UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

TRADUÇÃO DA ESPECIFICAÇÃO SCJ PARA LINGUAGEM DE PROGRAMAÇÃO C++

DISSERTAÇÃO DE MESTRADO

Ricardo Frohlich da Silva

Santa Maria, RS, Brasil

2015

TRADUÇÃO DA ESPECIFICAÇÃO SCJ PARA LINGUAGEM DE PROGRAMAÇÃO C++

Ricardo Frohlich da Silva

Dissertação apresentada ao Curso de Mestrado Programa de Pós-Graduação em Informática (PPGI), Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de

Mestre em Ciência da Computação

Orientador: Prof. Dr. Osmar Marchi dos Santos

Santa Maria, RS, Brasil

2015

da Silva, Ricardo Frohlich

Tradução da especificação SCJ para Linguagem de Programação C++ / por Ricardo Frohlich da Silva. – 2015.
94 f.: il.; 30 cm.

Orientador: Osmar Marchi dos Santos
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2015.

1. SCJ. 2. RTSJ. 3. RTOS. 4. Real Time. 5. Java. 6. Posix. 7. Thread. 8. Pthread. 9. Critical Systems. I. dos Santos, Osmar Marchi. II. Título.

E-mail: ricardosma@gmail.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

TRADUÇÃO DA ESPECIFICAÇÃO SCJ PARA LINGUAGEM DE PROGRAMAÇÃO C++

elaborada por
Ricardo Frohlich da Silva

como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

Osmar Marchi dos Santos, Dr.
(Presidente/Orientador)

Simone Regina Ceolin, Dr[a]. (UFSM)

Reiner Franthesco Perozzo, Dr. (UNIFRA)

Santa Maria, 28 de Abril de 2015.

Dedico este trabalho aos meus pais Braziliano Ferreira da Silva e Noemi Susana da Silva e a minha noiva Juliana Aires Rieta, por todo amor, carinho, compreensão e incentivo, sem os quais esta conquista seria muito árdua e penosa.

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Dr. Osmar Marchi dos Santos pela compreensão, por estar sempre disposto a esclarecer minhas dúvidas, dividir o seu conhecimento e me apoiar nos momentos mais difíceis.

A minha família, principalmente meus pais Braziliano Ferreira da Silva e Noemi Susana da Silva por estarem sempre do meu lado me apoiando.

A minha noiva Juliana Aires Rieta e meu enteado Luiz Henrique Rieta da Rocha por apoiar todas as minhas decisões e estar presente em todos os momentos.

A CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo apoio financeiro a este trabalho.

Aos meus amigos e colegas pelas eternas palavras de apoio, compreensão e momentos de descontração.

"Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito. Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes."

— (MARTHIN LUTHER KING)

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

TRADUÇÃO DA ESPECIFICAÇÃO SCJ PARA LINGUAGEM DE PROGRAMAÇÃO C++

AUTOR: RICARDO FROHLICH DA SILVA
ORIENTADOR: OSMAR MARCHI DOS SANTOS
Local da Defesa e Data: Santa Maria, 28 de Abril de 2015.

Sistemas críticos são sistemas em que suas falhas podem causar danos irreparáveis como colocar a vida de pessoas em risco e por este motivo envolve questões de segurança e exige uma validação rigorosa no processo de certificação. Processos de certificação são caros e demorados que seguem leis e regras rigorosas. Com a evolução contínua proporcionada por linguagens de programação de propósito geral, a facilidade de aprendizado, assim como a utilização destas linguagens na indústria e acadêmia, pesquisas vem sendo realizadas com o intuito de adaptar linguagens de programação de propósito gerais para serem utilizadas em aplicações críticas de tempo real. O objetivo destas adaptações é de tornar o escopo de comandos das linguagens para desenvolvimento de sistemas críticos mais restritos, como por exemplo, ao evitar ou reduzir a utilização de recursões. Alguns exemplos dessas adaptações são a Especificação de Tempo Real Java (*Real Time Specification for Java* - RTSJ) desenvolvida no ano de 1998, e a *Safety Critical Java* (SCJ) que utiliza objetos e conceitos definidos pela RTSJ com enfoque no desenvolvimento de aplicações para sistemas críticos. Na SCJ foi implementado o conceito de missões onde cada missão é composto por objetos escalonáveis definidos pela RTSJ. A portabilidade de uma aplicação desenvolvida em Java é um dos principais fatores dos quais desenvolvedores desejam utilizá-la. Todavia, existe uma grande dificuldade de encontrar máquinas virtuais para sistemas críticos embarcados, dificultando a portabilidade da qual a linguagem Java fornece. Por outro lado, uma aplicação desenvolvida na linguagem de programação C++ pode ser executada diretamente no dispositivo sem a necessidade de utilizar uma máquina virtual. Por este motivo, nesta dissertação é apresentada uma tradução da especificação *Safety Critical Java* na linguagem de programação C++, com o objetivo de manter o comportamentos de uma aplicação desenvolvida em SCJ e assim possibilitando a execução de uma aplicação com requisitos temporais em diversos dispositivos embarcados.

Palavras-chave: SCJ. RTSJ. RTOS. Real Time. Java. Posix. Thread. Pthread. Critical Systems.

ABSTRACT

Master's Dissertation
Post-Graduate Program in Informatics
Federal University of Santa Maria

TRANSLATION SCJ SPECIFICATION IN C ++ PROGRAMMING LANGUAGE

AUTHOR: RICARDO FROHLICH DA SILVA
ADVISOR: OSMAR MARCHI DOS SANTOS
Defense Place and Date: Santa Maria, April 28st, 2015.

Safety critical systems are systems where its failures can cause irreparable damage for this reason the development of safety critical systems involves safety issues and require rigorous validation in the certification process. Certification processes are expensive and lengthy to follow laws and rigorous rules. With the continuous evolution provided by general purpose programming languages, ease of learning, and the use of these languages in industry and academy, researches have been performed aiming to adapt general purpose programming languages for use in safety-critical applications. The purpose of these adaptations is to reduce the scope of commands found in general purpose languages in order to develop safety critical systems, for example, to avoid or reduce the use of recursions. Some examples of these adaptations include the Real Time Specification for Java (RTSJ), developed in 1998 and Safety Critical Java. SCJ uses objects and concepts defined by the RTSJ focusing on the development of safety-critical applications. In SCJ, the concept of missions is deployed where each mission consists of schedulable objects defined by the RTSJ. The portability of a Java application is one of the main factors for choosing this language. However, there is great difficulty in finding virtual machines for embedded safety-critical systems, therefore it is difficult to benefit from the portability provided by the Java virtual machine in this context. Nevertheless, an application developed in the C++ programming language can be executed directly on the device without using a virtual machine. This work presents a translation of the Safety Critical Java to the C++ programming language, maintaining the behaviour of objects that implement the concept of missions for SCJ in C++. This enables the execution of safety-critical applications in embedded devices without the use of a virtual machine.

Keywords: SCJ. RTSJ. RTOS. Real Time. Java. Posix. Thread. Pthread. Critical Systems. C++..

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ANEXOS

LISTA DE ABREVIATURAS E SIGLAS

ANSI	*American National Standards Institute*
APEH	*Aperiodic Event Handler*
ARM	*Advanced RISC Machine*
CPU	*Central Processing Unit*
CSP	*Communicating Sequential Processes*
HVM	*Hardware Virtual Machine*
JOP	*Java Optimized Processor*
JVM	*Java Virtual Machine*
PEH	*Periodic Event Handler*
POSIX	*Portable Operating System Interface*
RAM	*Random-access memory*
RTJTEG	*Real-Time for Java Experts Group*
RTSJ	*Real Time Specification For Java*
SCJ	*Safety Critical Java*
UTP	*Unshielded Twisted Pair*

SUMÁRIO

1 INTRODUÇÃO

Sistemas de tempo real (*Real-Time Systems*) são aplicações em que o tempo de execução da tarefa é essencial, pois a resposta deverá ocorrer dentro de um período pré-determinado, como por exemplo sistemas de controladores de voo (BURNS; WELLINGS, 2009). Estes sistemas podem ser de dois tipos: críticos e não críticos. Em sistemas não críticos o não cumprimento deste prazo é aceitável, até determinada taxas de erro, embora não seja desejado (TANENBAUM, 2012). Já em sistemas críticos é necessário que ocorra uma garantia de que uma ação seja executada em um intervalo de tempo, caso não haja uma resposta dentro deste tempo é considerado que houve falha do sistema (BURNS; WELLINGS, 2009).

Sistemas críticos (*Safety Critical Systems*) são sistemas em que suas falhas podem causar danos irreparáveis como danos materiais significativos, danos ao meio ambiente e até colocar a vida de pessoas em risco (SUNANDA; SEETHARAMAIAH, 2015). Estes sistemas vão desde sistema para controladores de voo, sistemas aeroespaciais, sistemas automotivos entre outros (SILVA; LOPES, 2012). A segurança de um sistema não se refere apenas ao *software*, mas também inclui o trabalho seguro do *hardware* (PARK et al., 2014).

Por estes motivos os sistemas críticos exigem uma validação rigorosa no processo de certificação (E. Y. HU; WELLINGS, 2002). Processos de certificação para sistemas críticos ficam por responsabilidade de órgãos governamentais através de leis jurídicas ou por autoridades de certificação (GROUP, 2013). Nos Estados Unidos, a Administração Federal de Aviação exige que os Sistemas *Safety Critical* sejam certificados por organizações independentes utilizando padrões controlados, como por exemplo, DO-176B ou ED-12B na Europa (GROUP, 2013). A certificação é um processo caro e demorado e em razão do rigor destas certificações, verificou-se a necessidade de tornar o escopo de comandos das linguagens para desenvolvimento destes sistemas mais restritas, como por exemplo, ao evitar ou reduzir a utilização de recursões (LOCKHART; PURDY; WILSEY, 2014).

Com a evolução contínua proporcionada por linguagens de programação de propósito geral, a facilidade de aprendizado, assim como a utilização destas linguagens na indústria e acadêmia, pesquisas vem sendo realizadas com o intuito de adaptar linguagens de programação de propósito gerais, como a linguagem Java, para serem utilizadas em aplicações críticas de tempo real (ANJOS, 2009).

Nos anos 90, mais especificamente em 1998, um grupo de pesquisadores e desenvolve-

dores definiram a Especificação para Tempo Real Java (*Real-Time Specification for Java* - RTSJ) que estende bibliotecas e interfaces, assim adicionando requisitos sobre a máquinas virtuais Java, permitindo o desenvolvimento de aplicações de tempo real nessa linguagem (G. BOLLELLA B. BROSGOL; TURNBULL, 2000). A RTSJ foi concebida com o intuito de executar aplicações de tempo real e genéricas na mesma maquina virtual (ANJOS, 2009). Por outro lado, existem diversas dificuldades em desenvolver aplicações críticas (*Safety Critical Applications*) utilizando a RTSJ devido à validação rigorosa e processos de certificação necessários para tais aplicações.

Por essas limitações, a especificação *Safety Critical Java* (SCJ) foi desenvolvida com objetivo de ser utilizada em aplicações críticas auxiliando nos processos de validação e certificação mais rigorosos (GROUP, 2013). A SCJ é uma especificação onde foi definido o conceito de missões. Uma missão consiste em um conjunto limitado de objetos escalonáveis definidos pela RTSJ (GROUP, 2013). A SCJ possibilita o desenvolvimento de aplicações através de uma especificação mais robusta comparada ao RTSJ, preparando-a para a validação (SCHOEBERL et al., 2007).

A portabilidade de uma aplicação desenvolvida em Java é um dos principais fatores que leva os programadores a selecionar esta linguagem para desenvolver, porém, existe uma grande dificuldade em encontrar máquinas virtuais de tempo real para sistemas embarcados (DAWSON, 2008) (ANJOS, 2009). Uma aplicação desenvolvida na linguagem de programação C++ pode ser executada diretamente no dispositivo sem a necessidade da utilização de uma máquina virtual (BROSGOL, 2007).

De acordo com a especificação SCJ, três diferentes níveis de desenvolvimento para aplicações críticas são suportados (GROUP, 2013): (i) sistemas sem concorrência - nível 0; (ii) sistemas com concorrência para arquiteturas uniprocessadas - nível 1; e (iii) sistemas com concorrência para arquiteturas multiprocessadas - nível 2.

Neste contexto, a presente dissertação tem como objetivo propor a tradução do comportamento de aplicações considerando o nível 1 da especificação SCJ (sistemas com concorrência para arquiteturas uniprocessadas) na linguagem de programação C++. Dessa forma, é possível manter o conceito de missões desenvolvido em SCJ na linguagem de programação C++, executando uma determinada aplicação diretamente no sistema embarcado, sem a necessidade de uma máquina virtual dedicada.

O principal diferencial deste trabalho é proporcionar ao desenvolvedor de aplicações

críticas a possibilidade de traduzir aplicações SCJ para a linguagem de programação C++. Uma aplicação utilizando a tradução proposta nesta pesquisa é possível obter benefícios fornecidos pela especificação como a possibilidade de validação da aplicação seguindo os processos rigorosos necessários para certificar os requisitos de segurança para o funcionamento de aplicações críticas (*safety critical applications*).

1.1 Objetivos

Esta dissertação tem como objetivo a tradução de componentes da especificação *Safety Critical Java* para a linguagem de programação C++ com o objetivo de manter o mesmo comportamento de uma aplicação originalmente desenvolvida utilizando a especificação SCJ. A tradução é proposta considerando a execução da aplicação traduzida em um sistema operacional que implemente o padrão *Portable Operating System Interface* (POSIX) (GROUP, 2007) como uma distribuição do Sistema Operacional Linux (YAGHMOUR; MASTERS; BEN, 2008).

1.1.1 Objetivos Específicos

O presente trabalho apresenta os seguintes objetivos específicos:

- Análise sobre o comportamento dos componentes da especificação *Safety Critical Java* (SCJ);
- Definição de mecanismos básicos da especificação SCJ para serem traduzidos para a linguagem C++;
- Desenvolvimento de uma estratégia de tradução;
- Aplicação da estratégia em estudos de caso para validação.

1.2 Estrutura

Esta dissertação é estruturada da seguinte forma; No Capítulo 2 é possvel visualizar o referencial teórico onde são apresentados conceitos pertinentes para a realização deste trabalho como, a *Real-Time Specification for Java* (RTSJ), a especificação *Safety Critical Java* (SCJ), o conceito do padrão POSIX (*Portable Operating System Interface*) e trabalhos relacionados a esta dissertação. O Capítulo 3 apresenta a proposta de tradução, considerando os componentes específicos da especificação SCJ, para a linguagem de programação C++. No Capítulo 4 é

descrita a validação e testes desenvolvidos, onde são visualizados estudos de caso da tradução, verificando se a tradução para a linguagem de programação C++ alcança o objetivo de manter o mesmo comportamento da aplicação original em SCJ. No Capítulo 5 são discutidas as considerações finais e os trabalhos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo é apresentada uma revisão sobre temas pertinentes para o desenvolvimento desta dissertação. Inicialmente é apresentada uma revisão sobre a especificação *Real Time Specification For Java* e os principais componentes. Na Seção 2.2 é apresentada a especificação *Safety Critical Java* descrevendo o seu funcionamento, seus níveis e um exemplo da utilização de SCJ que será utilizado no decorrer dessa dissertação. Uma revisão sobre *Portable Operating System Interface* (POSIX), descrevendo comportamentos e sua utilização nesta pesquisa será vista na Seção 2.3. E por último, na Seção 2.4, são apresentados os trabalhos relacionados encontrados na literatura.

2.1 *Real Time Specification For Java*

O grupo *Real-Time for Java Expert Group* (RTJEG) foi responsável por desenvolver uma especificação que fornece uma interface para desenvolvimento de aplicações que permite criar, verificar, analisar, executar e gerenciar *threads* Java com restrições temporais (G. BOLLELLA B. BROSGOL; TURNBULL, 2000). A especificação *Safety Critical Java* foi desenvolvida utilizando conceitos e objetos implementados na especificação RTSJ e, por este motivo é apresentada uma breve revisão sobre certos conceitos da RTSJ.

A linguagem de programação Java possui algumas propriedades que tornavam inviável o desenvolvimento de aplicações com enfoque de tempo real, como o gerenciamento de memória automático (*garbage collection)*. A RTSJ foi uma solução encontrada para permitir o desenvolvimento de aplicações de tempo real (ANJOS, 2009). O RTJEG criou e utilizou princípios que delimitaram o escopo do trabalho e impõe requisitos de compatibilidades para especificação em tempo real para Java (G. BOLLELLA B. BROSGOL; TURNBULL, 2000):

- Aplicabilidade para ambientes particulares Java;
- Compatibilidade com versões anteriores;
- *Write Once, run anywhere*;
- Tratar a prática atual do sistema de tempo real mas também permitir futuras implementações;
- Execução previsível;

- Não utilizar extensões sintáticas;
- Permitir variação em decisões de implementação.

Segundo SANTOS (2008), o principal objetivo da RTSJ é estender a linguagem de programação Java e a Máquina Virtual Java (*Java Virtual Machine* - JVM) fornecendo uma interface de programação de aplicativos onde as *threads* Java possuam restrições temporais que possam ser criadas, analisadas, executadas e gerenciadas corretamente. Para aceitação da especificação RTSJ como plataforma viável de desenvolvimento em tempo real é essencial levar em considerações oito áreas de semânticas estendidas e discutidas a seguir.

- Escalonamento: Para garantir a previsibilidade de execuções a RTSJ introduz o conceito de objeto escalonável (*schedulable object)*. A RTSJ requer três classes que são objetos escalonáveis (*ManagedSchedule*): *RealtimeThread, NoHeapRealtimeThread e AsyncEventHandler*. O termo "*scheduling*" ou "*scheduling algorithm*" foi utilizado para referir-se a sequência ou ordem de execução de um conjunto de *threads*.
- Gerenciamento da Memória: A utilização do *Garbage-collection* foi considerada um obstáculo à programação para aplicações de tempo real, pois impõe uma latência imprevisível. Para resolver isso, a RTSJ fornece extensões para que o gerenciamento de memória não interfira os requisitos temporais do sistema.
- Áreas de Memória: A RTSJ introduz um conceito de uma área de memória que possa ser utilizado para atribuições de objetos.
- Sincronização: A utilização do termo prioridade deve ser interpretado de uma forma mais flexível do que o usualmente utilizado, pois termo "*highest priority thread*" apenas indica que a *thread* é a de maior elegibilidade, ou seja, a *thread* que o *dispatcher* escolhe entre todas as outras *threads* que estão prontas para execução. *Threads* aguardando recursos devem ser lançados em uma fila para execução de ordem de elegibilidade.
- Manipulação de Eventos Assíncronos: Um evento assíncrono garante a execução de trechos de código na ocorrência do evento, por exemplo, um evento pode ser uma interrupção de *hardware*.
- Transferência Assíncrona de Controle: Muitas vezes é necessário finalizar uma execução, e para isso, a RTSJ permite a interrupção assíncrona de métodos. Isso facilita o

encerramento antecipado, preservando a segurança do código que não está aguardando interrupções.

- Finalização de Threads assíncrona: A RTSJ possui uma finalização segura assíncrona de *threads* que é uma combinação do tratamento de evento assíncrono e a transferência assíncrona de controle.

- Acesso à Memória Física: A RTSJ fornece classes para que seja possível acessar a memória física através do aplicativo.

- Exceções: A RTSJ introduz diversas exceções e um novo tratamento de exceções em função da transferência assíncrona do controle e alocadores de memória.

2.2 *Safety Critical Java*

Conforme já apresentada na Seção 2.1, a *Real-Time Specification for Java* (RTSJ) foi projetada para atender necessidades gerais de adaptação da linguagem de programação Java para aplicações [1] em tempo real (SANTOS, 2008) e tornou-se cada vez mais desejável a utilização desta linguagem para desenvolver aplicações. Um dos problemas na especificação RTSJ vem da utilização da máquina virtual tanto para aplicações com requisitos temporais [2], como para aplicações sem requisitos temporais.

Com isso, houve uma necessidade de restringir o escopo da RTSJ para o desenvolvimento de sistemas críticos e consequentemente houve a necessidade da criação de uma especificação para satisfazer os processos de validação e certificação mais rigorosos necessários para este tipo de aplicações (GROUP, 2013).

2.2.1 Conceitos de Missões

O conceito de missões é essencial para um projeto de uma aplicação *Safety Critical Java*. Foi disponibilizada a possibilidade de encapsular múltiplas *threads* independentes para controle, ou seja, várias *threads* executando em conjunto e paralelamente, denominadas como *ManagedSchedulables* com acompanhamento de estrutura de dados e comportamento funcio-

[1] Uma aplicação é um sistemas de informática projetado para utilização através de um dispositivo e pode ser desenvolvido em diversas linguagens de programação (TANENBAUM, 2012)

[2] Requisitos temporais são acordos temporais definidos no desenvolvimento de uma aplicação dos quais este aplicativo deverá fornecer uma resposta neste tempo determinado (GROUP, 2013)

nal. Uma aplicação desenvolvida em SCJ consiste de uma ou mais missões com cada Missão representando uma fase operacional diferente da aplicação (GROUP, 2013).

É possível estruturar uma aplicação SCJ com várias missões sendo executadas em simultâneo. A utilização do conceito de missões facilita com que uma complexa aplicação possa ser dividida em vários componentes ativos que podem ser desenvolvidos, certificados e mantidos isolados um do outro. A Missão consiste em um conjunto limitado de objetos escalonáveis limitados que são definidos pela RTSJ (GROUP, 2013). Para cada Missão, um bloco específico de memória é definido chamado *Mission Memory* e os objetos criados nela permanecerão armazenados até que a Missão termine. A noção de missões possui três fases:

- Inicialização (*initialize*), onde todos os objetos são criados;
- Missão (*exec*), que inicia após a inicialização e deixa todos os objetos prontos para execução;
- Terminação (*cleanup*), que é o final da fase Missão e é onde todas as execuções terminam e a aplicação é finalizada.

É possível também existir uma quarta fase chamada recuperação onde as fases de inicialização e Missão são executadas novamente, com o objetivo de executar diversas missões ou até mesmo para se recuperar de uma falha. Existem problemas associados ao uso do Java em um sistema *Safety Critical*, mas os principais são relacionados ao compartilhamento e o gerenciamento de memória (GROUP, 2013).

Conforme apresentada, a Missão inicia em uma fase de inicialização onde os objetos podem ser alocados na memória da Missão (*MissionMemory*) e na memória imortal (*ImmortalMemory*) e sendo instanciados e registrados os manipuladores de eventos periódicos e aperiódicos (*PeriodicEventHandler* e *AperiodicEventHandler*). Quando a inicialização for concluída é iniciada a fase de execução da aplicação, ou seja, a execução de uma Missão definida pelo usuário da aplicação. Na fase de execução é possível acessar objetos na memória da Missão e na memória imortal, mas não será possível alocar novos objetos em tempo de execução.

Cada Missão *Safety Critical Java* é executada sob a responsabilidade de um *MissionSequencer* que tem como tarefa selecionar qual será a próxima Missão a ser executada quando a atual Missão em execução finalizar. Cada *MissionSequencer* é estruturado com um objeto *ManagedSchedulable*. Todos os Níveis da SCJ são compostos por *ManagedSchedulables*.

2.2.2 Inicialização da aplicação do usuário

Uma aplicação SCJ é representada por uma implementação pelo desenvolvedor de uma interface em Java chamada de *Safelet*. A classe da aplicação que implementa o *Safelet* e define o tamanho da memória imortal (*immortalMemorySize*), executa o método *initializeApplication()* onde é possível a alocação de objetos e execução do método *getSequencer()*. O método *getSequencer()* retorna uma referência do *MissionSequencer*, responsável por supervisionar a sequência de missões para execução, contendo uma sequência de uma ou mais missões definidas pelo usuário para execução (GROUP, 2013).

Uma implementação de uma interface *Safelet* deverá implementar a semântica descrita abaixo:

- O objeto *Safelet* é alocado na memória imortal *ImmortalMemoryArea*;
- O método *immortalMemorySize()* do *Safelet* é chamado para determinar o tamanho desejado da área da memória imortal. Se o tamanho real da memória imortal for menor que o valor retornado da *immortalMemorySize*, a inicialização do *Safelet* é abortado imediatamente;
- É chamado o método *initializeApplication()* do *Safelet* para permitir que a aplicação possa alocar estruturas de dados globais no *ImmortalMemoryArea*;
- É chamado o método *getSequencer()* do *Safelet*, com a área de *ImmortalMemoryArea* como o contexto atual de alocação de dados. O valor retornado representa o *MissionSequencer* que está sendo executado, caso o retorno seja nulo, a aplicação é parada imediatamente;
- No método *getSequencer()* não é permitido chamar o método *enterPrivateMemory()*. Caso ocorra, será abortado e apresentada uma exceção *IllegalState-Exception*;
- As exceções geradas pela implementação do *getSequencer()* não serão tratadas durante a execução. O espaço para armazenamento dessas exceções devem ser incluídas na memória da Missão nos parâmetros de armazenamento (*StorageParameters*);
- O mecanismo para especificar o *StorageParameters* para a inicialização do *Safelet* é definida pela implementação;

- O mecanismo para especificar a prioridade que uma *thread* de inicialização *Safelet* é executada em relação às outras *threads* não-Java que podem ser executadas simultaneamente no mesmo *hardware* é definida pela implementação;
- É definida pela implementação a quantidade de memória total que é disponibilizada dentro do ambiente em tempo de execução SCJ para representar o total.

Uma aplicação *Safety Critical Java* é uma sequência de execuções de Missão. Na Figura 2.1 é apresentada a descrição de uma Missão que é composta de inicialização, execução e fases de limpeza (GROUP, 2013).

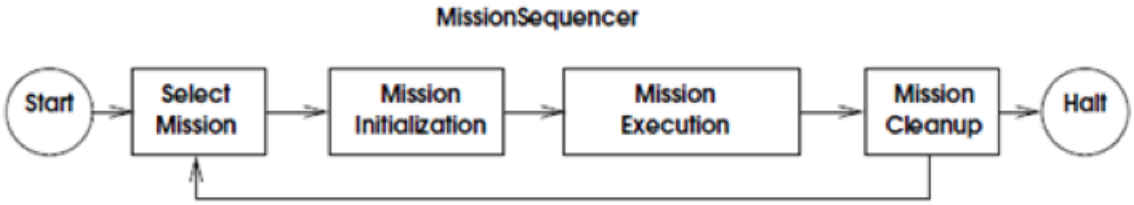

Figura 2.1 – Fases de aplicação Safety Critical (GROUP, 2013).

A inicialização do aplicativo é representada por uma aplicação definida pelo usuário da interface *Safelet* e define o método de configuração que pode criar objetos na área de memória imortal (GROUP, 2013). Também é definido o método *getSequencer()*, que retorna a *MissionSequencer* que representa uma sequência de missões definidas pelo usuário e instancia, registra, aloca e inicializa todos os objetos *ManagedSchedulable* referentes a esta Missão.

2.2.3 Execução de uma Missão

Na inicialização da Missão, o método *initialize()* associado a cada Missão é chamado pelo *MissionSequencer* deve instanciar e registrar todos os objetos *PeriodicEventHandler* (PEH) e *AperiodicEventHandler* (APEH) (*ManagedSchedulable*) atribuídos na área de *MissionMemory* contidos em cada Missão. Cada objeto Missão deve pertencer ao mesmo escopo que a seu *MissionSequencer* (GROUP, 2013).

Após o retorno do método *initialize()*, a SCJ inicia a execução dos manipuladores de eventos periódicos e aperiódicos (PEH e APEH) que estão vinculadas a esta Missão que foram registrados na inicialização. A SCJ oferece uma área de memória privada para cada manipulador de evento periódico chamada *privateMemory* utilizada como área de memória temporária da

thread. Cada manipulador de evento é livre para utilizar áreas de memórias privadas temporárias para armazenar objetos que possuam uma vida menor que a duração da *thread*. Também é possível a alocação de objetos na *ImmortalMemoryArea* ou em áreas *MissionMemory* externa.

A execução continua enquanto todas as *threads* vinculadas aos manipuladores de eventos não terminem. A finalização da execução só será concretizada ao terminar a execução do método *run()* ou ao chamar o método *requestTermination()* da Missão correspondente. Após todos os manipuladores de eventos associados a Missão tenham sidos finalizados a SCJ efetua a chamada do método *cleanup()* para que cada um manipulador de evento seja finalizado. A fase de limpeza pode ser usada para liberar recursos e para restaurar o estado do sistema.

A execução da Missão deverá ser conforme as seguintes etapas detalhadas abaixo:

- Com o *MissionMemory* como contexto de alocação atual o *MissionSequencer* invoca o método de inicialização da Missão desenvolvido pelo usuário. Na sequência é registrado todos os manipuladores de eventos periódicos e aperiódicos (PEH e APEH) que serão registrados dentro da inicialização. Um objeto *StorageParameters* é associado a cada manipulador de evento e deve descrever todos os recursos necessários para a sua execução;

- Durante a execução do método *initializeApplication()*, é possível alocar dados relevantes a Missão na *MissionMemory*. Os objetos podem ser compartilhadas entre objetos escalonáveis da Missão;

- Após o retorno da inicialização e a infraestrutura invoca o método *getSchedule()* da Missão de nível 0 em tempo de execução. É criado um *array* representando todos os objetos manipuladores de eventos que foram registrados pelo método de inicialização e este *array* é passado como parâmetro para o método *getSchedule()* da Missão;

- Cada uma das *threads* vinculada dos manipuladores de evento é iniciada e é reservado memória para o objeto *StorageParameters* de cada manipulador;

- A *thread* do objeto *MissionSequencer* aguarda a execução da Missão finalizar e deverá permanecer bloqueada até o método *requestTermination()* seja chamado ou seja finalizada a execução de todos os manipuladores de eventos;

- Quando a *thread* do manipulador de eventos do *MissionSequencer* detecta que a execução da Missão foi finalizada, confirmando que todas as *threads* vinculadas a ela foram termi-

nadas, ele inicia a organização para a execução dos métodos de limpeza correspondentes para cada uma das instâncias dos objetos;

- São chamados os métodos *cleanup()* através das *threads* do *MissionSequencer*. As chamadas dos métodos *cleanup()* somente ocorrem após a finalização de todas as execuções dos manipuladores de eventos periódicos e aperiódicos (PEH e APEH).

- Após a limpeza de todos os manipuladores de eventos da Missão, o *MissionSequencer* invoca o método *cleanup()* da Missão;

- Após a Missão finalizar sua execução, incluindo a execução do método *cleanup()*, o controle sai da área do *MissionMemory* liberando todos os objetos alocados, incluindo a própria Missão, caso ela tenha sido alocada dentro da área *MissionMemory*;

- Se uma exceção for propagada de uma chamada para inicializar ou de limpeza, a exceção é capturada e ignorada. Se ocorreu na inicialização o *MissionSequencer* deverá executá-la na próxima Missão.

2.2.4 Níveis

Para minimizar a complexidade e custo no desenvolvimento de sistemas *Safety Critical*, a SCJ fornece três níveis de conformidade para a implementação de aplicações. A especificação permite que um aplicativo seja compatível com qualquer um dos três níveis para satisfazer os objetivos mais restritos desde que o desenvolvedor e o fornecedor de infraestrutura SCJ satisfaçam os requisitos da certificação (GROUP, 2013).

O modelo nível 0 é um modelo descrito como uma linha do tempo, baseado em quadros (*frames*) ou de execução cíclica (GROUP, 2013) . Neste modelo os dados são processados de maneira repetitiva durante a Missão, periodicamente em um período de tempo pré determinado (GROUP, 2013). Na Figura 2.2 é apresentada a execução de uma simples aplicação nível 0 utilizando somente uma única Missão juntamente com sua alocação de memória.

São apresentados quatro manipuladores de eventos periódicos (*Periodic Event Handlers* - PEH) com sua memória privada que é limpa após cada início. Cada manipuladores de evento periódico é acionado por um temporizador sob o controle de um *cyclic executive*. O escalonamento é repetido em cima de um período fixo. Não são consideradas as prioridades quando executada em uma implementação de nível 0, pois o escalonamento cíclico é definido especifi-

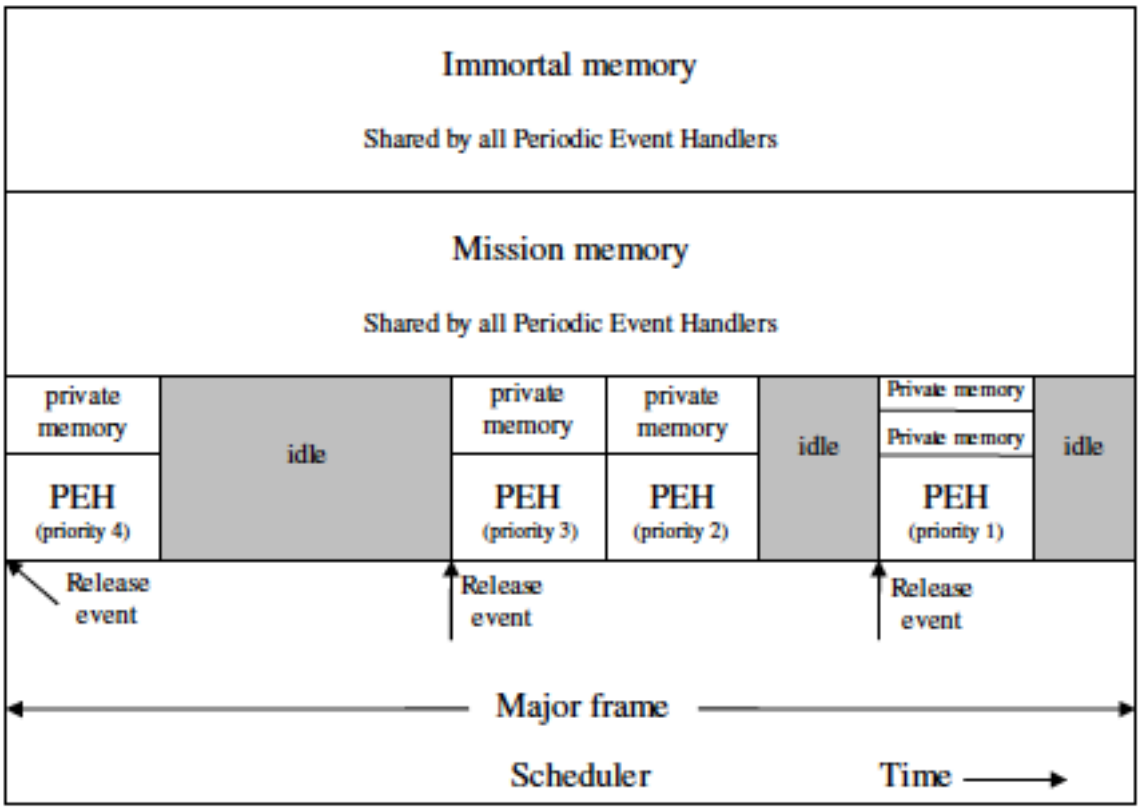

Figura 2.2 – Nível 0 (GROUP, 2013).

camente.

O nível 1 utiliza um modelo de programação multitarefa composto por uma sequência de missões únicas com um conjunto de processamentos simultâneos, cada um com uma prioridade. Para controlar a sequência de missões é utilizado um escalonador preemptivo ou de prioridade fixa (TANENBAUM, 2012). O processamento é realizado através de manipuladores de eventos periódicos (*Periodic Event Handler* - PEH) e/ou manipuladores de eventos aperiódicos (*Aperiodic Event Handler* - APEH). Cada aplicação compartilha objetos na memória da Missão (*mission_memory*) e na memória imortal (*immortal_memory*) entre seus PEHs e APEHs.

Na Figura 2.3 é apresentada a execução de uma aplicação simples, com uma única Missão incluindo sua alocação na memória. São apresentados três objetos escalonáveis (SO1, SO2 e SO3) com uma prioridade e uma área de memória privada que é apagada antes de lançamento. O escalonador de prioridade fixa executa em ordem de prioridade e quando um objeto de maior prioridade fica pronto para execução pode antecipar um objeto de menor prioridade a qualquer momento.

O nível 2 utiliza inicialmente uma única Missão, mas simultaneamente pode criar e

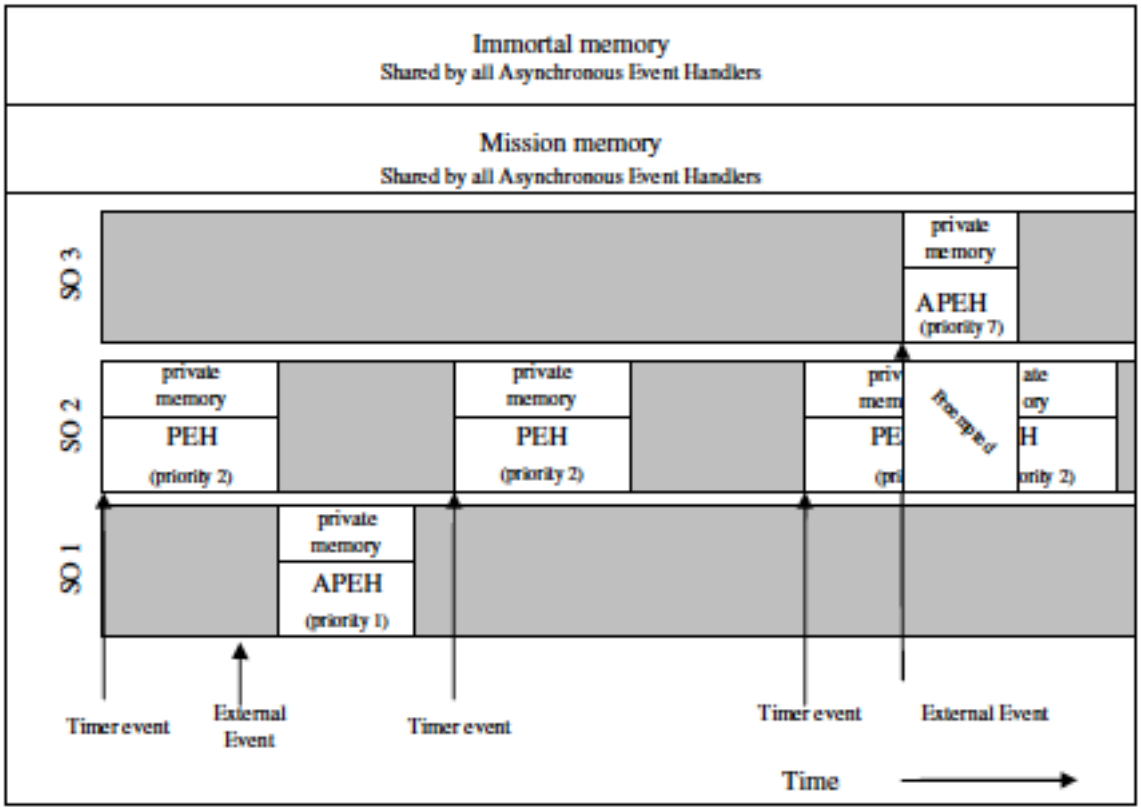

Figura 2.3 – Nível 1 (GROUP, 2013).

executar missões adicionais em conjunto com a Missão inicial. Cada Missão criada tem sua própria memória da Missão e, que também pode criar e executar outras missões. Na Figura 2.4 é ilustrado a execução de uma aplicação simples de nível 2 onde processamento é realizado em conjunto de objetos escalonáveis compostos por manipuladores de eventos periódicos (*Periodic Event Handlers* - PEHs) ou manipuladores de eventos aperiódicos (*Aperiodic Event Handlers* - APEHs). São apresentadas duas missões. A Missão 1 inicia primeiro e contém três objetos escalonáveis. Após é iniciada a Missão 2 que contém dois objetos escalonáveis. As prioridades de cada objeto determina a ordem de execução, independentemente da Missão de cada objeto.

2.2.5 Interação e exemplo de aplicação SCJ de nível 1

A presente dissertação tem como objetivo o desenvolvimento de uma tradução do comportamento da especificação SCJ de nível 1 para a linguagem de programação C++. Dessa forma, nesta seção é apresentado em maiores detalhes a interação entre a máquina virtual SCJ e a aplicação do usuário, descrevendo cada passo de execução do comportamento definido na especificação. Após essa explanação, é apresentado um exemplo base de aplicação de nível 1

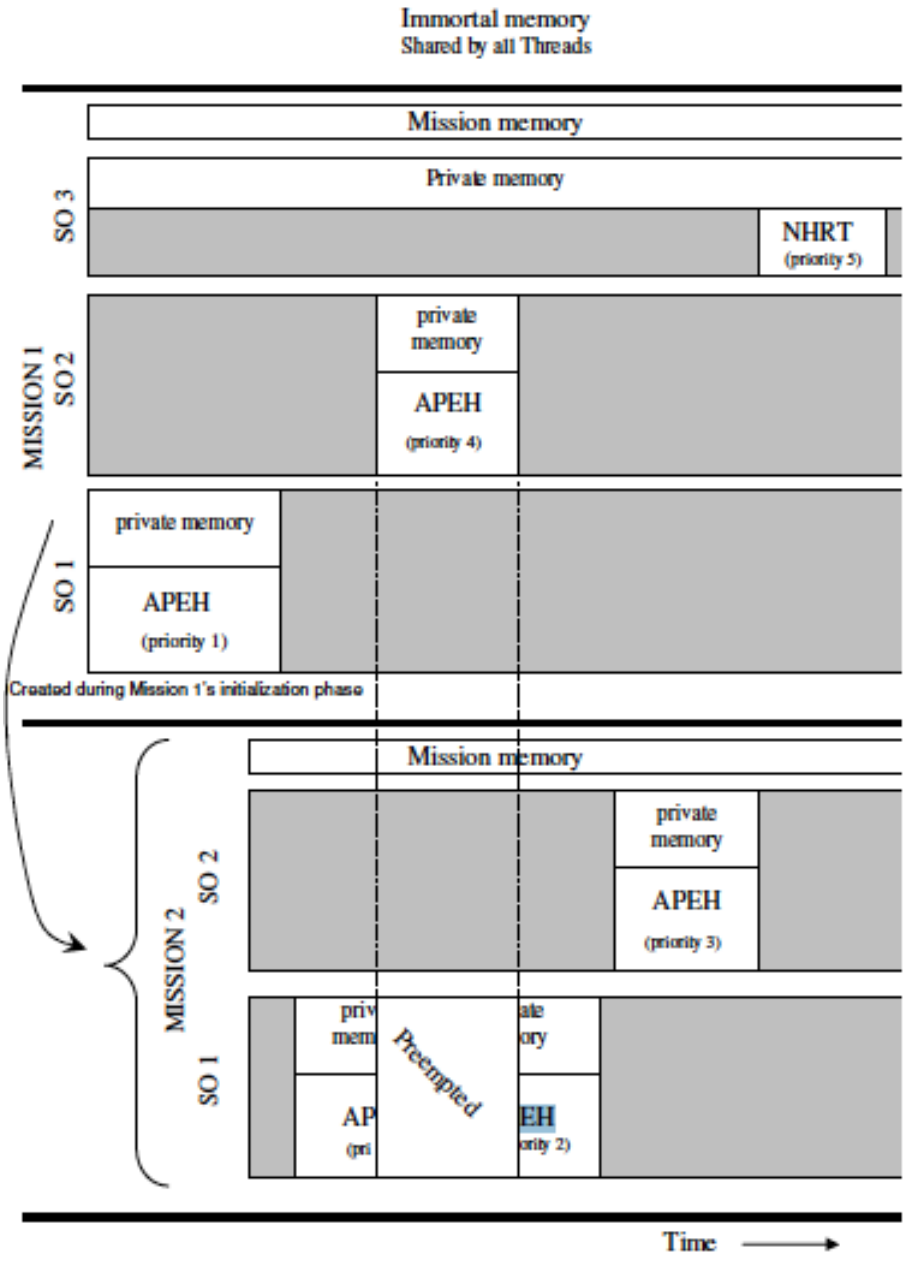

Figura 2.4 – Nível 2 (GROUP, 2013).

que será utilizado no decorrer desta pesquisa.

Na documentação da especificação *Safety Critical Java* são apresentados alguns exemplos de interação do comportamento da máquina virtual SCJ e as aplicações desenvolvidas pelo usuário. Na Figura 2.5 é apresentado o diagrama de sequência com o objetivo de representar a sequência de processos entre os objetos da especificação.

No diagrama são apresentados cinco objetos da aplicação: *Safelet*, *MissionSequencer*, *MissionMemory*, *Mission* e *PeriodicEventHandler*. Neste diagrama não consta manipuladores

de eventos aperiódicos, mas caso existissem, estariam posicionados juntamente com o *PeriodicEventHandler*.

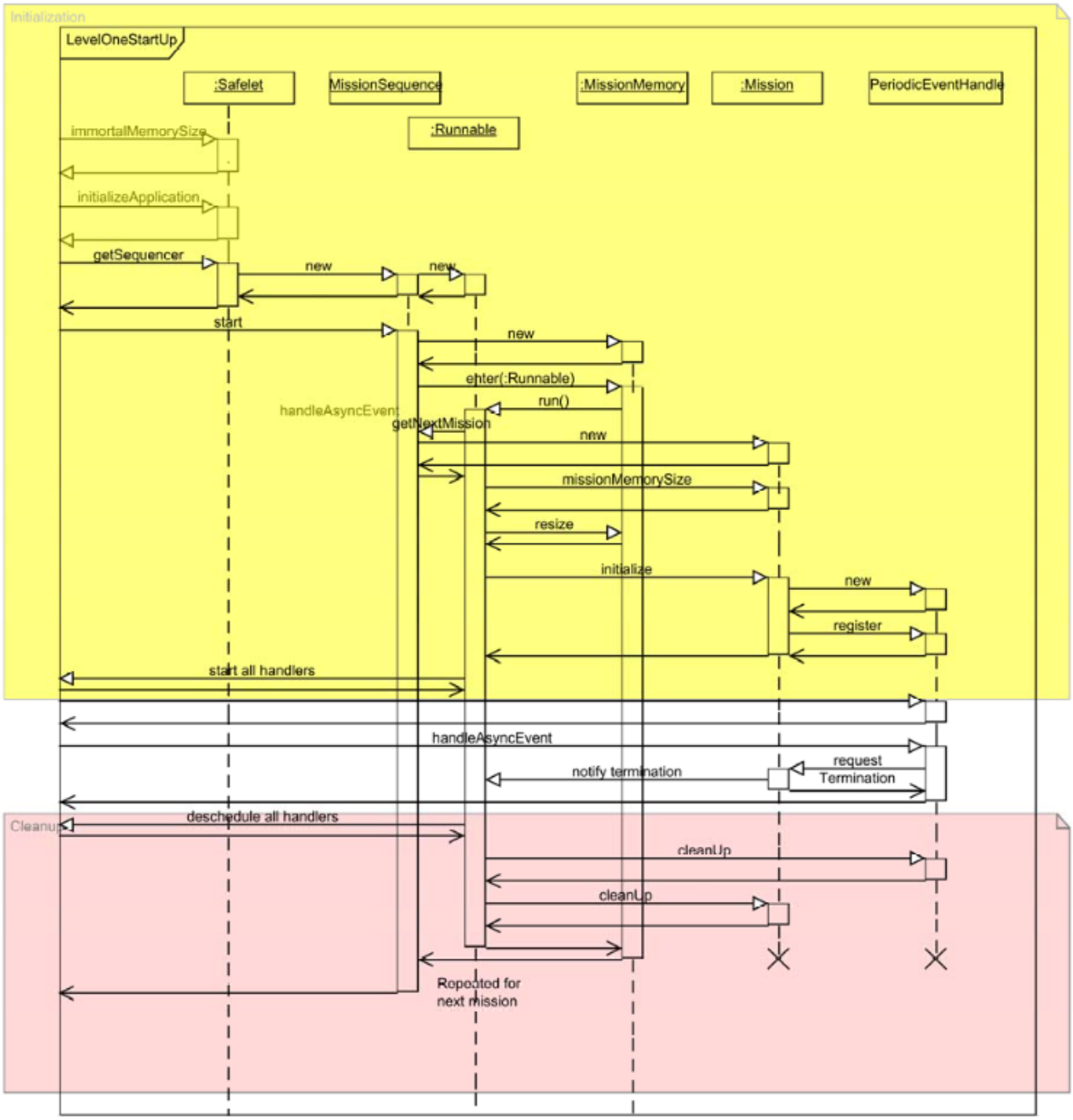

Figura 2.5 – Diagrama de Sequência de uma aplicação nível 1 (GROUP, 2013).

Com o objetivo de ter um exemplo de *software* SCJ que possa ser utilizado didaticamente no restante da dissertação, na Figura 2.6 é apresentada a classe *MySafelet* onde é implementada a interface *Safelet*. Nesta classe é implementado o método *getSequencer()* responsável por criar o *MissionSequencer* para execução conforme apresentado na linha 20 até a linha 25. Existe também os métodos *missionMemorySize()* apresentado na linha 4 responsável por retornar o tamanho da memória Missão, o método *imortalMemorySize()* responsável por retornar o tama-

nho da memória imortal da aplicação na linha 10 e o método *initializeApplication()* na linha 14 responsável por instanciar e alocar memória antes da execução da aplicação caso o usuário identifique a necessidade.

```
1  public class MySafelet implements Safelet {
2        final int MISSION_MEMORY_SIZE = 10000; final int SEQUENCER_PRIORITY = 10;
3
4        public long missionMemorySize()
5        {
6              return MISSION_MEMORY_SIZE;
7        }
8
9        @SCJAllowed(SUPPORT)
10       public long immortalMemorySize() {
11             return 10000;
12       }
13
14       public void intializeApplication()
15       {
16             // do nothing
17       }
18       // Safelet methods
19       @SCJAllowed(SUPPORT)
20       public MissionSequencer<Mission> getSequencer() {
21             // The returned LinearMissionSequencer is allocated in ImmortalMemory
22             return new LinearMissionSequencer<Mission>(
23             new PriorityParameters(SEQUENCER_PRIORITY), new StorageParameters(10000, null),
24             this);
25       }
26 }
```

Figura 2.6 – Classe MySafelet (GROUP, 2013).

Na Figura 2.7 é apresenada a classe estendida do *MissionSequencer*. É implementado o construtor na linha 3 e na linha 8 é apresentado o método *getNextMission()* responsável por retornar uma Missão para a execução e na linha 11 é retornado para sua execução.

```
1  public class MyMissionSequencer extends MissionSequencer {
2        public boolean mission_done;
3        public MainMissionSequencer(PriorityParameters pp, StorageParameters sp) {
4              super(pp,sp);
5              mission_done = false;
6        }
7
8        public MyMission getNextMission() {
9              if (!mission_done) {
10                   mission_done = true;
11                   return new MyMission();
12             }
13             else {
14                   return null;
15             }
16       }
17 }
```

Figura 2.7 – Classe MyMissionSequencer (GROUP, 2013).

Na Figura 2.8 é apresentada a classe estendida de *Mission* chamada *MyMission*. É apresentado o método *initialize()* na linha 4 onde são instanciadas e registradas três instâncias de *PeriodicEventHandlers* nas linhas 5, 6 e 7. Além disso, foi desenvolvida uma adaptação para instanciar e registrar um *AperiodicEventHandler*, conforme a linha 8, com o objetivo de pos-

sibilitar a tradução de todos os objetos envolvidos no nível 1 da especificação apresentada no Capítulo 3.

```
1  public class MyMission extends Mission {
2        private Events events;
3
4        public void initialize() {
5              (new MyPEH("A",new RelativeTime(0,0),new RelativeTime(500,0))).register();
6              (new MyPEH("B",new RelativeTime(0,0),new RelativeTime(1000,0))).register();
7              (new MyPEH("C",new RelativeTime(0,0),new RelativeTime(500,0))).register();
8              (new MyAPEH(events,
9              new PriorityParameters(PriorityScheduler.instance().getMaxPriority()),
10             new StorageParameters(4096, 4096, 4096), events.out)).register();
11       }
12 }
```

Figura 2.8 – Classe MyMission (GROUP, 2013).

No nível 1, o *MissionSequencer* é Linear, o que significa que a sequência de missões para execução devem ser conhecidas e alocadas antes de sua execução. A classe *MyPEH* apresentada na Figura 2.9 é uma classe estendida de *PeriodicEventHandler* que instancia a sua classe pai com os seus parâmetros de prioridade, periódicos e de armazenamento apresentado na linha 5. O método *handleAsyncEvent()* apresentado na linha 11 é responsável pela código que será executado por este manipulador de evento periódico.

```
1  public class MyPEH extends PeriodicEventHandler {
2        static final int priority = 13, mSize = 10000; int eventCounter;
3        String my_name;
4        public MyPEH(String nm, RelativeTime start, RelativeTime period) {
5              super(new PriorityParameters(priority),
6                    new PeriodicParameters(start, period),
7                    new StorageParameters(10000, null), 0);
8                    my name = nm;
9        }
10       @SCJAllowed(SUPPORT)
11       public void handleAsyncEvent() {
12             ++eventCounter;
13       }
14 }
```

Figura 2.9 – Classe MyPEH (GROUP, 2013).

Na Figura 2.10 é apresentada uma implementação da classe estendida de *AperiodicEventHandler*. É instanciado a classe pai passando por parâmetros prioridade, armazenamento, evento e descrição do manipulador de evento aperiódico conforme apresentado na linha 5. Assim como na Figura 2.9 na ilustração do *PeriodicEventHandler*, o método *handleAsyncEvent()* é responsável pelo código que será executado pelo objeto, porém, para manipuladores de eventos aperiódicos somente são executados ao ser chamado o método *fire()*. Esta classe foi implementada com o objetivo de ser utilizada como exemplo na tradução no restante da dissertação.

```
1  public class MyAPEH extends AperiodicEventHandler {
2    private Events events;
3
4    public MyAPEH(PriorityParameters priority,
5          StorageParameters storage, AperiodicEvent event) {
6      super(priority, storage, event, "MyAPEH");
7    }
8
9    @SCJAllowed(SUPPORT)
10   public void handleAsyncEvent() {
11         ++eventCounter;
12   }
13 }
```

Figura 2.10 – Classe MyAPEH (GROUP, 2013).

2.3 POSIX

O POSIX (*Portable Operating System Interface*) é um conjunto de normas no qual é descrito um conjunto de serviços fundamentais para a construção eficiente de uma aplicação (GROUP, 2007). O acesso a estes serviços é fornecido pela definição de uma biblioteca utilizando a linguagem de programação C. O seu principal objetivo é permitir que programadores possam desenvolver aplicações portáteis para diversas arquiteturas (GROUP, 2007). É desenvolvido e mantido pelo grupo Austin, formado por membros do Comitê de normas de aplicações portáteis da IEEE (*IEEE Portable Applications Standards Committee*), do *The Open Group* e comitê técnico da ISO/IEC. Embora tenha surgido para se referir a norma IEEE Std 1.003,1-1988, refere-se IEEE Std 1003.n e partes da ISO/IEC 9945 (GROUP, 2007).

O POSIX define uma interface padrão do sistema operacional e ambiente, incluindo um interpretador de comandos e programas utilitários para apoiar a portabilidade de aplicações ao nível de código fonte (GROUP, 2007). A sua utilização nesta dissertação é motivada através da portabilidade de aplicações, podendo ser utilizada em qualquer versão de sistema baseado em UNIX, como por exemplo, RTLinux (RT LINUX, 2014), Debian (DEBIAN OS, 2014) entre outras distribuições como o Solaris (SOLARIS OS, 2015). O POSIX compreende quatro componentes principais:

- Termos gerais, conceitos e interfaces comuns a todos os volumes de POSIX.1 200x;
- Definições para funções e sub-rotinas de serviço de sistema, serviços do sistema de linguagem específica para a linguagem de programação C, incluindo tratamento e recuperação de erros;
- As definições para a interface de origem para comandar serviços de interpretação (*shell*) e programas utilitários;

- Lógica estendida que não se adequam nos volumes anteriores estão inclusos no volume de Justificativa.

Além destes componentes, O POSIX dispõe de três versões onde é possível utilizar funções distintas (GHOSH; MUKHERJEE; SCHWAN, 1994):

- 1: Serviço de núcleo, incorporado ao padrão ANSI (American National Standards Institute) C que contém criação e controle de processos, sinais, entrada e saída, temporizadores, instruções ilegais entre outros;
- 1b: Extensão *Real-Time* que possuem os escalonamentos de prioridade, sinais de tempo real, relógios e temporizadores, memória compartilhada, entrada e saída síncronas e assíncronas, bloqueio de memória entre outros;
- 1c: Extensão de *Threads* que permite a criação, controle e limpeza de *threads*, escalonamento e sincronização de *threads* entre outras.

2.3.1 *Thread* POSIX

Existem diversas versões de *threads* desenvolvida por fornecedores de *hardware* que as diferem entre si, tornando a portabilidade uma preocupação para desenvolvedores de *software*. Cada fornecedor possui uma versão distinta e com isso não possibilitando uma portabilidade entre as arquiteturas. É também possível utilizar *threads* na linguagem de programação Java onde também é possível obter uma portabilidade, mas para que isso ocorra é necessário uma *Java Virtual Machine* (JVM).

Porém, para sistemas baseados em UNIX é possível utilizar *threads* POSIX, ou simplesmente *pthreads* com o objetivo de obter as vantagens dos recursos fornecidos. São definidas como um conjunto de tipos e de procedimentos implementados na biblioteca *ptread.h*. Com a utilização de *pthreads* é possível criar e gerenciar uma *thread* obtendo um melhor desempenho, ou seja, com uma menor sobrecarga do sistema operacional (BARNEY; LIVERMORE, 2014).

Para a comunicação entre *pthreads* não existe cópia de memória intermediária pois é compartilhada o mesmo endereço de memória em um único processo. É possível trabalhar com prioridades e escalonamento de tempo real onde as tarefas de maiores prioridades executem anteriormente que as demais (GROUP, 2007).

Na Figura 2.11 é apresentado um simples exemplo de criação e finalização de duas *pthreads*. Na *main* é declarada uma *ptread* onde são criadas duas *pthreads*. Para criação da

pthread é utilizada a chamada da função *pthread_create* que passa por parâmetro a *thread* que está sendo criada, os atributos da criação (se for NULL será criado com os atributos padrões), a rotina que será executada pela *thread* criada e por último, um argumento (único) que pode ser passado para a rotina que será executada após a criação. Para passar mais de um argumento para a rotina é necessário utilizar uma estrutura de dados.

Após a criação da *pthread* é executada a função "Ola" que imprime na tela a frase "Ola Mundo" com o id da *pthread*. Após isso é finalizada a execução da *pthread* através da chamada da função *pthread_exit*. Caso ocorra algum erro na criação da *pthread*, será apresentada na tela uma mensagem com o erro armazenada na variável "rc".

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void *Ola(void *id){
5        long tid;
6        tid = (long)id;
7        printf("Ola␣Mundo!␣%ld!\n", tid);
8        pthread exit(NULL);
9  }
10
11 int main (int argc, char *argv[]){
12       pthread_t threads[2];
13       int rc;
14       long i;
15       for(i=0; i<2; i++){
16             printf("Na␣Main:␣criando␣a␣thread␣%ld\n", i);
17             rc = pthread_create(&threads[i], NULL, Ola, (void *)i);
18             if (rc){
19                   printf("ERRO\n");
20                   exit(-1);
21             }
22       }
23       pthread exit(NULL);
24 }
```

Figura 2.11 – Exemplo de criação e finalização de uma *pthread* adaptado de BARNEY; LIVERMORE (2014).

2.3.1.1 Encapsulamento de *Pthread* em C++

Conforme apresentado, o POSIX *thread* foi desenvolvido com o objetivo de ser utilizado na linguagem de Programação C. Porém, a proposta desta dissertação é uma tradução da especificação *Safety Critical Java* em C++. Para uma *pthread* ser utilizada na linguagem de programação orientada a objetos C++ é necessário criar um objeto abstrato que oculta a complexidade da *pthread* com o objetivo de apresentar uma interface para o usuário.

Na Figura 2.12 é apresentado um cabeçalho exemplo de uma classe *Thread* onde são declarados os métodos e atributos de uma *thread* encapsulada em C++(BARNEY; LIVERMORE, 2014). A classe *Thread* é uma classe virtual onde existe um método *run()* que deve ser implementado pelo usuário.

```
1  class Thread {
2        private:
3              pthread_t _id;
4              Thread(const Thread& arg);
5              Thread& operator=(const Thread& rhs);
6        protected:
7              bool started;
8              void *arg; static void *exec(void *thr);
9        public:
10             Thread();
11             virtual ~Thread();
12             unsigned int tid() const;
13             void start(void *arg = NULL);
14             void join();
15             virtual void run() = 0;
16 };
```

Figura 2.12 – Definição da classe *Thread* (BHASKAR, 2008).

A função *pthread_create* responsável pela criação da *thread* necessita de um ponteiro para uma função global. A classe *Thread* é uma abstrata por possui um método *run() virtual*. Na Figura 2.13 é apresentada a classe *Thread* que possui o construtor, destrutor, método *start()* que é responsável pela criação da *pthread* (*pthread_create*), método *join()* e na linha 18 é apresentado o método *exec()* responsável pela chamada do método virtual *run()* da *pthread* encapsulada.

```
1  Thread::Thread() {
2        cout << "Thread::Thread()" << endl;
3  }
4  Thread::~Thread() {
5        cout << "Thread::~Thread()" << endl;
6  }
7  void Thread::start(void *arg) {
8        int ret;
9        this->arg = arg;
10       if ((ret = pthread create(&_id, NULL, &Thread::exec, this)) != 0) {
11             cout << strerror(ret) << endl;
12             throw "Error";
13       }
14 }
15 void Thread::join() {
16       pthread_join(_id, NULL);
17 }
18 void *Thread::exec(void *thr) {
19       reinterpret_cast<Thread *> (thr)->run();
20 }
```

Figura 2.13 – Classe *Thread* (BHASKAR, 2008).

Após a definição da classe, é necessário estender a classe *Thread* e criar um método *run()* que será responsável pela execução da rotina principal da *pthread*. Na Figura 2.14 é apresentada a classe *MyThread* que estende a classe *Thread* e possui um método *run()* que é responsável pela execução do código do usuário.

```
1  class MyThread : public Thread {
2        public:
3              void run() {
4              // Codigo do usuario
5              }
6  };
```

Figura 2.14 – Classe estendida *MyThread* (BHASKAR, 2008).

Na Figura 2.15 é apresentada a classe principal da aplicação contendo a declaração, incialização e finalização de uma *thread*. O método *start()*, conforme apresentado na Figura 2.13, efetua a criação de uma *thread*. O método inicial de execução da *pthread* é o método *exec()* que faz a chamada do método *run()* que é escrito pelo usuário.

```
1  int main() {
2        int x = 10;
3        MyThread thr;
4        thr.start(&x);
5        thr.join();
6  }
```

Figura 2.15 – Classe principal do exemplo (BHASKAR, 2008).

2.3.1.2 Periodicidade

Para o tipo de sistema abrodado nesta dissertação, é necessário tratar de tempo em tarefas, mais especificamente incluir períodos de execução para tarefas. Por este motivo, neste trabalho é necessário utilizar a periodicidade em *pthread* para obter um comportamento periódico para a execução de tarefas. Para isso, é necessário utilizar temporizadores, e em sistemas baseados em UNIX existem diversas interfaces de temporizadores diferentes desenvolvidas ao longo dos anos (SIMMONDS, 2009).

Um dos métodos desenvolvidos é POSIX *timers* que funciona através da geração de sinais (*signals*) POSIX. Se existir diversas *threads* dentro de um processo, cada uma deverá usar um sinal distinto para distingui-las. Os sinais que deverão ser utilizados são a partir de SIGRTMIN(33) até SIGRTMAX(64) (SIMMONDS, 2009). Na Figura 2.16 é apresentado o método *wait_period()* que recebe por parâmetro uma estrutura de dados contendo as informações da períodicidade. Na linha 3 é chamada o método *sigwait()* onde a execução é bloqueada e fica aguardando um sinal *sig* para o seu desbloqueio. Ao receber o sinal, a *thread* é desbloqueada e retorna para a execução.

```
1  static void wait_period (struct periodic info *info){
2        int sig;
3        sigwait (&(info->alarm_sig), &sig);
4        info->wakeups_missed += timer_getoverrun (info->timer_id);
5  }
```

Figura 2.16 – Função *wait_period* (SIMMONDS, 2009).

O método *make_periodic()* apresentado na Figura 2.17 faz com que a execução tenha um comportamento periódico através da criação de um *timer*. Inicialmente são inicializados os sinais para a utilização. Após é criado uma máscara de sinal (*signal mask*) que será utilizado no método *wait_period()*. Os tempos utilizados neste métodos devem ser fornecidos em milisegundos. Na linha 25 é criado o *timer* que irá gerar os sinais escolhidos e, por fim, na linha 35 o *timer* é feito periódico e retornado para classe a qual foi chamado.

```
1  static int make_periodic (int unsigned period, struct periodic info *info) {
2        static int next_sig;
3        int ret;
4        unsigned int ns;
5        unsigned int sec;
6        struct sigevent sigev;
7        timer_t timer_id;
8        struct itimerspec_itval;
9
10       if (next_sig == 0)
11             next_sig = SIGRTMIN;
12       if (next_sig > SIGRTMAX)
13             return -1;
14       info->sig = next_sig;
15       next_sig++;
16
17       info->wakeups_missed = 0;
18
19       sigemptyset (&(info->alarm_sig));
20       sigaddset (&(info->alarm_sig), info->sig);
21
22       sigev.sigev_notify = SIGEV_SIGNAL;
23       sigev.sigev_signo = info->sig;
24       sigev.sigev_value.sival ptr = (void *) &timer_id;
25       ret = timer_create (CLOCK_MONOTONIC, &sigev, &timer_id);
26       if (ret == -1)
27             return ret;
28
29       sec = period/1000000;
30       ns = (period - (sec * 1000000)) * 1000;
31       itval.it_interval.tv_sec = sec;
32       itval.it_interval.tv_nsec = ns;
33       itval.it_value.tv_sec = sec;
34       itval.it_value.tv_nsec = ns;
35       ret = timer_settime (timer_id, 0, &itval, NULL);
36       return ret;
37 }
```

Figura 2.17 – Função *make_periodic* (SIMMONDS, 2009).

Estas funções são definições estáticas para resolver a necessidade de periodícidade. Neste trabalho serão adaptadas estas funções e encapsuladas no objeto *PeriodicEventHandler* para que seja possível obter o comportamento periódico conforme apresentado na especificação *Safety Critical Java*

2.4 Trabalhos Relacionados

No trabalho de CAVALCANTI et al. (2011) foi apresentada uma proposta de utilização da especificação *Safety Critical Java* na linguagem Circus. Circus é uma linguagem para refinamento baseado em uma combinação flexível de elementos de Z, CSP e os comandos imperativos de cálculos de Morgan. A semântica da linguagem é baseada nas Teorias de Unificação de Programação (UTP) e tem sido utilizado para definir orientação à objetos e tempo. Foi proposta uma abordagem para desenvolvimento gradual de programas SCJ baseados em modelos abstratos que não consideram os detalhes de missões ou modelos de memória. A especificação Circus caracteriza as etapas do desenvolvimento: Âncoras, como é identificado objetivos intermediários para refinamento e aspectos de design tratados em cada etapa. Cada âncora é desenvolvida utilizando uma combinação diferente de notações da família Circus. A primeira âncora é a especificação de alto nível e a última é próximo de uma aplicação SCJ para permitir a geração automática de código. É escrito em SCJ-Circus, uma nova versão de Circus estendido com construções que correspondem aos componentes do paradigma de programação SCJ.

Em STRØM; PUFFITSCH; SCHOEBERL (2013) foi proposta uma análise de opções para bloqueio de arquiteturas multicore no contexto SCJ. O acesso a recursos compartilhados em sistemas multicore é protegido por um mecanismo de bloqueio de software implementados através de operações atômicas e podendo acarretar em sobrecarga de sincronização, o que, em um sistema de tempo real, pode ocasionar o aumento do tempo de execução e resposta e podendo anular a previsibilidade de execução de um conjunto de tarefas. Portanto, é apresentado um mecanismo de bloqueio de hardware para reduzir a sobrecarga de sincronização e implementada para a versão do multiprocessador da *Java Optimized Processor* (JOP) no contexto de *Safety Critical Java*.

No trabalho de SØNDERGAARD; KORSHOLM; RAVN (2012) foi apresentado o desenvolvimento de uma aplicação do perfil *Safety Critical Java* (SCJ), direcionado para plataformas embarcadas *low-end* com 16 kB de RAM e 256 kB de memória flash para armazenamento. É proposto a utilização de execução através de uma máquina virtual Java para embarcados chamada HVM, com um kernel *bare metal* implementando objetos de hardware, manipuladores de interrupção de primeiro nível, variáveis nativas, e uma infraestrutura escrita em Java. A máquina virtual HVM permite que a implementação seja portado para diferentes plataformas embarcadas possuam um compilador C como parte do ambiente de desenvolvimento. A abor-

dagem de *bare metal* elimina a necessidade de um consumo de recursos do sistema operacional ou da biblioteca C. A implementação SCJ deste trabalho é avaliada com um ponto de referência conhecido e apresentando uma possível redução de tamanho para que seja possível executar em uma configuração mínima.

Em LUCKOW; THOMSEN; KORSHOLM (2014) é apresentada a HVMTP, uma máquina virtual Java (JVM) portátil e de tempo previsível para aplicações com sistemas embarcados com recursos limitados de tempo real *hard* que implementa a especificação *Safety Critical Java* (SCJ) de nível 1. Na JVM a previsibilidade do tempo é adquirida por uma combinação de algoritmos de tempo previsível explorando modelo de programação e conhecimentos estáticos do sistema SCJ hospedado na JVM. São apresentados os termos, capacidades, modelo de escalonamento, e pior caso de tempo de execução e o melhor caso.

No trabalho de PIZLO; ZIAREK; VITEK (2009) é apresentada uma máquina virtual para embarcados de tempo real Fiji VM. São apresentadas cinco contribuições do trabalho comparado com o estado da arte. A primeira contribuição é a possibilidade de execução uma implementação Java em tempo real de alto desempenho de tamanho tamanho reduzido mas robusto o suficiente para arrancar a partir de *bare metal*. A segunda é a primeira aplicação de alto desempenho em tempo real Java que pode ser executado com manipulação de interrupção rotinas escritas em Java. A terceira contribuição do trabalho é um levantamento das limitações enfrentadas por máquinas virtuais Java quando destinada a sistemas de tempo real rígido. A quarta contribuição apresentada no trabalho é uma descrição concisa do sistema compilador e do *runtime* da Fiji VM. E para finalizar, a quinta contribuição é um comparativo de *benchmark* da Fiji VM e o estado da arte de outras JVMs.

No trabalho de A. PLSEK L. ZHAO; VITEK (2010) é apresentada uma implementação de uma aplicação utilizando a especificação *Safety Critical Java*. A aplicação desenvolvida foi de nível 0, ou seja, uma aplicação sequencial seguindo os conceitos definidos pela especificação. É apresentada uma descrição de alto nível utilizando a biblioteca chamada oSCJ e a máquina virtual Java chamada OVM. Segundo o autor, existem dificuldades ainda de encontrar diferentes máquinas virtuais para aplicações *Safety Critical*. Em cima da aplicação desenvolvida foram criados relatórios de desempenho em comparação a uma aplicação desenvolvida na linguagem de programação C executadas sob diferentes arquiteturas e sistemas operacionais de tempo real. Segundo os relatórios, a proposta alcançou um grau de previsibilidade e desempenho aproximado ao de uma aplicação desenvolvida em C.

Além destes trabalhos relacionados, foram pesquisados também tradutores de códigos desenvolvidos na linguagem de programação Java para C++ com o objetivo de testar os tradutores existentes com códigos desenvolvidos utilizando a especificação. O trabalho de TRANSCOMPILER (2014) é um tradutor que converte um código desenvolvido em Java para linguagem C. O código C gerado pelo tradutor é preparado para ser compilado pelo C65. Este trabalho se tornou inviável pois não traduzia códigos orientado objetos.

No trabalho de J2C (2014) é apresentada uma bilbioteca para eclipse onde é possíve traduzir um código desenvolvido na linguagem de programação Java para C++ orientado a objetos. Por outro lado, foi possível traduzir apenas os códigos exemplos disponiveis no site para *download*. Foi efetuado testes em diversos outros códigos de exemplo onde não foi possível compilar o código traduzido.

2.4.1 Considerações sobre os Trabalhos Relacionados

Esta dissertação tem como proposta a tradução do comportamento de objetos da *Safety Critical Java* na linguagem C++ de programação e por esse motivo, possui um diferencial a todos os trabalhos relacionados apresentados neste capítulo como a possibilidade de portabilidade para diferentes arquiteturas baseadas em UNIX. Uma aplicação desenvolvida em C++ possui portabilidade pois não é necessário a execução em cima de uma Máquina Virtual Java (JVM). A linguagem de programação C++, assim com o Java, é uma linguagem de programação orientada a objetos e como apresentado na Seção 2.3.1.1, é possível utilizar POSIX *Threads* em C++ encapsulando-as.

O trabalho de CAVALCANTI et al. (2011) possui uma proposta de tradução da SCJ na linguagem de programação Circus. Auxiliou o desenvolvimento deste trabalho, por mais que utilize uma sintaxe diferente, auxiliou a compreensão principalmente do comportamento dos objetos definidos pela SCJ através da apresentação de modelos definidos pelo Circus.

Conforme apresentado nos trabalhos de LUCKOW; THOMSEN; KORSHOLM (2014) e PIZLO; ZIAREK; VITEK (2009), as máquinas virtuais não possuem uma portabilidade para diversos tipos de arquiteturas, cada arquitetura necessita de uma implementação diferente. Isso ocasiona um desafio para desenvolvedores SCJ, pois é necessário escolher a arquitetura de execução antes de iniciar o desenvolvimento e caso a arquitetura seja alterada é possível que seja necessário uma adaptação do código desenvolvido da aplicação.

No trabalho de A. PLSEK L. ZHAO; VITEK (2010) foi apresentado o desenvolvimento

de uma aplicação utilizando a especificação SCJ de nível 0. Neste trabalho o autor utiliza uma máquina virtual Java chamada OVM e ainda cita a dificuldade de encontrar máquinas virtuais para diferentes arquiteturas e sistemas operacionais. Foram apresentados relatórios de desempenho e verificou-se que o desempenho de uma aplicação desenvolvida em SCJ de nível 0 possui um desempenho próximo de uma aplicação desenvolvida em C e após testes de perda de *deadline* a diferença entre as aplicações ficou em menos de 0,05 milisegundos.

Nos trabalhos J2C (2014) e TRANSCOMPILER (2014) foram desenvolvidos com o objetivo de traduzir códigos desenvolvidos em Java para a linguagem de programação em C++. Em ambos não foi possível traduzir os objetos definidos pela especificação SCJ, porém é de importância significativa neste trabalho a possível utilização destas ferramentas para a tradução do código do usuário desenvolvido em Java para a linguagem de programação em C++.

2.5 Conclusão

Neste capítulo foi apresentado o referencial teórico desta dissertação onde é possível visualizar conceitos da especificação *Real-Time Specification for Java* utilizados para implementar o *Safety Critical Java*. Após foi apresentado a especificação *Safety Critical Java* com suas características, requisitos, semânticas e um exemplo de código de uma aplicação SCJ de nível 1. Foi apresentada a definição e as características sobre POSIX, POSIX Threads, utilização de *pthreads* em C++ e periodicidade de *pthreads* necessária para desenvolver a tradução da especificação SCJ na linguagem C++. E, por fim, foram apresentados os trabalhos relacionados desta dissertação e uma discussão sobre eles.

3 ESTRATÉGIA DE TRADUÇÃO

Conforme apresentado anteriormente nesta dissertação (na Seção 2.2), na especificação *Safety Critical Java* (SCJ) foi implementado o conceito de missões. O conceito de missões é essencial para um projeto de uma aplicação SCJ. Uma aplicação desenvolvida utilizando a especificação *Safety Critical Java* consiste de uma ou mais missões. É possível encapsular múltiplas *threads* independentes para controle e acompanhamento da estrutura de dados e do comportamento funcional. Cada Missão representa uma diferente fase operacional da aplicação.

É possível estruturar uma aplicação *Safety Critical Java* com várias missões sendo executadas. A utilização do conceito de missões facilita com que uma complexa aplicação possa ser dividida em vários componentes ativos que podem ser desenvolvidos, certificados e mantidos isolados um do outro (GROUP, 2013).

O nível 0 é um modelo de linha do tempo baseado em quadros (*frames*) ou de execução cíclica. Neste modelo os dados são processados de maneira repetitiva durante a Missão, periodicamente, em um período de tempo pré-determinado. No nível 1 é utilizado um modelo de programação multitarefa composto por uma sequência de missões únicas com um conjunto de processamentos simultâneos, cada um com uma prioridade. Já o nível 2 utiliza inicialmente uma única Missão, mas simultaneamente pode criar e executar missões adicionais em conjunto com a Missão inicial.

Após uma análise da especificação SCJ considerando as funcionalidades para executar aplicações em sistemas embarcados utilizando a linguagem de programação C++, essa pesquisa focou na tradução do nível 1 da SCJ. O nível 0, por sua execução não concorrente, não foi selecionado para tradução. É importante salientar que, não é foco deste trabalho a tradução de escopos de memória em SCJ pois, diferentemente da linguagem de programação Java, a linguagem de programação C++ não possui coletor de lixo.

3.1 Componentes da SCJ

Nesta seção são apresentados os componentes da SCJ que fazem parte do escopo da tradução, descrevendo as características e comportamento de cada componente. Na Tabela 3.1 é apresentada uma breve descrição dos componentes a serem traduzidos.

Tabela 3.1 – Componentes SCJ considerados na tradução

Nome	**Descrição**
Safelet	Interface em Java responsável pela implementação da aplicação e pela chamada do método *getSequencer()* que retorna uma sequência de missões para execução.
MissionSequencer	Um objeto da classe *ManagedEventHandler* responsável por armazenar, supervisionar a execução das missões pertencentes a uma instância desta classe.
Mission	Composto por um ou conjunto limitado de objetos escalonáveis (*ManagedSchedulable*) definidos pela RTSJ. Cada objeto escolonável é uma *thread* independente de controle.
PeriodicEventHandler	É uma instância do objeto *BoundAsyncEventHandler* (*thread*) responsável pela execução automática de eventos periódicos através do método *handleAsyncEvent()*.
AperiodicEventHandler	Um objeto *BoundAsyncEventHandler* (*thread*) onde é executado a rotina atribuida ao método *handleAsyncEvent()* após cada chamada do método *fire()*.

3.1.1 *Safelet*

Cada aplicação *Safety Critical Java* (SCJ) consiste em uma ou mais missões executadas simultaneamente ou em sequência. Cada aplicação é representada pela implementação de uma interface *Safelet* onde é instanciado o *MissionSequencer* responsável pela execução de uma sequência de missões que compõem a aplicação SCJ. O método *getSequencer()* é chamado para obter o retorno de um objeto *MissionSequencer* que supervisiona a execução de missões da aplicação. Um elemento SCJ utiliza uma *thread* independente para iniciar a sua execução e aguarda a finalização da execução para terminar a *thread*.

3.1.2 *MissionSequencer*

O *MissionSequencer* é responsável por supervisionar uma sequência de execuções missões. A sequência pode incluir execução de missões independentes intercaladas e execução de missões repetidas. É uma subclasse de um *ManagedEventHandler* e é ligado a uma *thread* de manipulação de eventos. A prioridade de execução e memória da *thread* são especificadas por parâmetros no momento da construção. Executa código desenvolvido pelo fornecedor de infraestrutura que invoca implementações definidas pelo usuário do *MissionSequencer* (*getNextMission()*, *initialize()*, *cleanUp()*). O MissionSequencer permanece bloqueado durante a execução de uma Missão pertencente a ele até que a mesma finalize a execução. Uma chamada

do método *requestSequenceTermination()* irá desbloquear esta *thread* e aguardar a chamada do método *requestTermination()* da Missão em execução.

O método *getNextMission()* é chamado para selecionar a Missão inicial de execução, e posteriormente, para determinar a próxima Missão de executar cada vez que uma Missão é finalizada. Caso o retorno seja nulo, não há mais missões para serem executadas sob o controle deste *MissionSequencer*.

3.1.3 *Mission*

Cada Missão é uma implementação de uma classe abstrata definida pela SCJ e é composta por um ou mais objetos *ManagedSchedulable*, ou seja, *threads* independentes de controle e com dados compartilhados entre as *threads*. O construtor aloca e inicializa estruturas de dados associados a uma Missão e pode atribuir objetos de infraestruturas adicionais dentro da mesma *MemoryArea*. A quantidade de memória alocada na Missão é definida pela implementação do usuário.

3.1.4 *PeriodicEventHandler*

O *PeriodicEventHandler* permite a execução automática de eventos periódicos. O método *handleAsyncEvent()* responsável pela execução do manipulador de evento possui um comportamento periódico, como se existisse um *timer* de evento periódico. Os parâmetros passados para o construtor são aqueles que serão utilizados pela infraestrutura.

3.1.5 *AperiodicEventHandler*

É responsável pela execução automática de código que está vinculado a um evento aperiódico. É uma classe abstrata e deve implementar o método *handleAsyncEvent()* responsável pela execução do manipulador de eventos aperiódicos. Assim como *PeriodicEventHandler*, os parâmetros passados para o construtor são aqueles que serão utilizados pela infraestrutura.

3.2 Traduzindo os componentes SCJ para C++

Nesta seção será efetuada a tradução de um exemplo simples de um código *Safety Critical Java* apresentado na Seção 2.2 retirado e adaptado da documentação da especificação SCJ. Este exemplo é composto por dois objetos manipuladores de eventos periódicos (*PeriodicE-*

ventHandlers) e um terceiro objeto manipulador de eventos aperiódico (*AperiodicEventHandler*).

A tradução motiva-se, principalmente, pela questão de não necessitar de uma máquina virtual para execução da aplicação. Portanto, foi necessário estudar uma forma de substituir o comportamento da máquina virtual na aplicação, dividindo a tradução em duas partes onde:

- 1ª parte: Será responsável pela estrutura da SCJ/*Java Virtual Machine* (JVM);
- 2ª parte: É a tradução do código do usuário.

Na Tabela 3.2 são apresentadas as equivalências estudadas para traduzir cada componente do *Safety Critical Java* conforme os elementos apresentados na Tabela 3.1 para a linguagem de programação C++.

Tabela 3.2 – Descrição de comportamento de componentes SCJ em C++

Nome	**Descrição em C++**
Safelet	Implementação da aplicação do usuário e responsável por simular o comportamento da JVM.
MissionSequencer	Uma *pthread* encapsulada em C++ responsável por supervisionar a execução da(s) missão(ões).
Mission	Cada Missão será composta por uma *pthread* encapsulada em C++ para controle dos manipuladores periódicos e aperiódicos.
PeriodicEventHandler	Uma *pthread* encapsulada em C++ com comportamento periódico implementado através de sinais POSIX apresentada na Seção 2.3.1.2.
AperiodicEventHandler	Uma *pthread* encapsulada em C++ com a utilização de *MUTEX (Mutual Exclusion)* para efetuar o bloqueio da sua execução. A execução só deverá ocorrer somente a chamada do método *fire()*.

3.2.1 *Safelet*

A primeira parte da tradução tem como objetivo simular o comportamento da máquina virtual e pela simulação do funcionamento da interface *Safelet* da especificação, ou seja, deve ser responsável pela inicialização da aplicação. No *Safelet* é chamado o método *getSequencer()* para obter o retorno do *MissionSequencer* que é responsável por supervisionar a execução de missões da aplicação.

Na Figura 3.1 é apresentado o código de tradução para simular o comportamento da execução de uma máquina virtual e do *Safelet*. Este código será geral para todas as traduções

desenvolvidas pois será responsável só pela inicialização da aplicação. Inicialmente é necessário inicializar os sinais para o controle da periodicidade, conforme apresentado na Seção 2.3.1.2, implementado entre as linhas 5 e 8. Na linha 10 é instanciado um objeto da classe *Safelet* para que possibilite a chamada do seu método *getSequencer()*. Após, na linha 11, é declarada uma *pthread* do *MissionSequencer* que recebe o retorno do método *getSequencer()* da classe *Safelet*. E finalmente, a *pthread* do *MissionSequencer* deve ser executada conforme apresentada na linha 12.

```
1  int main(int argc, char *argv[]) {
2        int i;
3        sigset_t alarm_sig;
4
5        sigemptyset (&alarm_sig);
6        for (i = SIGRTMIN; i <= SIGRTMAX; i++)
7              sigaddset (&alarm_sig, i);
8        sigprocmask (SIG_BLOCK, &alarm_sig, NULL);
9
10       Safelet *app = new Safelet();
11       MissionSequencer *threadms = app->getSequencer();
12       threadms->start(NULL);
13       threadms->join();
14       return 0;
15 }
```

Figura 3.1 – Implementação da classe principal.

Na Figura 3.2 é apresentada a classe *Safelet* desenvolvida para a tradução. Nesta classe existe o método construtor na linha 1, *initializeApplication()* na linha 5 e o *getSequencer()* na linha 9. Na especificação SCJ, o *Safelet* era uma interface com os métodos *initializeApplication()* e *getSequencer()* especificados para a implementação pelo usuário.

O método *initializeApplication()* é disponibilizado para que o desenvolvedor da aplicação aloque recursos globais para a utilização na aplicação. Já o método *getSequencer()* retorna uma instância de uma *pthread* contendo um *MissionSequencer* para a execução das missões da aplicação.

```
1  Safelet::Safelet() {
2        //    TODO Auto-generated constructor stub
3  }
4
5  void Safelet::initializeApplication(){
6        //    do nothing
7  }
8
9  PThreadMissionSequencer *Safelet::getSequencer(){
10       PThreadMissionSequencer *thr = new PThreadMissionSequencer(priority, mission);
11       return thr;
12 }
```

Figura 3.2 – Proposta de tradução da classe *Safelet*.

A aplicação da tradução no exemplo apresentado na Seção 2.2 é apresentada na Figura

3.3. O método *initializeApplication()* na linha 6, conforme o exemplo, não necessita alocar nenhum recurso para a execução para a aplicação. Na linha 10 é apresentado o método *getSequencer()* que declara uma *pthread* do *MissionSequencer* na linha 12, definindo sua prioridade e Missão. Como no exemplo apresentado, a execução é de uma única Missão, já é passado por parâmetro uma Missão para a execução e, por fim, é retornada esta *pthread*.

```
1  static const int sequencer_priority = 10;
2  Safelet::Safelet() {
3        //    TODO Auto-generated constructor stub
4  }
5
6  void Safelet::initializeApplication(){
7        //    do nothing
8  }
9
10 PThreadMissionSequencer *Safelet::getSequencer(){
11       int storage = 100;
12       PThreadMissionSequencer *thr = new PThreadMissionSequencer(sequencer_priority, new
             PThreadMission("Mission␣1"));
13       return thr;
14 }
```

Figura 3.3 – Tradução da classe Safelet aplicada no exemplo.

3.2.2 MissionSequencer

Na Figura 3.4 é apresentada a classe *MissionSequencer* com seus métodos e sua *pthread* encapsulada. Para traduzir o comportamento do *MissionSequencer* é necessário encapsular uma *pthread* em C++ conforme apresentado na Seção 2.3.1.1 para supervisionar a execução da (s) missão (ões) da aplicação. Na linha 4 é apresentado o método *start()* responsável pela criação da *pthread* e na linha 13 é apresentado o método *exec()* responsável por chamar o método *run()* da *pthread* encapsulada apresentado na linha 16.

A partir da linha 16 é apresentada a *pthread* encapsulada do *MissionSequencer*. O método *run()* da *pthread* realizará a chamada do método *getNextMission()* responsável por obter a próxima Missão para execução. A primeira chamada deste método receberá a primeira Missão pronta para execução e após a finalização da mesma, será chamado novamente enquanto existir missões armazenadas no *MissionSequencer*.

Na linha 24 é apresentado o método *waitForTermination()* responsável pelo bloqueio da execução da *pthread* do *MissionSequencer* enquanto não é finalizada a execução da sequência de missões contidas nele. É apresentado o método *terminateMission()* na linha 29, responsável por efetuar o desbloqueio para finalizar a execução do *MissionSequencer*.

Os métodos *start()*, *exec()*, *run()*, *waitForTermination()* e *terminationMission()* serão responsáveis pelo comportamento do *MissionSequencer*, ou seja, serão fixos para qualquer tra-

dução de aplicação SCJ de nível 1.

O método construtor e o metodo *getNextMission()* são apresentados nas linhas 1 e 32 respectivamente. O método construtor deve receber os atributos passado na sua instanciação. O método *getNextMission()* deve ser implementado pelo desenvolvedor onde irá controlar a supervisão da sequência de missões para a execução.

```
1  MissionSequencer::MissionSequencer(){
2        //Codigo do usuario
3  }
4  void MissionSequencer::start(void *arg) {
5        pthread_mutex_init(&_mutex, NULL);
6        pthread_cond_init(&condmutex, NULL);
7        int ret;
8        this->arg = arg;
9        if ((ret = pthread_create(&_id, NULL, &MissionSequencer::exec, this)) != 0) {
10             throw "Error";
11       }
12 }
13 void *MissionSequencer::exec(void *thr) {
14       reinterpret_cast<MissionSequencer *> (thr)->run();
15 }
16 void PThreadMissionSequencer::run() {
17       while (true){
18             _mission = getNextMission();
19             _mission->start(arg);
20             waitForTermination();
21             mission_done = false;
22       }
23 }
24 void PThreadMissionSequencer::waitForTermination() {
25       pthread_mutex_lock(&_mutex);
26       pthread_cond_wait(&condmutex, &_mutex);
27       pthread_mutex_unlock(&_mutex);
28 }
29 void PThreadMissionSequencer::terminateMission(){
30       pthread_cond_signal(&condmutex);
31 }
32 PThreadMission *PThreadMissionSequencer::getNextMission() {
33       // Codigo do usuario
34 }
```

Figura 3.4 – Proposta de tradução da classe *MissionSequencer*.

Na Figura 3.5 é apresentada a aplicação da proposta de tradução no exemplo apresentado na Seção 2.2. Conforme já mencionado anteriormente somente o método construtor e o método *getNextMission()* não serão fixos e deverão ser traduzidos manualmente ou utilizando ferramentas para tradução, pois neles que se encontram o código criado pelo desenvolvedor da aplicação SCJ. Na linha 1 é apresentado o método construtor que recebe por parâmetro os atributos de prioridade e a Missão que deverá ser executada. A tradução do método *getNextMission()* é apresentado na linha 6 onde é registrado de qual instância do *MissionSequencer* esta Missão pertence, necessário para possibilitar a finalização do *MissionSequencer* após o fim da execução da Missão. Após, na linha 10, é retornada a Missão para sua execução.

```
1  MissionSequencer::MissionSequencer(int priority, PThreadMission *m){
2        this->priority = priority;
3        this->m = _m;
4  }
5
6  PThreadMission *PThreadMissionSequencer::getNextMission() {
7        if (!mission_done){
8              mission_done = true;
9              _m->setMissionSequencer(this);
10             return _m;
11       }
12       else
13             return NULL;
14 }
```

Figura 3.5 – Tradução da classe *MissionSequencer* aplicada no exemplo.

3.2.3 Mission

Assim como o *MissionSequencer*, a classe *Mission* também possui uma *pthread* encapsulada. Ao ser iniciada a execução da *pthread* de um objeto da classe *Mission*, inicialmente é executado o método *run()* da *pthread*.

Na Figura 3.6 é apresentado a proposta da tradução genérica desta classe. O método *run()* é apresentado na linha 1 e segue uma ordem definida na especificação, onde inicialmente é chamado o método *initialize()* na linha 4, responsável por criar e registrar no sistema todos os manipuladores de eventos periódicos e aperiódicos. Após é chamado o método *exec()* na linha 6, responsável pela execução do código do desenvolvedor da aplicação e após a finalizaçao da execução da Missão é necessário chamar o método *cleanup()* responsável pela limpeza de dados e informações referentes a Missão executada conforme apresentado na linha 8. Este método deverá ser fixo, sem necessitar a alteração ou adaptação para a tradução. É utilizado um atributo público chamado *phase* para fornecer a informação para todos os objetos da aplicação quanto a qual fase de execução a Missão se encontra em tempo de execução.

Ao ser executado a chamada do método *initialize()* do *Mission*, apresentado na linha 12, deverá ser efetuada a instanciação do(s) objeto(s) *PeriodicEventHandlers* e ou *AperiodicEventHandler*. O *PeriodicEventHandler*, conforme apresentado na Seção 2.2, é responsável pela manipulação de eventos periódicos, já o *AperiodicEventHandler* é responsável pela manipulação de eventos aperiódicos da aplicação. Uma aplicação pode conter uma ou mais instância do *PeriodicEventHandler* e ou *AperiodicEventHandler* onde cada instância será uma *pthread* encapsulada executada simultaneamente.

Na linha 16 é apresentada a proposta de tradução do método *exec()* responsável pelo código do desenvolvedor da aplicação.Inicialmente é necessário implementar o código que de-

verá ser executado pela Missão e após é chamado o método *StartAll()* apresentado na linha 24 responsável por iniciar todos os manipuladores de eventos. O atributo chamado *qtdHandler* é utilizado para controlar a quantidade de manipuladores de eventos periódicos que se encontram em execução na Missão. Quando não existirem mais manipuladores de eventos periódicos em execução é porque foi finalizada a execução de todos os manipuladores e a execução da Missão deve ser finalizada.

```
1  void PThreadMission::run() {
2        _terminateAll = false;
3        _phase = 0; // INITIAL
4        initialize();
5        _phase = 1; // EXECUTE
6        exec();
7        _phase = 2; //CLEANUP
8        cleanup();
9        _phase = -1; //INACTIVE
10 }
11
12 void PThreadMissionMission::initialize(){
13       //Codigo do usuario
14 }
15
16 void PThreadMission::exec() {
17       // Codigo do usuario
18       startAll();
19       pthread_mutex_lock(&_mutex);
20       pthread_cond_wait(&condmutex, &_mutex);
21       pthread_mutex_unlock(&_mutex);
22 }
23
24 void PThreadMission::startAll() {
25       //Codigo do usuario
26 }
```

Figura 3.6 – Proposta de tradução da classe *Mission*.

Na Figura 3.7 é apresentado a aplicação da tradução no exemplo da especificação onde são apresentados os métodos *initialize()*, *exec()* e *startAll()*. Os outros métodos definidos na classe *Mission* serão padrão para qualquer outra tradução de aplicação SCJ de nível 1, não necessitando de alteração pois são responsáveis pelo comportamento da SCJ.

O método *initialize()* é apresentado na linha 1 onde são instanciados quatro *PeriodicEventHandler* e um *AperiodicEventHandler*. Nas linhas 2, 4 e 6 são declarados os três manipuladores de eventos periódicos e instanciados com a sua descrição, prioridade, tempo para inicialização, periodicidade, armazenamento e tamanho. Como o escopo deste trabalho não faz parte o escopo de memória, os atributos de armazenamento e tamanho do *PeriodicEventHandler* se mantiveram para possíveis trabalhos futuros.

Na linha 8 é declarado um manipulador de evento aperiódico e instanciado. Nas linhas 3, 5, 7 e 9 são registrados os manipuladores onde é passado por parâmetro a qual Missão este manipulador de eventos pertence. O objetivo é manter a informação da Missão para que quando o

manipulador de eventos finalize a sua execução, possibilite que a Missão tenha o conhecimento e controle de quantos manipuladores ainda estão em execução.

Na linha 12 é apresentado o método *exec()* onde é chamado o método *StartAll()* e após é feito um bloqueio através de *MUTEX (Mutual Exclusion)* para aguardar a finalização da execução de todos os manipuladores de eventos. Após a finalização da execução dos manipuladores é enviado um sinal para desbloquear e assim finalizar sua execução.

Na linha 19 é apresentada a aplicação da tradução no exemplo. São inicializados os três manipuladores de eventos periódicos e incrementados no atributo responsável por controlar a quantidade de manipuladores de eventos periódicos. Após é inicializado a execução do manipulador de evento aperiódico e executado o método *fire()* para execução do manipulador de eventos aperiódicos.

```
1  void PThreadMission::initialize(){
2        thra = new PThreadPeriodicEventHandler("PeriodicEventHandler A", 2, 0, 10000, 100,
             100);
3        thra->reg(this);
4        thrb = new PThreadPeriodicEventHandler("PeriodicEventHandler B", 2, 500, 10000, 100,
             100);
5        thrb->reg(this);
6        thrc = new PThreadPeriodicEventHandler("PeriodicEventHandler C", 2, 750, 15000, 100,
             100);
7        thrc->reg(this);
8        thrd = new PThreadAperiodicEventHandler(1, 2, 3, "AperiodicEventHandler");
9        thrd->reg(this);
10 }
11
12 void PThreadMission::exec() {
13       startAll();
14       pthread_mutex_lock(&_mutex);
15       pthread_cond_wait(&condmutex, &_mutex);
16       pthread_mutex_unlock(&_mutex);
17 }
18
19 void PThreadMission::startAll() {
20       qtdHandler = 0;
21       thra->start();
22       qtdHandler++;
23       thrb->start();
24       qtdHandler++;
25       thrc->start();
26       qtdHandler++;
27       thrd->start();
28       thrd->fire();
29 }
```

Figura 3.7 – Tradução da classe *Mission* aplicada no exemplo.

3.2.4 PeriodicEventHandler

Para simular o comportamento da especificação dos manipuladores de eventos periódicos foi necessário implementar uma *pthread* encapsulada em C++ do tipo periódica. A períodicidade da *pthread* foi implementada através de sinais POSIX conforme apresentado na Seção 2.3.1.2. Uma *pthread* periódica possibilita com que a *pthread* tenha um comportamento

temporal, de inicialização e de finalização de execução.

Na Figura 3.8 é apresentada a proposta de tradução da classe *PeriodicEventHandler* com seu construtor e seus métodos para criação e execução de sua *pthread* encapsulada. Além destes métodos apresentados é necessário também, nesta classe, a implementação do comportamento de periodicidade apresentado na Seção 2.3.1.2. Na linha 1 é apresentado o seu construtor e para instanciar um *PeriodicEventHandler* é obrigatório necessário obter a informação temporal do manipulador, ou seja, o início e tempo de execução do manipulador.

O método *run()* da classe *PthreadPeriodicEventHandler* apresentado a partir da linha 17 é uma classe estendida do *PeriodicEventHandler* onde foi encapsulada a *pthread* periódica apresentada na Seção 2.3.1.2. Cada *pthread* periódica possui um tempo de inicialização. Para este controle é utilizado um *microsleep*, conforme apresentado na linha 19, para inicializar a execução conforme o parâmetro passado pelo usuário. O tempo passado por parâmetro deve ser em microsegundos. Para controlar a periodicidade da execução é utilizado o método *make_periodic()* que é responsável por controlar o tempo de execução conforme é mostrado na linha 20.

Na linha 22 é onde deverá ser inserido o código do usuário, ou seja, o código da implementação do que será executado pelo manipulador de evento periódico. A finalização da aplicação pode ocorrer de duas maneiras. A primeira forma é através da finalização da execução do código. A segunda forma é a solicitação da execução recebida da Missão de onde o manipulador está alocado. E para isso deve ser utilizado o atributo público do tipo *boolean* chamado *terminate*. Enquanto for *false* a execução do código do usuário segue em execução. Ao alterar esse atributo para *true*, a execução é finalizada através do comando *break* conforme apresentado na linha 24 e 25. Ao finalizar a execução a *pthread* decrementa o contador de manipuladores de eventos periódicos e, se não existir mais nenhum em execução, então é chamado o método *requestSequenceTermination()* da Missão.

```
1  PThreadPeriodicEventHandler::PThreadPeriodicEventHandler ()
2  : PeriodicEventHandler(){
3        //Codigo do usuario
4  }
5  void PeriodicEventHandler::start(void *arg){
6        int ret;
7        this->arg = arg;
8
9        if ((ret = pthread_create(&_id, NULL, &PeriodicEventHandler::exec, this)) != 0) {
10             cout << strerror(ret) << endl;
11             throw "Error";
12       }
13 }
14 void *PeriodicEventHandler::exec(void *thr) {
15       reinterpret_cast<PeriodicEventHandler *> (thr)->run();
16 }
17 void PThreadPeriodicEventHandler::run() {
18       struct periodic_info info;
19       microsleep (inicio);
20       make_periodic (periodo, &info);
21       while(true){
22             //codigo do usuario;
23             wait_period (&info);
24             pthread_mutex_lock(&_mutex);
25             if (terminate)
26                   break;
27             pthread_mutex_lock(&_mutex);
28       }
29       _mission->qtdHandler--;
30       if (_mission->qtdHandler == 0)
31       _mission->requestSequenceTermination();
32 }
```

Figura 3.8 – Proposta de tradução da classe *PeriodicEventHandler*.

Seguindo a proposta da tradução, os métodos *start()* e *exec()* serão padrões para qualquer tradução deste objeto. A tradução será focada no construtor da classe e no método *HandleAsyncEvent()* do *PeriodicEventHandler* apresentado na Seção 2.2. Na Figura 3.9 é apresentada a aplicação da tradução proposta no método construtor e do método responsável pela execução da rotina atribuída ao *PeriodicEventHandler* definido pela especificação.

Na linha 1 é apresentado o construtor do *PeriodicEventHandler* onde recebe por parâmetro o ID, nome, tempo de início, e período de execução. O método responsável pela execução do código do usuário será o *run()*. No exemplo, é incrementado o atributo chamado *eventCounter* e foi definido que a execução finalizará após cinquenta execuções conforme apresentado nas linhas 11 e 12. Caso seja desejável que a execução finalize somente quando a Missão definir através do atributo *terminate* conforme apresentado na linha 14, basta alterar o laço da linha 11 para um laço infinito. Na linha 17, 18 e 19 é feito o controle da quantidade de manipuladores de eventos periódicos em execução. Ao finalizar a execução do manipulador, é decrementado o atributo da instância de Missão a qual o manipulador pertence. Se for o último manipulador em execuçã, é chamado o método *requestSequenceTermination()* da Missão responsável por finalizar a execução.

```
1  PThreadPeriodicEventHandler::PThreadPeriodicEventHandler (int id, string nm, time_t inicio,
       time_t periodo) : PeriodicEventHandler(nm, inicio, periodo){
2        this->id = id;
3        this->nm = nm;
4        this->inicio = inicio;
5        this->periodo = periodo;
6  }
7  void PThreadPeriodicEventHandler::run() {
8        struct periodic_info info;
9        microsleep (inicio);
10       make_periodic (periodo, &info);
11       while(eventCounter < 50){
12             eventCounter++;
13             wait_period (&info);
14             if (terminate)
15                   break;
16       }
17       _mission->qtdHandler--;
18       if (_mission->qtdHandler == 0)
19             _mission->requestSequenceTermination();
20 }
```

Figura 3.9 – Tradução da classe *PeriodicEventHandler* aplicada no exemplo.

3.2.5 AperiodicEventHandler

A tradução do comportamento do *AperiodicEventHandler*, assim como o *PeriodicEventHandler*, é uma *pthread*. Mas como neste caso o manipulador de eventos é aperiódico, ou seja, uma execução não-periódica, não é necessário utilizar os métodos *make_periodic()*, *wait_period()* implementados para obter um comportamento periódico, que foram apresentados na Seção 2.3.1.2.

Na Figura 3.10 é possível visualizar a proposta de tradução da classe *AperiodicEventHandler*. Na linha 1 é apresentado o construtor da classe, na linha 4 é possível visualizar o método *start()* responsável por criar a *pthread* encapsulada da classe. O método *exec()* responsável pela chamada do método *run()* da *pthread* encapsulada é apresentado na linha 15.

A partir da linha 18 é encontra-se a *pthread* encapsulada do *AperiodicEventHandler* responsável por manter o seu comportamento. Na linha 18 é apresentado o método *run()* da *pthread* do *AperiodicEventHandler*, inicialmente é necessário entrar em bloqueio *lock* utilizando a váriavel de condição *MUTEX (Mutual Exclusion)* conforme a linha 20. A execução fica aguardando o sinal enviado pelo método *fire()*, apresentado na linha 33, para iniciar a execução do código do usuário conforme apresentado na linha 23. Ao receber o sinal então é efetuada a execução e ele fica em bloqueio novamente aguardando uma nova requisição de execução. O manipulador de eventos aperiódicos ficará durante toda a execução da Missão pronto para a execução e para finalizar a execuçao é necessário que o atributo público do tipo *boolean terminate* seja alterado para *true* que deverá ocorrer ao final da execução da Missão.

```
1  AperiodicEventHandler::AperiodicEventHandler() {
2        //Codigo do usuario
3  }
4  void AperiodicEventHandler::start(void *arg){
5        pthread_mutex_init(&_mutex, NULL);
6        pthread_cond_init(&condmutex, NULL);
7        int ret;
8        this->arg = arg;
9
10       if ((ret = pthread_create(&_id, NULL, &AperiodicEventHandler::exec, this)) != 0) {
11             cout << strerror(ret) << endl;
12             throw "Error";
13       }
14 }
15 void *AperiodicEventHandler::exec(void *thr) {
16       reinterpret_cast<AperiodicEventHandler *> (thr)->run();
17 }
18 void PThreadAperiodicEventHandler::run() {
19       terminate = false;
20       pthread_mutex_lock(&_mutex);
21       while (true){
22             while (execute == 0){
23                   //codigo do usuario
24                   pthread_cond_wait(&condmutex, &_mutex);
25             }
26             execute = 0;
27             if (terminate)
28                   break;
29       }
30       pthread_mutex_unlock(&_mutex);
31 }
32
33 void PThreadAperiodicEventHandler::fire() {
34       execute = 1;
35       pthread_mutex_lock(&_mutex);
36       pthread_cond_signal(&condmutex);
37       pthread_mutex_unlock(&_mutex);
38 }
```

Figura 3.10 – Proposta de tradução da classe *AperiodicEventHandler*.

Na Figura 3.11 é apresentada a aplicação da tradução proposta nesta dissertação. Na linha 1 é possível visualizar o construtor do manipulador de evento aperiódico. Na linha 5 é apresentado o método *run()* responsável pelo código do usuário. O manipulador de eventos aperiódicos após sua inicialização e execução entra em bloqueio (*lock*) conforme mostrado na linha 7. Após o bloqueio o manipulador fica pronto para execução aguardando a chamada do método *fire()*, apresentado na linha 19, para que ocorra a execução do código do usuário. Após a chamada do método *fire()* executa um incremento no atributo *eventCounter*.

```
1  AperiodicEventHandler::AperiodicEventHandler(int priority, int release) {
2        this->priority = priority;
3        this->release = release;
4  }
5  void PThreadAperiodicEventHandler::run() {
6        terminate = false;
7        pthread_mutex_lock(&_mutex);
8        while (true){
9              while (execute == 0){
10                   eventCounter++;
11                   pthread_cond_wait(&condmutex, &_mutex);
12             }
13             execute = 0;
14             if (terminate)
15                   break;
16       }
17       pthread_mutex_unlock(&_mutex);
18 }
19 void PThreadAperiodicEventHandler::fire() {
20       execute = 1;
21       pthread_cond_signal(&condmutex);
22 }
```

Figura 3.11 – Tradução da classe *AperiodicEventHandler* aplicada no exemplo.

3.3 Conclusão

Neste capítulo, foi apresentada a tradução proposta nesta dissertação onde é possível verificar as traduções dos componentes SCJ na linguagem de programação C++. Importante salientar que esta tradução não tem foco a tradução da manipulação de memória utilizada pela SCJ pois a linguagem de programação C++ não possui coletor de lixo e aplicações SCJ de nível 0 não foram selecionado para tradução. Diversos testes da tradução foram desenvolvidos com diferentes aplicações conforme apresentado no Capítulo 4 e em todos os casos o comportamento da aplicação manteve-se o mesmo.

4 VALIDAÇÃO E TESTES

Neste capítulo são apresentadas aplicações da tradução proposta em exemplos desenvolvidos em *Safety Critical Java* (SCJ), assim como testes efetuados em aplicações traduzidas. Os códigos originais completos das aplicações SCJ estão inseridos nos Anexos A, B e C desta dissertação. Como descrito anteriormente, o escopo deste trabalho é a tradução de aplicações SCJ de nível 1 sem a manipulação de memória. O objetivo é apresentar que após a aplicação da tradução proposta os softwares gerados mantenham o mesmo comportamento de uma aplicação desenvolvida em *Safety Critical Java*. (ABOWD et al., 1999)

4.1 Tradução do Exemplo adaptado SCJ Multi-Missão

Esta tradução foi desenvolvida em cima do exemplo geral disponível para visualização o código SCJ no Anexo A. Neste exemplo existe a possiblidade do usuário implementar diversas missões para serem executadas pelo aplicativo, ou seja, o desenvolvedor poderá implementar diversas missões para execução.

Na Figura 4.1 é apresentada a classe principal da aplicação. Inicialmente foram declarados e iniciados os sinais os quais são responsáveis pela execução periódica dos manipuladores de eventos periódicos (PeriodicEventHandlers). Após, na linha 9 é declarado e construida uma instância da classe *Safelet* em C++ e na linha 10 é chamado o método responsável pela declaração e inicialização de dados globais para utilização na aplicação. Na linha 11 é chamado o método *getSequencer()* que retorna a *thread* contendo a sequência de missões (*MissionSequencer*) que a aplicação deverá executar. E, na linha 12 é criada e iniciada a *thread* e inicializada a execução da aplicação.

```
1  int main(int argc, char *argv[]) {
2        int i;
3        sigset_t alarm_sig;
4        sigemptyset (&alarm_sig);
5        for (i = SIGRTMIN; i <= SIGRTMAX; i++)
6        sigaddset (&alarm_sig, i);
7        sigprocmask (SIG_BLOCK, &alarm_sig, NULL);
8
9        Safelet *app = new Safelet();
10       app->initializeApplication();
11       MissionSequencer *threadms = app->getSequencer();
12       threadms->start(NULL);
13       threadms->join();
14       return 0;
15 }
```

Figura 4.1 – Método principal da aplicação de exemplo.

Na Figura 4.2 é apresentando o método *getSequencer()* onde é criada uma *ptread* encapsulada do *MissionSequencer* que retorna a sequência de missões para execução, ou seja, uma *pthread* de um objeto de MissionSequencer. Neste exemplo é possível verificar na linha 19 a criação de um vetor de missões onde são inseridas três missões instanciadas entre as linhas 16 e 18. Ao criar o *MissionSequencer* é passado por parâmetro estas missões e ao chamar o método *getNextMission()*, será retornada uma Missão de cada vez em ordem de armazenamento.

```
1  static const int sequencer_priority = 10;
2
3  Safelet::Safelet() {
4        //    TODO Auto-generated constructor stub
5  }
6
7  Safelet::~Safelet() {
8        //     TODO Auto-generated destructor stub
9  }
10
11 void Safelet::initializeApplication(){
12       //    do nothing
13 }
14
15 PThreadMissionSequencer *Safelet::getSequencer(){
16       int storage = 100;
17       PThreadMission *_mission1 = new PThreadMission("Mission 1");
18       PThreadMission *_mission2 = new PThreadMission("Mission 2");
19       PThreadMission *_mission3 = new PThreadMission("Mission 3");
20       vector<PThreadMission*> m;
21       m.push_back(_mission1);
22       m.push_back(_mission2);
23       m.push_back(_mission3);
24       PThreadMissionSequencer *thr = new PThreadMissionSequencer(sequencer_priority, storage
             , m);
25       return thr;
26 }
```

Figura 4.2 – Classe Safelet do exemplo Multi-Missão.

Na Figura 4.3 é apresentado um trecho da classe *MissionSequencer* traduzido onde na linha 1 encontra-se o método *run()* da *pthread* do *MissionSequencer* responsável por sua execução. Na linha 3 é chamado o método *getNextMission()* responsável por obter a próxima Missão para execução pertencente a este *MissionSequencer*. A execução deste método principal da *pthread* do *MissionSequencer* mantém-se em um laço infinito. Na linha 4 é invocado o método *start()* da Missão responsável pela criação e execução da *pthread* da Missão. Por fim, na linha 5, é chamado o método *waitForTermination()*.

Na linha 9 é apresentado o método *getNextMission()* responsável por retornar a Missão para a execução. Na linha é apresentado o laço que percorre o vetor de missões e a cada Missão encontrada ela é retornada para execução. A execução de missões é efetuada de forma sequencial. Cada Missão dentro do vetor é retornada e o laço segue sua execução enquanto exista missões dentro do vetor. O método *waitForTermination()*, conforme apresentado na na linha 23, é responsável por bloquear a execução do *run()* do *MissionSequencer* até que seja

enviado um sinal para continuar a execução, ou seja, enquanto a execução da Missão e seus manipuladores não sejam finalizados.

```
1  void PThreadMissionSequencer::run() {
2        while (true){
3              _mission = getNextMission();
4              _mission->start(arg);
5              waitForTermination();
6              mission_done = false;
7        }
8  }
9  PThreadMission *PThreadMissionSequencer::getNextMission() {
10       PThreadMission *_m = NULL;;
11       if (!mission_done){
12             mission_done = true;
13             while (i < m.size()){
14                   _m = m.at(i);
15                   _m->setMissionSequencer(this);
16                   i++;
17                   return _m;
18             }
19       }
20       else
21             return NULL;
22 }
23 void PThreadMissionSequencer::waitForTermination() {
24       pthread_mutex_lock(&_mutex);
25       pthread_cond_wait(&condmutex, &_mutex);
26       pthread_mutex_unlock(&_mutex);
27 }
```

Figura 4.3 – Classe *MissionSequencer* do exemplo Multi-Missão.

Na Figura 4.4 é apresentado um trecho da classe *Mission* onde na linha 1 encontra-se o método principal de execução da *pthread* definida para traduzir o comportamento do *Mission*. Conforme definido na especificação é utilizado um atributo público chamado *phase* para que seja possível acompanhar em todo o escopo da aplicação a sua sequência de execução. Inicialmente é definido que a Missão está em sua fase inicial e chamado o método *initiliaze()* conforme apresentado na linha 5. Após é definido que a Missão está em execução e chamado o método *exec()* na linha 7. Após a finalização da execução, a Missão entra na fase de limpeza e chamado o método *cleanup()* conforme apresentado na linha 9.

Na linha 11 é apresentado o método *exec()* responsável pelo código do usuário da Missão e também responsável por invocar o método *StartAll()* e após isso ele entra em bloqueio e aguarda a finalização da execução dos manipuladores periódicos de evento.

O método *initilize()*, apresentado na linha 17, como definido na especificação, é responsável por declarar e registrar todos os manipuladores de eventos de cada Missão. São instanciados três manipuladores de eventos periódicos (PEH) nas linhas 18, 20 e 22. Também é instanciado um manipulador de evento aperiódico (APEH) na linha 24. Cada manipulador chama o método *reg()* onde é atribuido de qual Missão cada um é pertencente. O objetivo é obter a referência do *Mission* para desbloquear a Missão para sua finalização após o término da

execução de todos manipuladores periódicos de evento.

O método *StartAll()* apresentado na linha 27 é responsável por iniciar as *pthreads* dos manipuladores de evento. Os manipuladores de eventos periódicos (PEH), ao chamar o método *start()* inicia a sua execução conforme apresentado nas linhas 29, 31 e 33. Já os manipuladores de eventos aperiódicos ao chamar o *start()* é iniciado conforme a linha 35. O manipulador de eventos aperiódicos, para sua execução, é necessário chamar o método *fire()* conforme a linha 36 para que ele então seja desbloqueado e sua execução seja iniciada. O atributo *qtdHandler* é utilizado para armazenar a quantidade de manipulador de eventos periódicos em execução, pois enquanto existir manipuladores de eventos periódicos em execução não é possível finalizar a execução de uma miss

```
1  void PThreadMission::run() {
2        _terminateAll = false;
3        _phase = 0; // INITIAL
4        initialize();
5        _phase = 1; // EXECUTE
6        exec();
7        _phase = 2; //CLEANUP
8        cleanup();
9        _phase = -1; //INACTIVE
10 }
11 void PThreadMission::exec() {
12       startAll();
13       pthread_mutex_lock(&_mutex);
14       pthread_cond_wait(&condmutex, &_mutex);
15       pthread_mutex_unlock(&_mutex);
16 }
17 void PThreadMission::initialize(){
18       thra = new PThreadPeriodicEventHandler("PeriodicEventHandlerA", 2, 0, 10000, 100, 100)
             ;
19       thra->reg(this);
20       thrb = new PThreadPeriodicEventHandler("PeriodicEventHandlerB", 2, 500, 10000, 100,
             100);
21       thrb->reg(this);
22       thrc = new PThreadPeriodicEventHandler("PeriodicEventHandlerC", 2, 750, 15000, 100,
             100);
23       thrc->reg(this);
24       thrd = new PThreadAperiodicEventHandler(1, 2, 3, "AperiodicEventHandler");
25       thrd->reg(this);
26 }
27 void PThreadMission::startAll() {
28       qtdHandler = 0;
29       thra->start();
30       qtdHandler++;
31       thrb->start();
32       qtdHandler++;
33       thrc->start();
34       qtdHandler++;
35       thrd->start();
36       thrd->fire();
37 }
```

Figura 4.4 – Classe *Mission* do exemplo SCJ Multi-Missão.

Na Figura 4.5 é apresentado o método *run()* da *pthread* do manipulador de evento periódico. A funcionalidade deste manipulador de eventos periódico é o mesmo do exemplo simples apresentado na Seção 2.2 onde inicialmente é efetuado a declaração da estrutura de dados necessária para o controle do comportamento periódico deste objeto. Se ao construir o objeto o

usuário tenha passado por parâmetro um tempo para a inicialização entra em um *microsleep* e fica aguardando este tempo para sua execução. Então é chamada do método *make_periodic* responsável por criar um comportamento periódico conforme o tempo passado por parâmetro e recebido no atributo "periodo". Então é iniciada a execução do código do usuário, onde no caso é incrementado o contador chamado *eventCounter*. O manipulador de eventos periódico continua em execução enquanto não é alterado o atributo *terminate* para *true*.

```
1  void PThreadPeriodicEventHandler::run() {
2        struct periodic_info info;
3        terminate = false;
4        microsleep (inicio*1000);
5        make_periodic (periodo, &info);
6        while(eventCounter < 50){
7              eventCounter++;
8              wait_period (&info);
9              if (terminate)
10                   break;
11       }
12       _mission->qtdHandler--;
13       if (_mission->qtdHandler == 0)
14             _mission->requestSequenceTermination();
15 }
```

Figura 4.5 – Método *run* do *PeriodicEventHandler* do exemplo Multi-Missão.

Na Figura 4.6 é apresentado trecho da classe do AperiodicEventHandler. Na linha 1 é apresentado o método *run()* onde na linha 3, ao inicializar a execução a *pthread* é bloqueada para a execução. E fica aguardando o chamado do método *fire()*, apresentado na 15, para liberar a execução.

```
1  void PThreadAperiodicEventHandler::run() {
2        terminate = false;
3        pthread_mutex_lock(&_mutex);
4        while (true){
5              while (execute == 0){
6                    eventCounter++;
7                    pthread_cond_wait(&condmutex, &_mutex);
8              }
9              execute = 0;
10             if (terminate)
11                   break;
12       }
13       pthread_mutex_unlock(&_mutex);
14 }
15 void PThreadAperiodicEventHandler::fire() {
16       execute = 1;
17       pthread_cond_signal(&condmutex);
18 }
```

Figura 4.6 – Classe *AperiodicEventHandler* da aplicação do exemplo Multi-Missão.

4.2 Tradução do Exemplo de aplicação *Network*

Nesta seção é apresentada a tradução de uma aplicação SCJ nível 1 que está disponível para visualização no Anexo B. Este exemplo possui um manipulador de evento periódico (*PeriodicEventHandler*) e quatro manipuladores de eventos aperiódicos (*AperiodicEventHandler*).

Inicialmente foi efetuada a tradução da interface *Safelet* da aplicação apresentada na Figura 4.7 responsável por definir o tamanho da memória imortal e o método *initializeApplication()*. Da mesma forma que no exemplo anterior, foi implementada uma classe contendo os métodos definidos na especificação para manter o comportamento definido na especificação em substituição a interface em SCJ.

```
1  Safelet::Safelet() {
2        //    TODO Auto-generated constructor stub
3  }
4
5  Safelet::~Safelet() {
6        //     TODO Auto-generated destructor stub
7  }
8
9  void Safelet::initializeApplication(){
10       //    do nothing
11 }
12
13 PThreadMissionSequencer *Safelet::getSequencer(){
14       int storage = 100;
15       cout << "No getSequencer do MylEvel1App" << endl;
16       PThreadMissionSequencer *thr = new PThreadMissionSequencer(sequencer_priority, storage
             );
17       return thr;
18 }
19 long Safelet::immortalMemorySize(){
20       return 10000;
21 }
```

Figura 4.7 – Classe Safelet da aplicação *Network*.

É apresentado na Figura 4.8 um trecho da classe *MissionSequencer*. A tradução do método *run* da *pthread* é apresentada na linha 1. É declarada e instanciada uma Missão e é chamado o método *getNextMission()* apresentado na linha 9, responsável por obter a próxima Missão para execução pertencente a este *MissionSequencer*. A execução deste método principal da *pthread* do *MissionSequencer* entra em um laço infinito. Ao chamar o método *start()* do *Mission* é chamado o método *waitForTermination()* conforme apresentado na linha 4 da Figura 4.8.

O método *waitForTermination()*, conforme apresentado na linha 18, é responsável por bloquear a execução do *run()* até que seja enviado um sinal para continuar a execução, ou seja, enquanto a execução da Missão criada não seja finalizada.

```
1  void PThreadMissionSequencer::run() {
2        while (true){
3              _mission = getNextMission();
4              _mission->start(arg);
5              waitForTermination();
6              mission_done = false;
7        }
8  }
9  MainMission *PThreadMissionSequencer::getNextMission() {
10       if (!mission_done){
11             mission_done = true;
12             return new PThreadMission(this);
13       }
14       else{
15             return NULL;
16       }
17 }
18 void PThreadMissionSequencer::waitForTermination() {
19       pthread_mutex_lock(&_mutex);
20       pthread_cond_wait(&condmutex, &_mutex);
21       pthread_mutex_unlock(&_mutex);
22 }
```

Figura 4.8 – Classe *MissionSequencer* da aplicação *Network*.

Na Figura 4.9 é apresentado o método *run()* da *pthread* do *Mission* responsável por sua execução. Conforme definido na especificação é utilizado um atributo público chamado *phase* para que seja possível acompanhar em todo o escopo da aplicação a sua sequência de execução. Inicialmente é definido que a Missão está em sua fase inicial e chamado o método *initiliaze()* conforme apresentado na linha 5. Após é definido que a Missão está em execução e chamado o método *exec()* na linha sete. Após a finalização da execução, a Missão entra na fase de limpeza e chamado o método *cleanup()* conforme apresentado na linha 9.

O método *initilize()*, apresentado na linha 11, como definido na especificação é responsável por declarar e registrar todos os manipuladores de eventos de cada Missão. É instanciado um manipulador de evento periódico chamando *Handler1* na linha 12 e também são declarados 4 manipuladores de eventos aperiódicos nas linhas 14, 16 ,18 e 20 denominados respectivamente *Handler2*, *ConnectHandler*, *DisconnectHandler* e *SendHandler.* Cada manipulador chama o método *reg()* onde é atribuido de qual Missão cada um é pertencente com o objetivo de obter a referência do objeto *Mission* para desbloquear a Missão para finalizar sua execução após o término de execução dos manipuladores periódicos de evento.

É apresentado na linha 23 o método *exec()* responsável pelo código do usuário da Missão e também é responsável por chamar o método *StartAll()* e após isso ele entra em bloqueio e fica aguardando a finalização da execução dos manipuladores periódicos de evento.

Na linha 29 é apresentado o método *StartAll()* responsável por iniciar as *pthreads* dos manipuladores de evento. Neste exemplo são iniciados todos os manipuladores nas linhas 30, 32, 33, 34 e 35 e é incrementado o contador chamado *qtdHandler* que armazena a quantidade

de manipuladores periódicos em execução na aplicação.

```
1  void PThreadMission::run() {
2        _terminateAll = false;
3        _phase = 0; // INITIAL
4        initialize();
5        _phase = 1; // EXECUTE
6        exec();
7        _phase = 2; //CLEANUP
8        cleanup();
9        _phase = -1; //INACTIVE
10 }
11 void PThreadMission::initialize(){
12       thra = new PThreadPeriodicEventHandler(1, "Handler1", 500, 1000000);
13       thra->reg(this);
14       thrb = new PThreadAperiodicEventHandler (10, 10000, "teste", "Handler2");
15       thrb->reg(this);
16       thrc = new PThreadConnectHandler (10, 10000, "teste", "ConnectHandler");
17       thrc->reg(this);
18       thrd = new PThreadDisconnectHandler (10, 10000, "teste", "DisconnectHandler");
19       thrd->reg(this);
20       thrs = new PThreadSendHandler (10, 10000, "teste", "SendHandler");
21       thrs->reg(this);
22 }
23 void PThreadMission::exec() {
24       startAll();
25       pthread_mutex_lock(&_mutex);
26       pthread_cond_wait(&condmutex, &_mutex);
27       pthread_mutex_unlock(&_mutex);
28 }
29 void PThreadMission::startAll() {
30       thra->start();
31       qtdHandler++;
32       thrb->start();
33       thrc->start();
34       thrd->start();
35       thrs->start();
36 }
```

Figura 4.9 – Método *run* do *Mission* da aplicação *Network*.

Na Figura 4.10 é apresentado o método *run()* da *pthread* do manipulador de evento periódico *Handler1*. Este é o úncio manipulador de evento periódico da aplicação e para melhor exemplificar a execução foi adicionado algumas mensagens na tela em tempo de execução para acompanhar a execução do manipulador.

```
1  void PThreadPeriodicEventHandler::run() {
2        int contador = 0;
3        struct periodic_info info;
4        terminate = false;
5        thrb->start();
6        thrc->start();
7        thrd->start();
8        thrs->start();
9        microsleep (inicio);
10       make_periodic (periodo, &info);
11       while (true){
12             thrb->fire();
13             thrc->fire();
14             thrs->fire();
15             thrd->fire();
16             wait_period (&info);
17             if (terminate)
18                   break;
19       }
20       _mission->qtdHandler--;
21       if (_mission->qtdHandler == 0)
22             _mission->requestSequenceTermination();
23       else
24             cout << "Ainda␣tem␣PEH␣executando:␣"<< int (_mission->qtdHandler) <<endl;
25 }
```

Figura 4.10 – Método *run* do *Handler1* da aplicação *Network*.

Na Figura 4.11 é um trecho da classe *AperiodicEventHandler*. Na linha 1 é apresentado o método *run()* do manipulador de eventos aperiódicos chamado *Handler2*. Conforme apresentado na linha 3 do código, ao inicializar a execução a *pthread* é bloqueada para a execução. Ela fica aguardando o chamado do método *fire()*, apresentado na 15, para liberar a execução.

```
1  void PThreadAperiodicEventHandler::run() {
2        terminate = false;
3        pthread_mutex_lock(&_mutex);
4        while (true){
5              while (execute == 0){
6                    pthread_cond_wait(&condmutex, &_mutex);
7                    cout << "[Network]␣"<< (string)nt.getState() << endl;
8              }
9              execute = 0;
10             if (terminate)
11                   break;
12       }
13       pthread_mutex_unlock(&_mutex);
14 }
15 void PThreadAperiodicEventHandler::fire() {
16       execute = 1;
17       pthread_cond_signal(&condmutex);
18 }
```

Figura 4.11 – Classe *Handler2* da aplicação *Network*.

Nas Figuras 4.12, 4.13 e 4.14 são apresentados trechos dos manipuladores de eventos aperiódicos *DisconnectHandler*, *ConnectHandler* e *SendHandler*. São responsáveis pelas ações da rede da aplicação. Após a chamada do seu método correspondente é executado o método *getState()* da classe *Network* que retorna o estado da classe. Na linha 16 é apresentado o método *fire()* é responsável por enviar o sinal para desbloquear a execução do método *run()* do manipulador de evento aperiódico.

```
1  void PThreadDisconnectHandler::run() {
2        terminate = false;
3        pthread_mutex_lock(&_mutex);
4        while (true){
5              while (execute == 0){
6                    pthread_cond_wait(&condmutex, &_mutex);
7                    nt.disconnect();
8                    cout << "[Network]␣"<< (string)nt.getState() << endl;
9              }
10             execute = 0;
11             if (terminate)
12                   break;
13       }
14       pthread_mutex_unlock(&_mutex);
15 }
16 void PThreadDisconnectHandler::fire() {
17       execute = 1;
18       pthread_cond_signal(&condmutex);
19 }
```

Figura 4.12 – Classe *DisconnectHandler* da aplicação *Network*.

```
1  void PThreadConnectHandler::run() {
2        terminate = false;
3        pthread_mutex_lock(&_mutex);
4        while (true){
5              while (execute == 0){
6                    pthread_cond_wait(&condmutex, &_mutex);
7                    nt.connect();
8                    cout << "[Network]␣"<< (string)nt.getState() << endl;
9              }
10             execute = 0;
11             if (terminate)
12                   break;
13       }
14       pthread_mutex_unlock(&_mutex);
15 }
16 void PThreadConnectHandler::fire() {
17       execute = 1;
18       pthread_cond_signal(&condmutex);
19 }
```

Figura 4.13 – Classe *ConnectHandler* da aplicação *Network*.

```
1  void PThreadSendHandler::run() {
2        terminate = false;
3        pthread_mutex_lock(&_mutex);
4        while (true){
5              while (execute == 0){
6                    pthread_cond_wait(&condmutex, &_mutex);
7                    nt.connect();
8                    cout << "[Network]␣"<< (string)nt.getState() << endl;
9              }
10             execute = 0;
11             if (terminate)
12                   break;
13       }
14       pthread_mutex_unlock(&_mutex);
15 }
16 void PThreadSendHandler::fire() {
17       execute = 1;
18       pthread_cond_signal(&condmutex);
19 }
```

Figura 4.14 – Classe *SendHandler* da aplicação *Network*.

4.3 Desenvolvimento de um exemplo Jantar dos Filósofos

Este exemplo foi desenvolvido com o objetivo de verificar e validar o seu comportamento temporal e sincronização. O código desenvolvido em SCJ encontra-se disponível no Anexo C. É um problema clássico de sincronização formulado por Dijkstra no ano 1965, apresentado em TANENBAUM (2012) onde um número N de filósofos estão sentados em uma mesa circular e cada filósofo tem um prato de espaguete. O espaguete está escorregadio e cada filósofo precisa de dois garfos para comê-lo. Entre cada prato existe um garfo, ou seja, somente haverá N garfos para N filósofos. A vida dos filósofos é alterado entre períodos onde o filósofo come e pensa. Quando um filósofo fica com fome, ele tenta pegar o garfo que se encontra a sua direita e a sua esquerda. Se os filósofos que se encontram ao seu lado não estejam comendo, ele come por um tempo determinado e após este tempo, colocará os garfos novamente na mesa e continuará pensando.

Na Figura 4.15 é apresentado o método principal da aplicação jantar dos filósofos desenvolvido em C++ para instanciar e executar a aplicação SCJ traduzida. Inicialmente foram declarados e iniciados os sinais os quais são responsáveis pela execução periódica dos manipuladores de eventos periódicos (PeriodicEventHandlers). Após é declarado e construída uma instância da classe *Safelet* em C++ e chamado o método responsável pela declaração e inicialização de dados globais para utilização na aplicação e então é chamado o método *getSequencer()* que retorna a *thread* contendo a sequência de missões (*MissionSequencer*) que a aplicação deverá executar. E por final é criada e iniciada a *thread* e inicializada a execução da aplicação.

```
1  int main(int argc, char *argv[]) {
2        int i;
3        sigset_t alarm_sig;
4        sigemptyset (&alarm_sig);
5        for (i = SIGRTMIN; i <= SIGRTMAX; i++)
6        sigaddset (&alarm_sig, i);
7        sigprocmask (SIG_BLOCK, &alarm_sig, NULL);
8
9        Safelet *app = new Safelet();
10       app->initializeApplication();
11       MissionSequencer *threadms = app->getSequencer();
12       threadms->start(NULL);
13       threadms->join();
14       return 0;
15 }
```

Figura 4.15 – Método Principal da aplicação Jantar dos Filósofos.

Na Figura 4.16 é apresentada a classe *Safelet*. O método *getSequencer()* apresentado na linha 13 cria uma *ptread* do *MissionSequencer* que retorna a sequência de missões para execução, ou seja, uma *pthread* de um objeto de MissionSequencer. Os métodos referentes a

manipulação de memória implementados em SCJ não estão no escopo deste trabalho e por este motivo não foi implementada a tradução destes métodos.

```
1  Safelet::Safelet() {
2        //    TODO Auto-generated constructor stub
3  }
4
5  Safelet::~Safelet() {
6        //     TODO Auto-generated destructor stub
7  }
8
9  void Safelet::initializeApplication(){
10       //    do nothing
11 }
12
13 PThreadMissionSequencer *Safelet::getSequencer(){
14       int storage = 100;
15       PThreadMission *_mission = new PThreadMission("Mission_1");
16       PThreadMissionSequencer *thr = new PThreadMissionSequencer(sequencer_priority, storage
             , _mission);
17       return thr;
18 }
```

Figura 4.16 – Classe Safelet da aplicação Jantar dos Filósofos.

O método *run()* da *pthread* do *MissionSequencer* é apresentado na linha 1 da Figura 4.17. É declarada e instanciada uma Missão e é chamado o método *getNextMission()*, apresentado na linha 9, responsável por obter a próxima Missão para execução pertencente a este *MissionSequencer*.

O método *waitForTermination()*, conforme apresentado na linha 19, é responsável por bloquear a execução do *run()* até que seja enviado um sinal para continuar a execução, ou seja, enquanto a execução da Missão criada não seja finalizada.

```
1  void PThreadMissionSequencer::run() {
2        while (true){
3              _mission = getNextMission();
4              _mission->start(arg);
5              waitForTermination();
6              mission_done = false;
7        }
8  }
9  PThreadMission *PThreadMissionSequencer::getNextMission() {
10       PThreadMission *_m = NULL;;
11       if (!mission_done){
12             mission_done = true;
13             _m = m;
14             _m->setMissionSequencer(this);
15       }
16       else
17             return NULL;
18 }
19 void PThreadMissionSequencer::waitForTermination() {
20       pthread_mutex_lock(&_mutex);
21       pthread_cond_wait(&condmutex, &_mutex);
22       pthread_mutex_unlock(&_mutex);
23 }
```

Figura 4.17 – Classe *MissionSequencer* da aplicação Jantar dos Filósofos.

Na Figura 4.18 é apresentado um trecho da classe *Mission* onde na linha 1 pode-se visualizar o método principal de execução da *pthread* definida para traduzir o comportamento

do objeto *Mission* da especificação. Conforme definido na especificação é utilizado um atributo público chamado *phase* para que seja possível acompanhar em todo o escopo da aplicação a sua sequência de execução. Inicialmente é definido que a Missão está em sua fase inicial e chamado o método *initiliaze()* conforme apresentado na linha 5. Após é definido que a Missão está em execução e chamado o método *exec()* na linha 7. Após a finalização da execução, a Missão entra na fase de limpeza e chamado o método *cleanup()* conforme apresentado na linha 9.

O método *initilize()*, apresentado na linha 11, é responsável por declarar e registrar todos os manipuladores de eventos de cada Missão. São instanciados 4 manipuladores de eventos periódicos (PEH) como apresentado nas linhas 12, 14, 16 e 18. Cada manipulador será um filósofo mantendo um comportamento periódico, ou seja, ele periodicamente tentará pegar os garfos para comer.

Na linha 21 é apresentado o método *exec()* responsável pelo código do usuário da Missão e também é responsável por chamar o método *StartAll()* e após isso ele entra em bloqueio e fica aguardando a finalização da execução dos manipuladores periódicos de evento.

O método *StartAll()*, apresentado na linha 27, é responsável por iniciar as *pthreads* dos manipuladores de evento. Os manipuladores de eventos periódicos (PEH), ao chamar o método *start()* inicia a sua execução conforme apresentado nas linhas 3, 5 e 7. O atributo *qtdHandler* é utilizado para armazenar a quantidade de manipulador de eventos periódicos em execução, pois enquanto existir manipuladores de eventos periódicos em execução não é possivel finalizar a execução de uma Missão.

```
1  void PThreadMission::run() {
2        _terminateAll = false;
3        _phase = 0; // INITIAL
4        initialize();
5        _phase = 1; // EXECUTE
6        exec();
7        _phase = 2; //CLEANUP
8        cleanup();
9        _phase = -1; //INACTIVE
10 }
11 void PThreadMission::initialize(){
12       thra = new PThreadPeriodicEventHandler("Filosofo 0", 0 , 2, 0, 100000, 100, 100);
13       thra->reg(this);
14       thrb = new PThreadPeriodicEventHandler("Filosofo 1", 1, 3, 0, 100000, 100, 100);
15       thrb->reg(this);
16       thrc = new PThreadPeriodicEventHandler("Filosofo 2", 2 , 4, 0, 100000, 100, 100);
17       thrc->reg(this);
18       thrd = new PThreadPeriodicEventHandler("Filosofo 3", 3, 5, 0, 100000, 100, 100);
19       thrd->reg(this);
20 }
21 void PThreadMission::exec() {
22       startAll();
23       pthread_mutex_lock(&_mutex);
24       pthread_cond_wait(&condmutex, &_mutex);
25       pthread_mutex_unlock(&_mutex);
26 }
27 void PThreadMission::startAll() {
28       qtdHandler = 0;
29       thra->start();
30       qtdHandler++;
31       thrb->start();
32       qtdHandler++;
33       thrc->start();
34       qtdHandler++;
35       thrd->start();
36       qtdHandler++;
37 }
```

Figura 4.18 – Classe *Mission* da aplicação Jantar dos Filósofos.

Na Figura 4.19 é a apresentada a classe *PeriodicEventHandler* onde na linha 1 é possível verificar o método *run()* da *pthread* do manipulador de evento periódico. Neste método é possível verificar que a cada execução do laço o filósofo pensa, tenta pegar o talher, come e devolve o talher. Na linha 11 é apresentada a chamada do método pegaTalher(), apresentado na linha 25, em que o filósofo tenta comer, ele muda o estado dele para faminto e é chamado o método intencao() na linha 28.

Na linha 11 é possível verificar a chamada do método devolveTalher(), apresentado na linha 32, responsável por desbloquear o recurso alocado no método pegaColher(). É chamado o método intencao() onde verifica se os seus vizinhos estão com intenção de comer. O método intencao() é apresentado na linha 40 e é responsável por verificar se os seus vizinhos não estão comendo e verifica se é possível pegar o garfo para comer.

```
1  void PThreadPeriodicEventHandler::run() {
2        struct periodic_info info;
3        int i = priority - 1;
4        terminate = false;
5        microsleep (inicio);
6        make_periodic (periodo, &info);
7        while (eventCounter < 1000){
8              wait_period (&info);
9              pensar();
10             pegaTalher();
11             comer();
12             devolveTalher();
13             if (terminate)
14                   break;
15       }
16       _mission->qtdHandler--;
17       pthread_mutex_unlock (&mutex);
18       pthread_mutex_unlock(&(mesa_m[id]));
19       if (_mission->qtdHandler == 0)
20             _mission->requestSequenceTermination();
21       else
22             cout << "Ainda␣tem␣PEH␣executando:␣"<< int (_mission->qtdHandler) <<endl;
23 }
24 void PThreadPeriodicEventHandler::pegaTalher (){
25       pthread_mutex_lock(&mutex);
26       estado[id] = FAMINTO;
27       intencao(id);
28       pthread_mutex_unlock(&mutex);
29       pthread_mutex_lock(&(mesa_m[id]));
30 }
31 void PThreadPeriodicEventHandler::devolveTalher (){
32       pthread_mutex_lock (&mutex);
33       printf("Filosofo␣%d␣esta␣pensando\n", id);
34       estado[id] = PENSANDO;
35       intencao(ESQUERDA);
36       intencao(DIREITA);
37       pthread_mutex_unlock (&mutex);
38 }
39 void PThreadPeriodicEventHandler::intencao (int id_filosofo){
40       if( (estado[id_filosofo] == FAMINTO) &&
41       (estado[ESQUERDA] != COMENDO) &&
42       (estado[DIREITA] != COMENDO ) )
43       {
44             printf("Filosofo␣%d␣ganhou␣a␣vez␣de␣comer\n", id_filosofo);
45             estado[ id_filosofo ] = COMENDO;
46             pthread_mutex_unlock(&(mesa_m[id_filosofo]));
47       }
48 }
```

Figura 4.19 – Classe *PeriodicEventHandler* da aplicação Jantar dos Filósofos.

4.4 Testes de desempenho

Nesta seção serão apresentados os resultados de execução da aplicação do Jantar dos Filósofos desenvolvida utilizando a tradução proposta nesta dissertação. Os testes foram efetuados em Raspberry Pi B+ executando um sistema operacinal Raspbian com versão da Kernel Linux 3.18. A aplicação utilizada para efetuar os testes foi a aplicação do Jantar dos Filósofos apresentada na seção 4.3. A execução da aplicação foi inicializada com a execução de três filósofos simultâneos e cada filósofo teve 1000 execuções periódicas com o tempo de 100000 milisegundos. Após cada execução é armazenado o tempo maior, menor e a média entre as execuções de cada manipulador de evento periódico.

Na Figura 4.20 é apresentado gráfico do tempo em relação a quantidade de filósofos

em execução onde é mostrado menor tempo, tempo médio e maior tempo de execução de cada manipulador de evento periódico. Neste primeiro teste, foi utilizado um escolonador FIFO para sua execução, porém, todos os manipuladores de eventos periódicos possuíam a mesma prioridade.

É possível verificar que a execução com até seis filósofos mantém um comportamento temporal não ocorrendo *deadline* conforme a periodicidade instanciada nos manipuladores de eventos periódicos. A partir do sétimo filósofo é possível verificar que iniciam a perda de *deadline*, ou seja, ocorre uma perda temporal da execução. A perda de *deadline* ocorre pois todos os filósofos (manipuladores de eventos periódicos) possuem a mesma prioridade o que acabou levando a um bloqueio na utilização dos recursos de região crítica. Na execução do décimo primeiro filósofo é possível verificar que o maior tempo de execução chegou 50000 milissegundos de perda e na média de 38000 milissegundos.

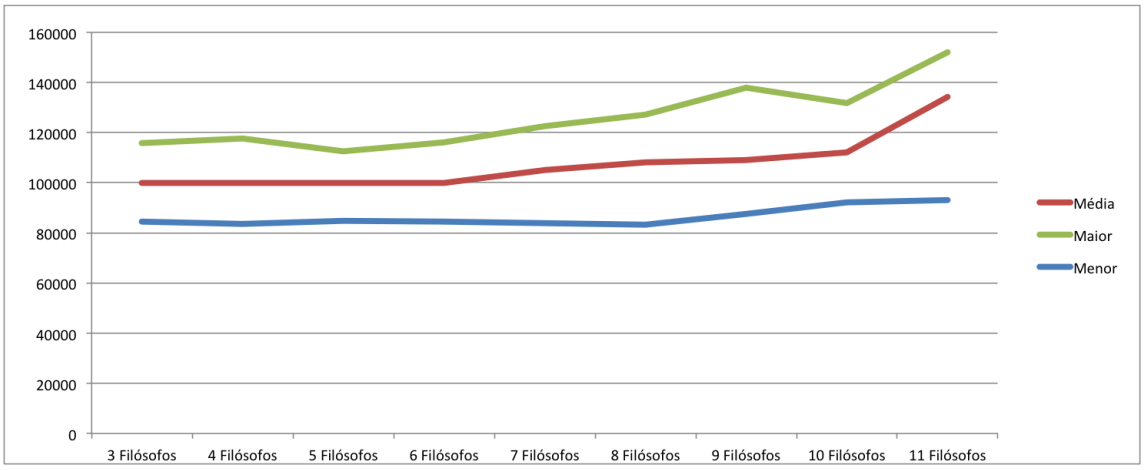

Figura 4.20 – Gráfico de tempo de resposta com os manipuladores com a mesma prioridade.

No segundo teste foi implementado um escalonador FIFO e cada manipulador de eventos possui uma prioridade que varia entre 2 e 12. A prioridade foi incrementada conforme era adicionado um novo filósofo na execução (manipulador de evento periódico). Na Figura 4.21 é apresentado o gráfico do tempo em relação a quantidade de filósofos em execução onde é possível verificar, assim como no teste anterior, o menor tempo, tempo médio e maior tempo de execução de cada manipulador de evento periódico.

É possível verificar que este teste de execução manteve-se mais linear comparado ao primeiro teste, mantendo o comportamento periódico e mantendo a média de suas execuções dentro do limite definido em cada manipulador de evento periódico, ou seja, não ocorrendo *deadline*. Neste caso não ocorreu a perda de *deadline* pois cada filósofo (manipulador de evento

periódico) possui uma prioridade distinta e por este motivo a prioridade de acessar os recursos de região crítica será do filósofo com maior prioridade. Apenas na execução com dez e onze filósofos, houve um aumento do maior tempo de execução chegando aos 9893 milissegundos, porém, a média se manteve abaixo do tempo pré-determinado.

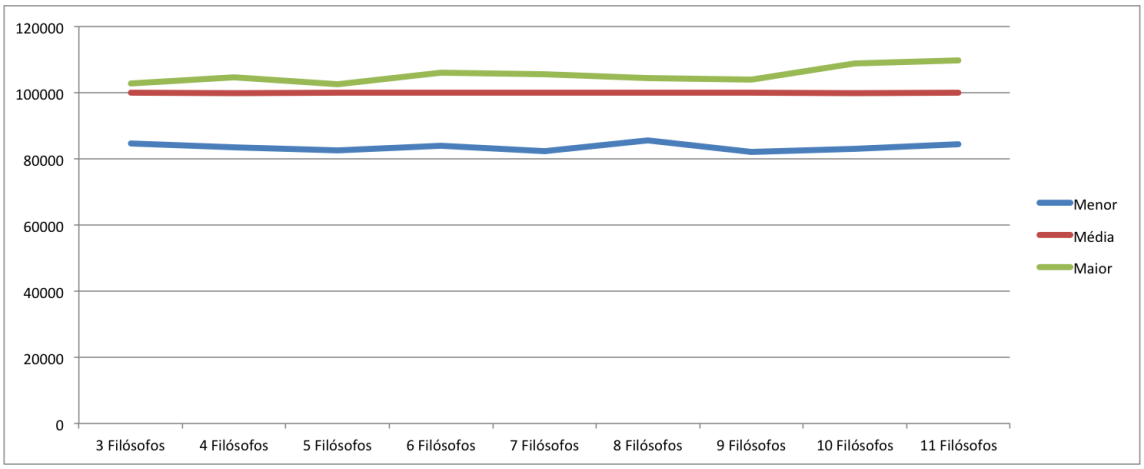

Figura 4.21 – Gráfico de tempo de resposta utilizando um escalonador FIFO.

4.5 Conclusão

Neste capítulo foi apresentada a tradução de alguns exemplos de aplicações SCJ, entre eles a tradução de um exemplo onde é implementado o Jantar dos Filósofos que soluciona um problema clássico da sincronização na utilização de recursos. Também foi desenvolvido alguns testes executando o exemplo em um Raspberry Pi B+ e foi medido o seu comportamento temporal e perdas de *deadline*.

No primeiro teste foi efetuado um teste com diversos filósofos (manipuladores de eventos periódicos) em execução com a mesma prioridade e foi verificado que após a inserção do sexto filósofo iniciou-se a perda de *deadline*, ou seja, a perda do tempo de execução. No segundo teste foi efetuado um teste com diversos filósofos (manipuladores de eventos periódicos) em execução, cada um com uma prioridade, variando de 2 a 12 e foi verificado que não ocorreu perda do deadline pois a média das execuções manteve-se dentro do esperado.

5 CONCLUSÃO

A portabilidade de uma aplicação desenvolvida na linguagem de programação Java é um dos principais fatores dos quais os programadores selecionam esta linguagem para desenvolver aplicações (DAWSON, 2008). Por outro lado, para sua execução é necessária a utilização de uma *Java Virtual Machine* (JVM), porém, existe uma grande dificuldade de encontrar máquinas virtuais de tempo real para sistemas embarcados (DAWSON, 2008). Uma aplicação desenvolvida em C++ pode ser executada diretamente no dispositivo sem a necessidade da utilização de uma máquina virtual (BROSGOL, 2007).

Esta dissertação apresenta uma tradução da especificação *Safety Critical Java* (SCJ) de níível 1 na linguagem de programação C++ com objetivo de manter o comportamento da aplicação para execução em arquiteturas de baixo custo. Uma aplicação desenvolvida na linguagem SCJ necessita de uma máquina virtual (JVM) compatível com *hardware* para sua execução e uma aplicação desenvolvida em C++ é possível sua execução em diversos sistemas operacionais como qualquer distribuição do sistema operacional Linux.

A tradução da especificação foi efetuada inicialmente um estudo do comportamento dos componentes definidos pela especificação *Safety Critical Java* e após foi definido como cada um dos componentes da SCJ seria traduzido obtendo a sintaxe e o comportamento mais próximo possível do definido pela especificação.

Além disso, é possível executar aplicações desenvolvidas utilizando esta tradução em diversos *hardwares*, sem a necessidade de uma máquina virtual, diferentemente de uma aplicação desenvonvida na linguagem de programação Java em cima da especificação *Safety Critical Java*.

No Capítulo 4 são apresentados estudos de caso onde são apresentados a tradução de exemplos de aplicações desenvolvidas utilizando a especificação *Safety Critical Java* de nível 1. No primeiro exemplo traduzido é uma aplicação SCJ onde é possível implementar diversas missões para execução e possui um manipulador de eventos periódicos (*PeriodicEventHandler*) e um manipulador de eventos periódicos *AperiodicEventHandler*.

O segundo exemplo é a tradução da aplicação *Network* em SCJ de nível 1 que implementa o comportamento de uma rede e possui um manipulador de eventos periódicos e quatro manipuladores de eventos periódicos. Os manipuladores de eventos aperiódicos são responsáveis pelos comandos da rede como *SendHandler*, *DisconnectHandler*, *ConnectHandler* e um

manipulador para outros eventos chamado *Handler2*. O manipulador de evento periódico é responsável pela execução destes manipuladores aperiódicos.

O terceiro exemplo é a implementação da aplicação Jantar dos Filósofos em SCJ onde inicialmente existem três filósofos representados por manipuladores de eventos periódicos. Os testes foram efetuados nesta aplicação onde após cada execução é aumentado o número de filósofos sentados a mesa com o objetivo de testar o seu comportamento temporal e verificar a perda de *deadline*.

No primeiro teste foi efetuada a execução da aplicação do Jantar dos Filósofos sem controle de prioridades, ou seja, todos os filósofos (manipuladores de eventos periódicos) com a mesma prioridade. A partir da inserção do sétimo filósofo é possível verificar que iniciam a perda de *deadline*, ou seja, ocorre uma perda temporal da execução. Esta perda temporal ocorreu pois todos os manipuladores de eventos periódicos (*PeriodicEventHandlers*) possuem a mesma prioridade e consequentemente levou a um bloqueio na utilização de recursos de região crítica. Na execução do décimo primeiro filósofo é possível verificar que o maior tempo de execução chegou 50000 milissegundos de perda e na média de 38000 milissegundos.

No segundo teste onde foi utilizado o mesmo exemplo, porém cada filósofo (manipulador de eventos periódicos) possui uma prioridade diferente. É possível verificar que a execução se mantém mais linear comparado ao primeiro teste, mantendo o comportamento periódico e mantendo a média de suas execuções dentro do limite definido em cada manipulador de evento periódico.

Portanto, com a análise dos resultados dos testes é possível verificar que a execução da aplicação traduzida consegue obter um controle temporal da execução. No primeiro teste é possível verificar que ao obter sete manipuladores em execução concorrente ocorreu a perda de *deadline* pois não existe um controle de prioridades. Em ambos os testes foi utilizado um escalonador FIFO, porém no segundo teste cada filósofo (manipulador de eventos periódicos) possui uma prioridade distinta e é possível ver que neste caso não houve perda e *deadline*.

5.1 Trabalhos futuros

Como trabalhos futuros, é necessário a análise da especificação *Safety Critical Java* de nível 2 para a implementação da tradução de aplicações desenvolvidas em SCJ de nível 2. Traduzir aplicações com diversas missões executando simultaneamente e em paralelo para dispositivos embarcados multiprocessados.

Efetuar um estudo sobre controle de inversão de prioridades para evitar com que uma *pthread* de maior prioridade fique bloqueada aguardando a execução de tarefas de menor prioridade. As *Threads* POSIX já possuem protocolos definidos em sua documentação como o Protocolo de Prioridade Teto (*Priotity Ceiling Protocol*) (BARNEY; LIVERMORE, 2014). O Protocolo de Prioridade Teto evita com que uma tarefa entre em uma seção crítica caso existam outras tarefas utilizando recursos que possam bloquear esta tarefa e para isso cada seção crítica tem uma prioridade teto associada que é igual a prioridade da tarefa de maior prioridade que possa acessar o recurso compartilhado (TANENBAUM., 2010).

Com o objetivo de facilitar o processo de tradução de aplicações utilizando a especificação *Safety Critical Java*, vê-se necessário o desenvolvimento de uma aplicação para tradução automatizada de um código desenvolvido utilizando a especificação *Safety Critical Java* na linguagem de programação C++. A aplicação deverá além de traduzir os componentes de especificação SCJ apresentados na Seção 3.1 conforme a proposta apresentada na Seção 3.2, também poderá traduzir os códigos do usuário na linguagem de programação Java. A aplicação desenvolvida deverá ler o código fonte da aplicação desenvolvida em SCJ e criar as classes e métodos traduzidos na linguagem de programação C++.

REFERÊNCIAS

A. PLSEK L. ZHAO, V. H. S. D. T. T. K.; VITEK, J. Developing Java applications with oSCJ/L0. **8th International Workshop on Java Technologies for Real-time and Embedded Systems**, República Tcheca, v.53, n.2, p.95–101, 2010.

ABOWD, G. D. et al. Towards a Better Understanding of Context and Context-Awareness. In: HUC '99: PROCEEDINGS OF THE 1ST INTERNATIONAL SYMPOSIUM ON HANDHELD AND UBIQUITOUS COMPUTING, London, UK. **Anais...** Springer Berlin / Heidelberg, 1999. p.304–307. (Lecture Notes in Computer Science, v.1707).

ANJOS, J. S. G. **Java development platform for real-time applications in multi-core architectures**. 2009. Tese (Doutorado em Ciência da Computação) — Departmento de Engenharia da Informática, Universidade de Lisboa, Lisboa, Portugal.

BARNEY, B.; LIVERMORE, L. **POSIX Threads Programming**. Acesso realizado em Março/2014, Disponível em: https://computing.llnl.gov/tutorials/pthreads/.

BHASKAR, S. **Practical Guide To Pthread Programming in C++**. Acesso realizado em Abril/2014, Disponível em: http://www.polarsparc.com/pdf/PThreads.pdf.

BROSGOL, B. M. Languages for Safety-Critical Software: issues and assessment. In: SOFTWARE ENGINEERING - COMPANION, 2007. ICSE 2007 COMPANION. 29TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2007. p.180–181.

BURNS, A.; WELLINGS, A. **Real-Time Systems and Programming Languages**: ada, real-time java and c/real-time posix. 4th.ed. USA: Addison-Wesley Educational Publishers Inc, 2009.

CAVALCANTI, A. et al. Safety-critical Java in Circus. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, 9., New York, NY, USA. **Proceedings...** ACM, 2011. p.20–29. (JTRES '11).

DAWSON, M. H. Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine. **11th IEEE Symposium on Object Oriented Real-Time Distributed Computing**, Estados Unidos, v.53, n.2, p.241–247, 2008.

DEBIAN OS, S. do. **Página oficial da distribuição Linux Debian**. [S.l.]: Disponível em: https://www.debian.org/index.pt.html. Acesso em: Janeiro, 2014.

E. Y. HU, G. B.; WELLINGS, A. A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems. **8th International Workshop on Java Technologies for Real-time and Embedded Systems**, Estados Unidos, v.53, n.2, p.77–84, 2002.

G. BOLLELLA B. BROSGOL, P. D. S. F. J. G. D. H.; TURNBULL, M. **The Real-Time Specification for Java**. Acesso realizado em Março/2014, Disponível em: http://www.jcp.org/aboutJava/communityprocess/first/jsr001/rtj.pdf.

GHOSH, K.; MUKHERJEE, B.; SCHWAN, K. **A Survey of Real-Time Operating Systems**. [S.l.: s.n.], 1994.

GROUP, T. O. **Standard for Information Technology—Portable Operating System Interface**. Acesso realizado em Setembro/2014, Disponível em: http://www.open-std.org/jtc1/sc22/open/n4217.pdf.

GROUP, T. O. **Safety Critical Java Technology Specification**. Acesso realizado em Agosto/2014, Disponível em: http://jcp.org/en/jsr/detail?id=302.

J2C, T. **Página oficial do projeto**. [S.l.]: Disponível em: https://code.google.com/a/eclipselabs.org/p/j2c/. Acesso em: Agosto, 2014.

LOCKHART, J.; PURDY, C.; WILSEY, P. Formal methods for safety critical system specification. In: CIRCUITS AND SYSTEMS (MWSCAS), 2014 IEEE 57TH INTERNATIONAL MIDWEST SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2014. p.201–204.

LUCKOW, K. S.; THOMSEN, B.; KORSHOLM, S. E. HVMTP: a time predictable and portable java virtual machine for hard real-time embedded systems. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, 12., New York, NY, USA. **Proceedings...** ACM, 2014. p.107:107–107:116. (JTRES '14).

PARK, H. et al. The Cardiac Pacemaker: systemj versus safety critical java. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, 12., New York, NY, USA. **Proceedings...** ACM, 2014. p.37:37–37:46. (JTRES '14).

PIZLO, F.; ZIAREK, L.; VITEK, J. Real Time Java on Resource-constrained Platforms with Fiji VM. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, 7., New York, NY, USA. **Proceedings...** ACM, 2009. p.110–119. (JTRES '09).

RT LINUX, S. do. **Página oficial do Real-Time Linux**. [S.l.]: Disponível em: https://rt.wiki.kernel.org/index.php/. Acesso em: Janeiro, 2014.

SANTOS, O. M. dos. **Run Time Detection of Timing Errors in Real-Time Systems**. 2008. Tese (Doutorado em Ciência da Computação) — Departmento de Ciência da Computação, Universidade de York, York, Reino Unido.

SCHOEBERL, M. et al. A Profile for Safety Critical Java. In: OBJECT AND COMPONENT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2007. ISORC '07. 10TH IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2007. p.94–101.

SILVA, N.; LOPES, R. Independent Assessment of Safety-Critical Systems: we bring data! In: SOFTWARE RELIABILITY ENGINEERING WORKSHOPS (ISSREW), 2012 IEEE 23RD INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2012. p.84–84.

SIMMONDS, C. **Over and over again**: periodic tasks in linux. Acesso realizado em Abril/2014, Disponível em: http://www.2net.co.uk/tutorial/periodic threads.

SOLARIS OS, S. do. **Página oficial do Sistema Operacional Solaris**. [S.l.]: Disponível em: http://www.oracle.com/us/products/servers-storage/solaris/solaris11/overview/index.html. Acesso em: Janeiro, 2015.

SØNDERGAARD, H.; KORSHOLM, S. E.; RAVN, A. P. Safety-critical Java for Low-end Embedded Platforms. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, 10., New York, NY, USA. **Proceedings...** ACM, 2012. p.44–53. (JTRES '12).

STRØM, T. B.; PUFFITSCH, W.; SCHOEBERL, M. Chip-multiprocessor Hardware Locks for Safety-critical Java. In: INTERNATIONAL WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, 11., New York, NY, USA. **Proceedings...** ACM, 2013. p.38–46. (JTRES '13).

SUNANDA, B. E.; SEETHARAMAIAH, P. Modeling of Safety-Critical Systems Using Petri Nets. **SIGSOFT Softw. Eng. Notes**, New York, NY, USA, v.40, n.1, p.1–7, Feb. 2015.

TANENBAUM, A. **Sistemas operacionais modernos**. [S.l.]: Prentice-Hall do Brasil, 2012.

TANENBAUM., A. S. **Sistemas Operacionais Modernos**. 7th.ed. São Paulo, SP, Brasil.: Editora LTC, 2010.

TRANSCOMPILER, T. J. **Página oficial do projeto**. [S.l.]: Disponível em: https://code.google.com/p/java2c-transcompiler/. Acesso em: Setembro, 2014.

YAGHMOUR, K.; MASTERS, J.; BEN, G. **Building Embedded Linux Systems, 2Nd Edition**. 2.ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2008.

ANEXOS

ANEXO A – Exemplo SCJ de nível 1 Multi-Missão

```
1  public class MySafelet implements Safelet {
2        final int MISSION_MEMORY_SIZE = 10000; final int SEQUENCER_PRIORITY = 10;
3
4        public long missionMemorySize()
5        {
6              return MISSION_MEMORY_SIZE;
7        }
8
9        @SCJAllowed(SUPPORT)
10       public long immortalMemorySize() {
11             return 10000;
12       }
13
14       public void intializeApplication()
15       {
16             // do nothing
17       }
18       // Safelet methods
19       @SCJAllowed(SUPPORT)
20       public MissionSequencer getSequencer() {
21             MyMission _mission1 = new MyMission();
22             MyMission _mission2 = new MyMission();
23             MyMission _mission3 = new MyMission();
24             List<MyMission> m = new ArrayList<MyMission>();
25             m.add(_mission1);
26             m.add(_mission2);
27             m.add(_mission3);
28             return new LinearMissionSequencer<Mission>(
29             new PriorityParameters(SEQUENCER_PRIORITY), new StorageParameters(10000, null),
30             m);
31       }
32 }
```

Figura A.1 – Classe Safelet do exemplo Multi-Missão.

```
1  public class MyMissionSequencer extends MissionSequencer {
2        public boolean mission_done;
3        public MainMissionSequencer(PriorityParameters pp, StorageParameters sp, List<
             MyMission> m) {
4              super(pp,sp);
5              mission_done = false;
6        }
7
8        public MyMission getNextMission() {
9              int i=0;
10             if (!mission_done) {
11                   mission_done = true;
12                   for(int i=0;i<m.size();i++)
13                         return m.get(i);
14             }
15             else {
16                   return null;
17             }
18       }
19 }
```

Figura A.2 – Classe MissionSequencer do exemplo Multi-Missão.

```
1  public class MyMission extends Mission {
2        public void initialize(){
3              (new MyPEH("A",new RelativeTime(0,0),new RelativeTime(500,0))).register();
4              (new MyPEH("B",new RelativeTime(0,0),new RelativeTime(1000,0))).register();
5              (new MyPEH("C",new RelativeTime(0,0),new RelativeTime(500,0))).register();
6              (new MyPEH("C",new RelativeTime(0,0),new RelativeTime(500,0))).register();
7              PriorityParameters priority,
8              StorageParameters storage
9        }
10 }
```

Figura A.3 – Classe Mission do exemplo Multi-Missão.

```
1  public class MyPEH extends PeriodicEventHandler {
2        static final int priority = 13, mSize = 10000; int eventCounter;
3        String my_name;
4        public MyPEH(String nm, RelativeTime start, RelativeTime period) {
5              super(new PriorityParameters(priority),
6              new PeriodicParameters(start, period),
7              new StorageParameters(10000, null), 0);
8              my name = nm;
9        }
10       @SCJAllowed(SUPPORT)
11       public void handleAsyncEvent() {
12             ++eventCounter;
13       }
14 }
```

Figura A.4 – Classe PeriodicEventHandler do exemplo Multi-Missão.

```
1  public class MyAPEH extends AperiodicEventHandler {
2        public MyAPEH(PriorityParameters priority,
3        StorageParameters storage) {
4              super(priority, storage, event, "MyAPEH");
5        }
6
7        @SCJAllowed(SUPPORT)
8        public void handleAsyncEvent() {
9              ++eventCounter;
10       }
11 }
```

Figura A.5 – Classe AperiodicEventHandler do exemplo Multi-Missão.

ANEXO B – Exemplo 2 SCJ de nível 1 *Network*

```
1  public class MainSafelet implements Safelet {
2        public void setUp() {
3
4        }
5
6        public MissionSequencer getSequencer() {
7              return new MainMissionSequencer();
8        }
9
10       public void tearDown() {
11
12       }
13 }
```

Figura B.1 – Classe MainSafelet do exemplo *Network*.

```
1  public class MainMissionSequencer extends MissionSequencer {
2        public boolean mission done;
3
4        public MainMissionSequencer() {
5              super(
6              /* Let MainMissionSequencer run at max priority. */
7              new PriorityParameters(
8              PriorityScheduler.instance().getMaxPriority()),
9              new StorageParameters(10000, 10000, 10000));
10             mission done = false;
11       }
12
13       public Mission getNextMission() {
14             if (!mission done) {
15                   mission done = true;
16                   /* Created in Immortal Memory */
17                   return new MainMission();
18             }
19             else {
20                   return null;
21             }
22       }
23 }
```

Figura B.2 – Classe MainMissionSequencer do exemplo *Network*.

```
1  public class MainMission extends Mission {
2        private Events events;
3        private Interrupts interrupts;
4        private Outputs outputs;
5
6        private Network network;
7
8        private List list;
9
10       public void initialize() {
11             network = new Network();
12             list = new List();
13             Handler1 handler1 = new Handler1(list,
14             new PriorityParameters(
15             PriorityScheduler.instance().getNormPriority()),
16             new PeriodicParameters(
17             new AbsoluteTime(0, 0), new RelativeTime(100, 0)),
18             new StorageParameters(4096, 4096, 4096));
19             handler1.register();
20             Handler2 handler2 = new Handler2(network, events,
21             new PriorityParameters(
22             PriorityScheduler.instance().getMaxPriority()),
23             new StorageParameters(4096, 4096, 4096),
24             events.out);
25             handler2.register();
26       }
27
28       public void initArchitecture() {
29             createEvents();
30             createInterrupts();
31             createOutputs();
32       }
33
34       /* Create SCJ Events */
35       public void createEvents() {
36             events = new Events();
37       }
38
39       /* Create Interrupt Handlers */
40       public void createInterrupts() {
41             interrupts = new Interrupts(events);
42             interrupts.register();
43             interrupts.setPriorities();
44       }
45
46       /* Create Output Handlers */
47       public void createOutputs() {
48             outputs = new Outputs(events, network, list);
49             outputs.register();
50       }
51
52       public void cleanup() {
53
54       }
55
56       public long missionMemorySize() {
57             return 131072;
58       }
59 }
```

Figura B.3 – Classe MainMission do exemplo *Network*.

```
1  public class Handler1 extends PeriodicEventHandler {
2        public final static long REGISTER ADDRESS = 0;
3
4        private final RawInt register =
5        RawMemory.createRawIntAccessInstance(
6        RawMemory.IO MEM MAPPED, REGISTER ADDRESS);
7
8        private List list;
9
10       public Handler1(List list,
11       PriorityParameters priority,
12       PeriodicParameters period,
13       StorageParameters storage) {
14             super(priority, period, storage, "Handler1");
15             this.list = list;
16       }
17
18       public void handleAsyncEvent() {
19             /* Read input value from hardware here. */
20             /* Since we cannot use an interrupt handler, and open question is how
21             * we implement this to be carried out at a high priority, and what
22             * design would result in a blocking read that causes that waits for
23             * the hardware to deliver the data. I will try and clarify this with
24             * Andy this week if he is available for a chat. */
25             int value = register.get();
26             System.out.println("[Handler1] input " + value + " received");
27             MissionMemory mission memory =
28             (MissionMemory) MemoryArea.getMemoryArea(this);
29             mission memory.executeInArea(new MissionMemoryEntry(value));
30       }
31
32       /* The insert() method allocates data and hence has to be carried out
33       * in MissionMemory. */
34
35       class MissionMemoryEntry implements Runnable {
36             public int value;
37
38             public MissionMemoryEntry(int value) {
39                   this.value = value;
40             }
41
42             public void run() {
43                   list.insert(value);
44             }
45       }
46 }
```

Figura B.4 – Classe Handler1 do exemplo *Network*.

```
1  public class Handler2 extends AperiodicEventHandler {
2        /* Object to access the network (Resides in MissionMemory). */
3        private Network network;
4
5        /* Object to access the output events. */
6        private Events events;
7
8        public Handler2(Network network, Events events,
9        PriorityParameters priority,
10       StorageParameters storage,
11       AperiodicEvent event) {
12             super(priority, storage, event, "Handler2");
13             this.network = network;
14             this.events = events;
15       }
16
17       public void handleAsyncEvent() {
18             /* I don't see a solution here other than busy wait. */
19             /* The wait() and notify() mechanisms are only support at level 2. */
20             while (network.getState() != NetworkState.DISCONNECTED) { }
21             events.connect.fire();
22             while (network.getState() != NetworkState.CONNECTED) { }
23             events.send.fire();
24             while (network.getState() != NetworkState.SENT) { }
25             events.disconnect.fire();
26       }
27 }
```

Figura B.5 – Classe Handler2 do exemplo *Network*.

```
1  public class DisconnectHandler extends AperiodicEventHandler {
2        private final Network network;
3
4        public DisconnectHandler(Network network,
5        PriorityParameters priority,
6        StorageParameters storage,
7        AperiodicEvent event) {
8              super(priority, storage, event, "DisconnectHandler");
9              this.network = network;
10       }
11
12       public void handleAsyncEvent() {
13             network.disconnect();
14       }
15 }
```

Figura B.6 – Classe DisconnectHandler do exemplo *Network*.

```
1  public class ConnectHandler extends AperiodicEventHandler {
2        private final Network network;
3
4        public ConnectHandler(Network network,
5        PriorityParameters priority,
6        StorageParameters storage,
7        AperiodicEvent event) {
8              super(priority, storage, event, "ConnectHandler");
9              this.network = network;
10       }
11
12       public void handleAsyncEvent() {
13             network.connect();
14       }
15 }
```

Figura B.7 – Classe ConnectHandler do exemplo *Network*.

```
1  public class SendHandler extends AperiodicEventHandler {
2        /* Object to access the network. (Resides in MissionMemory) */
3        private final Network network;
4
5        /* Object used to deduce the output value. (Resides in MissionMemory) */
6        private List list;
7
8        public SendHandler(Network network, List list,
9        PriorityParameters priority,
10       StorageParameters storage,
11       AperiodicEvent event) {
12             super(priority, storage, event, "SendHandler");
13             this.network = network;
14       }
15
16       public void handleAsyncEvent() {
17             /* Here java.util.Set is used in place of the type \power \num. */
18             java.util.Set elems = list.elems();
19             /* The method call below carries out the device access. */
20             network.send(elems);
21       }
22 }
```

Figura B.8 – Classe SendHandler do exemplo *Network*.

```
1  public class Network {
2        private NetworkState state;
3
4        public Network() {
5              state = NetworkState.DISCONNECTED;
6        }
7
8        public synchronized NetworkState getState() {
9              return state;
10       }
11
12       public synchronized void connect() {
13             System.out.println("[Network] connect");
14             assert state == NetworkState.DISCONNECTED;
15             state = NetworkState.CONNECTED;
16       }
17
18       public synchronized void send(java.util.Set set) {
19             System.out.println("[Network] sending " + set.toString());
20             assert state == NetworkState.CONNECTED;
21             state = NetworkState.SENT;
22       }
23
24       public synchronized void disconnect() {
25             System.out.println("[Network] disconnect");
26             assert state == NetworkState.CONNECTED;
27             state = NetworkState.DISCONNECTED;
28       }
29 }
```

Figura B.9 – Classe Network do exemplo *Network*.

ANEXO C – Exemplo 3 SCJ de nível 1 Jantar dos filósofos

```
1  public class MainSafelet implements Safelet {
2
3        public MissionSequencer getSequencer() {
4              return new MainMissionSequencer();
5        }
6
7        public void tearDown() {
8
9        }
10 }
```

Figura C.1 – Classe MainSafelet do exemplo *Jantar dos Filósofos*.

```
1  public class MainMissionSequencer extends MissionSequencer {
2        public boolean mission_done;
3
4        public MainMissionSequencer() {
5              super(
6              /* Let MainMissionSequencer run at max priority. */
7              new PriorityParameters(
8              PriorityScheduler.instance().getMaxPriority()),
9              new StorageParameters(10000, 10000, 10000));
10             mission_done = false;
11       }
12
13       public Mission getNextMission() {
14             if (!mission_done) {
15                   mission_done = true;
16                   return new MainMission();
17             }
18             else {
19                   return null;
20             }
21       }
22 }
```

Figura C.2 – Classe MainMissionSequencer do exemplo *Jantar dos Filósofos*.

```
1  public class MainMission extends Mission {
2
3        List<Garfo>garfos = new Arrays<Garfo>();
4        for (int i = 0; i<=4; i++){
5              Garfo garfo = new Garfo(i);
6              garfos.add(i,garfo);
7        }
8
9        public class MainMission extends Mission {
10             public void initialize(){
11                   (new MyPEH(garfo, 1, "Filosofo␣1",new RelativeTime(0,0),new RelativeTime
                         (500,0))).register();
12                   (new MyPEH(garfo, 2, "Filosofo␣2",new RelativeTime(0,0),new RelativeTime
                         (1000,0))).register();
13                   (new MyPEH(garfo, 3, "Filosofo␣3",new RelativeTime(0,0),new RelativeTime
                         (500,0))).register();
14                   (new MyPEH(garfo, 4, "Filosofo␣4",new RelativeTime(0,0),new RelativeTime
                         (500,0))).register();
15                   (new MyPEH(garfo, 5, "Filosofo␣5",new RelativeTime(0,0),new RelativeTime
                         (500,0))).register();
16             }
17       }
18
19       public void cleanup() {
20
21       }
22 }
```

Figura C.3 – Classe MainMission do exemplo *Jantar dos Filósofos*.

```
1  public class MyPEH extends PeriodicEventHandler {
2        private String nm;
3        RelativeTime period;
4        int id;
5        final int n = 5;
6        List <Garfo> garfos;
7        int filosofo;
8
9        public MyPEH(List <Garfo>garfos, int filosofo, String nm, RelativeTime start,
             RelativeTime period) {
10             super(new PriorityParameters(priority),new PeriodicParameters(start, period),
                   new StorageParameters(10000, null), 0);
11             this.nm = nm;
12             this.garfos = garfos;
13             this.filosofo = filosofo;
14       }
15
16       @SCJAllowed(SUPPORT)
17       public void handleAsyncEvent() {
18             pensar(filosofo);
19             pegaTalher(filosofo, filosofo);
20             pegaTalher((filosofo+1)%n,filosofo);
21             comer(filosofo);
22             devolveTalher(filosofo, filosofo);
23             devolveTalher((filosofo+1)%n,filosofo);;
24       }
25
26       private synchronized void pegaTalher(int pos, int dono){
27             System.out.println(nm+"␣faminto"+ pos);
28             ((Garfo)garfos.get(pos)).setEstadoGarfo(true);
29             ((Garfo)garfos.get(pos)).setDonoGarfo(dono);
30       }
31
32       private synchronized void devolveTalher(int pos, int dono){
33             System.out.println(nm+"␣devolve␣garfo␣"+ pos);
34             ((Garfo)garfos.get(pos)).setEstadoGarfo(false);
35             ((Garfo)garfos.get(pos)).setDonoGarfo(-1);
36       }
37
38       private synchronized void comer(int fil){
39             if (((Garfo)garfos.get(fil)).getEstadoGarfo() &&
40             ((Garfo)garfos.get((fil+1)%n)).getEstadoGarfo() &&
41             ((Garfo)garfos.get(fil)).getDonoGarfo()==fil &&
42             ((Garfo)garfos.get((fil+1)%n)).getDonoGarfo()==fil){
43             System.out.println(nm+"␣comendo");
44       }
45 }
46
47       private synchronized void pensar(int fil){
48             System.out.println(nm+"␣pensando");
49       }
50 }
```

Figura C.4 – Classe MyPEH do exemplo *Jantar dos Filósofos*.

```
1  public class Garfo {
2
3        private int idGarfo;
4        private boolean estadoGarfo;
5        private int dono;
6
7        public Garfo(int id){
8              idGarfo = id;
9              estadoGarfo = false;
10             dono = -1;
11       }
12
13       public int getIdGarfo(){
14             return idGarfo;
15       }
16
17       public void setIdGarfo(int g){
18             idGarfo = g;
19       }
20
21       public int getDonoGarfo(){
22             return dono;
23       }
24
25       public void setDonoGarfo(int d){
26             dono = d;
27       }
28
29       public boolean getEstadoGarfo(){
30             return estadoGarfo;
31       }
32
33       public void setEstadoGarfo(boolean ocupado){
34             estadoGarfo = ocupado;
35       }
36 }
```

Figura C.5 – Classe Garfo do exemplo *Jantar dos Filósofos*.