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

Taciano Tres

Um tradutor Java/RTR integrado ao ambiente Eclipse

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

Prof. Olinto José Varela Furtado, Dr.
Orientador

Florianópolis, Dezembro de 2004

Um tradutor Java/RTR integrado ao ambiente Eclipse

Taciano Tres

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

Prof. Raul Sidnei Wazlawick, Dr.

Coordenador do Curso

Banca Examinadora

Prof. Olinto José Varela Furtado, Dr.

Orientador

Prof. Rômulo Silva de Oliveira, Dr.

Prof. Luis Fernando Friedrich, Dr.

Prof. Frank Augusto Siqueira, Dr.

“Tudo tem uma razão de ser e nada acontece sem a permissão de Deus.”
(Allan Kardec, O Livro dos Espíritos, p.272, questão 536).

Para minha noiva, Mariana, que sempre me apoiou e me deu forças para concluir este trabalho.

Agradecimentos

Em primeiro lugar à Deus, pela oportunidade de estar aqui evoluindo.

Ao Professor Dr. Olinto José Varela Furtado, pela sua preciosa orientação, incentivos e participação durante o desenvolvimento deste trabalho.

Aos membros da banca, por terem aceitado julgar este trabalho e pelos expressivos e frutíferos comentários feitos.

Aos meus familiares, em especial meus pais, que sempre me apoiaram, pelo incentivo e pelo companheirismo, em todos os momentos.

À Universidade Federal de Santa Catarina, seus funcionários e professores, por todo esforço despendido em manter a educação, pesquisa e desenvolvimento do nosso país.

Aos colegas e amigos, que de certa forma contribuíram para a realização deste trabalho.

À minha noiva Mariana, que nunca deixou de me apoiar, me defender, me amar e, principalmente, compreender, pelo tempo que precisei dedicar ao trabalho e não pude estar com ela.

Sumário

Lista de Figuras

Lista de Segmentos de Códigos

Lista de Siglas

API	Application Program Interface
AST	Abstract Syntax Tree
BNF	Backus Naur Form
CMC	Clock Meta-Class
IDE	Integrated Development Environment
JCP	Java Community Process
JDT	JAVA™ Development Tooling
JIT	Just-in-Time
JSR	Java Specification Request
JVM	JAVA™ Virtual Machine
MMC	Manager Meta-Class
MOE	Meta-Objeto Escalonador
MOG	Meta-Objeto Gerenciador
MOR	Meta-Objeto Relógio
OBTR	Objeto Base de Tempo Real
PDE	Plug-in Development Environment
RI	Reference Implementation
RTBC	Real-Time Base Class
RTR	Reflexivo Tempo Real
RTSJ	Real-Time Specification for JAVA™
SMC	Scheduler Meta-Class
SMRH	Static Meta Reference Holder
STR	Sistemas de Tempo Real
TCK	Test Compatibility Kit
UML	Unified Modeling Language

Resumo

Sistemas Tempo Real requerem, além da correção lógica, que os resultados estejam disponíveis dentro do prazo, além de, geralmente, apresentarem restrições de memória, de processador e de consumo de energia. O desenvolvimento destes sistemas, pelos motivos apresentados, é mais complexo se comparado com sistemas computacionais comuns.

Este trabalho apresenta uma alternativa para o desenvolvimento de Sistemas Tempo Real sem restrições críticas, utilizando o Modelo Reflexivo Tempo Real implementado sobre a linguagem de programação JAVA™. A união destas tecnologias gerou a linguagem Java/RTR, que é formalmente definida neste trabalho, e para qual foi criado um tradutor, que gera código JAVA™ utilizando as funcionalidades da Especificação de JAVA™ para Tempo Real. Esta especificação introduz conceitos de tempo real em JAVA™ através de uma API de programação e uma nova Máquina Virtual JAVA™.

Objetivando facilitar a utilização de Java/RTR, também é apresentado o desenvolvimento de um *plug-in* para a Plataforma Eclipse, permitindo uma maior integração entre o desenvolvedor e a tecnologia criada. O *plug-in* reduz o tempo de desenvolvimento, permitindo ciclos de codificação–compilação–testes mais curtos, aumentando a produtividade dos programadores.

Palavras chaves: JAVA™, tempo real, Modelo RTR, Java/RTR, Eclipse

Abstract

After logical correction, Real-Time Systems require the results available within the specified time limit, usually despite memory, processor or energy consumption constraints. Because of this, the development of Real-Time Systems is more complex if compared to normal computing systems development.

This dissertation presents an alternative to soft Real-Time Systems development, using the Real-Time Reflexive Model implemented over the JAVA™ programming language. The result of this union is the Java/RTR programming language, that has its formal description detailed here, and for which a translator is developed. This translator processes the Java/RTR code and generates JAVA™ code with the Real-Time Specification for JAVA™ functionalities. This specification adds real-time concepts to JAVA™ through a programming API and a new, extended JAVA™Virtual Machine.

Aiming to make easy using Java/RTR, a *plug-in* for Eclipse Platform is included, allowing a better interaction between the developer and the new technology. The *plug-in* decreases the development time, making the coding–compilation–test cycle shorter, increasing the programmer performance.

Keywords: JAVA™, real-time, RTR Model, Java/RTR, Eclipse

Capítulo 1

Introdução

1.1 Desafios

A grande maioria dos processadores fabricados no mundo é utilizada em dispositivos embutidos (relógios, celulares, handhelds, automóveis, etc). As redes de comunicação sem fio são a mais nova área na utilização destes dispositivos, que já incluia aplicações de tele-medicina e de sistemas militares. A utilização de sistemas embutidos tempo real cresce cada vez mais, mas aplicações tempo real também são possíveis em outros tipos de equipamentos, como computadores de propósito geral e caixas automáticos de bancos (ATM, *Automatic Teller Machine*).

A grande dificuldade está em desenvolver aplicações para esta massa de dispositivos; aplicações estas que exigem certa previsibilidade, robustez e possuem restrições de memória, processamento e de consumo de energia, na sua grande maioria. Levando em consideração as dificuldades gerais de desenvolver software dentro do prazo e do orçamento estimados, o desenvolvimento de software tempo real é mais complexo que o desenvolvimento de aplicativos sem restrições temporais.

Também é necessário levar em conta a exigência do mercado por novos lançamentos em períodos cada vez mais curtos. Para que os desenvolvedores tenham seu trabalho atenuado, é essencial a utilização de ferramentas, linguagens e métodos que aumentem sua produtividade.

Entretanto, linguagens, modelos e ferramentas para desenvolvimento de sistemas tempo real ainda são pouco disseminados, fazendo com que alguns desenvolve-

dores de STR utilizem seus equivalentes para sistemas gerais, aumentando a dificuldade de se alcançar o objetivo. É necessário considerar também que há poucas ferramentas de desenvolvimento disponíveis para STR, custando, na sua maioria, valores na casa dos milhares de dólares (TIM 01b).

1.2 O estado da arte

Muitas aplicações tempo real ainda são desenvolvidas utilizando a linguagem de programação C, e está aumentando a utilização da linguagem C++. Mesmo sendo linguagens de mais alto nível que *assembly*, não são as mais produtivas, nem as mais seguras, além de não possuirem contruções tempo real nativas.

O uso de C++ para o desenvolvimento de STR em dispositivos embutidos é incentivado através de projetos como SystemC (SYN 04). Como definido no manual do usuário, "SystemC é uma biblioteca de classes em C++ e uma metodologia que pode ser utilizada para efetivamente criar algoritmos, arquiteturas de hardware, interfaces para *System On a Chip* (SoC) e projetos de sistemas." SystemC está se tornando o padrão *de facto* para se desenvolver STR para dispositivos embutidos, mas carrega consigo todos os pontos positivos e negativos de C++.

A utilização de linguagens de mais alto nível para o desenvolvimento de STR passa, quase que obrigatoriamente, por um *cross-compiler*[1]. Este tipo de compilador auxilia nesta tarefa, principalmente quando se trata de um novo processador ou nova arquitetura. Mas o mesmo fica amarrado na arquitetura alvo, dificultando a reutilização do código, e também sujeito às novas versões da mesma (mudanças no endereçamento de memória, nos registradores e na sua quantidade, nas instruções, etc).

Devido a tudo isto, a linguagem de programação JAVA™ (GOS 97; SM 04b) tem se tornado uma escolha atrativa, pelos seguintes motivos:

+ exitem aproximadamente quatro milhões de desenvolvedores que utilizam a linguagem JAVA™em todo mundo, segundo informações da Sun Microsystems divulga-

[1]Um compilador que é executado em uma arquitetura mas gera código para outra arquitetura. Por exemplo, quando se utiliza um computador do tipo padrão PC para o desenvolvimento de aplicações para dispositivos que utilizam processadores RISC ou ARM.

das na apresentação do *JavaOne* 2004;

+ é mais simples que C++, pois o sistema de coleta automática de lixo evita que o programador se preocupe com alocação de memória, e o sistema de segurança de tipos impede que ponteiros sejam manuseados diretamente, evitando problemas com ponteiros apontando para áreas proibidas;

+ pode ser aprendida facilmente, pois há diversos tutoriais disponíveis na Internet, incluindo o material oficial da Sun (SM 04b), assim como em diversos sites relativos a tecnologia JAVA™ (JuG.com.br, JavaFree.com.br e Java.net, por exemplo), além de vários livros sobre a linguagem, abrangendo de aspectos básicos a avançados, muitos na língua portuguesa;

+ sua arquitetura emprega uma máquina virtual — Java Virtual Machine (JVM) — que permite as aplicações serem executadas em qualquer plataforma que possua uma implementação da JVM;

+ são quase 650 milhões de computadores com uma máquina virtual JAVA™ instalada (também segundo informações da Sun Microsystems divulgadas na apresentação do *JavaOne* 2004);

+ possui uma biblioteca poderosa e portável, que reduz o tempo e o custo do desenvolvimento;

+ possui gerenciamento de memória automático, é fortemente tipada, é capaz de carregar classes dinamicamente, possui um mecanismo de reflexão e introspecção, suporta concorrência e sincronização, e o *byte code* gerado é menor que código nativo (COR 02);

+ todas as empresas da lista *Fortune 500* utilizam esta tecnologia (segundo informações da Sun Microsystems divulgadas na abertura do *JavaOne* 2003).

+ Existem várias propostas de extensão e/ou adaptação para uso de JAVA™ em aplicações tempo real, como “Real-Time JAVA™”, citado em (FUR 97).

Entretanto, utilizar implementações convencionais de JAVA™ para desenvolver aplicações tempo real embutidas não é recomendado devido aos seguintes problemas:

- o escalonamento de *threads* JAVA™ não é especificado formalmente (propositadamente para facilitar a implementação de novas máquinas virtuais);

- o coletor de lixo tem preferência na execução, o que pode atrapalhar a execução de *threads* com restrições temporais;

- o sistema de alocação de memória, por ser automático, não permite a escolha do tipo de memória que deve ser utilizado;

- historicamente, programas escritos em JAVA™ geralmente são mais lentos que em C/C++, devido à necessidade das JVMs interpretarem o *byte code*. Esta questão sempre foi controversa, pois melhorias nas implementações das JVMs têm reduzido as diferenças. Em alguns casos específicos este ponto não é mais verdadeiro. Isto também não se aplica quando um compilador JIT (*Just-in-Time*) é utilizado, pois o mesmo gera código diretamente para a máquina alvo. Um exemplo é o compilador HotSpot, da Sun Microsystems.

Para resolver estes problemas, o *Real-Time Java Experts Group*[2] definiu a Especificação de Tempo Real para JAVA™ (*Real Time Specification for* JAVA™, RTSJ) (RTJ 03; DIB 02; BOL 00b), a primeira especificação organizada e realizada pela comunidade JAVA™ através do JCP (*Java Community Process*) (PRO 04), em 1998, que define as seguintes carecterísticas:

- novos modelos de gerenciamento de memória que podem ser utilizados no lugar do coletor de lixo;

- acesso a memória física;

- uma granularidade de tempo maior, especial para sistemas tempo real;

[2]Grupo de especialistas em sistemas de tempo real e JAVA™, entre eles Greg Bollella (líder), James Gosling e Peter Dibble

- garantias mais fortes quanto à eligibilidade de *threads*.

Também objetivando aumentar a produtividade no desenvolvimento de aplicações tempo real, o Modelo Reflexivo Tempo Real (RTR) foi criado (FUR 97). Desenvolvido com o objetivo de adicionar conceitos de tempo real em linguagens de múltiplo uso, este Modelo eleva o nível destas linguagens, facilitando o desenvolvimento e a manutenção, além de aumentar a flexibilidade destes sistemas e diminuir o *gap* semântico entre as fases. Esta elevação no nível das linguagens praticamente permite que o produto da especificação seja o projeto do STR, e que este seja a entrada para a fase de implementação. A integração do Modelo RTR com JAVA™ iniciou-se em (KUE 01), com a implementação de um pré-processador para a linguagem Java/RTR (derivada da integração entre estes dois conceitos).

1.3 Caminho a ser seguido

Mesmo a especificação RTSJ sendo um importante passo para auxiliar o desenvolvimento de aplicações tempo real, seu planejamento foi feito com a idéia principal de generalidade. É uma qualidade louvável, mas em sistemas tempo real isto pode se tornar também um impedimento tanto quanto às restrições de recursos. Exemplo disto é a API de escalonamento, que foi desenvolvida para aceitar qualquer tipo de algoritmo. Mesmo cobrindo uma gama muito variada de aplicativos, é muito mais complexo para uma aplicação que precisa apenas de um escalonador preemptivo de prioridades.

Enquanto isto, o Modelo RTR propõe uma utilização menos complexa dos requisitos de controle das restrições temporais, separando o que deve ser a lógica do aplicativo (nível base) das questões de controle (nível meta).

Assim sendo, parece-nos altamente produtivo facilitar a utilização da especificação RTSJ utilizando o Modelo RTR, e é esta a proposta para este trabalho. Integrando a linguagem JAVA™ com o Modelo RTR, geramos uma nova linguagem, Java/RTR, que conterá as vantagens de ambos. Com o código escrito nesta nova linguagem, um tradutor será responsável por gerar código JAVA™ baseado na RTSJ, que então poderá ser compilado e executado.

1.4 Objetivos

1.4.1 Objetivo Geral

Este trabalho propõe a criação de um ambiente de desenvolvimento, focado para aplicações que possuam restrições temporais não-críticas, que utilizam a linguagem JAVA™ na Plataforma Eclipse. Esta integração consiste, basicamente, em desenvolver um *plug-in* que possa ser utilizado na Plataforma Eclipse e que adicione os conceitos necessários ao ambiente para facilitar o desenvolvimento e manutenção de sistemas tempo real escritos na linguagem Java/RTR.

1.4.2 Objetivos Específicos

- Definir formalmente a linguagem Java/RTR.
- Criar um tradutor que interprete código Java/RTR e gere código JAVA™ que utilize a especificação RTSJ.
- Integrar este tradutor na Plataforma Eclipse.

1.5 Metodologia

A implementação do trabalho passará por diversas etapas, sendo elas:

- Reestruturação e reimplementação do pré-processador para Java/RTR apresentado em (KUE 01), removendo as limitações do mesmo e permitindo uma melhor integração entre o Modelo RTR e o compilador de código JAVA™.
- Desenvolvimento de um tradutor que converta o código escrito utilizando Java/RTR em um código JAVA™ correto, compatível, aceito pelo RTSJ, e semanticamente equivalente ao mesmo.
- Implementação de um editor para o IDE Eclipse que permita a utilização da gramática proposta pelo Modelo RTR, contendo os seguintes recursos principais:
  - destacador de sintaxe (*syntax coloring*)

  - compilador integrado (*builder*)

- Desenvolvimento de assistentes (*wizards*) para a criação de objetos do Modelo RTR.

O escopo deste trabalho se restringe ao desenvolvimento de aplicações tempo real sem restrições críticas (soft), que queiram utilizar a linguagem JAVA™ como ponto de partida. O não atendimento destas restrições não resulta em perigos ou catástrofes, mas apenas reduz, de certa forma, a utilidade do sistema.

Praticamente inexistem trabalhos correlatos à este, pelos seguintes motivos principais: a especificação RTSJ é recente, e apenas agora está sendo explorada e utilizada; o Modelo RTR também é recente, e seu escopo de abrangência ainda é limitado a algumas regiões do Brasil; não existem *plug-ins* para a Plataforma Eclipse para desenvolvimento de STR, apesar da mesma se encontrar já na terceira versão e da comunidade estar crescendo rapidamente.

Não serão tratados assuntos como o projeto destas aplicações, nem metodologias do projeto. Parte-se do pressuposto que o projeto foi desenvolvido e que o momento é o de implementar a solução. Metodologias como ROOM (Real-Time Object-Oriented Modeling) (SEL 92; SEL 94) e Octopus (AWA 96) não serão discutidas, nem linguagens para modelagem de sistemas tempo real, como RT-UML (DOU 04). Obviamente que pode-se retornar ao projeto para corrigir ou alterar sua estrutura, mas estas iterações não serão abordadas neste trabalho.

1.6 Estrutura da Dissertação

No capítulo 2 é apresentado e detalhado o Modelo RTR, sua estrutura, seu funcionamento e seus requisitos, e a Especificação de Tempo Real em JAVA™ é descrita e comentada.

A linguagem Java/RTR é explicada no capítulo 3, e exemplos da sua sintaxe são mostrados e comentados.

As regras semânticas e de tradução de código ganham destaque no capítulo ‘Tradutor RTR2Java’, onde alguns detalhes da implementação e o framework

desenvolvido são descritos.

No capítulo seguinte, ‘Integração com a Plataforma Eclipse’, a Plataforma Eclipse é descrita e o *plug-in* criado tem sua implementação detalhada.

No capítulo final são apresentados os resultados do trabalho e são feitas algumas discussões, para, após, chegar a algumas conclusões.

Capítulo 2

Revisão Bibliográfica

2.1 Introdução

Neste capítulo serão feitas considerações sobre os pilares deste trabalho, notadamente o Modelo RTR e a Especificação Tempo Real em JAVA™. São estes conhecimentos que fundamentam o restante do texto, e o entendimento das seções aqui apresentadas é vital para o aproveitamento do trabalho como um todo.

2.2 Modelo RTR

Apresentado em (FUR 97) como uma tese de doutorado defendida no curso de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina, o Modelo Reflexivo Tempo Real visa facilitar o desenvolvimento de sistemas tempo real não-críticos (STR soft). Este tipo de aplicação utiliza a política de melhor esforço na sua execução, ao contrário das aplicações tempo real críticas. Para tanto, o Modelo RTR permite a associação de restrições e exceções temporais aos métodos de um objeto (caracterizando-os como tarefas) e inclui um escalonador tempo real no nível da aplicação.

O Modelo RTR é independente de linguagem de programação, facilita a reusabilidade de código e sua manutenção, permitindo o tratamento das questões temporais de uma forma flexível e natural para o desenvolvedor.

A reflexão computacional é introduzida no Modelo RTR através da

abordagem de meta-objetos. Segundo esta abordagem, para cada objeto do sistema existe um meta-objeto correspondente. Assim, os objetos possuem dados e métodos relativos, exclusivamente, às suas funcionalidades e são chamados de objetos-base, enquanto que os meta-objetos são responsáveis por controlar o comportamento temporal destes objetos-base. Isto contribui para o aumento da flexibilidade e da manutenibilidade do software, bem como sua evolução, pois as políticas de controle podem ser alteradas a qualquer momento, inclusive dinamicamente, sem que se modifique os objetos-base. Naturalmente, esta abordagem contribui ainda para a independência de ambiente operacional. No Modelo RTR, todos os aspectos temporais (restrições, exceções e escalonamento), aspectos de concorrência e aspectos de sincronização são tratados de forma reflexiva.

2.2.1 Estrutura do Modelo RTR

Estruturalmente, o Modelo RTR é composto por quatro tipos de objetos, detalhados a seguir. O quinto tipo seriam as classes básicas de JAVA™, que não são cobertas aqui.

2.2.1.1 Objeto Base de Tempo Real (OBTR)

Além de implementarem a lógica da aplicação, são objetos que suportam a associação de restrições temporais e manipuladores de exceção à declaração de seus métodos, e a declaração de novos tipos de restrições temporais. Cada nova declaração temporal deve prover uma implementação no Meta-Objeto Gerenciador associado.

Estes objetos são criados pelo desenvolvedor da aplicação, sendo os únicos a serem instanciados explicitamente. São definidos em uma *Real-Time Base Class (RTBC)*, que representa uma classe que define os OBTRs, e não possuem uma interface para ser implementada. Cada OBTR tem seu próprio Meta-Objeto Gerenciador, que será o responsável por controlar suas questões temporais, de sincronização e de concorrência.

2.2.1.2 Meta-Objeto Gerenciador (MOG)

São os objetos que gerenciam as chamadas com restrições temporais do seu OBTR associado, definem o controle da concorrência e fazem a sincronização, além

de conter as implementações das restrições temporais. São definidos através das *Manager Meta-Class (MMC)*.

A instanciação do MOG é feita automaticamente quando da criação de uma instância do OBTR, e o nome da MMC será sempre o nome da RTBC precedido por "Meta". Por exemplo, se o desenvolvedor escrever uma RTBC chamada "Teste", o nome da MMC correspondente deverá ser "MetaTeste".

As instâncias deste tipo trocam mensagens com o Meta-Objeto Escalonador e com o Meta-Objeto Relógio para prover o comportamento temporal desejado pelo OBTR controlado.

Segmento de Código 2.1: Interface dos objetos MOG do Modelo RTR

```
1  public interface ProtocolMMC {
2      // secao de gerenciamento
3      public void receiveRequest(...);
4      public void handleRequestWithoutTemporalConstraint(...);
5      // secao de concorrencia
6      public void releaseActivationRequest(...);
7      public void endOfExecution(...);
8      // secao de sincronizacao
9      public boolean verifySynchronization(...);
10     public void updateSynchronization(...);
11 }
```

Aqui está sendo utilizanda a sintaxe de JAVA™ para apresentar a interface básica das classes MMC, por ser a linguagem utilizada na implementação deste trabalho. Não estão sendo detalhados os parâmetros pois a definição do Modelo não faz nenhuma restrição aos mesmos. Os mesmos são descritos na seção de detalhamento da implementação.

Cada método tem sua funcionalidade:

receiveRequest ponto de partida para a execução de um método com restrição temporal. Responsável por iniciar o fluxo de controle interno no meta-objeto.

handleRequestWithoutTemporalConstraint gerenciador de métodos que não possuem

restrição temporal. Será chamado pelo `receiveRequest` quando o mesmo descobrir que o método chamado não posui restrição temporal.

releaseActivationRequest responsável pelo controle da concorrência interna no meta-objeto. Chamado antes da execução de um método, bloqueia o meta-objeto durante a mesma.

endOfExecution também responsável pela concorrência interna, é chamado quando a execução de um método é finalizada, liberando o meta-objeto para a próxima execução.

verifySynchronization em conjunto com `updateSynchronization`, controla a sincronização interna no meta-objeto, utilizando uma máquina de estados, por exemplo. Este método é chamado pelo método `releaseActivationRequest`.

updateSynchronization atualiza o estado de sincronização interna do meta-objeto. Esta atualização é iniciada pelo método `endOfExecution`.

2.2.1.3 Meta-Objeto Escalonador (MOE)

Sua função básica é receber, ordenar e liberar os pedidos de escalonamento vindos de todos os meta-objetos gerenciadores da aplicação, seguindo determinada política de escalonamento. Deve haver uma única instância para cada nodo (JVM) aonde a aplicação estiver sendo executada (no caso de execução distribuída).

Segmento de Código 2.2: Interface dos objetos MOE do Modelo RTR

```
1 public interface ProtocolSMC extends Runnable {
2     public void releaseNextRequest();
3     public boolean schedule(...);
4     public boolean removeFromSchedulingQueue(...);
5 }
```

Os métodos desta classe tem os seguintes obetivos:

releaseNextRequest informa ao escalonador que o método escalonado terminou, e que o próximo pode ser liberado para execução.

schedule adiciona à fila de escalonamento uma nova chamada à um método.

removeFromSchedulingQueue remove da fila de execução o método passado como parâmetro.

A Listagem 2.2 mostra os requesitos básicos que uma implementação deve prover. Isto não impede, entretanto, que determinada implementação suporte funções adicionais, como por exemplo:

- substituição dinâmica da política de escalonamento utilizada. Mais informações na Seção ‘Mudanças dinâmicas do algoritmo de escalonamento’ (p.16);
- realizar análise dinâmica do próprio funcionamento, antecipando a detecção de violações temporais das tarefas que não poderão ser escalonadas e garantindo a satisfação das restrições temporais das tarefas aceitas. Mais detalhes adiante, na Seção ‘Análise de escalonabilidade dinâmica’ (p.16);
- combinar diferentes políticas de escalonamento para as diferentes classes de tarefas tempo real (como periódicas e aperiódicas). Veja adiante a Seção ‘Uso simultâneo de diferentes políticas de escalonamento’ (p.17).

2.2.1.4 Meta-Objeto Relógio (MOR)

É o relógio do sistema no formato de um objeto. Sua tarefa básica é realizar verificações temporais e ativações futuras de métodos. Também deve ser único para cada nodo executando a aplicação (no caso distribuído).

Segmento de Código 2.3: Interface dos objetos MOR do Modelo RTR

```
1 public interface ProtocolCMC extends Runnable {
2     public void programFutureActivation(...);
3     public void activate(...);
4     public void cancelProgrammedActivation(...);
5 }
```

Para esta classe, os métodos têm as seguintes responsabilidades:

programFutureActivation programa uma ativação futura de um método. Utilizado, principalmente para métodos periódicos, ou com início programado para o futuro.

activate realiza a ativação do método indicado.

cancelProgrammedActivation cancela a ativação de um método. Utilizado para finalizar métodos periódicos, talvez por terem sofrido alguma violação temporal.

2.2.2 Funcionamento do Modelo RTR

A dinâmica de funcionamento do Modelo RTR prevê a interação entre os vários objetos e meta-objetos que constituem uma aplicação tempo real, e pode ser assim descrita: quando um Objeto Base de Tempo Real (OBTR) requisita a ativação de um método, este pedido é redirecionado para o Meta-Objeto Gerenciador (MOG) correspondente. O MOG interage com o MOE para que o pedido seja escalonado no sistema, e com o MOR para que as restrições temporais associadas ao método sejam processadas. Se as restrições não forem violadas, o MOG será notificado pelo MOE (no tempo devido) e ativará o método solicitado e depois retornará a resposta ao OBTR que originou a chamada. No caso de violação temporal, a exceção temporal associada ao método será executada. A Figura 2.1 mostra a seqüência de passos seguida para a execução de um método com uma restrição temporal associada.

2.2.3 Expressividade do Modelo RTR

Além do que já foi apresentado, o Modelo RTR também permite que tarefas comuns ao desenvolvimento de STR sejam representadas sem a necessidade de novas adições sintáticas ou semânticas. Neste caso, o suporte destas facilidades acontece no nível meta, apenas redefinindo (através de extensão, alteração ou substituição) as funções básicas providas pelos meta-objetos Escalonador e Gerenciador. Entre estas novas funcionalidades, algumas estão comentadas abaixo.

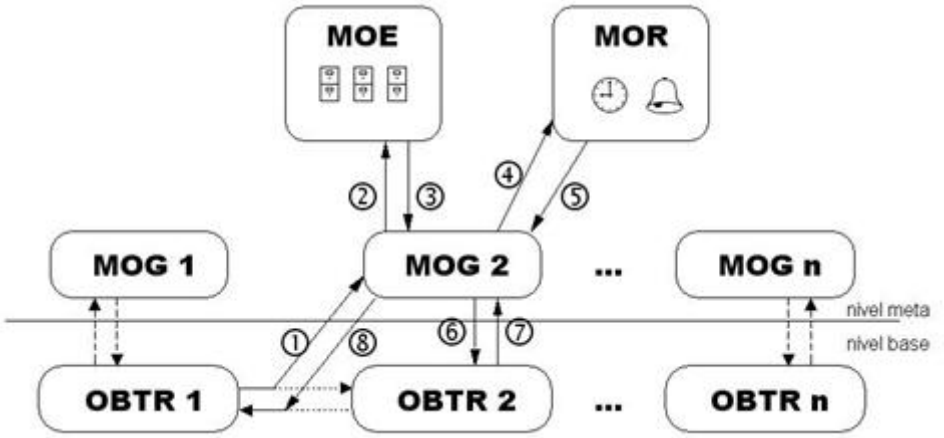

Figura 2.1: Chamada de um método com restrição temporal no Modelo RTR. Através do processo de compilação, a chamada é desviada para o meta-objeto (1), que será responsável por gerenciar a situação da tarefa no sistema. A mesma será escalonada (2)(3) e, se ainda houver tempo (4)(5), o método chamado será executado reflexivamente (6)(7), com o resultado retornando para o objeto que fez a chamada (8). Os números mostram a sequência seguida.

2.2.3.1 Reflexão de aspectos não temporais

Comportamentos como depuração, persistência, qualidade de serviço e controle estatístico são possíveis no modelo através das categorias às quais os métodos temporalmente restritos podem participar. Veja a Seção ‘Declaração de métodos’ (p.36) para mais detalhes.

2.2.3.2 Polimorfismo temporal

Muitas vezes é preferível uma solução aproximada dentro do prazo que uma precisa fora do prazo. Este tipo de técnica, conhecido como Computação Imprecisa (LIU 94; LIU 91), caracteriza-se por se fazer o possível dentro do prazo. Sua implementação no Modelo RTR é possível através de uma nova restrição temporal, e de diferentes versões do mesmo método, cada uma com diferentes tempos de execução. A nova restrição seria responsável por escolher dinamicamente o método que deve ser executado de acordo com o tempo disponível.

2.2.3.3 Ajuste dinâmico do tempo de execução

É muito importante o conhecimento do tempo de execução das tarefas em STR. Atualmente é difícil fazer um cálculo via análise de código, pricipalmente em linguagens que permitem laços infinitos e chamadas de métodos recursivas, e também é complexo fazer medições em tempo de execução. O Modelo RTR, através do seu mecanismo de reflexão, permite que medições sejam feitas pelos meta-objetos, de forma transparente ao desenvolvedor e adicionando o mínimo de sobrecarga no sistema, permitindo que os resultados sejam atualizados nos objetos-base dinamicamente.

2.2.3.4 Ajuste dinâmico dos atributos das restrições temporais

Os valores utilizados como parâmetros nas restrições temporais nem sempre são resultados de análises, mas apenas valores estimados. Em muitos casos uma alteração nestes parâmetros exige a parada total do sistema para sua reinicialização. É possível representar estes ajustes, previsíveis ou não, no Modelo RTR através, por exemplo, de uma restrição temporal periódica, mas que aceite um valor extra como delta para o período de ativação, dependendo de como o sistema está se comportando. E já que os parâmetros para as chamadas de métodos temporalmente restritos são passados a cada execução, os mesmos podem ser resultado de um cálculo que utilize o resultado da execução anterior.

2.2.3.5 Análise de escalonabilidade dinâmica

Funcionalidade vital para STR, a análise de escalonabiblidade permite que se saiba previamente se um conjunto de tarefas é escalonável ou não. Utilizando o Modelo RTR este problema pode ser tratado diretamente nas implementações dos escalonadores, implementando uma política de admissão de tarefas, por exemplo.

2.2.3.6 Mudanças dinâmicas do algoritmo de escalonamento

A mudança em tempo de execução da política de escalonamento é uma característica bem-vinda em muitos STR que apresentam diferentes modos de funcionamento. Esta qualidade pode ser inserida no Modelo apenas modificando-se a imple-

mentação do meta-objeto escalonador, para que o mesmo se adeque às características das tarefas sendo executadas, provavelmente utilizando o percentual de violações temporais ocorridas em um determinado intervalo como evento ativador da troca.

2.2.3.7 Uso simultâneo de diferentes políticas de escalonamento

A escolha de uma política de escalonamento adequada é requisito básico para que um STR satisfaça as restrições temporais. Mas nem sempre é fácil escolher uma única política que funcione bem todo o tempo de execução do sistema. Para diferentes tipos de tarefas, utilizar diferentes algoritmos de escalonamento geralmente é uma boa idéia. Obter diversas políticas de escalonamento no Modelo RTR requer apenas modificações no Escalonador, que poderia suportar diferentes algoritmos e diferentes filas de tarefas para diferentes classes, sendo que a classe de cada tarefa pode ser obtida dinamicamente.

2.2.3.8 Controlando reflexivamente a disponibilidade de memória

Mesmo sendo fundamental, a disponibilidade de tempo não é garantia de satisfação de requisitos temporais da aplicação. Também são necessários os demais recursos utilizados, principalmente a memória, disponíveis tanto no tempo quanto na quantidade desejadas. Geralmente STR críticos fazem a alocação de memória estaticamente, nem iniciando se a mesma não estiver disponível. Mas aplicações tempo real dinâmicas ou projetadas para serem executadas em ambiente de propósito geral não possuem esta garantia estática, e devem verificar dinamicamente se há memória disponível. Este problema pode ser tratado no Modelo RTR também de forma reflexiva, quando a memória será alocada na criação de um objeto. No caso de não haver recurso disponível, a execução do coletor de lixo pode ser requisitada.

2.3 Especificação Tempo Real para JAVA™

2.3.1 Histórico

Por ser uma linguagem interpretada por uma máquina virtual, a idéia de utilizar JAVA™ para aplicações que necessitassem suportar restrições temporais não parece plausível, mesmo com os avanços nas implementações das mesmas. Mas a idéia começou a ganhar forma em 1998, três anos após o lançamento da linguagem.

Ironicamente, a solicitação de uma especificação para tempo real em JAVA™, feita pela IBM, foi a primeira aceita no JCP (*Java Community Process*) (PRO 04; PRO 98), em dezembro daquele mesmo ano. Muito trabalho foi feito até que em junho de 2000 a especificação estivesse pronta para ser distribuída (BOL 00a).

Alguns princípios foram seguidos pelo *Expert Group* para se escrever a especificação, e os principais são os seguintes:

- compatibilidade com a plataforma JAVA™ 2;
- não fazer adições sintáticas a JAVA™ (novas palavras reservadas);
- permitir execução com previsibilidade;
- boa relação entre práticas atuais e características avançadas.

2.3.1.1 Sistemas tempo real multi-plataforma

Quando se trata de aplicações com restrições temporais não se pode depender da plataforma na qual a mesma será executada. Por ser multi-plataforma, a linguagem JAVA™ pode ser executada em computadores com diversos processadores ultra-rápidos, ou em dispositivos embutidos com processadores de baixo custo. Estas diferenças devem ser levadas em consideração quando do desenvolvimento de aplicações com restrições temporais. O mote da linguagem de que se pode escrever o código apenas uma vez e executar diversas vezes (*Write Once, Run Anywhere*) nem sempre funciona para STR. Citando Paul Bowman, no encontro do grupo de especialistas responsável pela RTSJ em Mendocino, Califórnia, 1999, é melhor pensarmos em WOCRAC (*Write Once* **Carefully**, *Run Anywhere* **Conditionally**).

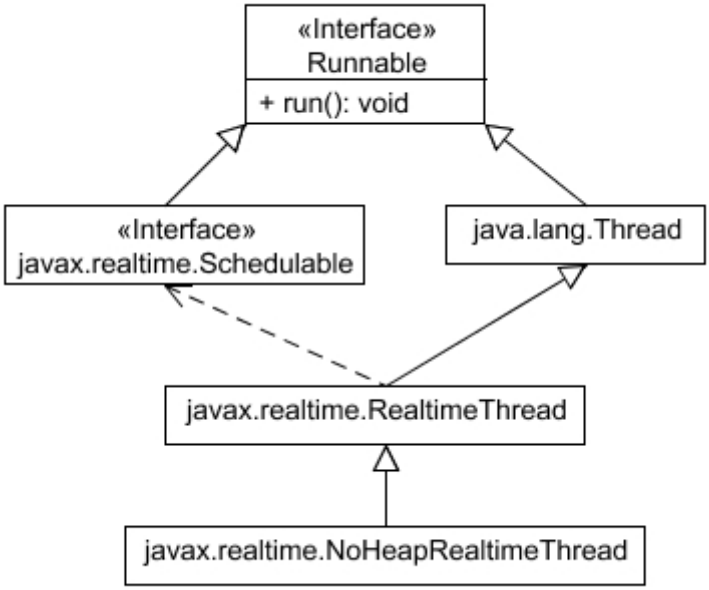

Figura 2.2: Diagrama das classes relativas a threads do RTSJ.

2.3.2 Especificação

A seguir, os seis pontos relevantes da especificação proposta serão detalhados.

2.3.2.1 Threads e escalonamento

O escalonamento de tarefas é um ponto central na computação com restrições temporais, independente do algoritmo de escalonamento utilizado. A especificação segue uma linha entre ser extremamente específica, permitindo aos desenvolvedores saberem exatamente como as coisas irão funcionar, e ao mesmo tempo sendo flexível para permitir novas implementações.

A especificação estende o conceito de *threads* de JAVA™ com dois novos tipos: `RealtimeThread` e `NoHeapRealtimeThread`. Esta última pode parar o coletor de lixo para executar, pois a mesma não pode referenciar nem alocar objetos na pilha de memória, ficando restrita a objetos permanentes. A Figura 2.2 mostra o diagrama das classes envolvidas.

Também é definido o conceito de objetos `Schedulable` (escalonáveis),

que serão gerenciados pelo escalonador. A API permite a implementação de diversos algoritmos, mas exige apenas a implementação de um algoritmo de prioridades preemptivo, que trabalhe com 28 prioridades, por ser uma técnica largamente utilizada nos sistemas tempo real comerciais e por sua utilização nas aplicações legadas de JAVA™. Também é requerido que o protocolo de herança de prioridades seja implementado por padrão, além de permitir a implementação de outros protocolos.

2.3.2.2 Coletor de lixo

Um dos maiores problemas no desenvolvimento de aplicações tempo real em JAVA™ é seu mecanismo de liberação de memória. Na verdade, a especificação de JAVA™ (GOS 97) não exige um coletor de lixo: um sistema de alocação dinâmica de memória é necessário, mas não um para liberá-la. Evidentemente, a não inclusão deste mecanismo na máquina virtual reduziria drasticamente a utilização da mesma.

Da mesma forma, a RTSJ também não requer um coletor de lixo. Entretanto, é especificado pelo menos uma API que provê uma classe particular para um algoritmo de coletor de lixo.

Uma maneira encontrada para evitar que a *thread* que executa o coletor de lixo preemptasse as outras *threads* (principalmente as que possuem restrições temporais), foi criar um tipo especial da mesma (*no-heap real-time thread*), que não tem permissão para acessar a pilha de memória. Desta forma, este tipo especial de *threads* pode preemptar o coletor de lixo a qualquer tempo, pois não poderia alterar referência de forma a tornar inconsistente o trabalho sendo realizado pelo coletor. Assim, este tipo de *thread* pode cumprir com mais facilidade e segurança suas restrições temporais.

2.3.2.3 Tratadores de eventos assíncronos

Na maior parte dos sistemas de tempo real eventos são gerados e devem ser respondidos o mais rápido possível. Mas uma implementação que crie uma *thread* para responder cada evento gerado, mesmo sendo de fácil implementação, impõe uma sobrecarga muito grande de tempo e memória. A criação de uma *thread*, por ser um serviço de alocação de recursos, é cara, e por isto é evitada pelos programadores de sistemas tempo real.

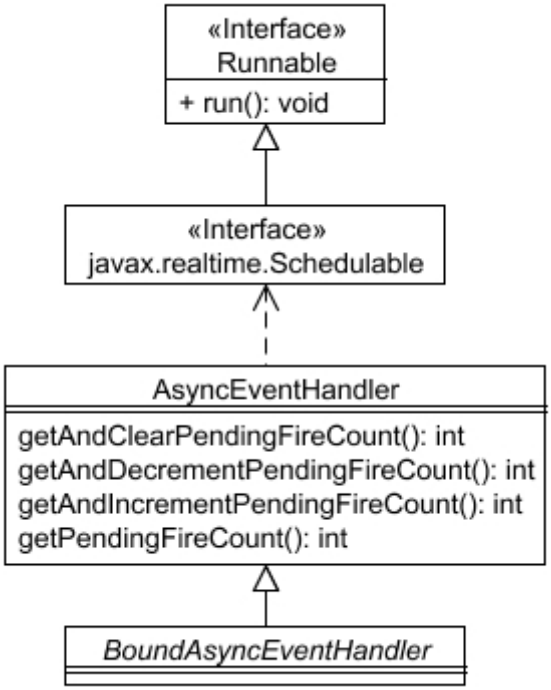

Figura 2.3: Diagrama das classes relativas a eventos assíncronos do RTSJ.

A utilização de manipuladores de eventos assíncronos é uma tentativa de juntar as vantagens de se criar uma *thread* para servir o evento sem pagar o preço de criação da mesma. A especificação introduz o conceito de "acontecimentos" para fazer a ligação entre os eventos externos à JVM e os manipuladores de eventos assíncronos.

Cada instância de `AsyncEventHandler` é responsável por um tipo de `AsyncEvent`, mas não tem uma *thread* associada. Se uma for necessária, então deve-se utilizar a classe `BoundAsyncEventHandler`. A Figura 2.3 mostra o diagrama das classes definidas na especificação.

2.3.2.4 Transferência assíncrona de controle

Uma das últimas adições à especificação, este mecanismo permite que uma *thread* levante uma exceção dentro do fluxo de outra *thread*. Um mecanismo parecido já existe em JAVA™, mas por ser fraco e perigoso não é recomendado nem foi utilizado.

A transferência assíncrona de controle é importante pelos seguintes mo-

tivos:

1. É uma maneira de se cancelar uma *thread* de uma forma forçada mas controlada.

2. É um modo de retirar o fluxo de um laço de uma *thread* sem exigir que a mesma fique verificando uma variável de controle.

3. É um mecanismo de uso geral para controle de estouro de tempo (*timeout*).

4. Permite um controle poderoso do escalonador sobre as *threads* sendo executadas.

Por outro lado, sua implementação é muito complexa, pois:

1. O código que não está preparado para ser interrompido pode falhar seriamente se a JVM subitamente sai dele.

2. Não é possível simplesmente pular do ponto de execução para a cláusula **`catch`** correta. A plataforma deve, obrigatoriamente, passar pelas cláusulas **`catch`** e **`finally`** dos métodos não interrompíveis para corretamente servir a exceção.

3. Métodos aninhados podem estar esperando por diferentes exceções assíncronas. O sistema de execução deve garantir que o bloco **`catch`** correto seja executado.

2.3.2.5 Tempo e Temporizadores

O controle de tempo é de suma importância em STR, pois tempo é essencial nestes sistemas. Devido a isto, o RTSJ define uma precisão de nanosegundos, diferente da precisão de milisegundos especificada pela linguagem JAVA™. Também foram definidas novas classes para representar as diferenças de tempo: `AbsoluteTime` representa um determinado momento no tempo; `RelativeTime` significa um intervalo de tempo; e `RationalTime` representa um intervalo dividido por certa frequência. A Figura 2.4 mostra o diagrama de classes.

É comum a aplicações tempo real utilizarem temporizadores para executarem tarefas em um tempo futuro, ou executarem tarefas periódicas (tocar músicas, coletarem dados, verificarem a situação de algum equipamento, etc). A especificação define dois tipos de temporizadores: `OneShotTimer`, que gera um evento no final do seu

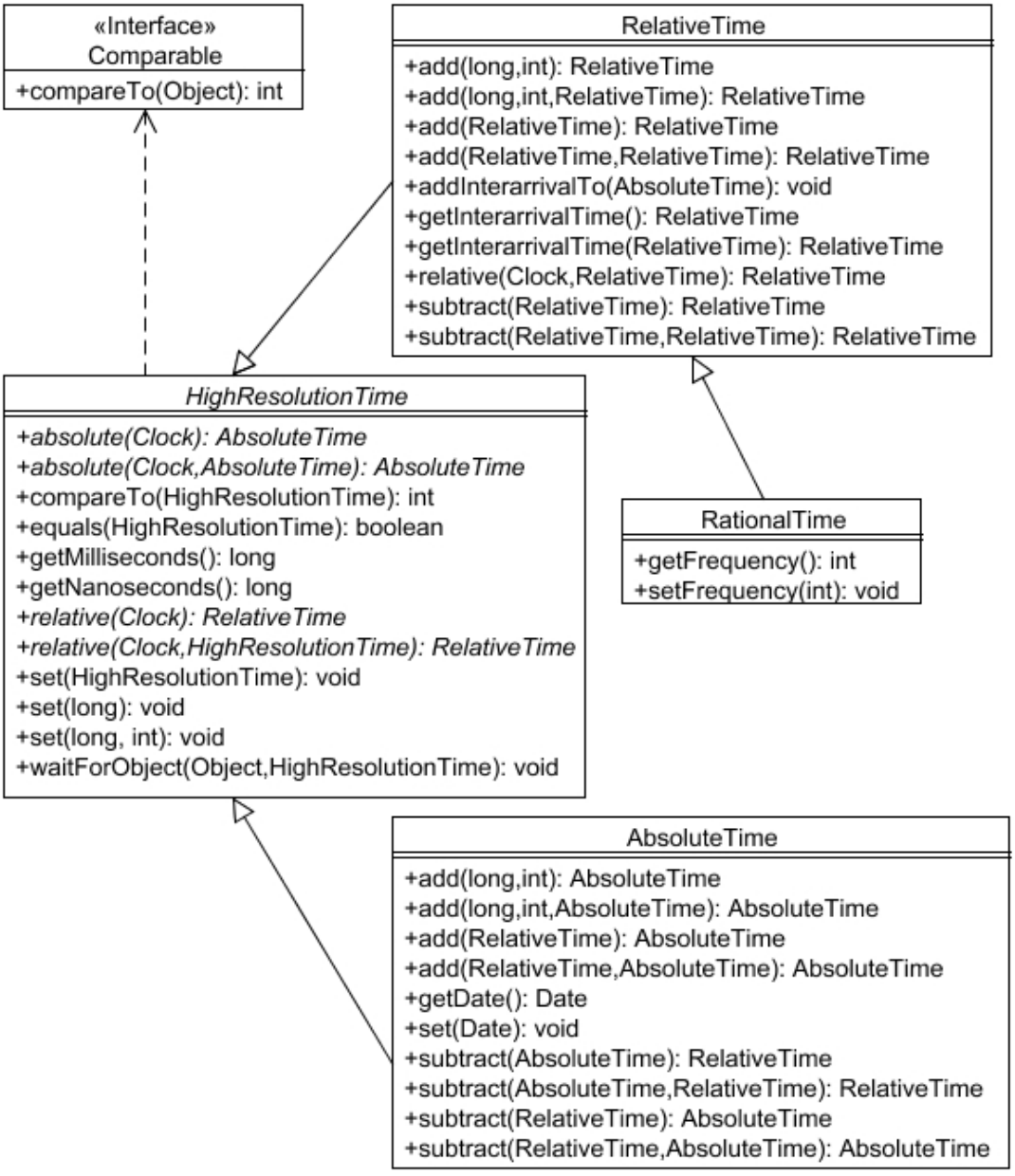

Figura 2.4: Diagrama das classes que representam o conceito de tempo do RTSJ.

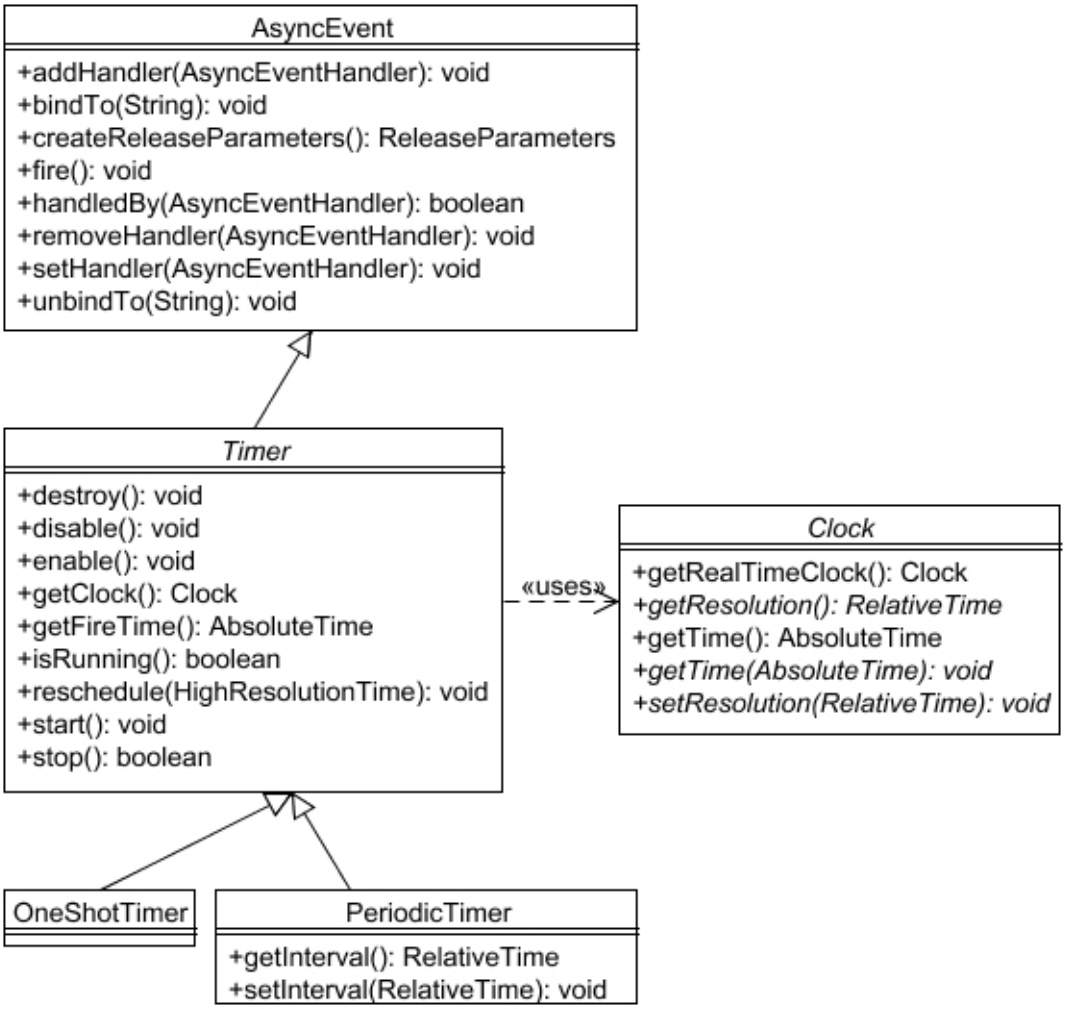

Figura 2.5: Diagrama das classes relativas a temporizadores do RTSJ.

intervalo; e `PeriodicTimer`, que gera eventos periódicos. Estes eventos são tratados por objetos `AsyncEventHandler`, vistos na Seção ‘Tratadores de eventos assíncronos’ (p.20). Na Figura 2.5 é possível visualizar o diagrama de classes dos tipos envolvidos.

2.3.2.6 Alocação de memória

A utilização de *threads* tempo real que não acessem a memória é inútil isoladamente, pois estaria restrita aos tipos básicos de dados. Para expor sua utilidade, dois domínios de alocação de memória novos foram criados: a memória permanente e a memória de escopo.

A memória permanente, como o próprio nome sugere, nunca tem seu conteúdo avaliado pelo coletor de lixo, sendo que seu tempo de vida é o mesmo da JVM. Ela serve, principalmente, para os sistemas tempo real que alocam todos seus recursos durante a fase de inicialização e então são executados eternamente sem necessitar alocar ou liberar este conteúdo.

Mesmo sendo fácil de explicar e implementar este tipo de memória, sua utilização deve ser cautelosa pelos seguintes motivos:

- JAVA™ não encoraja o programador a reutilizar os objetos existentes. Alguns exemplos são os objetos nos quais suas propriedades são definidas apenas na sua criação (geralmente são passados parâmetros para o construtor). Também um mesmo objeto não pode mudar seu tipo durante sua existência, devido à JAVA™ ser fortemente tipada.

- As bibliotecas JAVA™ criam livremente novos objetos. As chamadas às bibliotecas de coleções (ArrayList `public Object set(int index,Object element)`, por exemplo) e de funções matemáticas (`public static double random()`, por exemplo) criam objetos descartáveis para executarem suas funções. Isto pode levar à uma sobrecarga na memória permanente, impossibilitando a criação de objetos necessários ao sistema. Não é obrigatório que o código dos objetos que possuem restrição temporal utilizem as bibliotecas padrões, mas estas são uma das grandes vantages de JAVA™, e o esforço envolvido para reescrevê-las utilizando conceitos de código tempo real seria muito grande e desanimador.

A memória de escopo não é tão simples quanto a memória permanente, mas resolve os problemas citados acima. De uma forma simplificada, a memória de escopo funciona como uma pilha de objetos. Quando uma *thread* entra neste tipo de memória, ela passa a alocar seus objetos todos nesta memória, até quando ela acessar uma memória de escopo aninhada ou sair da memória atual. Ao sair do escopo, a *thread* não pode mais acessar os objetos alocados lá, e a JVM fica livre para recuperar toda a memória utilizada de uma só vez.

Desta forma o problema da utilização das bibliotecas está resolvido: se a *thread* entra em um escopo antes de chamar um método de uma biblioteca e sai logo

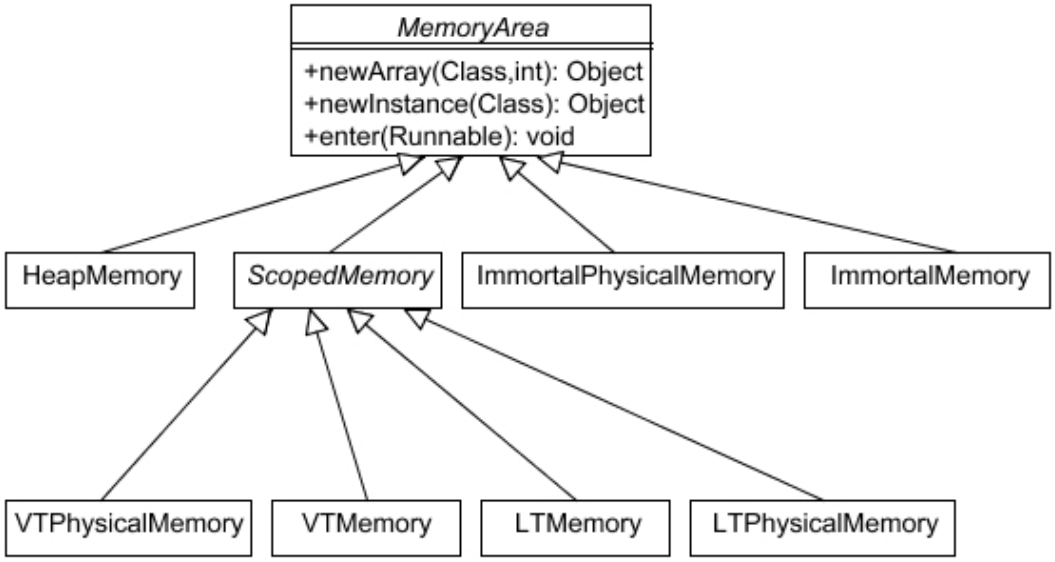

Figura 2.6: Diagrama das classes relativas a tipos de memórias do RTSJ.

após o retorno, os objetos eventualmente criados pela chamada serão automaticamente liberados para serem recolhidos pelo coletor de lixo.

A memória de escopo tem implementações com tempo de alocação linear (`LTMemory` e `LTPhysicalMemory`) e tempo de alocação variáveis (`VTMemory` e `VTPhysicalMemory`). A Figura 2.6 mostra a hierarquia das classes relativas às memórias do RTSJ.

O principal motivo para a existência das memórias permanente e de escopo é o desempenho. A especificação define regras de acesso para as *threads* de tempo real, e regras que governam a manutenção de referências a objetos na memória geral e na de escopo. Estas regras devem ser reforçadas pelo verificador de classe ou pelo motor de execução, mesmo afetando o desempenho da JVM.

2.3.2.7 Acesso à memória

Tipos especiais de memórias, dispositivos de I/O que podem ser acessados com operações de carregar e salvar, e comunicação com outras tarefas via memória compartilhada são pontos importantes para sistemas embutidos.

Tipos especiais de memórias estão fortemente relacionados com desem-

penho. A previsibilidade do desempenho é estritamente de interesse de aplicações tempo real, e alguns atributos das memórias (como cache, compartilhamento e paginação) têm grande impacto na previsibilidade do código que utiliza a mesma.

A especificação define uma classe para acesso direto à memória, que suporta operações do tipo “pegar” e “colocar”. Por serem executadas através do sistema de segurança da JVM, estas operações permitem a utilização de objetos como ponteiros, sem comprometer a plataforma, e provendo acesso direto à memória para suportar o desenvolvimento de drivers de dispositivos escritos em JAVA™ e programas que compartilham a memória com outras tarefas.

A inclusão de classes que representam diferentes tipos de memórias não traz vantagens no desempenho das aplicações tempo real na plataforma. Sua adição se deve ao interesse no desenvolvimento de drivers de dispositivos em JAVA™, aumentando também a utilidade da linguagem para a comunidade de sistemas embutidos com restrições temporais.

2.3.3 Implementação

O que foi explicado até agora faz parte da especificação formal. Para que isto se torne prático e possa ser utilizado, é necessário transformar todas estas exigências em código. Isto é um dos requisitos do JCP: cada especificação deve conter, além da documentação formal, uma implementação de referência (*Reference Implementation*, RI) e um conjunto de testes de compatibilidade (*Test Compatibility Kit*, TCK).

2.3.3.1 Implementação de referência

A implementação de referência (RI, *Reference Implementation*) foi desenvolvida pela empresa TimeSys (TIM 01b; TIM 01a), uma das participantes do JCP e do *Experts Group*, e foi disponibilizada em Novembro de 2001. A RI exige que certas funcionalidades sejam providas pela JVM sobre a qual a mesma será executada. Mas para que a JVM possa fornecer mais operações é necessário que as mesmas sejam providas pelo sistema operacional, e que possam ser acessadas via código JAVA™. É aqui que entram os métodos declarados nativos, cuja implementação não é feita diretamente

em JAVA™, mas sim na linguagem nativa do sistema operacional sobre o qual a JVM está implementada. Isto exige que sejam disponibilizadas bibliotecas com as implementações destas funções. A RI delega diversas tarefas diretamente para o SO através de métodos nativos, o que acaba por destruir a portabilidade do código que utiliza as estruturas da implementação de referência do RTSJ. A máquina virtual disponibilizada com a implementação de referência é a TJVM.

Neste ponto é preciso fazer uma ressalva: a implementação de referência foi homologada para funcionar apenas no sistema operacional Linux 2.4 (sem o sistema de herança de prioridades) ou no TimeSys Linux 3.0 (e posteriores), este último provendo suporte completo aos aplicativos tempo real sobre o kernel do Linux. Estas restrições se devem ao código do kernel do Linux ser distribuído livremente, o que permitiu sua modificação para adição dos controles tempo real, enquanto o mesmo não é possível em sistemas operacional como Microsoft Windows, HP-AUX, IBM OS/2 ou Sun Solaris, todos estes proprietários.

Outro detalhe sobre a implementação de referência é a necessidade dos arquivos serem gerados para a versão 1.3 de JAVA™. Para isto deve-se utilizar a opção `-source 1.3` quando é feita a compilação dos arquivos fontes. Esta restrição impede a utilização de versões mais novas da tecnologia JAVA™, como o uso de asserções (disponível no 1.4), e tipos genéricos e enumerações (disponíveis na versão 5.0).

2.3.3.2 jRate

jRate (COR 04; COR 02) é uma implementação da especificação RTSJ *open-source* para JAVA™ tempo real, ainda em fase de desenvolvimento. Ele, mais do que uma simples implementação da especificação, estende o sistema de execução do compilador GNU *open-source* para JAVA™ (GNU Compiler fo JAVA™, GCJ), provendo uma plataforma para o desenvolvimento de aplicações compatíveis com RTSJ.

Ao invés de prover uma máquina virtual que interprete o *byte code*, o compilador *ahead-of-time* do jRate gera código nativo. Como o próprio autor reconhece, o ponto negativo é ter que recompilar o código cada vez que o mesmo precisa ser portado para uma nova arquitetura. Mesmo assim, também é comentado que este preço é pequeno se comparado a melhoria no desempenho proporcionada pela solução.

Mesmo sendo um trabalho interessante e estando em uma fase bastante adiantada no desenvolvimento, o jRate está aqui apenas para conhecimento geral, pois o mesmo não se encaixa na solução proposta por este trabalho, exatamente por gerar código nativo, e por servir como *front-end* para outra ferramenta (GCJ).

O ponto mais proveitoso desta implementação são os testes de desempenho realizados (COR 04) entre o jRate e a implementação de referência, mostrando grande vantagem para o primeiro.

Capítulo 3

Java/RTR

Java/RTR consiste da integração do Modelo RTR na linguagem de programação JAVA™, produzindo uma linguagem de programação de alto nível e ao mesmo tempo com suporte a restrições temporais.

A seguir os detalhes desta nova linguagem são apresentados.

3.1 Estrutura léxica

Lexicamente, novas palavras reservadas foram incluídas na linguagem JAVA™[1] para se chegar até Java/RTR[2]:

- ***rtbc***
- ***mmc***
- ***smc***
- ***cmc***
- ***eventtrigger***
- ***timetrigger***

[1] Palavras reservadas de JAVA™ estão destacadas em negrito

[2] Palavras reservadas de Java/RTR estão destacadas em negrito e itálico, tanto no texto como nas listagens de códigos

- ***rtconstraint***
- ***rtthrow***
- ***rtcategory***
- ***rttype***

Para seguir o modelo da linguagem JAVA™ e as convenções propostas pela Sun (SM 99), todas as palavras são escritas em caracteres minúsculos, e sem espaço entre suas expressões (por exemplo, ***timetrigger*** em Java/RTR assim como **instanceof** em JAVA™). Por outro lado, foram escolhidas siglas para representar os tipos das classes especiais, ao invés das expressões por extenso, por se considerar que as siglas representam de forma completa e não-ambígua os conceitos representados.

Nenhuma outra alteração léxica foi feita nas palavras reservadas já existentes, nos operadores ou nos seus significados.

Nas seções seguintes será possível identificar a utilização de cada nova palavra reservada.

3.2 Estrutura sintática

Sintaticamente, entre JAVA™ e Java/RTR existem diferenças apenas nas estruturas necessárias para suportar os conceitos previstos no Modelo. A seguir são apresentadas as estruturas criadas para Java/RTR (representadas em BNF). Antes, porém, um breve comentário sobre esta notação.

3.2.1 Notação BNF

Para representar a sintaxe de Java/RTR utilizaremos a notação BNF[3], tipicamente utilizada para representar a sintaxe de linguagens de programação representadas formalmente por Gramáticas Livre de Contexto. Nesta notação são utilizadas as seguintes convenções:

[3]Backus Naur Form, ou Forma de Backus Naur

Atribuições: representadas pelo operador **::=**, permitem a descrição das produções da gramática utilizada. No lado esquerdo do operador fica o símbolo não-terminal, e no seu lado direito sua expansão, que pode conter símbolos terminais e não terminais.

Exemplo: $< rtrClassType > ::= \ rtbc$

No exemplo, o não-terminal (entre $<$ e $>$) "rtrClassType" deve ser substituído pelo terminal ***rtbc***, sempre que o primeiro aparecer.

Operador de opção: o caracter | permite expressar produções alternativas em uma mesma regra, onde apenas uma delas é escolhida.

Exemplo: $< rtrClassType > ::= \ rtbc \mid mmc \mid smc \mid cmc$

Neste caso, o não-terminal rtrClassType deve ser substituído por ***rtbc***, por ***mmc***, por ***smc*** ou por ***cmc***, mas não por duas opções ao mesmo tempo, e também não pode ser substituído por ϵ (caracter que representa o terminal vazio).

Expressões opcionais: são expressões entre colchetes, que podem ou não aparecer na sintaxe.

Exemplo: $< rtrClassType > ::= \ public \, [\, rtbc \,] \, class$

Agora, o não-terminal deve ser susbtituído pela expressão **public *rtbc* class** ou por **public class**. Neste último caso, a opção não seria escolhida, mas sim o terminal vazio ϵ.

Expressões repetitivas: são expressões entre chaves que podem aparecer zero ou mais vezes na gramática.

Exemplo: $< rtrClassType > ::= \ public \, \{\, rtbc \,\} \, class$

Neste exemplo existem infinitas possibilidades de substituição, dependendo do número de vezes que ***rtbc*** aparece. Esta expressão deverá ser transformada em **public class**, ou **public *rtbc* class**, ou **public *rtbc rtbc* class**, e assim sucessivamente, para quantos ***rtbc*** forem desejados.

As expressões entre aspas que aparecem nas declarações das sintaxes apresentadas nas próximas seções são identificadores e caracteres terminais que devem aparecer entre os não-terminais.

3.2.2 Declaração de classe

A declaração de classes de JAVA™ foi incrementada para que fosse permitida uma identificação opcional sobre o tipo da classe que está sendo declarada. Os tipos de classes especiais são os permitidos pelo Modelo RTR: Real-Time Base Class, Manager Meta Class, Scheduler Meta Class e Clock Meta Class, cada tipo devidamente representado pela palavra reservada criada com seu acrônimo.

```
<class_declaration> ::= <modifiers>
                        [ <rtr_class_type> ]
                        "class" <identifier>
                        [ "extends" <name> ]
                        [ "implements" <name_list> ]
                        <class_body>

<rtr_class_type> ::= ["rtbc" | "mmc" | "smc" | "cmc"]
```

Código 1: Sintaxe da declaração de classe em Java/RTR.

Com a informação do tipo da classe sendo declarada é possível fazer algumas verificações durante a compilação do código, conforme será descrito nas seções ‘Implementação do protocolo do Modelo RTR’ (p.57), ‘Implementação das restrições temporais definidas’ (p.58) e ‘Restrições nas subclasses do Escalonador e do Relógio’ (p.58).

Entretanto, não são todos os modificadores que fazem sentido em todos os tipos de classe. Excluindo-se as interfaces, que são representadas por outras estruturas sintáticas, classes internas, classes declaradas dentro de métodos, classes anônimas e classes estáticas aninhadas também não podem conter os modificadores de classes especiais.

3.2.2.1 Exemplos

Abaixo temos alguns exemplos de declarações de classes que ilustram a utilização da sintaxe e das palavras reservadas criadas.

Segmento de Código 3.1: Exemplo de declaração de classe do tipo RTBC em Java/RTR

```
1 public rtbc class TesteRTBC {
2    private int valor = 20;
3    public String nome = "Sem nome";
4 }
```

Esta listagem declara uma classe RTBC com apenas dois atributos e nenhum método. Note-se que é a mesma declaração de classe de JAVA™, exceto pela adição da palavra reservada ***rtbc***.

Segmento de Código 3.2: Exemplo de declaração de classe do tipo MMC em Java/RTR

```
1 public mmc class MetaTesteRTBC {
2   public void erro() {
3     System.out.prinln("Ocorreu um erro!");
4   }
5 }
```

A Listagem 3.2 declara uma classe do tipo MMC contendo um método que pode ser utilizado como uma alternativa para o caso de um descumprimento de uma restrição temporal.

Segmento de Código 3.3: Exemplo de declaração de classe do tipo CMC em Java/RTR

```
1 import br.ufsc.inf.javartr.lib.AbstractCMC;
2 public cmc class CMC extends AbstractCMC {
3   public void run() {}
4 }
```

Com uma freqüência muito menor de implementação (talvez uma única para um sistema completo), as classes do tipo CMC são muito importantes para o correto funcionamento de um STR. O código mostrado na Listagem 3.3 declara uma classe chamada CMC que não estende nenhum dos métodos pré-definidos.

Segmento de Código 3.4: Exemplo de declaração de classe do tipo SMC em Java/RTR

```
1 import javax.realtime.Schedulable;
2 import br.ufsc.inf.javartr.lib.AbstractSMC;
```

```
3  public smc class FIFOScheduler extends AbstractSMC {
4    protected boolean addToFeasibility(Schedulable s) {
5      listWithConstraint.add(s); return true;
6    }
7    public void fireSchedulable() {
8      SMRH.getClock().activate(
9          (MethodLogic)listWithConstraint.get(0));
10   }
11   public String getPolicyName() {return "FIFO";}
12   public boolean isFeasible() {return true;}
13   protected boolean removeFromFeasibility(Schedulable s) {
14     listWithConstraint.remove(s); return true;
15   }
16   public void run() {}
17 }
```

A declaração de classe apresentada acima representa uma implementação simples e mínima de um escalonador de tarefas. As tarefas são ordenadas por chegada (FIFO), independente da sua prioridade ou tempo restante para execução, e a execução das tarefas é repassada para o objeto CMC disponível.

3.2.3 Declaração de restrições temporais

Restrições temporais servem para definir um tipo de comportamento que os métodos do Modelo RTR devem seguir. Pode-se definir uma nova restrição temporal utilizando a sintaxe mostrada no Código 2.

Com esta nova estrutura é possível definir novas restrições temporais que devam ser respeitadas por alguma parte do código. Por padrão, as novas restrições são disparadas por eventos (***eventtrigger***), quando não informado o tipo da nova restrição temporal.

As restrições declaradas como ***timetrigger*** fazem com que o método que possua esta restrição temporal seja chamado logo após a instanciação de um objeto que declare este método. No caso do tipo da restrição ser ***eventtrigger***, não será

```
<temporal_constraint> ::= "rttype" <identifier> "="
                          [ <temporal_constraint_type> ]
                          <formal_parameters> ";"

<temporal_constraint_type> ::= ["eventtrigger" | "timetrigger"]
```

Código 2: Sintaxe da declaração de restrições temporais em Java/RTR.

gerado código para fazer a chamada do método automaticamente, ficando a cargo do desenvolvedor esta tarefa.

As restrições básicas já implementadas, `Periodic`, `Aperiodic` e `Sporadic`, são todas ***`eventtrigger`***, conforme mostrado nas suas declarações formais abaixo.

Segmento de Código 3.5: Exemplo de declaração de restrições em Java/RTR

```
10 rttype Aperiodic = eventtrigger(long custo, long deadline);
11 rttype Periodic = eventtrigger(long start,
12                   long period, long custo, long deadline);
13 rttype Sporadic = eventtrigger(long minInterArrival,
14                   long custo, long deadline);
```

Detalhes sobre a implementação das restrições temporais estão na Seção ‘Implementação das restrições temporais definidas’ (p.58).

3.2.4 Declaração de métodos

Uma das partes mais importantes do Modelo é a definição de métodos que possuam comportamento temporalmente restrito. Métodos com estas características são declarados em Java/RTR conforme apresentado na sintaxe apresentada nesta seção.

Pode-se perceber que a diferença entre a sintaxe de Java/RTR e de JAVA™ é apenas a inclusão do não-terminal opcional <`rt_method_declarator`>. Este não-terminal é responsável por:

1. conter o nome de uma restrição temporal a ser seguida;

```
<method_declaration> ::= <modifiers> <return_type> <identifier>
                         <formal_parameters> { "[" "]" }
                         [ "throws" <name_list> ]
                         [ <rt_method_declarator> ]
                         (<block> | ";")

<rt_method_declarator> ::= "rtconstraint" <identifier>
                           <default_arguments>
                           [ "rtthrow" <identifier> ]
                           [ "rtcategory" <identifier> ]
```

Código 3: Sintaxe da declaração de métodos em Java/RTR.

2. definir os valores padrão dos parâmetros da mesma. Estes valores serão utilizados quando os parâmetros para a chamada do método estivem fora dos limites permitidos (menores que zero, por exemplo);

3. opcionalmente, definir o nome de um método para fazer a correção temporal, ou seja, um método alternativo a ser executado em caso de falha temporal do método sendo definido;

4. opcionalmente, definir o nome de uma categoria a que este método pertence.

Um detalhe da implementação é de que os métodos de correção temporal não podem receber parâmetros, pois sua chamada será feita automaticamente pelo Meta-Objeto Gerenciador associado ao objeto. Também devem possuir o mesmo tipo de retorno que o método que se propuseram a corrigir, pois seu resultado será retornado no lugar daquele.

3.2.5 Chamada de métodos

No momento da declaração da restrição temporal são definidos os parâmetros formais (com seus tipos e quantidades) que a mesma precisa receber para ser executada. Quando um método é declarado e uma restrição temporal é associada ao mesmo,

são passados valores padrão para os parâmetros declarados formalmente. Isto serve para facilitar a programação, pois em alguns casos a chamada de um mesmo método terá sempre os mesmos valores como parâmetros da restrição temporal.

```
<arguments> ::= "(" [ <argument_list> ] ")"
                [ "#" "(" [ <argument_list> ] ")" ]
```

Código 4: Sintaxe dos argumentos de uma chamada de método em Java/RTR.

Entretanto, para se incrementar a flexibilidade, também é possível passar os parâmetros da restrição temporal no momento da chamada do método. Isto pode ser feito com a sintaxe dos argumentos modificada para aceitar o conjunto de argumentos da restrição temporal. Este conjunto extra de argumentos é obrigatório para chamadas de métodos temporalmente restritos, e proibido para métodos sem restrição temporal.

Basicamente, a sintaxe consiste em um conjunto de argumentos (um conjunto vazio é válido no caso de um método sem parâmetros) para o método, seguido do caracter '#' e outro conjunto de parâmetros (também podendo ser vazio), desta vez relativo a restrição temporal relacionada ao método.

O caracter '#' serve apenas para marcar o início de outro conjunto de argumentos, facilitando a visualização desta mudança. A idéia de não colocar um caracter separador tornou a sintaxe mais difícil de ser lida, e o uso da vírgula entre os dois conjuntos (tentativa seguinte) foi descartado pois criaria ambiguidade na sintaxe quando a chamada de um método, seguido de uma expressão entre parênteses, serviam como parâmetros de outro método.

O Modelo RTR também define que a chamada de métodos temporalmente restritos pode ser marcada para que o método execute de forma assíncrona, i.e., paralelamente com a execução do código que efetuou a própria chamada do método.

3.2.5.1 Exemplos

Segmento de Código 3.6: Chamada de métodos temporalmente restritos

```
10  // chamadas simples
```

```
11  resultado = objeto.metodo(30);          // sem restricao
12  objeto2.set(novoValor)#(300, 20, 100); // com restricao
13
14  // chamada assincrona
15  #objeto.executa()#(100, 750);  // SEMPRE com restricao
```

Como visto neste exemplo, também é utilizado o caracter ‘#’ para marcar a chamada de um método como sendo assíncrona. É importante ressaltar que apenas métodos temporalmente restritos podem ser executados desta forma, e que os mesmos não poderão ter um valor de retorno, pois o mesmo será perdido quando o método terminar sua execução. Se um método declarado com valor de retorno diferente de **`void`** for chamdo assincronamente, uma exceção é lançada em tempo de execução.

Capítulo 4

Tradutor RTR2Java

4.1 Introdução

Neste capítulo serão descritos a criação do tradutor RTR2Java e a reestruturação do *framework* provido para a linguagem Java/RTR.

4.2 Problema

A linguagem Java/RTR não é reconhecida por nenhum processador ou microcontrolador existente, e sua utilização, como também a de outras linguagens de alto nível, está condicionada a um processo de tradução. Partindo de Java/RTR até uma linguagem de máquina, diversos caminhos podem ser seguidos. Entre as possibilidades temos as seguintes, sendo as vantagens e desvantagens de cada uma brevemente descritas abaixo:

Compilação para código nativo: compilar o código Java/RTR diretamente para a linguagem nativa do processador escolhido.

- Desvantagens
  - perda de portabilidade
  - necessidade de se implementar compiladores específicos para cada equipamento desejado
- Vantagens

+ desempenho otimizado para a arquitetura utilizada
+ tamanho do código gerado tende a ser menor

Compilar para linguagem intermediária: compilar o código Java/RTR para uma linguagem intermediária, que possa ser interpretada na arquitetura escolhida, ou que possua um compilador JIT.

- Desvantagens
  - necessidade de um interpretador para cada arquitetura desejada
  - código gerado não pode depender de determinada arquitetura
  - necessidade de criar uma linguagem intermediária
- Vantagens
  + gerar código intermediário pode ser mais fácil que código de máquina
  + ganho de portabilidade, bastando reimplementar o interpretador

Traduzir para linguagem de alto nível: traduzir o código Java/RTR para outra linguagem de alto nível, utilizando as ferramentas já existentes desta linguagem para gerar o código específico da arquitetura desejada.

- Desvantagens
  - tempo de tradução tende a ser maior por utilizar estruturas mais complexas
  - código gerado não pode ser otimizado para determinada arquitetura
- Vantagens
  + código gerado independe da arquitetura, aumentando a portabilidade
  + aproveitamento das ferramentas que suportam a linguagem alvo
  + facilidade na implementação pois apenas o tradutor é necessário

O método escolhido para resolver este problema foi traduzir o código gerado pelo desenvolvedor para código JAVA™, utilizando as ferramentas já disponíveis para JAVA™ como suporte (compiladores e máquinas virtuais). As questões temporais

descritas em Java/RTR são traduzidas através da utilização da API disponibilizada pela RTSJ.

O trabalho do tradutor e sua implementação são descritos a seguir.

4.3 Tradutor RTR2Java

4.3.1 Conceitos

Conceitualmente, um compilador é um programa que lê um programa escrito numa linguagem — a linguagem fonte — e o traduz num programa equivalente em outra linguagem — a linguagem alvo. Como importante parte desse processo de tradução, o compilador relata a seu usuário a presença de erros no programa fonte (AHO 86).

A diferença entre um compilador e um tradutor está na linguagem alvo. Se esta for uma linguagem de máquina, então os programas que a geram são compiladores. Se for uma linguagem intermediária ou de alto nível, então quem a gera é um tradutor.

Neste trabalho, a linguagem fonte é Java/RTR, e a linguagem alvo é JAVA™, sendo que o processo de tradução irá utilizar a API descrita na especificação RTSJ e sua implementação de referência. Com o apresentado acima define-se que este trabalho apresenta um tradutor, e não um compilador.

4.3.2 Histórico

Em 2001 foi desenvolvido um pré-processador (KUE 01), que processava o código escrito em Java/RTR (uma versão anterior da apresentada neste trabalho, não definida formalmente) e gerava código JAVA™ genérico, para que o mesmo pudesse ser compilado e executado nas diversas plataformas que possuem suporte a esta linguagem.

Por questões de tempo e de problemas na gramática de JAVA™ utilizada como base na implementação do pré-processador, algumas restrições quanto às construções aceitas pela linguagem tiveram que ser impostas, reduzindo a possibilidade de utilização deste pré-processador.

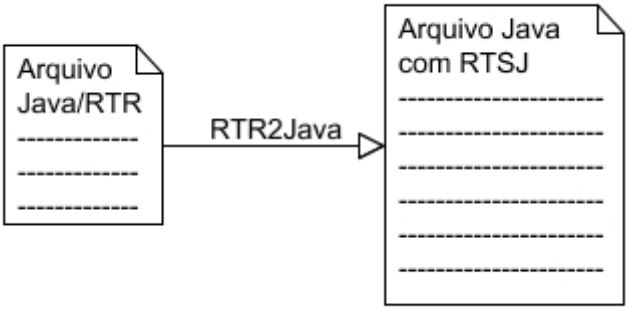

Figura 4.1: Esquema do Tradutor RTR2Java. Os arquivos de entrada são escritos em Java/RTR, e na saída são gerados arquivos JAVA™ que utilizam as funcionalidades da RTSJ. Geralmente o arquivo de saída será maior que a entrada, pois o tradutor gera código para converter as funcionalidades do Modelo RTR para a especificação de tempo real de JAVA™.

O tradutor desenvolvido neste trabalho é uma versão melhorada e atualizada deste pré-processador, além de trabalhar com outra versão de Java/RTR.

4.3.3 Ponto de partida da implementação

Partindo-se da gramática completa e validada da linguagem JAVA™ e utilizando-se a ferramenta JavaCC (JAV 04), um analisador léxico e um sintático para a linguagem Java/RTR foram gerados.

A gramática inicial para JAVA™ foi escrita por Sriram Sankar (Sun Microsystems Inc., em Maio de 1997), para a versão 1.1, foi atualizada por David Williams para JAVA™ 1.2 (em Julho de 1999), foi novamente atualizada por Andrea Gini para a versão 1.4 (em Fevereiro de 2002), e depois modificada ainda por Marco Savard (em Março de 2002) e por Andrea Gini (em Maio de 2002) para corrigir pequenos problemas para as mudanças acrescidas nas últimas versões.

Depois disto, as alterações necessárias para a compilação de código Java/RTR foram incluídas, assim como código extra nas produções da gramática para a montagem de uma Árvore Sintática Abstrata (*Abstract Syntax Tree*). Durante a montagem da árvore algumas verificações sintáticas mais simples são realizadas, de forma a evitar

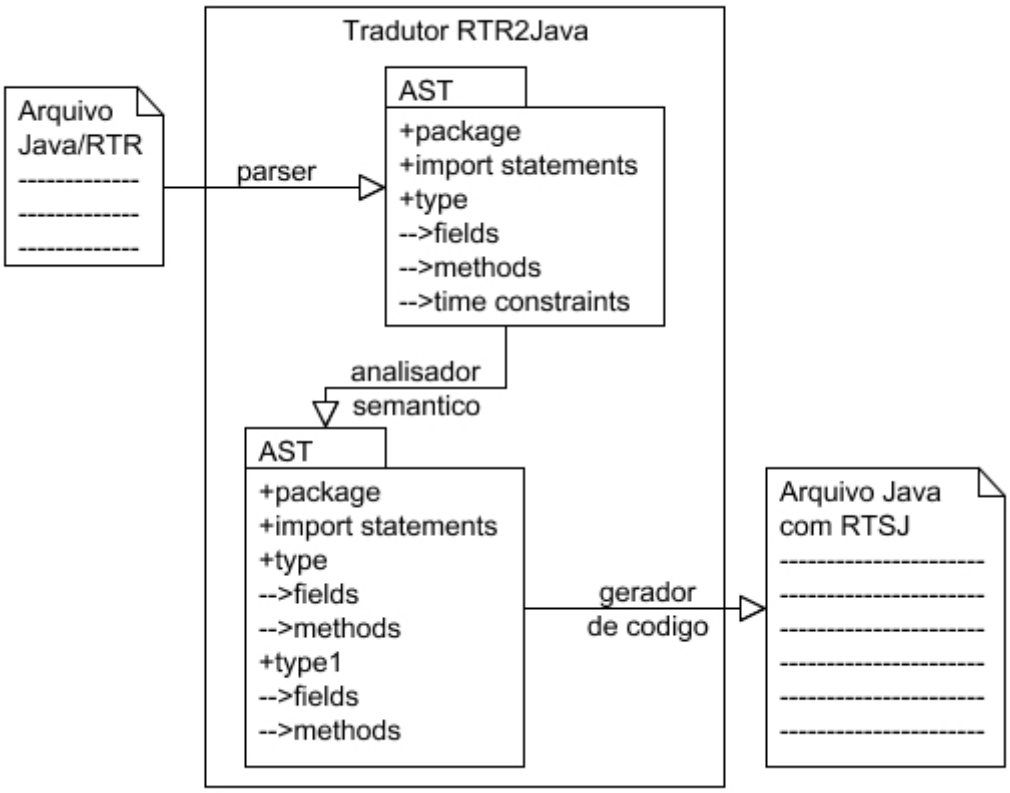

Figura 4.2: Esquema detalhado do Tradutor RTR2Java. Os arquivos de entrada passam por três passos: o parser, que constrói a AST a partir do arquivo de entrada; o analisador semântico, que analisa e traduz os nodos Java/RTR da AST para nodos JAVA™; e o gerador de código, que gera o código JAVA™ referente a cada nodo da AST.

a análise de uma estrutura inconsistente ou incorreta. Este trabalho adicional, realizado com a ferramenta *JJTree* (JJT 04), se mostrou de grande valia no restante do desenvolvimento, pois permitiu a utilização do padrão de projeto (GAM 95) Visitante (*Visitor*) para se percorrer a árvore criada.

4.3.4 Principais desafios

Utilizar o código da RTSJ introduz novas formas de representar as necessidades de um STR. Para que se faça uso das vantagens providas, o código escrito deve utilizar explicitamente suas características, como, por exemplo, as *threads* de tempo real e as novas áreas de memória.

Um dos principais desafios na implementação do tradutor RTR2Java é utilizar as estruturas disponibilizadas pelo RTSJ para representar o código Java/RTR que o desenvolvedor projetou e programou.

Dois pontos merecem destaque nesta tarefa.

4.3.4.1 Código temporalmente restrito

O Modelo RTR permite que métodos possuam restrições temporais associadas, para controlar seu comportamento durante cada execução do corpo do mesmo. Isto significa que uma mesma classe pode declarar quantos métodos forem necessários, cada um com uma restrição temporal diferente, e que fica sob responsabilidade do seu meta-objeto Gerenciador fazer este controle.

Já na especificação RTSJ, a menor unidade de código que pode conter comportamento temporalmente restrito é uma classe JAVA™, sendo o menor exemplo uma classe anônima local que implementa a interface `Runnable` e implementa apenas o método `run`.

Segmento de Código 4.1: Menor porção de código RTSJ temporalmente restrito

```
10  Runnable r = new Runnable() {
11    public void run() {
12      System.out.println("Ola");
13    }
14  };
```

Disto já pode-se perceber que não é possível uma tradução simples, pelos seguintes motivos:

- métodos em Java/RTR podem retornar o resultado de uma computação, enquanto o método `run` da interface `Runnable` não permite o retorno de resultados;
- métodos em Java/RTR podem levantar exceções se alguma coisa sair errada, mas objetos `Runnable` ou `RealtimeThread` devem tratar as exceções que ocorrem no seu método `run` sob pena de terem sua *thread* de execução finalizada;
- métodos em Java/RTR podem conter o modificador **`abstract`**, que não é passível de representação em classes RTSJ, nem faz sentido neste contexto.

O MOG responsável pelo controle temporal prevê, através de seu protocolo e da definição do Modelo, que a chamada de um método seja feita através do método `receiveRequest`, contendo os seguintes parâmetros necessários para sua execução:

- nome do método que deve ser executado;
- parâmetros para execução do método;
- referência a instância do OBTR na qual o método será aplicado;
- nome da restrição temporal associada;
- parâmetros da restrição temporal;
- nome do método que deve ser executado no caso de falha no comportamento temporal.

Acontece que passar todos estes parâmetros a cada chamada do método torna o código de díficil entendimento e manutenção, além de exigir a utilização do mecanismo de reflexão provido por JAVA™ para efetivar a execução do método no objeto passado. E ainda existe a questão das chamadas assíncronas de métodos.

Para tentar reduzir esta complexidade foi criada uma classe abstrata que, além de agrupar os dados necessários na chamada de um método restrito, também encapsula o resultado gerado pelo mesmo. Isto resolve o problema, mas joga a responsabilidade para o tradutor que terá que gerar implementações desta classe, chamada `MethodLogic`. Veja os detalhes da implementação desta classe abstrata na Seção ‘MethodLogic’ (p.67).

Segmento de Código 4.2: Método com restrição temporal em Java/RTR

```
10 public rtbc class Classe {
11   public int calc(int n) rtconstraint Aperiodic (10, 100) {
12     return n - 3;
13   }
14 }
```

A solução encontrada para resolver este problema foi transformar cada método que possua uma restrição temporal em uma subclasse de `MethodLogic`. Esta

subclasse conterá as informações necessárias para executar o método como definido no código Java/RTR. Informações como os parâmetros do método, a restrição temporal associada e o método de exceção temporal estarão disponíveis através desta subclasse, que será passada para o método `receiveRequest` do meta-objeto Gerenciador, pricipal consumidor destas informações.

O código mostrado na Listagem 4.3 é gerado pelo código da Listagem 4.2, contendo uma variável para armazenar o resultado, variáveis que armazenam os parâmetros da restrição temporal e uma referência para a instância na qual o método deve ser executado.

Segmento de Código 4.3: Classe gerada de um método com restrição temporal em Java/RTR

```
1  import javax.realtime.*;
2  import br.ufsc.inf.javartr.lib.MethodLogic;
3  import br.ufsc.inf.javartr.lib.SMRH;
4  public class MethodLogic_2e6f1204 extends MethodLogic {
5    private Classe instance = null;
6    public int result;
7    private int n;
8    private long __custo = 10;
9    private long __deadline = 100;
10   public MethodLogic_2e6f1204(Classe __i, int _n, long ___custo,
11     long ___deadline) {
12       super(); this.instance = __i; this.n = _n;
13       if (___custo >= 0) { this.__custo = ___custo; }
14       if (___deadline >= 0) { this.__deadline = ___deadline; }
15       this.aeh = null;
16   }
17   public void run() { instance.__meta.receiveRequest(this); }
18   public void runTemporalConstraint(MethodLogic logic) {
19     SMRH.getSched().schedule(logic);
20     instance.__meta.releaseActivationRequest(logic);
21     try { this.execute(); } finally {
22       instance.__meta.endOfExecution(logic);
```

```
23            SMRH.getSched().releaseNextRequest();
24          }
25    }
26    public void execute() {
27      this.result = instance.__calc(n);
28    }
29    public ReleaseParameters getReleaseParameters() {
30      if (relParam == null) {
31        setReleaseParameters(new AperiodicParameters(
32          new RelativeTime(__custo, 0),
33          new RelativeTime(__deadline, 0), null,
34            getRTThrow()));
35      }
36      return relParam;
37    }
38    public String toString() { return "Method Logic para calc"; }
39  }
```

Para que cada chamada do método não precise ser traduzida para a instanciação de um objeto, o corpo do método é substituído pela criação e execução da subclasse criada, retornando o resultado da execução se o mesmo exigir isto. O corpo real do método é movido para outro método, este sim chamado pela execução da subclasse, efetivamente realizando o trabalho a que o mesmo se propõe.

Na Listagem 4.3 também é possível visualizar que a subclasse possui atributos responsáveis por manter os valores dos parâmetros das restrições temporais (neste caso, `__custo` e `__deadline`), inicializados com os valores padrão para cada um (10 e 100, respectivamente, conforme declarado na Listagem 4.2). Estes atributos são atualizados no corpo do construtor da subclasse, quando é feita uma verificação com os valores passados como parâmetros. Partindo da premissa que valores negativos são valores inválidos (um custo e uma *deadline* não podem ser negativos, assim como também não podem um período e um *delay* inicial), um teste é feito antes da atualização. Este mecanismo, além de evitar que o sistema aceite valores inválidos, facilita a chamada de

métodos com restrição temporal na medida que não exige do desenvolvedor que se preocupe com todos os valores da restrição temporal. Se os valores padrão definidos na declaração do método são aceitáveis, então pode-se passar qualquer valor negativo que o mesmo será substituído automaticamente na construção da subclasse. A opção de gerar exceções no caso dos parâmetros passados serem inválidos possui a restrição de dificultar a programação, além de exigir um mecanismo adicional para que o desenvolvedor possa informar ao tradutor que os valores padrão devem ser utilizados.

Neste primeiro momento, o nome das subclasses de `MethodLogic` é formado unindo-se o prefixo "MethodLogic_" com o resultado de uma função *hash* que recebe como parâmetros as informações do método que está sendo traduzido para uma classe. Não é possível utilizar apenas o nome do método, pois métodos com nomes iguais mas com assinaturas diferentes devem gerar subclasses diferentes. Mesmo gerando um nome não pronunciável (como "MethodLogic_2e6f1204", por exemplo), esta solução tem a vantagem de ser única para cada assinatura de método, evitando que métodos com assinaturas parecidas gerem classes com o mesmo nome. Em defesa desta solução também conta que estas subclasses serão utilizadas apenas pelo Tradutor, não havendo necessidade de seu conhecimento por parte do desenvolvedor.

4.3.4.2 Restrições temporais

O segundo ponto de maior complexidade no processo de tradução é a representação de novas restrições temporais.

A especificação RTSJ provê três restrições temporais para as *threads* tempo real, através das subclasses de `ReleaseParameters`: periódica, com a implementação `PeriodicParameters`; aperiódica, com a classe `AperiodicParameters`; e esporádica, representada por `SporadicParameters`. Estas classes servem, na verdade, apenas para conter e manter informações como período, *deadline*, custo e tratadores de perda de *deadline*. A lógica que utiliza estas informações para controle das restrições está centralizada na classe `RealtimeThread`. Veja na Figura 4.3 o diagrama das classes envolvidas.

Assim, a classe `RealtimeThread` (e a implementação de referência como um todo) conheçe apenas estas restrições temporais, tratando qualquer nova imple-

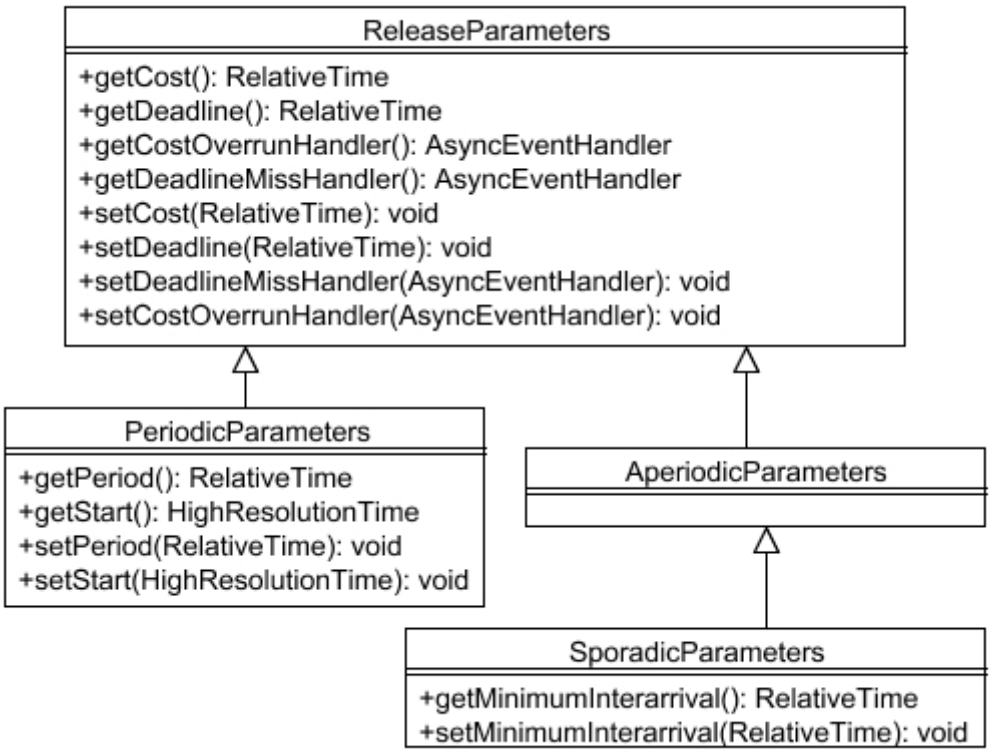

Figura 4.3: Diagrama das classes relativas a restrições temporais do RTSJ.

mentação como se fosse aperiódica (por ser a implementação mais simples).

Se não fosse necessário permitir a criação de novas restrições temporais, a saída mais elegante seria mapear diretamente as restrições básicas previstas no Modelo RTR para suas respectivas classes na RTSJ. Mas impor esta restrição resultaria em uma grande perda de flexibilidade, além de não suportar por completo o Modelo RTR.

A solução encontrada foi definir e prover o máximo possível de restrições temporais, reduzindo a necessidade da criação de novas restrições. E estas novas restrições são mapeadas em subclasses de `ReleaseParameters`, de forma a conter a lógica da restrição internamente. Esta solução evita a reimplementação da classe `RealtimeThread` para suportar novas restrições, mas ainda exige conhecimento da especificação RTSJ, já que as restrições serão mapeadas para as subclasses.

De qualquer forma, novas restrições temporais definidas em um objeto RTBC necessitam implementar uma classe que estenda `ReleaseParameters`, ou uma das suas subclasses, e que o nome da classe seja o nome da restrição com o sufixo “Pa-

rameters". Esta implementação irá manter a lógica da nova restrição temporal, e exigirá conhecimento da RTSJ para ser concluída.

4.3.5 Aspectos semânticos

A verificação semântica é feita por um Analisador Semântico, especificamente implementado para verificar estes quesitos na árvore sintática gerada pelo *parser*.

É neste passo que as estruturas do Modelo RTR são traduzidas para estruturas de JAVA™, e a utilização do RTSJ tem início. Nas próximas seções serão destacadas e detalhadas as principais atividades realizadas pelo analisador semântico.

4.3.5.1 Método `main` das classes do tipo RTBC

Quando uma classe do tipo Real-Time Base Class implementa um método com a assinatura válida para iniciar uma aplicação JAVA™ (**`public static void`** `main(String[])`), este método é considerado como sendo o início do código que o desenvolvedor deseja executar, tanto para a JVM como para o tradutor.

No caso específico deste trabalho, existem algumas atividades que devem ser realizadas antes do início da aplicação. Estas atividades incluem criar objetos necessários ao controle da aplicação, além de modificações em algumas declarações. Os detalhes são explicados a seguir.

Criação dos meta-objetos Como o Modelo RTR possui meta-objetos responsáveis pelo controle do sistema, é neste método que os mesmos devem ser declarados, instanciados e inicializados. Assim, deste ponto em diante os mesmos tornam-se disponíveis para os meta-objetos criados indiretamente pelo usuário.

Os passos identificados acima (declaração, instanciação e inicialização dos meta-objetos) são de responsabilidade do tradutor, que os esconde do usuário através da camada de abstração proposta pelo Modelo.

Segmento de Código 4.4: Método main de objetos OBTR traduzido

```
34  public static void main(String[] args) throws
35      IllegalAccessException, InstantiationException {
```

```
36      try {
37        __initRTR();
38        // application code
39      } finally {
40        __terminateRTR();
41      }
42    }
```

O código da Listagem 4.4 mostra como fica o corpo do método `main` após a inserção de código para fazer a instanciação e interrupção dos objetos necessários. Como forma de garantir que o código de finalização será executado, o método de inicialização e todo o código escrito pelo programador são executados em um bloco **`try`**, fazendo com que o método `__terminateRTR()` sempre execute, independente de alguma exceção ter sido lançada.

Segmento de Código 4.5: Referências às threads dos meta-objetos

```
1   private static RealtimeThread rttClock = null;
2   private static RealtimeThread rttSched = null;
```

Como descrito no Modelo RTR, os meta-objetos Escalonador e Relógio devem ser *threads* no sistema, ficando disponíveis por todo o tempo de execução. Utilizando as funcionalidades do RTSJ e por serem cruciais no processamento dos requisitos temporais, ambos objetos serão mapeados para *threads* de tempo real. O primeiro passo é manter uma referência das mesmas para que se possa fazer sua interrupção no final da aplicação.

Segmento de Código 4.6: Instanciação dos meta-objetos do Modelo RTR

```
4   private static void __initRTR() throws IllegalAccessException,
5             InstantiationException {
6     StandardCMC __clock = ((StandardCMC) ImmortalMemory
7         .instance().newInstance(StandardCMC.class));
8     SMRH.setClock(__clock);
9     RMScheduler __scheduler = ((RMScheduler) ImmortalMemory
10        .instance().newInstance(RMScheduler.class));
```

```
11      SMRH.setSched(__scheduler);
12
13      SchedulingParameters schedParam = new PriorityParameters(
14              PriorityScheduler.MAX_PRIORITY - 5);
15      ReleaseParameters relParam = new AperiodicParameters(null,
16              null, null, null);
17
18      rttClock = new RealtimeThread(schedParam, relParam, null,
19              null, null, __clock);
20      rttClock.setDaemon(true);
21      rttClock.start();
22
23      rttSched = new RealtimeThread(schedParam, relParam, null,
24              null, null, __scheduler);
25      rttSched.setDaemon(true);
26      rttSched.start();
27  }
```

Aqui é apresentado o corpo do método responsável pela criação dos meta-objetos. Um detalhe importante que pode ser notado no código acima é a criação de objetos utilizando-se o modelo de memória disponibilizado pelo RTSJ. Também é importante frizar que a primeira ação após a criação dos meta-objetos é seu registro junto à classe SMRH (apresentada na Seção ‘SMRH’ (p.64)), para que os meta-objetos Gerenciadores tenham acesso à instância criada.

Inicialmente, as *threads* criadas possuem cinco pontos a menos da prioridade máxima permitida pela RTSJ, sendo este valor uma estimativa inicial, precisando ainda ser testado e validado.

Após as instanciações, são criadas duas *threads* tempo real para executar a lógica dos meta-objetos criados, e as referências a estas *threads* são mantidas na classe que contém o código, como já mostrado. As *threads* são marcadas para executarem indefinidamente, através do método `setDaemon(true)`, e são então iniciadas.

Segmento de Código 4.7: Interrupção dos meta-objetos do Modelo RTR

```
29   private static void __terminateRTR() {
30     if (rttSched != null) { rttSched.interrupt(); }
31     if (rttClock != null) { rttClock.interrupt(); }
32   }
```

Para que as *threads* tempo real não continuem executando após o término da aplicação, as mesmas devem ser avisadas que podem parar. Isto é feito através do mecanismo de Transferência Assíncrona de Controle, apresentado na Seção ‘Transferência assíncrona de controle’ (p.21), chamando o método `interrupt()` nas referências mantidas quando da sua criação.

Com o explicado na Seção ‘Alocação de memória’ (p.24) e seguindo as recomendações de (DIB 02), decidiu-se que todos os objetos instanciados dentro do método `main` serão considerados objetos permanentes. É importante salientar que aplicativos tempo real costumam ter uma seção de inicialização (às vezes demorada, mas executada apenas uma vez), que antecede a execução da grande parte do código. Esta inicialização, em um programa JAVA™ ou Java/RTR pode muito bem ser o corpo do método principal.

Este código faz uso da hierarquia de classes disponibilizada junto com o tradutor, apresentada mais adiante, na Seção ‘Pacote br.ufsc.inf.javartr.lib’ (p.61).

Alteração na criação de novos objetos Assim como a criação dos meta-objetos no método `main` é feita de forma a utilizar as características dos modelos de memória disponibilizados pela especificação RTSJ, o mesmo deve ocorrer com os objetos criados pelo usuário. Deve ficar bem claro que, desta forma, todos os objetos explicitamente criados pelo usuário dentro do método principal serão considerados objetos permanentes, mesmo que necessários apenas para a inicialização do sistema, i.e., independente da sua semântica de criação ou objetivo de utilização.

Com isto em mente, o código apresentado na Listagem 4.8 é convertido para o código mostrado na Listagem 4.9.

Segmento de Código 4.8: Método main de um RTBC em Java/RTR

```
10 public static void main(String[] a) {
11     // classes simples
```

```
12      StringBuffer sb = new StringBuffer();
13      sb.append(2004);
14      System.out.println(sb.toString());
15
16      // arrays de objetos
17      String[] s = new String[4];
18
19      // array de tipos basicos
20      float[] f = new float[s.length];
21  }
```

Segmento de Código 4.9: Método main de um RTBC traduzido para Java com RTSJ

```
10  public static void main(String[] args) throws
11       IllegalAccessException, InstantiationException {
12    try {
13      __initRTR();
14      StringBuffer sb = ((StringBuffer) ImmortalMemory.instance()
15                      .newInstance(StringBuffer.class));
16      sb.append(2004);
17      System.out.println(sb.toString());
18
19      String[] s = ((String[]) ImmortalMemory.instance()
20                           .newArray(String.class,4));
21
22      float[] f = ((float[]) ImmortalMemory.instance()
23                           .newArray(Float.TYPE,s.length));
24    } finally {
25      __terminateRTR();
26    }
27  }
```

Perceba-se que a criação de séries[1] também é substituída, mas os ele-

[1] Arrays

mentos não são inicializados (portanto não são criados nem na memória permanente nem na memória *heap*!). A criação e inicialização de cada elemento fica a cargo do desenvolvedor. Séries de tipos básicos são permitidos, sendo que o tradutor utiliza o objeto `Class` obtido através da constante `TYPE`, que as classes JAVA™ que representam os tipos primitivos disponibilizam. A Tabela 4.1 mostra estes detalhes.

Tabela 4.1: Relação entre tipos primitivos e Classes em JAVA™

Tipo primitivo	**Classe Java**
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
void	java.lang.Void

O tipo **`void`** foi incluído apenas para conhecimento, já que não é possível declarar séries e variáveis com o mesmo.

4.3.5.2 Declarações

Declaração de novas restrições temporais O Modelo RTR define uma nova sintaxe para a declaração de novas restrições temporais. Mas estas declarações apenas são possíveis nas classes do tipo RTBC, pois são as únicas que podem conter métodos que utilizem este mecanismo de controle. Se uma restrição estiver sendo declarada fora de uma classe do tipo RTBC, uma exceção será gerada.

Como já explicado anteriormente, as restrições temporais são mapeadas diretamente para uma subclasse de `ReleaseParameters`, exigindo que uma implementação seja provida.

Declaração de métodos com restrição temporal A declaração de métodos que utilizem restrições temporais são possíveis apenas nas classes RTBC, pois serão as únicas a estarem sendo controladas pelos objetos gerenciadores. Métodos temporalmente restritos sendo declarados dentro de qualquer outro tipo de classe irão gerar uma exceção.

4.3.5.3 Implementação

Implementação do protocolo do Modelo RTR Com a informação do tipo de classe RTR que o usuário está declarando é possível verificar se a mesma provê o protocolo definido e exigido pelo Modelo RTR. Dos tipos especiais, o Gerente, o Escalonador e o Relógio possuem interfaces (`ProtocolMMC`, `ProtocolSMC` e `ProtocolCMC`, respectivamente) que definem um conjunto mínimo de funcionalidades que suas implementações precisam prover.

O analisador semântico realiza testes para verificar se a classe sendo declarada implementa diretamente a interface correspondente (acima mencionadas), ou se estende uma classe que implemente a interface do protocolo relativa ao seu tipo. No caso da classe especial não estender nenhuma classe nem implementar interfaces, a classe especial padrão será adicionada automaticamente como super-classe. Este comportamento foi escolhido ao invés de se gerar uma exceção pelo motivo de ser mais fácil para o desenvolvedor, evitando que o mesmo se preocupe diretamente com a hierarquia das classes implementadas.

Segmento de Código 4.10: Meta Classe sem hierarquia

```
1 public mmc class MetaClasse {
2     // codigo da classe aqui
3 }
```

A declaração acima é traduzida para a declaração abaixo destacada, e da mesma forma acontece com classes dos tipos ***smc*** e ***cmc***, obviamente estendendo suas respectivas classes abstratas.

Segmento de Código 4.11: Meta Classe com hierarquia

```
1 public class MetaClasse extends AbstractMMC {
2     // codigo da classe aqui
```

```
3 }
```

As classes do tipo RTBC não precisam implementar nenhum protocolo.

Implementação das restrições temporais definidas Outra exigência do Modelo RTR é de que as restrições temporais declaradas nas classes RTBC devem possuir uma implementação na classe MMC. Isto significa que para cada nova restrição temporal declarada em uma classe RTBC deve haver um método com o mesmo nome e com parâmetros compatíveis em número e tipo na classe MMC correspondente. O corpo deste método será a implementação da restrição propriamente dita.

Mas como explicado anteriormente, para unir Java/RTR com RTSJ é necessário que as restrições temporais estendam a classe `ReleaseParameters`, para que possam ser utilizadas na criação de novas *threads* tempo real.

No caso de não haver uma classe que implemente a restrição temporal declarada, uma exceção será gerada pelo analisador semântico.

Restrições nas subclasses do Escalonador e do Relógio Novamente, não se deve misturar os objetos do nível meta (que controlam a passagem do tempo e a aplicação em geral) com o código principal da aplicação (que implementa a regra do negócio).

Mesmo sendo sintaticamente permitido pela gramática utilizada, métodos das classes MMC, SMC e CMC não podem conter restrições temporais associadas, pois, obviamente, não haveria quem controlasse sua execução.

Resumindo, a implementação de meta-objetos pode conter apenas código JAVA™ válido. A mesma pode fazer referência a tipos definidos no *framework* disponibilizado, mas não pode utilizar estruturas de Java/RTR.

4.3.5.4 Verificações semânticas de JAVA™

Embora a análise semântica das estruturas Java/RTR seja sempre executada, não significa que a árvore neste ponto estará semanticamente correta. Isto porque são muitas as verificações a serem feitas nas construções JAVA™ para se fazer qualquer garantia (verificações de tipos e resolução de nomes estão entre as tarefas mais pesadas e

complicadas a serem realizadas). Por questões de tempo de implementação, apenas algumas verificações são realizadas, exigindo consciência do desenvolvedor de que o código Java/RTR deverá estar semanticamente correto para que o código gerado também esteja.

Obviamente, o código JAVA™ gerado somente será compilado corretamente e poderá ser executado se o mesmo estiver semanticamente validado. Isto não significa que o código gerado pelo tradutor RTR2Java deverá ser alterado pelo desenvolvedor (mesmo isto podendo ser feito), mas sim que o código Java/RTR deverá ser revisado de forma a confirmar sua validade semântica.

Uma solução alternativa seria fazer a tradução das mensagens de erro geradas pelo compilador JAVA™ para referenciar o código Java/RTR escrito pelo programador, e não o código JAVA™ analisado. Mas isto acarreta na sobrecarga do tradutor, pois seria necessário manter referência do código Java/RTR original com o código JAVA™ gerado, para que se tivesse informações sobre o número da linha e da coluna aonde o erro foi encontrado, por exemplo.

4.3.6 Geração de código

Para que o código resultante pudesse ser utilizado, um gerador de código foi desenvolvido. Ele trabalha visitando todos os nós da árvore sintática, escrevendo código JAVA™ que reproduza estes nós. No caso de visitar um nó que representa uma estrutura Java/RTR, uma exceção é gerada, já que a mesma já deveria ter sido traduzida. Obviamente, caso isto acontecesse, seria uma falha na implementação do Analisador Semântico.

Neste ponto, todas as estruturas léxicas e sintáticas adicionadas por exigência do Modelo RTR já foram substituídas pelo analisador semântico, resultando em uma árvore sintática com nós que contêm apenas código JAVA™, evidentemente gerando código totalmente compatível com a especificação desta.

4.3.7 Testes da implementação

Uma grande suíte de testes foi escrita, utilizando-se a ferramenta JUnit (JUN 04), de forma a certificar-se de que as alterações feitas na gramática inicial não se-

riam danosas às estruturas JAVA™ aceitas normalmente. Para garantir isto, todas as classes JAVA™ que compõem a biblioteca que acompanha a máquina virtual na versão 1.4 (mais de 3800 classes) foram reprocessadas de maneira a verificar sua validade quando se utiliza o parser de Java/RTR. Note-se que a idéia da nova linguagem é apenas acrescentar novas estruturas (todas opcionais), mantendo-se a validade dos arquivos JAVA™ criados anteriormente a este trabalho, de forma a permitir que os mesmos possam vir a ser utilizados em sistemas com restrições temporais.

Para as estruturas criadas para representar a AST e para os analisadores desenvolvidos, mais de 950 unidades de testes foram escritas. Os testes objetivam exercitar o código, aumentando a confiabilidade entre o mesmo e o desenvolvedor, que pode verificar a corretude da sua implementação tão logo o teste esteja implementado. Mesmo não fazendo garantias, este método permite que alterações feitas no código possam ser rapidamente testadas quando a suíte de testes é executada freqüentemente. Resumidamente, cada estrutura da AST possui uma classe que contém os testes da implementação. Esta estratégia de desenvolvimento é conhecida por Desenvolvimento Dirigido por Testes (*Test Driven Development*) (BEC 02), e é uma das bases para a Programação Extrema (*Extreme Programming*) (WEL 04).

4.4 Framework Java/RTR

4.4.1 Conceito

Em desenvolvimento de software, o termo *framework* significa uma estrutura de suporte sobre a qual outros projetos podem ser organizados e desenvolvidos. Isto é feito particionando-se o projeto em interface e classes abstratas, e definindo suas responsabilidades e colaborações. O desenvolvedor especializa o *framework* para uma aplicação particular estendendo e compondo instâncias das classes existentes.

No caso específico deste trabalho, o principal item do *framework* é a biblioteca de classes, responsável pela definição da estruturas de suporte e por prover implementações básicas das classes necessárias.

4.4.2 Framework básico

O pré-processador criado por (KUE 01) teria sua utilização complicada e elevaria o nível de conhecimento técnico necessário para sua utilização. Para facilitar as coisas, um *framework* básico foi desenvolvido em (TRE 02), de forma a tornar a escrita de aplicações com restrições temporais mais simples.

A utilidade do *framework*, apesar da mudança de implementação de um pré-processador para um tradutor e da utilização da RTSJ, ainda é concreta. Assim como a maior parte das linguagens de programação, Java/RTR também possui uma hierarquia básica de classes e interfaces para disciplinar o desenvolvimento de aplicações que a utilizem.

Este *framework* serve, além de uma biblioteca com implementações básicas para os protocolos especificados pelo Modelo RTR, como uma estrutura padrão para a construção de aplicações em Java/RTR. Como visto na Seção ‘Modelo RTR’ (p.9), os conceitos do Modelo RTR precisam ser assimilados para que seja possível o desenvolvimento de aplicações nesta nova linguagem.

Para alcançar este objetivo, classes básicas, contendo as definições das restrições temporais mais comuns, foram implementadas, de forma a exigir um mínimo esforço por parte do desenvolvedor para a criação de aplicativos com restrições temporais na plataforma JAVA™.

4.4.3 Pacote br.ufsc.inf.javartr.lib

Este pacote contém as interfaces e classes básicas e de apoio para um sistema Java/RTR. Cada conjunto de classes está separado pelo seu tipo.

4.4.3.1 Relógio – Clock Meta-Class

São as classes e interfaces responsáveis pelo controle da passagem do tempo no sistema em execução. Sua hierarquia está mostrada na Figura 4.4.

Como a especificação RTSJ não exige um objeto relógio no sistema (apesar de definir uma classe que faça esta tarefa), este objeto acabou perdendo um pouco do seu propósito na implementação. Isto se deve ao fato de que o agendamento das novas

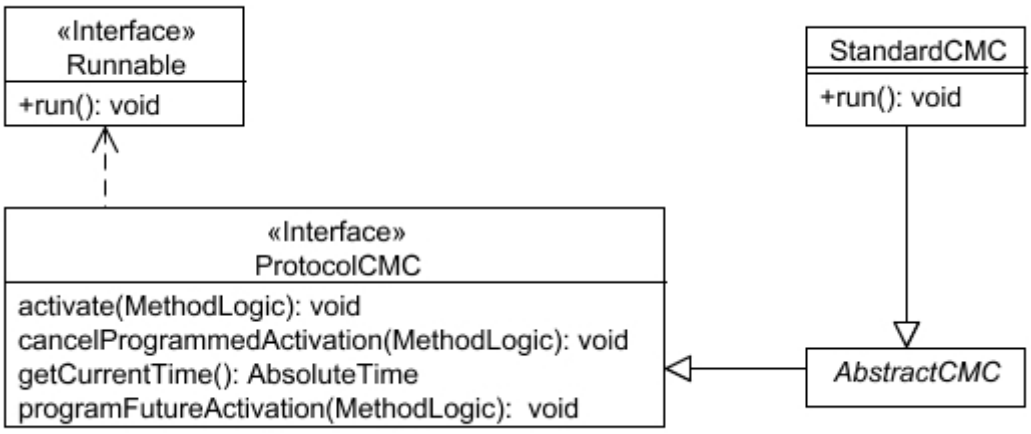

Figura 4.4: Diagrama das classes relativas ao Relógio do Modelo RTR.

execuções de uma tarefa periódica e seu eventual cancelamento são responsabilidades do escalonador e da própria *thread* sendo executada, praticamente tornando desnecessária a utilização de um relógio como o definido pelo Modelo RTR.

ProtocolCMC Interface que contém os métodos definidos pelo Modelo RTR. Operando sobre objetos do tipo `MethodLogic`, utiliza a classe `AbsoluteTime` da RTSJ para representar o tempo atual do sistema. É esta interface que deve ser implementada pelos desenvolvedores que queiram criar sua própria implementação a partir do zero. Esta interface estende a interface `Runnable` pois suas implementações servirão como lógica de uma *thread*.

AbstractCMC Classe abstrata que implementa os métodos necessários pela interface `ProtocolCMC`. Apenas o método `getCurrentTime` é implementado verdadeiramente, enquanto os outros não possuem código no seu corpo. Esta classe foi desenhada para ser estendida e facilitar a implementação do Protocolo por parte dos clientes.

StandardCMC Classe concreta, estende de `AbstractCMC`, mas apenas define o corpo do método `run()` como um laço infinito que pode ser interrompido por outra *thread*.

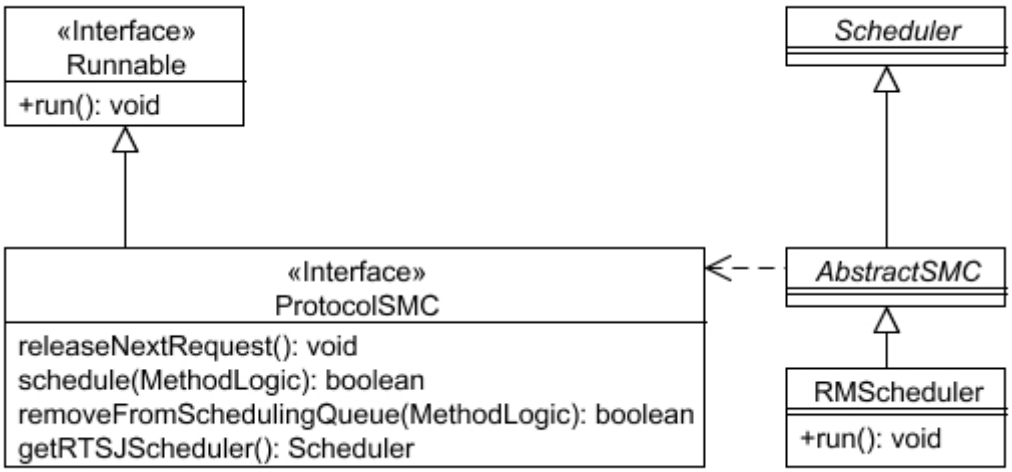

Figura 4.5: Diagrama das classes relativas ao Escalonador do Modelo RTR.

4.4.3.2 Escalonador – Scheduler Meta-Class

É o objeto responsável pelo escalonamento das tarefas no sistema em execução. São nestas classes e interfaces que o algoritmo de escalonamento deve ser implementado (neste ponto as tarefas se confundem com as *threads* que as executam). A hierarquia das classes e interfaces envolvidas é mostrada na Figura 4.5.

ProtocolSMC Interface que define os métodos que devem ser providos pelas implementações de um escalonador que suporta o Modelo RTR. Estende a interface `Runnable` para que suas implementações sirvam como lógica na *thread* do escalonador.

AbstractSMC É o ponto de contato entre o Modelo RTR e o escalonador definido pela RTSJ. Esta classe abstrata implementa a interface do Protocolo e estende a classe abstrata `Scheduler`. Possui campos e implementações de métodos que servem para as futuras implementações. É altamente recomendado que esta classe seja estendida ao invés de implementar diretamente a interface `ProtocolSMC`.

RMScheduler Classe concreta, implementa o algoritmo de escalonamento *Rate Monotonic*, onde as tarefas são ordenadas de acordo com a sua periodicidade, isto é, quanto menor o período, antes a tarefa será escalonada.

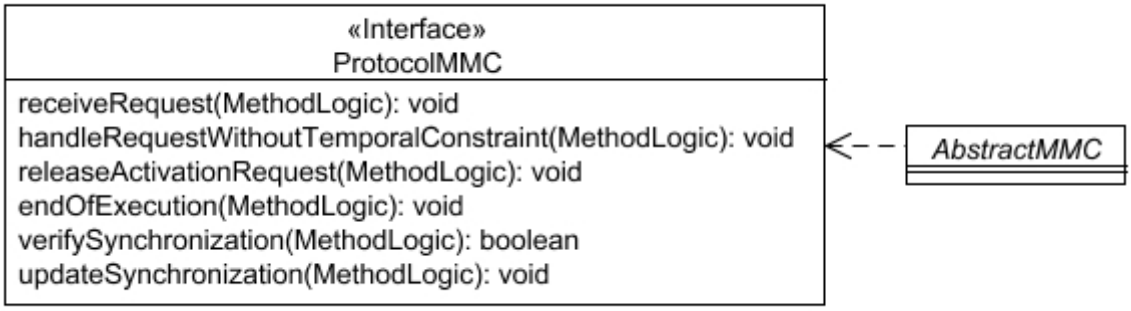

Figura 4.6: Diagrama das classes relativas ao Gerenciador do Modelo RTR.

4.4.3.3 Gerenciador – Manager Meta-Class

É o objeto responsável pelo controle das instâncias de objetos base no sistema em execução. A hierarquia é mostrada na Figura 4.6.

ProtocolMMC Interface que define os métodos previstos no Modelo RTR para objetos gerenciadores de objetos básicos.

AbstractMMC Classe abstrata que implementa a interface do protocolo e os métodos necessários para seu correto funcionamento. Não implementa as questões de concorrência interna, mas segue o fluxo definido pelo Modelo RTR. As meta-classes definidas pelos desenvolvedores devem estender esta classe abstrata, provavelmente sobrescrevendo algum método para realizar o controle necessário.

4.4.3.4 Outras classes

Além das classes e interfaces necessárias para a representação do Modelo RTR, outras foram necessárias para que o processo de tradução pudesse ser realizado. Estas classes tem suas estruturas discorridas a seguir.

SMRH O nome SMRH é apenas o acrônimo de “Static Meta Reference Holder”, fazendo referência à sua responsabilidade. Esta classe serve para manter referências aos meta-objetos Escalonador e Relógio, e para disponibilizá-las aos meta-objetos Gerencia-

dores que as precisarem (praticamente todos), de forma estática, garantindo a unicidade das instâncias.

Uma abordagem alternativa seria a utilização do padrão de projeto *Singleton* (GAM 95; FOX 01; GEA 03) na implementação das meta-classes Escalonador e Relógio, permitindo que as próprias classes controlassem suas referências. Este caso teria a desvantagem de não permitir que as meta-classes fossem estendidas (pois o construtor seria privado), e a utilização deste padrão não poder ser exigida via modelagem (o método estático para obter a referência não pode ser definido na interface nem sobrescrito nas subclasses).

Outra opção para tentar evitar o estilo utilizado na implementação da classe seria a passagem das referências dos meta-objetos via construtor para cada meta-objeto Gerenciador criado, mas isto também exigiria que as referências estivessem disponíveis no contexto da instanciação, o que seria uma ingrata tarefa ao tradutor e poderia alterar a semântica do código gerado.

Aceitando sua necessidade, a implementação desta classe possui duas variáveis estáticas privadas, para manter as referências às instancias do Escalonador e do Relógio. O construtor, desnecessário, é privado, para que a classe não possa ser instanciada de fora do seu próprio código.

Segmento de Código 4.12: Código da classe SMRH

```
1  package br.ufsc.inf.javartr.lib;
2  public class SMRH {
3    private static ProtocolCMC clock = null;
4    private static ProtocolSMC sched = null;
5    private SMRH() { super(); }
```

Para se obter e alterar a referência à variável do Relógio são disponibilizados os métodos `getClock()` e `setClock(ProtocolCMC)`, respectivamente, únicos meios de se ter acesso a mesma.

Segmento de Código 4.13: Código da classe SMRH (continuação)

```
6    public static ProtocolCMC getClock() {
7      synchronized (ProtocolCMC.class) {
```

```
8       return clock;
9     }
10  }
11  public static void setClock(ProtocolCMC c) {
12    if (c == null) {
13      return;
14    }
15    synchronized (ProtocolCMC.class) {
16      SMRH.clock = c;
17    }
18  }
```

O mesmo acontece para a variável do Escalonador, com os métodos `getSched()` e `setSched(ProtocolSMC)`. Todos os acessos são sincronizados, pois, invariavelmente, estaremos em um ambiente com múltiplas *threads*.

Segmento de Código 4.14: Código da classe SMRH (continuação)

```
19  public static ProtocolSMC getSched() {
20    synchronized (ProtocolSMC.class) {
21      return sched;
22    }
23  }
24  public static void setSched(ProtocolSMC s) {
25    if (s == null) {
26      return;
27    }
28    synchronized (ProtocolSMC.class) {
29      SMRH.sched = s;
30      Scheduler.setDefaultScheduler(s.getRTSJScheduler());
31    }
32  }
```

Tanto a atualização das referências como sua obtenção são feitos através de código gerado pelo tradutor, sem que o desenvolvedor precise fazer isto. Chamadas

escritas manualmente somente são necessárias na programação de novas restrições temporais.

Com a solução adotada, o próprio tradutor gera código para guardar a primeira referência de cada instância criada, fazendo isto uma única vez, na inicialização da aplicação, como descrito na Seção 'Criação dos meta-objetos' (p.51). A alteração destas referências é permitida (alteração do algoritmo de escalonamento é um dos motivos para isto), mas deve ser feita de forma controlada para manter a integridade do sistema.

MethodLogic As chamadas a métodos com restrição temporal carregam muito valor semântico agregado, sendo praticamente impossível representá-las utilizando as estruturas disponibilizadas na linguagem JAVA™. Somando-se a isto a questão da especificação RTSJ trabalhar apenas com objetos `Runnable` para representar um código com restrição temporal, foi criada a classe abstrata `MethodLogic`. Seu objetivo principal é manter informações sobre a chamada de um método temporalmente restrito.

Esta classe implementa a interface `Schedulable`, já que irá conter o código a ser executado em uma *thread* tempo real, provendo uma implementação básica para a maioria dos métodos desta interface, exceto `run`. Além disto, define mais dois métodos abstratos: `public void runTemporalConstraint(MethodLogic)`, que será o responsável por seguir a lógica de gerenciamento temporal descrita no Modelo RTR; e `public void execute() throws Exception`, que será o método executado contendo o conteúdo definido pelo programador. Outro método foi definido, mas já com uma implementação básica, chamado `public AsyncEventHandler getRTThrow()`, responsável por manter uma referência ao objeto que será chamado no caso de expirar o tempo de execução.

Por ser uma classe abstrata, `MethodLogic` não pode ser instanciada, e deve ser estendida para ser utilizada. Ao contrário das outras classes do framework, esta não deve ser explicitamente estendida pelos desenvolvedores, pois isto será feito automaticamente pelo tradutor RTR2Java.

Cada método que possua restrição temporal no código Java/RTR irá gerar um método normal de JAVA™ e uma subclasse de `MethodLogic`, contendo a implementação específica para sua necessidade.

4.5 Execução

O tradutor RTR2Java foi implementado de forma a não necessitar uma interface gráfica para ser executado. Desta forma e com a portabilidade de JAVA™, pode ser executado em qualquer computador com uma JVM instalada.

Seguem abaixo algumas imagens do tradutor sendo executado.

Conforme mostrado na Figura 4.7, além de passar os arquivos a serem traduzidos, o tradutor também aceita como parâmetros de entrada os nomes das classes SMC e CMC que devem ser utilizadas no processo de tradução. Esta informação é importante pois são estas classes que serão instanciadas pelo código gerado, sendo necessário re-traduzir o código se alguma destas classes necessitar ser alterada.

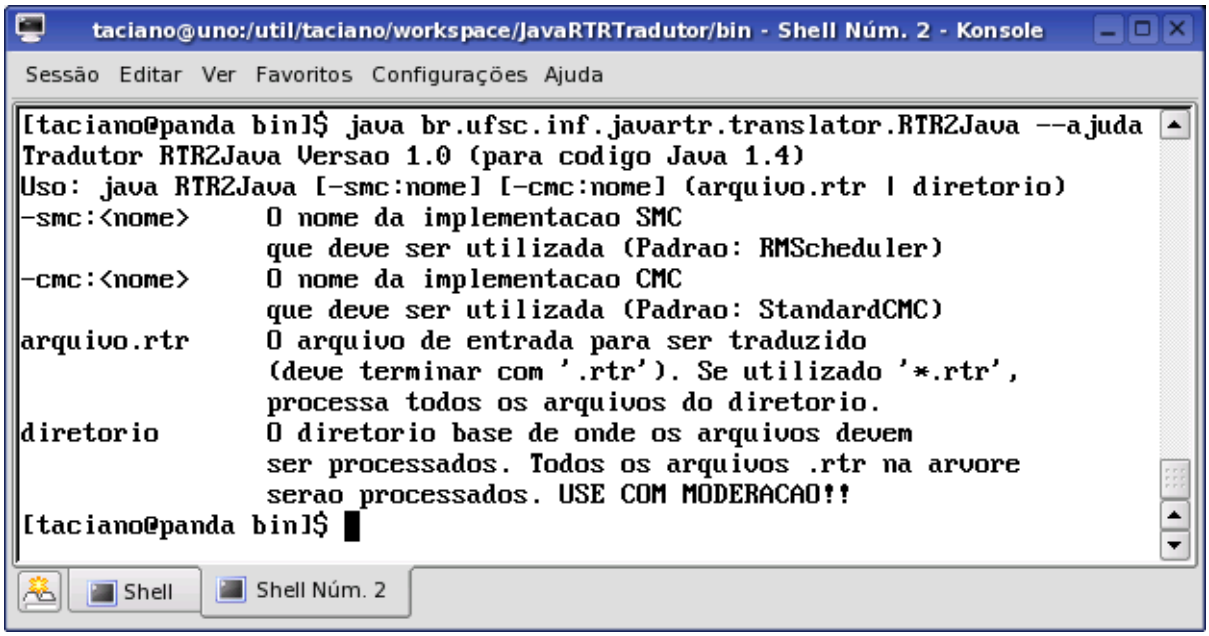

Figura 4.7: Com o parâmetro `--ajuda` é possível ter mais informações sobre a sintaxe de utilização do tradutor RTR2Java.

Na Figura 4.8 é possível observar as informações fornecidas pelo tradutor após o processo ser concluído com sucesso.

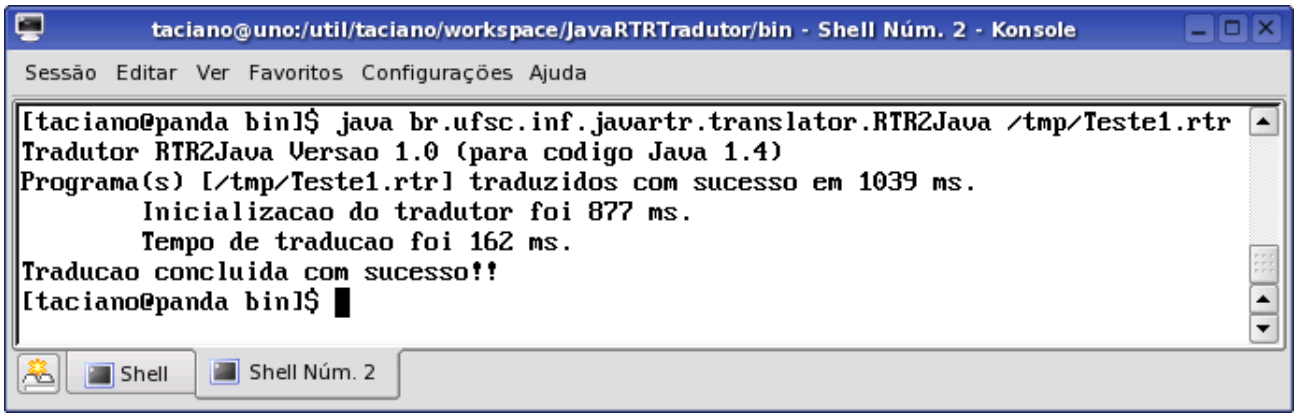

Figura 4.8: O arquivo cujo nome foi passado como parâmetro é traduzido, e um ou mais arquivos JAVA™ são gerados. Informações sobre o tempo consumido nas operações são reportadas no final do processo.

Capítulo 5

Integração com a Plataforma Eclipse

5.1 A plataforma Eclipse

O Projeto Eclipse.org (CON 04a) foi criado pela IBM, em uma subsidiária em Ottawa, no Canadá. Após algum tempo, a IBM disponibilizou para a comunidade a Plataforma Eclipse como código aberto[1], quando o projeto Eclipse.org já contava com muitos outros membros (Borland, OMG, Ericsson, QNX, RedHat, SAP, Fujitsu, Hitachi, HP, Suse, Intel e TymeSys, entre outros). Recentes modificações transformaram o projeto Eclipse.org na *Eclipse Foundation* (Fundação Eclipse), mantendo a tecnologia e o código desenvolvidos disponíveis livremente. Estas mudanças fizeram com que 17 novas empresas se juntassem à Fundação (um aumento de 30 por cento no número de membros) nos últimos seis meses. Também nove novos projetos de código aberto foram iniciados, e mais de 18 companhias desenvolvem produtos baseados na Plataforma Eclipse (Fonte: www.eclipse.org).

Atualmente na sua versão 3.0.1 (CON 04b), o ambiente, que é escrito em JAVA™, conta com a participação de toda uma comunidade de desenvolvedores que auxiliam reportando erros, solicitando e testando novas características e funcionalidades.

A Plataforma Eclipse é estruturada sobre o conceito de *pontos de extensão*[2]. Os pontos de extensão são lugares bem definidos do sistema onde outros componentes podem prover funcionalidades. Estes componentes estruturados que se descrevem

[1] Do inglês, *open source*

[2] Do inglês, *extension points*

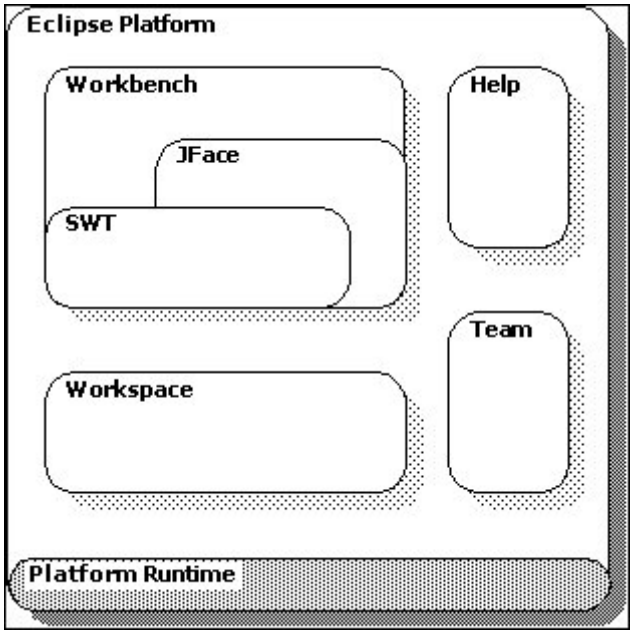

Figura 5.1: A plataforma Eclipse é estruturada em subsistemas que são implementados por um ou mais plug-ins. Estes subsistemas são desenvolvidos sobre um pequeno sistema de execução. Fonte: (CON 04a)

para o sistema utilizando um arquivo de manifesto (*plugin.xml*) são chamadas *plug-ins*. A plataforma possui um pequeno cerne, responsável por descobrir dinamicamente os *plug-ins* disponíveis e os carregar quando requisitado pelas atividades do usuário. Este núcleo também mantém um registro de todos os componentes disponíveis, assim como dos pontos de extensão. Excetuando-se o núcleo, todas as outras funcionalidades da plataforma são providas por *plug-ins*.

O alto custo dos ambientes de desenvolvimento profissionais faz com que os pequenos e médios desenvolvedores procurem outras opções. Iniciativas como o Eclipse e o Netbeans, da Sun Microsystems, recebem pontos extras por serem livres e permitirem a adição de novas características. No caso específico do Eclipse, ainda deve-se levar em conta sua alta modularidade (como descrito no parágrafo anterior).

É pensando na redução de custos que diversos *plug-ins* já foram desenvolvidos, tanto por empresas como por desenvolvedores solitários. Entre eles, podemos citar:

- EclipseUML, para o projeto de sistemas orientados a objetos utilizando a linguagem

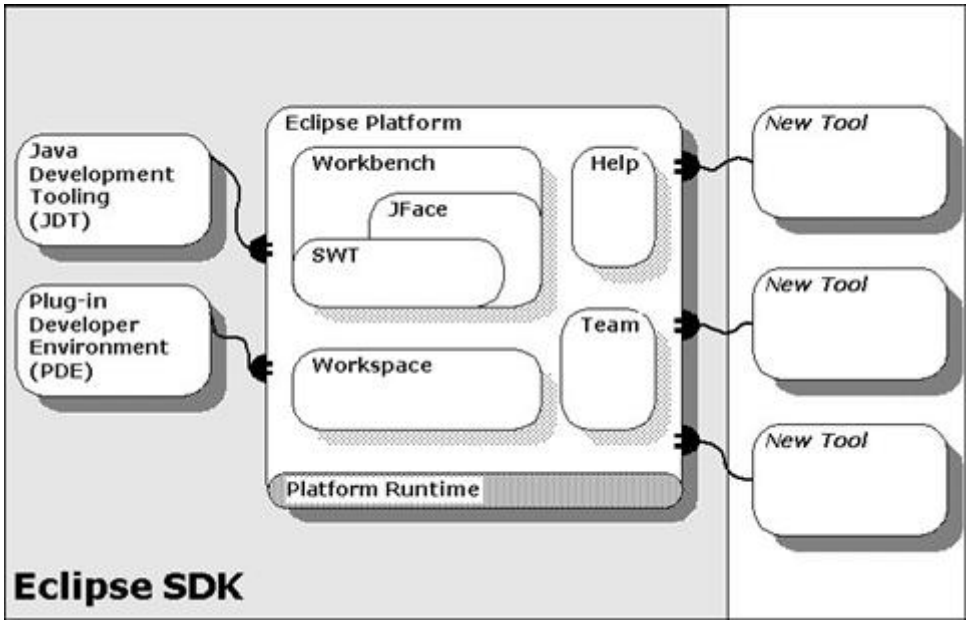

Figura 5.2: Como identificado nesta figura, mesmo as funcionalidades já providas pela plataforma são implementadas por plug-ins. Não está visível na figura, mas os plug-ins JDT e PDE (à esquerda) também possuem pontos de extensão e podem ser estendidos. Fonte: (CON 04a)

UML;

- SWT Advanced Designer, para a criação de interfaces gráficas utilizando o pacote *Standard Widget Toolkit*, o mesmo utilizado pela plataforma Eclipse;
- ecletex, para a escrita de artigos científicos utilizando a linguagem LaTeX (TeX). O texto deste trabalho foi inteiramente escrito utilizando este *plug-in;*
- jmechanic, para a mensuração e avaliação do desempenho de aplicações JAVA™.

Os *plug-ins* também podem prover outros pontos de extensão, ou simplesmente contribuir com uma extensão para os pontos já existentes. Na Figura 5.2 pode-se notar que o ambiente de desenvolvimento em JAVA™ provido (JDT, JAVA™ *Development Tooling*) e o ambiente para desenvolvimento de novos plug-ins (PDE, *Plug-in Development Environment*) são *plug-ins* providos juntamente com a plataforma.

Como diferencial para os ambientes concorrentes, a vantagem primordial do Eclipse é a possibilidade de se incorporar novas ferramentas à plataforma através

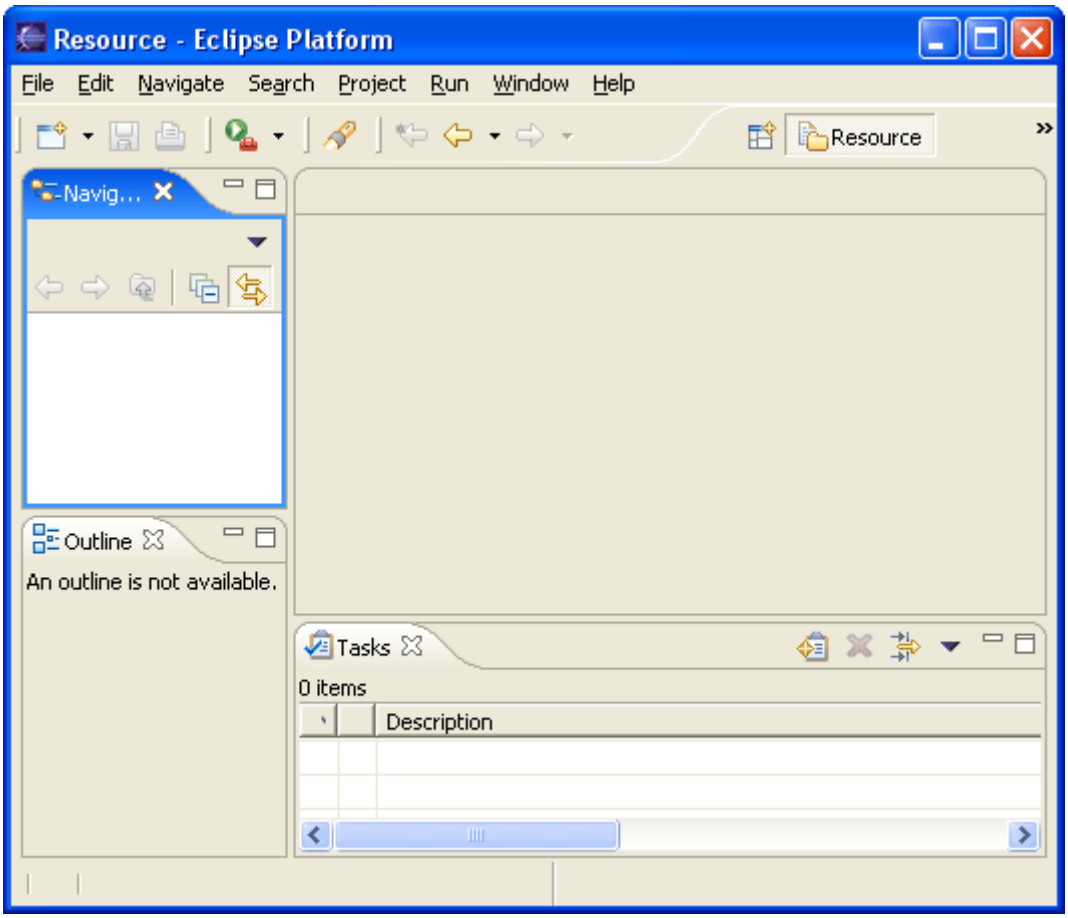

Figura 5.3: Esta é a interface do Eclipse após a instalação. São os plug-ins instalados que irão definir quais funcionalidades serão providas pelo ambiente. Fonte: (CON 04a)

de *plug-ins*, também desenvolvidos em JAVA™. Estas adições podem prover funcionalidades tão diversas quanto um ambiente de programação para outras linguagens ou uma ferramenta para criação e edição de diagramas UML. São os *plug-ins* instalados que definem as possibilidades de utilização do ambiente.

O resultado é “um IDE para tudo, e para nada em particular”[3].

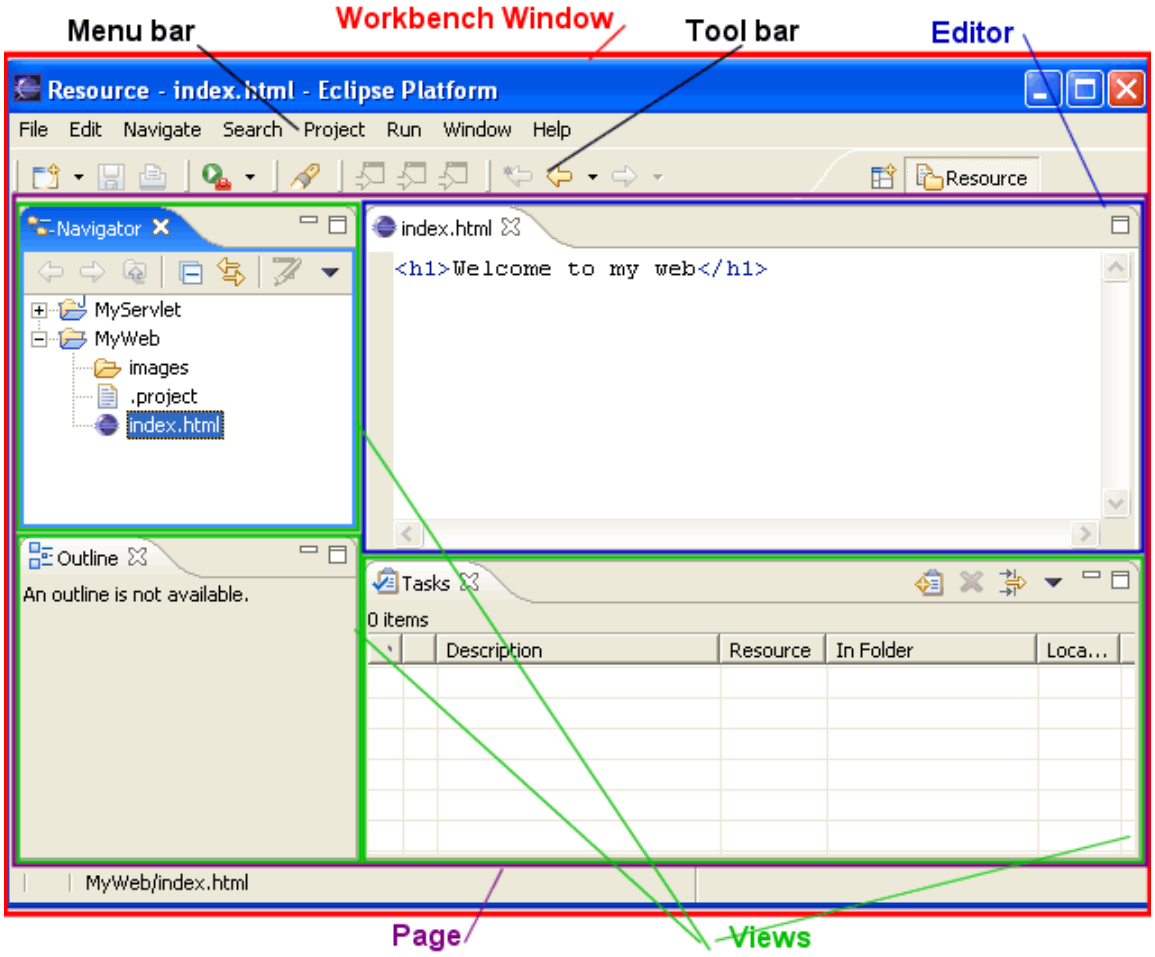

Figura 5.4: Identificação das partes da interface do Eclipse. Fonte: (CON 04a)

5.1.1 Componentes da Plataforma Eclipse

A seguir são apresentados os principais conceitos da Plataforma, e as partes da interface gráfica são detalhadas.

[3]Do inglês, “IDE for anything, and nothing in particular”

5.1.1.1 Plug-ins

São componentes estruturados de código JAVA™ adequadamente formatados e compostos de forma a aderir aos padrões exigidos pela plataforma do Eclipse. Cada um destes provê um arquivo de manifesto contendo informações sobre suas tarefas, suas interações com a interface gráfica e como o mesmo deve ser criado. Utilizando os *plug-ins* disponibilizados pela Plataforma, é possível prover novas funcionalidades ao IDE escrevendo poucas linhas de código, ainda que utilizando a interface gráfica provida (*Workbench*) para a interação com o usuário. Também é possível ter acesso aos recursos do *Workspace*, a área de trabalho da Plataforma.

5.1.1.2 Editores

São estruturas onde é possível visualizar e editar o conteúdo de arquivos. Esta edição pode ser feita de forma manual, com o usuário digitando as alterações, ou de forma visual, quando o usuário é auxiliado por menus para a modificação de determinado conteúdo.

Para os fins deste trabalho, um editor que permita a manutenção de código Java/RTR será suficiente para que os objetivos sejam alcançados.

5.1.1.3 Visões

São áreas no ambiente de trabalho que mostram informações que ajudam na tarefa que está sendo realizada. Podem ser utilizadas para visualizar o conteúdo de um arquivo de forma estruturada, por exemplo, ou para exibir as propriedades de um objeto selecionado.

Não serão desenvolvidas novas visões, mas o editor será integrado com as disponíveis pela plataforma, de forma a facilitar o trabalho do desenvolvedor.

5.1.1.4 Perspectivas

Uma perspectiva é um conjunto de editores e visões, dispostos de certa forma na interface, e que tem um objetivo comum. Junto com a Plataforma já estão disponíveis perspectivas para depurar código (Debug), escrever código JAVA™, gerenciar

conteúdo em servidores de CVS e para manusear os recursos do sistema operacional que se encontram dentro da pasta de trabalho do Eclipse.

Inicialmente, código JavaRTR será tratado como código JAVA™ simples, então uma nova perspectiva não será criada, pois as já existentes suportam bem esta característica e são adequadas para esta tarefa. Esta necessidade surgirá quando do desenvolvimento de outras ferramentas para o desenvolvimento de código tempo real.

5.1.1.5 Assistentes

Os assistentes[4] servem para auxiliar na execução de tarefas que consistem em diversos passos seqüenciais, como a criação de um novo projeto ou de uma nova classe, por exemplo. O conceito de assistentes se tornou comum nas áreas de desenvolvimento justamente por automatizar tarefas repetitivas e acelerar o processo de desenvolvimento.

Para este trabalho estão previstos assistentes para a criação de classes que possuam restrições temporais, e para a criação de projetos que utilizem a linguagem Java/RTR.

5.1.1.6 Builders

O Eclipse utiliza o conceito de builders para se referir ao ato de tratar algum recurso que esteja sendo gerenciado pela Plataforma. Os builders são registrados para determinados tipos de arquivos, e são executados sempre que arquivos deste tipo são alterados. Desta forma não é necessário um botão na interface que execute o compilador de uma linguagem, pois no momento que um arquivo é salvo, os builders associados aquele tipo são chamados, para que tratem as alterações efetuadas. Desta forma os arquivos estarão sempre em um estado de “já compilado”, sem a necessidade de se fazer isto manualmente.

Tomando como exemplo o builder de arquivos JAVA™, temos que o caso mais comum é que poucas classes sejam modificadas entre cada operação de salvamento. Quando isto ocorre, apenas estes recursos e os relacionados são compilados novamente. Esta compilação parcial é chamada de “incremental”, pois os recursos vão sendo

[4] Do inglês, *wizards*

compilados na medida que estão sendo modificados, sem a necessidade de re-compilar os que não foram modificados.

5.1.2 Integração da Plataforma com outras ferramentas e conceitos

Como já explicado, as funcionalidades do Eclipse são providas por *plug-ins*, e alguns dos mais importantes nas tarefas de desenvolvimento de código já estão disponíveis juntamente com a Plataforma.

5.1.2.1 CVS

Qualquer desenvolvimento de médio porte justifica a utilização de uma ferramenta para fazer o controle de versões dos arquivos gerados. Em projetos grandes, a grande utilidade é manter os diversos desenvolvedores em sincronia, enquanto em projetos com menos programadores serve principalmente como backup de segurança.

Um dos principais *plug-ins* disponibilizados com o Eclipse é o que faz a interface com os servidores de CVS. Composto de duas perspectivas (CVS Repository Exploring e Team Synchronizing) e algumas visões (como CVS Editors e CVS Repositories, por exemplo), este *plug-in* integra de forma completa o gerenciador de versões ao ambiente de desenvolvimento.

5.1.2.2 JUnit

Os primeiros desenvolvedores de programas de computador já se deparavam com a questão de como garantir que o código escrito iria funcionar. A utilização de suítes de testes unitários é uma forma de fazer isto, mostrando que o mesmo faz o que se propõe. Estes testes são responsáveis por exercitar o código sendo desenvolvido, de forma a mostrar que o mesmo funciona e, principalmente, como funciona.

Praticamente um padrão *de facto*, o pacote JUnit (JUN 04) suporta o desenvolvimento de suítes de testes na linguagem JAVA™, e está totalmente integrado ao ambiente Eclipse através do *plug-in* com o mesmo nome, também disponibilizado com a Plataforma. O *plug-in* disponibiliza funcionalidades responsáveis principalmente por executar os testes e reportar os erros encontrados durante a execução, garantindo um cres-

cimento seguro e consistente do código já que qualquer falha poderá ser imediatamente corrigida antes da adição de novas funcionalidades.

5.1.2.3 Refactoring

Focando nos conceitos de engenharia de software, o conceito de *refactoring* (FOW 99) também está fortemente integrado na Plataforma, principalmente no desenvolvimento de código JAVA™. Renomear um campo ou uma classe, extrair um método de outro, ou extrair uma interface de uma classe concreta são apenas alguns exemplos de atividades que podem ser executadas dentro do ambiente, de forma rápida, precisa e extremamente eficiente.

Com certeza há um ganho considerável de produtividade com a utilização correta destas funcionalidades.

5.1.3 Motivo da escolha

Tendo feito a apresentação inicial da Plataforma Eclipse, algumas características se mostraram mais atrativas na sua escolha como sendo o ambiente de desenvolvimento para a integração com a linguagem Java/RTR:

- sem custo de aquisição e manutenção;
- código fonte aberto e disponível;
- diversas fontes de informação sobre o ambiente: página Web, listas de discussão por *e-mail* e por *news*;
- desenvolvido em JAVA™;
- fácil de estender e modificar;
- naturalmente integrado com CVS;
- ótimo suporte ao JUnit;
- suporte a *refactoring* completo;

- editor de código JAVA™ com funcionalidades interessantes (Quick fix, formatação automática do código e adição automática de *imports*, por exemplo);
- conceito de builder automático e incremental;
- baseado em padrões da indústria (OSGi).

5.2 EclipseRTR2Java *plug-in*

5.2.1 Introdução

Como já explicado em 'Plug-ins' (p.75), os *plug-ins* são a menor unidade de código que provê uma funcionalidade na Plataforma Eclipse.

Cada *plug-in* adiciona novas características através dos *extension points* da Plataforma. Neste trabalho as seguintes extensões são utilizadas pelo *plug-in* EclipseRTR2Java:

- org.eclipse.ui.editors, para prover o editor de código Java/RTR;
- org.eclipse.ui.newWizards, para a criação dos projetos e das novas classes através de assistentes;
- org.eclipse.core.resources.natures, que permite a definição de uma nova natureza de projeto no ambiente. Neste trabalho foi definida a natureza JavaRTRNature;
- org.eclipse.core.resources.builders, que inclui um construtor associado a determinada natureza de projeto. Este é o ponto de contato da Plataforma com o tradutor RTR2Java.
- org.eclipse.ui.propertyPages, é a extensão que permite configurar as propriedades (tipos dos meta-objetos) dos projetos Java/RTR.

A classe `JavaRTRPlugin` é o núcleo da implementação do *plug-in*, sendo conhecida pelas outras classes que integram o mesmo. É ela que faz o contato com o tradutor RTR2Java.

5.2.2 Editor

De suma importância, é onde o usuário irá escrever o código Java/RTR, contando com o auxílio de recursos como *syntax coloring*.

Na Figura 5.5 é possível ter uma idéia da interface do editor na Plataforma Eclipse.

5.2.3 Nature

Para definir que determinado projeto contém arquivos com código Java/RTR, uma nova natureza (ART 03) foi criada. Esta natureza nada mais faz que configurar o builder para executar nos projetos do seu tipo.

5.2.4 Builder

Os projetos que possuem a natureza Java/RTR tem seu conteúdo avaliado pelo builder (ART 03) definido no *plug-in,* permitindo que os arquivos com extensão '.rtr' sejam processados pelo tradutor RTR2Java, e deles gerados arquivos JAVA™. Quando o compilador JAVA™ é executado, já encontra os arquivos escritos pelo tradutor RTR2Java.

A Figura 5.6 mostra a visão de progresso enquanto o *Workspace* está sendo reconstruído, e umas das partes disto é invocar o builder no projeto que contém os arquivos Java/RTR.

5.2.5 Assistentes

Os assistentes são uma funcionalidade fundamental em IDE's profissionais, pois agilizam o trabalho de construções de peças repetitivas de código, além de garantirem uma estrutura mínina e funcional.

A Figura 5.7 mostra a tela inicial de todos os assistentes disponíveis na Plataforma Eclipse, desde classes e interfaces JAVA™, até arquivos comuns e novos *plug-ins*. Neste ponto já é possível visualizar a categoria criada, Java/RTR, para organizar os assistentes referentes a código Java/RTR.

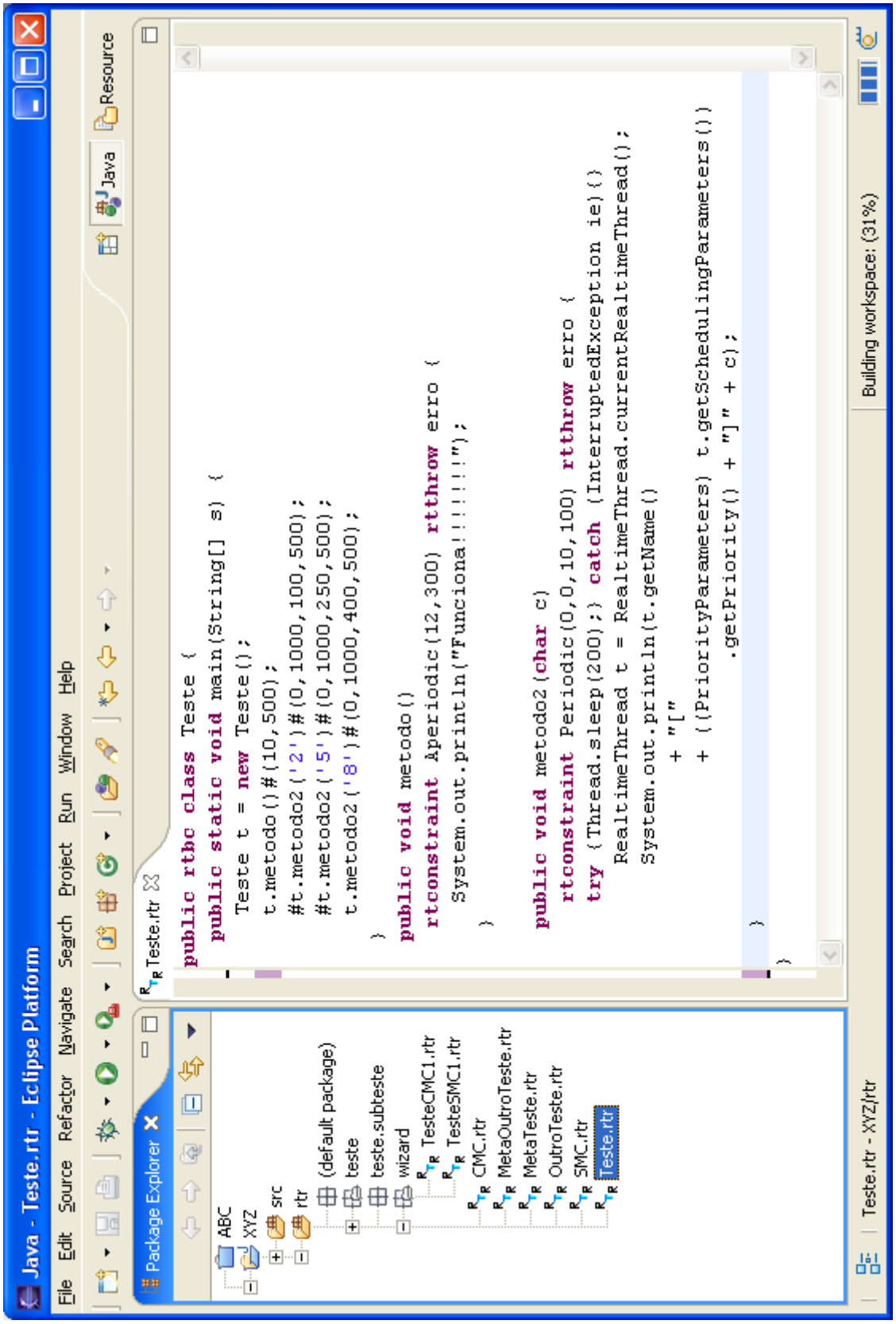

Figura 5.5: Editor de código Java/RTR. Note-se o destaque de palavras como ‘rtbc’, ‘rtconstraint’ e ‘rtthrow’. Também é possível visualizar algumas chamadas assíncronas no corpo do método ‘main’.

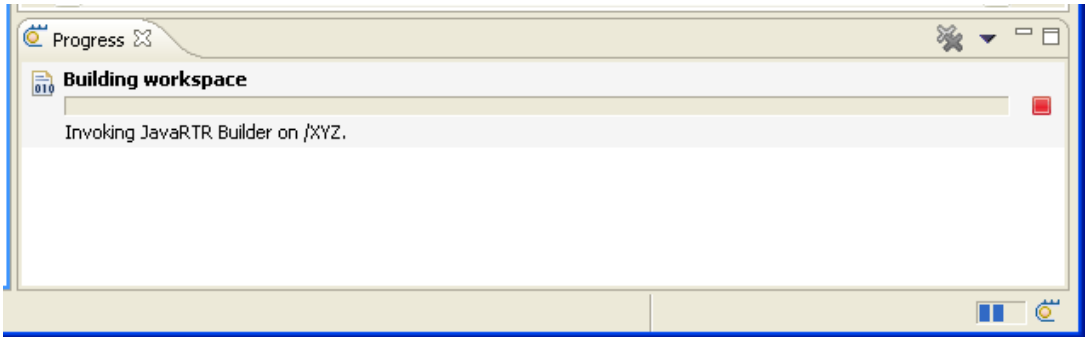

Figura 5.6: Visão de progresso da operação de compilação no projeto XYZ.

Dentro da categoria criada, é possível observar, selecionado, o assistente de criação de Projetos Java/RTR, além dos assistentes de criação de novas classes SMC e CMC. Para a criação dos assistentes foram utilizadas as recomendações de construção de interface detalhadas em (EDG 04).

Já na Figura 5.8 é possível ver o primeiro passo na criação de um novo projeto Java/RTR, onde é possível escolher a localização do projeto na árvore de diretórios (dentro do *Workspace* ou fora dele), e também configurar se serão utilizadas pastas diferentes para os arquivos fontes e os compilados.

5.2.6 Limitações

Mesmo estando funcional, outras características seriam muito úteis para facilitar ainda mais o desenvolvimento de código Java/RTR na Plataforma Eclipse. Podemos citar, por exemplo:

- um completador de código, para agilizar ainda mais a digitação e evitar erros tipográficos;
- uma visão com a estrutura do arquivo sendo editado, para facilitar a navegação pelas estruturas;
- suporte a *code folding*, para tirar da visão blocos de código que sejam desnecessários no momento;
- formatador de código, para manter o estilo no código sendo escrito.

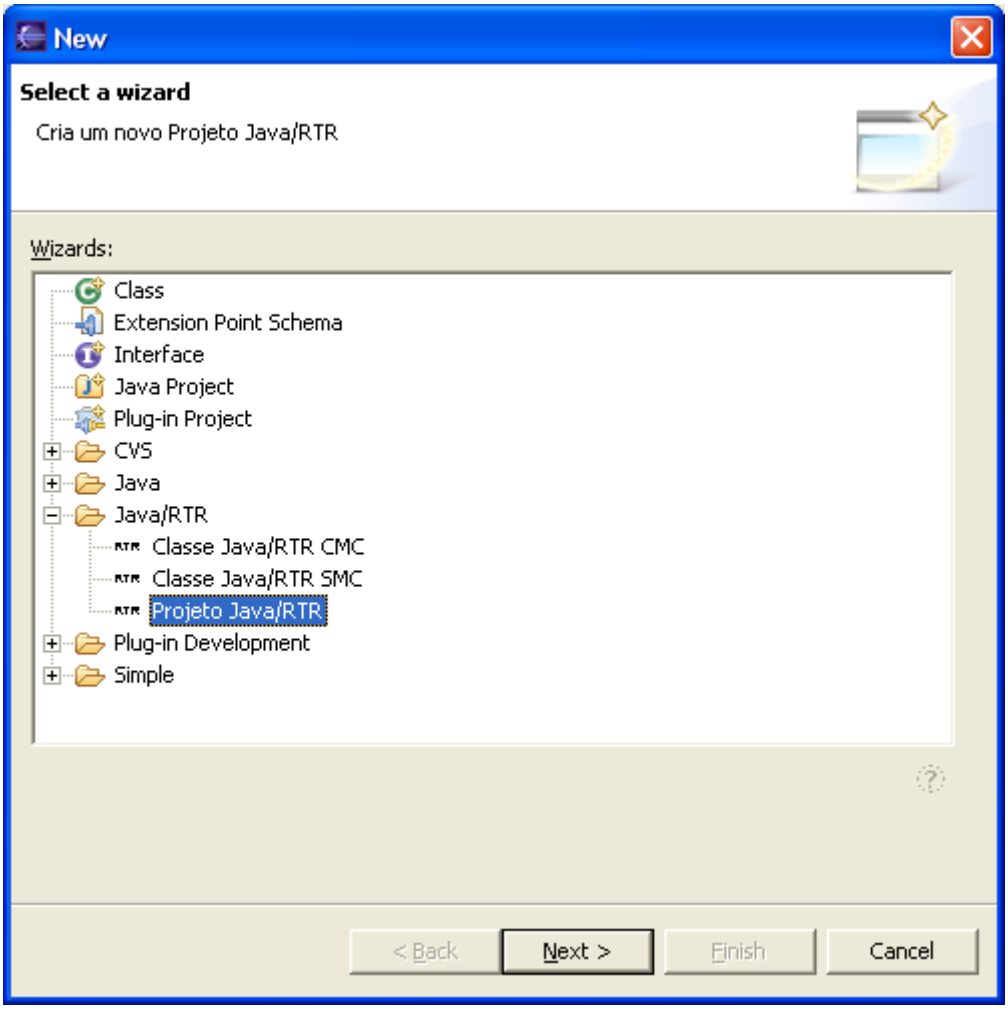

Figura 5.7: Assistente de criação de um novo projeto Java/RTR entre as escolhas disponíveis.

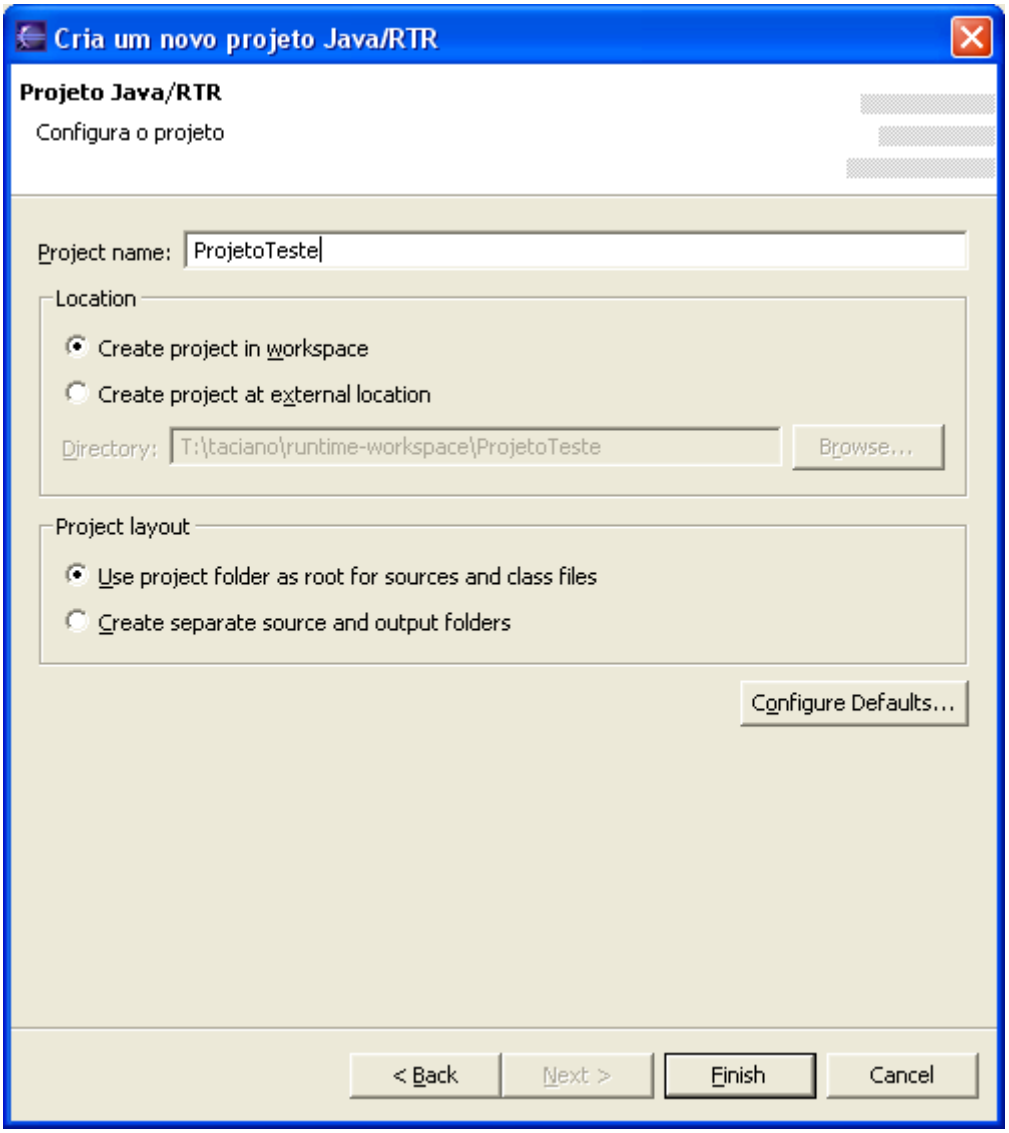

Figura 5.8: Assistente de criação de um novo projeto Java/RTR. De acordo com o nome do projeto já é criada a localização para o mesmo na árvore de diretórios.

Capítulo 6

Considerações Finais

6.1 Resultados

Como resultados, três pontos positivos principais foram alcançados com este trabalho. Os detalhes de cada um são apresentados nas seções a seguir.

6.1.1 Especificação formal da sintaxe de Java/RTR

Seguindo o Modelo RTR e a partir de uma especificação formal da linguagem JAVA™, esta última foi alterada e incrementada para mesclar com o Modelo, resultando na especificação formal da sintaxe da linguagem de programação Java/RTR.

Esta especificação define claramente o que é possível representar com a nova linguagem, suas estruturas e suas palavras reservadas, sem deixar de suportar as estruturas da linguagem JAVA™ original.

6.1.2 Implementação do tradutor RTR2Java

Para suportar a linguagem Java/RTR, o tradutor RTR2Java foi implementado, permitindo que código escrito nesta linguagem pudesse ser traduzido para a linguagem JAVA™, utilizando a especificação RTSJ como base para a representação dos aspectos temporais, acumulando as vantagens de ambas.

A implementação foi escrita em JAVA™, e não possui uma interface gráfica com o usuário, sendo acessada via linha de comando.

6.1.3 Implementação do plug-in EclipseRTR2Java

O *plug-in* desenvolvido faz com que os resultados anteriores usufruam do suporte e das vantagens da Plataforma Eclipse para o desenvolvimento de aplicações. O editor, os assistentes e a integração do tradutor fazem parte da implementação do *plug-in.*

A sintaxe da linguagem está realçada no editor, facilitando a editoração do código. E a plataforma fornece o sistema de *builder* para servir como *front-end* do tradutor RTR2Java. Os assistentes (*wizards*) auxiliam nas tarefas de criar novas implementações de algoritmos de escalonamento ou de meta-classes.

6.1.4 Disponibilização do código

Para que o código, a documentação e os resultados apresentados até aqui fiquem disponíveis após o término deste trabalho, um novo projeto foi criado no sítio SourceForge.net (OST 05). O SourceForge.net é um dos maiores sítios do mundo que hospeda projetos de aplicativos de código aberto, provendo ferramentas e serviços aos desenvolvedores. Atualmente hospeda mais de 90 mil projetos e conta com quase 1 milhão de usuários registrados.

Através do endereço `http://javartr.sourceforge.net/` é possível verificar as atualizações e baixar as últimas versões do tradutor, do *framework* e do *plug-in.* O código fonte também está disponibilizado através do CVS provido pelo sítio, e possui permissão de leitura para todos os visitantes (além de ser o objetivo do trabalho, é uma exigência do sítio, já que todos os projetos hospedados devem ter o código aberto).

Um detalhe importante é que as informações estão disponibilizadas em inglês, idioma oficial do serviço. Se isto atrapalha aos visitantes nacionais que não possuem fluência na língua, por outro lado permite que pessoas de todos os lugares do mundo possam ter conhecimento da tecnologia desenvolvida, sendo possível inclusive sua participação no projeto.

6.2 Discussão

6.2.1 Especificação formal da sintaxe de Java/RTR

A especificação formal de uma linguagem de programação serve de alicerce para a evolução da mesma. A partir dela podem ser construídos exemplos didáticos, um manual de utilização para desenvolvedores e para criadores de ferramentas, permitindo um completo entendimento da linguagem.

6.2.2 Implementação do tradutor RTR2Java

É ferramenta essencial para a utilização da linguagem Java/RTR. Além disto, a implementação é flexível o suficiente para suportar novas adições e incrementos de forma fácil e com pouca ou nenhuma mudança estrutural.

As novas adições são suportadas através da representação completa da estrutura dos arquivos '.rtr' através da AST criada pelo parser, o que permite que a implementação de um objeto *Visitor* (GAM 95) seja suficiente para executar as tarefas desejadas.

Os testes realizados inicialmente mostraram que a implementação é robusta e, mesmo utilizando apenas casos de testes relativamente pequenos, o desempenho foi considerado bom. Nesta implementação inicial a funcionalidade e praticidade tiveram preferência sobre o desempenho.

6.2.3 Implementação do plug-in EclipseRTR2Java

A integração do tradutor RTR2Java na Plataforma Eclipse permite grandes facilidades no desenvolvimento de aplicações que utilizem a linguagem Java/RTR, na sua maior parte características herdadas do ambiente.

O grande mote para a utilização deste *plug-in* é o editor integrado à Plataforma, com a funcionalidade de *syntax coloring*, mas os assistentes realizam uma tarefa essencial, principalmente para os iniciantes, que podem ver resultados em pouco tempo com a ferramenta.

6.3 Conclusão

Pelos resultados apresentados, é facilmente verificável que os objetivos propostos foram alcançados.

A linguagem Java/RTR foi formalmente especificada, sua utilização foi permitida através do tradutor RTR2Java e ambos foram integrados à Plataforma Eclipse. Com isto é possível desenvolver aplicações utilizando esta linguagem dentro do Eclipse, e, com poucas linhas de código, obter um resultado funcional.

Evidentemente, durante a execução deste trabalho novos objetivos foram estudados e analisados, sendo que alguns estão descritos na seção sobre trabalhos futuros.

6.3.1 Vantagens

Uma das grandes vantagens detes trabalho é a definição formal da linguagem Java/RTR, sendo sua utilização suportada pelo tradutor RTR2Java. Esta linguagem permite definir de forma concisa código que possua comportamento temporalmente restrito, de uma forma mais simples que a provida pela simples utilização da RTSJ.

Também temos a vantagem da reutilização de código, já que novas restrições temporais, novos escalonadores e novos relógios podem ser reutilizados entre diferentes aplicações.

A utilização da Plataforma Eclipse também permite uma maior facilidade no desenvolvimento de código Java/RTR através do editor especializado e da integração do tradudor, gerando automaticamente os arquivos ‘.java’ resultados da tradução dos arquivos ‘.rtr’.

Também pode-se citar como uma vantagem deste trabalho a utilização de assistentes na Plataforma Eclipse, reduzindo a complexidade para novos usuários e agilizando o desenvolvimento das aplicações.

6.3.2 Limitações

Neste ponto, merece ressalva a questão do ambiente de execução. Como já foi dito na Seção ‘Implementação de referência’ (p.27), a execução dos arquivos com

o *bytecode* gerado é possível apenas em computadores pessoais (PCs) com o sistema operacional Linux instalado, por ser esta uma das únicas configurações na qual a implementação de referência da RTSJ pode ser executada. Este limitação impede, por exemplo, de executar o código passo–a–passo, tarefa comum no processo de depuração de uma aplicação.

Sobre o *plug-in* implementado, uma limitação, relegada para ser resolvida nas futuras implementações, é a questão da representação da estrutura dos arquivos com código Java/RTR. Mesmo sendo um detalhe da implementação, é isto que falta para proporcionar funcionalidades na Plataforma como uma visão com a estrutura do arquivo (*Outline View*), auto-completação de código, auto-formatação de código e *code folding*, principalmente.

A respeito da tradução de código Java/RTR para código JAVA™ que utiliza a RTSJ, não há uma fórmula para garantir a manutenção da semântica entre as duas linguagens, mesmo que a proximidade das sintaxes ajude nesta tarefa. Não há prova formal de que o código gerado será semanticamente igual ao código fonte.

6.4 Trabalhos futuros

Uma sugestão para ser desenvolvida num próximo passo seria a união dos conceitos do Modelo RTR com a versão 5 da linguagem JAVA™ (SM 04c). Seria necessária uma nova versão do tradutor RTR2Java para suportar as alterações léxicas e sintáticas, principalmente o novo recurso de “anotações” (SM 04a), com o qual seria possível até reduzir a complexidade da linguagem Java/RTR. Mas para que esta sugestão possa ser desenvolvida também é necessária a alteração da máquina virtual que suporta a RTSJ, para que suporte a nova versão da linguagem.

Outra tecnologia que poderia ser utilizada é a Programação Orientada a Aspectos (*Aspect Oriented Programming*) (KIC 97; COM 04) que está integrada à linguagem JAVA™ com o projeto AspectJ (ASP 04), mantido pelo Eclipse.org. A mesma poderia ser utilizada para encapsular a complexidade de gerenciamento das restrições temporais em aspectos, isolando a camada de negócios. É verdade que seria necessária muita disciplina por parte do desenvolvedor para separar a solução do controle temporal

da mesma, mas o resultado seria a execução da aplicação como soma dos seus elementos principais com os aspectos que controlariam as questões temporais da mesma. Desta forma, seria possível executar a aplicação sem restrições temporais (sem aspectos), ou com diferentes níveis de restrição (representados por diferentes implementações de aspectos), dependendo da necessidade da execução.

Acompanhando o desenvolvimento de novas tecnologias, deve-se estar atento ao pedido de uma especificação no JCP (PRO 04) para sistemas de tempo real distribuídos (PRO 00). A proposta cita a necessidade de suporte para atividades distribuidas que possuam características e comportamento temporalmente restrito, sugerindo a extensão da tecnologia RMI (*Remote Method Invocation*) e da própria RTSJ. Esta nova especificação, que sozinha já seria campo para muito estudo, juntamente com o Modelo RTR permitem a expansão deste trabalho para facilitar o desenvolvimento de aplicações distribuídas.

Referências Bibliográficas

[AHO 86] AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers, Principles, Techniques and Tools**. 1. ed. Reading, USA: Addison-Wesley Pub Co, 1986.

[ART 03] ARTHORNE, J. **Project Builder and Natures**. Disponível em <http://www.eclipse.org/articles/Article-Builders/builders.html>. Acesso em: Novembro 2004.

[ASP 04] ASPECTJ. **AspectJ Home Page**. Disponível em <http://www.eclipse.org/aspectj/>. Acesso em: Outubro 2004.

[AWA 96] AWADA, M.; KUUSELA, J.; ZIEGLER, J. **Octopus: Object-Oriented Technology for Real-Time Systems**. Upper Saddle River, USA: Prentice Hall PTR, 1996.

[BEC 02] BECK, K. **Test Driven Development: By Example**. 1. ed. Reading, USA: Addison-Wesley Pub Co, Novembro, 2002.

[BOL 00a] BOLLELLA, G. et al. **The Real-Time Specification for JAVA™**. Disponível em <http://rtsj.dev.java.net/rtsj-V1.0.pdf>. Acesso em: Outubro 2004.

[BOL 00b] BOLLELLA, G. et al. **The Real-Time Specification for JAVA™**. Java Series. Upper Saddle River, USA: Prentice Hall PTR, Junho, 2000.

[BUR 01] BURNS, A.; WELLINGS, A. **Real-Time Systems and Programming Languages**. 3. ed. Reading, USA: Addison-Wesley Pub Co, 2001.

[COM 04] COMMITTEE, A. S. **Aspect-Oriented Software Development Home Page**. Disponível em <http://aosd.net/>. Acesso em: Outubro 2004.

[CON 04a] CONSORTIUM, E. **Eclipse.org Home Page**. Disponível em <http://www.eclipse.org/>. Acesso em: Outubro 2004.

[CON 04b] CONTE, P. **Inside Eclipse 3.0**. Disponível em <http://www.eclipsenews.com/en_articles/conte_0404.html>. Acesso em: Novembro 2004.

[COR 02] CORSARO, A.; SCHMIDT, D. C. The design and performance of the jrate real-time java implementation. **The 4th International Symposium on Distributed Objects and Applications**, Irvine, USA, v., Outubro, 2002.

[COR 04] CORSARO, A. **jRate Home Page**. Disponível em <http://www.cs.wustl.edu/∼corsaro/jRate/>. Acesso em: Outubro 2004.

[DIB 02] DIBBLE, P. C. **Real-Time JAVA™ Platform Programming**. 1. ed. Upper Saddle River, USA: Prentice Hall PTR, 2002.

[DOU 04] DOUGLASS, B. P. **Real Time UML: Anvances in the UML for Real-Time Systems**. 3. ed. Reading, USA: Addison-Wesley Pub Co, Fevereiro, 2004.

[EDG 04] EDGAR, N. et al. **Eclipse User Interface Guidelines**. Disponível em <http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html>. Acesso em: Outubro 2004.

[FOW 99] FOWLER, M. et al. **Refactoring: Improving the Design of Existing Code**. 1. ed. Reading, USA: Addison-Wesley Pub Co, Junho, 1999.

[FOX 01] FOX, J. **When is a Singleton not a Singleton?** Disponível em <http://java.sun.com/developer/technicalArticles/Programming/singletons/>. Acesso em: Novembro 2004.

[FUR 97] FURTADO, O. J. V. **RTR - Uma Abordagem Reflexiva para Programação de Aplicações Tempo Real**. Florianópolis, Brasil: Universidade Federal de Santa Catarina, Novembro, 1997. Tese de Doutorado.

[GAM 95] GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1. ed. Reading, USA: Addison-Wesley Pub Co, Janeiro, 1995.

[GEA 03] GEARY, D. **Simply Singleton**. Disponível em <http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns_p.html>. Acesso em: Novembro 2004.

[GOS 97] GOSLING, J.; JOY, B.; STEELE, G. **The JAVA™ Language Specification**. Reading, USA: Addison-Wesley Pub Co, 1997.

[JAV 04] JAVACC. **JavaCC Home Page**. Disponível em <http://javacc.dev.java.net/>. Acesso em: Outubro 2004.

[JJT 04] JJTREE. **JJTree Home Page**. Disponível em <http://javacc.dev.java.net/doc/JJTree.html>. Acesso em: Outubro 2004.

[JUN 04] JUNIT. **JUnit Home Page**. Disponível em <http://www.junit.org/>. Acesso em: Outubro 2004.

[KIC 97] KICZALES, G. et al. Aspect-oriented programming. In: Akşit, M.; Matsuoka, S., editors, PROCEEDINGS EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, v.1241, p.220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KUE 01] KUELKAMP, K. Implementação da linguagem JAVA™/RTR. Trabalho de Conclusão de Curso - INE - CTC - UFSC, Junho, 2001.

[LAP 97] LAPLANTE, P. A. **Real-Time Systems Design and Analysis**. 2. ed. Piscataway, USA: IEEE Press, 1997.

[LIU 91] LIU, J. W. S. et al. **Algorithms for scheduling imprecise computations**. Maio, 1991.

[LIU 94] LIU, J. W. S. et al. Imprecise computations. **IEEE Proceedings**, [S.l.], v.82, p.1–12, Janeiro, 1994.

[LIU 00] LIU, J. W. S. **Real-Time Systems**. 1. ed. Upper Saddle River, USA: Prentice Hall PTR, 2000.

[MAE 88] MAES, P. **Issues in Computational Reflection**. Amsterdam, Netherlands: Elsevier Science Publishers B. V., 1988.

[OST 05] OSTG, I. **SourceForge.net Home Page**. Disponível em <http://sourceforge.net/>. Acesso em: Janeiro 2005.

[PRO 98] PROCESS, J. C. **JSR-1: Real-Time Specification for JAVA™**. Disponível em <http://www.jcp.org/en/jsr/detail?id=1>. Acesso em: Novembro 2004.

[PRO 00] PROCESS, J. C. **JSR-50: Distributed Real-Time Specification**. Disponível em <http://www.jcp.org/en/jsr/detail?id=50>. Acesso em: Novembro 2004.

[PRO 04] PROGRAM, T. J. C. P. **The Java Community Process Program Web Site**. Disponível em <http://www.jcp.org>. Acesso em: Outubro 2004.

[RTJ 03] RTJ. **Real-Time Specification for Java Home Page**. Disponível em <http://rtsj.dev.java.net/>. Acesso em: Outubro 2004.

[SEL 92] SELIC, B. et al. Room: An object-oriented methodology for developing real-time systems. In: PROCEDINGS INTERNATIONAL WORKSHOP ON COMPUTER-AIDED SOFTWARE ENGINEERING, 1992. **Proceedings...** Piscataway, USA: [s.n.], 1992. p.6–10.

[SEL 94] SELIC, B.; GULLEKSON, G.; WARD, P. T. **Real-Time Object-Oriented Modeling**. 1. ed. Hoboken, USA: John Wiley and Sons, Inc., 1994.

[SEL 98] SELIC, B.; RUMBAUGH, J. Using UML for modeling complex real-time systems. ObjecTime Limited, 1998. Relatório técnico.

[SM 99] SUN MICROSYSTEMS, I. **Code Conventions for the JAVA™ Programming Language**. Disponível em <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>. Acesso em: Outubro 2004.

[SM 04a] SUN MICROSYSTEMS, I. **Annotations Guide**. Disponível em <http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>. Acesso em: Outubro 2004.

[SM 04b] SUN MICROSYSTEMS, I. **Java Home Page**. Disponível em <http://java.sun.com/>. Acesso em: Outubro 2004.

[SM 04c] SUN MICROSYSTEMS, I. **Java Programming Language 5.0 Guide**. Disponível em <http://java.sun.com/j2se/1.5.0/docs/guide/language/>. Acesso em: Outubro 2004.

[SON 95] SON, S. H. **Advances in Real-Time Systems**. Upper Saddle River, USA: Prentice Hall PTR, 1995.

[SYN 04] SYNOPSYS, I. **SystemC Home Page**. Disponível em <http://www.systemc.org/>. Acesso em: Outubro 2004.

[TIM 01a] TIMESYS. **Real-Time Specification for Java Reference Implementation**. Disponível em <http://www.timesys.com/index.cfm?bdy=java_bdy_ri.cfm>. Acesso em: Outubro 2004.

[TIM 01b] TIMESYS. **TimeSys Home Page**. Disponível em <http://www.timesys.com/>. Acesso em: Outubro 2004.

[TRE 02] TRES, T. Um framework para aplicações tempo real em JAVA™ usando JAVA™/RTR. Trabalho de Conclusão de Curso - INE - CTC - UFSC, Maio, 2002.

[WEL 04] WELLS, D. **Extreme Programing Home Page**. Disponível em <http://www.extremeprogramming.org/>. Acesso em: Dezembro 2004.

[ZIM 96] ZIMMERMANN, C. **Advances in Oriented Metalevel Architectures and Reflection**. 1. ed. Boca Raton, USA: CRC Press, 1996.

Apêndice A

Sintaxe de Java/RTR

```
/**
 This is a grammar for the Java/RTR language.
 It's based on Java 1.4 grammar file, with new
 constructs and new keywords added. Bellow is the list
 of the changes:

 # keywords
 - "rtbc"
 - "mmc"
 - "smc"
 - "cmc"
 - "rtcategory"
 - "eventtrigger"
 - "timetrigger"
 - "rtconstraint"
 - "rtthrow"
 - "rttype"

 # constructions
 - class declaration
 - method declaration
 - temporal constraint declaration

 Taciano Tres, 24/11/2003
 taciano@inf.ufsc.br
============================================================
 This is a bugfix of the grammar written by Sriram
 Sankar for Java 1.1, modified by David Williamns for
 Java 1.2, by Andrea Gini for Java 1.4  * and finally by
 Marco Savard to include a missing construct.

 According wiht the Java Language Sytax Specification,
 rules Classbody, ClassBodyDeclarations, InterfaceBody
```

```
and InterfaceBodyDeclarations, you can put any number
of semicolon between any production you can find inside
a class or an interface. These extra semicolons must be
treated in a different way from semicolons that
concludes some production (like abstract method
declarations or field declarations), because the latter
are mandatory. So I removed the modification previously
added by Marco Savard to the production
FieldDeclaration() (duplication of ";") and added ";"
as an optional derivation inside ClassBodyDeclaration()
and InterfaceBodyDeclaration(), in a way that mimic the
official Java Grammar, that can be found at the
following adress:
http://java.sun.com/docs/books/jls/second_edition/html/
syntax.doc.html

As an informal proof of correctness, now this grammar
accepts all the 3887 files of the JDK1.4 API.
Thanks to Marco Savard for having pointed out the
problem. To track changes, search for the string
"// added by Andrea Gini2"
Andrea Gini, 05/05/2002
ginian@tiscali.it
============================================================
According the point #19 of the java 1.2 compatibility
document of Sun
http://java.sun.com/products/jdk/1.2/compatibility.html
the following construct is valid since java 1.2 :

class D extends C {
  void f() {
  // overrides f() to run it in a new thread
      new Thread(new Runnable() {
          public void run() {
       D.super.f();
   }
}).start();
   }
}

Unfortunely, this was not supported by the previous
grammar. So I have adapted the grammar in order to
support it. To track changes, search for the string
"Modified by Marco Savard"
Marco Savard, 28/03/2002
marco.savard@magna-solutions.de
```

```
=============================================================
 This is a modified version of the grammar written by
 Sriram Sankar for Java 1.1 and modified by David
 Williamns for Java 1.2. It has been modified to accept
 Java sources for Java 1.4. The grammar have been
 modified in four parts: <br>
 1) 'assert' has been included to the keyword list<br>
 2) AssertStatement() production has been added<br>
 3) the production Statement() has been modified in
 order to support AssertStatement()<br>
 4) in the main the string "for Java1.2 code" has been
 changed with "for Java1.4 code"<br>

 To track changes, search for the string
 "// added by Andrea Gini"
 Andrea Gini, 24/02/2002
 ginian@tiscali.it
 ============================================================
 This file is a modified version of one originally found
 in the JavaGrammars Examples directory of JavaCC1_1. It
 has been modified to accept Java source code for Java
 1.2. Basically, this means a new key word was added,
 'strictfp', and that keyword added to the appropriate
 productions and LOOKAHEADs (where other, similar
 keywords are listed as possible choices). This involved
 changing 11 lines. Minor changes were also made to the
 main program for testing.
 The goal of this effort was for the grammar to be able
 to parse any legal Java 1.2 source code. It does not
 reject all illegal cases, but neither did the original.
 Plus, when it comes to the new 'strictfp' keyword, the
 Java Compiler from Sun (JDK1.2.1) also does not reject
 all illegal cases, as defined by the Updates document
 found at
 http://java.sun.com/docs/books/jls/strictfp-changes.pdf
(see the testcases.txt file for details).
 David Williams, 7/99
=============================================================
 Copyright (C) 1996, 1997 Sun Microsystems Inc.

 Use of this file and the system it is part of is
 constrained by the file COPYRIGHT in the root directory
 of this system. You may, however, make any
 modifications you wish to this file.
 Java files generated by running JavaCC on this file (or
 modified versions of this file) may be used in exactly
```

```
 the same manner as Java files generated from any
 grammar developed by you.

 Author: Sriram Sankar
 Date: 3/5/97

 This file contains a Java grammar and actions that
 implement a front-end.
*/

options {
  JAVA_UNICODE_ESCAPE = true;
  BUILD_NODE_FILES=true;
  NODE_SCOPE_HOOK=false;
  STATIC=false;
  MULTI=true;
  VISITOR=true;
  NODE_USES_PARSER=false;
  NODE_FACTORY=false;
}

PARSER_BEGIN(JavaRTRParser)
package br.ufsc.inf.javartr.tradutor;
public class JavaRTRParser
{
// modified main Taciano Tres 13/10/2004
// modified main Andrea Gini 24/02/2002
// modified main. DW, 7/99
public static void main (String [] args) {}
}

PARSER_END(JavaRTRParser)

/* WHITE SPACE */

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

/* COMMENTS */
```

```
MORE :
{
  "//" : IN_SINGLE_LINE_COMMENT
|
  <"/**" ~["/"]> { input_stream.backup(1); } :
   IN_FORMAL_COMMENT
|
  "/*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
}

<IN_FORMAL_COMMENT>
SPECIAL_TOKEN :
{
  <FORMAL_COMMENT: "*/" > : DEFAULT
}

<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <MULTI_LINE_COMMENT: "*/" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT,IN_FORMAL_COMMENT,
   IN_MULTI_LINE_COMMENT>
MORE :
{
  < ~[] >
}

/* RESERVED WORDS AND LITERALS */

TOKEN :
{
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
| < BREAK: "break" >
| < BYTE: "byte" >
| < CASE: "case" >
| < CATCH: "catch" >
| < CHAR: "char" >
```

```
| < CLASS: "class" >
| < CONST: "const" >
| < CONTINUE: "continue" >
| < _DEFAULT: "default" >
| < DO: "do" >
| < DOUBLE: "double" >
| < ELSE: "else" >
| < EXTENDS: "extends" >
| < FALSE: "false" >
| < FINAL: "final" >
| < FINALLY: "finally" >
| < FLOAT: "float" >
| < FOR: "for" >
| < GOTO: "goto" >
| < IF: "if" >
| < IMPLEMENTS: "implements" >
| < IMPORT: "import" >
| < INSTANCEOF: "instanceof" >
| < INT: "int" >
| < INTERFACE: "interface" >
| < LONG: "long" >
| < NATIVE: "native" >
| < NEW: "new" >
| < NULL: "null" >
| < PACKAGE: "package">
| < PRIVATE: "private" >
| < PROTECTED: "protected" >
| < PUBLIC: "public" >
| < RETURN: "return" >
| < SHORT: "short" >
| < STATIC: "static" >
| < SUPER: "super" >
| < SWITCH: "switch" >
| < SYNCHRONIZED: "synchronized" >
| < THIS: "this" >
| < THROW: "throw" >
| < THROWS: "throws" >
| < TRANSIENT: "transient" >
| < TRUE: "true" >
| < TRY: "try" >
| < VOID: "void" >
| < VOLATILE: "volatile" >
| < WHILE: "while" >
| < STRICTFP: "strictfp" >

// added by Andrea Gini
```

```
| < ASSERT: "assert" >

// added by Taciano Tres
| < RTBC: "rtbc" >
| < MMC: "mmc" >
| < SMC: "smc" >
| < CMC: "cmc" >
| < RTCATEGORY: "rtcategory" >
| < ET: "eventtrigger" >
| < TT: "timetrigger" >
| < RT: "rtconstraint" >
| < RTTHROW: "rtthrow" >
| < RTTYPE: "rttype" >
}

/* LITERALS */

TOKEN :
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"]
            (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
|
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
                    (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)?
                                  (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
|
  < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
```

```
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >
|
  < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

/* IDENTIFIERS */

TOKEN :
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
```

```
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}

/* SEPARATORS */

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >
| < COMMA: "," >
| < DOT: "." >
}

/* OPERATORS */

TOKEN :
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
| < BANG: "!" >
```

```
| < TILDE: "~" >
| < HOOK: "?" >
| < COLON: ":" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < SC_OR: "||" >
| < SC_AND: "&&" >
| < INCR: "++" >
| < DECR: "--" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < BIT_AND: "&" >
| < BIT_OR: "|" >
| < XOR: "^" >
| < REM: "%" >
| < LSHIFT: "<<" >
| < RSIGNEDSHIFT: ">>" >
| < RUNSIGNEDSHIFT: ">>>" >
| < PLUSASSIGN: "+=" >
| < MINUSASSIGN: "-=" >
| < STARASSIGN: "*=" >
| < SLASHASSIGN: "/=" >
| < ANDASSIGN: "&=" >
| < ORASSIGN: "|=" >
| < XORASSIGN: "^=" >
| < REMASSIGN: "%=" >
| < LSHIFTASSIGN: "<<=" >
| < RSIGNEDSHIFTASSIGN: ">>=" >
| < RUNSIGNEDSHIFTASSIGN: ">>>=" >
}


/*****************************************
 * THE JAVA/RTR LANGUAGE GRAMMAR STARTS HERE *
 *****************************************/

/*
 * Program structuring syntax follows.
 */

ASTCompilationUnit CompilationUnit() :
{} {
```

```
  [ PackageDeclaration() ]
  ( ImportDeclaration() )*
  ( TypeDeclaration() )*
  <EOF>
  { return jjtThis; }
}

void PackageDeclaration() :
{} { "package" Name() ";" }

void ImportDeclaration() :
{} { "import" Name() [ "." "*" ] ";" }

void TypeDeclaration() :
{} {
  // modified by Taciano Tres 24/11/2003
  LOOKAHEAD( ("abstract"|"final"|"public"|"strictfp" )*
  ( "rtbc" | "mmc" | "smc" | "cmc" )? "class" )
  ClassDeclaration()
|
  InterfaceDeclaration()
|
  ";"
}

/*
 * Declaration syntax follows.
 */
void ClassDeclaration() :
{} {
  ( "abstract" | "final" | "public" | "strictfp")*
  // added by Taciano Tres 24/11/2003
  [ RTRClassType() ]
  UnmodifiedClassDeclaration()
}

void RTRClassType() :
{} { ( "rtbc" | "mmc" | "smc" | "cmc" )? }

void UnmodifiedClassDeclaration() :
{} {
  "class" <IDENTIFIER> [ "extends" Name() ]
  [ "implements" NameList() ] ClassBody()
}

void ClassBody() :
```

```
{} { "{" ( ClassBodyDeclaration() )* "}" }

void NestedClassDeclaration() :
{} {
  ( "static" | "abstract" | "final" | "public" |
   "protected" | "private" | "strictfp")*
  UnmodifiedClassDeclaration()
}

/* Modified by Andrea Gini 2
 * According rules ClassBody and ClassBodyDeclaration
 * in the Java Language Specification,
 * semi-colons can be duplicated.
 * Source : http://java.sun.com/docs/books/jls/
 *              second_edition/html/syntax.doc.html
 */

void ClassBodyDeclaration() :
{} {
  LOOKAHEAD(2)
  Initializer()
|
  LOOKAHEAD( ("static"|"abstract"|"final"|"public" |
   "protected" | "private" | "strictfp")* "class" )
  NestedClassDeclaration()
|
  LOOKAHEAD( ("static"|"abstract"|"final"|"public"|
   "protected" | "private" | "strictfp")* "interface" )
  NestedInterfaceDeclaration()
|
  LOOKAHEAD(["public"|"protected"|"private"] Name() "(")
  ConstructorDeclaration()
|
  LOOKAHEAD( MethodDeclarationLookahead() )
  MethodDeclaration()
|
  FieldDeclaration()
|             // added by Andrea Gini2
  ";"         // added by Andrea Gini2
|
  TemporalConstraintDeclaration()//added by Taciano Tres
}

void TemporalConstraintDeclaration() :
{} {
  "rttype" <IDENTIFIER> "="
```

```
  [ TemporalConstraintType() ]
  FormalParameters() ";"
}

void TemporalConstraintType() :
{} { ( "eventtrigger" | "timetrigger" ) }

// This production is to determine lookahead only.
void MethodDeclarationLookahead() :
{} {
  ( "public" | "protected" | "private" | "static" |
   "abstract" | "final" | "native" | "synchronized" |
    "strictfp")*
  ResultType() <IDENTIFIER> "("
}

void InterfaceDeclaration() :
{} {
  ( "abstract" | "public" | "strictfp")*
  UnmodifiedInterfaceDeclaration()
}

void NestedInterfaceDeclaration() :
{} {
  ( "static" | "abstract" | "final" | "public" |
   "protected" | "private" | "strictfp")*
  UnmodifiedInterfaceDeclaration()
}

void UnmodifiedInterfaceDeclaration() :
{} {
  "interface" <IDENTIFIER> [ "extends" NameList() ]
  "{" ( InterfaceMemberDeclaration() )* "}"
}

/* Modified by Andrea Gini 2
 According to rules InterfaceBody and
 InterfaceBodyDeclaration in the Java Language
 Specification, semi-colons can be duplicated.
 Source : http://java.sun.com/docs/books/jls/
                  second_edition/html/syntax.doc.html
 */
void InterfaceMemberDeclaration() :
{} {
  LOOKAHEAD( ("static"|"abstract"|"final"|"public"|
   "protected" | "private" | "strictfp")* "class" )
```

```
  NestedClassDeclaration()
|
  LOOKAHEAD( ("static"|"abstract"|"final"|"public"|
   "protected" | "private" | "strictfp")* "interface" )
  NestedInterfaceDeclaration()
|
  LOOKAHEAD( MethodDeclarationLookahead() )
  MethodDeclaration()
|
  FieldDeclaration()
|             // added by Andrea Gini2
  ";"         // added by Andrea Gini2
}

void FieldDeclaration() :
{} {
  ("public"|"protected"|"private"|"static"|"final"|
   "transient" | "volatile" )*
  Type() VariableDeclarator()
                    ( "," VariableDeclarator() )* ";"
}

void VariableDeclarator() :
{} {
  VariableDeclaratorId() [ "=" VariableInitializer() ]
}

void VariableDeclaratorId() :
{} { <IDENTIFIER> ( "[" "]" )* }

void VariableInitializer() :
{} {
  ArrayInitializer()
|
  Expression()
}

void ArrayInitializer() :
{} {
  "{" [ VariableInitializer() ( LOOKAHEAD(2) ","
   VariableInitializer() )* ] [ "," ] "}"
}

void MethodDeclaration() :
{} {
  ( "public" | "protected" | "private" | "static" |
```

```
   "abstract" | "final" | "native" | "synchronized" |
    "strictfp")*
  ResultType() MethodDeclarator() ["throws" NameList()]

  // added by Taciano Tres 24/11/2003
  [ RTMethodDeclarator ]
  ( Block() | ";" )
}

void RTMethodDeclarator() :
{} {
"rtconstraint" <IDENTIFIER> Arguments()
     [ "rtthrow" <IDENTIFIER> ]
     [ "rtcategory" <IDENTIFIER> ]
}

void MethodDeclarator() :
{} {
  <IDENTIFIER> FormalParameters() ( "[" "]" )*
}

void FormalParameters() :
{} {
  "(" [FormalParameter() ( "," FormalParameter() )*] ")"
}

void FormalParameter() :
{} {
  [ "final" ] Type() VariableDeclaratorId()
}

void ConstructorDeclaration() :
{} {
  [ "public" | "protected" | "private" ]
  <IDENTIFIER> FormalParameters() ["throws" NameList()]
  "{"
    [ LOOKAHEAD(ExplicitConstructorInvocation())
        ExplicitConstructorInvocation() ]
    ( BlockStatement() )*
  "}"
}

void ExplicitConstructorInvocation() :
{} {
  LOOKAHEAD("this" Arguments() ";")
  "this" Arguments() ";"
```

```
|
  [ LOOKAHEAD(2) PrimaryExpression() "." ]
   "super" Arguments() ";"
}

void Initializer() :
{} { [ "static" ] Block() }


/*
 * Type, name and expression syntax follows.
 */
void Type() :
{} { ( PrimitiveType() | Name() ) ( "[" "]" )* }

void PrimitiveType() :
{} {
  "boolean"
|
  "char"
|
  "byte"
|
  "short"
|
  "int"
|
  "long"
|
  "float"
|
  "double"
}

void ResultType() :
{} { "void" | Type() }

void Name() :
/*
 * A lookahead of 2 is required below since "Name" can
 * be followed by a ".*" when used in the context of an
 * "ImportDeclaration".
 */
{} {
  <IDENTIFIER> ( LOOKAHEAD(2) "." <IDENTIFIER> )*
}
```

```
void NameList() :
{} {
  Name()
  ( "," Name()
  )*
}

/*
 * Expression syntax follows.
 */
void Expression() :
/*
 This expansion has been written this way instead of:
   Assignment() | ConditionalExpression()
 for performance reasons. However, it is a weakening of
 the grammar for it allows the LHS of assignments to be
 any conditional expression whereas it can only be a
 primary expression. Consider adding a semantic
 predicate to work around this.
 */
{} {
  ConditionalExpression()
  [
    AssignmentOperator() Expression()
  ]
}

void AssignmentOperator() :
{} {
  "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" |
   ">>=" | ">>>=" | "&=" | "^=" | "|="
}

void ConditionalExpression() :
{} {
  ConditionalOrExpression()
   [ "?" Expression() ":" ConditionalExpression() ]
}

void ConditionalOrExpression() :
{} {
  ConditionalAndExpression()
   ( "||" ConditionalAndExpression() )*
}
```

```
void ConditionalAndExpression() :
{} {
  InclusiveOrExpression() ("&&" InclusiveOrExpression())*
}

void InclusiveOrExpression() :
{} {
  ExclusiveOrExpression() ("|" ExclusiveOrExpression())*
}

void ExclusiveOrExpression() :
{} {
  AndExpression() ( "^" AndExpression() )*
}

void AndExpression() :
{} {
  EqualityExpression() ( "&" EqualityExpression() )*
}

void EqualityExpression() :
{} {
  InstanceOfExpression()
   ( ( "==" | "!=" ) InstanceOfExpression() )*
}

void InstanceOfExpression() :
{} {
  RelationalExpression() [ "instanceof" Type() ]
}

void RelationalExpression() :
{} {
  ShiftExpression()
   ( ( "<" | ">" | "<=" | ">=" ) ShiftExpression() )*
}

void ShiftExpression() :
{} {
  AdditiveExpression()
   ( ( "<<" | ">>" | ">>>" ) AdditiveExpression() )*
}

void AdditiveExpression() :
{} {
  MultiplicativeExpression()
```

```
   ( ( "+" | "-" ) MultiplicativeExpression() )*
}

void MultiplicativeExpression() :
{} {
  UnaryExpression() ( ( "*" | "/" | "%" )
   UnaryExpression() )*
}

void UnaryExpression() :
{} {
  ( "+" | "-" ) UnaryExpression()
|
  PreIncrementExpression()
|
  PreDecrementExpression()
|
  UnaryExpressionNotPlusMinus()
}

void PreIncrementExpression() :
{} { "++" PrimaryExpression() }

void PreDecrementExpression() :
{} { "--" PrimaryExpression() }

void UnaryExpressionNotPlusMinus() :
{} {
  ( "~" | "!" ) UnaryExpression()
|
  LOOKAHEAD( CastLookahead() )
  CastExpression()
|
  PostfixExpression()
}

// This production is to determine lookahead only. The
// LOOKAHEAD specifications below are not used, but they
// are there just to indicate that we know about this.
void CastLookahead() :
{} {
  LOOKAHEAD(2)
  "(" PrimitiveType()
|
  LOOKAHEAD("(" Name() "[")
  "(" Name() "[" "]"
```

```
|
  "(" Name() ")" ( "~" | "!" | "(" | <IDENTIFIER> |
   "this" | "super" | "new" | Literal() )
}

void PostfixExpression() :
{} {
  PrimaryExpression() [ "++" | "--" ]
}

void CastExpression() :
{} {
  LOOKAHEAD("(" PrimitiveType())
  "(" Type() ")" UnaryExpression()
|
  "(" Type() ")" UnaryExpressionNotPlusMinus()
}

void PrimaryExpression() :
{} {
  PrimaryPrefix() ( LOOKAHEAD(2) PrimarySuffix() )*
}

void PrimaryPrefix() :
{} {
  Literal()
|
  "this"
|
  "super" "." <IDENTIFIER>
|
  "(" Expression() ")"
|
  AllocationExpression()
|
  LOOKAHEAD( ResultType() "." "class" )
  ResultType() "." "class"
|
  Name()
}

/* Add "." "super" to be compatible with Java 1.2,
 * Modified by Marco Savard
 */
void PrimarySuffix() :
{} {
```

```
  LOOKAHEAD(2)
  "." "this"
|
  LOOKAHEAD(2)
  "." "super"
|
  LOOKAHEAD(2)
  "." AllocationExpression()
|
  "[" Expression() "]"
|
  "." <IDENTIFIER>
|
  ExtendedArguments()
}

void Literal() :
{} {
  <INTEGER_LITERAL>
|
  <FLOATING_POINT_LITERAL>
|
  <CHARACTER_LITERAL>
|
  <STRING_LITERAL>
|
  BooleanLiteral()
|
  NullLiteral()
}

void BooleanLiteral() :
{} { "true" | "false" }

void NullLiteral() :
{} { "null" }

void Arguments() :
{} { "(" [ ArgumentList() ] ")" }

// added by Taciano Tres
void ExtendedArguments() :
{} {
  "(" [ ArgumentList() ] ")"
  [ "#" "(" [ ArgumentList() ] ")" ]
}
```

```
void ArgumentList() :
{} {
  Expression() ( "," Expression() )*
}

void AllocationExpression() :
{} {
  LOOKAHEAD(2)
  "new" PrimitiveType() ArrayDimsAndInits()
|
  "new" Name()
    (
      ArrayDimsAndInits()
    |
      Arguments() [ ClassBody() ]
    )
}

/*
 The second LOOKAHEAD specification below is to parse
 to PrimarySuffixif there is an expression between
 the "[...]".
 */
void ArrayDimsAndInits() :
{} {
  LOOKAHEAD(2)
  ( LOOKAHEAD(2) "[" Expression() "]" )+
   ( LOOKAHEAD(2) "[" "]" )*
|
  ( "[" "]" )+ ArrayInitializer()
}

/*
 * Statement syntax follows.
 */
void Statement() :
{} {
  LOOKAHEAD(2)
  LabeledStatement()
|
  Block()
|
  EmptyStatement()
|
  StatementExpression() ";"
```

```
|
  SwitchStatement()
|
  IfStatement()
|
  WhileStatement()
|
  DoStatement()
|
  ForStatement()
|
  BreakStatement()
|
  ContinueStatement()
|
  ReturnStatement()
|
  ThrowStatement()
|
  SynchronizedStatement()
|
  TryStatement()
// added by Andrea Gini
|
  AssertStatement()
}

void LabeledStatement() :
{} { <IDENTIFIER> ":" Statement() }

void Block() :
{} { "{" ( BlockStatement() )* "}" }

void BlockStatement() :
{} {
  LOOKAHEAD([ "final" ] Type() <IDENTIFIER>)
  LocalVariableDeclaration() ";"
|
  Statement()
|
  UnmodifiedClassDeclaration()
|
  UnmodifiedInterfaceDeclaration()
}

void LocalVariableDeclaration() :
```

```
{} {
  [ "final" ] Type() VariableDeclarator()
   ( "," VariableDeclarator() )*
}

void EmptyStatement() :
{} { ";" }

void StatementExpression() :
/*
 The last expansion of this production accepts more than
 the legal Java expansions for StatementExpression. This
 expansion does not use PostfixExpression for
 performance reasons.
 */
{} {
  PreIncrementExpression()
|
  PreDecrementExpression()
|
  "#" Name() ExtendedArguments()
  ( "." Name() ExtendedArguments() )*
|
  PrimaryExpression()
  [
    "++"
  |
    "--"
  |
    AssignmentOperator() Expression()
  ]
}

void SwitchStatement() :
{} {
  "switch" "(" Expression() ")" "{"
    ( SwitchLabel() ( BlockStatement() )* )*
  "}"
}

void SwitchLabel() :
{} {
  "case" Expression() ":"
|
  "default" ":"
}
```

```
void IfStatement() :
/*
 The disambiguating algorithm of JavaCC automatically
 binds dangling else's to the innermost if statement.
 The LOOKAHEAD specification is to tell JavaCC that we
 know what we are doing.
 */
{} {
  "if" "(" Expression() ")" Statement()
   [ LOOKAHEAD(1) "else" Statement() ]
}

void WhileStatement() :
{} {
  "while" "(" Expression() ")" Statement()
}

void DoStatement() :
{} {
  "do" Statement() "while" "(" Expression() ")" ";"
}

void ForStatement() :
{} {
  "for" "(" [ ForInit() ] ";" [ Expression() ] ";"
   [ ForUpdate() ] ")" Statement()
}

void ForInit() :
{} {
  LOOKAHEAD( [ "final" ] Type() <IDENTIFIER> )
  LocalVariableDeclaration()
|
  StatementExpressionList()
}

void StatementExpressionList() :
{} {
  StatementExpression() ( "," StatementExpression() )*
}

void ForUpdate() :
{} { StatementExpressionList() }

void BreakStatement() :
```

```
{} { "break" [ <IDENTIFIER> ] ";" }

void ContinueStatement() :
{} { "continue" [ <IDENTIFIER> ] ";" }

void ReturnStatement() :
{} { "return" [ Expression() ] ";" }

void ThrowStatement() :
{} { "throw" Expression() ";" }

void SynchronizedStatement() :
{} {
  "synchronized" "(" Expression() ")" Block()
}

void TryStatement() :
/*
 Semantic check required here to make sure that at
 least one finally/catch is present.
 */
{} {
  "try" Block()
  ( "catch" "(" FormalParameter() ")" Block() )*
  [ "finally" Block() ]
}

// added by Andrea Gini
void AssertStatement() :
{} {
   "assert" Expression() [ ":" Expression() ] ";"
}
```

Índice Remissivo