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

ESQUEMA PARA A TRADUÇÃO DE APLICAÇÕES ENTRE AS LINGUAGENS CIRCUS E SAFETY CRITICAL JAVA

DISSERTAÇÃO DE MESTRADO

Nathan Leidemer

Santa Maria, RS, Brasil

2016

ESQUEMA PARA A TRADUÇÃO DE APLICAÇÕES ENTRE AS LINGUAGENS CIRCUS E SAFETY CRITICAL JAVA

Nathan Leidemer

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

2016

Leidemer, Nathan

Esquema para a tradução de aplicações entre as linguagens Circus e Safety Critical Java / por Nathan Leidemer. – 2016.
69 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, 2016.

1. Sistemas Críticos. 2. Linguagens Formais. 3. Circus. 4. Safety Critical Java. 5. Esquema de Tradução. I. Santos, Osmar Marchi dos. II. Título.

E-mail: nathanl@inf.ufsm.br

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

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

ESQUEMA PARA A TRADUÇÃO DE APLICAÇÕES ENTRE AS LINGUAGENS CIRCUS E SAFETY CRITICAL JAVA

elaborada por
Nathan Leidemer

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

COMISSÃO EXAMINADORA:

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

Andrei Piccinini Legg, Dr. (UFSM)

Reiner Franthesco Perozzo, Dr. (UNIFRA)

Santa Maria, 29 de Março de 2016.

A todos aqueles que, direta ou indiretamente contribuíram para esse trabalho.

AGRADECIMENTOS

Agradeço à minha família por todo o apoio e incentivo durante toda a minha formação.

Ao professor Osmar Marchi dos Santos pela dedicação e apoio que tornaram possível a realização desse trabalho.

Aos professores Andrei Piccinini Legg e Reiner Franthesco Perozzo pela disponibilidade e dedicação na participação na banca desse trabalho.

Aos professores do curso de graduação e mestrado da Universidade Federal de Santa Maria pela contribuição com minha formação.

A Universidade Federal de Santa Maria pela oportunidade, permitindo o meu aperfeiçoamento profissional.

A todos aqueles que, direta ou indiretamente contribuíram para esse trabalho.

“Todos os problemas se tornam infantis, depois de explicados.”
— SHERLOCK HOLMES

RESUMO

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

ESQUEMA PARA A TRADUÇÃO DE APLICAÇÕES ENTRE AS LINGUAGENS CIRCUS E SAFETY CRITICAL JAVA

AUTOR: NATHAN LEIDEMER
ORIENTADOR: OSMAR MARCHI DOS SANTOS
Local da Defesa e Data: Santa Maria, 29 de Março de 2016.

Em sistemas críticos de segurança, missão ou negócios o alto custo das falhas faz com que sejam necessários o uso de métodos e técnicas para garantir a confiabilidade da aplicação. É neste contexto que foram criadas linguagens formais como o Circus ou versões específicas de linguagens como o *Safety-Critical Java* para facilitar a verificação e validação das aplicações criadas e aumentar consequentemente a confiabilidade geral da aplicação. Apesar de aumentar a confiabilidade, os sistemas modelados em linguagens formais não podem ser executados e então precisam ser implementados em uma linguagem de programação tradicional. É nesse processo de livre tradução do sistema especificado onde ocorrem a maioria dos erros que acabam por não garantir que o código gerado esteja de acordo com a especificação. Baseando-se nessa premissa o presente trabalho propõem-se a apresentar uma estratégia de tradução de modelos escritos na linguagem Circus para programas executáveis na linguagem SCJ. Entre os principais objetivos e contribuições do trabalho estão a criação das EBNFs das duas linguagens e a descrição detalhada da tradução de todos os elementos entre as duas linguagens.

Palavras-chave: Sistemas Críticos. Linguagens Formais. Circus. Safety Critical Java. Esquema de Tradução.

ABSTRACT

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

TRANSLATION SCHEME FOR APPLICATIONS BETWEEN THE LANGUAGES CIRCUS AND SAFETY CRITICAL JAVA

AUTHOR: NATHAN LEIDEMER
ADVISOR: OSMAR MARCHI DOS SANTOS
Defense Place and Date: Santa Maria, March 29, 2016.

At safety-critical, mission-critical and business-critical systems the high cost of failure makes required the use of methods and techniques to ensure application reliability. In this context, formal languages, as Circus or specific languages versions like Safety-Critical Java, were created to facilitate the verification and validation of applications so consequently assisting to increase the overall reliability. Despite of the reliability increase, the modeled systems in formal languages can not be executed subsequently has to be implemented in a traditional programming language. It is in this process of free translation where occur most mistakes that end up not ensuring that the generated code conforms to the specification. Based on that premise, this paper propose to expound a strategy of translation from models written in Circus language to executable programs in SCJ language. Among the main objectives and contributions include the creation of EBNFs of the two languages and the detailed description of the translation of all elements between the two languages.

Keywords: Critical Systems. Formal Languages. Circus. Safety Critical Java. Translation Scheme..

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

APEH	*Aperiodic Event Handler*
CSP	*Communicating Sequential Processes*
EBNF	*Extended Backus-Naur Form*
JCSP	*Communicating Sequential Processes for Java*
PEH	*Periodic Event Handler*
RTSJ	*Real-Time Specification for Java*
SCJ	*Safety Critical Java*

SUMÁRIO

1 INTRODUÇÃO

As falhas em aplicações são relativamente comuns e na maioria dos casos essas falhas causam inconveniências, mas não danos sérios a longo prazo, segundo (SOMMERVILLE et al., 2008). Em sistemas críticos de segurança, missão ou negócios porém, o alto custo das falhas faz com que sejam necessários o uso de métodos e técnicas para garantir a confiabilidade da aplicação. É neste contexto que foram criadas linguagens formais como o *Circus* ou versões específicas de linguagens como o *Safety Critical Java* para facilitar a verificação e validação das aplicações criadas e aumentar consequentemente a confiabilidade geral da aplicação.

O *Circus* (WOODCOCK; CAVALCANTI, 2001) é uma linguagem formal cuja sintaxe é formada pela combinação das sintaxes das linguagens Z (WOODCOCK; DAVIES, 1996) e *Communicating Sequential Processes* (CSP) (HOARE, 1978). A linguagem foi criada para possibilitar a descrição de sistemas complexos tanto do ponto de vista de estruturas de dados complexas presentes em Z quanto do ponto de vista da álgebra de processos de CSP.

Durante os anos diversas variações e extensões da linguagem *Circus* foram definidas e incluem o *Circus Time* (WEI; WOODCOCK; CAVALCANTI, 2013) que adiciona os componentes de tempo do CSP à linguagem, o *OhCircus* (CAVALCANTI; SAMPAIO; WOODCOCK, 2005) que adiciona o conceito de orientação à objetos do Java e o *SCJ-Circus* (CAVALCANTI et al., 2013) que adiciona as estruturas do SCJ à linguagem.

O *Safety Critical Java* (SCJ) (LOCKE et al., 2010), por sua vez, é uma versão do Java adequada para o desenvolvimento de *software* de tempo real verificável. Ele incorpora parte do *Real-Time Specification for Java* (RTSJ) (BOLLELLA, 2000) e introduz novas abstrações como os *Safelets* e as *Missions* e remove os coletores de lixo ao impor a utilização de regiões de memória com escopo definido. Essas modificações garantem comportamentos de tempo previsível para a aplicação.

Ao modelar uma aplicação em uma linguagem formal como o *Circus*, é possível atingir uma profundidade de análise através de uma gama de ferramentas e métodos existentes que seriam extremamente complicadas em outras linguagens. Dessa forma as linguagens formais aumentam a confiabilidade dos sistemas criados, garantindo que eles satisfaçam um conjunto de propriedades e requisitos.

Apesar de aumentar a confiabilidade, os sistemas modelados em linguagens formais não podem ser executados e então precisam ser implementados em uma linguagem de programação

tradicional. É nesse processo de livre tradução do sistema especificado onde ocorrem a maioria dos erros que acabam por não garantir que o código gerado esteja de acordo com a especificação.

Baseando-se nessa premissa o presente trabalho busca primeiramente estudar as linguagens *Circus* e SCJ para posteriormente propor um esquema para a tradução das especificações criadas em *Circus* para código executável em uma aplicação SCJ.

1.1 Escopo e Principais Contribuições

Uma aplicação na linguagem SCJ pode ser escrita em três níveis (nível 0, nível 1 e nível 2) onde o nível 0 suporta aplicações mais simples e o nível 2 as mais complexas. O nível 0 permite a escrita de aplicações de execução cíclica, enquanto o nível 1 é focado em aplicações concorrentes e o nível 3 é focado em sistemas multinúcleo.

Este trabalho propõem-se a apresentar uma estratégia de tradução de modelos escritos na linguagem *Circus* para programas executáveis na linguagem SCJ. Escolheu-se como base as aplicações em SCJ de nível 1 (foco em concorrência) abrindo mão de conceitos como alocação de memória e prioridades presentes na linguagem SCJ mas que até o presente momento ainda não foram adicionados na linguagem *SCJ-Circus* via refinamento. Esses conceitos podem ser incorporados ao resultado do trabalho posteriormente.

Entre os principais objetivos e contribuições do trabalho estão a criação das EBNFs das duas linguagens e a descrição detalhada da tradução de todos os elementos entre as duas linguagens. Como passos para alcançar os objetivos propostos, procurou-se:

1. estudar a sintaxe e estrutura da linguagem *Circus*;
2. estudar a sintaxe e estrutura da linguagem *SCJ*;
3. estudar a linguagem *SCJ-Circus* criada a partir do refinamento da linguagem *Circus*;
4. revisar trabalhos que já propuseram a tradução de linguagens formais para código;
5. criar as estruturas EBNF das duas linguagens e detalhar o processo de tradução entre elas;
6. criar estudos de caso para demonstrar a viabilidade do esquema proposto para a tradução.

1.2 Organização do Texto

Este trabalho está estruturado em cinco Capítulos. No Capítulo 2 será apresentada um revisão da literatura que serve como base para o trabalho, serão abordadas as linguagens SCJ, *Circus* e *SCJ-Circus* além de um estudo de trabalhos relacionados que já propuseram a tradução de linguagens formais para código executável. No Capítulo 3 é apresentada a estratégia de tradução adotada no trabalho apresentando a EBNF das duas linguagens e o processo de tradução entre elas. No Capítulo 4 serão apresentados três estudos de caso para ilustrarem o processo de tradução proposto na seção anterior. Por fim, no Capítulo 5 serão apresentadas as conclusões do trabalho juntamente com algumas sugestões de trabalhos futuros.

2 REVISÃO DA LITERATURA

Este Capítulo tem como objetivo apresentar ao leitor o embasamento necessário para a compreensão do esquema de tradução proposto nesta dissertação. Na Seção 2.1 será apresentada uma visão geral da linguagem SCJ focando-se nas especificações de nível 1. Na Seção 2.2 será apresentada uma visão geral da linguagem *Circus* e na Seção 2.3 a extensão da linguagem que inclui os elementos do SCJ na linguagem *Circus* chamada *SCJ-Circus*. Por fim, na Seção 2.4 serão apresentados alguns trabalhos relacionados que já propuseram a tradução de aplicações descritas em linguagens formais para código executável.

2.1 Safety Critical Java

O SCJ (LOCKE et al., 2010) é uma versão do Java adequada para o desenvolvimento de software de tempo real verificável. Ele incorpora parte do RTSJ (BOLLELLA, 2000) e introduz novas abstrações como *Safelets* e *Missions* e remove os coletores de lixo ao impor a utilização de regiões de memória com escopo definido. Essas modificações garantem comportamentos de tempo previsível para a aplicação.

Conforme apresentado em (LOCKE et al., 2010) no SCJ uma aplicação é composta de uma ou mais missões. Cada missão consiste de um conjunto limitado de objetos escalonáveis definidos pelo RTSJ. Para cada missão um bloco de memória é criado (*mission memory* e os objetos criados são mantidos neste bloco até que a missão tenha terminado.

Cada missão começa na fase de inicialização (*initialization phase*) onde os objetos são alocados pela aplicação na memória da missão e na memória imortal. Ao finalizar a etapa de inicialização a aplicação entra então na fase de execução (*execution phase*), onde os dados das memórias são consultados mas geralmente não são criados novos objetos.

A complexidade das aplicações críticas de segurança apresentam grandes variações. Diversas aplicações contém apenas uma *thread*, suportam apenas uma função e possuem restrições de tempo simples. Outras, porém, são altamente complexas tendo múltiplos modos de operação, diversas missões aninhadas e devem satisfazer restrições de tempo complexas. Para resolver esse problema e simplificar o processo de desenvolvimento e certificação de uma aplicação desenvolvida em SCJ foram criados três níveis (nível 0, nível 1 e nível 2) onde o nível 0 suporta as aplicações mais simples e o nível 2 as mais complexas.

O modelo de programação no Nível 0 pode ser descrito como um modelo de linha do

tempo, um modelo baseado em quadros ou um modelo de execução cíclica. Neste modelo cada missão pode ser considerada um conjunto computável que é executado periodicamente.

O modelo de programação de Nível 1, foco deste trabalho, pode ser descrito como um modelo multitarefa que consiste de uma única missão executada periodicamente com um conjunto de execuções concorrentes, com prioridades definidas e controladas por um único escalonador preemptivo. A execução é realizada em um conjunto de *PeriodicEventHandler* (PEH) e/ou *AperiodicEventHandler* (APEH), com o uso das memórias de missão e imortal. Os métodos *Object.wait* e *Object.notify* não estão disponíveis.

Na Figura 2.1 está ilustrada a execução de uma aplicação simples no Nível 1, incluindo sua alocação de memória. São mostrados três objetos escalonáveis (SO1, SO2 e SO3), cada um com uma prioridade e memória privada que é liberada após sua execução. O escalonador de prioridade fixa executa os objetos conforme sua ordem de prioridade, quando um objeto está pronto para execução ele pode antecipar sua execução sobre um objeto de menor prioridade, como mostrado quando SO3 de prioridade 7 antecipa sua execução antes de SO2 de prioridade 2.

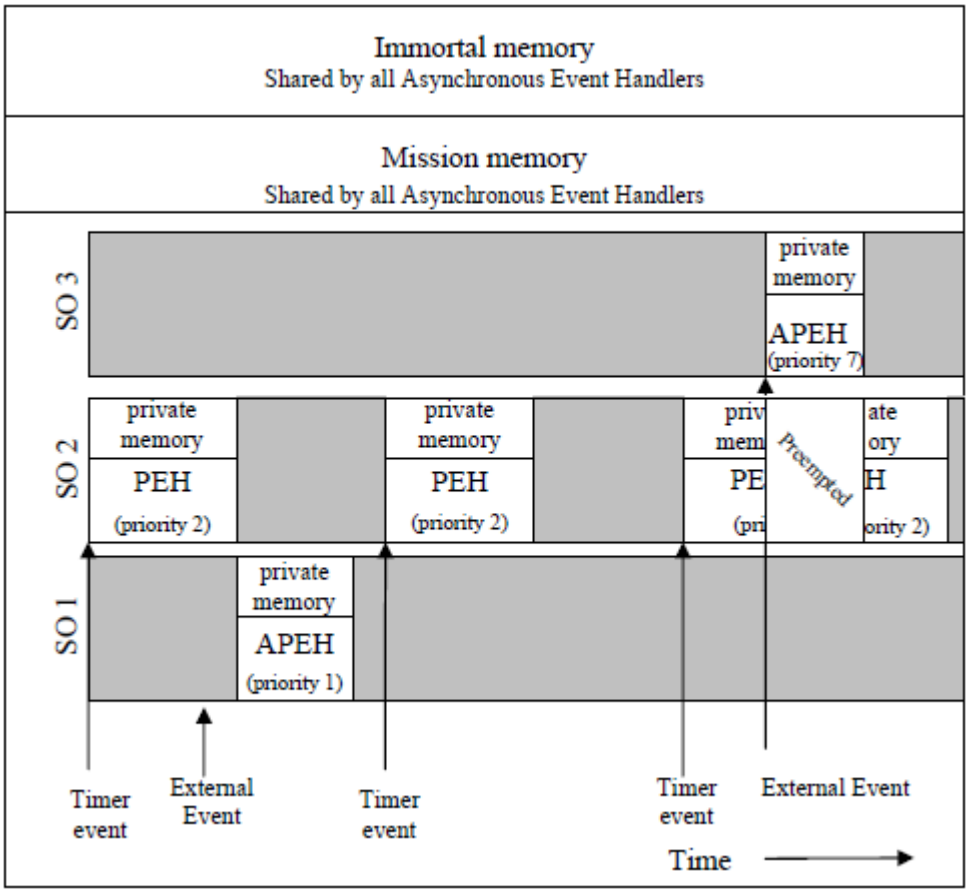

Figura 2.1 – Nível 1. Fonte: (LOCKE et al., 2010).

Uma aplicação de Nível 2 começa com uma única missão, mas pode criar e executar outras missões concorrentemente com a missão inicial. A execução é realizada em um conjunto de objetos escalonáveis consistindo de PEHs, APEHs e/ou *threads* de tempo real sem pilha. Cada missão possui sua memória de missão própria e pode criar e executar outras sub missões.

Uma aplicação SCJ é formada por uma sequência de execuções de missões. Cada missão é composta pelas fases de inicialização, execução e limpeza conforme ilustrado na figura 2.2.

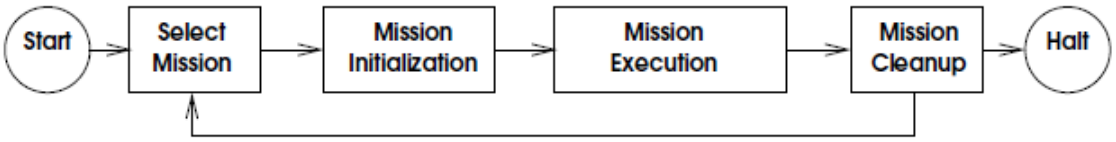

Figura 2.2 – Fases de uma aplicação. Fonte: (LOCKE et al., 2010).

Na fase de **inicialização da aplicação** uma aplicação em SCJ é definida pelo usuário com a interface *Safelet*. A aplicação define o método *setUp* que cria objetos na *ImmortalMemoryArea* e realiza outras ações como a inicialização de dispositivos de *hardware*. A aplicação também define o método *getSequencer* que retorna o *MissionSequencer* que representa a aplicação. O *MissionSequencer* possui um conjunto de missões definidas pelo usuário com a classe *Mission*.

Na etapa de **inicialização da missão** o *MissionSequencer* invoca o método de inicialização de cada missão. Esse método inicializa e registra todos os objetos *ManagedSchedulable* relativos à missão. Todos os objetos do tipo *ManagedSchedulable* devem ser criados e registrados durante a etapa de inicialização.

Na **execução da missão** as missões serão executadas sob o controle do *MissionSequencer*, seguindo a order retornada pelo método *getNextMission*. Por padrão, as alocações de memória são feitas nesta etapa alteram a *PrivateMemory*, mas é possível também trabalhar com a *ImmortalMemoryArea* ou a *MissionMemory*. Antes que uma missão possa ser finalizada é preciso aguardar que todos os *managedSchedulables* associados à missão completem sua execução.

Na etapa de **limpeza da missão** a aplicação define o método *cleanup*. Essa etapa pode ser usada para liberar recursos, sendo necessária em missões finalizadas assincronamente por missões pai. Por fim a fase de **limpeza da aplicação** ocorre ao término da execução da aplicação onde o método *tearDown* é chamado e todos os recursos usados pela aplicação são liberados.

2.2 Circus

O *Circus* (WOODCOCK; CAVALCANTI, 2001) é uma linguagem formal cuja sintaxe é formada pela combinação das sintaxes das linguagens Z (WOODCOCK; DAVIES, 1996) e CSP (HOARE, 1978). A linguagem foi criada para possibilitar a descrição de sistemas complexos tanto do ponto de vista de estruturas de dados complexas presente em Z quanto do ponto de vista da álgebra de processos de CSP.

Conforme apresentado em (FREITAS, 2005) um programa em *Circus*, assim como em uma especificação em Z, é formado por uma sequência de parágrafos. Esses parágrafos podem ser um parágrafo em Z, uma definição de canal, um conjunto de canais ou a declaração de um processo. A sintaxe simplificada de um programa em *Circus* é apresentada na Figura 2.3.

$$
\begin{array}{lcl}
\mathit{Program} & ::= & \mathit{CircusPar}^* \\
\mathit{CircusPar} & ::= & \mathit{ZParagraph} \mid \mathit{ChanDecl} \mid \mathit{CSetDecl} \mid \mathit{ProcDecl} \\
\mathit{ChanDecl} & ::= & \mathbf{channel}\ \mathit{CDecl} \\
\mathit{SeqCDecl} & ::= & \mathit{CDecl} \mid \mathit{SeqCDecl};\ \mathit{CDecl} \\
\mathit{CDecl} & ::= & N^+ \mid N^+ : \mathit{ZExpr} \\
\mathit{CSDecl} & ::= & \mathbf{chanset}\ N == \mathit{CSExpr} \\
\mathit{CSExpr} & ::= & \{\!|\,|\!\} \mid \{\!|\ N^+\ |\!\} \mid N \mid \mathit{CSExpr} \setminus \mathit{CSExpr} \\
 & \mid & \mathit{CSExpr} \cup \mathit{CSExpr} \mid \mathit{CSExpr} \cap \mathit{CSExpr} \\
\mathit{PDecl} & ::= & \mathbf{process}\ N\ \widehat{=}\ \mathit{ProcDef} \\
\mathit{PDef} & ::= & \mathbf{begin}\ \mathit{PPar}^*\ \mathbf{state}\ \mathit{Schema\text{-}Expr}\ \mathit{PPar}^* \bullet \mathit{Act}\ \mathbf{end} \\
\mathit{PPar} & ::= & \mathit{ZParagraph} \mid N\ \widehat{=}\ \mathit{Act} \mid \mathit{NameSetDecl} \\
\mathit{Act} & ::= & \mathit{Schema\text{-}Expr} \mid \mathit{CSPAct} \\
\mathit{CSPAct} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathit{Comm} \rightarrow \mathit{Act} \mid \mathit{Cmd} \mid \mathit{ZPred}\ \&\ \mathit{Act} \\
 & \mid & N \mid \mu\ N \bullet \mathit{Act} \mid \mathit{Act}\,;\ \mathit{Act} \mid \mathit{Act} \sqcap \mathit{Act} \mid \mathit{Act} \,\Box\, \mathit{Act} \\
 & \mid & \mathit{Act}\ [\![\ \mathit{NSExpr} \mid \mathit{CSExpr} \mid \mathit{NSExpr}\ ]\!]\ \mathit{Act} \mid\ \mathit{Act} \setminus \mathit{CSExpr} \\
 & \mid & |||\ \mathit{ZDecl} \bullet \mathit{Act} \\
\mathit{Cmd} & ::= & \mathbf{var}\ x : \mathit{ZExpr} \bullet \mathit{Act} \mid N^+ := \mathit{ZExpr}^+ \\
\mathit{Comm} & ::= & N\ \mathit{CParam}^* \\
\mathit{CParam} & ::= & ?\,N \mid ?\,N : \mathit{ZPred} \mid !\,\mathit{ZExpr} \mid .\,\mathit{ZExpr} \\
\mathit{NSDecl} & ::= & \mathbf{nameset}\ N == \mathit{NSExpr} \\
\mathit{NSExpr} & ::= & \{\,\} \mid \{\,N^+\,\} \mid N \mid \mathit{NSExpr} \setminus \mathit{NSExpr} \\
 & \mid & \mathit{NSExpr} \cup \mathit{NSExpr} \mid \mathit{NSExpr} \cap \mathit{NSExpr}
\end{array}
$$

Figura 2.3 – BNF simplificada da sintaxe *Circus*. Fonte: (FREITAS, 2005).

Os canais apresentados na sintaxe como *ChanDecl* são a interface entre o sistema e o ambiente externo. Eles são declarados seguindo a sintaxe da linguagem CSP: a palavra-chave *channel*, o nome do canal e o tipo do valor que está sendo comunicado. A declaração de tipos na linguagem *Circus* segue o sistema de tipos da linguagem Z. Como em CSP é possível ainda declarar um canal omitindo a declaração de tipo, neste caso o canal serve apenas como evento

de sincronização.

A declaração de um processo é formada pelo seu nome e a especificação do processo. A especificação mais básica de um processo define o estado do processo, uma sequência de parágrafos de processo e uma ação principal que define o comportamento do processo delimitados pelas palavras-chave *begin* e *end*. Um parágrafo de processo pode ser um parágrafo de Z, uma definição de ação ou uma definição de *nameset*.

Uma ação em *Circus* pode ser uma expressão *schema*, um comando, uma invocação à uma ação definida anteriormente ou uma combinação dessas estruturas usando operadores CSP. Três ações primitivas estão disponíveis: *Skip*, *Stop* e *Chaos*.

2.3 SCJ-Circus

O *SCJ-Circus* (CAVALCANTI et al., 2013) estende a sintaxe do *OhCircus* (CAVALCANTI; SAMPAIO; WOODCOCK, 2005) e do *Circus Time* (WEI; WOODCOCK; CAVALCANTI, 2013) com parágrafos que permitem a especificação de *safelet*, *mission sequencers*, *missions* e *handlers*. A sintaxe simplificada de um programa em *SCJ-Circus* é apresentada na Figura 2.4.

```
SCJProgram      ::=   SCJParagraph*
SCJParagraph    ::=   Safelet | MissionSequencer | Mission | Handler | CircusParagraph
Safelet         ::=   safelet N ≙ begin
                        SCJSSafeletProcessParagraph*
                        state Schema-Expression
                        SCJSafeletProcessParagraph*
                        initialize ≙ SCJSafeletAction
                        SCJSafeletProcessParagraph*
                        getSequencer ≙ res return : sequencer • SCJSafeletAction
                        SCJSafeletProcessParagraph*
                      end
```

Figura 2.4 – BNF simplificada da sintaxe *SCJ-Circus*. Fonte: (CAVALCANTI et al., 2013).

Como apresentado em (CAVALCANTI et al., 2013), em geral um programa em *SCJ-Circus* é formado por uma sequencia de *SCJParagraphs* que podem ser um parágrafo de *Circus* ou uma declaração de *safelet*, *mission sequencer*, *mission* ou *handler*. A estrutura de cada uma das abstrações especificas do SCJ é determinada pelos valores e comportamentos que devem ser especificadas para uma aplicação de acordo com o padrão SCJ. Por exemplo, um *safelet* deve implementar o método *initialize* que permite a alocação de objetos globais na área de

memória imortal, o método *getSequencer* que gera um sequenciador de missões e o método *immortalMemorySize* que define o tamanho da memória imortal alocada pelo programa.

A sintaxe dos parágrafos *mission sequencer*, *mission* e *handler* do *SCJ-Circus* são similares, fornecendo componentes para a especificação de estados (*state*), construtores (*initial*) e os métodos que são definidos pelo desenvolvedor.

2.4 Trabalhos Relacionados

Diversos trabalhos já foram realizados buscando traduzir programas escritos em linguagens formais para código executável. No trabalho "*Synthesis of C++ software from verifiable CSPm specifications*"(DOXSEE; GARDNER, 2005) os autores propõem um *framework* para a tradução de programas da linguagem formal CSP para código em C++. No trabalho os autores utilizam-se de uma definição de linguagem intermediária chamada CSPm que incorpora alguns elementos básicos da linguagem C++ na sintaxe do CSP. Da mesma forma no presente trabalho é utilizada a especificação *SCJ-Circus* que incorpora elementos do SCJ na linguagem *Circus* para possibilitar a tradução.

Outro trabalho com uma temática semelhante propõem a tradução especificações da linguagem CSP para código em C#. No trabalho "*Generating C# Programs from CSP# Models*"(ZHU et al., 2013) os autores utilizam-se também da especificação de linguagem intermediária chamada CSP# para traduzir os seus modelos para código em C#.

No trabalho "*From Circus to Java: Implementation and Verification of a Translation Strategy*"(FREITAS; CAVALCANTI, 2006) os autores propõem uma ferramenta para a tradução de especificações na linguagem *Circus* para código em Java. A estratégia de tradução faz uso da biblioteca *Communicating Sequential Processes for Java* (JCSP) para implementar as construções do CSP presentes na linguagem Circus focando-se nos conceitos de processos. A grande diferença do presente trabalho é o foco em sistemas críticos, sendo utilizada então a linguagem SCJ como objetivo da tradução para garantir comportamentos de tempo previsível para a aplicação.

2.5 Conclusão

Neste Capítulo, foi apresentado um estudo sobre a linguagem SCJ com o enfoque em aplicações de nível 1 que serão o objetivo das traduções propostas neste trabalho. Foram apre-

sentadas ainda as linguagens *Circus* e o *SCJ-Circus* que introduz os conceitos do SCJ na linguagem *Circus* e que servirá como ponto de partida para o esquema de tradução proposto no próximo Capítulo. Foram ainda apresentados trabalhos relacionados, onde os dois primeiros propuseram a tradução de código em CSP para as linguagens C# e C++ enquanto o último parte também da linguagem *Circus* mas tem como objetivo final a criação de código na linguagem JCSP ao invés da linguagem SCJ usada neste trabalho.

3 ESTRATÉGIA DE TRADUÇÃO

Neste Capítulo será apresentada a estratégia adotada para a tradução de aplicações modeladas em *Circus* para código em Safety-Critical Java. A tradução de cada um dos elementos básicos de uma aplicação SCJ é apresentada em cada uma das seções subsequentes: *Safelet* (Seção 3.1), *Mission Sequencer* (Seção 3.2), *Mission* (Seção 3.3), *Handlers* (Seção 3.4) e *Aperiodic Event* (Seção 3.5).

Para cada uma dessas estruturas foi definida a estrutura EBNF em *Circus* e a sua EBNF correspondente em SCJ, com o detalhamento completo da transformação de cada elemento em uma tradução "um para um". Os terminais das EBNFs das linguagens estão representados em negrito ou contidos entre aspas como pode ser visto na Figura 3.1.

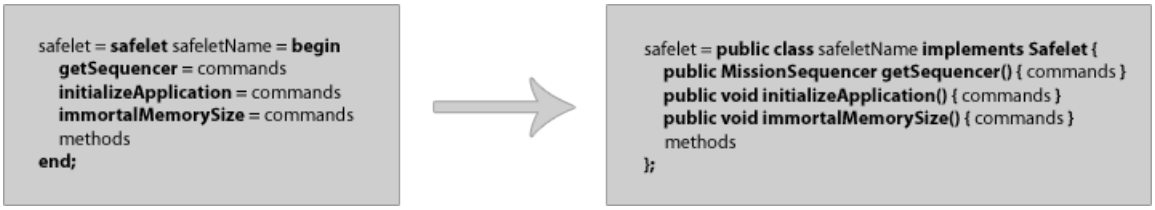

Figura 3.1 – Modelo do esquema de tradução.

Estão presentes ainda as seções 3.6 onde está descrito o processo de tradução de classes oriundas da linguagem *OhCircus* e as seções 3.7, 3.8 e 3.9 que abordam respectivamente as declarações de identificadores, declarações e comandos.

3.1 Safelet

Em um programa SCJ cada aplicação é representada por uma implementação de *Safelet*. Nele será definido o *MissionSequencer* que é o responsável por executar a sequência de missões críticas da aplicação.

Um *Safelet* em Circus, como pode ser visto na Figura 3.2, é composto por três funções básicas: *getSequencer*, *initializeApplication* e *immortalMemorySize*. Essas funções são obrigatórias, pois precisam necessariamente ser implementadas na aplicação SCJ gerada após a tradução. Adicionalmente é possível ainda a definição de métodos que terão sua sintaxe detalhada posteriormente.

safelet = **safelet** safeletName $\widehat{=}$ **begin**
getSequencer $\widehat{=}$ commands
initializeApplication $\widehat{=}$ commands
immortalMemorySize $\widehat{=}$ commands
methods
end ;

Figura 3.2 – EBNF do Safelet em Circus

O *Safelet* em SCJ, apresentado na Figura 3.3, é necessariamente uma implementação de *Safelet*. Os métodos *getSequencer*, *initializeApplication* e *immortalMemorySize* são redefinidos pois são métodos abstratos na interface. É possível ainda a definição de métodos adicionais.

safelet = **public class** safeletName **implements Safelet {**
public MissionSequencer getSequencer() { commands **}**
public void initializeApplication() { commands **}**
public void immortalMemorySize() { commands **}**
methods
} ;

Figura 3.3 – EBNF do Safelet em SCJ

O processo de tradução de um *Safelet* entre as linguagens Circus e SCJ é simples já que as estruturas são similares nas duas definições de EBNF o que resulta em um processo de tradução direto.

3.2 Mission Sequencer

O *MissionSequencer* declarado dentro do *Safelet* será o responsável por controlar a execução das missões na aplicação SCJ. A sequência pode incluir execuções intercaladas de missões independentes ou execuções cíclicas de missões.

Como apresentado na Figura 3.4, um *MissionSequencer* em Circus é composto pela função *getNextMission* que deve ser obrigatoriamente declarada, pois deve estar presente no código gerado em SCJ após a tradução. Ainda estão presentes as estruturas *state* e *initial*, a primeira que será responsável pela declaração dos atributos e a segunda pelo construtor da classe. Adicionalmente podem ainda ser declarados métodos na classe *MissionSequencer*.

```
sequencer = sequencer sequencerName ≙ begin
    state sequencerStateName == variableDeclarations
    initial ≙ variableInitializations
    getNextMission ≙ commands
    methods
end ;
```

Figura 3.4 – EBNF do Sequencer em Circus

O *MissionSequencer* em SCJ, apresentado na Figura 3.5, é uma extensão da classe *MissionSequencer*. Como este trabalho não possui em seu escopo o trabalho com as prioridades e o gerenciamento de memória da linguagem SCJ o comando *super* presente no construtor do *sequencer* é traduzido com os parâmetros de armazenamento fixos e a prioridade alternando entre máxima caso periódico e mínima caso aperiódico.

O método *getNextMission* deve ser redefinido por tratar-se de método abstrato da classe *MissionSequencer*. Podem estar presentes ainda a declaração de métodos adicionais.

```
sequencer = public class sequencerName extends MissionSequencer {
    variableDeclarations
    public sequencerName() {
      super(
        new PriorityParameters( priorityDecl ),
        new StorageParameters(10000, 10000, 10000)
      );
      variableInitializations
    }
    public Mission getNextMission() {
      commands
    }
    methods
};
```

Figura 3.5 – EBNF do Sequencer em SCJ

3.3 Mission

Dentro de cada *MissionSequencer* poderão ser declaradas diversas missões. As missões são as estruturas responsáveis por encapsular as diversas ações da aplicação.

Uma missão padrão em Circus é mostrada na Figura 3.6 e é composta por um *state* onde ocorrerá a declaração de atributos. Estão presentes ainda os métodos *initialize* e *cleanup*, o primeiro responsável pela execução de comandos no início da missão e o segundo responsável

pela execução ao término da missão. Podem ainda ser declarados métodos utilizando a sintaxe padrão.

```
mission = mission missionName ≙ begin
    state missionStateName == variableDeclarations
    initialize ≙ commands
    cleanup ≙ commands
    methods
end;
```

Figura 3.6 – EBNF do Mission em Circus

Em SCJ a missão é declarada como uma extensão da classe *Mission* como apresentado Figura 3.7. Os métodos *initialize* e *cleanup* devem ser definidos por serem métodos abstratos da classe *Mission*.

```
mission = public class missionName extends Mission {
    variableDeclarations
    public void initialize() {
      commands
    }
    public void cleanup() {
      commands
    }
    methods
};
```

Figura 3.7 – EBNF do Mission em SCJ

3.4 Handler

Na Figura 3.8 é apresenta a sintaxe dos *handlers* em Circus, ambos os tipos (periódico e aperiódico) foram definidos usando a mesma EBNF.

A primeira diferença das demais estruturas é a inclusão do elemento *handlerType* que pode ser do tipo *aperiodic* (sem passagem de parâmetros) ou *periodic* (que recebe parâmetros de inicialização). O *handler* é composto por um *state* onde ocorre a declaração de atributos, um método *initial* onde ocorre a inicialização do *handlers* e suas variáveis e o *handleAsyncEvent* responsável pela execução principal do *handler*. Ainda é possível declarar métodos adicionais usando a sintaxe padrão.

```
handler = handlerType handler handlerName ≙ begin
   state handlerStateName == variableDeclarations
   initial handlerInitName ≙
     [ handlerInitName' ; handlerParameters | variableInitializations ]
   handleAsyncEvent untypedParameters ≙
     commands
   methods
end;
```

Figura 3.8 – EBNF do Handler em Circus

Em SCJ a classe *handler* é uma extensão de um *AperiodicEventHandler* ou de um *PeriodicEventHandler* como apresentado na Figura 3.9. As variáveis declaradas no *state* em Circus são declaradas como atributos da classe em SCJ. A função *initial* é traduzida para o construtor da classe e os seus comandos estarão no corpo do construtor. A função *handleAsyncEvent* é traduzida para o método *handleAsyncEvent* de forma direta.

```
handler = public class handlerName extends handlerType {
   variableDeclarations
   public handlerName () {
     variableDeclarations
   }
   public void handleAsyncEvent() {
     commands
   }
   methods
};
```

Figura 3.9 – EBNF do Handler em SCJ

3.5 Aperiodic Event

Um *AperiodicEvent* em Circus, apresentado na Figura 3.10, é composto por um *state* onde ocorre a declaração dos atributos, um *initial* onde ocorre a inicialização do evento e um conjunto de métodos adicionais.

```
event = aperiodicEvent eventName ≙ begin
   state eventStateName == variableDeclarations
   initial eventInitName ≙ variableInitializations
   methods
end;
```

Figura 3.10 – EBNF do Aperiodic Event em Circus

Na Figura 3.11 é apresentado o *AperiodicEvent* em SCJ que é uma extensão de *AperiodicEvent*. As variáveis declaradas em *state* em Circus são traduzidas para atributos da classe em SCJ. As inicializações feitas em *initial* são traduzidas para o construtor da classe e os métodos declarados são traduzidos usando a sintaxe apropriada.

```
class = public class eventName extends AperiodicEvent {
    variableDeclarations
    public eventName ( ) {
      super( );
      variableInitializations
    }
    methods
};
```

Figura 3.11 – EBNF do Aperiodic Event em SCJ

3.6 Classes

Outra estrutura presente em um programa Circus e que pode ser traduzida para SCJ são as classes usando a sintaxe da especificação *OhCircus* apresentada em (CAVALCANTI; SAMPAIO; WOODCOCK, 2005). Uma classe em Circus, como apresentado na Figura 3.12, é composta por um *state* onde ocorrerá a declaração de atributos e um *initialize* cujo corpo será traduzido para o construtor da classe. Podem ainda ser declarados métodos utilizando a sintaxe padrão e classes internas.

```
class = class className ≙ begin
    state classStateName == variableDeclarations
    initial classInitName ≙ variableInitializations
    methods
    classes
end;
```

Figura 3.12 – EBNF da Classe em Circus

Na Figura 3.13 é apresentada a classe em SCJ declarada como pública. As variáveis declaradas em *state* em Circus são traduzidas para atributos da classe em SCJ. As inicializações feitas em *initial* são traduzidas para o construtor da classe e os métodos declarados são traduzidos usando a sintaxe apropriada.

```
class = public class className {
    variableDeclarations
    public className () {
      variableInitializations
    }
    methods
    classes
};
```

Figura 3.13 – EBNF da Classe em SCJ

3.7 Identificadores

Em Circus, como pode ser visto na Figura 3.14, os nomes de métodos, classes, missões e outras estruturas foram mantidos como entradas separadas na EBNF para facilitar o processo de tradução mas todos são identificadores. O identificador é iniciado por uma letra e uma sequência de letras, dígitos ou caracteres especiais. Os tipos das variáveis na linguagem estão declarados em *varType* e os tipos de retorno em *returnType*. Um valor em Circus pode ser um identificador, número, *true* ou *false*.

```
safeletName = identifier;
sequencerName = identifier;
sequencerStateName = identifier;
missionName = identifier;
missionStateName = identifier;
handlerName = identifier;
handlerStateName = identifier;
handlerInitName = identifier;
eventName = identifier;
eventStateName = identifier;
eventInitName = identifier;
className = identifier;
classStateName = identifier;
classInitName = identifier;
methodName = identifier;

varType = “Bool” | “List” | “Long” | “Int” . . . ;
returnType = varType | “void”;
value = identifier | number | “true” | “false”;
identifier = letter , { letter | digit | “ _ ” | “ ’ ” | “ . ” };
number = [ “ - ” ] , digit , { digit };
letter = “A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” | “J” | “K” | “L” | “M” | “N” | “O” | “P” |
“Q” | “R” | “S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z”;
digit = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”;
```

Figura 3.14 – EBNF dos Identificadores em Circus

Em SCJ assim como em Circus os nomes de métodos, classes, missões e outras estruturas são declarados separadamente como identificadores para facilitar o processo de tradução como visto na Figura 3.15. Um identificador é iniciado por uma letra e uma sequência de letras, dígitos ou caracteres especiais. Os tipos possíveis para a variáveis estão em *varType* e os tipos de retorno em *returnType*. Um valor pode ser um identificador, número, *true* ou *false*.

```
safeletName = identifier;
sequencerName = identifier;
sequencerStateName = identifier;
missionName = identifier;
missionStateName = identifier;
handlerName = identifier;
handlerStateName = identifier;
handlerInitName = identifier;
eventName = identifier;
eventStateName = identifier;
eventInitName = identifier;
className = identifier;
classStateName = identifier;
classInitName = identifier;
methodName = identifier;

varType = “Boolean” | “List” | “int” | “long” | . . . ;
returnType = varType | “void”;
value = identifier | number | “true” | “false”;
identifier = letter , { letter | digit | “ _ ” | “ ’ ” | “ . ” };
number = [ “ - ” ] , digit , { digit };
letter = “A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” | “J” | “K” | “L” | “M” | “N” | “O” | “P” |
“Q” | “R” | “S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z”;
digit = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”;
```

Figura 3.15 – EBNF dos Identificadores em SCJ

3.8 Declarações

A sintaxe para as declarações em Circus é apresentada na Figura 3.16. Uma declaração de variável em Circus (*variableDeclaration*) segue o padrão "nome:tipo"e pode ser declarada em uma lista separando os elementos por ponto-e-vírgula e delimitando a lista por colchetes. A inicialização de uma variável (*variableInitialization*) pode ser declarada no padrão "nome:=valor"para uma atribuição comum, "nome:=*new* identifier"para a criação e atribuição de um objeto ou ainda pela palavra chave *setMaxCeiling* inserida na linguagem para possibilitar a tradução de seu elemento correspondente em SCJ.

A inicialização de um novo *handler* (*handlerInitialization*) é feita utilizando as palavras chave *new Handler* seguidas pelo nome do *handler* e seus parâmetros de inicialização entre parênteses. As entradas *handlerType*, *handlerInitParameter*, *startTime* e *periodTime* são usadas na declaração do *handler* já apresentada na Figura 3.8. O *handler* pode ainda receber a passagem de parâmetros em seu construtor (*parameterDeclaration*) seguindo a sintaxe

"nome?:tipo"formando uma lista separada por ponto-e-vírgula.

A declaração de um método (*method*) é composta pelo escopo do método (*methodScope*), opcionalmente o valor *syncronized*, o tipo de retorno, o nome do método e o símbolo de definição. Os comandos que estarão contidos no corpo do método são declarados entre colchetes.

```
variableDeclarations = "["variableDeclaration { ";" variableDeclaration } "]";
variableDeclaration = identifier ":" varType;
variableInitializations = variableInitialization { ";" variableInitialization };
variableInitialization = identifier ":=" value | identifier ":=" "new" varType | setMaxCeiling;
handlerInitializations = "(" handlerInitialization ";" { handlerInitialization ";" } ")";
handlerInitialization = "new" "Handler" handerName "(" untypedParameters ")";
handlerType = "periodic" handlerInitParameters | "aperiodic";
handlerInitParameters = "(" startTime "," periodTime ")";
startTime = number;
periodTime = number;
handlerParameters = parameterDeclaration { ";" parameterDeclaration };
parameterDeclaration = identifier? ":" varType;
untypedParameters = "("identifier { "," identifier "," } ")";
methods = [method] { ";" method ";" };
method = methodScope [ "synchronized" ] returnType methodName ≙ "[" commands "]";
methodScope = "public" | "private";
classes = [class] { ";" class ";" };
```

Figura 3.16 – EBNF das Declarações em Circus

A sintaxe para as declarações em SCJ é apresentada na Figura 3.17. Uma declaração de variável em SCJ (*variableDeclaration*) segue o padrão "*private* tipo nome"e pode ser declarada em uma lista separando os elementos por ponto-e-vírgula. A inicialização de uma variável (*variableInitialization*) pode ser declarada no padrão "nome=valor"para uma atribuição comum, "nome=*new* identifier()"para a criação e atribuição de um objeto ou ainda pela palavra chave *setMaxCeiling* que é traduzido para o comando *Services.setCeiling* da arquitetura do SCJ.

A inicialização de um novo *handler* (*handlerInitialization*) é feita utilizando a palavra chave *new* seguida pelo nome do *handler* e seus parâmetros de inicialização entre parênteses. Caso o *handler* seja do tipo periódico os dois últimos parâmetros passados para o construtor da classe serão o tempo inicial e o tempo do período. As entradas *handlerType*, *handlerParameter* são usadas na declaração do *handler* já apresentada na Figura 3.9.

A declaração de um método (*method*) é composta pelo escopo do método (*methodScope*), opcionalmente o valor *syncronized*, o tipo de retorno, o nome do método e o símbolo de definição. Os comandos que estarão contidos no corpo do método são declarados entre colche-

tes.

```
variableDeclarations = variableDeclaration { ";" variableDeclaration };
variableDeclaration = "private" varType identifier;
variableInitializations = variableInitialization { ";" variableInitialization };
variableInitialization = identifier "=" value | identifier "=" "new" varType "()" | setMaxCeiling;
setMaxCeiling = Services.setCeiling(this, PriorityScheduler.instance().getMaxPriority());
handlerInitializations = "(" handlerInitialization ";" { handlerInitialization ";" } ")";
handlerInitialization = "( new" handlerName "(" untypedParameters ").register();"
handlerType = "PeriodicEventHandler" | "AperiodicEventHandler";
handlerParameters = "(" variableDeclaration { ";" variableDeclaration } ")";
untypedParameters = "("identifier { "," identifier "," } ")";
priorityDecl = "PriorityScheduler.instance().getMaxPriority()" |
               "PriorityScheduler.instance().getNormPriority()";
methods = [method] { ";" method ";" };
method = methodScope [ "synchronized" ] returnType methodName "()" "{" commands "}";
methodScope = "public" | "private";
classes = [class] { ";" class ";" };
```

Figura 3.17 – EBNF das Declarações em SCJ

3.9 Comandos

Na Figura 3.18 são apresentados os comandos mapeados da linguagem Circus. O comando de espera (*commandWait*) é definido pela palavra chave *wait* e pode ser temporal quando acompanhado de um número em nanosegundos ou condicional quando acompanhado por uma expressão que irá interromper a execução do programa até que a condição seja verdadeira. O comando de retorno (*commandReturn*) pode retornar *null*, um identificador ou um número.

O comando condicional (*commandIf*) é declarado na sintaxe padrão da linguagem Circus com duas expressões, dois blocos de comandos e os termos chave *if,* $\rightarrow$*,* $\Box$ e *fi*. A expressão (*expression*) utilizada como condição do comando pode comparar a igualdade ou a diferença de uma variável com *true* ou *false*.

O comando de impressão (*commandPrint*) é formado pela palavra *print* com o valor a ser impresso entre aspas. Uma chamada de método (*commandMethods*) é realizado usando o identificador do objeto e seus parâmetros entre parenteses (*untypedParameter*).

```
commands = Skip | variableInitializations | handlerInitializations | commandIf |
           commandWait | commandReturn | commandPrint | commandMethods ...
commandWait = "wait" number | "wait" expression
commandReturn = "ret" ":=" null | "ret" ":=" identifier | "return" number;
commandIf = "if" expression "→" commands "□" expression "→" commands "fi"
expression = identifier "=" expression | identifier "<>" expression | "true" | "false";
commandPrint = "print" " (' " identifier " ') "
commandMethods = identifier untypedParameters";" | identifier"."identifier untypedParameters";"
```

Figura 3.18 – EBNF dos Comandos em Circus

Na Figura 3.19 são apresentados os comandos traduzidos para a linguagem SCJ. O comando de espera (*commandWait*) é traduzido para o comando *nanoSpin* do SCJ quando for do tipo temporal ou para um comando *while* para a espera condicional. O comando de retorno (*commandReturn*) pode retornar *null*, a criação de um objeto apartir do identificador em Circus ou um número.

O comando condicional (*commandIf*) é declarado na sintaxe padrão da linguagem com duas expressões, dois blocos de comandos e os termos chave *if* e *else* delimitados por chaves. A expressão (*expression*) utilizada como condição do comando pode comparar a igualdade ou a diferença de uma variável com *true* ou *false*.

O comando de impressão (*commandPrint*) é formado pela palavra *print* com o valor a ser impresso entre aspas. Uma chamada de método (*commandMethods*) é realizado usando o identificador do objeto e seus parâmetros entre parenteses (*untypedParameter*).

```
commands = "System.out.println("Skip");" | variableInitializations | commandIf |
           commandWait | commandReturn | commandPrint | commandMethods ...
commandWait = "nanoSpin" "(" number ")" | "while" "(" expression ")" "{" "}" ;
commandReturn = "return null ;" | "return new" identifier "();" | "return" number ";";
commandIf = "if" "(" expression ")" "" commands "" "else" "if" "(" expression ")" "" commands ""
expression = identifier "==" expression | identifier "!=" expression | "true" | "false";
commandPrint = "System.out.println" " (' " identifier " '); "
commandMethods = identifier untypedParameters";" | identifier"."identifier untypedParameters";"
```

Figura 3.19 – EBNF dos Comandos em SCJ

3.10 Conclusão

Neste Capítulo foi apresentado o esquema de tradução proposto para transformar aplicações em *Circus* para código executável em SCJ. A estratégia usada para a tradução foi a criação da EBNF para cada uma das duas linguagens e o detalhamento da tradução de cada um dos

elementos. Foram apresentados os elementos *Safelet*, *Mission Sequencer*, *Mission*, *Handlers* e *Aperiodic Event* oriundos da linguagem *SCJ-Circus*, a declaração de classes da linguagem *OhCircus* além de identificadores, declarações e comandos.

4 ESTUDOS DE CASO

Neste Capítulo serão apresentados alguns exemplos clássicos em *Circus* e o processo para a geração do código correspondente na linguagem SCJ após aplicada a estratégia de tradução proposta no Capítulo 3. O Capítulo propõem-se a demostrar que seguindo os passos descritos no Capítulo anterior com o uso das EBNFs é possível gerar código em SCJ.

Atualmente a execução do código em SCJ de nível 1 não é possível pela falta de uma máquina virtual compatível para isso, uma alternativa para essa limitação será proposta na Seção de trabalhos futuros. Na Seção 4.1 será apresentado o exemplo clássico de sincronização do Jantar dos Filósofos, na Seção 4.2 será apresentado um exemplo básico simulando o compartamento de uma rede (*Network*), enquanto que na Seção 4.3 será apresentado o exemplo de um *Persistent Signal*.

4.1 Jantar dos Filósofos

Nesta Seção será demostrado o processo de tradução do exemplo Jantar dos Filósofos, um problema clássico de sincronização criado por Dijkstra em 1965 e apresentado em (TANENBAUM, 2012). Como pode ser visto na Figura 4.1, o problema consiste de uma mesa circular onde sentam um determinado número de filósofos, cada um com seu prato e um grafo entre dois pratos. Os filósofos podem comer ou pensar, mas para comer cada filósofo precisa usar dois garfos. É preciso então encontrar uma forma para evitar que situações como quando todos os filósofos ficam travados (cada um pegou um garfo e não soltou mais).

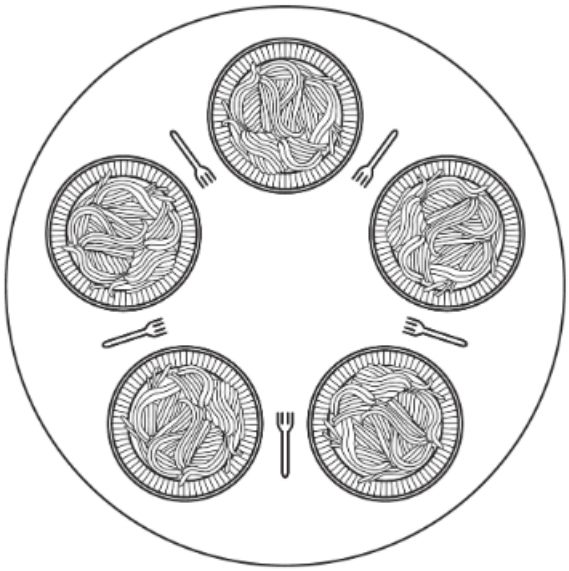

Figura 4.1 – Jantar dos Filósofos. Fonte: (TANENBAUM, 2012).

A aplicação foi modelada em *SCJ-Circus* com o uso de cinco classes como pode ser visto no diagrama de classes da Figura 4.2. A aplicação é iniciada com um *Safelet* com nome *MainSafelet* que declara um *Mission Sequencer* chamado *MainMissionSequencer*. O *Mission Sequencer* por sua vez inicia uma missão chamada *MainMission* que é a responsável por criar os quatro garfos declarados como classes e os quatro filósofos declarados como *Handlers* periódicos.

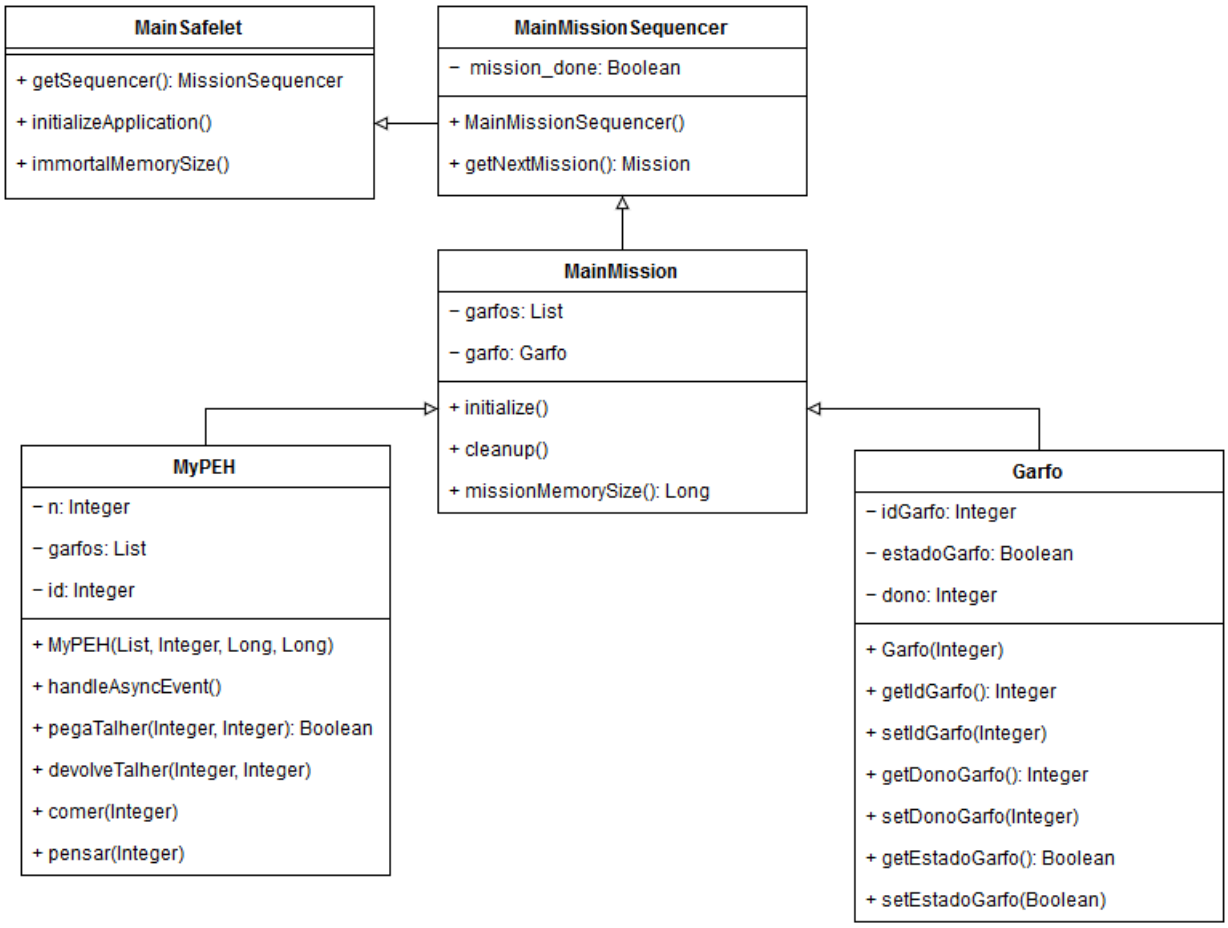

Figura 4.2 – Diagrama de Classes do Jantar dos Filósofos.

4.1.1 Safelet

Conforme pode ser visto na Figura 4.3 o *Safelet* do Jantar dos Filósofos contém a função *getSequencer* que inicializa um novo *Mission Sequencer*. São declaradas ainda as funções *initializeApplication* e *immortalMemorySize* apenas com um comando *Skip*.

```
safelet MainSafelet ≙ begin
    getSequencer ≙ ret := MainMissionSequencer
    initializeApplication ≙ Skip
    immortalMemorySize ≙ Skip
end
```

Figura 4.3 – Jantar dos Filósofos: *Safelet* em Circus

O *safelet MainSafelet* em Circus é traduzido para a classe *MainSafelet* que implementa *Safelet* na Figura 4.4. A função *getSequencer* é traduzida para o método *getSequencer* que retorna um novo *sequencer* do tipo *MainSequencer*.

As funções *initializeApplication* e *immotalMemorySize* são ambas compostas por um comando *Skip* e são traduzidas diretamente para os métodos *initializeApplication* e *immotalMemorySize* vazios em SCJ.

```
public class MainSafelet implements Safelet {
   public MissionSequencer getSequencer() {
     return new MainMissionSequencer();
   }
   public void initializeApplication() { }
   public void immortalMemorySize() { }
}
```

Figura 4.4 – Jantar dos Filósofos: *Safelet* em SCJ

4.1.2 Classe

Na Figura 4.5 é apresentada a classe *Garfo* que contém as funções *state* onde três atributos são declarados e *initial* onde o valor dos atributos é definido. Estão presentes ainda os métodos *getIdGarfo*, *getDonoGarfo* e *getEstadoGarfo* que retornam o valor dos atributos e os métodos *setIdGarfo*, *setDonoGarfo* e *setDonoGarfo* que recebem valores via parâmetros e atualizam os atributos.

```
class Garfo ≙ begin
   state GarfoState == [ idGarfo : int; estadoGarfo : Bool; dono : int ]
   initial GarfoInit ≙
     [ GarfoState'; id? : int | idGarfo := id?; estadoGarfo := false; dono := -1; ]
   public int getIdGarfo ≙ [ ret := idGarfo; ]
   public void setIdGarfo ≙ [ g? : int | idGarfo := g?; ]
   public int getDonoGarfo ≙ [ ret := dono; ]
   public void setDonoGarfo ≙ [ d? : int | dono := d?; ]
   public Bool getEstadoGarfo ≙ [ ret := estadoGarfo; ]
   public void setEstadoGarfo ≙ [ ocupado? := Bool | estadoGarfo = ocupado?; ]
end
```

Figura 4.5 – Jantar dos Filósofos: Classe em Circus

A classe *Garfo* é traduzida para a classe *Garfo* na Figura 4.6. O estado *GarfoState* em Circus composto pela declaração de três atributos é traduzido para a declaração dos atributos *idGarfo*, *estadoGarfo* e *dono* dos tipos *int* e *boolean*. A função *initial* por sua vez é traduzida para o construtor padrão da classe que recebe um parâmetro e inicializa os atributos.

Os métodos *getIdGarfo*, *getDonoGarfo* e *getEstadoGarfo* são traduzidos para os métodos de mesmo nome em SCJ apenas com o comando de retorno. Os métodos *setIdGarfo*,

setDonoGarfo e *setEstadoGarfo* são traduzidos para os métodos de mesmo nome com a atribuição dos parâmetros nos atributos da classe.

```
public class Garfo {
   private int idGarfo;
   private boolean estadoGarfo;
   private int dono;
   public Garfo(int id) {
      idGarfo = id;
      estadoGarfo = false;
      dono = -1;
   }
   public int getIdGarfo(){ return idGarfo; }
   public void setIdGarfo(int g){ idGarfo = g; }
   public int getDonoGarfo(){ return dono; }
   public void setDonoGarfo(int d){ dono = d; }
   public boolean getEstadoGarfo(){ return estadoGarfo; }
   public void setEstadoGarfo(boolean ocupado){ estadoGarfo = ocupado; }
}
```

Figura 4.6 – Jantar dos Filósofos: Classe em SCJ

4.1.3 Mission Sequencer

Na Figura 4.7 é apresentado o *Mission Sequencer* do Jantar dos Filósofos que contém a função *getNextMission* que inicializa uma nova missão caso ainda não esteja definida ou retorna nulo. Além disso estão presentes as funções *state* onde um atributo é declarado e *initial* onde o valor do atributo é definido.

```
sequencer MainMissionSequencer ≙ begin
   state MainMissionSequencerState == [ mission_done : Bool ]
   initial ≙ [ MainMissionSequencerState'; mission_done := false ]
   getNextMission ≙
      if mission_done = false →
         mission_done := true;
         ret := MainMission;
      □
         return null;
      fi
end
```

Figura 4.7 – Jantar dos Filósofos: *Mission Sequencer* em Circus

O *MissionSequencer MainMissionSequencer* é traduzido para a classe *MainMissionSequencer* que estende *MissionSequencer* na Figura 4.8. A função *getNextMission* é traduzida

para o método *getNextMission* que retorna uma nova missão do tipo *Mission* caso ela ainda não tenha sido criada ou retorna o valor *null*.

O estado *MainMissionSequencerState* em Circus composto pela declaração de um atributo é traduzido para a declaração do atributo *mission_done* do tipo *boolean*. A função *initial* por sua vez é traduzida para o construtor padrão da classe com a inicialização do atributo *mission_done* para falso.

```
public class MainMissionSequencer extends MissionSequencer {
   public boolean mission_done;
   public MainMissionSequencer() {
     super(
       new PriorityParameters(
       PriorityScheduler.instance().getMaxPriority()),
       new StorageParameters(10000, 10000, 10000));
     mission_done = false;
   }
   public Mission getNextMission() {
     if (mission_done == false) {
       mission_done = true;
       return new MainMission();
     } else {
       return null;
     }
   }
}
```

Figura 4.8 – Jantar dos Filósofos: *Mission Sequencer* em SCJ

4.1.4 Mission

A missão do Jantar dos Filósofos é apresentada na Figura 4.9. Ela é composta por um *state* onde são declarados dois atributyos e a função *cleanup* apenas com um comando *Skip* em seu corpo. Além disso é declarada a função *missionMemorySize* que retorna um valor inteiro e a função *initialize* onde é criada e populada uma lista de Garfos e são declarados quatro *Handlers* do tipo periódico.

```
mission MainMission ≙ begin
   state missionState == [ garfos : List; garfo : Garfo ]
   initialize ≙
      garfos := new ArrayList;
      garfo := new Garfo(0);
      garfos.add(garfo);
      garfo := new Garfo(1);
      garfos.add(garfo);
      garfo := new Garfo(2);
      garfos.add(garfo);
      garfo := new Garfo(3);
      garfos.add(garfo);
      new Handler MyPEH(garfos,0,0,500);
      new Handler MyPEH(garfos,1,0,1000);
      new Handler MyPEH(garfos,2,0,500);
      new Handler MyPEH(garfos,3,0,500);
   cleanup ≙ Skip
   missionMemorySize ≙ ret := 1000000
end
```

Figura 4.9 – Jantar dos Filósofos: *Mission* em Circus

A estrutura *mission* com nome *MainMission* em Circus é traduzida para a classe *MainMission* que estende *Mission* na Figura 4.10. Os dois atributos declarados no estado *missionState* são traduzidos para os atributos garfos e garfo da classe SCJ.

A função *initialize* com os comandos de atribuição, chamada de métodos e quatro declarações de *handlers* são traduzidos respectivamente para o método *initialize* com os comandos de atribuições, chamadas de métodos e a declaração e registro dos *handlers*.

A função *cleanup* composta pelo comando *Skip* é traduzida para o método *cleanup* vazio. Por fim a função *missionMemorysSize* com um comando de retorno de um valor inteiro é traduzida para o método *missionMemorySize* com o retorno correspondente.

```
public class MainMission extends Mission {
   private List garfos;
   private Garfo garfo;
   private void initialize() {
      garfos = new ArrayList();
      garfo = new Garfo(0);
      garfos.add(garfo);
      garfo = new Garfo(1);
      garfos.add(garfo);
      garfo = new Garfo(2);
      garfos.add(garfo);
      garfo = new Garfo(3);
      garfos.add(garfo);
      (new MyPEH(garfos,0,0,500)).register();
      (new MyPEH(garfos,1,0,1000)).register();
      (new MyPEH(garfos,2,0,500)).register();
      (new MyPEH(garfos,3,0,500)).register();
   }
   public void cleanup() {}
   public long missionMemorySize() { return 1000000; }
}
```

Figura 4.10 – Jantar dos Filósofos: *Mission* em SCJ

4.1.5 Handler

Na Figura 4.11 é apresentado o *periodic handler* que conta com a declaração de três atributos na função *state*. Na função *initial* os atributos são inicializados com os valores recebidos como parâmetro do construtor. Está definida a função *handleAsyncEvent* que contém algumas chamadas de métodos e os comandos *wait* condicionais. O código contém ainda as funções *pegaTalher*, *devolveTalher*, *comer* e *pensar* que recebem parâmetros, realizam chamadas de métodos, retornos e impressões.

```
periodic(start,period) handler MyPEH ≙ begin
  state MyPEHState == [ n : int; garfos : List; id : int ]
  initial MyPEHInit ≙
    [MyPEHState';
    garfos? : List; id? : int; |
    this.garfos := garfos?; this.id := id?; n := 4 ]
  handleAsyncEvent ≙
    pensar(id);
    wait pegaTalher(id,id) = false;
    wait pegaTalher((id+1)%n,id) = false;
    comer(id);
    devolveTalher(id, id);
    devolveTalher((id+1)%n,id);
  public synchronized Bool pegaTalher ≙ [
    pos? : int; dono? : int |
    print("O seguinte filósofo pega o talher "+ pos?);
    if garfos.get(pos).getEstadoGarfo() = false →
      garfos.get(pos)).setEstadoGarfo(true);
      garfos.get(pos)).setDonoGarfo(dono);
      return true;
    □
      return false;
    fi ]
  public synchronized void devolveTalher ≙ [
    pos? : int; dono? : int |
    print("O seguinte filósofo devolve o talher "+ pos?);
    garfos.get(pos).setEstadoGarfo(false);
    garfos.get(pos).setDonoGarfo(-1); ]
  public synchronized void comer ≙ [
    fil? : int | print("O seguinte filósofo está comendo "+ fil); ]
  public synchronized void pensar ≙ [
    fil? : int | print("O seguinte filósofo está pensando "+ fil); ]
end
```

Figura 4.11 – Jantar dos Filósofos: *Periodic Handler* em Circus

Na Figura 4.12 o *handler* periódico possui os valores de inicialização (*start* e *period*) e o nome *MyPEH* e é traduzido para a classe *MyPEH* que estende um *PeriodicEventHandler*. Os atributos declarados no *state* são traduzidos como atributos da classe em SCJ.

Os comandos presentes no *initial* de Circus e seus parâmetros são traduzidos para o construtor da classe em SCJ. O construtor ainda recebe os parâmetros de inicialização (*start* e *period*) como parâmetros. A função *handleAsyncEvent* possui seus comandos traduzidos de forma direta para as chamadas de método e o comando *wait* condicionar que é traduzido para o *while* em SCJ.

A função *pegaTalher* recebe dois parâmetros,possui um comando *print* que é traduzido para *println*, o comando *if-else* é traduzido usando a sintaxe apropriada e em seu interior são realizadas chamadas de métodos e o comando de retorno. As funções *devolveTalher*, *comer* e *pensar* recebem valores via parâmetro, possuem comandos *print* e chamadas de métodos na sintaxe e tradução padrão.

```
public class MyPEH extends PeriodicEventHandler {
    private int n;
    private List garfos;
    private int id;
    public MyPEH(List garfos, int id, long start, long period){
      super(
        new PriorityParameters(PriorityScheduler.instance().getMaxPriority()),
        new PeriodicParameters(start,period),
        new StorageConfigurationParameters(10000, null), 0
      );
      this.garfos = garfos;
      this.id = id;
      n = 4;
    }
    public void handleAsyncEvent() {
      pensar(id);
      while(pegaTalher(id,id)==false);
      while (pegaTalher((id+1)%n,id)==false);
      comer(id);
      devolveTalher(id, id);
      devolveTalher((id+1)%n,id);
    }
    public synchronized boolean pegaTalher(int pos, int dono) {
      System.out.println("O seguinte filósofo pega o talher "+ pos);
      if(((Garfo)garfos.get(pos)).getEstadoGarfo() == false){
        ((Garfo)garfos.get(pos)).setEstadoGarfo(true);
        ((Garfo)garfos.get(pos)).setDonoGarfo(dono);
        return true;
      } else { return false; }
    }
    public synchronized void devolveTalher(int pos, int dono) {
      System.out.println("O seguinte filósofo devolve o talher "+ pos);
      (Garfo)garfos.get(pos).setEstadoGarfo(false);
      (Garfo)garfos.get(pos).setDonoGarfo(-1);
    }
    public synchronized void comer(int fil) {
      System.out.println("O seguinte filósofo está comendo "+ fil);
    }
    public synchronized void pensar(int fil) {
      System.out.println("O seguinte filósofo está pensando "+ fil);
    }
}
```

Figura 4.12 – Jantar dos Filósofos: *Periodic Handler* em SCJ

4.2 Network

Nesta Seção será demostrado o processo de tradução do exemplo *Network* que simula o comportamento de uma rede com os métodos *connect*, *send*, *disconnect* e *getState*. As mensagens são trocadas entre dois *Handlers*, um do tipo periódico (*Handler1*) e outro do tipo aperiódico (*Handler2*).

A aplicação foi modelada em *SCJ-Circus* com o uso de seis classes como pode ser visto no diagrama de classes da Figura 4.13. A aplicação é iniciada com um *Safelet* com nome *MainSafelet* que declara um *Mission Sequencer* chamado *MainMissionSequencer*. O *Mission Sequencer* por sua vez inicia uma missão chamada *MainMission* que é a responsável por criar a classe *Network* que desempenha o papel de canal no exemplo e os dois *Handlers* que atuam como agentes das trocas de mensagens.

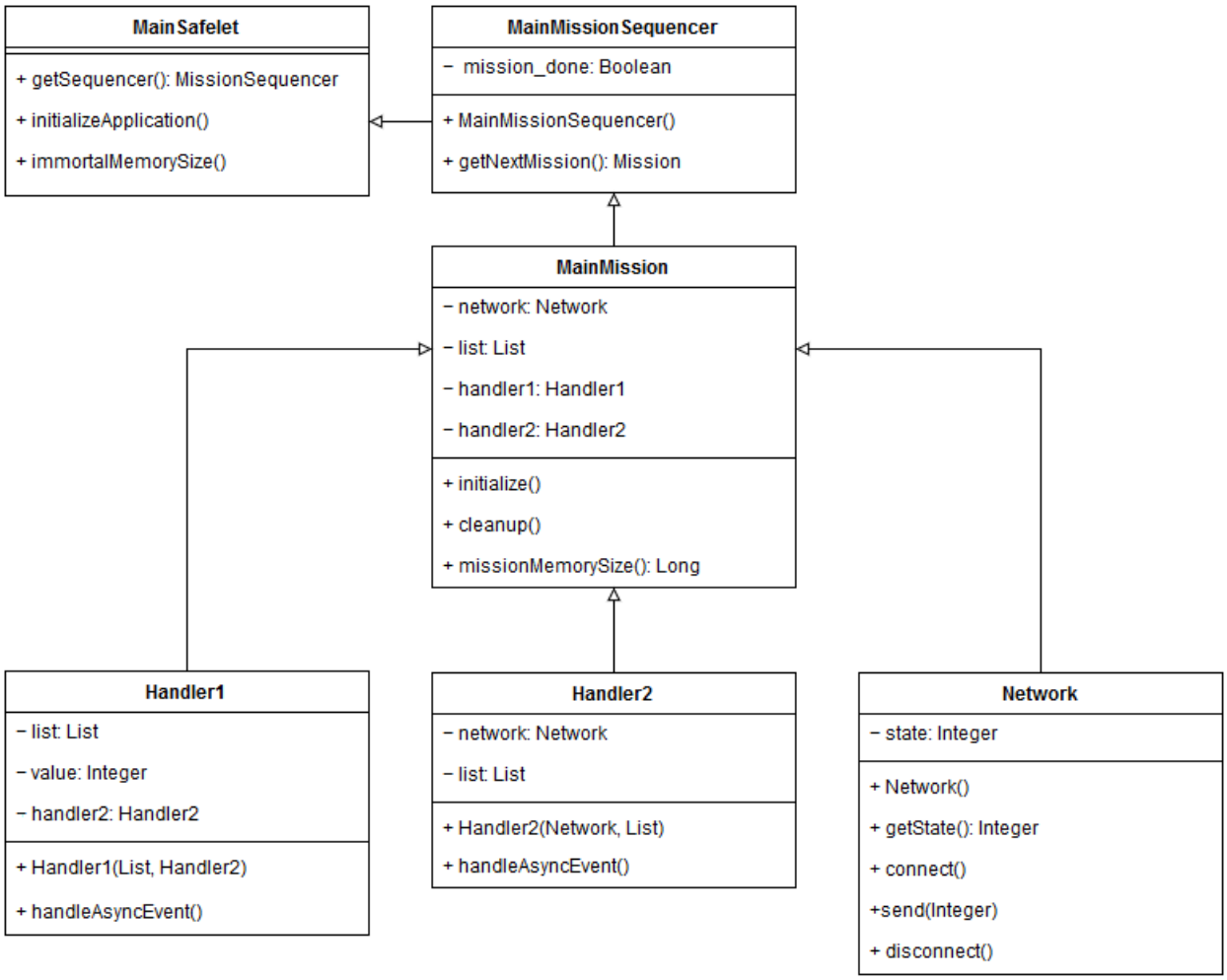

Figura 4.13 – Diagrama de Classes do Network.

4.2.1 Safelet

Conforme pode ser visto na Figura 4.14 o *Safelet* da *Network* contém a função *getSequencer* que inicializa um novo *Mission Sequencer*. São declaradas ainda as funções *initializeApplication* e *immortalMemorySize* apenas com um comando *Skip*.

```
safelet MainSafelet ≙ begin
   getSequencer ≙ ret := MainMissionSequencer
   initializeApplication ≙ Skip
   immortalMemorySize ≙ Skip
end
```

Figura 4.14 – *Network: Safelet* em Circus

O *safelet MainSafelet* em Circus é traduzido para a classe *MainSafelet* que implementa *Safelet* na Figura 4.15. A função *getSequencer* é traduzida para o método *getSequencer* que retorna um novo *sequencer* do tipo *MainSequencer*.

As funções *initializeApplication* e *immotalMemorySize* são ambas compostas por um comando *Skip* e são traduzidas diretamente para os métodos *initializeApplication* e *immotalMemorySize* vazios em SCJ.

```
public class MainSafelet implements Safelet {
   public MissionSequencer getSequencer() {
     return new MainMissionSequencer();
   }
   public void initializeApplication() { }
   public void immortalMemorySize() { }
}
```

Figura 4.15 – *Network: Safelet* em SCJ

4.2.2 Classe

Na Figura 4.16 é apresentada a classe *Network* que contém as funções *state* onde um atributo é declarado e *initial* onde o valor do atributo é definido. Estão presentes ainda o método *getState* que retorna o valor do atributo e os métodos *connect*, *send* e *disconnect* que realizam impressões e atualizam o valor do atributo. O atributo *state* do tipo inteiro é utilizado para controlar o estado da rede (0 quando desconectado, 1 quando conectado e 2 quando enviado).

```
class Network ≙ begin
  state NetworkState == [ state : int ]
  initial NetworkInit ≙ [ NetworkState'; state := 0 ]
  public synchronized int getState ≙ [
    ret := state; ]
  public synchronized void connect ≙ [
    print("Network connect");
    state := 1; ]
  public synchronized void send ≙ [
    value? : int |
    print("Network sending value "+ value);
    state := 2; ]
  public synchronized void disconnect ≙ [
    print("Network disconnect");
    state := 0; ]
end
```

Figura 4.16 – *Network: Classe* em Circus

A classe *Network* é traduzida para a classe *Network* na Figura 4.17. O estado *NetworkState* em Circus composto pela declaração de um atributo é traduzido para a declaração do atributo *state* do tipo *int*. A função *initial* por sua vez é traduzida para o construtor padrão da classe com a inicialização do atributo *state* para zero.

O método *getState* é traduzido para o método de mesmo nome em SCJ apenas com o comando de retorno. Os métodos *connect*, *send* e *disconnect* são traduzidos para os métodos de mesmo nome com o comando *print* que é traduzido para *System.out.println* em SCJ e a atribuição tem sua tradução conforme a sintaxe apropriada.

```
public class Network {
    private int state;
    public Network() {
      state = 0;
    }
    public synchronized int getState() {
      return state;
    }
    public synchronized void connect() {
      System.out.println("Network connect");
      state = 1;
    }
    public synchronized void send(int value) {
      System.out.println("Network sending value "+ value);
      state = 2;
    }
    public synchronized void disconnect() {
      System.out.println("Network disconnect");
      state = 0;
    }
}
```

Figura 4.17 – *Network: Classe* em SCJ

4.2.3 Mission Sequencer

Na Figura 4.18 é apresentado o *Mission Sequencer* da *Network* que contém a função *getNextMission* que inicializa uma nova missão caso ainda não esteja definida ou retorna nulo. Além disso estão presentes as funções *state* onde um atributo é declarado e *initial* onde o valor do atributo é definido.

```
sequencer MainMissionSequencer ≙ begin
   state MainMissionSequencerState == [ mission_done : Bool ]
   initial ≙ [ MainMissionSequencerState'; mission_done := false ]
   getNextMission ≙
     if mission_done = false →
       mission_done := true;
       ret := MainMission;
     □
       return null;
     fi
end
```

Figura 4.18 – *Network: Mission Sequencer* em Circus

O *MissionSequencer MainMissionSequencer* é traduzido para a classe *MainMissionSe-*

quencer que estende *MissionSequencer* na Figura 4.19. A função *getNextMission* é traduzida para o método *getNextMission* que retorna uma nova missão do tipo *Mission* caso ela ainda não tenha sido criada ou retorna o valor *null*.

O estado *MainMissionSequencerState* em Circus composto pela declaração de um atributo é traduzido para a declaração do atributo *mission_done* do tipo *boolean*. A função *initial* por sua vez é traduzida para o construtor padrão da classe com a inicialização do atributo *mission_done* para falso.

```
public class MainMissionSequencer extends MissionSequencer {
   public boolean mission_done;
   public MainMissionSequencer() {
     super(
       new PriorityParameters(
       PriorityScheduler.instance().getMaxPriority()),
       new StorageParameters(10000, 10000, 10000));
     mission_done = false;
   }
   public Mission getNextMission() {
     if (mission_done == false) {
       mission_done = true;
       return new MainMission();
     } else {
       return null;
     }
   }
}
```

Figura 4.19 – *Network: Mission Sequencer* em SCJ

4.2.4 Mission

A missão do *Network* é apresentada na Figura 4.20. Ela é composta por um *state* onde ocorre a declaração de quatro atributos e a função *cleanup* apenas com um comando *Skip* em seu corpo. Além disso é declarada a função *missionMemorySize* que retorna um valor inteiro e a função *initialize* que declara dois objetos e dois *Handlers* um do tipo periódico e outro do tipo aperiódico.

```
mission MainMission ≙ begin
  state missionState ==
    [ network:Network; list:List, handler1:Handler1, handler2:Handler2 ]

  initialize ≙
    network := new Network;
    list := new List;
    handler2 := new Handler Handler2(list, network);
    handler1 := new Handler Handler1(list, handler2, 0, 100);

  cleanup ≙ Skip

  missionMemorySize ≙ ret := 131072
end
```

Figura 4.20 – *Network: Mission* em Circus

A estrutura *mission* com nome *MainMission* em Circus é traduzida para a classe *MainMission* que estende *Mission* na Figura 4.21. O estado *missionState* composto pela declaração de atributos é traduzido para uma lista de atributos da classe.

A função *initialize* com duas criações de objetos e duas declarações de *handlers* são traduzidos respectivamente para o método *initialize* com os objetos e a declaração e registro dos *handlers*.

A função *cleanup* composta pelo comando *Skip* é traduzida para o método *cleanup* vazio. Por fim a função *missionMemorysSize* com um comando de retorno de um valor inteiro é traduzida para o método *missionMemorySize* com o retorno correspondente.

```
public class MainMission extends Mission {
   private Network network;
   private List list;
   private Handler1 handler1;
   private Handler2 handler2;

   public void initialize() {
      network = new Network();
      list = new List();
      handler2 = (new Handler2(list, network)).register();
      handler1 = (new Handler1(list, handler2, 0, 100)).register();
   }

   public void cleanup() { }

   public long missionMemorySize() { return 131072; }
}
```

Figura 4.21 – *Network: Mission* em SCJ

4.2.5 Handlers

Na Figura 4.22 é apresentado o *periodic handler* que conta com a declaração de três atributos na função *state*. Na função *initial* os atributos *list* e *handler2* são inicializados com os valores recebidos como parâmetros do construtor. Está definida ainda a função *handleAsyncEvent* que realiza os comandos *print*, atribuição e chamada de métodos.

```
periodic(offset_ms,period_ms) handler Handler1 ≙ begin
   state Handler1State ==
      [ list : List; handler2 : Handler2; value : Int ]
   initial Handler1Init ≙
      [Handler1State';
      list? : List; handler2? : Handler2; |
      this.list := list?; this.handler2 := handler2?; value := 0; ]
   handleAsyncEvent ≙
      print("Handler1 input received");
      value := value + 1;
      list.insert(value);
      handler2.fire();
end
```

Figura 4.22 – *Network: Handler1* em Circus

Na Figura 4.23 o *handler* periódico possui os valores de inicialização (*offser_ms* e *period_ms*) e o nome *Handler1* e é traduzido para a classe *Handler1* que estende um *Perio-*

dicEventHandler. Os atributos declarados no *state* são traduzidos como atributos da classe em SCJ.

Os comandos presentes no *initial* de Circus e seus parâmetros são traduzidos para o construtor da classe em SCJ. O construtor ainda recebe os parâmetros de inicialização (*offser_ms* e *period_ms*) como parâmetros. A função *handleAsyncEvent* possui seus comandos traduzidos de forma direta para *println*, atribuição e as chamadas dos métodos *insert* e *fire*.

```
public class Handler1 extends PeriodicEventHandler {
    private List list;
    private int value;
    private Handler2 handler2;

    public Handler1(List list, Handler2 handler2) {
        this.list = list;
        this.handler2 = handler2;
        value = 0;
    }

    public void handleAsyncEvent() {
        System.out.println("Handler1 input received");
        value = value + 1;
        list.insert(value);
        handler2.fire();
    }
}
```

Figura 4.23 – *Network: Handler1* em SCJ

Na Figura 4.24 é apresentado o *Handler2* do tipo aperiódico que contém a função *state* onde são declarados dois atributos, a função *initial* onde os atributos são inicializados e onde são definidos os parâmetros do construtor. Ainda está presente a função *handleAsyncEvent* onde são executados comandos *wait* e são feitas três chamadas de métodos.

```
aperiodic handler Handler2 ≙ begin
   state Handler2State == [ network : Network; list : List ]
   initial Handler2Init ≙
      [Handler2State';
      network? : Network; list? : List; |
      this.network = network?;
      this.list = list?]
   handleAsyncEvent ≙
      wait network.getState() <> 0
      network.connect();
      wait network.getState() <> 1
      network.send(list.get(list.size()-1));
      wait network.getState() <> 2
      network.disconnect();
end
```

Figura 4.24 – *Network: Handler2* em Circus

O *aperiodic handler* com nome *Handler2* é traduzido, na Figura 4.25, para a classe *Handler2* que estende um *AperiodicEventHandler*. Os atributos declarados dentro do *state* são traduzidos para os atributos privados da classe.

A função *initial* que recebe dois parâmetros (*network* e *list*) e duas inicializações de atributos (*network* e *list*) é traduzida para o construtor da classe que também recebe o valor de *network* e *list* como parâmetros e realiza as inicializações em seu corpo. O *handleAsyncEvent* é traduzido para o método *handleAsyncEvent* e os comandos presentes em seu corpo são traduzidos diretamente para as chamadas de método e comandos *wait*.

```
public class Handler2 extends AperiodicEventHandler {
   private Network network;
   private List list;
   public Handler2(Network network, List list) {
      this.network = network;
      this.list = list;
   }
   public void handleAsyncEvent() {
      while (network.getState() != 0) { }
      network.connect();
      while (network.getState() != 1) { }
      network.send(list.get(list.size()-1));
      while (network.getState() != 2) { }
      network.disconnect();
   }
}
```

Figura 4.25 – *Network: Handler2* em SCJ

4.3 Persistent Signal

Nesta Seção será demostrado o processo de tradução do exemplo *Persistent Signal* apresentado em (HRISTAKIEV, 2013). Neste exemplo uma *thread* de controle deseja delegar algumas tarefas para um *Logger* que realiza uma saída assíncrona como pode ser visto na Figura 4.26.

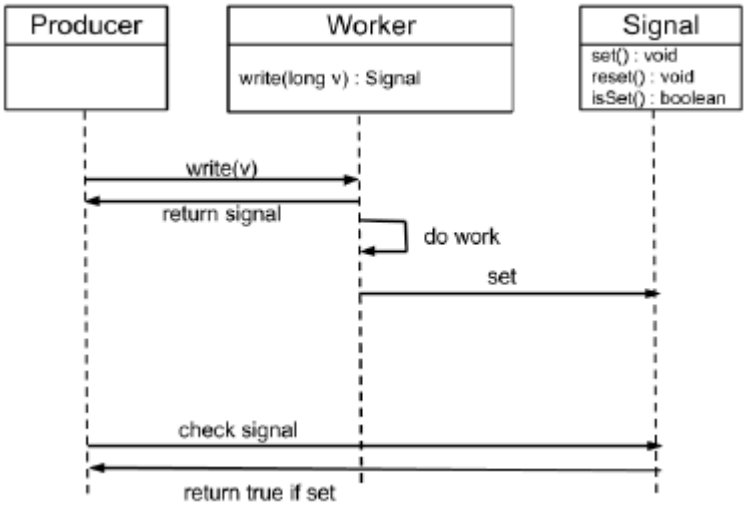

Figura 4.26 – Delegação assíncrona. Fonte: (HRISTAKIEV, 2013).

A *thread* que realiza a chamada não é bloqueada ao chamar o *Worker* mas precisa posteriormente verificar se o trabalho foi feito. Após a sinalização, o *Worker* enfileira o trabalho e retorna uma referência para o produtor. A *Thread* original pode então prosseguir e posteriormente verificar se o trabalho foi executado. Quando o *Worker* completar sua execução, ele define o *Signal* associado.

Os componentes da aplicação são apresentados no diagrama de classe da Figura 4.27. A aplicão consiste de uma missão que inicializa dois *handlers*, um do tipo periódico (*Producer*) e outro do tipo aperiódico (*Worker*) e um sinal do tipo *AperiodicEvent*.

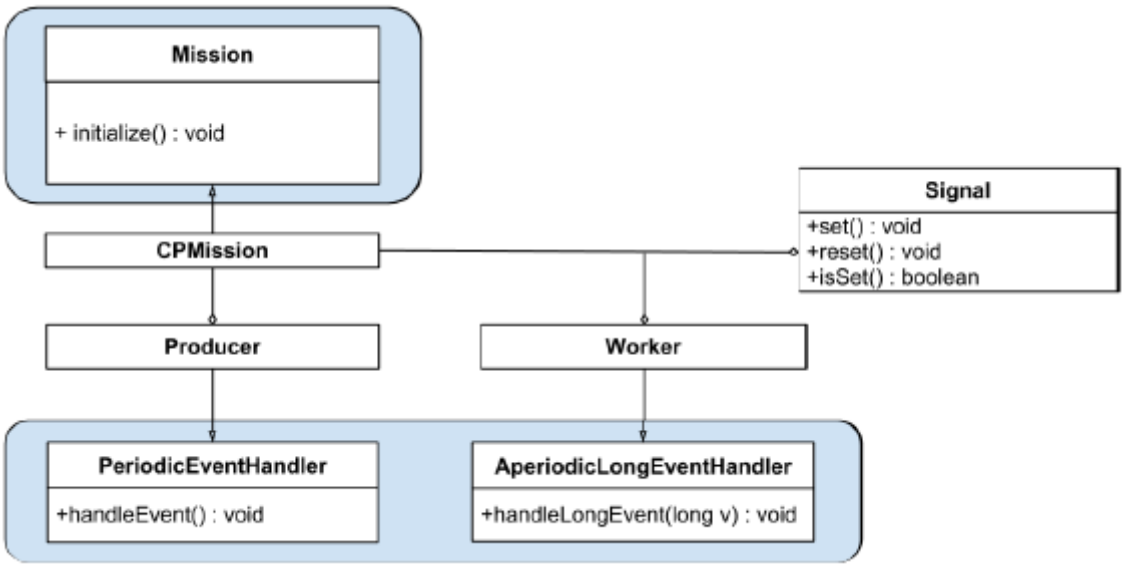

Figura 4.27 – Diagrama de Classe do Persistent Signal. Fonte: (HRISTAKIEV, 2013).

4.3.1 Safelet

Conforme pode ser visto na Figura 4.28 o *Safelet* do *Persistent Signal* contém a função *getSequencer* que inicializa um novo *Mission Sequencer*. São declaradas ainda as funções *initializeApplication* e *immortalMemorySize* apenas com um comando *Skip* e um método adicional *getLevel* que retorna um novo objeto do tipo *Level*.

```
safelet MySafelet ≙ begin
   getSequencer ≙ ret := MainSequencer
   initializeApplication ≙ Skip
   immortalMemorySize ≙ Skip
   public Level getLevel ≙ [ ret := new Level(1) ]
end
```

Figura 4.28 – *Persistent Signal: Safelet* em Circus

O *safelet MySafelet* em Circus é traduzido para a classe *MySafelet* que implementa *Safelet* na Figura 4.29. A função *getSequencer* é traduzida para o método *getSequencer* que retorna um novo *sequencer* do tipo *MainSequencer*.

As funções *initializeApplication* e *immotalMemorySize* são ambas compostas por um comando *Skip* e são traduzidas diretamente para os métodos *initializeApplication* e *immotalMemorySize* vazios em SCJ.

É ainda definido o método *getLevel* que é traduzido diretamente para o método *getLevel* com retorno do tipo *Level* em ambos os casos que cria um objeto com parâmetro de construtor

1.

```
public class MySafelet implements Safelet {
   public MissionSequencer getSequencer() {
     return new MainSequencer();
   }
   public void initializeApplication() { }
   public void immortalMemorySize() { }
   public Level getLevel() {
     return new Level(1);
   }
}
```

Figura 4.29 – *Persistent Signal: Safelet* em SCJ

4.3.2 Mission Sequencer

Na Figura 4.30 é apresentado o *Mission Sequencer* do *Persistent Signal* que contém a função *getNextMission* que inicializa uma nova missão. Além disso estão declaradas as funções *state* e *initial*, ambas compostas por um comando *Skip*.

```
sequencer MainSequencer ≙ begin
   state MainSequencerState == [ Skip ]
   initial ≙ Skip
   getNextMission ≙ ret := MainMission
end
```

Figura 4.30 – *Persistent Signal: Sequencer* em Circus

O *MissionSequencer MainSequencer* é traduzido para a classe *MainSequencer* que estende *MissionSequencer* na Figura 4.31. A função *getNextMission* é traduzida para o método *getNextMission* que retorna uma nova missão do tipo *Mission*.

O estado *MainSequencerState* em Circus composto apenas por um comando *Skip* é traduzido para uma lista vazia de atributos da classe. A função *initial* por sua vez também composta pelo comando *Skip* é traduzida para o construtor padrão da classe sem a adição de nenhum comando adicional.

```
public class MainSequencer extends MissionSequencer {
   public MainSequencer() {
      super(new PriorityParameters(PriorityScheduler.instance().getMaxPriority()),
         new StorageConfigurationParameters(131072, 4096, 4096)
      );
   }
   private Mission getNextMission() {
      return new MainMission();
   }
}
```

Figura 4.31 – *Persistent Signal: Sequencer* em SCJ

4.3.3 Mission

A missão do *Persistent Signal* é apresentada na Figura 4.32. Ela é composta por um *state* e a função *cleanup*, ambos apenas com um comando *Skip* em seu corpo. Além disso é declarada a função *missionMemorySize* que retorna um valor inteiro e a função *initialize* que realiza uma impressão, declara um sinal do tipo *aperiodicEvent* e dois *Handlers* um do tipo periódico e outro do tipo aperiódico.

```
mission MainMission ≙ begin
   state missionState == [ Skip ]
   initialize ≙
      print("Initializing main mission");
      signal := new PersistentSignal;
      new Handler Worker(signal);
      new Handler Producer(signal, 0, 2000);
   cleanup ≙ Skip
   missionMemorySize ≙ ret := 1000000
end
```

Figura 4.32 – *Persistent Signal: Mission* em Circus

A estrutura *mission* com nome *MainMission* em Circus é traduzida para a classe *MainMission* que estende *Mission* na Figura 4.33. O estado *missionState* composto pelo comando *Skip* é traduzido para uma lista de atributos vazia.

A função *initialize* com os comandos *print*, declaração de *persistentSignal* e duas declarações de *handlers* são traduzidos respectivamente para o método *initialize* com os comandos *println*, declaração de *PersistentSignal* e a declaração e registro dos *handlers*.

A função *cleanup* composta pelo comando *Skip* é traduzida para o método *cleanup* vazio. Por fim a função *missionMemorysSize* com um comando de retorno de um valor inteiro é

traduzida para o método *missionMemorySize* com o retorno correspondente.

```
public class MainMission extends Mission {
   private void initialize() {
      System.out.println("Initializing main mission");
      PersistentSignal signal = new PersistentSignal();
      (new Worker(signal)).register();
      (new Producer(signal, 0, 2000)).register();
   }
   public void cleanup() {}
   public long missionMemorySize() { return 1000000; }
}
```

Figura 4.33 – *Persistent Signal: Mission* em SCJ

4.3.4 Handler

Na Figura 4.34 é apresentado o *aperiodic handler* que contém a função *state* onde são declarados dois atributos, a função *initial* onde os dois atributos são inicializados e onde é definido o parâmetro do construtor. Ainda estão presentes as funções *handleAsyncEvent* onde um contador é incrementado, uma impressão é realizada e o sinal é definido e o método *getSignal* que retorna o sinal.

```
aperiodic handler Worker ≙ begin
   state WorkerState == [ _signal : PersistentSignal; _iteration : int ]
   initial WorkerInit ≙ [WorkerState_State'; event? : PersistentSignal |
                              _signal := event?; _iteration := 0 ] ;
   handleAsyncEvent ≙
      _iteration := _iteration+1;
      print("Executing worker");
      _signal.set();
   public PersistentSignal getSignal ≙ [ ret := _signal'; ]
end
```

Figura 4.34 – *Persistent Signal: Aperiodic Handler* em Circus

O *aperiodic handler* com nome *Worker* é traduzido, na Figura 4.35, para a classe *Worker* que estende um *AperiodicEventHandler*. Os dois atributos declarados dentro do *state* são traduzidos para dois atributos privados da classe. A função *initial* que recebe um parâmetro (*event*) e duas inicializações de variáveis (*signal* e *iteration*) é traduzida para o construtor da classe que também recebe o valor de *event* como parâmetro e realiza as inicializações em seu corpo.

A função *handleAsyncEvent* é traduzida para o método *handleAsyncEvent* e os comandos presentes em seu corpo são traduzidos diretamente para a atribuição, o comando *println* e o comando *set* do sina. Ainda existe a declaração da função *getSignal* com retorno do tipo *PersistentSignal* que é traduzido diretamente para o método *getSignal* em SCJ com o retorno do sinal.

```
public class Worker extends AperiodicEventHandler {
   private PersistentSignal _signal;
   private int _iteration;
   public Worker(PersistentSignal event){
      super(
         new PriorityParameters(PriorityScheduler.instance().getMaxPriority()),
         new AperiodicParameters(),
         new StorageConfigurationParameters(32768, 4096, 4096),
         event,
         "Logger"
      );
      this._signal = event;
      this._iteration = 0;
   }
   public void handleAsyncEvent() {
      this._iteration=this._iteration+1;
      System.out.println("Executing worker");
      this._signal.set();
   }
   public PersistentSignal getSignal() { return this._signal; }
}
```

Figura 4.35 – *Persistent Signal: Aperiodic Handler* em SCJ

Na Figura 4.34 é apresentado o *periodic handler* que conta com a declaração de um atributo na função *state*. Na função *initial* o atributo é inicializado como valor recebido como parâmetro do construtor. Está definida ainda a função *handleAsyncEvent* que realiza os comandos *reset* e *release* no sinal e os comandos *wait* e *print* além do comando *if*.

```
periodic(offset_ms,period_ms) handler Producer ≙ begin
  state ProducerState == [ _event : PersistentSignal ]
  initial ProducerInit ≙
    [ProducerState';
    log? : PersistentSignal; |
    _event := log?]
  handleAsyncEvent ≙
    _event.reset();
    _event.release();
    print("+ Producer - starting computation");
    wait(100);
    print("+ Producer - finishing computation");
    if _event'.isSet() →
      print("+ Producer - output done")
    □
      print("+ Producer - output not done yet")
    fi
end
```

Figura 4.36 – *Persistent Signal: Periodic Handler* em Circus

Na Figura 4.37 o *handler* periódico possui os valores de inicialização (*offser_ms* e *period_ms*) e o nome *Producer* e é traduzido para a classe *Producer* que estende um *PeriodicEventHandler*. O atributo declarado no *state* é traduzido como um atributo da classe em SCJ.

Os comandos presentes no *initial* de Circus e seu parâmetro são traduzidos para o construtor da classe em SCJ. O construtor ainda recebe os parâmetros de inicialização (*offser_ms* e *period_ms*) como parâmetros. A função *handleAsyncEvent* possui seus comandos traduzidos de forma direta para *reset* e *release*, *println*, *nanoSpin* e *if-else* no método *handleAsyncEvent*.

```
public class Producer extends PeriodicEventHandler {
   private PersistentSignal _event;
   public Producer(PersistentSignal log, long period_ms, long offset_ms){
      super(
         new PriorityParameters(PriorityScheduler.instance().getNormPriority()),
         new PeriodicParameters(new RelativeTime(offset_ms, 0),
         new RelativeTime(period_ms, 0)),
         new StorageConfigurationParameters(32768, 4096, 4096),
         65523,
         "Producer"
      );
      this._event = log;
   }
   public void handleAsyncEvent() {
      this._event.reset();
      this._event.release();
      System.out.println("+ Producer - starting computation ");
      nanoSpin(100);
      System.out.println("+ Producer - finishing computation ");
      if (this._event.isSet()){
         System.out.println("+ Producer - output done");
      }else{
         System.out.println("+ Producer - output not done yet");
      }
   }
}
```

Figura 4.37 – *Persistent Signal: Periodic Handler* em SCJ

4.3.5 Aperiodic Event

Na Figura 4.38 é apresentado o *Aperiodic Event* formado por um *state* onde ocorre a declaração de um atributo. Na função *initial* o atributo é inicializado e a função *setMaxCeiling* é executada. São definidos ainda os métodos adicionais *reset*, *set* e *isSet* com comandos de atribuição ou retorno dentro de si.

```
aperiodicEvent PersistentSignal ≙ begin
  state signalState == [ set : Bool ]
  initial signalInit ≙ [
    setMaxCeiling();
    set := false;
  ]
  public synchronized Bool reset ≙ [ set := false; ]
  public synchronized Bool set ≙ [ set := true; ]
  public synchronized Bool isSet ≙ [ ret := set; ]
end
```

Figura 4.38 – *Persistent Signal: Aperiodic Event* em Circus

O *Aperiodic Event* com nome *PersistentSignal* é traduzido para a classe *PersistentSignal* que estende *AperiodicEvent* na Figura 4.39. O atributo set declarado em *state* é traduzido como um atributo da classe *PersistentSignal* em SCJ. As inicializações feitas em *initial* são transportadas para o construtor da classe e os três métodos (*reset*, *set* e *isSet*) e seus comandos são traduzidos para métodos da classe conforme sintaxe apresentada.

```
public class PersistentSignal extends AperiodicEvent {
  private boolean set;
  public PersistentSignal(){
    super();
    Services.setCeiling(this, PriorityScheduler.instance().getMaxPriority());
    this.set = false;
  }
  public synchronized void reset() { this.set = false; }
  public synchronized void set() { this.set = true; }
  public synchronized boolean isSet() { return this.set; }
}
```

Figura 4.39 – *Persistent Signal: Aperiodic Event* em SCJ

4.4 Conclusão

Neste Capítulo foram apresentados os exemplos do Jantar dos Filósofos, *Network* e *Persistent Signal* ambos modelados na linguagem *SCJ-Circus*. Com os exemplos e levando em conta a estratégia proposta no Capítulo 3, foi possível realizar a tradução de todos os elementos que compunham os três exemplos, desta forma foi possível verificar a viabilidade do processo de tradução proposto.

5 CONCLUSÃO

O presente trabalho teve como objetivo desenvolver uma estratégia para a tradução de aplicações modeladas em uma linguagem formal (*Circus*) para código executável em uma linguagem de programação para sistema críticos (*Safety Critical Java*). Para atingir esse objetivo o trabalho primeiramente apresentou um estudo sobre a estrutura e sintaxe das linguagens *Circus* e SCJ. Posteriormente foi apresentada a linguagem *SCJ-Circus* que estende a sintaxe padrão da linguagem *Circus* com os elementos básicos da linguagem SCJ.

Foram apresentados ainda durante a revisão bibliográfica alguns trabalhos que também buscaram traduzir programas modelados em linguagens formais para código executável. Ambos os trabalhos possuem semelhanças com o presente trabalho ao usarem linguagens intermediárias que aproximam a sintaxe das linguagens e possibilitam o seu processo de tradução. Nenhum dos trabalhos porém, apresenta o foco na tradução de aplicações para sistemas críticos usando como base as linguagens *Circus* e SCJ.

No capítulo seguinte foi apresentado o esquema de tradução proposto para traduzir as aplicações modeladas em *Circus* para código em SCJ. A estratégia apresentada foi através da criação da EBNF para cada uma das linguagens e um detalhamento da tradução entre cada um dos seus elementos. Foram apresentados os elementos *Safelet*, *Mission Sequencer*, *Mission*, *Handlers* e *Aperiodic Event* oriundos da linguagem *SCJ-Circus*, a declaração de classes da linguagem *OhCircus* além de identificadores, declarações e comandos.

Posteriormente foram apresentados três estudos de caso com os exemplos do Jantar dos Filósofos, *Network* e *Persistent Signal* ambos modelados na linguagem *SCJ-Circus*. Com os exemplos e levando em conta a estratégia proposta no trabalho, foi possível realizar a tradução de todos os elementos que compunham os três exemplos, desta forma foi possível verificar a viabilidade do processo de tradução proposto.

A principal contribuição do presente trabalho foi a definição da gramática de ambas as linguagens e o detalhamento completo da transformação de cada elemento em uma tradução "um para um". Com o uso das EBNFs e o detalhamento do processo é possível, com certa facilidade, implementar um processo de tradução automatizando para o código entre as duas linguagens em um trabalho futuro.

Ao concluir este estudo, sugere-se ainda como trabalhos futuros a adição novos elementos às gramáticas criadas neste trabalho com a inclusão de gerenciamento de memória ou

prioridades da linguagem SCJ ou ainda a geração de código para aplicações de nível 0 ou 2 que ficaram de fora do escopo do presente trabalho. Como atualmente a execução do código em SCJ de nível 1 ainda não é possível pela falta de uma máquina virtual compatível para isso, propõem-se ainda a utilização do esquema proposto em (SILVA, 2015) que realiza a tradução de código SCJ em C++ para realizar o teste e execução do código em uma plataforma embarcada.

REFERÊNCIAS

BOLLELLA, G. **The real-time specification for Java**. [S.l.: s.n.], 2000.

CAVALCANTI, A. et al. Safety-critical Java programs from Circus models. **Real-Time Systems**, [S.l.], v.49, n.5, p.614–667, 2013.

CAVALCANTI, A.; SAMPAIO, A.; WOODCOCK, J. Unifying classes and processes. **International Journal on Software Systems Modelling**, [S.l.], v.4, n.3, p.277–296, 2005.

DOXSEE, S.; GARDNER, W. Synthesis of C++ software from verifiable CSPm specifications. In: ENGINEERING OF COMPUTER-BASED SYSTEMS, 2005. ECBS '05. 12TH IEEE INTERNATIONAL CONFERENCE AND WORKSHOPS ON THE. **Anais...** [S.l.: s.n.], 2005. p.193–201.

FREITAS, A. **From *Circus* to Java: implementation and verification of a translation strategy**. 2005. Dissertação (Mestrado em Ciência da Computação) — Department of Computer Science, The University of York.

FREITAS, A. F.; CAVALCANTI, A. L. C. Automatic Translation from Circus to Java. In: FM 2006: Formal Methods. **Anais...** Springer-Verlag, 2006. p.115 – 130. (Lecture Notes in Computer Science, v.4085).

HOARE, C. A. R. Communicating Sequential Processes. **Commun. ACM**, New York, NY, USA, v.21, n.8, p.666–677, Aug. 1978.

HRISTAKIEV, I. **Programming in Safety Critical Java**. 2013. Dissertação (Mestrado em Ciência da Computação) — The University of York.

LOCKE, D. et al. **Safety-Critical Java Technology Specification, Public draft**. 2010.

SILVA, R. F. **Tradução da especificação SCJ para Linguagem de Programação C++**. 2015. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de Santa Maria.

SOMMERVILLE, I. et al. **Engenharia de software**. [S.l.]: ADDISON WESLEY BRA, 2008.

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

WOLFF, B.; GAUDEL, M.-C.; FELIACHI, A. (Ed.). **Unifying Theories of Programming**: 4th international symposium, utp 2012, paris, france, august 27-28, 2012, revised selected papers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p.68–87.

WOODCOCK, J. C. P.; CAVALCANTI, A. L. C. The steam boiler in a unified theory of Z and CSP. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC 2001), 8. **Anais...** IEEE Press, 2001.

WOODCOCK, J.; DAVIES, J. **Using Z**: specification, refinement, and proof. [S.l.]: Prentice Hall, 1996. (Prentice-Hall international series in computer science).

ZHU, H. et al. Generating C Programs from CSP Models. In: SOFTWARE TESTING, VERIFICATION AND VALIDATION WORKSHOPS (ICSTW), 2013 IEEE SIXTH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2013. p.21–26.