Sandro Luís Schmidtke

Uso de Computação Imprecisa e Reflexão Computacional como mecanismo de adaptação para aplicações Tempo Real

Florianópolis – SC

2001

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Sandro Luís Schmidtke

Uso de Computação Imprecisa e Reflexão Computacional como mecanismo de adaptação para aplicações Tempo Real

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Olinto José Varela Furtado

Florianópolis, fevereiro de 2001.

Uso de Computação Imprecisa e Reflexão Computacional como mecanismo de adaptação para aplicações Tempo Real

Sandro Luís Schmidtke

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Computação, área de concentração Sistemas de Computação, e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Olinto José Varela Furtado, Dr.
(Orientador)

Fernando A Ostuni Gauthier, Dr.
(Coordenador do Curso)

Banca Examinadora

Olinto José Varela Furtado, Dr.
(Presidente)

Luiz Carlos Zancanella, Dr.

Luiz Fernando Friedrich, Dr.

Rômulo Silva Oliveira, Dr.

A meus pais, Ari e Isoldi Schmidtke, por seu carinho,
apoio e dedicação em todos os momentos de minha vida.
Para Carla Giane Ojczenasz, com muito amor.

AGRADECIMENTOS

Ao Professor Olinto José Varela Furtado, pela orientação, dedicação e empenho, sem os quais este trabalho não teria se realizado.

Aos membros da banca, por terem aceitado julgar este trabalho e pelos valiosos comentários que fizeram.

Agradeço aos meus colegas do CPGCC, pela amizade e apoio durante o curso, em especial aos meus amigos Luis Cláudio Gubert, Marcos Pisching e Hugo Klauck.

Ao NPD da UNIJUÍ –RS pelo apoio concedido.

Aos meus amigos que de alguma maneira ou de outra contribuíram para obtenção deste título.

Agradeço aos meus familiares, pela paciência, por suas palavras de incentivo, companheirismo em todos os momentos.

À Carla Giane por todo o amor, o apoio e a compreensão ao longo desses anos.

A DEUS, pela oportunidade concedida.

RESUMO

Este trabalho mostra como a técnica de Computação Imprecisa, implementada através de Reflexão Computacional, pode ser utilizada para permitir a adaptação de aplicações de Tempo Real a diferentes plataformas no contexto da Internet, bem como em sistemas de uso geral. Utilizamos o modelo de programação RTR para ilustrar a forma como esta adaptação poderá ser implementada.

Uma das técnicas existentes na literatura para flexibilizar o escalonamento tempo real é a Computação Imprecisa. Na Computação Imprecisa, onde as tarefas da aplicação são capazes de gerar resultados com diferentes níveis de qualidade ou precisão, porque cada tarefa é dividida em parte obrigatória (mandatory) e parte opcional (optional). A parte obrigatória da tarefa é capaz de gerar um resultado com qualidade mínima necessária para manter o sistema operando de maneira segura. A parte opcional refina este resultado, até que ele alcance a qualidade desejada.

A Reflexão Computacional pode facilitar a implementação da Computação Imprecisa, separando as questões funcionais das questões de controle responsáveis pela adaptação da aplicação. Entre os modelos de programação propostos na literatura que incluem reflexão está o Modelo Reflexivo Tempo Real (RTR), que é um modelo de programação reflexivo e de tempo real caracterizado por permitir, de forma flexível e sistemática, a representação e o controle de aspectos temporais de aplicações tempo real que seguem uma abordagem de melhor esforço.

Demonstramos através de um protótipo, a validade do uso da reflexão computacional juntamente com a técnica de computação Imprecisa como um mecanismo de adaptação para aplicações tempo real, e que o modelo RTR através de suas especificações é capaz de suportar esta implementação.

ABSTRACT

This work shows as the technique of Imprecise Computation, implemented through Computational Reflection, can be used to allow to the adaptation of applications of Real Time the different platforms in the context of the Internet, as well as in systems of general use. We use the model of programming RTR to illustrate the form as this adaptation could be implemented.

One of the existent techniques in the literature for flexibility into the scheduling real time is the Imprecise Computation. In the Imprecise Computation, where the tasks of the application are capable to generate results with different quality levels or precision, because each task is divided partly mandatory and it leaves optional . The mandatory part of the task is capable to generate a result with necessary minimum quality to maintain the system operating in a safe way. The optional part refines this result, until that he reaches the wanted quality.

The Computational Reflection can facilitate the implementation of the Imprecise Computation, separating the functional questions of the responsible questions of control for the adaptation of the application. It enters the considered models of programming in literature that include reflection are "Modelo Reflexivo Tempo Real" (RTR), that it is a reflective model of programming and real time characterized by allowing, of flexible and systematic form, the representation and the control of time aspects of applications real time that follow a boarding of better effort.

We demonstrate through the prototype, the validity of the use of the computational reflection together with the technique of Imprecise Computation as a mechanism of adaptation for applications real time, and that model RTR through its specifications is capable to support this implementation.

SUMÁRIO

ÍNDICE DE FIGURAS E TABELAS

1. Introdução

Sistemas computacionais de tempo real são definidos como aqueles submetidos a requisitos de natureza temporal. Nestes sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. Os aspectos temporais não estão limitados a uma questão de maior ou menor desempenho, mas estão diretamente associados às próprias funcionalidades do sistema.

Na literatura os sistemas de tempo real são, em geral, classificados conforme a criticidade dos seus requisitos temporais. Nos sistemas tempo real críticos (*hard real-time*) o não atendimento de um requisito temporal pode resultar em conseqüências catastróficas tanto no sentido econômico quanto em vidas humanas. Para sistemas deste tipo é necessária uma análise de escalonabilidade ainda em tempo de projeto (*off-line*). Esta análise procura determinar se o sistema vai ou não atender os requisitos temporais mesmo com um cenário de pior caso, quando as demandas por recursos computacionais são maiores. Quando os requisitos temporais não são críticos (*soft real-time*) eles apenas descrevem o comportamento desejado. O não atendimento de tais requisitos reduz a utilidade da aplicação, mas não a elimina completamente, nem resulta em conseqüências catastróficas.

A tecnologia empregada nos sistemas computacionais para o mercado de massa (*off-the-shelf*) evoluiu no sentido de prover um bom comportamento médio, sem prestar atenção ao comportamento de pior caso. Mecanismos como arquiteturas *pipelines*, memória cache e acesso direto a memória (direct memory access – dma) dificultam a análise de escalonabilidade. A mesma dificuldade está presente nos sistemas operacionais mais utilizados. Felizmente, a maioria das aplicações de tempo real não são

críticas, o que permite sua implementação em arquiteturas convencionais, apesar das dificuldades citadas antes.

Uma das possibilidades abertas com o emprego da Internet e do WWW (Word-Wide-Web) é a disseminação de aplicações na forma, por exemplo, de applets Java ou componentes Active-X. Entre as aplicações disseminadas estão algumas que incluem restrições de tempo real. Por exemplo, aplicações que lidam com áudio e vídeo, ferramentas para trabalho cooperativo e jogos individuais ou coletivos onde o tempo de reação do jogador é importante.

A forma como a Internet opera atualmente não suporta, de maneira geral, aplicações tempo real críticas [OLI97a]. Mesmo para aplicações tempo real não críticas (soft real-time) o uso da Internet é problemático. Uma aplicação disseminada através do WWW vai encontrar, como sua plataforma de execução, os mais diferentes processadores e sistemas operacionais. Existe uma enorme dificuldade em construir uma aplicação tempo real capaz de apresentar um comportamento temporal aceitável tanto em uma estação de trabalho último tipo quanto em um microcomputador com vários anos de uso. No sentido de viabilizar a disseminação em larga escala de aplicações tempo real, busca-se um mecanismo de adaptação para estas aplicações.

A questão da adaptação de aplicações tempo real ao seu ambiente de execução não está limitada ao contexto da Internet. Ela ocorre também quando aplicações possuem um comportamento dinâmico que impede uma reserva de recursos antecipada ou são executadas em ambientes de propósito geral onde a carga do sistema não é previsível. Entretanto, nestes sistemas a adaptação é feita, em grande parte, pelo sistema operacional ou suporte de execução. No caso de uma aplicação tempo real disseminada via Internet nada pode ser suposto a respeito do sistema operacional alvo. A adaptação deve ser provida exclusivamente pela aplicação.

Uma das técnicas existentes na literatura para flexibilizar o escalonamento tempo real é a Computação Imprecisa. Na computação imprecisa as tarefas da aplicação são capazes de gerar resultados com diferentes níveis de qualidade ou precisão. Para isto, cada tarefa é dividida em parte obrigatória (mandatory) e parte opcional (optional). A

parte obrigatória da tarefa é capaz de gerar um resultado com qualidade mínima necessária para manter o sistema operando de maneira segura. A parte opcional refina este resultado, até que ele alcance a qualidade desejada.

Em sistemas tempo real críticos (hard real-time) as partes obrigatórias são garantidas através de análise de escalonabilidade em tempo de projeto (off-line). Desta forma é possível maximizar a utilidade do sistema, através de partes opcionais, tendo o comportamento crítico garantido pelas partes obrigatórias. Quando os requisitos temporais não são críticos (soft real-time) é possível que determinadas tarefas não consigam executar sequer suas partes obrigatórias. Partes obrigatória e opcional passam a ser, na verdade, partes com precisão mínima e máxima, respectivamente. A motivação para usar computação imprecisa em sistemas deste tipo reside no fato dela permitir que a aplicação adapte sua utilidade dinamicamente, conforme a disponibilidade de recursos.

Também, a técnica de Reflexão Computacional pode facilitar a implementação da Computação Imprecisa, separando as questões funcionais das questões de controle responsáveis pela adaptação da aplicação. Entre os modelos de programação propostos na literatura que incluem reflexão está o Modelo Reflexivo Tempo Real RTR [FUR97]. RTR é um modelo de programação reflexivo e de tempo real, que se caracteriza por permitir, de forma flexível e sistemática, a representação e o controle de aspectos temporais de aplicações tempo real que seguem uma abordagem de melhor esforço.

O objetivo deste trabalho é mostrar como a técnica de computação imprecisa implementada através de reflexão computacional pode ser utilizada para permitir a adaptação de aplicações de tempo real a diferentes plataformas no contexto da Internet, bem como em sistemas de uso geral. O modelo RTR será utilizado para ilustrar a forma como esta adaptação pode ser implementada.

Para chegarmos às conclusões a que este trabalho se propõe, foi desenvolvido um protótipo de um jogo utilizando a estrutura proposta pelo Modelo RTR, utilizando a linguagem Java com a extensão Javassist [CHI00].

Os resultados alcançados neste trabalho de dissertação visam contribuir para a validação do Modelo RTR, explorando suas potencialidades na disseminação de aplicações em ambientes com máquinas e sistemas operacionais diversos (como a Internet, por exemplo).

O Capítulo 2 apresenta os principais aspectos dos sistemas de Tempo Real, além de abordagens da criticidade do sistema, escalonamento das tarefas, uso de linguagens Tempo Real e sua adaptabilidade em Sistemas Tempo Real na Internet, trata também do aspecto distribuído de tais aplicações usando CORBA.

O Capítulo 3 demonstra as características de Computação Imprecisa e suas formas de programação, escalonamento e aplicabilidade em Sistemas Tempo Real.

O Capítulo 4 descreve o paradigma da Reflexão Computacional. Demonstra suas características, requisitos e limitações. Completando o capítulo com algumas linguagens e modelos reflexivos. Apresenta o modelo RTR, com sua dinâmica de funcionamento e descrevendo sua expressividade e potencialidade para adaptação de aplicativos tempo real utilizando-se da técnica de Computação Imprecisa.

O Capítulo 5 apresenta os mecanismos de adaptação, bem como comparação entre as técnicas de Computação Imprecisa e Reflexão Computacional com o Modelo RTR e Sistemas Tempo Real, especificando as definições de implementação das adaptações.

O Capítulo 6 descreve o Protótipo utilizado com a descrição das técnicas aplicadas e o formato dos objetos adaptados para o Modelo RTR.

O Capítulo 7 apresenta as conclusões do trabalho, destacando as contribuições, vantagens e limitações da abordagem proposta. Apresenta ainda as perspectivas relativas à continuidade do trabalho.

2. Tempo Real

A complexidade do desenvolvimento de sistemas tempo real tem aumentado nas últimas décadas, exigindo um profundo entendimento de toda a tecnologia envolvida. A crescente exigência de precisão (como, por exemplo, nos sistemas de exploração remota para ambientes de alto risco) e confiabilidade desencadeou a necessidade de aprimorar as atuais formas de desenvolvimento destes sistemas, buscando mecanismos para facilitar a compreensão, delimitar o problema e gerenciar o tempo e os custos da solução.

Além das abordagens metodológicas para o desenvolvimento dos sistemas tempo real é importante a presença de suporte computacional eficiente disponibilizando facilidades para modelar, simular, gerar código automático como também possibilidade de consistir o trabalho realizado.

1 Deadline em Sistemas Tempo Real

Deadlines são críticos em sistemas de tempo real hard. Neste tipo de sistema é necessário ter uma garantia em projeto de que todas as tarefas serão sempre concluídas antes do respectivo deadline.

O desenvolvimento de aplicações tempo real críticas é dificultado por uma série de fatores. Entre eles está a dificuldade de calcular o tempo máximo de computação dos componentes da aplicação.

É possível desenvolver arquiteturas especiais que facilitam a análise de pior caso do comportamento da aplicação. Entretanto, tais arquiteturas especiais não são economicamente atrativas e limitam o uso da aplicação às instalações que dispõe de tal arquitetura. O uso de arquiteturas especiais na prática está limitado aos sistemas que são críticos a ponto de justificar o custo.

Este trabalho não se preocupa com aplicações de tempo real críticas, já que implementa um protótipo usando uma linguagem que não é a apropriada para este tipo de aplicações (Java), além de utilizar máquina de propósito geral com arquiteturas e/ou sistemas operacionais diferentes onde não se pode medir, com precisão absoluta, antecipadamente a duração de qualquer processamento.

Por isso passaremos a tratar especificamente do problema em aplicações de tempo real não críticas (Soft).

2 Escalonamento em Sistemas Tempo Real

Aplicações tempo real críticas demandam uma previsibilidade determinista. Este tipo de previsibilidade pode ser obtido, entre outras formas, através do escalonamento baseado em prioridades([FUR97]). Tarefas recebem prioridades segundo alguma política específica e um teste de escalonabilidade é executado em tempo de projeto, determinando se existe a garantia de que todas as tarefas serão executadas dentro dos deadlines. O teste de escalonabilidade deve ser compatível com a política de atribuições de prioridades adotada. Em tempo de execução, um escalonador preemptivo produz a escala de execução usando as prioridades das tarefas.

As soluções baseadas em prioridades podem ser divididas em prioridades fixas e prioridades variáveis. Quando prioridades fixas são usadas, cada tarefa recebe em projeto uma prioridade que vale durante toda a vida do sistema. Políticas clássicas para atribuir prioridades fixas são o taxa monotônica ("rate monotonic") e o deadline monotônico ("deadline monotonic"). Também existem algoritmos para atribuir prioridades em modelos de tarefas mais complexos.

Podemos ver em [OLI96] uma solução de escalonamento tempo real também baseada em prioridades preemptivas. Ao contrário das abordagens descritas anteriormente, emprega prioridades fixas e variáveis simultaneamente no sistema. A solução proposta apresenta vantagens tanto de prioridades fixas como de prioridades variáveis, considerando apenas aplicações tempo real críticas, onde a garantia em tempo de projeto é uma necessidade imprescindível. Tarefas periódicas ou esporádicas, com release jitter e deadline menor ou igual ao período, receberam prioridades fixas. Tarefas periódicas, com deadline igual ao período, receberam em conjunto uma prioridade menor e foram escalonadas segundo EDF.

Em sistemas distribuídos, o escalonamento tempo real é usualmente dividido em duas fases: alocação e escalonamento local. Inicialmente cada tarefa é alocada a um processador específico. A alocação original é permanente pois normalmente não é possível migrar tarefas durante a execução. A segunda fase analisa a escalonabilidade de cada processador.

Relações de precedência são comuns em sistemas distribuídos. Elas são criadas pela necessidade de sincronização e/ou transferência de dados entre tarefas. É possível usar offsets [OLI98] para implementar relações de precedência. Definindo um offset entre as liberações de duas tarefas é possível garantir que a tarefa sucessora iniciará sua execução após a predecessora estar concluída. Esta técnica é as vezes chamada de "liberação estática de tarefas".

Também é possível implementar relações de precedência fazendo a tarefa predecessora enviar uma mensagem para a tarefa sucessora. Esta mensagem informa ao escalonador que a tarefa predecessora terminou e a tarefa sucessora pode ser liberada. Esta técnica é às vezes chamada "liberação dinâmica de tarefas" ("dynamic release of tasks"). A incerteza sobre o momento da liberação da tarefa sucessora pode ser modelada como jitter na liberação [OLI98] (a palavra inglesa jitter, como empregada na literatura de tempo real, poderia ser traduzida como "tremor" ou "variação").

A maioria dos estudos sobre relações de precedência trata apenas com precedência linear ("pipelines"), onde cada tarefa tem no máximo um único predecessor e um único

sucessor. Modelos de tarefas que admitem relações arbitrárias de precedência não colocam tal restrição. Estudos que consideram relações arbitrárias de precedência normalmente assumem tarefas liberadas estaticamente.

3 Sistemas Tempo Real Distribuídos

Uma das formas de utilizarmos sistemas tempo real distribuídos segundo [MON98], é através da utilização de padrões como CORBA, ATM, POSIX e componentes de prateleira (off the shelf), em detrimento de soluções e protocolos proprietários.

3.1. CORBA

O CORBA (Commom Object Request Broker Architecture) é um middleware criado com objetivo de permitir que as dificuldades existentes na programação distribuída em ambientes heterogêneos sejam superadas. Suas especificações abertas, padronizadas pela OMG (Object Management Group) [OMG95], vêm recebendo crescente aceitação. A arquitetura CORBA é formada por um ORB (Object Request Broker) e por objetos de serviço e de facilidades. O ORB é responsável por gerenciar, de forma transparente, a comunicação entre objetos distribuídos. A OMG também padroniza um conjunto de objetos de serviços e de facilidades comuns que suportam algumas funções básicas usadas por objetos da aplicação.

3.2. RT-CORBA

Para suprir as necessidades relacionadas anteriormente, em outubro de 1998, conforme [MON98] [MON99] cinco propostas apresentadas previamente foram reunidas em uma única proposta [OMG98] – o RT CORBA 1.0. Esse documento enfatiza a definição de mecanismos de ORB, que permitem aos projetistas selecionarem as políticas para escalonamento de tempo real. Não existem suposições prévias sobre o ambiente operacional subjacente, mas o documento reconhece que um sistema operacional em conformidade com o POSIX é adequado para a obtenção de previsibilidade. O RT CORBA 1.0 descreve threads como as entidades escalonáveis.

Interfaces são especificadas para gerir threads o que permite uma grande portabilidade às aplicações. Threads podem ser definidas e controladas de forma uniforme, independentemente se a plataforma de execução possui threads Solaris, NT, ou POSIX.1c. RT_CORBA define uma série de extensões para CORBA.

4 Análise do Tempo de Execução

A análise de escalonabilidade de sistemas de tempo real depende do conhecimento do tempo máximo de execução (TME) das tarefas que compõem o sistema, o qual deve ser medido ou calculado previamente, conforme constatado na literatura sobre escalonamento e análise de escalonabilidade de sistemas de tempo real [FUR97].

Além do conhecimento do TME das tarefas, vários outros fatores devem ser considerados para que a análise de escalonabilidade de um sistema tempo real possa ser realizada. Dentre estes fatores, podemos citar: modelo de execução, técnica de escalonamento utilizada, restrições temporais, restrições de sincronização, exclusão mútua, compartilhamento de recursos e comunicação interprocessos.

O tempo de execução de uma tarefa ou de um programa pode ser obtido através de dois métodos básicos [FUR97]: medição, que é um método dinâmico; ou cálculo, que é um método estático. Adicionalmente, a combinação destes métodos também é possível.

A medição do TME de um programa pode ser realizada através de várias abordagens e consiste em, literalmente falando, "medir" o tempo gasto na execução de um segmento, uma tarefa ou um programa; medição esta que é realizada antes do sistema ser colocado em produção. Dentre as abordagens existentes, podemos destacar:

- Medição direta => que consiste na utilização de um analisador lógico para medir o tempo de execução de um segmento de código.

- Monitoração de software => que consiste na introdução de instruções (tais como leitura do relógio do sistema) no código objeto para obtenção de dados temporais;

- Simulação => que consiste na utilização de um software simulando o processador alvo;

Apesar de possível e comumente utilizado na prática, o modelo de medição de TME apresenta várias dificuldades técnicas, resultando em muitas desvantagens com relação ao método de cálculo.

O cálculo do tempo de execução é baseado na análise do código fonte e/ou código assembly dos programas, permitindo assim que a correção temporal de um sistema tempo real possa ser avaliada sem a necessidade de sua execução prévia. Esta análise pode ser manual, automática ou mista; entretanto a análise manual só é factível para programas simples e pequenos.

O cálculo sistemático do TME, apesar de ser mais complexo, apresenta uma série de vantagens com relação ao método de medição conforme citado em [FUR97].

Temos como fatores que influenciam no cálculo do TME, o código fonte do programa, linguagem e compilador utilizados, suporte de execução e sistema operacional, hardware envolvido (processador e outros dispositivos).

5 Linguagens Tempo Real

Nesta seção descrevemos as principais linguagens tempo real orientadas a objetos existentes.

5.1. ADA95

O novo padrão emergente para ADA95 é visto como uma tentativa de endereçar os requisitos de aplicações específicas em uma linguagem de uso geral (conforme [FUR97]). ADA95 é composta por um núcleo e por vários anexos (extensões), dentre os quais destacamos os anexos de orientação a objetos e tempo real.

Com relação a tempo real, a principal característica introduzida em ADA95 no nível de núcleo é a comunicação assíncrona entre tarefas, suportada diretamente pela construção "protected type" e pelo mecanismo ATC – Assynchronous Transfer of

Control (conhecido como select assíncrono). O anexo tempo real é composto por uma série de pacotes que provêm, entre outras, as seguintes facilidades: controle dinâmico de prioridade, relógio de tempo real monotônico e capacidade para cálculo do tempo de CPU das tarefas.

Com relação a análise de escalonabilidade, apesar das inovações introduzidas, claramente ADA95 continua permitindo que programadores escrevam programas cuja escalonabilidade não pode ser analisada, já que muitos aspectos, tal como uso de construções com tempo de execução ilimitado/imprevisível, não foram alterados no novo padrão proposto.

Destacam-se em ADA95 a robustez, derivada de sua semântica precisa de tipos, e a sua flexibilidade relativa a configuração da política de escalonamento. Por outro lado, conforme [FUR97], a representação de restrições temporais de forma indireta, não só dificulta o gerenciamento da complexidade como também o reuso e a manutenção dos sistemas tempo real produzidos; além disso, a análise de escalonabilidade só é possível através de uma disciplina de programação extremamente rígida, o que é problemático na medida em que ADA95 destina-se a programação de sistemas tempo real hard.

5.2. Real-Time C++ (RTC++)

Desenvolvida na Carnegie Mellon University, RTC++ (citado em [FUR97]) é uma extensão de C++ para programação de aplicações tempo real, implementada sobre o Kernel ARTS. RTC++ foi projetada com base em um modelo de objetos ativos com restrições temporais, denominados objetos de tempo real, que interagem através de passagem de mensagens e sincronizam-se através de regiões críticas.

Segundo [FUR97], RTC++ tem provisões suficientes para permitir análise de escalonabilidade e satisfaz a maioria dos requisitos de uma linguagem tempo real. Quanto ao escalonamento, RTC++ utiliza a abordagem "rate monotonic".

Destaca-se em RTC++, conforme [FUR97], a simplicidade com que as restrições temporais são expressas (associadas diretamente aos métodos dos objetos) e o

fornecimento de garantias temporais, viabilizando a realização de análise de escalonabilidade estática. Por outro lado, a especificação de restrições temporais em nível de comando e o mecanismo de concorrência utilizado dificultam o gerenciamento da complexidade e comprometem a capacidade de reuso e manutenção da linguagem. Além disso, a dependência de um ambiente operacional específico, embora possibilite o fornecimento de garantias, impede a utilização da linguagem em ambientes de propósito geral.

5.3. DROL

Desenvolvida na Keio University (Japão), Drol, analisada em [FUR97] é uma linguagem tempo real orientada a objeto projetada como uma extensão da C++, fundamentada no modelo DRO e implementada sobre o kernel ARTS. DROL suporta a programação de sistemas tempo real distribuídos, oferecendo facilidades para: expressão de restrições temporais, invocação polimórfica temporal, detecção e manipulação de exceções temporais, definição da semântica de comunicação (através de protocolos especializados) e separação parcial dos aspectos funcionais dos aspectos temporais e de sincronização (através da filosofia de meta-objetos).

As principais vantagens de DROL, citadas em [FUR97], são sua expressividade e sua flexibilidade, decorrentes da separação das questões temporais, de comunicação e de sincronização das funcionalidades da aplicação, possibilitada pelo uso de meta-objetos. Esta separação não só facilita o gerenciamento da complexidade, como também incrementa a capacidade de reuso e de manutenção dos programas produzidos. Também deve ser destacada em DROL, sua capacidade para definição de protocolos tempo real e a flexibilidade provida pelo mecanismo de invocação polimórfica.

Por outro lado, DROL não explora o potencial da reflexão para definição e controle de novos tipos de restrição temporal no nível de aplicação, apresentando um conjunto fixo de restrições temporais controladas pelo seu suporte de execução, o qual é dependente de um ambiente operacional específico; da mesma forma o controle do escalonamento também não pode ser realizado no nível de comando; aspecto este que

não parece adequado à filosofia reflexiva e que pode comprometer as vantagens advindas desta filosofia.

5.4. RealTimeTalk (RTT)

Desenvolvida na Suécia (Royal Institute of Technology), RTT se propõe a ser um sistema completo para desenvolvimento de sistemas tempo real hard, onde análise, projeto e implementação estejam fortemente relacionadas e sejam baseados em conceitos e ferramentas que permitam que cada estágio do processo de desenvolvimento seja visto como um mero refinamento do estágio anterior [FUR97]. A linguagem de programação RTT é baseada em Smalltalk, à qual são introduzidas algumas extensões.

Uma das principais contribuições de RTT é demonstrar que mecanismos típicos de linguagens orientadas a objeto, tais como ligação dinâmica e coleta de lixo automática, podem ser implementados de forma determinista; estes aspectos mais a proibição do uso de construções com tempo de execução ilimitado (tais como recursões e loops ilimitados), permite que o tempo máximo de execução (worstcase) dos programas RTT possam ser calculados e que estes programas tenham sua escalonabilidade analisada off-line.

Por outro lado, as limitações introduzidas reduzem a flexibilidade característica de Smalltalk e dificultam a programação de aplicações tempo real complexas. Além disso, a abordagem estática que caracteriza o modelo básico de programação RTT, embora necessária no contexto tempo real hard, não se mostra adequada aos requisitos de flexibilidade e composição com outros sistemas tempo real e mesmo com sistemas computacionais convencionais. Um aspecto questionável em RTT é o uso de uma linguagem extremamente dinâmica como Smalltalk, para implementação de um modelo de programação extremamente estático (segundo seus autores, a principal diferença entre RTT e MARS, um ambiente voltado para sistemas tempo real estáticos, é que MARS baseia-se em processos enquanto RTT baseia-se em objetos). A questão, portanto, é saber se o uso de Smalltalk justifica-se, uma vez que muitas de suas potencialidades não serão utilizadas.

5.5. FLEX

Desenvolvida na Universidade de Illinois como uma extensão de C++, FLEX é uma linguagem modular baseada no paradigma de computação imprecisa. Um programa FLEX, dependendo da disponibilidade de tempo e de recursos, pode produzir resultados mais ou menos precisos, porém sempre estáveis.

A abordagem adotada por FLEX citada em [FUR97] é diferente das demais linguagens na forma como ela tenta garantir a satisfação das restrições temporais e de recursos. O tempo de execução dos blocos com restrições pode ser ajustado dinamicamente através da substituição (progressiva e monotônica) de computações previstas originalmente, por alternativas menos precisas.

Destaca-se em FLEX seu modelo de programação baseado em computação imprecisa, o qual introduz flexibilidade ao mesmo tempo em que procura garantir a satisfação das restrições temporais. Outro aspecto positivo de FLEX é a possibilidade de composição dinâmica de novos tipos de restrições temporais, aspecto este que torna FLEX mais expressiva que a maioria das linguagens aqui apresentadas; contudo, tais restrições são limitadas a combinação dos atributos previstos, sendo controladas pelo suporte de execução.

Por outro lado, o mecanismo usado para representação das restrições temporais, além de pouco legível, fere a uniformidade do modelo de objetos, dificulta o entendimento de programas e reduz a capacidade de reuso e manutenção da linguagem.

5.6. Real-Time Java (RT-Java)

Real-Time Java é um superset de Java1.0 que introduz sintaxe adicional e semântica relacionada a tempo e memória para programas Java. Por outro lado, RT-Java é um subnet de Java 1.0, na medida em que ela proíbe o uso de determinados aspectos de Java quando estes interferem na habilidade do sistema em suportar confiavelmente requisitos tempo real. Especificamente, RT-Java consiste de uma combinação de bibliotecas de classes especiais, protocolos padrão para comunicação com estas

bibliotecas e a adição de duas estruturas de controle (timed e atomic) à sintaxe padrão de Java.

RT-Java objetiva o desenvolvimento de sistemas de tempo real em geral e de sistemas embutidos em particular, sendo que um dos principais benefícios das extensões propostas é permitir a negociação e o gerenciamento confiável dos recursos de tempo e memória durante o desenvolvimento de aplicações tempo real. Embora RT-Java apresente características para representação e controle de aspectos temporais, a obtenção de um comportamento tempo real mais ou menos preciso depende da máquina virtual usada para execução de seus programas e também do ambiente(sistemas embutidos ou ambientes de propósito geral) no qual ela esta inserida. Máquinas Virtuais Java (JVM) convencionais não estão aptas a garantir tal comportamento, provendo no máximo um comportamento aproximado ao desejado. Por outro lado, JVM estendidas adequadamente para tempo real (RTJVM) podem garantir comportamento tempo real, mesmo em ambientes de propósito geral.

Conforme encontramos em [FUR97], destaca-se em RT-Java a possibilidade de se controlar recursos de tempo de CPU e memória diretamente no nível da aplicação. Outro aspecto positivo de RT-Java é a realização de análise de escalonabilidade na fase de negociação para admissão de uma atividade na carga de trabalho do sistema. A existência de um coletor de lixo automático determinista é outra contribuição importante da linguagem.

A existência de um conjunto fixo de tipos de tarefas (representando tipos de restrições temporais) e a necessidade de uma disciplina de programação rígida para enquadrar uma aplicação aos tipos de tarefas disponíveis, são as principais limitações da linguagem; isto, juntamente com as limitações impostas para a obtenção de um comportamento previsível, dificultam a programação e comprometem a capacidade de reuso e manutenção características da linguagem Java.

6 Sistemas Tempo Real na Internet

Nos últimos anos a computação foi revolucionada pela Internet e, mais recentemente pelo World-Wide Web (WWW). Entre as inúmeras possibilidades abertas com a introdução destas tecnologias é possível destacar [OLI97a]:

A construção de aplicações distribuídas que usam a Internet como meio de comunicação entre seus componentes. Entre essas aplicações estão algumas que incluem restrições de tempo real. A forma como a Internet opera atualmente não suporta, de uma maneira geral, aplicações tempo real hard e é problemático o uso de tempo real soft. É possível imaginar a dificuldade de construir uma aplicação tempo real capaz de apresentar um comportamento temporal aceitável tanto em uma estação de trabalho executando Unix quanto em um microprocessador Intel-486 executando Microsoft Windows. As aplicações distribuídas enfrentam também o problema da enorme variabilidade nos atrasos associados com o envio de mensagens através da Internet.

A questão da adaptação de aplicações tempo real ao seu ambiente de execução não está limitada ao contexto da Internet. Ela ocorre também quando aplicações possuem um comportamento dinâmico que impede uma reserva de recursos antecipada. Também ocorre quando um mesmo sistema operacional suporta várias aplicações tempo real simultaneamente e variações no comportamento de uma aplicação interferem na quantidade de recursos disponíveis para as outras aplicações. Entretanto, nestes sistemas a adaptação é feita, em grande parte, pelo sistema operacional ou suporte de execução. No caso de uma aplicação tempo real disseminada via internet nada pode ser suposto a respeito do sistema operacional alvo. A adaptação deve ser provida exclusivamente pela aplicação.

6.1. Adaptação de Aplicações Tempo Real

Embora a maior parte da literatura de tempo real trate de adaptação através da negociação da qualidade de serviço, a adaptação também pode ocorrer através de uma ação unilateral [OLI97a], que no caso da Internet, pode ser:

Uma variação no comportamento do suporte gera como resposta uma adaptação da aplicação. Variações no atraso associado com o envio de mensagens através da internet devem ser tratadas pela própria aplicação, pois os suportes de execução existentes atualmente não são capazes de isolar a aplicação de tais variações. Mais importante ainda, ao ser disseminada através da Internet, e chegar a uma nova plataforma para execução, uma aplicação tempo real terá que adaptar-se ao desempenho desta plataforma. Atualmente a maioria dos programas ignora este problema e não oferece qualquer tipo de mecanismo de adaptação. Neste caso, a adaptação fica por conta do usuário, sujeito ao desempenho de uma aplicação executando em uma plataforma completamente diferente daquela para qual foi projetada.

6.2. Flexibilização do Deadline

Embora deadlines não possam ser garantidos em aplicações disseminadas ou distribuídas via Internet, o conceito "deadline" é útil por duas razões:

Uma aplicação pode usar os deadlines de suas tarefas para definir as prioridades dos seus fluxos de execução. Embora os sistemas operacionais em geral não suportem o conceito de deadline, a grande maioria deles suporta o conceito de prioridades. Por exemplo, é possível alterar a prioridade de uma thread na linguagem Java [OLI97a] com o método Thread.setPriority(). Desta forma, a aplicação pode usar os deadlines das tarefas como ponto de partida para a definição das respectivas prioridades. As políticas mais conhecidas neste sentido são o EDF (Earliest-Deadline-First), o Taxa Monotônica (Rate Monotonic) e o Deadline Monotônico (Deadline Monotonic).

A aplicação pode comparar os deadlines das tarefas com o tempo de resposta efetivamente observado. Desta forma, é possível obter uma quantificação do desempenho da aplicação com respeito aos seus requisitos temporais. Existem duas quantificações básicas: o somatório dos atrasos de todas as tarefas e a taxa de tarefas que perderam o respectivo deadline. Na situação mais comum, quando a tarefa deve ser executada mesmo com atraso, o somatório dos atrasos das tarefas fornece uma medida mais apropriada do desempenho. Por outro lado, a taxa de tarefas que perderam o

deadline é uma medida útil quando as tarefas possuem deadline firme ([OLI97a]), isto é, não existe benefício em executar uma tarefa após o seu deadline.

A forma mais simples e freqüente de adaptação é simplesmente relaxar o conceito de deadline.

A dificuldade encontrada no escalonamento tempo real levou alguns autores a proporem uma abordagem do tipo "fazer o trabalho possível dentro do tempo disponível". Esta técnica, conhecida pelo nome de Computação Imprecisa (Imprecise Computation [OLI97a]), flexibiliza o escalonamento tempo real na medida em que sacrifica a qualidade dos resultados para poder cumprir os prazos exigidos.

Uma forma mais radical de flexibilização é simplesmente não executar algumas tarefas quando o desempenho estiver abaixo do desejado (flexibilização do tempo de execução).

Muitos algoritmos dividem o trabalho a ser feito em partes. As partes podem então ser executadas em paralelo, com o objetivo de diminuir o tempo de resposta da tarefa. Caso a arquitetura sendo usada não suporte execução em paralelo, as partes são executadas seqüencialmente. O resultado é o mesmo, apenas o tempo de execução será maior. Esta organização do algoritmo permite sua adaptação a diferentes níveis de paralelismo.

Conforme [OLI97] [FUR97], em [STA95] é discutido como a "reflexão computacional" (reflection) pode ser empregada para introduzir flexibilidade em sistemas de tempo real complexos. Um sistema é considerado reflexivo quando ele raciocina sobre o seu estado corrente e o do ambiente e, em função disto, atua sobre seu próprio comportamento. As políticas e os parâmetros que definem o comportamento da aplicação formam um meta-nível (meta-level) onde a reflexão acontece. Informações sobre o estado da aplicação são mantidas e usadas para tomar decisões a respeito de seu comportamento, isto é, a respeito das políticas e dos parâmetros em vigor. Segundo [STA95] a reflexão computacional permite uma relação mais flexível às alterações que venham a ocorrer na dinâmica da aplicação e do ambiente.

3. Computação Imprecisa

Em sistemas tempo real existe uma dificuldade intrínseca em compatibilizar dois objetivos fundamentais: garantir que os resultados serão produzidos no momento desejado e dotar o sistema de flexibilidade para adaptar-se a um ambiente dinâmico e, assim, aumentar sua utilidade.

Em um extremo existem soluções de escalonamento que supõe um conjunto fixo de tarefas a serem executadas. Estas soluções reservam recursos para o pior caso e são capazes de garantir que todas as tarefas serão concluídas no momento correto. Entretanto, aplicações construídas desta forma resultam em sistemas pouco flexíveis e na subutilização dos recursos computacionais. No outro extremo temos as soluções de escalonamento que não garantem o comportamento temporal da aplicação. Tarefas são escalonadas na medida do possível. Embora os recursos computacionais sejam plenamente utilizados e o sistema resultante seja bastante flexível, a falta de uma garantia prévia para o seu comportamento temporal inviabiliza este tipo de solução para a classe de aplicações com requisitos temporais críticos.

Uma conseqüência imediata destas abordagens dinâmicas é a possibilidade de sobrecargas ("overload") no sistema. O sistema se encontra em estado de sobrecarga quando não é possível executar todas as tarefas dentro dos seus respectivos prazos. É importante observar que a sobrecarga não é um estado anormal, mas uma situação que ocorre naturalmente em sistemas que empregam uma abordagem tipo melhor esforço.

Logo, é necessário um mecanismo para tratar a sobrecarga. As abordagens usuais para o tratamento de sobrecarga são descarte por completo de algumas tarefas ou execução de todas as tarefas com sacrifício do prazo de execução de algumas delas.

Uma das técnicas existentes na literatura para resolver o problema de escalonamento tempo real é a Computação Imprecisa.

1 Características

Na Computação Imprecisa ([OLI97]) cada tarefa da aplicação possui uma parte obrigatória ("mandatory") e uma parte opcional ("optional"). A parte obrigatória da tarefa é capaz de gerar um resultado com a qualidade mínima necessária para manter o sistema operando de maneira segura. A parte opcional refina o resultado, até que ele alcance a qualidade desejada. O resultado da parte obrigatória é dito impreciso ("imprecise result"), enquanto o resultado das partes obrigatória+opcional é dito preciso ("precise result"). Uma tarefa é chamada de tarefa imprecisa ("imprecise task") se for possível decompo-la em parte obrigatória e parte opcional. Esta técnica procura, de certa forma, conciliar os dois objetivos fundamentais citados anteriormente.

Em situações normais, tanto a parte obrigatória quanto a parte opcional são executadas e o sistema gera resultados com a precisão desejada. Se, por algum motivo, não for possível executar todas as tarefas do sistema, algumas partes opcionais serão deixadas de lado. Este mecanismo permite a degradação controlada do sistema, na medida em que se pode determinar o que não será executado em caso de sobrecarga.

É possível dizer que Computação Imprecisa faz uma composição das abordagens tipo melhor esforço (partes opcionais) com as abordagens que oferecem garantia (partes obrigatórias). A técnica como um todo é do tipo melhor esforço, pois não oferece uma previsibilidade determinista para todas as tarefas do sistema.

Computação Imprecisa diferencia-se das demais abordagens tipo melhor esforço exatamente na forma como a sobrecarga é tratada. Nas demais abordagens do tipo "melhor esforço", a sobrecarga é tratada através de simples descarte de tarefas ou da

flexibilização do conceito de deadline. Na Computação Imprecisa, o tempo de computação das tarefas é negociado a partir da introdução do conceito de qualidade do resultado. Tarefas não são simplesmente descartadas, mas a qualidade do resultado produzido é parcialmente sacrificada. Isto gera uma redução na demanda pelos recursos do sistema que permite o atendimento dos deadlines, no seu sentido mais rigoroso.

O modelo de tarefas, normalmente associado com Computação Imprecisa, não exclui a existência de tarefas somente como parte obrigatória ou somente como parte opcional. Cabe à semântica da aplicação definir quais são as partes obrigatórias e quais são as opcionais. Este é um modelo de tarefas mais flexível do que o encontrado nas outras abordagens.

Computação Imprecisa pode ser usada para viabilizar o emprego, em sistemas de tempo real, de algoritmos cujo tempo de execução no pior caso torna o escalonamento inviável. Alguns autores acreditam que este pode ser o caminho para obter-se uma previsibilidade determinista em ambientes não deterministas, que requerem algoritmos complexos, cujo tempo de execução no pior caso é difícil de prever.

Computação Imprecisa pode ainda ser vista como um mecanismo de tolerância a faltas temporais. Mais exatamente, como um mecanismo capaz de tolerar sobrecargas no sistema sem comprometer seus requisitos temporais críticos. A não conclusão de uma tarefa completa antes do seu respectivo deadline é uma falta tolerada, na medida que sua parte obrigatória é sempre executada antes do deadline. Neste caso, temos uma degradação controlada do sistema na medida em que sua qualidade é reduzida.

Em aplicações de tempo real geralmente não existe migração de tarefas em tempo de execução, o que demandaria recursos tanto de processamento como de comunicação. Como cada tarefa é alocada permanentemente a um processador, aumenta a importância de uma alocação adequada.

Em função da complexidade do problema, as soluções propostas na literatura tendem a ser subótimas. É possível dividir as soluções existentes em duas linhas gerais: pesquisa heurística e pesquisa aleatória (ou algoritmos de otimização).

O objetivo primário da alocação é garantir que todas as tarefas sempre poderão concluir suas respectivas partes obrigatórias antes do deadline. A alocação possui como objetivo secundário aumentar as chances das partes opcionais das tarefas serem executadas.

2 Formas de Programação

Existem três formas básicas de programar tarefas imprecisas normalmente citadas na literatura: a programação pode ser feita com funções monotônicas, funções de melhoramento ou múltiplas versões.

As funções monotônicas ("monotone functions") são aquelas cuja qualidade do resultado aumenta (ou pelo menos não diminui) na medida em que o tempo de execução da função aumenta. As computações necessárias para obter-se um nível mínimo de qualidade correspondem à parte obrigatória. Qualquer computação além desta será incluída como parte opcional. O nível mínimo de qualidade deve garantir uma operação segura do sistema, enquanto a parte opcional refina progressivamente o resultado da tarefa. Segundo [OLI97], algoritmos deste tipo podem ser encontrados nas áreas de cálculo numérico, estimativa probabilista, pesquisa heurística, ordenação e consulta a banco de dados. Este tipo de função faz com que o escalonador tenha que decidir quanto tempo de processador cada parte opcional deve receber. É a forma de programação que fornece maior flexibilidade ao escalonador.

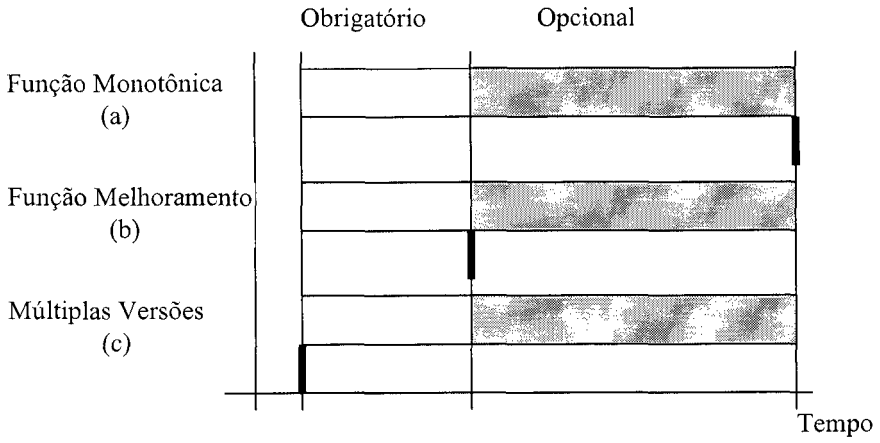
Funções de melhoramento ("sieve functions") são aquelas cuja finalidade é produzir saídas no mínimo tão precisas quanto as correspondentes entradas. O valor de entrada é melhorado de alguma forma pela função. Se o resultado recebido como entrada por uma função de melhoramento é aceitável como saída, então a função pode ser completamente omitida (não executada). As funções de melhoramento normalmente formam partes opcionais que seguem algum cálculo obrigatório. Tipicamente, não existe benefício em executar uma função de melhoramento parcialmente. Isto significa que o escalonador deve optar, antes de iniciar a tarefa, em executá-la completamente ou não executá-la. Um exemplo citado em [OLI97] é o processamento de sinais de radar.

Nestes, o passo que computa uma nova estimativa para o nível de ruído no sinal recebido pode ser omitido e a estimativa anterior usada no lugar. Também algoritmos de processamento de imagens são capazes de produzir imagens razoáveis mesmo se algumas etapas forem descartadas.

Uma tarefa imprecisa também pode ser implementada através de múltiplas versões ("multiple versions"). Normalmente são empregadas duas versões. A versão primária gera um resultado preciso, porém possui um tempo de execução no pior caso desconhecido ou muito grande. A versão secundária gera um resultado impreciso, porém seguro para o sistema, em um tempo de execução menor e conhecido. A cada ativação da tarefa, cabe ao escalonador escolher qual versão será executada. No mínimo, deve ser garantido tempo de processador para a execução da versão secundária, a qual corresponde a parte obrigatória. A parte opcional é definida pela diferença entre os tempos máximos de execução das versões primária e secundária. Esta técnica é a mais flexível do ponto de vista da programação.

No contexto das aplicações que usam a Internet, o mecanismo de adaptação de múltiplas versões é o mais promissor. A possibilidade de empregar múltiplas versões permite ao projetista da aplicação determinar comportamentos diferentes, conforme o desempenho do sistema onde a aplicação executa ou conforme os atrasos observados na rede enquanto a aplicação executa. A maior desvantagem deste mecanismo é a necessidade de explicitamente projetar e programar a aplicação para suportar os conceitos da Computação Imprecisa.

A forma de programação empregada interfere no comportamento do escalonador como podemos ver na Figura1.[OLI97]



| Último instante para decidir sobre parte opcional.

Figura 1 – Último instante para decidir sobre a execução da parte

Quando uma função monotônica é empregada, o tempo de processador alocado à parte opcional pode ser qualquer valor entre zero e o tempo máximo de execução da parte opcional. Isto porque, em uma função monotônica, qualquer tempo de processador fornecido ajuda a melhorar a qualidade do resultado. Além disto, a decisão de interromper a execução da parte opcional pode ser tomada a qualquer instante, mesmo quando esta já estiver executando (Figura1-a).

Quando a parte opcional executa uma função de melhoramento, não existe benefício em executá-la parcialmente. Assim, o escalonador deve decidir se executa a função de melhoramento completamente ou a descarta. Esta característica é chamada na bibliografia de restrição 0/1 ("0/1 constraint"). Ela restringe de certo modo a flexibilidade do escalonador. Além disto, a decisão deverá ser tomada, no mais tardar, quando a parte obrigatória é concluída e a parte opcional deve (ou não) ser iniciada. Uma vez iniciada a execução da parte opcional, ela é executada até o final (Figura1-b).

De forma semelhante, múltiplas versões criam uma restrição 0/1. Isto é, o emprego de múltiplas versões obriga o escalonador a executar completamente a parte opcional (escolhendo a versão primária) ou descartá-la completamente (escolhendo a versão secundária). Esta decisão deve ser tomada antes de iniciar a parte obrigatória da

tarefa pois, uma vez escolhida a versão, a execução ou não da parte opcional já estará automaticamente definida(Figura 1-c).

3 Escalonamento de Tarefas Imprecisas

Toda a proposta dentro da Computação Imprecisa necessita explicitar qual o objetivo a ser considerado no escalonamento das partes opcionais. Este objetivo será utilizado em caso de sobrecarga. Quando não é possível executar completamente todas as partes opcionais, é necessário algum critério para escolher quais partes opcionais terão tempo de execução sacrificado (parcialmente ou totalmente). Normalmente, este critério é definido a partir de uma medida do erro introduzido no sistema pelas tarefas tomadas individualmente.

3.1. Função Erro

Quando a parte opcional de uma tarefa é executada completamente, ela gera um resultado com qualidade máxima. Porém quando esta mesma parte opcional não é executada completamente, o resultado gerado possui uma qualidade inferior. Para efeitos de escalonamento, muitas propostas associam um valor de erro a cada parte opcional não executada completamente. Este valor de erro quantifica a diferença entre a qualidade do resultado preciso e a qualidade do resultado efetivamente gerado. É suposto que os erros associados com tarefas individuais contribuem, de alguma forma, para a redução da qualidade do sistema como um todo. Na literatura também pode ser encontrado o conceito de benefício gerado pela parte opcional. O benefício é definido como um sentido oposto ao do erro. Em outras palavras, uma parte opcional executada completamente gera um erro zero e um benefício máximo. Uma parte opcional completamente descartada gera um erro máximo e um benefício zero. Na maioria das propostas encontradas na literatura este erro é suposto proporcional ao tempo de execução que faltou para concluir a parte opcional em questão. É suposta uma linha reta para a relação "erro da parte opcional" versus "tempo de execução" .

Quando a tarefa imprecisa apresenta uma restrição 0/1 (funções de melhoramento e múltiplas versões) a parte opcional deve ser completamente executada ou então

completamente descartada. Em outras palavras, não existe redução do erro quando a parte opcional é parcialmente executada.

É reconhecido que, na prática, a função de erro não tem sempre o formato de uma linha reta. Dependendo do algoritmo em questão, esta curva pode ser côncava, convexa ou ainda possuir um formato mais complexo. Por exemplo, um algoritmo para cálculo numérico que realiza iterações que convergem para o resultado, possivelmente possui uma curva côncava. No início da execução do algoritmo, o erro do resultado diminui rapidamente, pois cada iteração representa um salto em direção ao resultado final. No final, com o resultado impreciso próximo do resultado exato, cada iteração faz apenas um pequeno refinamento.

[OLI97] cita a proposta de que o erro associado com uma parte opcional seja função do seu tempo de execução e também da qualidade dos seus dados de entrada. É normal que os resultados computados por uma tarefa sejam os dados de entrada de outra tarefa. Neste caso, se a qualidade destes dados for muito baixa, a tarefa que os recebe pouco poderá fazer. Uma outra possibilidade é citada em [OLI97], onde a qualidade dos dados de entrada não afeta diretamente a qualidade final, mas sim o tempo de execução da tarefa. Neste caso, dados de entrada com melhor qualidade resultam em um tempo de execução menor.

3.2. Objetivo Global do Escalonamento

Não existem trabalhos na bibliografia discutindo, sob a ótica da engenharia de software, a utilidade ou aplicabilidade das funções de erro propostas e como são empregadas no escalonamento das partes opcionais. O que encontramos na literatura com relação ao uso da função de erro, são diversos tipos de uso tentando minimizar ao máximo o erro para melhorar a qualidade (benefício). Este é na verdade o principal objetivo global do escalonamento.

Existem na bibliografia diversas propostas de modelos usando técnicas identificadas como Computação Imprecisa. Cada proposta define as propriedades do modelo de tarefas considerado, define o conceito de escalonamento ótimo e então

propõe algoritmos de escalonamento. A partir de variações nas propriedades das tarefas, na definição da função erro e na forma como a função de erro é usada no escalonamento, é possível criar inúmeros problemas de escalonamento tempo real baseados em Computação Imprecisa.

A parte obrigatória de uma tarefa deve ter seu deadline garantido. Para tanto, podem ser empregados, em tempo de projeto, algoritmos que trabalham com o pior caso e fornecem uma previsibilidade determinista, tais como o taxa monotônica e o deadline monotônico. Já as partes opcionais são escalonadas (ou não) visando maximizar a utilidade do sistema em tempo de execução ou, de forma equivalente, minimizar o erro gerado pela não execução de algumas partes opcionais.

O termo escalonamento preciso ("precise schedule") é usado para descrever uma solução de escalonamento onde as partes opcionais são completamente executadas. Desta forma, todas as tarefas sempre geram um resultado preciso. Já um escalonamento satisfatório ("feasible schedule") é aquele onde as tarefas sempre executam sua parte obrigatória, mas as partes opcionais podem ou não ser executadas completamente. Todo escalonamento preciso é também satisfatório, mas o recíproco não é verdadeiro.

4 Propostas – Carga do Sistema

Existem propostas que definem como carga do sistema um conjunto dinâmico de ativações de tarefas. Neste modelo de carga, tarefas imprecisas são dinamicamente ativadas, durante a execução do sistema. Em tempo de execução, o algoritmo de escalonamento procura aumentar o benefício gerado pelo sistema da melhor maneira possível. Como esta carga é potencialmente ilimitada, este algoritmo sozinho não consegue garantir que as partes obrigatórias de todas as tarefas serão executadas antes do respectivo deadline. Em função disto, as propostas que seguem esta linha consideram que as tarefas apresentadas ao escalonador satisfazem a restrição de que as partes obrigatórias são sempre escalonáveis ("feasible mandatory constraint"). Em outras palavras, é suposto que sempre é possível obter um escalonamento satisfatório. Existem dois caminhos para satisfazer a restrição de que partes obrigatórias são escalonáveis:

- A carga representada apenas pelas partes obrigatórias não nulas é, na realidade, limitada e conhecida antes da execução do sistema. Em tempo de projeto, um teste de escalonabilidade fornece a garantia de que todas as partes obrigatórias serão concluídas antes do respectivo deadline. em tempo de execução, o escalonador trabalha como se a carga fosse completamente dinâmica, porém apresentando um comportamento que não compromete o resultado do teste de escalonabilidade.

- O escalonador rejeita tarefas imprecisas ativadas dinamicamente cuja parte obrigatória não pode ser executada dentro do deadline. Neste caso, a aplicação é obrigada a realizar algum tipo de tratamento de exceção em função das rejeições.

Existem propostas que consideram como carga do sistema um conjunto estático de tarefas periódicas ou esporádicas. Neste caso, a carga é limitada e conhecida, permitindo que todas as partes obrigatórias sejam garantidas em tempo de projeto. As propostas deste grupo fornecem uma solução completa para o escalonamento.

Existem propostas limitadas às partes opcionais em que todas as propostas desta classe trabalham a partir da suposição que partes obrigatórias são sempre escalonáveis ("feasible mandatory constraint"). E existem propostas completas para o escalonamento de tarefas imprecisas, que consideram tanto as partes obrigatórias quanto partes opcionais.

5 Ambiente Distribuído

Existem poucos trabalhos na literatura analisando o emprego de Computação Imprecisa em ambientes distribuídos. Em [OLI97] uma tarefa possui versão primária (qualidade máxima) e versão secundária(resultado minimamente aceitável). As duas versões executam paralelamente em processadores diferentes. Se a versão primária termina dentro do prazo, seus resultados são utilizados. Caso contrário, são utilizados os resultados da versão secundária.

Em sistemas distribuídos também é comum o surgimento de relações de precedência entre tarefas. Uma relação de precedência tem origem em uma necessidade

de sincronização e/ou passagem de dados entre duas tarefas da aplicação. Uma mensagem cria uma relação de dependência entre a tarefa remetente e a tarefa destinatária. A tarefa destinatária somente pode iniciar sua execução após receber a mensagem. Embora isto ocorra também em sistemas centralizados, o fato das tarefas estarem distribuídas por diferentes processadores dificulta o escalonamento no sistema. Qualquer análise de escalonabilidade para ambiente distribuído deve ser capaz de lidar com relações de precedência.

Uma necessidade básica neste contexto são os protocolos de comunicação com atraso limitado. Garantir, em tempo de projeto, que todas as partes obrigatórias serão concluídas antes do respectivo deadline implica em reservar recursos para o pior caso. Pior caso aqui se refere tanto aos tempos de execução quanto à pior combinação possível de liberações de tarefas.

O problema de alocação de tarefas em ambiente distribuído é bastante complexo. Na maioria dos casos, obter uma solução ótima implica em custo de processamento proibitivo. Desta forma, são utilizadas abordagens subótimas. Em [OLI97] é usado um método de pesquisa aleatória bastante difundido, o recozimento simulado ("simulated annealing"). A justificativa para o uso de técnicas de otimização vem da dificuldade de construir boas heurísticas, considerando a complexidade do problema tratado.

A maior parte dos trabalhos publicados emprega prioridades fixas e liberação estática de tarefas para implementar relações de precedência. Em [OLI97] é proposto o emprego de prioridades fixas e liberação dinâmica de tarefas para escalonar sistemas tempo real críticos em ambiente distribuído. A teoria de escalonamento relacionada com prioridades fixas evoluiu bastante nos últimos anos, passando a suportar modelos de tarefas bem mais complexos que aqueles empregados nos primeiros trabalhos. Ao mesmo tempo a liberação dinâmica de tarefas é mais flexível que a liberação estática.

[OLI97] considera que tarefas recebem prioridades seguindo a política do deadline monotônico (DM, "deadline monotonic"). O deadline monotônico é uma forma simples e rápida de atribuir prioridades. Em sistemas de tarefas com relações de precedência o DM não é ótimo. Entretanto, DM é ótimo na classe das políticas que geram prioridades

decrescentes dentro dos grafos de precedência. Além disto, DM pode ser considerada uma boa heurística de propósito geral.

4. Reflexão Computacional

O conceito de reflexão computacional no modelo de objetos foi introduzido em 1987 por Patti Maes [MAE87]. Em seu conceito, reflexão computacional é a atividade executada por um sistema computacional quando faz computações sobre (e possivelmente afetando) suas próprias computações.

Reflexão Computacional é toda a atividade de um sistema computacional realizada sobre si mesmo, e de forma separada das computações em curso, com o objetivo de resolver seus próprios problemas e obter informações sobre suas computações.

O objetivo da Reflexão Computacional, é permitir que um sistema atue sobre si mesmo e não sobre o que o sistema deve produzir, realizando deduções e computações sobre dados internos do próprio sistema. Seu efeito, no modelo de objetos pode ser assim descrito: ao ser enviada uma mensagem a um objeto, ela é desviada para o seu meta-objeto correspondente, e, como resultado dessa reflexão, passa a realizar computações no meta-nível.

Segundo Steel [LIS97]: a reflexão computacional é a capacidade de um sistema computacional de interromper o processo de execução (por exemplo, quando ocorre um erro), realizar computações ou fazer deduções no meta-nível e retornar ao nível de execução traduzindo o impacto das decisões, para então retomar o processo de execução.

Como vemos em [LIS97], a reflexão computacional possui um significado mais diretamente relacionado com o modelo de objetos, embora seja encontrada também no modelo de programação funcional e programação em lógica, No modelo de objetos, refere-se à obtenção de dados sobre as propriedades de classes e de objetos de uma aplicação, e a possibilidades de modificar esses dados, modificando-se assim as propriedades das classes e dos objetos da aplicação.

Quando uma aplicação é executada, ela realiza computações sobre dados de seu próprio domínio e com o objetivo de atender os requisitos especificados para aquela particular aplicação; quando a computação é reflexiva, ela realiza computações sobre dados do próprio sistema, tendo por objetivo resolver problemas do sistema em si ou atuar sobre a aplicação. A auto-representação de um sistema permite a alteração e adaptação de determinados aspectos de sua estrutura e ou de seu comportamento.

Por domínio de um sistema computacional entende-se todo o conjunto de informação, expresso por objetos e operações, relacionado com uma determinada área.

A reflexão computacional define uma arquitetura em níveis, composta por um meta-nível, onde se encontram as estruturas de dados e as ações a serem realizadas sobre o sistema objeto, localizado no nível base. Assim, em uma arquitetura reflexiva, um sistema computacional possui dois componentes interrelacionados: um subsistema objeto, que faz computações sobre um domínio externo ao sistema, e um subsistema reflexivo, que faz computações sobre o sistema objeto. Os dados do nível base são usados no meta-nível para a realização de computações reflexivas que podem interferir nas computações de nível base. Podemos ver na figura abaixo um modelo de arquitetura reflexiva:

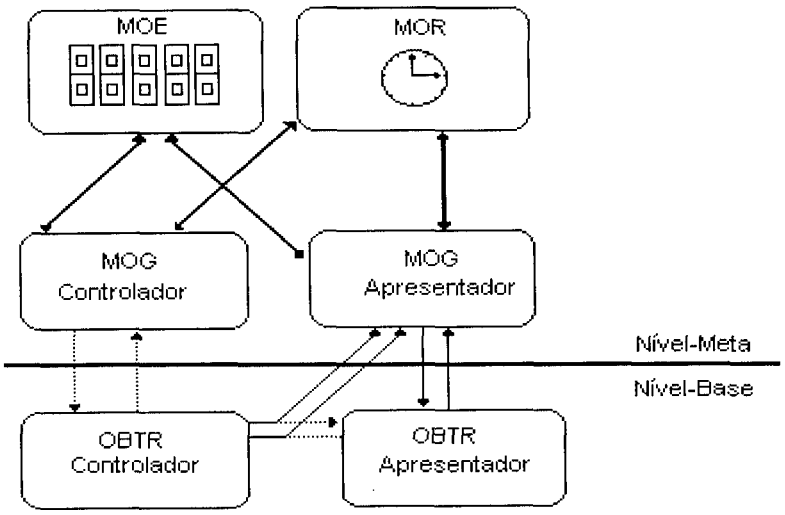


Figura 2 - Arquitetura Reflexiva

No modelo de objetos, a computação reflexiva pode ser feita sobre classes ou instâncias de classe (Objeto). No primeiro caso, o meta-nível é composto por meta-classes, as quais contém informações sobre os aspectos estruturais de componentes do nível base tais como descrição de variáveis e de métodos que irão compor todas as suas instâncias. Esta estrutura pode ser alterada e, em decorrência disso, todas as instâncias também serão alteradas. No segundo caso, o meta-nível é composto por meta-objetos, que contém informações sobre os aspectos de comportamento dos objetos de nível base, como por exemplo, como um determinado objeto trata uma mensagem.

Algumas linguagens orientadas a objetos utilizam meta-classe para descrever a estrutura de todas as suas classes, no sentido de que as informações necessárias para a construção de uma classe se encontram em sua meta-classe. Mais especificamente, meta-classes podem ser assim definidas (conforme encontramos em [LIS97]) : Uma meta-classe Mcx é uma classe que descreve a estrutura de uma classe x e cujas instâncias também são classes. Ou, simplesmente, uma meta-classe é a classe de uma classe.

Meta-classes, quando acessíveis aos usuários, permitem a realização de reflexão estrutural, que pode ser assim definida, com base em [LIS97]: Reflexão estrutural de

uma classe x é toda a atividade realizada em uma meta-classe Mcx , com o objetivo de obter informações e realizar transformações sobre a estrutura estática da classe x.

Objetos e meta-objetos são interconectados de tal forma que uma modificação em qualquer um deles provoca efeitos no outro, de forma dinâmica.

Genericamente, o conceito de meta-objeto é assim colocado por Foote [LIS97]: meta-objetos são objetos que definem, implementam, dão suporte ou participam de alguma maneira da execução da aplicação, ou dos objetos de nível base.

Um meta-objeto é um objeto instanciado a partir de uma classe, este objeto descreve alguns aspectos (não necessariamente todos) de um outro objeto ao qual se refere, e de algum modo, participa do processo de execução de seu objeto referente. Um meta-objeto Mox é um objeto que representa aspectos estruturais e comportamentais de um objeto O, a ele conectado. A estrutura de O é representada como atributos de Mox e os aspectos computacionais são descritos como métodos de Mox.

A arquitetura reflexiva admite diversos meta-níveis, caracterizando uma torre de reflexão: cada meta objeto pode ter um ou mais meta-objetos a ele associado. Meta-objetos, ao serem considerados objetos, podem enviar e receber mensagens de outros meta-objetos. Quando um meta-objeto recebe diretamente mensagens de outro meta-objeto, realiza-se uma computação reflexiva do sistema.

Componentes que podem ser observados ou alterados durante o processo de execução são denominados componentes abertos pois alguns detalhes de sua implementação são revelados a seus clientes. De forma oposta, componentes fechados (caixa-preta) revelam as suas funcionalidades através de interfaces bem definidas e escondem detalhes de sua implementação. Ainda de acordo com Kiczales, [LIS97], componentes abertos oferecem ao programador da aplicação mais opções, permitindo:

- Usar apenas as funcionalidades básicas de um componente, quando a sua implementação for adequada;

- Controlar a estratégia de implementação do componente, quando necessário;

- Decidir sobre a funcionalidade e a estratégia de implementação de forma completamente separada.

A reflexão computacional tem atraído a atenção de pesquisadores como forma de extensão de linguagens de programação e pela sua utilidade na implementação de diversos mecanismos, como por exemplo, mecanismos de programação distribuída, de forma simplificada e transparente à aplicação.

É no modelo de objetos, com sua natural flexibilidade e facilidade de programação incremental, que a reflexão computacional tem mostrado a sua eficácia e elegância na obtenção de novas soluções de problemas de programação.

Para incorporar o conceito de reflexão computacional ao processo de criação de programas, é necessário adotar um novo modelo de abstração: a separação de domínios. Um dos domínios compreende as funcionalidades da aplicação e o outro domínio é a própria aplicação.

Um programa reflexivo é estruturado na forma de dois programas: um programa de nível base e um programa de meta-nível. Segundo Kiczales [LIS97], "o programa de nível base expressa o comportamento desejado pelo cliente, em termos das funcionalidades fornecidas pelo ambiente de suporte. O programa de meta-nível pode adequar aspectos particulares de implementação do ambiente de suporte de forma que melhor atenda às necessidades do programa de nível base".

1 Modelo de programação Tempo Real Reflexivo Orientado a Objetos – Trabalhos Existentes

Um modelo de programação para tempo real é uma abstração capaz de representar tanto o sistema computacional de controle a ser produzido, quanto as entidades do mundo real (o ambiente da aplicação) por ele controladas; assim sendo, ele deve apresentar uma estrutura e uma semântica de funcionamento capaz de representar integralmente tanto aspectos funcionais quanto aspectos temporais das aplicações tempo real. Além disso, modelos de programação tempo real devem ser vistos como o aspecto

central no processo de desenvolvimento de sistemas tempo real, servindo como base para o estabelecimento de metodologias e linguagens de programação tempo real.

Podemos destacar os seguintes modelos de programação tempo real reflexivo baseados em objetos:

1.1. Modelo "DRO"

Proposto por Takashio e Tokoro [FUR97], DRO é um modelo de objetos tempo real distribuído (mono-thread) que estende o modelo de objetos convencional para suportar computação tempo real através da encapsulação de restrições temporais. O modelo DRO suporta as propriedades de melhor esforço ("best-effort") e menor prejuízo ("least-suffering") através do mecanismo de invocação polimórfica e da associação de restrições temporais aos métodos dos objetos.

Uma característica fundamental do modelo DRO é a utilização de meta-objetos como forma de separar explicitamente questões funcionais (definidas nos objetos básicos) das questões não-funcionais (controle de concorrência, de tempo e comunicação – definidas nos meta-objetos). Os meta-objetos introduzidos por DRO são: AbstracStateMeta, que gerenciam os conjuntos habilitados (usados para controlar concorrência) e a fila de mensagens; e ProtocolMeta, que define e controla os protocolos de comunicação (dando semântica para a comunicação inter-objetos) e gerencia as restrições temporais associadas a cada método invocado.

O modelo DRO suporta tanto a definição de tarefas periódicas ("active methods") quanto a especificação do tempo de execução de todos os métodos dos objetos de tempo real. Estas restrições temporais são associadas aos métodos dos objetos. Adicionalmente, o modelo pressupõe a existência de manipuladores de exceções tanto a nível de declaração como a nível de invocação de métodos, sendo que no caso de invocação também é previsto tratamento de exceção relativo a rejeição de uma invocação ("reject") e terminação anormal de um método invocado ("abort").

O mecanismo básico de interação entre objetos do modelo DRO é o polimorfismo temporal.

O controle de concorrência é realizado através do mecanismo de estados habilitados. Este mecanismo consiste na identificação dos possíveis estados em que um objeto pode encontrar-se, e na definição das transições que podem ser realizadas a partir de cada um desses estados com relação aos métodos dos objetos.

Destaca-se no modelo DRO sua flexibilidade relativa a interação entre objetos, decorrente da possibilidade de definição de protocolos a nível de programação e da introdução do mecanismo de invocação polimórfica temporal. Da mesma forma, a separação entre o controle da concorrência e das restrições temporais dos aspectos funcionais da aplicação (via filosofia de meta-objetos) contribui para o entendimento, manutenção e reusabilidade do software produzido.

Por outro lado, a capacidade reflexiva de DRO não é utilizada para definição e controle de novos tipos de restrições temporais, reduzindo a flexibilidade e a expressividade do modelo. Além disso, a questão de escalonamento tempo real não é considerada a nível de aplicação, sendo portanto dependente do ambiente operacional no qual o modelo está implementado.

1.2. Modelo de objetos "RealTimeTalk- RTT"

RealTimeTalk [FUR97] é um framework destinado ao desenvolvimento de aplicações tempo real, que caracteriza-se por assistir o projetista em todos os estágios do processo de desenvolvimento de sistemas tempo real, através de uma linguagem de descrição comum usada nos diferentes níveis de abstração deste processo. O principal objetivo de RTT é simplificar a modelagem e o projeto de sistemas tempo real previsíveis.

O modelo de objetos RTT define como uma aplicação deverá ser configurada, e, ao contrário dos demais modelos aqui descritos, não encapsula as restrições temporais nos objetos, mas sim estabelece-as em um nível de abstração mais alto. Segundo este

modelo, uma aplicação é decomposta em seus diferentes modos operacionais, juntamente com os modos-transição que especificam as atividades a serem realizadas quando ocorre uma transição de modo operacional.

Modos são refinados em um conjunto de "usecase's", os quais definem atividades específicas do sistema. Cada "usecase" é definido por um grafo de precedência e um grafo de relação de objetos, e possui um período associado; os grafos de relação de objetos são definidos com base nos PEO's (Parallel Executable Objects) que constituem um "usecase" e pelas relações existentes entre eles. Adicionalmente, objetos de interface e objetos de comunicação de dados também são visíveis em um grafo de relação de objetos. Os grafos de precedência definem a ordem na qual os métodos dos objetos serão escalonados; logo, o tempo de execução e o deadline de cada método devem ser definidos. Tais grafos são especificados a partir dos PEO's envolvidos em uma atividade, juntamente com os atributos temporais destes PEO's. PEO's são os objetos básicos do modelo, os quais destina-se a implementação das funcionalidades da aplicação.

No modelo RTT as restrições temporais não são especificadas explicitamente no corpo dos métodos dos objetos, e sim quando da definição dos grafos de precedência, onde cada tarefa é descrita por seu tempo de execução, seu deadline e seu "release time" relativo ao início do período especificado para o "usecase" que o grafo de precedência em questão está representado.

Exceções temporais em RTT podem ser definidas através de modos de execução, que por sua vez podem ser definidos a nível de sistema (na fase de configuração do sistema) e a nível de modo, podendo ser reconfigurados em tempo de execução.

No modelo RTT, a sincronização é especificada via grafos de precedência. Objetos RTT comunicam-se via passagem de mensagem, como no modelo convencional. Escalonamento em RTT é realizado off-line, por modo (o que significa que cada modo transição terá seu próprio "schedule") e com base nos grafos de precedência. As unidades de escalonamento (tarefas) são os métodos de ativação dos PEO's, os quais são não-preemptivos.

Um dos principais aspectos do modelo RTT é sua abrangência e uniformidade, envolvendo todas as fases do processo de desenvolvimento de sistemas tempo real. Aspectos estes que contribuem para a redução do "gap" semântico existente entre projeto e programação de sistemas.

Por outro lado, a especificação estática e em separado das restrições temporais e de precedência, embora adequada para sistemas tempo real hard, não satisfaz as necessidades de flexibilidade e integração que caracterizam a maioria dos sistemas tempo real soft e nem contribui para facilitar o gerenciamento da complexidade, o reuso e a manutenção destes sistemas. Além disso, o modelo proposto não é flexível com relação as questões temporais (restrições e escalonamento), cuja implementação é dependente do suporte de execução subjacente.

1.3. Modelo "R^2"

Proposto por Honda e Tokoro (citado em [FUR97]), "R^2" (Real-time Reflective) é um modelo de computação concorrente baseado em objetos e reflexão computacional, segundo o qual um sistema computacional é estruturado como uma coleção de objetos básicos controlados por meta-objetos. Tais objetos comunicam-se através de passagem de mensagem (como no modelo convencional) e executam concorrentemente; entretanto, apenas um método (um fragmento de um script) pode executar por vez, visto que os objetos são "single-threaded".

O modelo proposto é voltado para sistemas de tempo real soft, visto que não existe garantia com relação a satisfação de restrições temporais; o que o modelo oferece é a garantia de que a violação de qualquer restrição temporal será detectada e que as ações alternativas especificadas serão realizadas.

Além dos objetos básicos, o modelo R^2 introduz quatro tipos de meta-objetos: meta-objetos padrão, meta-objetos de tempo real, meta-objeto alarm-clock e meta-objeto escalonador. A interação entre os diversos tipos de objetos e meta-objetos que compõe o modelo R^2 é, esquematicamente, mostrada na figura 3. As funcionalidades destes objetos podem ser, resumidamente, assim descritas:

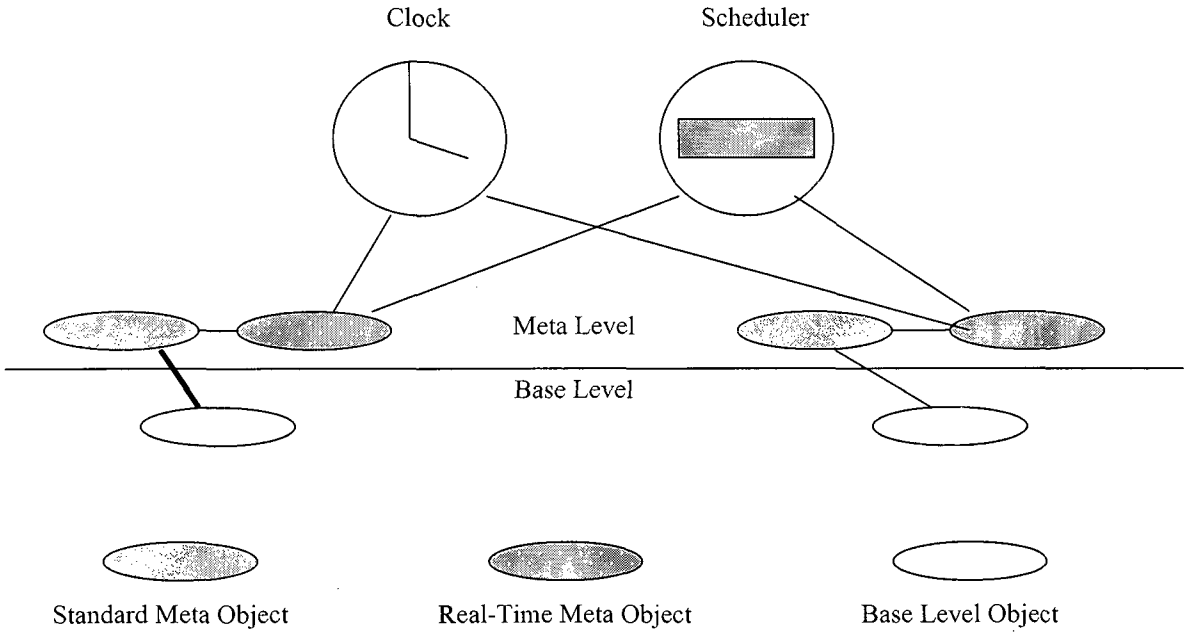


Figura 3 – Estrutura geral do modelo R^2

- Objetos Básicos – Seu script (correspondendo a uma coleção de métodos) implementa as funcionalidades da aplicação;

- Meta-Objeto Padrão – Responsável pelo controle de concorrência, pela manipulação de meta-mensagens relativas ao seu objeto básico e pelo suporte ao meta-objeto de tempo real correspondente;

- Meta-Objeto Tempo Real – Processa as especificações temporais das tarefas interagindo com os meta-objetos alarm-clock e escalonador, visando prover as propriedades de melhor esforço e menor prejuízo. Também é responsável pela detecção de eventuais violações das restrições temporais especificadas e pela realização das ações alternativas apropriadas.

- Meta-Objeto Alarm-Clock – É uma abstração da função clock provida pelo sistema operacional/hardware subjacente, cujas funções básicas são fornecer o valor corrente do clock do sistema e notificar a passagem de um determinado intervalo de tempo.

- Meta-Objeto Escalonador – Determina a ordem de execução das tarefas (fragmentos de código do script), visando satisfazer as restrições temporais especificadas e melhorar a utilização da CPU. Na arquitetura proposta existe flexibilidade para escolha de um algoritmo de escalonamento que seja mais adequado à aplicação em questão.

As restrições temporais básicas consideradas no modelo são o tempo de start e o deadline de uma tarefa (uma seção de um script), as quais são associadas a segmentos de códigos dos objetos-base e reportadas para seu meta-objeto de tempo real que ficará responsável pelo processamento das restrições temporais especificadas. Além disso, o modelo proposto habilita a introdução de novas construções temporais, visando melhor se enquadrar às especificações temporais relativas a uma determinada aplicação.

O modelo R^2 foi implementado como uma extensão da linguagem ABCL/R2, a qual é uma linguagem que incorpora os conceitos de concorrência de ABCL/1 e o conceito de reflexão individual de ABCL/R. Nesta implementação, os objetos e meta-objetos que compõem a arquitetura R^2 foram implementados com objetos ABCL/R2.

O modelo R^2 introduz uma nova e interessante forma de desenvolvimento de sistemas tempo real usando a idéia de reflexão computacional baseada em meta-objetos. As principais vantagens decorrentes desta filosofia residem na flexibilidade e na expressividade que podem ser obtidas tanto pela definição de novas construções temporais quanto pela utilização de diferentes algoritmos de escalonamento (introduzidos via meta-objetos), sem afetar a programação das funcionalidades da aplicação e do suporte subjacente.

Apesar de ser claramente endereçado para sistemas tempo real soft, o modelo proposto empenha-se no sentido de realizar as propriedades de melhor esforço e menor

prejuízo, buscando maximizar a taxa de satisfação das restrições temporais através do oferecimento de suporte para a definição de construções temporais e de algoritmos de escalonamento mais adequados a uma determinada classe de aplicações.

Um aspecto discutível do modelo R^2 é o fato de que as restrições temporais são associadas a segmentos de código e não a um "script"; desta forma, as restrições temporais especificadas não influenciarão o escalonamento dos "script's", mas somente o escalonamento de segmentos de código dos mesmos, reduzindo a efetividade do escalonamento realizado a nível de aplicação.

Adicionalmente, o esquema utilizado na expressão de restrições temporais dificulta a definição e o uso de restrições temporais do tipo time-trigger (como periodicidade, por exemplo), além de ferir a uniformidade do modelo de objetos e dificultar o reuso e a manutenção dos objetos-base.

Outro aspecto discutível no modelo R^2, é a não consideração do controle de sincronização condicional a nível de meta-objetos, uma vez que este aspecto pode afetar o escalonamento das tarefas do sistema. além disso, o modelo R^2 é fortemente influenciado pelo paradigma de programação de ABCL/R (programação por continuação), de forma que a implementação de seu comportamento em outras linguagens, implicaria na mudança de sua filosofia e estruturação básicas.

1.4. Real-Time Meta-Object Protocol (RT-MOP)

Mitchel (citado em [FUR97]), apresenta uma proposta inicial de um MOP (Meta-Object Protocol) de tempo real baseado em grupos de escalonamento, o qual tem como objetivo disciplinar e controlar mudanças em tempo de execução e servir como um mecanismo de estruturação de sistemas. Neste sentido o RT-MOP proposto pode ser visto como sendo um modelo de programação tempo real baseado em reflexão computacional.

Segundo o modelo proposto, um sistema tempo real deve ser estruturado na forma de objetos de aplicação (objetos-base) e meta objetos, sendo que cada objeto da

aplicação possui um meta-objeto associado, o qual é responsável pelo controle e pela modificação da estrutura e do comportamento do objeto de aplicação correspondente. Conjuntos de objetos de aplicação (e seus respectivos meta-objetos) são estruturados em grupos de escalonamento, sendo que cada grupo possui um meta-objeto "scheduler" responsável pelo escalonamento das tarefas de seus objetos constituintes segundo uma política de escalonamento própria; além disso, cada grupo possui também um gerente responsável pelo gerenciamento da entrada e saída de elementos no grupo (membership) em tempo de execução, sendo que o acesso a um grupo é controlado para manter garantias de escalonabilidade em tempo de execução. Adicionalmente, os meta-grupos de escalonamento também são reflexivos e podem ter seu comportamento alterado (mudança da política de escalonamento, por exemplo) através da invocação de operações do meta-meta-grupo.

No modelo proposto, nem todos os objetos da aplicação necessitam ser reflexivos, mas em qualquer caso as informações temporais necessárias ao escalonamento devem estar presentes na interface destes objetos; entretanto, na referência disponível [FUR97], nada é adiantado sobre a forma pela qual tais instruções são associadas aos objetos (ou a seus métodos...), e de que forma objetos não reflexivos serão considerados no escalonamento realizado pelo escalonador do grupo.

O modelo de concorrência proposto adota o conceito de objetos ativos formados pelo encapsulamento de tarefas (threads) dentro dos objetos; tal modelo é fortemente influenciado pelo modelo de concorrência usado em TAO [FUR97]. Cada objeto ativo pode possuir várias threads, as quais são criadas quando o objeto é criado e são executadas de acordo com o escalonador do grupo de escalonamento ao qual o objeto pertence. Este modelo suporta tanto concorrência externa quanto concorrência interna, as quais são controladas através do uso de métodos sincronizados.

A questão de como os grupos serão distribuídos em uma rede ainda é uma questão em aberto; entretanto os autores consideram o grupo como sendo a unidade adequada para distribuição, e propõem que grupos de escalonamento dentro de um sistema

possam residir em diferentes nodos da rede, não permitindo que os grupos possuam distribuição interna (visando tornar o escalonamento tempo real mais fácil).

Embora a proposta seja preliminar e muitos aspectos ainda estão por ser definidos, o uso de grupos de escalonamento é sem dúvida uma boa alternativa para prover a flexibilidade, de forma controlada, que muitos sistemas tempo real atuais necessitam; além disso por ser reflexivo, o modelo proposto herda as vantagens do uso do paradigma de reflexão computacional para aplicações tempo real. Contudo, a realização prática deste modelo depende da definição de uma estrutura reflexiva concreta no âmbito de alguma linguagem de programação.

2 O Modelo Reflexivo Tempo Real RTR

Além dos modelos anteriormente vistos, temos o Modelo Reflexivo Tempo Real RTR, descrito em [FUR97]. RTR é um modelo de programação reflexivo e de tempo real, que caracteriza-se por permitir, de forma flexível e sistemática, a representação e o controle de aspectos temporais de aplicações tempo real que seguem uma abordagem de melhor esforço. O Modelo RTR objetiva diminuir os problemas da programação de sistemas tempo real relacionados com a estruturação, falta de flexibilidade e representação/controle de restrições temporais.

2.1. O Modelo RTR

O Modelo RTR (Reflexivo Tempo Real) é um modelo de programação para aplicações tempo real que se caracteriza como sendo uma extensão reflexiva e tempo real do modelo de objetos convencional.

No modelo proposto, todos os aspectos temporais (restrições, exceções e escalonamento) e mais os aspectos de concorrência e sincronização são tratados de forma reflexiva, possibilitando o uso de diferentes mecanismos e políticas no controle do comportamento dos aspectos refletidos.

Por outro lado, o Modelo RTR estende o modelo de objetos convencional para suportar a representação e o controle de aspectos temporais. Esta capacidade é obtida a

partir da representação explicita das restrições temporais a nível de objetos-base e da verificação e controle destas restrições a nível de meta-objetos. Outra questão fundamental relativa a obtenção de correção temporal é a presença de um escalonador tempo real, escolhido pelo usuário, operando no meta-nível da aplicação.

Estruturalmente o modelo RTR é composto por objetos-base de tempo real, meta-objetos gerenciadores (um para cada objeto-base tempo real existente), um meta objeto escalonador e um meta-objeto relógio, os quais integram através de mensagens (ativações de métodos) síncronas e assíncronas, visando a realização das funcionalidades da aplicação de acordo com as restrições temporais associadas aos métodos dos objetos-base.

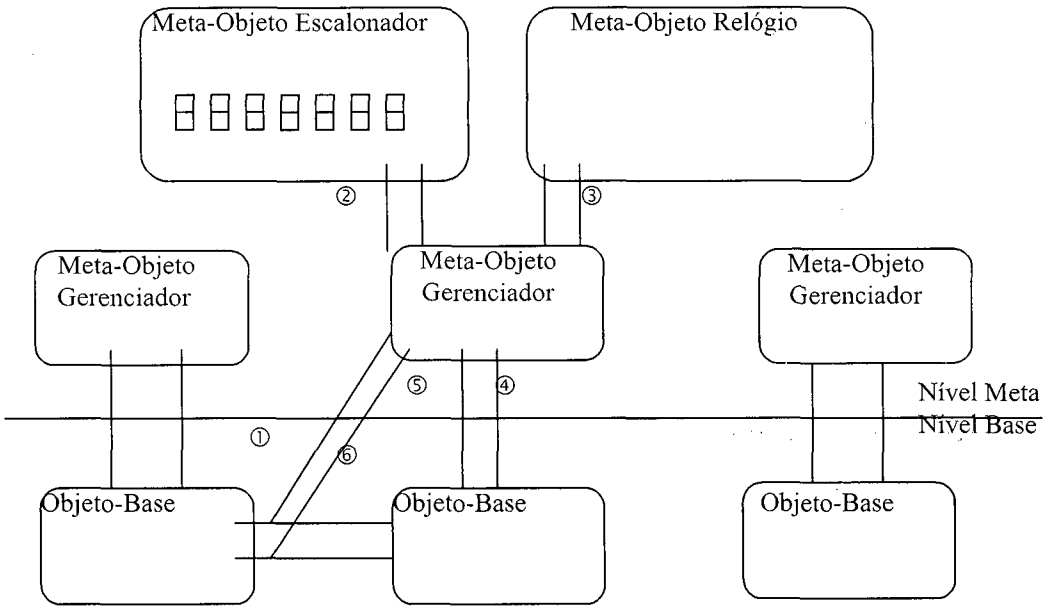


Figura 4 - Estrutura Geral do modelo RTR

Tomando como base a Figura 4, a dinâmica de funcionamento do modelo RTR pode ser assim descrita: Ativação de um método de um determinado objeto-base é desviado para seu meta-objeto gerenciador correspondente (ação 1 da figura 4). O meta-objeto gerenciador interage com o meta-objeto escalonador (ação 2) e com o meta-objeto relógio (ação 3) para processar as restrições temporais associadas ao método solicitado e verificará as restrições de concorrência e de sincronização; caso estas

restrições não sejam violadas, ativará o método solicitado no objeto-base (ação 4), retornando em seguida, via meta-objeto gerenciador para o objeto que deu origem a chamada(ações 5 e 6).

2.2. Visão Geral dos Componentes Básicos do Modelo

Objetos-base – Os Objetos-Base de tempo real implementam a funcionalidade da aplicação e, em adição ao modelo de objetos convencional, podem ter restrições temporais e manipuladores de exceções temporais associados à declaração e a ativação de métodos. Além das restrições temporais pré-definidas (Periodic, Aperiodic e Sporadic), novos tipos de restrições temporais podem ser declarados e utilizados pelo programador da aplicação. O exemplo a seguir ilustra a declaração de um método com uma restrição temporal do tipo "Periodic" associada; semanticamente a ativação deste método fará com que os frames que compõem a animação sejam apresentados na freqüência (período) especificada ou então sejam ignorados,

```
void Animation (...) Periodic (Period, EndTime, MaxET=10), FrameIgnore()

{ ... }
```

Meta-Objetos Gerenciadores (MOG) – Os meta-objetos gerenciadores são objetos multi-threads responsáveis pelo gerenciamento dos pedidos de ativação dos métodos de seus objetos-base correspondentes, pelo controle de concorrência e sincronização, pela implementação dos manipuladores de exceções temporais e pelo processamento das restrições temporais. Estas funcionalidades são tratadas separadamente nas diversas seções (gerenciamento, sincronização, exceções temporais e restrições temporais) que compõem o MOG.

Particularmente, na seção de restrições temporais são implementados os tipos de restrições temporais (um método para cada restrição) especificados nas declarações dos métodos do objeto-base. Para prover a semântica de execução correspondente a estas restrições temporais, os métodos que as implementam interagem com o meta-objeto escalonador (MOE), o qual determina, de acordo com a política de escalonamento

implementada, o momento a partir do qual o pedido de ativação que está sendo analisado pode ser liberado para execução. Neste momento, em função da disponibilidade de tempo e observados os aspectos de sincronização e concorrência (através da interação com as seções de sinalização e de gerenciamento), é decidido pela liberação da execução do método solicitado ou pelo levantamento de uma exceção temporal.

Meta-Objeto Escalonador (MOE) – Este meta-objeto tem como função básica receber, ordenar e liberar pedidos de escalonamento de métodos dos objetos base, de acordo com uma determinada política de escalonamento, visando influenciar o escalonamento realizado pelo suporte subjacente. A existência de um MOE no modelo possibilita que diferentes pedidos de escalonamento, originados do mesmo ou de diferentes meta-objetos gerenciadores, possam ser analisados globalmente e ordenados de acordo com uma política de escalonamento (escolhida pelo programador) que melhor se adapte às especificidades da aplicação em questão.

Meta-Objeto Relógio (MOR) – O MOR é uma abstração do relógio do sistema, estruturada na forma de objeto. Sua função básica é receber pedidos dos meta-objetos gerenciadores (MOG) para ativar métodos num tempo futuro. Adicionalmente, o MOR é utilizado para detectar violação de timeouts e eventuais violações de restrições associadas com métodos que estão aguardando para serem escalonados (na fila de escalonamento do MOE) ou que foram atrasados (na fila de pendências do MOG) por motivo de concorrência ou sincronização.

3 Linguagem Reflexiva

A partir do trabalho de Maes [MAE87], que apontou a reflexão como uma característica intrínseca de linguagens orientadas a objetos, a reflexão computacional passou a ser considerada um importante mecanismo de extensão estática e dinâmica de linguagens orientadas a objetos e mesmo para integração de diferentes paradigmas de programação.

As características reflexivas são implementadas de diferentes modos, dependendo da linguagem de nível base e seu enfoque particular do modelo de objetos. Os mecanismos de reflexão, da mesma forma que outros mecanismos de linguagens de programação, exigem uma associação ('binding') entre componentes de um programa.

A idéia básica da programação em meta-nível é tornar disponível partes da implementação da linguagem, para permitir a sua reprogramação. Para suportar o conceito de reflexão computacional, uma linguagem de programação necessita de mecanismos que : (a) permitam a conexão entre o nível base e o meta-nível; (b) tornam disponíveis, no meta-nível, informações sobre componentes do nível base.

Através da reificação, é possível ver componentes do programa e o estado da computação como dados internos, que podem ser modificados de forma a alterar dinamicamente o programa de nível base. Portanto, o programa de meta-nível pode ser visto como uma interface de alto-nível(meta-interface) ao programa de nível base, habilitando o usuário a fazer adaptações do programa para atender necessidades particulares.

Durante o processo de tradução (compilação ou interpretação) de um programa, o processador da linguagem possui todas as informações necessárias à implementação de características reflexivas: o nome de todas as classes e sua hierarquia de herança, nomes e tipos de todos os atributos e métodos definidos em todas as classes, e os nomes dos possíveis objetos a serem instanciados e as mensagens a serem enviadas. Estas informações são utilizadas pelo compilador, mas normalmente são descartadas ou escondidas após a geração do código objeto. A reflexão computacional exige que algumas destas informações sejam acessíveis ao programador, na forma de uma interface ao processador da linguagem.

Durante o processo de execução de um programa, as informações reflexivas sobre classes e objetos devem estar presentes no código objeto do programa, no caso de programas compilados, ou no ambiente de execução, no caso de programas interpretados.

A seguir veremos alguns modelos de linguagens reflexivas, baseadas nas linguagens Smalltalk, C++, Java e o Modelo Reflexivo RTR.

3.1. SMALLTALK

A linguagem SmallTalk possui, por sua concepção, características reflexivas, sendo particularmente adequada para a construção de aplicações baseadas no conceito de reflexão computacional. Sendo uma linguagem na qual todos os componentes são objetos, SmallTalk oferece uma organização típica do modelo: os objetos encapsulam seu próprio estado e a meta-classe correspondente descreve as suas propriedades estruturais. As meta-classes fornecem ao usuário as interfaces relacionadas com a implementação de todas as classes.

Entre as facilidades encontradas para a utilização de reflexão computacional, destacam-se:

- Todas as entidades da linguagem são consideradas como objetos, inclusive a entidade que contém descrições de objetos(classes). Sendo uma classe considerada um objeto, é possível fazer operações sobre a classe, inclusive alterando a estrutura dos objetos da classe. Assim, computação reflexiva estrutural de classes atua sobre instâncias da classe.

- Todas as classes de SmallTalk são instâncias de uma classe Object, que é a raiz da hierarquia de classes do sistema SmallTalk. A partir da classe Object são derivadas as demais classes e cada classe tem a sua representação descrita em uma meta-classe. Cada classe é uma instância de sua própria meta-classe. Por exemplo, uma classe denominada Point é uma instância da meta-classe Point class; isto significa que todos os objetos instanciados a partir da classe Point são descritos pela meta-classe Point class. A figura abaixo esquematiza o relacionamento entre as instâncias p1,p2 e p3, suas classes ancestrais Point e Object, e suas respectivas meta-classes.

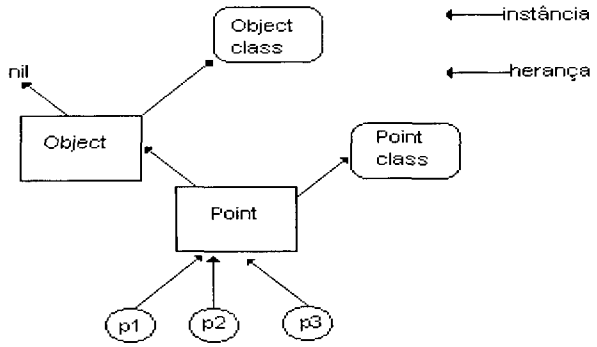


Figura 5 - Relacionamentos entre Classes

Todas as meta-classes são instâncias da classe MetaClass. Portanto, a meta-classe PointClass é uma subclasse de Object class, e também uma instância de MetaClass. As facilidades reflexivas são implementadas em meta-classes especiais da hierarquia, de forma distribuída.

O ambiente Smalltalk fornece a representação dos principais componentes das classes, como informações a respeito de hierarquia e representação das variáveis da classe, que podem ser acessados e modificados por outros objetos em tempo de execução. A liberdade de, por exemplo, alterar a hierarquia de classes ou a representação de uma variável, torna este modelo bastante dinâmico e extremamente frágil sob o aspecto de robustez de programas.

3.2. Open C++

A linguagem C++ [FUR97] não oferece, por concepção, facilidades reflexivas. Para possibilitar acesso, em tempo de execução, às informações a respeito das classes e objetos do programa é necessário modificar a implementação da linguagem ou fazer extensões baseadas em pré-processadores.

O pré-processador OpencC++ - Versão 1.2 [CHI93] adota um modelo de reflexão computacional no qual chamadas de métodos e acessos a variáveis de instância são

tornados disponíveis ao programador, em tempo de execução. O protocolo possui as seguintes características:

- Meta-objeto: objetos selecionados do programa podem ser instanciados como objetos reflexivos, associados a um meta-objeto; outros objetos da mesma classe podem ser instanciados como objetos não reflexivos.

- Relação: a associação é única entre objetos e meta-objetos, isto é, um meta-objeto não pode ser compartilhado por mais de um objeto;

- Controle: um meta-objeto pode controlar mensagens dirigidas a seus referentes (chamadas de função-membro) e acesso à variáveis de instância (variáveis-membro);

- Todas as classes usadas para instanciar os meta-objetos descendem da classe MetaObj;

Um programa em OpenC++ distingue-se de um programa C++ apenas por suas diretivas, incluídas no texto fonte sob a forma de comentários que iniciam por //MOP. O texto resultante é compilado por OCC (compilador de OpenC++), o qual gera um arquivo de trabalho para cada classe reflexiva e dá prosseguimento à compilação, chamando o compilador C++. Exemplo: //MOP reflect class Person: VerboseClass

Meta-objetos podem ser criados como classes comuns de C++, tendo como classe ancestral a classe MetaObj, a qual define os métodos que um meta-objeto pode usar para controlar seu referente. Por ser uma classe comum de C++, um meta-objeto pode receber herança múltipla, i.e., ter uma ou mais classes ancestrais além de MetaObj, e pode sofrer especializações, i.e., pode ser usado como classe base de outras classes descendentes, assim transmitindo indiretamente o protocolo herdado da classe MetaObj.

A computação no meta nível é realizada por redefinição dos métodos adquiridos da classe MetaObj e pelos métodos particulares (definidos ou herdados) da classe do meta-objeto em questão. Os métodos oferecidos pela classe MetaObj se dividem em três categorias:

1 – Métodos que implementam chamadas e acessos à variáveis: são executados quando um método reflexivo é chamado ou uma variável reflexiva é acessada.

2 – Métodos usados pelo meta-objeto para executar operações sobre o objeto de nível base: incluem métodos usados para executar um método de nível base, consultar/atribuir valor de variável reflexiva.

3 – Métodos diversos usados para operações internas de reflexão, como construtores/destruidores especiais e informações sobre nomes de classes, métodos e identificadores de variáveis de objetos reflexivos.

Em resumo, o protocolo de OpenC++ - Versão 1.2 oferece um ambiente de programação reflexiva que possibilita que certos aspectos de um programa C++ sejam redefinidos no meta-nível, através de interfaces bem definidas. Sendo implementado através de um pré-processador, suas características reflexivas são estabelecidas de forma estática.

Na sua versão mais recente, o protocolo de OpenC++ - Versão 2.0 [CHI96] é compilado, originando meta-classes que são tratadas pelo programa como objetos, de forma similar à SmallTalk. Os principais conceitos envolvidos são:

- class object: representação de uma classe no meta-nível;
- metaclass Class: uma classe cuja instâncias são meta-objetos;
- metaclass Class: a meta-classe padrão;

3.3. JAVA

Devido a emergência de novas áreas de aplicações , tais como aplicações para computação móvel e multimídia interativas, sistemas de software necessitam cada vez mais atender a demanda e expectativa dos usuários. Porque tais aplicações têm variações e até mesmo exigências contrárias.

Computação móvel define um mundo de computação dinâmico com intermitentes comunicações. Em momentos diferentes, computadores são usados em diferentes pontos de acesso na rede. Sua conectividade pode variar ambos em desempenho e confiabilidade; a variação da largura de banda e latência pode ser grande. Para permitir aplicações executarem em diferentes ambientes, diferentes configurações de suporte a sistemas de software podem ser fornecidas em cada ambiente. Java ficou popular como uma linguagem de programação para aplicações móveis na Internet porque ela tem habilidade de simplificar o desenvolvimento tornando-o flexível para, aplicações portáteis com interfaces gráficas de usuários [JAV96]. Usando Java um usuário pode escrever um programa e executar a qualquer hora em qualquer plataforma. Esta é a exigência chave para Internet onde um programa carregado deve ser capaz de executar em qualquer computador do mundo. Porém, Java não fornece um bom mecanismo para suportar adaptabilidade de software de sistema. Java usa o método API para implementar serviços subjacentes. Embora estes métodos trabalhem bem para fornecer funcionalidade básica tal como I/O, redes e threads (funcionalidades que estão em aplicações independentes). Ele é pouco apropriado para fornecer funcionalidades a sistemas, tais como distribuição de processamento, controle de concorrência e dados persistentes (funcionalidades que são maiores em aplicações dependentes).

Um dos destaques de Java é que foi projetada com a suposição que o ambiente em que está executando pode ser mudado dinamicamente. Classes são carregadas dinamicamente, ligações são feitas dinamicamente, e instâncias de objetos são criadas dinamicamente quando necessárias. O que não tem se mostrado muito dinâmico historicamente, é a habilidade de manipular classes "anônimas". No contexto, classes anônimas são classes que são carregadas ou apresentadas para a classe Java em tempo de execução e possui um tipo que não é reconhecido previamente pelo programa Java.

Com a introdução do suporte a reflexão através do API (Application Program Interface), programas Java podem agora "refletir" sobre eles mesmos ou sobre uma classe arbitrária para determinar os métodos e campos definidos pela classe.

Fundamentalmente, o API de Reflexão consiste de dois componentes: objetos que representam as várias partes do arquivo de classes, e os meios para extrair estes objetos num modo seguro. O último é o mais importante, como Java fornece muita proteção de segurança, poderia não fazer sentido fornecer um conjunto de classes que invalidasse estas proteções.

O primeiro componente do API de Reflexão é o mecanismo usado para buscar informações sobre a classe. Este mecanismo é embutido na classe nomeada Class. A classe especial Class é um tipo universal para as meta informações que descrevem objetos dentro do sistema Java.

Métodos novos para java.lang.Class para JDK1.1 incluem: getDeclaredConstructor, getDeclaredConstructors, getMethod, getMethods, getField e getFields.

Introspeção é o termo que melhor caracteriza este modelo de reflexão de Java: métodos permitem examinar a estrutura de classes e objetos da aplicação. As meta-informações são tornadas disponíveis por meio de classes específicas, que separam as informações sobre classes, atributos, construtores, métodos e outros.

Com base neste exame, é possível a realização de algumas ações no meta-nível, como a conversão de valores de variáveis públicas de classe ou de instância, criar novas instâncias de uma classe e executar métodos, com chamadas feitas no meta-nível. Porém, não é possível fazer a interceptação de chamadas a métodos, e, como conseqüência, o meta-nível não dispõe de informações dinâmicas, como argumentos usados na chamada de um método.

O modelo abstrato da Máquina Virtual Java – JVM determina que todas as classes e interfaces são objetos do tipo class. Objetos do tipo Class são automaticamente criados quando novas classes são carregadas para serem executadas, e contém informações sobre o nome da classe, a super-classe, a lista de interfaces implementadas e o carregador de classes utilizado. Estes objetos do tipo Class são apenas descritores de classes, não sendo permitido alterar a estrutura de classes.

As informações de identificação, como os nomes das classes e da super-classe de um objeto são disponibilizados ao programador através de chamadas de métodos que se referem ao próprio objeto (por exemplo, this.getClass()). Portanto, por concepção, a linguagem java possui características reflexivas, onde informações sobre objetos do programa são fornecidas em tempo de execução pelo seu ambiente.

A partir de 1996, a linguagem Java passou a incluir novos mecanismos de Reflexão Computacional, através da extensão da linguagem Java onde meta-objetos são inseridos em seu ambiente de execução. Sendo recentes, estas extensões ainda não estão consolidados e não permitem determinar um único modelo de reflexão em Java. A seguir, serão brevemente descritas:.

3.4. A Extensão MetaJava

Esta Extensão proposta por Golm (analisado em [LIS97]), implementa reflexão estrutural e comportamental em Java através da extensão da máquina virtual Java – JVM.

MetaJava permite a associação de meta-objetos a objetos, classes e referências, e utiliza eventos síncronos para realizar a transferência da computação do programa de nível base para o programa de meta-nível. Se um meta-objeto é associado a um objeto, todos os eventos originados no objeto referente são transmitidos ao meta-objeto correspondente; quando associado a uma classe, todos os eventos originados nas instâncias da classe referente são transmitidos ao meta-objeto; e, quando associado a uma referência, somente os eventos gerados por operações que usam essa referência são transmitidos ao meta-objeto correspondente.

Múltiplos meta-objetos podem ser associados a um componente de nível base, formando uma hierarquia (torre) de meta-objetos, na qual a ordem é importante: o meta-objeto de mais recente associação é executado em primeiro lugar.

A extensão MetaJava foi implementada através da modificação da máquina virtual Java, em dois níveis: o nível inferior fornece a interface de meta-nível para a máquina

virtual Java e o nível superior implementa mecanismos de transferência de controle para o meta-nível.

A interface de meta-nível – MLI fornece métodos que fazem a reificação a máquina virtual Java, disponibilizando estes dados ao programa de meta-nível. Estes métodos são agrupados nas seguintes categorias:

- Acesso a dados de instância: fornecem acesso aos nomes, tipos e estrutura das variáveis de um objeto.

- Suporte para modificação de código: faz a reificação do código de um método, fornecendo a seqüência de bytecodes correspondente.

- Associação entre o nível base e o meta-nível: associa um meta-objeto a um objeto de nível base e fornece informação sobre o meta-objeto associado a um objeto.

- Modificação da estrutura de classes: cria e instala uma classe 'shadow' de um objeto; alterações estruturais podem ser realizadas sem afetar outros objetos da mesma classe.

- Modificação de constantes: permite alterar as constantes de uma classe.

- Suporte para código nativo: permite instalar e executar código nativo, diretamente pelo processador.

- Execução: permite executar métodos arbitrários.

- Criação de instâncias: cria novas instâncias de uma dada classe.

MetaJava é bastante flexível, permitindo associações de meta-objetos a componentes de nível base representados por classes, objetos e referências, e fornecendo ao programa de meta-nível informações que devem ser cuidadosamente utilizadas. A reflexão estrutural viola o encapsulamento de classes e a torre de meta-objetos pode provocar conflitos semânticos.

Implementação

A versão inicial de MetaJava usou uma biblioteca compartilhada para estender a Máquina Virtual Java Sun's (JVM). O desenvolvimento adicional requer extensas mudanças para o JVM, assim decidiu-se construir uma máquina virtual própria, a MetaJava Máquina Virtual (MJVM). O MJVM é um superset do JVM. Usa o mesmo formato do arquivo de classe e executa o mesmo bytecode fixado como o JVM, mas fornece uma interface de meta-nível para a máquina virtual .

3.5. Sistema MetaXa

O sistema MetaXa [LIS97] pode ser considerado uma evolução do sistema MetaJava. O sistema MetaXa foi projetado para ser um sistema reflexivo, que além de permitir reflexão estrutural, também fornecesse de algum modo reflexão comportamental.

Em um sistema reflexivo deve ser claro e fácil de compreender:

- como o meta sistema é conectado ao sistema base.
- como esta conexão pode ser customizada.
- como a mudança de nível é realizada.

Na maioria dos sistemas reflexivos há três tipos de código:

- código de nível-base que resolve o problema de aplicação
- código de nível-base que adapta o código de nível-base para diferentes ambientes.
- código de configuração que conecta base e meta.

O objetivo de MetaXa é separar estas três categorias de código, conservando tipos seguros e eficientes.

O propósito para reflexão em MetaXa não é baseado em linguagem, mas baseado em sistemas. Então não houve uma extensão de linguagem porém foi construída uma própria extensão da Máquina Virtual Java (JVM).

Eventos são os métodos de MetaXa para controle de transferência de um nível-base para o nível-meta. Há eventos para vários mecanismos da JVM, como invocações de métodos, acessos à variáveis, operações de monitor, criação de objeto, ou carga de classe. O manipulador de eventos do meta objeto pega um objeto evento que contém a informação necessária para executar a operação requisitada. Eventos ocorrem sincronamente, isto significa, que a computação base é suspensa até que os eventos sejam tratados.

Um metaobjeto tem acesso a máquina virtual por uma interface-de-meta-nível (MLI). A MLI contém métodos para reflexão estrutural, além de alguns métodos auxiliares para, por exemplo, criar objetos cujo estado (suas variáveis de instância) é completamente administrado pelo metaobjeto. Nenhuma memória é reservada para tal objeto pela JVM. Este mecanismo de criação é útil para array esparsos ou para objetos que servem como stubs ou proxies.

O sistema MetaXa usa uma linguagem popular, podendo assim contar com grandes bibliotecas e ambientes de desenvolvimento sofisticados . Devido a separação de código do nível-base e código do meta-nível o programador pode começar com programação normal e gradualmente introduzir metaobjetos.

As vantagens de MetaXa podem ser resumidas como segue:

Há uma relação n-para-n entre classes e metaobjetos de instâncias de classe. Vários sistemas reflexivos conectam a classe do objeto de nível-base e a classe do meta-objeto, em uma fase inicial de desenvolvimento. Em tais sistemas não é possível usar metaobjetos diferentes para objetos de nível-base da mesma classe. O sistema MetaXa permite modificação da relação de metaobjeto em tempo de execução.

Metaobjetos tem controle completo através de argumentos e passagem de valor de retorno. Contanto que satisfaça as regras de tipo estático, quaisquer dados podem ser passados como parâmetro ou valor de retorno.

Por outro lado, MetaXa tem algumas desvantagens. A maior desvantagem é a perda do suporte da linguagem para programação reflexiva. Durante a configuração do metaobjeto e comunicação pelo MLI, o programador têm que se referir a certas entidades do modelo de programação.

3.6. A extensão Reflective Java

Numa linguagem de programação completamente reflexiva, todas as propriedades de uma linguagem são acessadas e mudadas pelos programadores(em um modo controlado). Porém, tornar Java completamente reflexivo poderia ser uma tarefa muito complexa, e não pode ser feito sem apoio de sua máquina virtual e compilador.

Para manter Java Reflexivo simples e seguro, este protocolo fornece propriedades limitadas de reflexão, as quais são simples, mas poderosas o bastante para habilitar substituição de um sistema alternativo a nível componentes .

A extensão de reflexão Java Reflexivo descrito por Wu [WU97] somente oferece reflexão de métodos. Quando um método é chamado, o seu meta-objeto correspondente intercepta esta chamada e passa a fazer computações no meta-nível. Deste modo, os programadores podem fazer invocações de métodos se comportarem conforme suas necessidades particulares pela implementação de metaobjetos.

De forma similar a Open C++, Java Reflexivo utiliza um pré-processador para implementar a reflexão. O pré-processador cria uma classe reflexiva como uma subclasse da classe da aplicação. A classe reflexiva possui uma instância de uma classe metaobject, que implementa a interceptação de chamadas aos métodos do objeto referente. O objeto referente é instanciado a partir da classe reflexiva.

Java Reflexivo possibilita aos programadores mudar o comportamento dos métodos invocados, pela especificação do que deve feito antes ou depois da execução do método normal. Por exemplo, figura 6 mostra como fornecer controle de concorrência para um objeto que é implementado para um ambiente seqüencial. O objeto é ligado a um metaobjeto que bloqueia o objeto e faz um backup de seu estado antes de chamar o método do objeto original, desbloqueia objeto e por fim libera o backup.

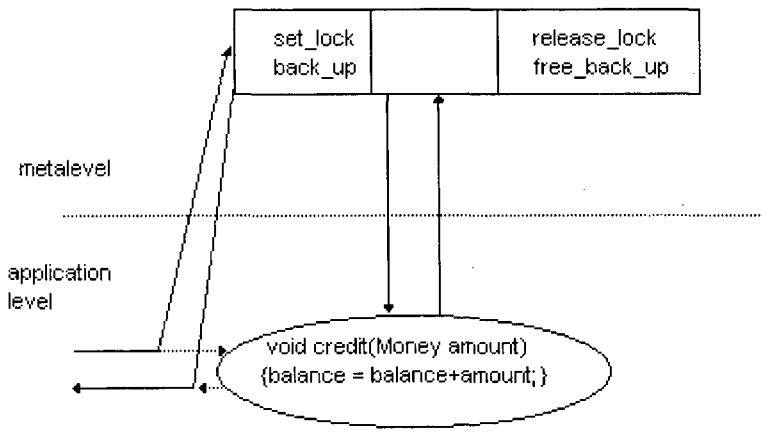


Figura 6: Comportamento de uma chamada a um método Reflexivo

A partir do exemplo pode-se ver que usando reflexão, controle de concorrência é direto, e totalmente transparente para programadores. Outras questões típicas de sistema também podem ser dirigidas deste modo.

Implementação

A questão chave de Java Reflexivo é encontrar um modo de interceptar uma invocação de métodos, sem exigir fazer qualquer mudança na linguagem Java em seu compilador, ou em uma máquina virtual. A questão esta sendo resolvida pelo uso de classes.

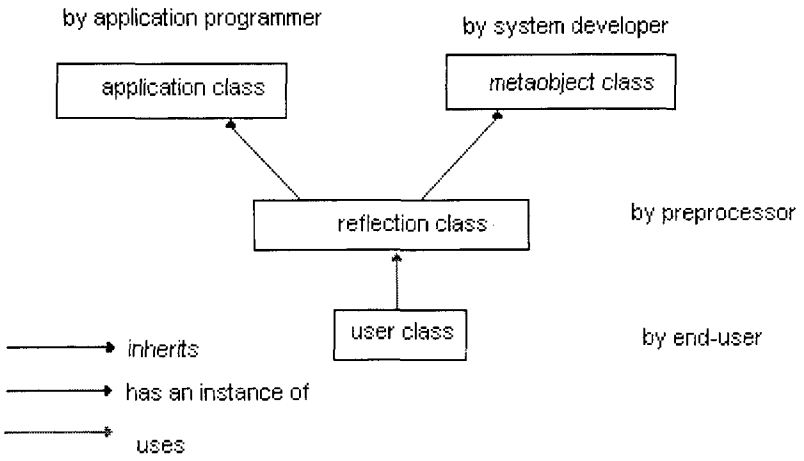


Figura 7 - Ligação de Objetos.

A idéia geral é implementar uma classe reflexiva como uma subclasse de uma classe da aplicação. Cada operação de uma classe de aplicação é sobrescrita na classe reflexiva de modo que uma invocação seja redirecionada para um metaobjeto.

Isto é descrito na figura 7. Uma classe reflexiva tem metaobjeto variáveis que são referência para uma instância de um metaobjeto de classe.

Deste modo, se programadores criam objetos ao invés de classes reflexivas instanciadas da classe de aplicação original, a invocação de um método objeto irá ser interceptada e negociar com o metaobjeto.

Uma classe reflexiva é gerada automaticamente para uma classe de aplicação pelo preprocessador Java Reflexivo. Uma classe reflexiva também fornece um par de métodos públicos, getMeta e changeMeta, para checagem e mudança do metaobjeto de um objeto de aplicação.

Java Reflexivo tem fornecido a flexibilidade exigida para implementar aplicações flexíveis e abertas. Faltam porém algumas questões como por exemplo, desenvolver conceitos que permitam usar flexibilidade de modo confortável, porém controlado. Por

exemplo, se um objeto da aplicação ligado a um metaobjeto que forneça algum mecanismo de controle de concorrência, tiver seu metaobjeto mudado dinamicamente, e o novo metaobjeto não fornecer um controle de concorrência correto, o sistema entrará em um estado inconsistente. Dessa forma, alguns mecanismos podem ser necessários para evitar mau uso da flexibilidade permitida por Java Reflexivo.

3.7. OpenJava

OpenJava é uma extensão de linguagem baseado em Java [LIS97]. As características estendidas de OpenJava são especificadas por um programa de meta-nível em tempo de compilação. Por distinção, programas escritos em OpenJava são chamados de programas de nível-base. Se nenhum programa meta-nível é determinado, OpenJava é idêntico a Java.

O programa de meta-nível estende OpenJava pela interface chamada OpenJava MOP (Protocolo de Metaobjeto). O Compilador de OpenJava consiste em três fases: preprocessador, tradutor de fonte-para-fonte de OpenJava para Java, e o back-end compilador de Java. O MOP de OpenJava é uma interface para controlar o tradutor na segunda fase. Permite especificar como uma característica estendida de OpenJava é traduzida em código de Java.

Uma característica estendida de OpenJava é fornecida como um software adicional para o compilador. O software adicional não só consiste de um programa de meta-nível mas de um código suporte para tempo de execução. O código suporte para tempo de execução fornece classes usadas pelo programa de nível-base traduzido em Java. O programa de nível-base em OpenJava é traduzido primeiro em Java pelo programa de nível-meta e é dinamicamente linkado com o código suporte em tempo de execução.

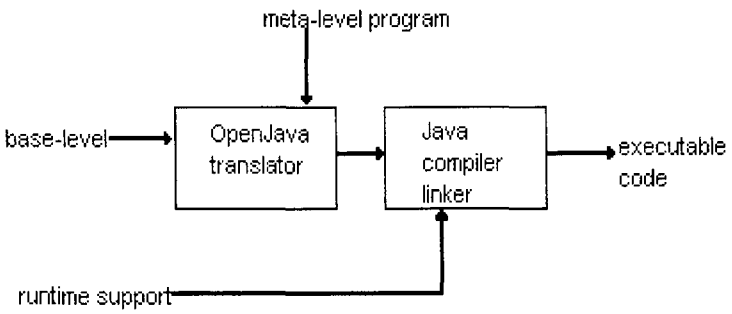


Figura 8 – Data Flow of OpenJava Compiler

O MOP Programado em OpenJava segue os três passos:

- decide com o que o programa de nível-base deveria se parecer,
- Compreende o que deveria ser traduzido e que código suporte é necessário em tempo de execução,
- escreve um programa de meta-nível para executar a tradução e também escreve o código suporte em tempo de execução.

3.8. Javassist.

Proposta por Chiba esta extensão de Java propõem reflexão comportamental e estrutural.

Uma das extensões propostas para a API de Reflexão de Java com a habilidade de interceptar uma operação do método invocado e alterar seu comportamento. Se uma operação é interceptada, em tempo de execução, o sistema chama o método no meta-objeto para notifica-lo daquele evento. O programador pode definir sua própria versão do metaobject, assim que o metaobjeto executa a interceptação da operação com a semântica customizada.[CHI00].

Reflexão comportamental somente provê a habilidade de alterar o comportamento da operação no programa, mas não provê a habilidade para alterar a estrutura dos dados usados no programa, que são fixados estaticamente em tempo de compilação.

A habilidade chamada reflexão estrutural permite um programa mudar, por exemplo, a definição da classe, do método, ou de um campo.

O API e reflexão Java não provê capacidade de reflexão plena. Ela não habilita alteração comportamental do programa, mas somente suporta introspecção da estrutura de dados.

Javassist somente permite alteração do comportamento do tipo específico de operação, tal como chamada de métodos, acesso a dados e criação de objetos. O programador pode selecionar algumas dessas operações e alterar seu comportamento.

API Javassist

Reificação e Reflexão – O primeiro passo para usar o Javassist é criar a CtClass (Compile Time Class) do objeto representando o bytecode da classe carregada na JVM. Se stream é um InputStream para uma classe então:

CtClass c = new CtClass(stream);

Cria um novo objeto CtClass representando o bytecode da classe lida do arquivo class, o qual contém muitas informações pra reificar a classe. Assim, o construtor da CtClass pode receber uma String nomeando a classe instanciada por InputStream.

Alteração - Uma diferença entre Javassist e a API padrão de reflexão Java é que Javassist provê métodos para alterar definições de classes. Estes métodos são categorizados nos métodos de troca, classes, hierarquia e métodos para adicionar novos membros.

Tipos de Implementação – Javassist provê três tipos de implementação:

- BCA (Adaptação de Código Binário) – que altera automaticamente a definição da classe de acordo com o arquivo escrito pelo usuário;

- Reflexão comportamental – A idéia chave desta implementação é inserir "hooks" (ganchos) no programa quando a classe é carregada na JVM.

- Método de invocação Remota – gerando um stub para invocar remotamente um método através do API RMI.

Javassist permite que o programa altere a definição de classe e a definição dinâmica de novas classes, além de implantar com simplicidade a reflexão estrutural e comportamental.

Para evitar problemas de performance e portabilidade, tabela1, o projeto do Javassist é baseado na sua nova arquitetura de reflexão estrutural. Javassist chama reflexão estrutural pela manipulação do bytecode da classe carregada. Além do mais, pode ser usada com a JVM e compilador padrão do Java. Embora reflexão estrutural ser permitidas somente antes da classe ser carregada na JVM.

Assim a JVM padrão é usada, e as classes são processadas pelo Javassist são objetos para o bytecode ser verificado pela Security Manager do Java. Javassist nunca quebra a garantia de segurança oferecida pelo Java.

Aspectos importantes do Javassist:

- Javassist é portável. Ele é implementado m Java puro e roda com a JVM padrão. Não tem dependência de plataforma.

- Provê abstração no nível de código para manipulação de bytecode de maneira segura quando traduz o bytecode.

- Nunca necessita do código fonte para fazer reflexão.

A arquitetura utilizada pelo Javassist pode ser aplicada para outras linguagens orientadas a objeto.

Tabela 1 – Comparação de performance entre Javassist e OpenJava [CHI00]:

		Tempo de execução (msec)	Tamanho do programa (linhas)	Quantidade de linhas do código original	Tamanho do arquivo class original (bytes)	Modificação do arquivo class (bytes)
BCA	Javassit	42	26	24	372	551
	OpenJava	543	17	24		548
Reflexão	Javassit	142	205	35	946	3932
	OpenJava	4108	247	35		2244

5. Proposta de um Mecanismo de adaptação para aplicações tempo real, baseado em Computação Imprecisa e Reflexão Computacional

Diversas aplicações com restrições de tempo real são disseminadas através da Internet na forma de applets Java ou componentes Active-X. A satisfação das restrições temporais torna-se um problema, pois estas aplicações devem executar em diferentes plataformas (processadores, sistemas operacionais), as quais apresentam os mais variados níveis de desempenho e ocupação. Um problema importante é como projetar os componentes do software de forma que eles apresentem desempenho satisfatório nestas diferentes plataformas. Neste sentido, busca-se mecanismo de adaptação para aplicações tempo real disseminadas através da Internet ou mesmo de uma rede de computadores de uso geral. Uma das técnicas que possibilitam esta adaptação é a Computação Imprecisa, na medida em que ela flexibiliza o tempo de execução das tarefas. O uso da Reflexão Computacional facilita a implementação da Computação Imprecisa, separando as questões funcionais das questões de controle responsáveis pela adaptação da aplicação.

O objetivo deste trabalho é mostrar como variações da Computação Imprecisa, implementadas através da Reflexão Computacional, podem ser utilizadas para permitir a adaptação de aplicações de Tempo Real a diferentes plataformas no contexto da Internet ou fora dela. O modelo de programação RTR [FUR97] será usado para ilustrar e implementar a forma como esta adaptação pode ser realizada.

A principal motivação para este trabalho está na dificuldade de atender restrições temporais quando aplicações devem executar em diferentes plataformas (processadores

e sistemas operacionais), as quais apresentam os mais variados níveis de desempenho e ocupação. Embora este seja um problema geral, ele torna-se mais relevante na medida em que aplicações são disseminadas através da Internet, em escala global.

1 Mecanismos de Adaptação para a Internet

No caso de uma aplicação tempo real disseminada via Internet nada pode ser suposto a respeito do sistema operacional alvo. Ao ser disseminada através da Internet, e chegar a uma nova plataforma para execução, uma aplicação tempo real terá que se adaptar ao desempenho desta plataforma. Neste caso, a adaptação deverá ocorrer através de uma ação unilateral da aplicação. Atualmente a maioria dos programas ignora este problema e não oferece qualquer tipo de mecanismo de adaptação. Desta forma, o usuário fica sujeito ao desempenho de uma aplicação executando em uma plataforma diferente daquela para a qual ela foi projetada.

É suposto que a aplicação tempo real possui requisitos temporais não críticos que deverão ser atendidos mesmo quando esta aplicação executa em plataformas com diferentes níveis de desempenho. É suposto ainda que o suporte de execução (middleware, sistema operacional, arquitetura) ignora os requisitos temporais da aplicação e fornece a mesma qualidade de serviço, não negociável, para todas as aplicações. Ainda, a qualidade do serviço que a aplicação tempo real recebe do suporte durante sua execução pode variar em função do início e término de outras aplicações que compartilham os recursos existentes. Logo, existe a necessidade de uma adaptação contínua e não somente durante a etapa de inicialização. Podemos ver aqui, conforme [OLI99], mecanismos de adaptação descritos na literatura.

1.1. Atrasar a conclusão da tarefa

A forma mais simples e freqüente de adaptação é simplesmente relaxar o conceito de deadline. Um dos primeiros trabalhos seguindo esta abordagem propondo que a conclusão de cada tarefa contribui para o sistema com um benefício e o valor deste benefício pode ser expresso em função do instante de conclusão da tarefa (time-value function). O conceito tradicional de deadline seria uma simplificação desta visão mais

ampla. Qualquer aplicação que ignore os aspectos temporais está, de certa forma, implementando este mecanismo. Entretanto, o conhecimento dos deadlines e de suas respectivas importâncias permite uma degradação mais suave do que quando deadlines são perdidos de forma aleatória.

1.2. Variar o período da tarefa

Em uma aplicação tempo real muitas tarefas são executadas periodicamente. Por exemplo, a exibição dos quadros de uma animação, o processamento de informações de áudio e vídeo, o ciclo de execução de uma máquina de simulação suportando um vídeo-game. Em geral estas tarefas possuem o seu período definido em tempo de projeto. Uma forma de prover adaptabilidade à aplicação é permitir que este período possa variar dinamicamente, durante a execução da aplicação. Desta forma, a qualidade da aplicação, representada aqui pelo período das tarefas, seria adaptada ao desempenho do suporte onde estiver executando. Por exemplo, a taxa de exibição dos quadros em um vídeo (frame rate) pode ser alterada conforme o desempenho do suporte onde a aplicação executa.

1.3. Cancelar a execução de uma tarefa

Em uma forma mais radical de flexibilização é simplesmente não executar algumas tarefas quando o desempenho estiver abaixo do desejado. No caso de aplicações com tarefas repetitivas, é possível cancelar uma ativação específica da tarefa ou cancelar completamente a tarefa. Embora o cancelamento de ativações isoladas seja menos perceptível, existem situações onde uma tarefa repetitiva pode ser completamente cancelada. Em [OLI99] é mostrada uma solução para situações de sobrecarga onde inicialmente são descartadas ativações individuais e, caso a sobrecarga persista, passam a ser descartadas tarefas completas.

1.4. Alterar o grau de paralelismo

Muitos algoritmos dividem o trabalho a ser feito em partes. As partes podem então ser executadas em paralelo, com o objetivo de diminuir o tempo de resposta da tarefa.

Caso a arquitetura que estiver sendo usada não suporte execução em paralelo, as partes são executadas seqüencialmente. O resultado é o mesmo, apenas o tempo de execução será maior. Na medida em que as arquiteturas paralelas tornam-se mais comuns, este mecanismo de adaptação poderá ser utilizado para tirar proveito do paralelismo existente, sem impedir a execução da aplicação em máquinas monoprocessadas.

1.5. Variar o tempo de execução da tarefa

Neste mecanismo de adaptação, as tarefas são escalonadas de forma a cumprirem os seus deadlines. Em caso de sobrecarga, o tempo de execução da tarefa é reduzido. Para isto, é necessário que cada tarefa possua opções do tipo "qualidade versus tempo de execução" (Computação Imprecisa).

2 Reflexão Computacional X Tempo Real

Uma das formas de fazermos adaptação de Sistemas Tempo Real a diferentes plataformas é usando os recursos da Reflexão Computacional, pois conforme podemos ver em [OLI99], embora pouco explorada no domínio tempo real ([FUR97] e [MIT97]), a abordagem de reflexão computacional é vista como uma abordagem promissora no que se refere à estruturação de sistemas tempo real complexos [STA96], apta a contribuir nas questões de flexibilização e gerenciamento da complexidade dos sistemas tempo real atuais e futuros.

O uso da reflexão no domínio tempo real permite:

- adicionar ou modificar construções temporais (via meta-objetos) específicas de um domínio (ou de uma aplicação);

- definir um comportamento alternativo para o caso de exceções temporais (não satisfação de deadlines, por exemplo);

- substituir/alterar o algoritmo de escalonamento, adequando-o a aplicação e/ou ao ambiente em questão;

- mudar o comportamento do programa, em função de informações obtidas em tempo de execução, como por exemplo, disponibilidade de tempo, carga do sistema e atributos de QoS.

- definir e/ou ajustar o tempo de execução das atividades do sistema, em função da experiência adquirida com a evolução deste;

- prover independência entre aplicação e ambiente operacional, através da implantação de controles (tipicamente realizados no nível de runtime ou sistema operacional) no meta-nível da aplicação;

- incrementar a portabilidade dos sistemas favorecendo sua utilização em ambientes abertos.

Entretanto, apesar do potencial da abordagem reflexiva, seu uso para tempo real pode ser questionado com relação aos aspectos de performance e previsibilidade. Com relação à performance, admite-se um overhead adicional devido ao processamento reflexivo. Com relação à previsibilidade, o problema não está na reflexão em si, mas sim no fato de que ela habilita a produção de sistemas tempo real muito mais flexíveis [MIT97], para os quais a análise de pior caso ainda é uma questão de pesquisa em aberto; contudo, embora a questão de previsibilidade possa dificultar (ou mesmo impedir) o uso de reflexão na programação de sistemas tempo real hard, sua utilização na programação de sistemas tempo real soft (onde as exigências de previsibilidade são menos severas) é perfeitamente viável e promissora.

3 Computação Imprecisa X Reflexão Computacional

Usando a Computação Imprecisa com recursos da Reflexão Computacional, podemos implementar formas de adaptação de aplicações Tempo Real a diferentes plataformas, utilizando seus recursos para fazermos uma implementação mais limpa e organizada, pois a computação imprecisa, como vimos, preocupa-se em "fazer o trabalho possível dentro do tempo disponível". Para isso, sacrifica a qualidade dos resultados para poder cumprir os prazos exigidos.

A reflexão computacional é toda a atividade de um sistema computacional realizada sobre si mesmo, e de forma separada das computações em curso, com o objetivo de resolver seus próprios problemas e obter informações sobre suas computações.

O emprego da computação imprecisa juntamente com reflexão computacional, do qual é alvo este trabalho, nos permite fazer com que a própria aplicação tenha condições de escolher o que executar de acordo com o tempo disponível. Usando a técnica da Computação Imprecisa, temos a oportunidade da própria aplicação (através da reflexão computacional) escolher se o resultado gerado será completo (qualidade total) ou parcial (qualidade mínima) de acordo com o tempo disponível.

4 Modelo RTR X Computação Imprecisa

O suporte a computação imprecisa no modelo RTR, como em qualquer outro modelo ou linguagem de programação, envolve dois aspectos básicos [FUR98][FUR98a]: a especificação da imprecisão através de construções que representem a estrutura inerente a cada uma das formas de programação imprecisa usuais e o controle relativo à execução dessas construções.

No modelo RTR, o uso de reflexão computacional possibilita a separação explícita entre estes dois aspectos, sendo que a questão da representação é tratada no nível base enquanto que o controle (semântica de execução e escalonamento) é tratado no nível meta, de forma independente tanto do programador dos objetos-base quanto do suporte de execução.

A representação da imprecisão dá-se através da definição de novos tipos de restrições temporais (representando as diferentes formas de programação de computação imprecisa) as quais são associadas à declaração dos métodos dos objetos-base que representam tarefas imprecisas. Por outro lado, o controle destas restrições, da mesma forma que os demais tipos de restrições temporais suportados pelo modelo, é implementado através de métodos dos meta-objetos gerenciadores, os quais interagindo

com o meta-objeto escalonador (MOE) proverão a semântica de execução correspondente a cada restrição considerada.

De acordo com a abordagem RTR, as restrições temporais representando as diferentes formas de programação de computação imprecisa devem ser implementadas através de métodos (um para cada restrição) nos meta-objetos gerenciadores (MOG). Assim sendo, sempre que o MOG interceptar a ativação de um método ao qual esteja associada uma dessas restrições, o fluxo de execução será desviado para o método que implementa a restrição em questão, o qual controlará a execução do método do objeto-base solicitado originalmente.

A implementação de computação imprecisa no modelo RTR é facilitada em decorrência de sua estrutura reflexiva. Nesta estrutura, as informações temporais relativas aos diferentes tipos de tarefas imprecisas (múltiplas versões, funções monotônicas e funções de melhoramento) são acessíveis no nível meta e podem subsidiar as decisões de escalonamento ao mesmo tempo em que são atualizados/modificados para refletir estas decisões. A combinação do modelo RTR com a técnica Computação Imprecisa resulta em um ambiente bastante flexível para a construção de aplicações tempo real bem como, flexíveis no que diz respeito ao seu comportamento em tempo de execução.

5 Especificação e Gerência da Adaptação

Qualquer mecanismo de adaptação empregado envolve o sacrifício de alguma propriedade funcional ou temporal da aplicação. Por exemplo, não executar uma tarefa significa abrir mão ou, pelo menos, diminuir a qualidade de alguma funcionalidade. O mesmo acontece com os outros mecanismos de adaptação descritos anteriormente. Alterações no comportamento da aplicação em função da adaptação serão percebidas pelo usuário. Logo, elas devem fazer parte da própria especificação da aplicação.

Quando uma aplicação é especificada em termos de níveis de qualidade, os níveis de qualidade descritos podem ser usados diretamente na etapa de projeto. Cada

componente de software pode ter comportamentos variados. O comportamento exibido é selecionado em tempo de execução, conforme o nível de qualidade requisitado.

No contexto deste trabalho, a adaptação é originada pela própria aplicação. Existe a necessidade da aplicação monitorar o seu próprio desempenho e solicitar aos seus componentes de software o nível de qualidade apropriado, conforme a situação no momento. Esta ação pode ser resumida nos seguintes itens:

- Coleta de informações a respeito do seu próprio desempenho temporal;
- Identificação da situação, a qual pode ser:

- Indesejada, o comportamento resultante não é satisfatório e exige uma redução no nível de qualidade;

- Equilibrada, os resultados temporais do nível de qualidade atual estão sendo aproximadamente respeitados, indicando que o nível de qualidade da aplicação deve ser mantido;

- Favorável, os requisitos temporais do nível de qualidade atual estão sendo respeitados com folga, indicando que o nível de qualidade da aplicação pode ser elevado;

- Seleção do novo nível de qualidade desejado;
- Comunicação aos componentes da aplicação do novo comportamento selecionado, caso este seja diferente do atual.

Uma forma simples de realizar a monitoração é comparar os deadlines das tarefas com o tempo de resposta efetivamente observado. Desta forma, é possível obter uma qualificação do desempenho da aplicação com respeito aos seus requisitos temporais. Existem duas quantificações básicas:

- somatório dos atrasos de todas as tarefas e;
- a taxa de tarefas que perderam o respectivo deadline.

O somatório dos atrasos das tarefas fornece uma medida mais apropriada do desempenho quando as tarefas devem ser executadas mesmo com atraso. Por outro lado, a taxa de tarefas que perderam o deadline é uma medida útil quando tarefas possuem deadline firme (firm deadline [BUR97]), isto é, não existe benefício em executar uma tarefa após seu deadline.

A vantagem da gerência feita pela própria aplicação é a possibilidade de aproveitar o conhecimento semântico que ela tem do seu estado. Este conhecimento semântico permite uma gerência superior ao que seria conseguido, por exemplo, por um sistema operacional que ignorasse completamente o significado das restrições temporais da aplicação.

6 Definição da Implementação

A implementação dos mecanismos de adaptação para aplicações Tempo Real na Internet devem ser definidos de acordo com as funcionalidades apresentadas pelo modelo RTR. Utilizaremos o mecanismo de variação do tempo de execução da tarefa como adaptador para a Internet.

A implementação deste mecanismo dá-se através da abordagem RTR, onde as restrições temporais representando as diferentes formas de programação de computação imprecisa devem ser implementadas através de métodos (um para cada restrição) nos meta-objetos gerenciadores. Assim sempre que o meta objeto gerenciador (MOG) interceptar a ativação de um método ao qual esteja associada uma dessas restrições, o fluxo de execução será desviado para o método que implementa a restrição em questão, o qual controlará a execução do método do objeto-base solicitado originalmente.

De acordo com a semântica de execução associada a cada uma das restrições temporais da Computação Imprecisa, vistas na seção 3 (função dos métodos que as implementam), elas podem variar em cada caso, dependendo da estratégia de escalonamento considerada [FUR98],[FUR98a].

6.1. Múltiplas versões

Forma de programação que permite maior grau de liberdade tanto com relação à representação quanto com relação à semântica associada ao controle da execução das tarefas imprecisas. A implementação através desta forma dá-se através da definição de um método cuja função consiste em escolher dinamicamente qual das versões deverá ser executada em resposta à ativação de um método declarado. Esta escolha envolve necessariamente uma interação com o MOE (Meta-Objeto Escalonador) e dependerá da estratégia de escalonamento em uso.

Adicionalmente este método poderá ser redefinido para considerar um número fixo de versões e/ou implementar diferentes semânticas de execução.

6.2. Função de Melhoramento

A semântica associada à restrição temporal que representa esta forma de programação de computação imprecisa é implementada através de um método do MOG (Meta-Objeto Gerenciador), cuja função básica será interagir com o MOE (Meta-Objeto Escalonador) para verificar a possibilidade ou não de execução da tarefa solicitada antes do esgotamento do deadline especificado. Outros controles podem ser implementados no contexto desta restrição.

6.3. Funções Monotônicas

Nesta forma de programação de computação imprecisa é impossível isolar-se completamente os aspectos de controle dos aspectos algoritmos da aplicação, sendo usada, portanto a capacidade reflexiva do modelo RTR para definição dinâmica do tempo durante o qual o método ativado deverá executar. Assim o método que implementa a restrição em questão determinará o tempo e comunicará ao método do objeto-base.

Para a determinação do tempo que poderá ser usado na execução do método do objeto-base, o MOG (através do método que implementa a restrição em questão) deverá

interagir com o MOE. A comunicação deste tempo para o objeto base poderá ser implícita (uma exceção temporal poderá ser sinalizada pelo MOG e detectada pelo objeto-base) ou explícita (transmitido na forma de um parâmetro funcional quando da efetiva ativação do método do objeto-base).

Além de controlar o comportamento do objeto-base o MOG deve também monitorar o desempenho temporal daquele. Isto é feito através da medição do tempo de resposta de cada método em comparação com o respectivo deadline. O meta-objeto relógio é usado como objeto auxiliar na manipulação das informações temporais. As métricas típicas a serem obtidas são o número de deadlines perdidos e o atraso médio dos métodos que não atenderam o deadline. Estes valores são comunicados ao MOE periodicamente. A iniciativa da comunicação pode tanto partir do MOG (informe periódico) quanto do MOE(consulta periódica).

6.4. Tratamento de tarefas no Meta-Objeto Escalonador

Como vemos em [FUR98], a função do MOE é determinar qual entre as tarefas liberadas será a próxima a ocupar o processador, com o emprego da computação imprecisa, ele recebe mais uma função que é a de escolher o nível de precisão a ser atingido em cada ativação de cada tarefa imprecisa.

Uma das soluções de escalonamento mais simples é baseada no tempo restante até o deadline de cada tarefa. Quando uma tarefa fica pronta para executar o escalonador recebe o seu deadline e uma estimativa para o tempo de execução (ETE) das suas opções de imprecisão. A forma como a tarefa imprecisa foi programada interfere neste aspecto:

- função monotônica – O ETE refere-se a uma execução completa (precisa da tarefa);
- função melhoramento – O ETE refere-se a execução da parte opcional;
- múltiplas versões – Cada versão possui um ETE associado.

O MOE inicialmente calcula o tempo restante até o deadline da tarefa, obtido através da operação "deadline – instante atual". Em seguida, o MOE seleciona a maior precisão possível tal que seu respectivo ETE seja menor ou igual ao tempo restante calculado antes. Após a seleção do nível de precisão, as tarefas são executadas segundo o EDF ("earliest deadline first").

Esta é uma solução muito simples e otimista, enfrenta problemas óbvios, pois ignora interferências e supõe que o tempo restante será usado pela tarefa em questão.

Entre os algoritmos com propriedades interessantes para o problema em questão é possível destacar o AVDT (Adaptive Threshold Policy), usando como política de admissão para partes opcionais. Cada tarefa é associada com um deadline, estimativas para o tempo de execução e um valor associado com cada nível de precisão.

Nesta política o MOE deve manter atualizada a densidade média de valor do sistema, com base em informações recebidas dos MOG. A densidade média é obtida dividindo-se o valor total do sistema pelo tempo do processador utilizado. O valor total do sistema corresponde ao somatório dos valores das tarefas já concluídas. No AVDT a densidade média de valor do sistema é utilizada como limite mínimo (threshold) para a densidade de valor das partes opcionais. Somente partes opcionais que possuem uma densidade de valor maior ou igual à densidade média do sistema serão liberadas pelo MOE para execução. A densidade de valor de uma parte opcional é obtida dividindo-se o valor da tarefa pela expectativa do seu tempo de execução. As vantagens do AVDT incluem um baixo custo computacional (overhead) e a capacidade de automaticamente adaptar-se a variações na carga.

6.5. Seleção do Nível de Qualidade

A função do MOE é definir o "nível de qualidade escolhido" para a aplicação. Esta informação será comunicada aos vários MOGs e será usada para definir o nível médio de precisão de cada método. A escolha do nível de qualidade é feita a partir do comportamento atual da aplicação, monitorado pelos MOGs e comunicado periodicamente ao MOE. O nível de resposta atual da aplicação é comparado com

valores estabelecidos em projeto e poderá disparar uma mudança global de comportamento. A mudança entre níveis de qualidade é determinada por gatilhos definidos também em projeto.

Quando do projeto, existe uma série de decisões a serem tomadas, tais como: quantidade de níveis de qualidade; tipo de métrica a ser usada na construção de gatilhos, valores da métrica associados com cada gatilhos em particular.

6. Protótipo

Por ser um modelo abstrato, independente de linguagem de programação, a utilização do Modelo RTR depende de uma implementação concreta que estabeleça a disciplina de programação necessária para obtenção do comportamento especificado. Para realizarmos este trabalho usamos a linguagem de programação JAVA e sua extensão Javassist [CHI00] (devido aos inúmeros recursos que dispões e principalmente a independência de plataforma) para implementar o protótipo e demonstrar as funcionalidades do modelo, com os mecanismos de adaptação enumerados neste trabalho.

1 Justificativa:

O protótipo foi implementado usando Java por ser uma linguagem que permite a portabilidade entre plataformas, o que é extremamente importante para que seja usado na Internet, ou em qualquer outra máquina com sistemas operacionais e/ou arquiteturas diferentes. A escolha da extensão Javassist (Reflexão Estrutural e Comportamental), se deu porque o modelo RTR exige que a linguagem utilizada tenha estes dois aspectos para que seja implementado com todas as funcionalidades propostas originalmente.

Durante o período de confecção deste trabalho várias versões da extensão Javassist foram utilizadas, sendo que a última foi a versão 0.8.

2 Implementação:

Foi utilizado para implementação deste protótipo um jogo "TELEJOGO" [OLI00] (aplicativo Java), implementado de várias formas para testar as mais diversas possibilidades de implementação conforme abaixo descrito:

O jogo "TeleJogo" implementa vários jogos

Jogo Fechado:

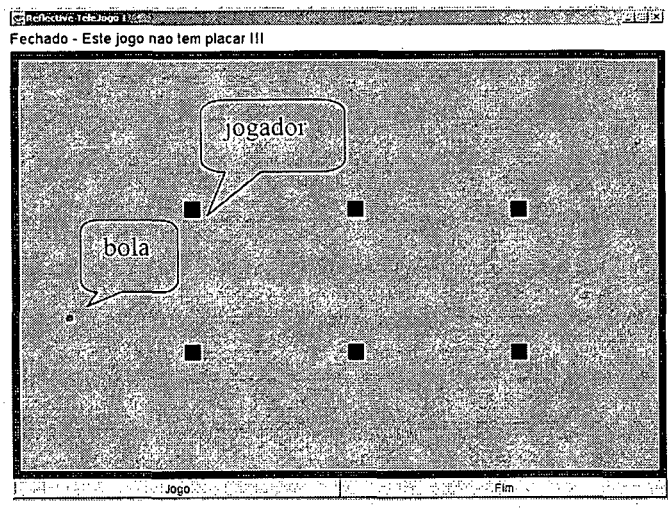


Figura 9 - Jogo

Os jogadores (os seis quadrados maiores) são colocados na janela do jogo e movimentam-se dentro de um espaço determinado, fazendo um vai-e-vem. Quando a bola (quadrado menor à esquerda da tela) toca qualquer um deles ou as bordas é automaticamente desviado de sua trajetória como se fosse rebatida.

Este jogo ainda tem outras variações (futebol, tênis...) que não convém nos atermos, pois o objetivo do trabalho não é analisar o jogo em si, mas sim, a aplicação da Reflexão Computacional através do modelo RTR e da Computação Imprecisa em sua execução.

Implementação

A implementação foi realizada criando uma Meta Classe Gerenciadora para cada classe, sendo ela responsável pela interceptação da chamada do Objeto-Base. A meta-classe possui métodos que podem alterar valores dos argumentos passados como parâmetro, bem como interagir com o MOE (Meta Objeto Escalonador) e o MOR (Meta Objeto Relógio).

```
    // exemplo de Meta Classe Gerenciadora

    public class MetaJogoFechado extends Metaobject {
        public MetaJogoFechado(Object self, Object[] args) { ...  }

        public Object trapFieldRead(String name) { ...   }

        public void trapFieldWrite(String name, Object value) {... }

        public Object trapMethodcall(int identifier, Object[] args) {
        /* método onde é tratado a Imprecisão e a interação com o
MOE e o MOR */
        }
    }
```

3 Estrutura e Funcionamento

3.1. Objeto Base

Os Objetos-Base foram adaptados para incluir métodos com comportamento diferente dos originais da classe, para que o meta-objeto pudesse, através de suas computações e usando os recurso da extensão Javassist, escolher qual o método indicado para a situação atual.

3.2. Meta-Objeto Gerenciador.

O Meta-Objeto Gerenciador ficou responsável pela interceptação do acesso à classe original, e a interagir com o MOE e com o MOR para que através de suas computações e conforme o tipo de restrições temporais adotadas fosse alterado o método acessado tanto para piorar a qualidade de apresentação ou até mesmo, na pior das hipóteses não efetuar o processamento solicitado.

3.3. Meta Objeto-Escalonador

O Meta-Objeto Escalonador é o responsável pelo escalonamento das tarefas solicitadas pelo MOG. Têm no corpo de sua classe vários métodos implementando formas de escalonamento diferentes, que são ativadas conforme o comportamento e desempenho dos processamentos solicitados.

O MOE tem uma fila de escalonamento que é organizada dependendo do método de escalonamento adotado, e libera o processamento do MOG quando for a vez dele na fila, de forma que quando estiver passado o tempo hábil de execução (perda do deadline) retorna um erro ao MOG para que este tome as devidas providências.

3.4. Meta Objeto-Relógio

Responsável por ativar processamentos com tempo de ativação programada, bem como controla a passagem do tempo. Recebe os processamentos de ativação programada e guarda-os em uma fila de ativações futuras, ordenadas por prioridade e sempre observando o deadline para que este não seja perdido.

4 Resultados

Podemos verificar na tabela abaixo o resultado da avaliação do uso do Modelo RTR juntamente com a Computação Imprecisa na execução de um método da aplicação "Jogo".

	Diferenças na execução do Jogo sem o uso de Reflexão Computacional (Modelo RTR) (tempo em milissegundos)		Diferenças na execução do Jogo usando Reflexão Computacional (Modelo RTR) e Computação Imprecisa. (tempo em milissegundos)	
Descrição	Máquina Vazia	Máquina Cheia	Máquina Vazia	Máquina Cheia
1 - Criação do Objeto	480	721	430	1001
Tempo entre a execução de 1 e 2	170	4276	481	3976
2 – execução do método tick	10	4086	0	0
Tempo entre a execução de 2 e 3	181	10	220	0
3 – execução do método tick	0	0	10	0
Tempo entre a execução de 3 e 4	130	10	221	0
4 – execução do método tick	30	3855	0	0
Tempo entre a execução de 4 e 5	170	0	0	0
5 - execução do método paint	110	3885	30	0
Tempo entre a execução de 5 e 6	-	-	-	3985
Métodos 6,7,8 descartados(deadline)	-	-	-	-
9 – execução do método tick	-	-	-	10
Tempo entre a execução de 9 e 10	-	-	-	20
10 – execução do método tick	-	-	-	10
Tempo entre a execução de 10 e 11	-	-	-	70
11 – execução do método tick	-	-	-	1000
Tempo entre a execução de 11 e 12	-	-	-	10
12 - execução do método paint	-	-	-	20
Tempo total da execução	1281	13108	1352	10102

Tabela 2 – Tempos comparativos entre a execução normal e com o proposta.

Dada a tabela, consideramos os seguintes dados para sua análise: Tempo máximo para a execução do método: 1000 ms. " – " significa que o método não foi executado simplesmente porque não fazia parte da análise, ou porque perdeu o deadline (caso dos itens 6, 7 e 8). A "Máquina Vazia" representa o computador sem nenhum outro software sendo executado, enquanto "Máquina Cheia" representa o computador executando um aplicativo que consome praticamente todos os recursos de memória da máquina. O método tick é o responsável pelo gerenciamento da Bola (calcula posição, velocidade e consistência com os objetos animados ou paredes para verificar se não houve choque).

Quando este método perdia o deadline foi executado um método sem nenhuma instrução. Os métodos considerados para o teste dos tempos e análise dos resultados foram o método paint mostra a bola na tela na posição calculada.

A tabela acima representa que a execução normal do jogo com a máquina vazia mostra a bola pela primeira vez depois de 1281 ms, enquanto que rodando com a máquina cheia a bola é mostrada depois de 13108 ms. A bola aparece no mesmo ponto que a execução na máquina vazia depois de 11,82 segundos.

Com a utilização da solução proposta nesta dissertação a execução do jogo com a máquina vazia mostra a bola pela primeira vez 1352 ms após seu início, enquanto que com a máquina cheia a bola vai aparecer pela primeira vez depois de 911 ms, com o detalhe de não aparecer na mesma posição que o primeiro, pois os métodos de cálculo perderam o deadline e foram descartados. A bola vai aparecer na mesma posição que na anterior depois de 10102 ms.

Podemos verificar então que a execução normal na máquina vazia foi 5,54% mais rápida que a utilizando a solução proposta, o que podemos atribuir aos controles extras utilizados. Já com a máquina cheia observamos que a execução normal leva 29,76% mais tempo que a execução com o modelo proposto, ainda sem considerar que a bola vai aparecer na tela pela primeira vez antes que a execução do jogo original com a máquina vazia.

Colocando tempos de deadline diferentes para os métodos de cálculo de posição e método de desenho na tela, podemos, através da computação imprecisa, alterar a execução do jogo de forma que o cálculo sempre seja efetuado utilizando métodos diferentes na classe (uma com a execução completa e pelo menos outro com o mínimo a ser executado – como o cálculo da nova posição) o que possibilitará mostrar a bola sempre dentro do tempo estimado, mesmo que sua visualização não seja contínua (possibilidade de vê-la passando da posição n para a posição n+x, onde x pode ser qualquer posição dentro da rota normal da bola e não necessariamente na posição seguinte (n+1)). Isso garante que a bola vai estar na posição final, no pior caso, na soma total dos tempos para o deadline, enquanto na execução normal não pode ser garantido

nada, pois vai depender da situação do processador, da memória do computador em que estiver sendo executado. No exemplo citado o método alternativo não calculava a nova posição da bola porque poderia afetar no perfeito funcionamento do jogo, já que ficaria inconsistente se a bola ultrapassasse qualquer objeto jogador ou parede.

5 Análise

A implementação consistiu no desenvolvimento da proposta apresentada neste trabalho, com o uso de Computação Imprecisa e Reflexão Computacional em Tempo Real.

Foi utilizado o Modelo RTR [FUR97] para implementar a solução proposta, de forma que a estrutura geral do modelo incorporasse a Computação imprecisa. Isso se deu na colocação de classes opcionais nos objetos-base para que o MOG quando da análise dos resultados do MOR e MOE pudesse trocar dinamicamente a classe pela sua substituta, inclusive trocando o nome da classe para a próxima carga na JVM.

Foram realizados vários testes com o aplicativo, comparando os resultados com sua forma original. Obtivemos melhores resultados quando houve retardo no processamento de algumas computações que normalmente atrasariam a execução destas tarefas, mas com o emprego do modelo proposto, as computações que começaram a perder seus deadlines foram saindo da fila de execução e liberando a máquina para os que ainda poderiam ser executados. O próprio gerenciador (MOG) passou a controlar qual das versões da classe iria executar alterando não só o seu comportamento naquele momento, mas também os bytecodes (class) daquela classe quando outro processamento fosse efetuado.

Quando o aplicativo perdia muitos deadlines, ou seja, havia uma grande quantidade de erros, o MOG automaticamente reduzia a qualidade de apresentação do aplicativo e também passava a carregar classes com o mínimo de processamento

necessário para manter o aplicativo funcionando de forma satisfatória, bem como uma qualidade de visualização pelo menos razoável.

Os Objetos-Base passaram a incorporar várias formas de processamento para cada classe, enquanto os seus Gerenciadores (MOGs) verificavam quais dessas classes iriam ser executadas, e quais eram os resultados visualizados, dependendo do escalonamento e liberação do MOE e do MOR.

O algoritmo de escalonamento também poderia se alterado dependendo da quantidade de erros e da performance desejada para que se adapte melhor. Essa alteração não foi implementada neste protótipo, mas é perfeitamente possível utilizando a forma de programação e a extensão adotada pelo protótipo.

Depois de realizadas várias experiências de comunicação e testes de chamada dos objetos, Meta-Objetos e MOE e MOR, chegamos a alguns pontos positivos desta implementação quanto a Reflexão Computacional, Computação Imprecisa, modelo RTR e Tempo Real:

- Permite que construções temporais específicas de uma aplicação sejam modificadas, adicionadas ou substituídas (Meta-Objeto Gerenciador intercepta a requisição ao objeto base e propõe as mudanças, se necessário). Pode-se estabelecer um comportamento alternativo para as exceções temporais;

- O algoritmo de escalonamento pode ser alterado a qualquer tempo, conforme auto-análise do comportamento temporal das filas de requisições, adequando a aplicação ao ambiente.

- Como existe a possibilidade de auto-análise do próprio programa, pode-se estabelecer regras no código para que com as informações obtidas desta análise, mude-se o comportamento do programa.(MOG e/ou MOE analisam o comportamento do programa e altera as chamadas a determinada requisição).

- Já que Javassist possibilita alterar o bytecode da classe, podemos estabelecer um algoritmo evolutivo com alternativas para as mais diversas situações que venham a

ocorrer em tempo de execução (O MOE faz uma análise evolutiva do programa adaptando-o a situações que já ocorreram e que assim já estão contempladas para novas chamadas);

- Com o autogerenciamento é possível adquirir, no nível Meta, uma independência entre a aplicação e o sistema operacional, já que pode ser previsto no MOG até qual método deve rodar em que tipo de sistema operacional;

- Com a utilização de Java como linguagem para a implementação do modelo e da extensão Javassist, existe um favorecimento a utilização em ambientes abertos, pela portabilidade apresentada.

- Com a Computação Imprecisa, surge a oportunidade da própria aplicação, através da Reflexão Computacional, escolher que tipo de resultado será fornecido, ou seja, que qualidade seja apresentada. (O MOG, analisando o resultado das perdas de deadline e atraso nas tarefas, pode escolher que tipo de implementação usar – qualidade total – ou – qualidade parcial (aceitável) de acordo com o tempo disponível);

- O MOG é o responsável pelo controle das restrições temporais representando as diversas formas de programação (múltiplas versões, função monotônica e função de melhoramento);

- Como a aplicação coleta informações a respeito de seu próprio desempenho temporal, pode identificar a sua situação e selecionar automaticamente um novo nível de qualidade desejada comunicando-se entre os componentes e comunicando aos componentes o novo comportamento. A vantagem da gerência feita pela própria aplicação e a possibilidade de aproveitar o conhecimento semântico que ela tem do seu estado, gerando uma qualidade superior.

- A implementação do mecanismo de adaptação para as aplicações de tempo real podem variar conforme a estratégia adotada. Para utilização na Internet o melhor mecanismo é o da variação do tempo de execução da tarefa. O MOE determina qual entre as tarefas da fila será a próxima a ser liberada para o processamento, escolhendo o

nível de precisão a ser atingido em cada ativação, por conta da implementação da computação imprecisa.

- O Modelo RTR dá o suporte necessário para que a Computação Reflexiva combinada com a Computação Imprecisa gere um ambiente bastante flexível e estável para a construção de aplicações tempo real, gerando também garantias de um comportamento adequado em tempo de execução.

Embora vários sejam os aspectos positivos desta metodologia, encontramos alguns aspectos que podem ser questionados:

- Performance: ocorre um overhead adicional devido ao processamento reflexivo e alterações de métodos resultantes da Computação Imprecisa. (O MOE, MOR e MOG fazem processamentos gerenciais da classe base para só depois ativar o processamento em si, mas esse tempo em máquinas atuais não representa grande expressão);

- Previsibilidade: dificulta o cálculo do pior caso, visto que existe uma flexibilidade muito grande em função da reflexão e computação imprecisa. (O MOG pode alterar métodos enquanto o MOE altera algoritmo de escalonamento, fazendo com que a aplicação possa sofrer várias combinações de códigos em tempo de execução);

- A extensão Javassist gera algumas restrições também, tal que a alteração de bytecodes não pode ser assumida depois que a classe já está carregada na JVM, ficando disponível apenas para aplicações que ainda não estejam carregadas;

- O Javassist ainda não implementa substituição do corpo do método;

Para a solução destes dois últimos, adotou-se a opção de incluir na própria classe do objeto base métodos alternativos para cada tipo de restrição que possa ser encontrada. Assim o próprio MOG pode, através de métodos de desvios disponíveis no Javassist (reflexão estrutural e comportamental), alterar o comportamento da classe sem a necessidade de alterar o bytecode.

Quanto aos dois primeiros questionamentos, não daremos maior importância neste trabalho, pois as máquinas hoje em dia possuem processadores cada vez mais rápidos, diminuindo a preocupação com performance cada vez menos, e a complexidade das aplicações tempo real aqui estudadas não são críticas suficiente a ponto de não poderem ter alguns deadlines perdidos.

7. Conclusão

Este trabalho mostrou como a técnica de Computação Imprecisa, implementada através de Reflexão Computacional, foi utilizada para permitir a adaptação de aplicações de Tempo Real a diferentes plataformas, inclusive no contexto da Internet. Inicialmente foram considerados diversos aspectos relacionados com a adaptação de aplicações de Tempo Real. Em seguida técnicas de Computação Imprecisa e Reflexão Computacional foram discutidos. Após consideramos como o modelo de programação RTR foi projetado e de que maneira foi utilizado para ilustrar a forma como esta adaptação foi implementada.

Na maioria dos projetos de software os aspectos temporais da aplicação são ignorados nas fases iniciais do desenvolvimento e considerados apenas depois que o código já está escrito e os aspectos funcionais depurados. O resultado desta atitude é a necessidade de empregar remendos de última hora com o objetivo de tornar a aplicação aceitável para os usuários, no que diz respeito aos seus aspectos temporais. Implícita na abordagem apresentada por este trabalho está a idéia de que os aspectos temporais devem ser considerados desde as fases iniciais de especificação e projeto. Desta forma, os requisitos temporais da aplicação serão mais bem entendidos e serão atendidos com menores custos de desenvolvimento.

1 Considerações sobre o trabalho

Apesar do protótipo não estar preocupado com a questão temporal para fins de aplicabilidade, mas sim para validação do modelo RTR conjuntamente com Computação imprecisa, a questão temporal foi levada em conta para verificar a performance das aplicações e sua adaptabilidade no uso de redes, principalmente a Internet.

Hoje existem muitos problemas quanto à utilização da Internet para se fazer aplicações tempo real, já que a Internet não é nem rápida o suficiente e nem estável para que possa garantir qualquer tipo de aplicabilidade tempo real. Mas com as tecnologias que estão surgindo existe a possibilidade real do modelo proposto ser aplicado com sucesso, inclusive utilizando a linguagem Java que por ser portável, também é muito mais lenta que outras linguagens de baixo nível. Existem várias pesquisas para que o problema da velocidade de execução desta linguagem também seja resolvido, bem como a própria evolução da extensão JAVASSIST [CHI00] que vem ocorrendo regularmente.

Após a conclusão dos estudos aqui propostos, mostramos a validade do uso de reflexão computacional juntamente com a técnica de computação Imprecisa como um mecanismo de adaptação para aplicações tempo real e que o modelo RTR através de suas especificações é capaz de suportar esta implementação.

Dentro do que se propõe o modelo RTR e a Computação Imprecisa com a utilização da reflexão Computacional (estrutural e comportamental) demonstrou grande flexibilidade para o desenvolvimento de várias aplicações de tempo real não-crítico (além de jogos, aplicações multimídia, tutoriais para ensino à distância, comércio eletrônico, simulações de concorrência, cálculos matemáticos e até telecomunicações – videoconferência).

2 Trabalhos Futuros

Para dar continuidade ao trabalho aqui apresentado, está previsto um estudo sobre a aplicabilidade do Modelo RTR, Computação Imprecisa para projetar sistemas cooperativos tolerantes á faltas.

Estudo de aplicações que envolvem consultas à banco de dados e que devam ter um tempo de resposta aceitável garantido. Aplicativos de Compra e Venda com atualização em Tempo Real do estoque (ex: Passagens aéreas evitando o overbook).

Em curto prazo, a implementação de um simulador que entre outras coisas, controla filas, congestionamentos e propõe soluções para acesso à Internet, com garantia de tempo de resposta e estimativa de qualidade de linha (acesso à Internet) para que seja garantido o tempo de resposta.

No que diz respeito ao aproveitamento deste trabalho, a perspectiva é a seguinte:

- Utilização da abordagem proposta para construir mecanismos de adaptação multimídia, com controle de qualidade;

- Implementação do modelo com vários algoritmos de escalonamento;

- Interação com outros trabalhos desenvolvidos pelo CPGCC;

- Proposição de formas de controle de tutoriais para ensino a distância com a garantia de tempo não só para a apresentação quanto para a avaliação dos alunos;

8. Bibliografia

[BUR97] A.Burns, A.Wellings. Real-Time Systems and Programming Languages. Addison-Wesley, 2nd edition, 1997.

[CHI93] CHIBA, S. OpenC++ programmer's guide version 1.2. Tokyo: Dept. of Information Science, University of Tokyo, 1993. 21p. (Technical Report n. 93-3).

[CHI96] CHIBA, S. OpenC++ programmer's guide version 2.0. Palo Alto: PARC Xerox, 1996.

[FUR97] Furtado, Olinto. RTR – Uma Abordagem Reflexiva para Programação de Aplicações Tempo Real. Tese de Doutorado, UFSC. 1997.

[FUR98] Furtado, Olinto; Oliveria, R. ; Farines J-M. ; Fraga, J. "Implementação de Tarefas Imprecisas no Modelo Reflexivo Tempo Real RTR", XXIII CLEI, 1998.

[FUR98a] Furtado, Olinto; Oliveria, R. ; Farines J-M. ; Fraga, J. "Suporte para Computação Imprecisa no Modelo ReflexivoTempo Real RTR", I WTR, 1998.

[JAV96] _____. Java Core Reflection: API and Specification. Java Soft, Mountain View, CA, USA, october 1996.

[LIS97] Lisbôa, Maria L. B. Reflexão Computacional no Modelo de Objetos, agosto 1997.

[MAE87] MAES, P. Concepts and experiments in computational reflection. SIGPLAN Notices, New York, v.22, n.12, p. 147-169. Trabalho apresentado no OOPSLA, 1987, Orlando, Flórida.

[MIT97] Mitchel, S.E., Burns, A. and Wellings,A.J. "Developing a Real-Time Metaobject Protocol", WORDS'97, Newport Beach, California, USA, february 5-7, 1997.

[MON98] Montez, Carlos. Fraga, Joni. Farines, Jean-Marie. Oliveira, Rômulo. Extensões dos Padrões CORBA para Aplicações de Tempo Real. 1998

[MON99] Montez, Carlos. Fraga, Joni. Farines, Jean-Marie. Oliveira, Rômulo. Escalonamento Adaptativo Usando Real-Time CORBA. SBRC'99, Salvador, BA, Maio de 1999

[OLI96] Oliveira, Rômulo Silva. Fraga, Joni. Uma Solução Mista para o Escalonamento Baseado em Prioridades de Aplicações Tempo Real Críticas. Semish96.

[OLI97] Oliveira, Rômulo Silva. Fraga, Joni. Escalonamento de Tarefas Imprecisas em Ambiente Tempo Real Distribuído. SCTF97.

[OLI97a] Oliveira, Rômulo. Mecanismos de Adaptação para Aplicações Tempo Real na Internet. 1997

[OLI97b] Oliveira, R. "Computação Imprecisa". Revista de Informática Teórica e Aplicada, vol.4, n.1, pp. 87-106, 1997.

[OLI98] Oliveira, Rômulo Silva. Fraga, Joni. Escalonamento de Tarefas com Relações Arbitrárias de Precedência em Sistemas Tempo Real Distribuídos. SBRC 1998.

[OLI99] Oliveira, R., Furtado,O. "Um mecanismo de Adaptação para aplicações Tempo Real Baseado em Computação Imprecisa e Reflexão Computacional", ... 1999.

[OMG95] OMG, The Common Object Request Broker : Architecture and

Specification – Revision 2.0, Object Management Group (OMG), Jul.1995.

[OMG98] OMG , Realtime CORBA – Joint Revised Submission, Document orbos/98-10-05, Oct.98.

[STA95] J.A.Stankovic, K. Ramamritham. A Reflective Architecture for Real-Time Operating Systems. Chapter 2 in "Advances in Real-Time Systems", edited by S.H. Son, Prentice-Hall, 1995.

[STA96] Stankovic, J.A., et al., "Strategic Directions in Real-Time and Embedded Systems", ACM Computing Surveys, vol 28, n.4, pp 751-763, december, 1996.

[WU97] WU, Z., SCHWIDERSKI, S. Reflective Java : Making Java even more flexible. FTP: Architeture Projects Management Limited (apm@ansa.co.uk), Cambridge,UK,1997.

[CHI00] Chiba, S. "Load-time structural reflection in Java" : ECOOP2000 – Oriented Programming LNCS 1850, Springer Verlag, pp 313- 336, 2000.