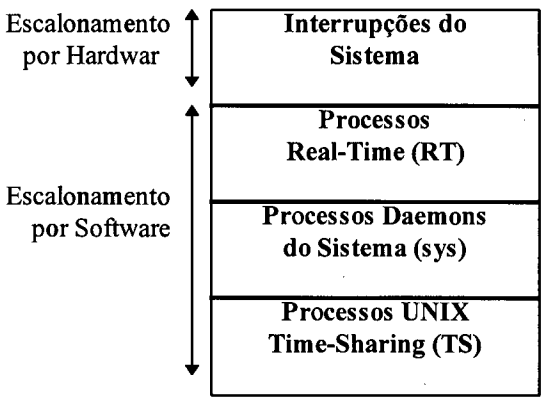


Figura B1. Classes de processos no sistema operacional Solaris

A classe de mais alta prioridade é a de Interrupções do Sistema, cujos processos são tratados por *hardware* e aos quais o usuário não tem acesso. A classe RT, com processos tempo real, tem a prioridade mais alta dentre as classes escalonadas pelo núcleo do sistema. Em seguida, em ordem decrescente de prioridades, vêm os processos *daemons* do sistema operacional e, por último, os processos UNIX, que seguem a política de escalonamento *time-sharing.*