Um Protótipo do Modelo Reflexivo Tempo Real RTR sobre a Linguagem Java

Danielle Nishida, Olinto Furtado, Joni Fraga e Jean-Marie Farines

Laboratório de Controle e Microinformática (LCMI)
DAS - Departamento de Automação e Sistemas - CTC - UFSC
Cx. P. 476 - CEP 88040-900 - Florianópolis - SC
E-mail: {danielle, olinto, fraga, farines}@lcmi.ufsc.br

RESUMO

Este artigo apresenta um protótipo obtido do estudo e implementação do Modelo Reflexivo Tempo Real RTR [Fur95] sobre a linguagem de programação Java. O Modelo RTR estabelece uma filosofia para o desenvolvimento de aplicações tempo real aliando os paradigmas de orientação a objetos e reflexão computacional. Desta forma, características como facilidade de manutenção, capacidade de reutilização e um alto nível de organização interna do sistema são obtidas, incorporando ao protótipo alguns dos principais conceitos ditados pela engenharia de *software*. Neste sentido, é apresentado o mapeamento do modelo sobre a linguagem Java, visando a máxima utilização dos recursos oferecidos por esta na satisfação das especificações do modelo. O protótipo resultante é testado com aplicações para exibição de animações gráficas.
Palavras chave: orientação a objetos, reflexão computacional, sistemas tempo real, Java.

ABSTRACT

This paper presents a Real Time Reflective Model prototype on the Java language. The RTR model establish a philosophy for development of real-time applications, joining the object-orientation and reflective computation paradigms. Hence, adding some characteristics like maintainability, reusability and high level internal organization of the system to the prototype. By this way, a mapping of this model is done onto the Java programming language with the aim of use all resources to satisfy the model specifications. The prototype resulting is tested with aplications to display graphic animations.
Key words: object orientation, reflective computation, real time systems, Java language.

1. INTRODUÇÃO

Este trabalho baseia-se no Modelo Reflexivo para Tempo Real RTR [Fur95]. Este modelo 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 [Mae87]. A orientação a objetos confere ao modelo características como modularidade, capacidade de reutilização e facilidade de manutenção. A reflexão computacional, por 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.

A linguagem Java, um produto Sun MicroSystems Inc., foi escolhida para a implementação do protótipo por ser totalmente orientada a objetos, além de apresentar características como concorrência a nível de linguagem e suporte para distribuição [Sun95a].

Neste artigo são apresentados aspectos relacionados ao desenvolvimento de um protótipo do modelo anteriormente citado. Segundo sua arquitetura, resultante da abordagem de meta-objetos, foram separadas as questões referentes a aplicação das questões de gerenciamento, que neste caso envolvem procedimentos de verificação de restrições temporais, escalonamento e monitoração temporal. O protótipo conta, ainda, com tipos pré-definidos de restrições temporais para representação de tarefas periódicas, aperiódicas e tarefas com início programado, o que não impede que outras restrições sejam implementadas pelo usuário, inclusive a partir da reutilização das já existentes. Além disso, as questões referentes ao gerenciamento da aplicação podem ser facilmente alteradas segundo a conveniência do usuário, que pode, por exemplo, alterar o algoritmo de escalonamento simplesmente substituindo o meta-objeto escalonador do modelo.

Esse texto apresenta na seção 2 uma breve definição do paradigma da reflexão computacional e a seção 3 descreve de forma sucinta o Modelo RTR e os objetos que o compõe. A seção 4 descreve o mapeamento do modelo sobre a linguagem de programação Java realizado no nosso trabalho, detalhando a dinâmica resultante deste mapeamento. A seção 5 refere-se a algumas considerações a respeito do protótipo implementado e dos testes realizados sobre este protótipo e a seção 6 apresenta as conclusões deste trabalho.

2. REFLEXÃO COMPUTACIONAL

Entende-se por reflexão computacional a atividade realizada por um sistema computacional quando este controla e atua sobre si mesmo [Mae87]. A técnica de reflexão computacional vem sendo largamente utilizada com o intuito de obter-se um maior grau de flexibilidade em aplicações, imprimindo aos sistemas computacionais maior capacidade de gerenciamento, maior legibilidade e facilidade de manutenção. Um sistema computacional com arquitetura reflexiva, segundo a abordagem de meta-objetos [Mae87] deve ser constituído por um nível base, onde são definidos os objetos-base e um nível meta onde são definidos os meta-objetos. De forma simplificada, seguindo a Figura 2.1, pode-se dizer que uma chamada a um método do objeto-base (①) será desviada ao seu meta-objeto correspondente (②), responsável por todos os procedimentos relacionados ao gerenciamento da execução do método solicitado. O método do objeto-base será ativado, segundo este gerenciamento, a partir do meta-objeto (③ e ④) que retornará os resultados ao objeto chamador (⑤). O propósito da reflexão é contribuir para uma maior organização interna do sistema, garantindo o funcionamento desejado dos objetos a nível base, e permitir mudanças nas políticas de controle sem que o nível base ou o suporte de execução sejam modificados.

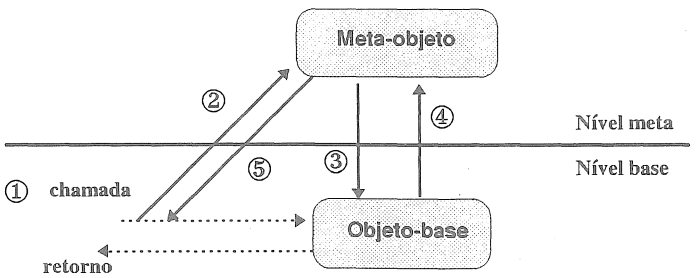

Figura 2.1: Exemplo de uma chamada a método reflexiva.

3. O MODELO REFLEXIVO PARA TEMPO REAL RTR

O Modelo Reflexivo para Tempo Real RTR adota o paradigma da reflexão individual, onde cada objeto-base possui um meta-objeto associado. O modelo suporta 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. A arquitetura adotada para o modelo é composta por objetos-base e seus respectivos meta-objetos, cujas funcionalidades serão apresentadas a seguir.

Os **objetos-base** do modelo são semelhantes aos objetos convencionais, com a diferença que restrições temporais e manipuladores de exceções podem ser associados às declarações de seus métodos.

Como dito anteriormente, os **meta-objetos Manager (MOM)** são responsáveis pelas questões referentes ao gerenciamento de seus respectivos objetos-base. Os pedidos de ativação de métodos dos objetos-base devem ser tratados verificando-se sua restrição temporal e processando-os conforme a mesma. Os MOM são responsáveis, ainda, pela garantia da exclusão mútua na execução dos métodos dos objetos-base, pela sincronização na execução dos mesmos (se houver) e pela sinalização de métodos de exceção nos casos em que as restrições temporais associadas não puderem ser cumpridas. Estes meta-objetos devem possuir uma fila denominada Fila de Pendências para armazenar os pedidos de execução que não estiverem aptos a executar devido a questões de concorrência e sincronização.

O **meta-objeto Scheduler (MOS)** é responsável pelo escalonamento dos pedidos de ativação de métodos dos objetos-base. O processo de escalonamento consiste em receber os pedidos de ativação, ordená-los segundo a política de escalonamento especificada e, por fim, liberá-los. O MOS deve atender pedidos originados a partir de um ou mais MOM's pertencentes ao mesmo nodo. Para acomodar estes pedidos, o MOS deve 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.

O **meta-objeto Clock (MOC)** é uma representação do relógio do sistema. As atividades desempenhadas por ele devem consistir, basicamente, na programação de ativações futuras bem como na monitoria do tempo destas ativações e cancelamento das mesmas. O armazenamento dos pedidos de ativação futura será feito em uma fila denominada Fila de Pedidos Futuros.

3.6. DINÂMICA DE FUNCIONAMENTO

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 é desviada ao meta-objeto correspondente ②, para que sejam tratadas as questões temporais e de sincronização. O meta-objeto Manager interagirá 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 solicita 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.

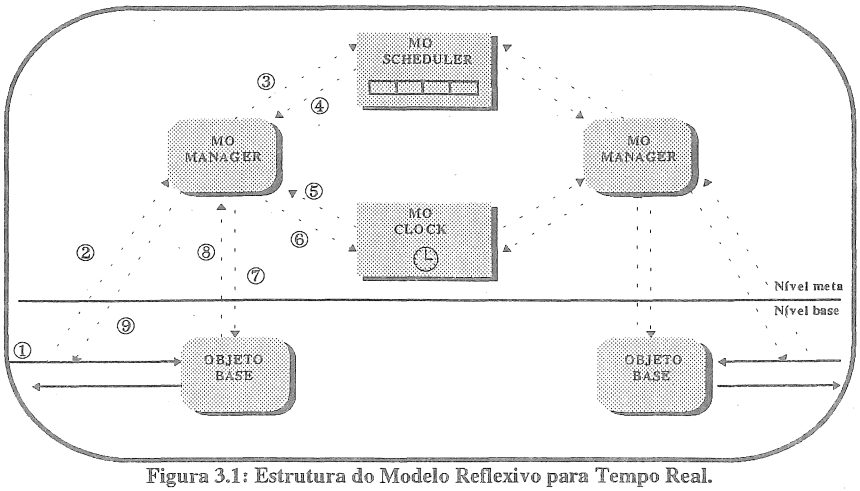

Figura 3.1: Estrutura do Modelo Reflexivo para Tempo Real.

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 de um sistema distribuído.

4. MAPEAMENTO DO MODELO RTR SOBRE A LINGUAGEM JAVA

Nesta seção, descreveremos como os recursos da linguagem Java foram utilizados para representar o Modelo Reflexivo para Tempo Real e como as necessidades impostas pelo modelo foram satisfeitas nessas representações.

4.1. A LINGUAGEM JAVA

Java é uma linguagem de programação orientada a objetos que tem entre suas principais características a portabilidade, robustez, concorrência a nível de linguagem, suporte à programação distribuída e amplos recursos para implementação de aplicações multimídia. A concorrência a nível de linguagem, realizada através de múltiplas *threads* de controle, foi um dos recursos mais explorados neste trabalho. Em Java pode-se criar e manipular *threads* de forma bastante simples, suspendê-las, reassumir sua execução e finalizá-las. Todas as *threads* possuem um valor de prioridade que pode variar de 1 a 10. Esse valor é herdado da *thread* criadora e pode ser verificado e alterado por métodos disponíveis na linguagem. As *threads* são preemptivas e podem ser interrompidas a qualquer instante por outra de maior prioridade [Sun95b].

4.2. DESCRIÇÃO DO MAPEAMENTO

A princípio, foram definidos objetos-base e meta-objetos como objetos Java, sendo que a reflexão computacional foi obtida através da arquitetura implementada. A concorrência desejada no nível meta foi alcançada com a utilização de *multithreading*.

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 mutuamente exclusivos. No item a seguir, descreveremos com mais detalhes como foi inserida a

concorrência no nível meta, supondo-se a utilização de um algoritmo de escalonamento do tipo Earliest Deadline First (EDF).

4.2.1. 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. 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 executável).

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:

- ***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 e seu processamento, a verificação das condições de sincronização 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.

- ***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 *thread* Clock é ativada periodicamente, a partir do relógio do sistema, com um período determinado em função da aplicação, permanecendo suspensa durante este período e permitindo que as outras *threads* do sistema com menor prioridade possam executar. A prioridade da *thread Clock* será máxima para permitir que ela interrompa a execução de outras *threads* quando expirar o tempo especificado.

- ***Threads* periodic** - as *threads Periodic* são criadas para atender a restrição temporal de periodicidade. 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.

- ***Thread* Libera Próximo Pedido** - todas as vezes em que um método do objeto-base acaba sua execução, é chamado um método do MOS (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.

4.2.2. ATRIBUIÇÃO DE PRIORIDADES ÀS *THREADS*

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, 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 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, de modo a permitir que os pedidos de escalonamento cheguem ao MOS e possam ser escalonados junto com outros pedidos. Se esta prioridade não fosse diminuída para 1, os pedidos que chegassem ao MOM seguiriam todos os procedimentos de escalonamento, sem permitir que outros pedidos acessassem o MOS. A Figura 4.1 ilustra este procedimento.
  - o fluxo de controle retornar do MOS 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, já que atuarão como ***threads* principais** depois de sua ativação pelo MOC;

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

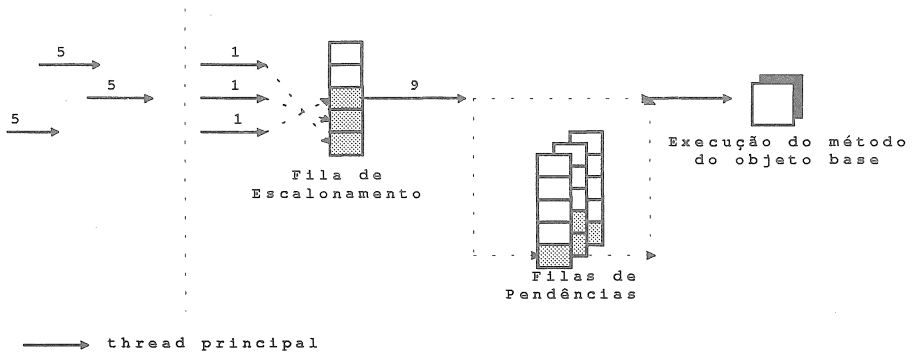

Figura 4.1: Fluxo de execução do modelo RTR Java devido a atribuição de prioridades às threads

4.3. 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 protótipo. Assim, com o auxílio da Figura 4.2, analisaremos todos os procedimentos que serão executados nesta *thread* e, a partir dela, toda a dinâmica do sistema.

• Inicialmente, o pedido de execução do método do objeto-base é desviado ao meta-objeto correspondente (①), onde sua restrição temporal é verificada e processada;

• O método correspondente a restrição temporal solicita o escalonamento do método do objeto-base ao meta-objeto Scheduler (②). Antes disso, é atribuída a *thread* principal a prioridade mínima;

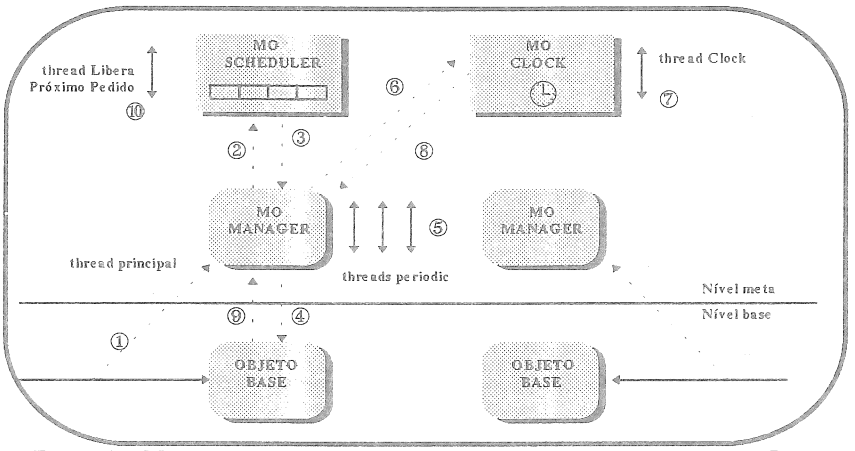

Figura 4.2: Mapeamento do Modelo Reflexivo Tempo Real sobre a linguagem Java.

• Se a Fila de Escalonamento estiver vazia e nenhum outro método do objeto-base estiver sendo executado, o fluxo de controle retorna ao MOM 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 for suspensa - 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 MOS a liberação do próximo pedido na Fila de Escalonamento.

• o comportamento do MOM depende da restrição temporal associada ao método da aplicação. Por exemplo, se a restrição temporal associada for de periodicidade, após retornar do MOS com permissão para execução do método são verificadas as condições temporais para que isso ocorra. Se estas condições forem favoráveis, são verificadas as condições de sincronização. Se o método solicitado estiver apto a executar no estado atual de sincronização do objeto-base, verifica-se se ainda há tempo para execução. Se houver, o mesmo é executado (④), caso contrário

um método de exceção é sinalizado; após qualquer um destes casos, o resultado é retornado ao objeto chamador e um novo pedido é liberado (⑨ e ⑩). Após a execução do método do objeto-base, deverá ser feita a programação de ativação para o próximo período. Uma ***thread* periodic** é criada e armazenada 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 MOM sua iniciação (⑧).

• os métodos sem restrição temporal terão tratamento especial por parte do MOM, sendo submetidos ao MOS com parâmetros de escalonemento tais que não interfiram na execução de outros métodos com restrição temporal.

5. TESTES E RESULTADOS

O protótipo do Modelo RTR foi implementado sobre a linguagem Java, versão 1.02 beta, sobre a plataforma de execução Solaris © Sun MicroSystems, versão 2.3. Este programa foi implementado como uma aplicação Java.

Para simular a reflexão na operação de ativação de métodos são feitas chamadas explícitas aos métodos dos meta-objetos. Isto porque a linguagem Java não apresenta recursos para reflexão automática, como, por exemplo, a linguagem Open C++ [Chi93].

Para expressar restrições de sincronização, que estabelecem relações de precedência na execução dos métodos da aplicação, foram utilizadas máquinas de estados finitos, sendo que os métodos são interpretados como transições.

Muitos testes foram realizados sobre o protótipo com o intuito de verificar aspectos como sua coerência lógica e temporal, capacidade de expressar restrições temporais e eficiência dos mecanismos para tratamento de exceções. 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, a coerência do mapeamento proposto, envolvendo múltiplas *threads* de controle e as interferências *time trigger* efetuadas pelo meta-objeto Clock, além disso, foram verificados os mecanismos de controle de concorrência e sincronização através de exemplos de aplicações do tipo *produtor x consumidor*. Pôde-se verificar que o mapeamento descrito no item 4 mostrou-se eficiente ao prever e evitar problemas de sincronização e exclusão mútua aos dados compartilhados, permitindo o escalonamento e priorizando a execução dos métodos liberados pelo escalonador.

Os **testes de expressividade** tiveram o objetivo de avaliar a capacidade do modelo na representação de cenários comuns ao domínio tempo real. Assim, foram testadas aplicações associando-se seus métodos a restrições temporais de periodicidade, aperiodicidade e de início programado. Estas aplicações são responsáveis pela exibição de animações gráficas. Pode-se obter uma animação, por exemplo, associando-se a um método que exibe um quadro de imagem na tela uma restrição de periodicidade, e atribuindo-lhe, em cada período, o número de um quadro na seqüência de animação. Este procedimento poderia ser realizado sem o auxílio do protótipo, simplesmente chamando-se o método da aplicação dentro de uma rotina que executasse ciclicamente. Porém, desta forma não seria possível o monitoramento temporal, não poderíamos precisar o instante de início da execução, o período entre as execuções, nem o instante de finalização das mesmas. Já com a utilização do protótipo isto torna-se possível, podendo o usuário, além disso, estabelecer uma relação de precedência na execução dos quadros. É

importante salientar que, devido a estrutura reflexiva, não é necessária nenhuma alteração no código da aplicação para executá-la através do protótipo. As restrições temporais foram implementadas com total liberdade de expressão, sendo o caráter flexível do Modelo RTR evidenciado quando as restrições implementadas são utilizadas para criar novas restrições, como foi o caso da criação de uma restrição temporal de periodicidade com início programado (*periodic_at*), criada a partir da restrição de periodicidade com início de execução imediato.

Os **testes de manipuladores de exceções** tiveram como objetivo verificar a capacidade do modelo no tratamento de erros. Foram implementados dois métodos distintos, utilizados sobre as aplicações citadas anteriormente. Quando um quadro de imagem não pode cumprir suas restrições, um método manipulador de exceções é sinalizado com o intuito de corrigir as falhas na exibição. Um destes métodos suprime o quadro de imagem que não pôde cumprir seu *deadline* e tenta exibir o próximo quadro na seqüência de animação, podendo este procedimento resultar numa seqüência entrecortada de imagens. O segundo método implementado mantém na tela o quadro de imagem anterior aquele que não conseguiu cumprir seu *deadline*, tentando exibi-lo novamente no próximo período. Assim, no caso de perdas de *deadline*, não existirão "quebras" na seqüência de animação, mas imagens "congeladas". Os métodos manipuladores de exceções implementados foram apenas dois exemplos das inúmeras possibilidades permitidas aos usuários, demonstrando a simplicidade de manipulação de erros através do protótipo.

Os testes funcionais, de expressividade e de manipuladores de exceções são mais detalhadamente descritos em [Nis96].

A linguagem Java mostrou-se extremamente adequada para a implementação do protótipo do Modelo RTR. Porém, alguns aspectos tiveram que ser contornados para satisfazer as exigências do modelo. Um destes aspectos é a impossibilidade, em Java, de referenciar os métodos da aplicação através de identificadores, a fim de ativá-los no meta-objeto. Desta forma, estes métodos foram passados como parâmetro para os meta-objetos Manager na forma de *strings*. Este problema já pode ser contornado na versão 1.1 da linguagem Java, que apresenta uma interface com mecanismos de reflexão que permitem, entre outras coisas, a identificação dos métodos de um objeto. Porém, a reflexão disponível nesta versão é introspectiva, não solucionando o problema do desvio para o nível meta, feito através de chamadas explícitas ao meta-objeto. Outro aspecto relevante é o gerenciamento automático de memória realizado por Java. O *Garbage Collector* (coletor de lixo) pode inserir algum grau de indeterminismo às aplicações tempo real. Porém, para sistemas tempo real *soft*, este problema pode ser contornado interpretando-se o *Garbage Collector* como uma tarefa periódica e estimando o tempo gasto em sua execução. Outra alternativa seria a adoção de um coletor de lixo determinista, como proposto em [Nil96].

Em relação às perspectivas deste trabalho, pretende-se que seus resultados contribuam para a especificação 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 simplificar a utilização do modelo, suprimindo as dificuldades anteriormente citadas através da construção de um pré-processador.

6. CONCLUSÕES

Este artigo apresentou o protótipo implementado a partir do Modelo Reflexivo Tempo Real RTR. Os resultados obtidos a partir dos testes sobre este protótipo mostraram sua eficiência no auxílio a implementação de aplicações tempo real. A arquitetura reflexiva, baseada na abordagem de meta-objetos, possibilitou a separação entre as funções de gerenciamento e de aplicação, resultando em um alto nível de organização interna do sistema e, conseqüentemente, em maior facilidade na utilização do protótipo. Segundo os testes realizados, pôde-se verificar que a arquitetura reflexiva não resultou num *overhead* tal que influenciasse a execução das aplicações. Devido as características de flexibilidade do modelo adotado, o conjunto de restrições temporais, os algoritmos de escalonamento e os métodos manipuladores de exceções não são questões impostas, mas altamente negociáveis, conforme as necessidades de uma aplicação específica. Por sua vez, a utilização da linguagem Java mostrou-se adequada para este trabalho. Por ser totalmente orientada a objetos, o código resultante desta implementação é bastante legível e de fácil entendimento. A concorrência a nível de linguagem possibilitou a fiel representação do modelo no sentido de implementar a concorrência desejada a nível de meta-objetos. Além disso, devido a portabilidade de Java, o protótipo pode ser executado em todas as plataformas para as quais a linguagem encontra-se disponível.

BIBLIOGRAFIA

[Chi93] 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.

[Fra97] Fraga, J.; Farines, J-M., Furtado, O. "RTR Model: An Approach for Dealing with Real-Time Programming in Open Distributed Systems". WORDS'97, Newport Beach, Ca, USA. February 5-7, 1997.

[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.

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

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

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

[Nis96] Nishida, D., "Estudo e Implementação do Modelo Reflexivo Tempo Real RTR sobre a Linguagem Java". Dissertação de Mestrado. Curso de Pós-Graduação em Eng. Elétrica. UFSC. Florianópolis, SC. December, 1996.

[Sun95a] Sun Microsystems Inc. "The Java Language Environment: A White Paper", Mountain View, CA, 1995.

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