UNIVERSIDADE FEDERAL DO AMAZONAS
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
DEPARTAMENTO DE APOIO A PESQUISA
PROGRAMA INSTITUCIONAL DE INICIAÇÃO CIENTÍFICA

ESPECIFICAÇÃO EXECUTÁVEL USANDO UMA LINGUAGEM DE REDES DE PETRI COLORIDA NO DOMÍNIO DE SISTEMAS EMBARCADOS

Bolsista: Romário Lira Batista, FAPEAM

ITACOATIARA

2012

UNIVERSIDADE FEDERAL DO AMAZONAS
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
DEPARTAMENTO DE APOIO A PESQUISA
PROGRAMA INSTITUCIONAL DE INICIAÇÃO CIENTÍFICA

RELATÓRIO PARCIAL

PIB–E–0149/2011

ESPECIFICAÇÃO EXECUTÁVEL USANDO UMA LINGUAGEM DE REDES DE PETRI COLORIDA NO DOMÍNIO DE SISTEMAS EMBARCADOS

Bolsista: Romário Lira Batista, FAPEAM

Orientador: Profº. MSc. Christophe Saint-Christie de Lima Xavier

ITACOATIARA

2012

Todos os direitos deste relatório são reservados à Universidade Federal do Amazonas, ao Núcleo de Estudo e Pesquisa em Ciência da Informação e aos seus autores. Parte deste relatório só poderá ser reproduzida para fins acadêmicos ou científicos.

Esta pesquisa, foi financiada pela Fundação de Amparo à Pesquisa do Estado do Amazonas – FAPEAM, através do Programa Institucional de Bolsa de Iniciação Científica da Universidade Federal do Amazonas, foi desenvolvida pelo Núcleo de Estudo e Pesquisa em Ciências da Informação e assemelha-se à Dissertação de Mestrado Especificação Executável Usando uma Linguagem de Redes de Petri no Domínio de Sistemas Embarcados.

SUMÁRIO

1. INTRODUÇÃO..06

2. REVISÃO BIBLIOGRÁFICA..07

3. MÉTODOS UTILIZADOS..11

4. RESULTADOS E DISCUSSÕES..12

5. CONCLUSÕES...29

6. REFERÊNCIAS BIBLIOGRÁFICAS...30

Resumo

Este trabalho descreve uma metodologia para uma geração automática de código para sistemas embarcados, a partir de uma Rede de Petri Colorida. A abordagem proposta utiliza Especificação Executável baseada em Redes de Petri Colorida, na qual podem ser verificadas algumas de suas propriedades como *deadlock*, vivacidade, limitação, alcançabilidade, entre outras. E estas Redes contêm transições que podem ter códigos escritos na linguagem de programação C, as quais executarão partes específicas que serão modeladas. Esta especificação serve de base para exibir ao usuário as funcionalidades do sistema que será modelado, proporcionando ao usuário uma visão das características específicas do sistema. Desta forma, contribuindo com desenvolvedores e engenheiros na geração de protótipos que constituem uma especificação executável, facilitando a avaliação de diferentes modelos e ajudando a reduzir as diferenças de interpretação na construção de software. Este trabalho apresenta ainda uma ferramenta com base nesta metodologia e um estudo de caso baseado em um protótipo de um robô para demonstrar a utilização e aplicação da metodologia proposta.

Palavras-chaves: Redes de Petri Colorida, Prototipação, Geração de Códigos.

Abstract

This paper describes a methodology for automatically generating code for embedded systems, from a Coloured Petri Net. The proposed approach uses executable specification based on Colored Petri Nets, which can be verified in some of its properties like deadlock, vivacity, limitation, reachability, among others. And these networks contain transitions that can have code written in the C programming language, which perform specific parts to be modeled. This specification provides the basis to show the user the functionality of the system to be modeled, providing the user with an overview of the specific features of the system. Thus, contributing developers and engineers in the generation of prototypes that constitute an executable specification, facilitating the evaluation of different models and helping to reduce the differences in interpretation in building software. This work also presents a tool based on this methodology and a case study based on a prototype robot to demonstrate the use and application of the proposed methodology.

Keywords: Colored Petri Nets, Prototyping, Code Generation.

1. Introdução

Dia após dia, nossas vidas se tornam mais dependentes de sistemas embarcados. Isso não inclui apenas sistemas críticos de segurança, tais como automóveis, ferrovias, aviões, naves espaciais, dispositivos médicos, robôs e esteiras industriais, mas também de automação residencial, consoles de videogames, eletrodomésticos, *drives* de discos, impressoras de rede, caixas eletrônicos, telefones celulares, e assim por diante. Devido a esta diversidade de aplicações, os projetos de sistemas embarcados podem ser sujeitos a diferentes restrições, como tempo de projeto, tamanho, peso, consumo de energia, confiabilidade e custo. [da Silva Barreto, 2005]

Nestes diferentes ambientes, aplicações de softwares necessitam ser desenvolvidas rapidamente e atender a um alto nível de qualidade. Os métodos formais desempenham um importante papel para mensurar a previsibilidade e dependência no projeto de aplicações criticas. Como exemplo, pode-se citar as Redes de Petri para modelagem de tais sistemas.

O uso de métodos formais no desenvolvimento de softwares apresenta varias vantagens, por exemplo, programas (ou protótipos) podem ser gerados automaticamente e formalmente a partir de suas especificações. Pode-se provar também que determinados programas satisfazem determinadas propriedades e que o programa é uma realização da sua especificação.

A especificação executável permite evitar inconsistências, erros e garantir a completude do sistema. Esse processo assegura uma interpretação não ambígua para a especificação do sistema, valida a funcionalidade dele antes do inicio da sua implementação, criando modelos de desempenho e avaliando seu próprio desempenho.

Este trabalho propõe gerar uma especificação executável baseada em um modelo formal conhecido como Redes de Petri focado para Coloridas, no qual será possível checar propriedades especificas e características de uma Rede de Petri, tais como, se a rede é segura, limitada, ordinária, entre outras. Nesta, há transições e lugares que possuem códigos anotados na linguagem C, com base nestes itens, é efetuada a geração do código.

Também é apresentada a ferramenta denominada PNTCG (*Petri Net Tool for Code Generation*) a ser otimizada utilizando Redes de Petri Colorida com base nesta metodologia e um estudo de caso baseado em um protótipo de automatização de embalagens de produtos, no qual é utilizado uma esteira e um braço robô para demonstrar a utilização e aplicação da metodologia proposta.

2. Revisão Bibliográfica

O principal objetivo ao gerar-se uma especificação executável utilizando as Redes de Petri Coloridas para modelar *softwares* de Sistemas Embarcados é reduzir o tamanho da Rede, comparado a uma Rede de Petri Elementar, e evitar inconsistências e erros ao executar este protótipo, vale ressaltar, também, que os Sistemas Embarcados estão cada vez mais presentes em nosso cotidiano, já fazendo parte de nossas vidas.

Quando abordamos um protótipo de *software* objetivamos reduzir os erros provenientes da análise de requisitos através de uma verificação antecipada da especificação do software que está em desenvolvimento. Assim, quando temos um protótipo executável é possível testar os estados do sistema, por execução, bem antes do inicio da fase de implementação.

Segundo [Budde et al., 1992], protótipos adequados fornecem aos usuários e gestores uma ideia tangível das soluções dos problemas. Para os desenvolvedores, os protótipos que constituem uma especificação executável que facilita a avaliação de diferentes modelos e ajuda a reduzir as diferenças de interpretação na construção de *softwares* [Alcoforado, 2007].

Redes de Petri foram desenvolvidas por Carl Adam Petri em 1962, em sua tese de doutorado, intitulada “Comunicação com autômatos”, e foi defendida na Universidade de Darmstadt, na Alemanha. A teoria geral foi desenvolvida a fim de modelar e analisar sistemas de comunicação. Porém, com o passar dos anos, sua tese foi sendo estendida em várias direções, abrangendo assim mais áreas de aplicações, tais como modelagem de protocolos de comunicação, controle de fábricas, concepção de software de tempo real, sistemas de informação e gerenciamento de base de dados [Cardoso e Valette 1997].

Uma Rede de Petri pode ser representada por um grafo (ou, também, matematicamente por meio de matrizes), sendo composta de: [Loyola, 2008] lugares, indicados por nós elípticos ou circulares, e transições entre lugares, indicadas por nós retangulares ou em forma de barra. Arcos orientados ligam lugares às transições ou vice-versa, nunca entre elementos do mesmo conjunto.

Atualmente existem várias ramificações das Redes de Petri dentro dos Métodos Formais, como por exemplo, Redes de Petri Estocásticas, Temporais, Orientadas a Objetos, Coloridas, entre outras. As redes de Petri Coloridas [Jensen 1997] permitem a criação de tipos de dados complexos para as marcas, o que nos permite simplificar alguns modelos de rede de Petri Lugar/Transição.

Com o interesse de reduzir os erros inseridos na fase da análise dos requisitos funcionais de software, evidencia-se, que devemos ir além de protótipos e utilizar métodos formais, que são

estruturados por linguagens e ferramentas para especificar e verificar sistemas, baseados em fundamentos lógicos matemáticos [Clarck & Wing, 1996]. A utilização de métodos formais busca principalmente aumentar a boa compreensão do software, revelando ambiguidades, inconsistências e erros que podem não ser detectados. Além disso, essas especificações facilitam a modularização e o reuso no desenvolvimento do software [Rangel, 2006].

A seguir, serão descritos alguns trabalhos correlatos à proposta deste trabalho.

2.1 Linguagem Pencil

A linguagem Pencil, segundo [Conway, 2002], foi projetada para ser simples, intuitiva, flexível, formal, poderosa e altamente modular. Esta possui uma modularidade e integração Java que permite aos programadores para combinar módulos e verificação formal de forma integrada. A sintaxe simples e mecanismo de tradução automática ajuda a reduzir erros de implementação. Pencil herda o modelo matemático de Redes de Petri tal que o comportamento da aplicação pode ser descrito simples e econômico. Esta linguagem especifica uma Rede de Petri apenas num arquivo texto. Com relação ao trabalho aqui proposto, é que nossa abordagem permite especificar uma Rede através de uma interface e que s pode anotar códigos não somente em transições, mas também em lugares.

2.2 CoOperative Objects (COO)

No trabalho de [Sibertin-Blanc, 2001] é demonstrado uma abordagem chamada de CoOperative Objects (COO) que tem por objetivo ser uma ponte entre o formalismo e o projeto detalhado. Usando como compilador de objetos CoOperativos a ferramenta SYROCO estendido. A descrição de Redes de Petri habilita e traduz modelos COO em classes C++. O autor compara, ou melhor, transforma conceitos de Rede de Petri em classes de objetos, usando uma linguagem de programação orientada a objeto, sendo os tokens instâncias dessas classes. Para processar tokens, cada transição é provida com um pedaço de código: quando uma transição ocorre, este código está aplicado com os tokens envolvidos. Neste trabalho, além de nos basearmos na ideia de código anotado (na linguagem C) na transição e lugares numa Rede de Petri, nos baseamos em orientação a objeto na linguagem Java.

2.3 ACG8081 Automatic Code Generation for Intel'8031

No trabalho de [Dezani, 2006], descreve-se um programa de geração automática de código para microcontrolador 8051 da Intel, a partir de uma Rede de Petri, com o objetivo de minimizar o tempo gasto na codificação do programa e automatizar completamente esse processo de transformação. Definiu-se o uso de Rede de Petri Lugar/Transição como modelo de entrada, pois, segundo o autor do trabalho, mesmo tendo um modelo mais compacto, as Redes de Petri Coloridas, quando transformada em códigos Assembly é consideravelmente maior que o código Assembly para

a Rede de Petri Lugar/Transição. Nesta metodologia, não geramos códigos Assembly, todavia, geramos códigos na linguagem C, porém nós utilizamos das Redes de Petri Coloridas para essa geração de código.

2.4 LOOPN++

LOOPN++ utiliza o modelo de Redes de Petri de alto nível orientado a objeto e contem um tradutor, que gera um código na linguagem de programação C++ referente à Rede de Petri. Ao contrario da LOOPN++, que possui um compilador e não possui um simulador, esta proposta se utiliza de um simulador para, obviamente, simular as Redes.

Para a conclusão deste trabalho fez-se necessário a pesquisa referente a como seriam representados os códigos das Redes de Petri Coloridas na ferramenta, encontrou-se então duas formas: a Linguagem SML e o Projeto JPetriSim.

2.5 Linguagem SML

A linguagem SML(Standard ML), é uma linguagem funcional completa muito utilizada no ensino e na pesquisa. Foi criada por pesquisadores da LFCS (*Laboratory for Foundations of Computer Science*) na década de 1980. Em 1987, Robin Milner e LFCS ganharam o prêmio BCS *Award for Technical Excellence* por trabalhar no Standard ML.

SML foi originalmente concebida como uma metalinguagem para o sistema de prova de teorema de Edimburgo LCF, mas evoluiu para uma linguagem de propósito geral de sucesso. Esta linguagem foi padronizada em 1990 e revista em 1997 como Standard ML 97. Sendo conhecida como uma linguagem funcional e impura, por permitir efeitos colaterais e, por esta razão também é conhecida como uma Linguagem de Programação multi-paradigma.

2.6 Projeto JPetriSim

O projeto *JPetriSim* é na verdade uma tese de TCC apresentada por César Augusto Lins de Oliveira, essa tese foi intitulada de Simulação de Redes de Petri em Ambiente Java, e foi apresentada em 04 de julho de 2006, em Recife – Pernambuco, na Universidade de Pernambuco.

O trabalho apresentado nesta monografia consistiu na implementação de um conjunto de simuladores de Redes de Petri na linguagem Java. O objetivo desta implementação é oferecer a outros desenvolvedores a possibilidade de incluírem a capacidade de simular Redes de Petri em suas próprias ferramentas. Para facilitar sua utilização, os simuladores foram agrupados em uma biblioteca, na qual o desenvolvedor poderá encontrar também facilidades para realizar a comunicação entre sua aplicação e os simuladores. A biblioteca recebeu o nome de *JPetriSim (Java Petri Simulatation)*, e foi desenvolvida a partir de uma pesquisa abrangente da teoria de Redes de Petri.

Tais formas serão discutidas mais adiante no tópico Resultados e Discussões.

Dessa forma, a utilização conjunta de um protótipo de software com métodos formais enfatizam uma especificação categórica do problema e expõe o usuário a um sistema utilizável o mais rápido possível, de modo que usuários e engenheiros de softwares sejam capazes de executar e validar as especificações dos requisitos funcionais do sistema. Assim, os benefícios da especificação formal e da abordagem de protótipos para o desenvolvimento de software são combinados e uma descrição de sistema bem estruturada, executável e sem ambiguidade pode ser desenvolvida.

As pesquisas bibliográficas contribuíram para o crescimento do conhecimento à cerca dos conceitos de Redes de Petri Coloridas e qual o diferencial usado na construção dessa ferramenta.

Neste deste trabalho, foi adotado como estudo de caso a simulação dos movimentos dos braços e pernas do robô (**Figura 01**). O robô construído neste trabalho vai efetuar as seguintes tarefas: **(1)** O robô, inicialmente, vai estar em repouso, **(2)** e **(4)** ao caminhar, o robô deverá movimentar a perna e o braço correspondente, caso levante a perna esquerda levantará o braço direito e vice-versa para manter o equilíbrio, sendo que estes movimentos tenham sido modelados em Rede de Petri Colorida, já evitando os erros, como por exemplo, que o robô levante as duas pernas e caia, **(3)** após seu trajeto ele fará uma volta e repetirá os movimentos **(2) e (4)**, terminando com o robô em repouso **(1).** O projeto adquiriu, fomentado pela Fundação de Amparo à Pesquisa do Estado do Amazonas, os seguintes componentes para a construção do robô:

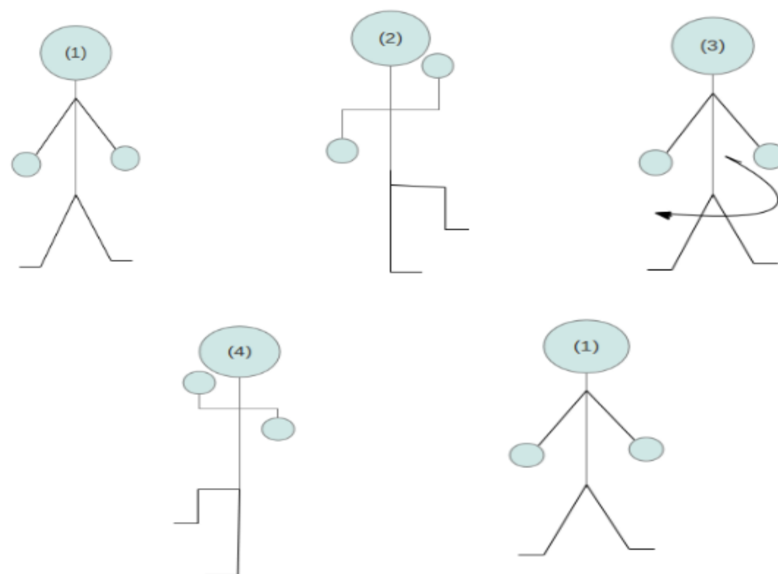

Figura 01. Simulação dos movimentos do robô.

3. Métodos Utilizados

O presente trabalho foi realizado no Instituto de Ciências Exatas e Tecnologia (ICET), do período de Agosto de 2011 à Junho de 2012, financiado pela Fundação de Amparo à Pesquisa do Amazonas (FAPEAM) e está pautado nas seguintes etapas metodológicas, cumpridas:

Pesquisa bibliográfica e estudo dos trabalhos levantados: Foram feitos estudos de trabalhos relacionados a proposta desse projeto para conhecer a fundo os trabalhos já publicados na área e experimentos com as ferramentas já existentes.

Pesquisou-se as propriedades de Rede de Petri. Para checar estas propriedades utilizamos a ferramenta INA (*Integrated Net Analyser*), que contribui na análise de Redes de Petri de diferentes tipos de investigação relacionada a disparos na Rede e análise de propriedades gerais;

Experimento. Também realizou-se experimentos utilizando a ferramenta PNTCG do trabalho Uma Especificação Executável Usando uma Linguagem de Redes de Petri no Domínio de Sistemas Embarcados;

Geração de códigos. Pesquisou-se um método para realizar a coleta e geração de códigos contidos em lugares e transições, decidindo-se usar o algoritmo de Busca em Profundidade.

Estudo de Caso. Estudou-se de que forma será utilizado Redes de Petri Colorida para representar o comportamento de um robô. Buscando-se os componentes que foram necessários para a construção do robô que servirá para realizar o caso de teste ao fim do projeto.

7. Resultados e Discussões

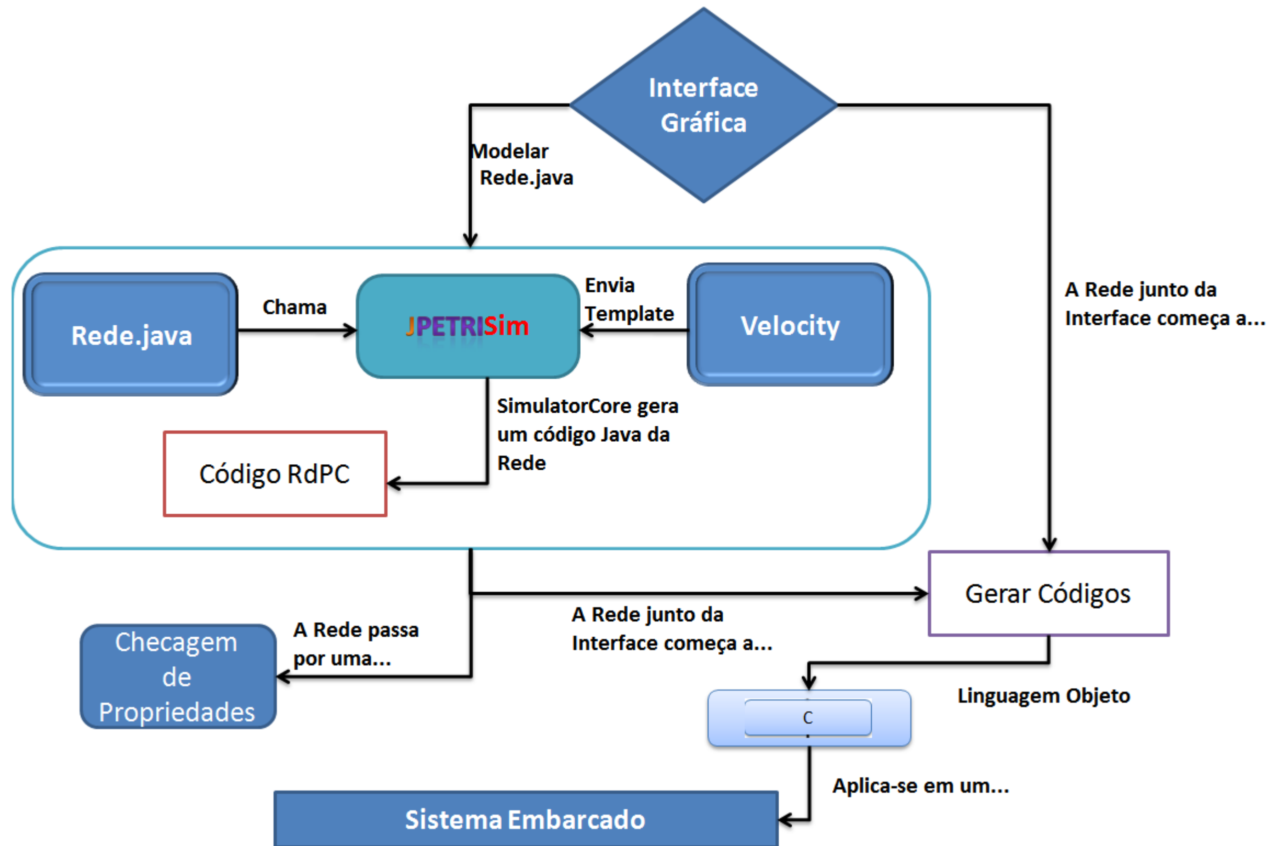

Figura 02. Arquitetura: Aplicação.

Baseado na **Figura 02**, a aplicação desenvolvida neste trabalho teve por objetivo servir para modelagem de Redes de Petri Colorida, onde estas possam ter suas propriedades analisadas e verificadas e a partir das Redes modeladas nela, gerar códigos e aplica-los a um Sistema Embarcado. Funcionando, basicamente, da seguinte maneira: a partir da interface gráfica, o usuário irá modelar uma Rede de Petri Colorida, criando assim um código da Rede na linguagem Java (Rede.java). Este código irá convocar o pacote *JPetriSim*, que, por intermédio do *plug-in Velocity*, irá enviar um *Template* que automaticamente gera um segundo código Java contendo a descrição dos disparos de Lugar-Transição e vice-versa. A partir daí, a Rede passa por uma checagem de suas propriedades, utilizando a ferramenta conhecida como INA (*Integrated Net Analyzer*), esta por sua vez mostrará ao usuário quais propriedades foram checadas na Rede, como por exemplo, se a Rede é Ordinária, se é Viva, Homogênea, se é Pura, se é Conservativa, se possui *Deadlock*, entre outras.

Seguidamente, a Rede, com códigos anotados em lugares e/ou transições, junto da interface, irá promover a geração de códigos, em uma linguagem de destino (linguagem C), códigos estes que serão aplicados em um Sistema Embarcado, no caso, as pernas robóticas.

Foto 1: Construção do robô.

A construção do robô bípede (**Foto 1** e **Foto 2**) foi relativamente exaustiva, visto que, ele foi feito de alumínio e, portanto, tivemos que procurar peças para sua estrutura. Felizmente encontramos as peças e elas nos foram doadas, sem qualquer custo. Dessa forma, a ferramenta elaborada gera códigos na linguagem C, os quais são aceitos pelo software da empresa Arduíno, servindo para programação do sistema embarcado (Robô).

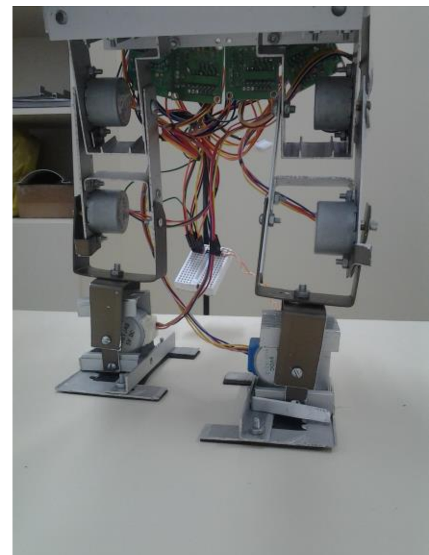

Foto 2: Pernas robóticas.

Foi dado início a construção do compilador com o auxílio do *Flex*, que auxilia na análise léxica, onde seria analisado *token* por *token* do código de origem, e *Bison*, que auxilia na formação do *parser* ou análise sintática. Este seria feito na linguagem de programação C, que serviria para identificar os Conjuntos de Cores das Redes de Petri Colorida, já que estes conjuntos é que designam os disparos de lugares e transições. O projeto do compilador foi abandonado porque o aluno participante do projeto ainda encontrava-se incapacitado para tal atividade. Então foram feitas mais pesquisas, a fim de encontrar outro meio para resolver esse impasse de como representar conjuntos de cores nas Redes de Petri Colorida, assim, encontrou-se o projeto da Biblioteca *JPetriSim,* esta por sua vez, é um conjunto de classes responsáveis por simular o comportamento de três tipos de Rede de Petri, as Redes Lugar-Transição, Temporarizadas, e, principalmente, as Coloridas. Referente ao reconhecimento de Redes de Petri Colorida dentro da biblioteca *JPetriSim*, é feito da seguinte forma: pode tanto ser descrita em um “arquivo.xsd”, sendo este uma extensão da Linguagem XML, que é responsável por realizar interoperatividade entre sistemas, onde dentro destes arquivos estarão detalhadas por *tags* os componentes de uma Rede. Ela ainda reconhece detalhamentos de Redes a partir de códigos em Java, por intermédio de classes responsáveis em identificar os componentes da

Rede, como por exemplo, a classe *CPNPlace.java*, esta por sua vez, identifica um Lugar com “id”, nome do lugar, *Colorset* (Conjunto de cor) e as expressões de inicialização (se tiver), além de outras também de extrema importância, como a *CPNTransition.java* (que reconhece Transições), as *CPNArc.java* (que reconhecem os Arcos que ligam Lugares e Transições), entre outras. A equipe optou por trabalhar apenas com o detalhamento em linguagem Java, por já estar familiarizado com tal linguagem de programação. A forma como essa biblioteca trabalha é bem simples (**Figura 03**):

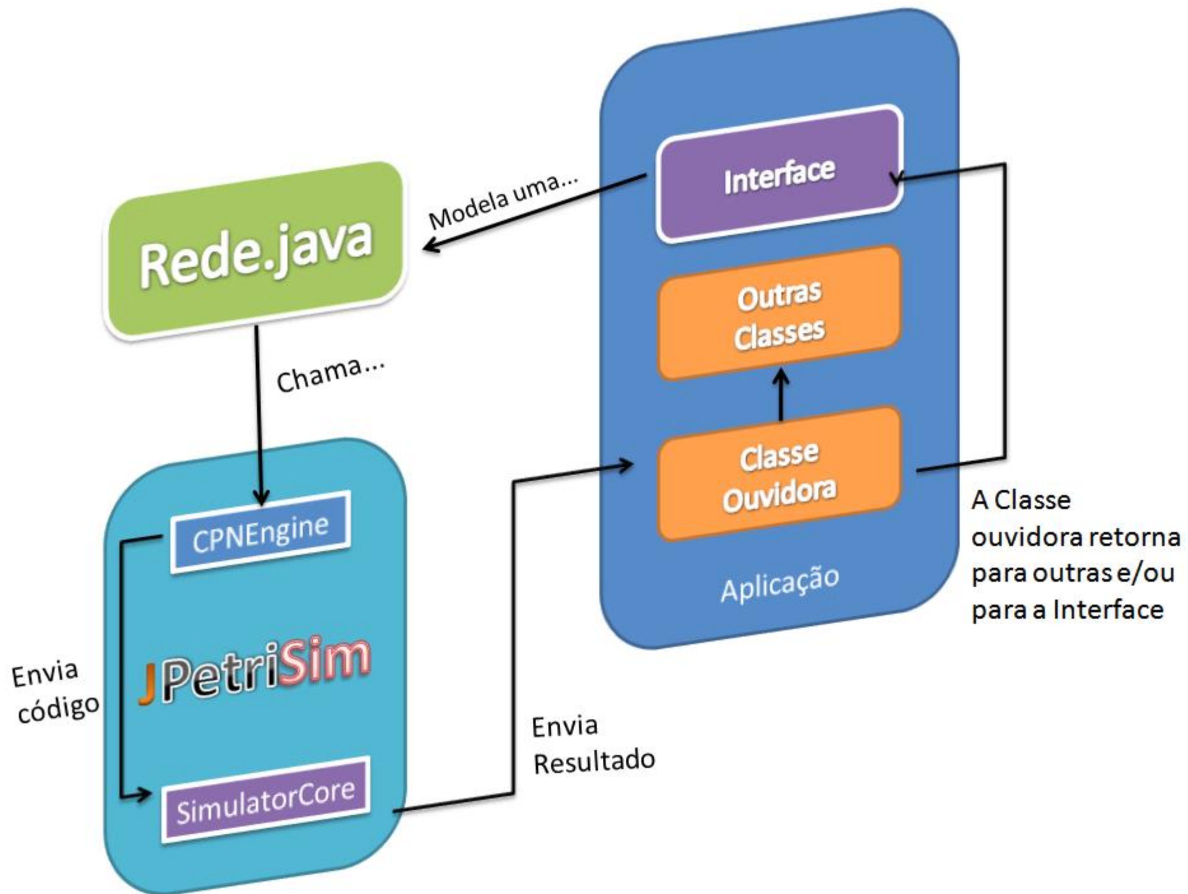

Figura 03. Arquitetura: JPetriSim - Aplicação.

Primeiramente, como de praxe, tudo parte de uma Interface, já que é ela a responsável pela comunicação usuário e aplicação, a partir daí é modelada e detalhada uma Rede de Petri Colorida em um “arquivo.java”, esse “arquivo” tem dentro de sua estrutura uma instância chamando as classes da Biblioteca *JPetriSim*, em principal, a classe *CPNEngine* que é responsável pela engenharia e tratamento de funções específicas da Rede, nessa classe há uma chamada a um *Template* da biblioteca “Velocity.jar”. Após esta etapa, a classe *CPNEngine* junto do *Template* da “Velocity.jar”

geram um código e este passa por uma simulação pelo *SimulatorCore*, também contida no pacote da *JPetriSim*, esta simulação é responsável por verificar os disparos de transições e lugares por intermédio de arcos. O resultado desta simulação é enviado para a classe ouvidora da aplicação, essa classe ouvidora pode ser o próprio código principal, contendo as principais chamadas de funções, ou classes secundarias. Essa classe principal pode ainda trocar dados com classes secundárias e com a própria interface da aplicação.

4.1 Representação Interna das Redes de Petri Colorida na JPetriSim

Para que a simulação pudesse apresentar um desenvolvimento satisfatório, as Redes de Petri Coloridas foram representadas internamente da forma mais otimizada possível. Os seguintes critérios foram seguidos durante o desenvolvimento da *JPetriSim* [OLIVEIRA, 2006]:

- **Não Redundância:** Informações redundantes na representação foram excluídas ao máximo. Desta forma, a representação poderia ser compactada, evitando o desperdiço de memória.
- **Acesso Rápido:** O simulador precisa obter rapidamente as informações que procura. A representação elimina ao máximo a necessidade de buscas ao longo da estrutura.
- **Relevância:** Apenas as informações necessárias para a simulação são mantidas. Assim, informações gráficas foram totalmente descartadas, como as coordenadas de um nó no plano, para isso é atribuído identificadores aos nós.

O modelo foi desenvolvido da seguinte forma [OLIVEIRA, 2006]:

Tabelas Hashing. Todas as informações estão contidas em tabelas, estas tabelas são implementadas como tabelas *Hashing*. As tabelas *hashing* são estruturas de dados que tem alto desempenho na realização de buscas, pois utilizam um algoritmo de codificação que permite que os dados sejam encontrados imediatamente, independe da quantidade de dados armazenados (algoritmo de complexidade constante ou *O(1)*).

Tabelas de Entrada e Saída. Poder-se-ia imaginar que as informações relevantes de uma Rede de Petri Coloridas que precisam ser mantidas pelo simulador são seus lugares, transições e arcos. Na verdade, estas informações apresentam excessiva redundância sob o ponto de vista da simulação. Ocorre que a única informação que descreve uma Rede de Petri, não levando em consideração a sua marcação, é o seu conjunto de arcos. A partir de seu conjunto de arcos, toda a estrutura da Rede pode ser recuperada. A única ressalva é que este conjunto precisa estar separado em dois conjuntos menores: um contendo apenas os arcos que tem como origem

lugares e outro contendo os que têm como origem transições.

Esta estrutura é mais indicada (**Figura 4**), pois, em geral, os simuladores precisam obter todos os arcos para uma determinada transição e, depois, obter o peso de acordo com cada lugar de entrada ou de saída.

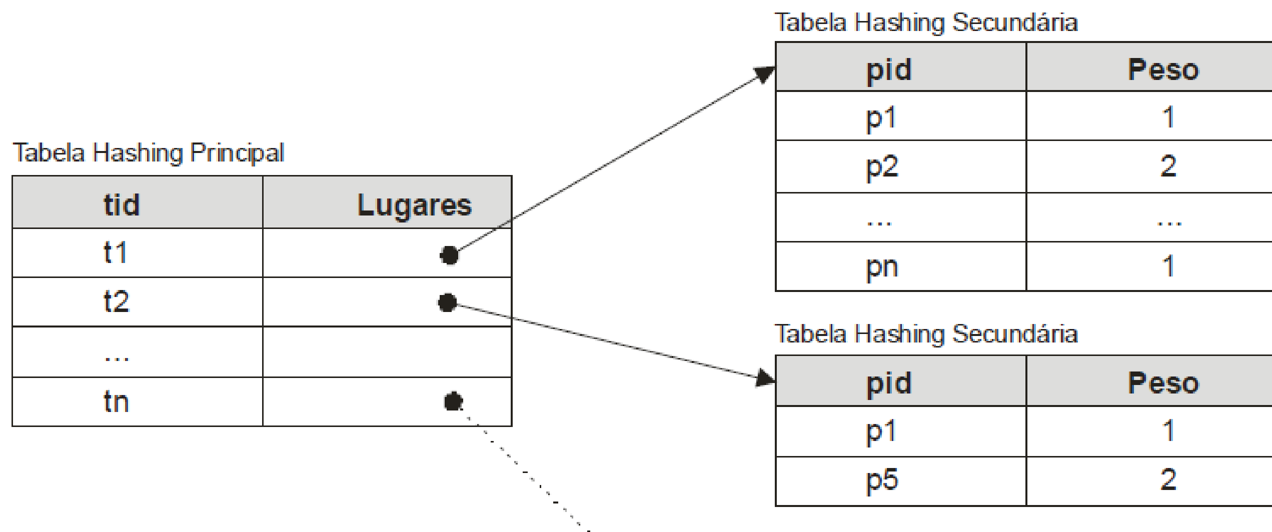

Figura 04. Hierarquia em que foram estruturadas as tabelas [OLIVEIRA, 2006].

4.2 Principais Funções da JPetriSim

Neste tópico, serão descritas as principais funções que podem ser utilizadas pela JPetriSim. Elas serão demonstradas a partir de tópicos contendo a classe Java a que pertence seguida das funções.

4.2.1 CPNEngine()

- *generateSimulation()* – responsável por gerar o código de simulação da Rede de Petri a partir do detalhamento dos nós(lugares e transições) e dos arcos da Rede previamente estabelecidos pelo usuário.
- *Velocity.init()* – esta é responsável por inicializar a biblioteca *Velocity*, que, por sua vez, é a responsável pelo reconhecimento do *Template* usado pela função *generateSimulation()* para gerar o código de simulação da Rede.
- *getTemplate(“ ”)* – esta serve para identificar o local onde se encontra o *Template*, para então ser utilizado na função *generateSimulation()*.
- *ParseErrorException* – é outro tratamento a erro, esta é convocada a mostrar ao usuário uma mensagem de erro, caso seja detectado que há erro(s) de sintaxe em alguma declaração feita no código.

4.2.2 SimulatorCore()

- *Simulate(CPNSimulationListener)* – responsável por organizar a lista de Lugares e Transições e por verificar que marcações há em um lugar e que lugar é este.
- *evalTransitionAfter(String pid)* – responsável por realizar o disparo de uma transição para um lugar posterior à ela, sendo que este deve estar ligado a mesma por intermédio da declaração do arco no código da Rede.
- *getMark(String pid)* - esta função seleciona a marcação que o usuário deseja dispara de um nó da Rede para outro.
- *addTokens(String pid, Multiset tokens)* – verifica se há alguma marcação no lugar e/ou coloca uma nova marcação no lugar.
- *removeTokens(String pid, Multiset tokens)* – ao contrário da função *addTokens(String pid, Multiset tokens)*, esta função fica a cargo de remover marcações de lugares.

4.2.3 CPNNode()

- *setId(String id), getId( )* – estas funções servem para editar um Identificador (Id) e para carregá-lo, respectivamente. Os identificadores declarados nesta função podem ser usados nas classes *CPNPlace( )* e *CPNTransition( )* **(Figura 05)**.
- *setName(String name), getName( )* – estas funções servem para editar um Nome a um nó da Rede e para carregá-lo, respectivamente. Esta também pode ser usada nas mesmas classes que a função anterior.

4.2.4 CPNPlace()

- *setInitExpression(String initExpression), getInitExpression( )* – estas funções se encarregam de, respectivamente, editar uma expressão inicial para um lugar, caso seja necessário (sendo portanto de uso optativo do usuário), e de carregar as edições feitas para um expressão inicial **(Figura 05)**.
- *setColorSet(String colorset), getColorSet( )* – a primeira, fica a cargo de permitir ao usuário que defina a cor ou conjunto de cor que o lugar irá trabalhar. Já a segunda encarrega-se de carregar esta descrição da cor/conjunto de cor.

4.2.5 CPNTransition()

- *setGuardExp(String guard), getGuardExpr( )* – estas funções são responsáveis por editar e armazenar, respectivamente, expressões Guarda, que estão presentes em modelos de Rede de Petri Colorida, guardando um valor booleano (*true* ou *false*) em uma determinada transição **(Figura 05)**.
- *setPreBinding(String preBinding), getPreBinding*( ) – estas funções são responsáveis por

permitir ao usuário que edite e armazene, respectivamente, um pré-lugar de uma transição.

4.2.6 CPNArc()

- *setExpression(String expr), getExpression( )* – estas funções são encarregadas de editar e armazenar, respectivamente, as encriptações dos arcos, ou seja, condições de passagens para os mesmos **(Figura 05)**.
- *setId(String id), getId( )* – edita e armazena, respectivamente, um identificador para o arco.
- *setSource(CPNNode source), getSource( )* – permite ao usuário editar e armazenar, respectivamente, um nó de origem, ou seja, a entrada de um arco.
- *setTarget(CPNNode target), getTarget( )* – permite ao usuário editar e armazenar, respectivamente, um nó de destino, ou seja, a saída de um arco.
- *netError(Enumeration<CPNPlace> enumeration, Enumeration<CPNNode> enumeration2)* – nas Redes de Petri Coloridas, assim como nos outros tipos de Redes de Petri, não é permitida a passagem de mesmos tipos de nós, como por exemplo: um arco passar marcações de um lugar para outro sem antes passar por uma transição; um arco passar marcações de uma transição para outra sem antes passar por um lugar. Esta função trata este tipo de erro comparando a origem e o alvo do arco, caso sejam do mesmo tipo, a função retorna uma mensagem de erro ao usuário dizendo que não é possível tal ação.

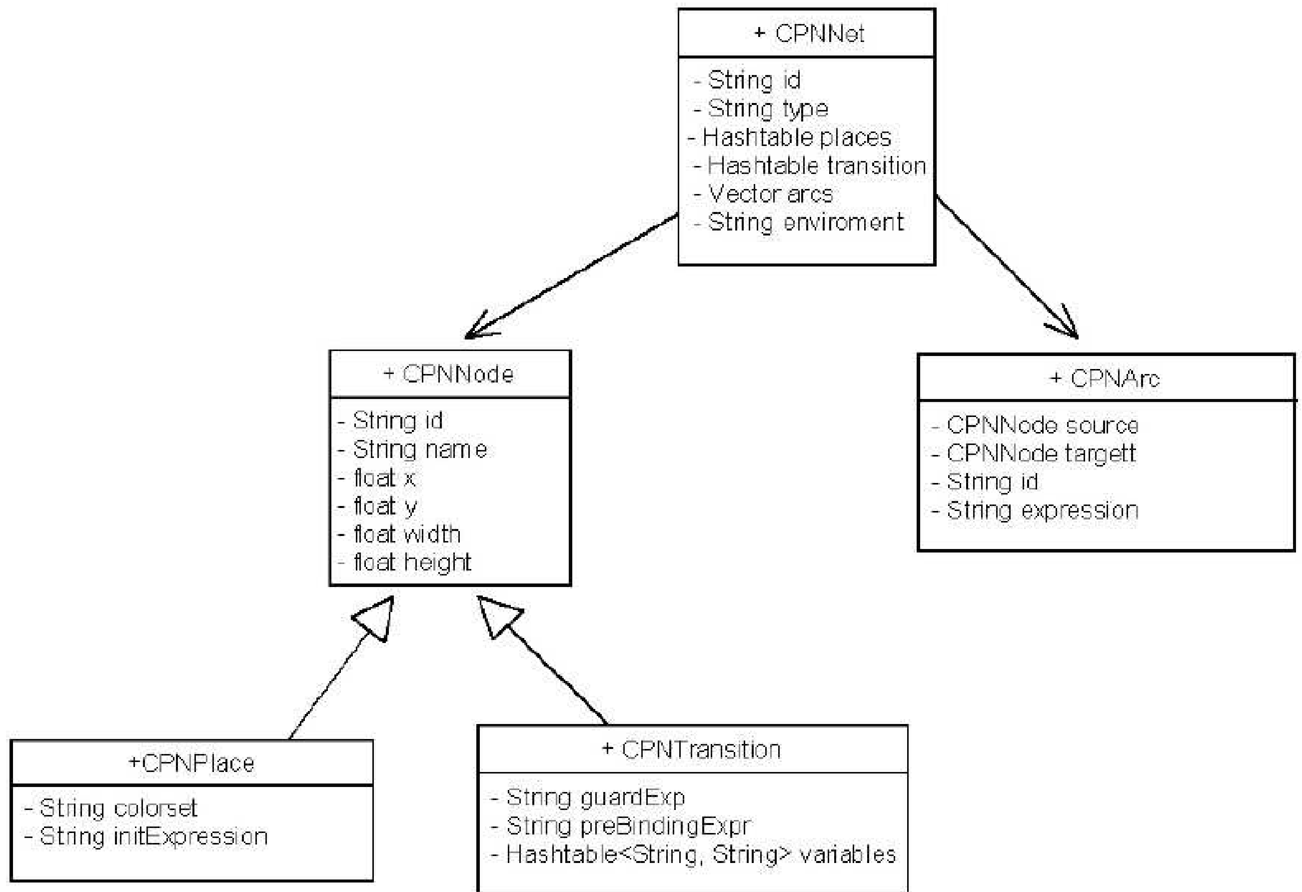

Figura 05. Diagrama UML das Classes do modelo de Rede de Petri Colorida [OLIVEIRA, 2006].

4.3 Geração de Código

Para a geração de código, foi criado dois novos métodos e mais três classes importantes para o pacote/biblioteca *JPetriSim* da seguinte forma:

Foi criado um método que permite ao usuário que, assim que declarar um lugar ou uma transição, ambas estendidas da classe *CPNNode()*, seja possível declarar uma instância chamada de *setCode(String code*), em que, por intermédio do parâmetro passado em forma de *String*, o usuário declara o código anotado no lugar ou transição. A partir daí, outro método é chamado indiretamente, o método *getCode()* que armazena esse código anotado em um parâmetro do tipo *String.*

Os dados digitados pelo usuário ao detalhar um nó da Rede (Lugar ou Transição) são então transformados em um grafo, representado por uma lista de adjacência, esta lista é implementada pelas classes *Grafo_Lista_Encadeada()* e *GraphPetriNet()*, estas sendo novas classes criada para a *JPetriSim.* Segundo [ZIVIANI, 2007], a representação de um grafo por lista de adjacência consiste de

um arranjo de adjacentes de cada vértice do grafo, ou seja, cada vértice terá uma lista adjacente para si, enquanto ele tiver vértices posteriores a ele. Uma vez que, a Rede de Petri tenha sido transformada em grafo, representada por uma lista de adjacência no caso, será aplicado um método para percorrer o grafo criado a partir da Rede. O método usado foi o Busca em Profundida (do inglês *deph-first search (DFS)*), este método é implementado pela classe *DFS()*, a estratégia adotada pelo Busca em Profundidade segundo [CORMEN, 2002] é, como o próprio nome já diz, "procurar mais fundo" no grafo sempre que possível. As arestas do grafo são exploradas a partir de um vértice *v* mais recente descoberto que ainda tem arestas inexploradas saindo dele. Quando todas as arestas de *v* são exploradas, a busca "regressa" para explorar as arestas que deixam o vértice a partir do qual *v* foi descoberto, ou seja, voltando ao vértice de origem. Esse processo continua até descobrirmos todos os vértices acessíveis a partir do vértice de origem inicial.

A recuperação dos códigos dos nós da Rede é feita no momento em que é realizada a busca em profundidade, enquanto um nó da Rede está sendo visitado o método da classe *CPNNode()*, *getCode()*, é chamado e recupera o código anotado na descrição do nó, escrevendo-o, posteriormente, em "*arquivo.c*".

Tomemos como exemplo a (**Listagem 01**), nela é detalhado um lugar, observe que para esse lugar é atribuído um ID ("p1"), um nome para o lugar ("p1"), o conjunto de cor a que o lugar pertence *("CPNInteger",* classe pertencente ao pacote *JPetriSim*), a expressão inicial do lugar (sendo opcional) e o código anotado no lugar ("#include<stdio.h>"). Este código será recolhido no momento em que for feita a busca em profundida e escrito no "*arquivo.c".*

```
//criação dos nós do grafo
        CPNPlace p1 = new CPNPlace();
        p1.setId("p1");
        p1.setName("p1");
        p1.setColorset("CPNInteger");
        p1.setInitExpr("$ = new CPNInteger(new Integer(3));");
        p1.setCode("#include<stdio.h>");
```

Listagem 01. Exemplo da escrita de um código em um nó da Rede.

As figuras a seguir demonstram como os códigos são alocados junto a lugares e transições:

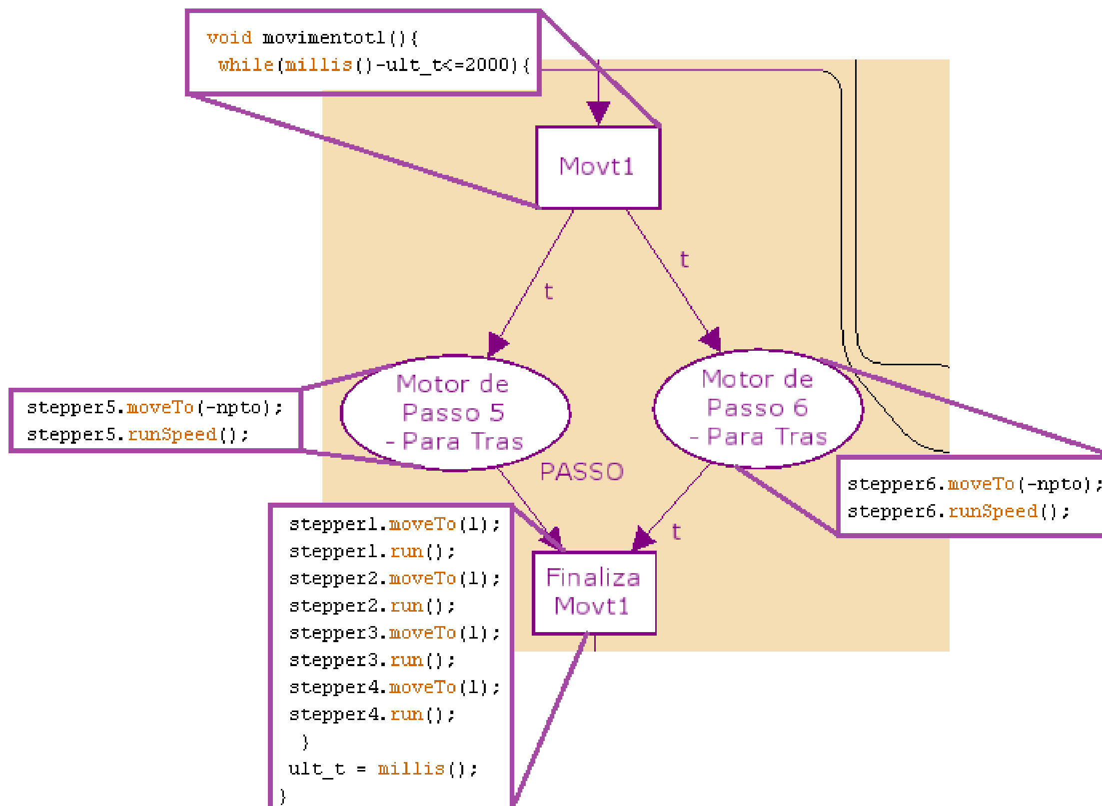

Figura 6. Representação da alocação de códigos na Rede (1 de 4).

Ao iniciar a movimentação do robô, durante um tempo inferior a 2000 milésimos de segundos, é feito o movimento dos tornozelos, iniciando por *Movt1* (**Figura 6, 1- 4**), passando pelos lugares *Motor de Passo 5* e *Motor de Passo 6*, contendo os códigos, em *C*, para a ação dos dois motores e finalizando na transição *Finaliza Movt1*, este contendo ações para manter os outros motores de passo ligados enquanto os dos tornozelos ainda estão em ação.

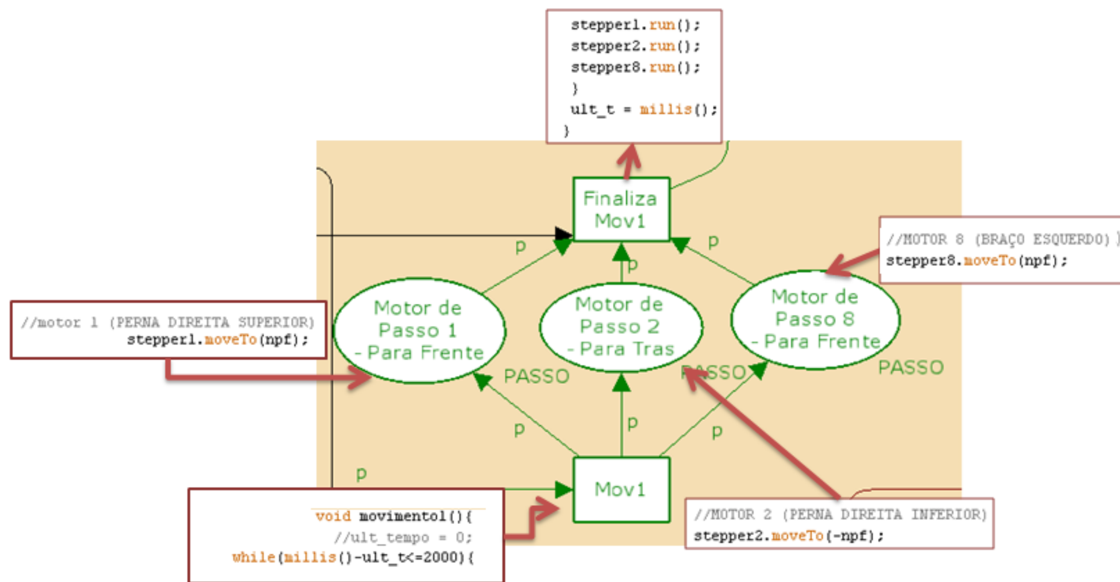

Figure 6. Representação da alocação de códigos na Rede (2 de 4).

A partir desse momento, focaremos no movimento da perna direita do robô. Após o movimento dos tornozelos serem finalizado inicia-se o movimento das pernas, representado na figura pela transição *Mov1* (**Figura 6, 2 – 4**), já dando início ao código da ação, passando, logo após pelos lugares *Motor de Passo 1* (representando o motor responsável pelo movimento da coxa, movendo-se para frente), pelo *Motor de Passo2* (representando o motor responsável pelo movimento do joelho, movendo-se para traz) e pelo *Motor de Passo 8* (representando o motor responsável pelo movimento do braço esquerdo, movendo-se para frente). Finalizando na transição *Finaliza Mov1* executando a ação dos três motores.

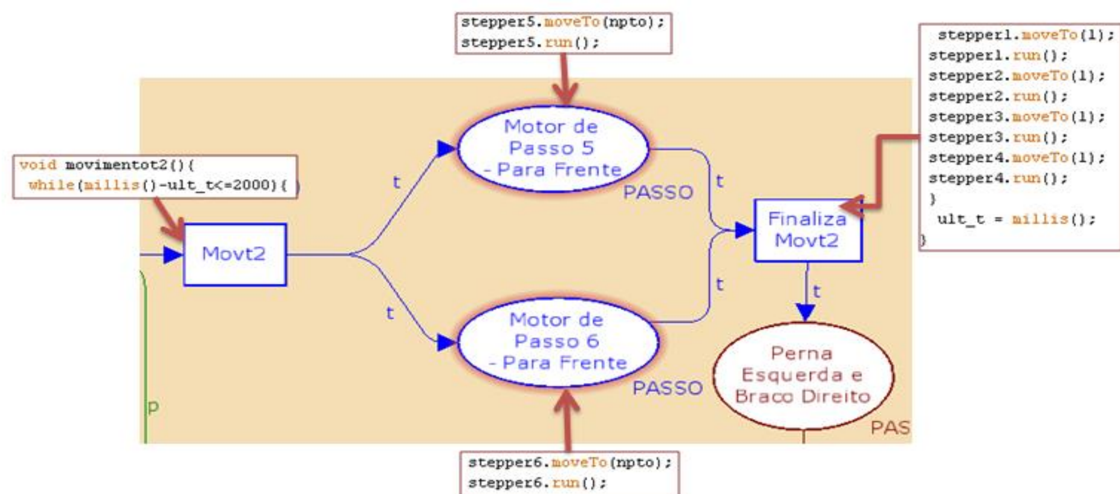

Figura 6. Representação da alocação de códigos na Rede (3 de 4).

Após o movimento da perna direita, o robô volta seus tornozelos à posição original. O movimento inicia na *transição Movt2* (**Figura 6, 3 – 4**), dando inicio também ao código da ação, passando pelo lugar *Motor de Passo 5* (representando um dos tornozelos, com o código voltando-o a posição original, uma vez que, este fora posto para traz) e pelo lugar *Motor de Passo 6* (também com o código voltando-o a posição original). Este movimento é finalizado na transição *Finaliza Movt2* da mesma maneira que o movimento responsável por pôr os tornozelos para traz, ativando os outros motores.

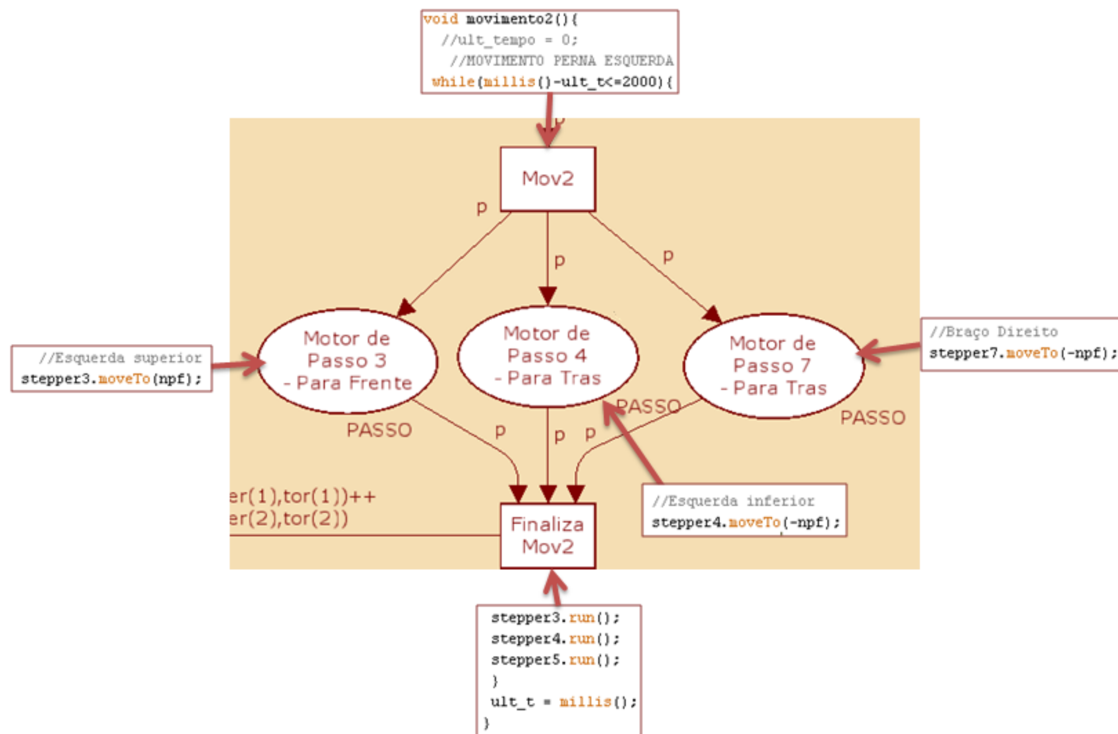

Figura 6. Representação da alocação de códigos na Rede (4 de 4).

Após o movimento de retorno dos tornozelos inicia-se o movimento da perna esquerda. Este começa na transição *Mov2* (**Figura 6, 4 – 4**), já dando início ao código da ação, partindo em seguida para os lugares *Motor de Passo 3* (representando o movimento da coxa esquerda, movendo-se para frente), *Motor de Passo 4* (representando o movimento do joelho esquerdo, movendo-se para traz) e *Motor de Passo 7* (representando o movimento do braço direito, movendo-se para traz). A ação encerra na transição *Finaliza Mov2*, executando as ações dos três pré-lugares.

4.4 Resultados Experimentais

Como resultado experimental foi detalhado em Java uma Rede para a movimentação de um

Robô bípede (Sistema Embarcado) construído no decorrer do projeto. A seguir é mostrado na (**Listagem 02**) um trecho do código do detalhamento da Rede para o robô.

```
//Transições
CPNTransition Movt1 = new CPNTransition();
Movt1.setId("t6");
Movt1.setName("Movt1");

CPNTransition Fim_t1 = new CPNTransition();
Fim_t1.setId("t7");
Fim_t1.setName("Finaliza Movt1");

CPNTransition Mov1 = new CPNTransition();
Mov1.setId("t8");
Mov1.setName("Mov1");

CPNTransition Fim_1 = new CPNTransition();
Fim_1.setId("t9");
Fim_1.setName("Finaliza Mov1");
```

Listagem 02. Parte do código detalhando transições da Rede do Movimento do Robô.

A **Listagem 02** mostra um trecho do código, onde são declaradas transições da Rede, estas transições serão responsáveis por representar ações específicas do movimento do robô. A transição *t6 (Movt1)* representa o início do movimento dos tornozelos, já a transição *t7 (Finaliza Movt1)* representa a finalização do movimento dos tornozelos. As transições *t8 (Mov1)* e *t9 (Finaliza Mov1),* são responsáveis por representar o movimento dos joelhos e coxa das pernas do robô. Claro que o código de detalhamento da Rede é bem maior, contendo declarações de lugares e arcos, se fosse posto neste relatório ocuparia espaço demais.

```
private Multiset t7_getBindings()
{
    Multiset allEnabledBindings = null;
    BindingSet thisBinding = new BindingSet();

    String thisTrans = "t7";
    String[] inputPlaces =
        new String[] {

               "p11", "p10"};


    allEnabledBindings.add(thisBinding);

    //TODO binding algorithm
    return allEnabledBindings;
}
```

Listagem 03. Código gerado pelo CPNEngine() e pelo Velocity para obter marcações.

A **Listagem 03** mostra o código para Simulação da Rede de Petri Colorida gerado a partir da classe *CPNEngine()* juntamente com o *Template* do *Velocity*. Este trecho de código demonstra a recuperação de marcações para a transição *t7 (Finaliza Movt1)*, observe que é declarado que, as marcações que irão adentrar a transição vêm do lugar *p11* e *p10* já reconhecidos e registrados pela classe *CPNEngine().*

```
private void t7_occur(BindingSet $bindings){
    Set transList$ = new HashSet();
    Vector pList$;
    Iterator i$;
    //TODO concretize bindings
    //Remove tokens
            ((Multiset) $).clear();
    ((Multiset) $).add(x);          removeTokens("p11", (Multiset) $);
    pList$ = (Vector) outputs.get("p11");
    if (pList$ != null){
     i$ = pList$.iterator();
     while (i$.hasNext()){
       transList.add(i$.next());
     }
    }
        ((Multiset) $).clear();
    ((Multiset) $).add(x);          removeTokens("p10", (Multiset) $);
    pList$ = (Vector) outputs.get("p10");
    if (pList$ != null){
     i$ = pList$.iterator();
     while (i$.hasNext()){
       transList.add(i$.next());
     }
    }
    //Add tokens
            ((Multiset) $).clear();
    ((Multiset) $).add(x);          addTokens("p12", (Multiset) $);
    pList$ = (Vector) outputs.get("p12");
    if (pList$ != null){
     i$ = pList$.iterator();
     while (i$.hasNext()){
       transList.add(i$.next());
     }
    }
    //evaluate modified transitions
    try{
        evalTransitionSet(transList$);
    }
    catch (Exception e){ e.printStackTrace(); }
}
```

Listagem 04. Código gerado pelo CPNEngine() e pelo Velocity para realizar disparos de transição.

A **Listagem 04** demonstra o código gerado para realizar disparos de transição, neste caso,

para a transição *t7 (Finaliza Movt1),*observe que, primeiramente a transição, por intermédio de funções do *SimulatorCore()*, remove as marcações dos pré-lugares (no caso, *p11* e *p10*) e adicionam as marcações destes lugares a pós-lugares (no caso *p12*), realizando assim, o disparo da transição.
Estes dois métodos serão gerados para todas as transições que forem registradas na Rede, ou seja, para cada *Tn* (sendo T uma transição) será gerado um *Tn_getBindings()*, para recuperar marcações de entradas) e um *Tn_occur(BindingSet $binding)*, para realizar os disparos da transição.

```
void setup()
{
  stepper2.setMaxSpeed(vel);
  stepper2.setAcceleration(1000.0);
  stepper2.setSpeed(1000);
  stepper2.moveTo(0);
  stepper1.setMaxSpeed(vel);
  stepper1.setAcceleration(1000.0);
  stepper1.setSpeed(1000);
  stepper1.moveTo(0);

  stepper3.setMaxSpeed(vel);
  stepper3.setAcceleration(1000.0);
  stepper3.setSpeed(1000);
  stepper3.moveTo(0);
  stepper4.setMaxSpeed(vel);
  stepper4.setAcceleration(1000.0);
  stepper4.setSpeed(1000);
  stepper4.moveTo(0);

  stepper5.setMaxSpeed(vel);
  stepper5.setAcceleration(1000.0);
  stepper5.setSpeed(2000);
  stepper5.moveTo(0);
  stepper6.setMaxSpeed(vel);
  stepper6.setAcceleration(100.0);
  stepper6.setSpeed(2000);
  stepper6.moveTo(0);
}
```

Listagem 05. Trecho do código em C usado para a programação do robô (inicialização-motores de passo).

A **Listagem 05** mostra um trecho do código na linguagem C usado na programação do movimento do robô. Neste trecho, trata-se da inicialização dos motores de passo (como visto na **Tabela 01**) usados para realizar a ação de movimentar o robô.

```
void movimentotl(){
 while(millis()-ult_t<=2000){
 stepper5.moveTo(-npto);
 stepper5.runSpeed();
 stepper6.moveTo(-npto);
 stepper6.runSpeed();

 stepper1.moveTo(1);
 stepper1.run();
 stepper2.moveTo(1);
 stepper2.run();
 stepper3.moveTo(1);
 stepper3.run();
 stepper4.moveTo(1);
 stepper4.run();
  }
 ult_t = millis();
}
```

Listagem 06**. Trecho do código em C usado na programação do robô (Movimento 1).**

A **Listagem 06** mostra o trecho do código em C do primeiro movimento feito pelo robô, logo, o movimento dos tornozelos (como visto na **Listagem 02**), sendo que os motores responsáveis por tal ação são denominados aqui como *stepper5* e *stepper6,* os outros *steppers* são chamados para que os motores permaneçam ativos e impeçam que o robô caia.

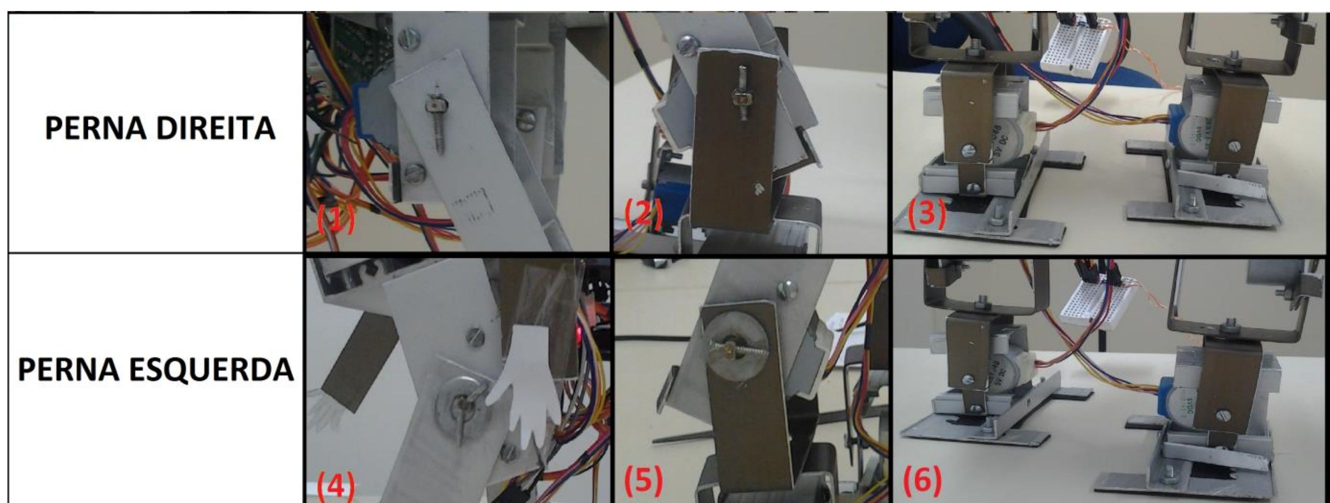

Figura 07. Movimentos robô.

A **Figura 07**, dá ênfase na movimentação individual dos motores da coxa das coxas, dos joelhos e dos tornozelos, de ambas as pernas, baseado nos códigos em C gerados da modelagem/ detalhamento da rede em Java, mostrando o momento em que o motor da coxa ((1) ou (4)), de uma determinada perna, avança, o motor do joelho ((2) ou (5)) recua e os tornozelos ((3) ou (6)) flexionam-se, ajudando os braços a manter o equilíbrio do robô, para realizar a ação de movimentar a perna.

A seguir, uma tabela, finalizando este tópico, comparando os feitos deste trabalho com os trabalhos correlatos.

Trabalho/ Critério	**Ferramenta deste trabalho**	**PNTCG**	**Linguagem Pencil**	**CoOperative Objects**	**Automatic Code Generation for Intel'8031**	**LOOPN++**
Linguagem Desenvolvida	Biblioteca JPetriSim (Java)	Linguagem PNTCG	Linguagem Pencil	C++	Não	Linguagem LOOPN++
Geração de Código	Sim	Sim	Não	Sim	Sim	Sim
Foco	Redes de Petri Colorida	Redes de Petri Lugar/ Transição	Redes de Petri Lugar/Transição	Rede de Petri Lugar/Transição	Rede de Petri Lugar/ Transição	Redes de Petri Orientada a Objetos
Código Gerado	Linguagem C	Linguagem C ou NXC	Não	C++	Assembly	C++

Tabela 01. Comparação deste trabalho com os trabalhos correlatos descrito no referencial teórico.

Como podemos perceber na **Tabela 01**, a metodologia apresentada neste trabalho assemelha-se aos trabalhos correlatos desenvolvendo novos métodos e atualizando os antigos da biblioteca *JPetriSim*, fazendo com que esta gere códigos a partir da especificação de uma Rede de Petri Colorida para a linguagem C, aplicando-se então a um sistema embarcado.

Conclusões

Diante do exposto, percebe-se que, a utilização dos Métodos Formais desempenha um importante papel para mensurar a previsibilidade e dependência no projeto de aplicações críticas e que o uso de métodos formais no desenvolvimento de software apresenta várias vantagens, por exemplo, programas (ou protótipos) podem ser gerados automaticamente e formalmente a partir de suas especificações. Pode-se provar também que determinados programas satisfazem determinadas propriedades e que o programa é uma realização da sua especificação, uma vez que, o interesse científico no estudo dos Métodos Formais é motivado basicamente pelo seu evidente potencial em aplicações industriais e pelo fato de demandarem a implementação de estratégias de solução complexas. Além do mais, a realização do projeto permitiu o estudo mais aplicado a Linguagem de Programação Java, sendo esta uma das mais potentes linguagens de programação usadas por profissionais da informática, aumentando ainda mais o conhecimento do aluno participante do projeto.

Referências bibliográficas

Alcoforado, M. G. Comunicação intermediada por protótipos. Master's thesis, Universidade Federal de Pernanbuco, 2007.

Batista, I. J. L., Modelelo de Navegação para Robôs Móveis Baseado em Redes de Petri Coloridas. Fortaleza, 2008.

Budd, R.; Kuhlenkamp, K.; Kautz, K. & Zulighoven, H. (1992). Prototyping: An Approach to Evolutionary System Development. Springer-Verlag New York, Inc., Secaucos, NJ, USA.

Cardoso, J.; Valette, R. Redes de Petri. Florianópolis: Editora da Universidade Federal de Santa Catarina, 1997, 212p.

Clarck, E. M. & Wing, J. M. Formal methods: State of the art and future directions. Volume 28. ACM Computing Surveys, 1996.

Conway, C.; Li, C. –H. & Pengelly, M. Pencil: A Petri Net Specification Language for Java. Math Departament Macquire University. Australia, 2002.

Cormen, T. H. [et al.]. Algoritmos: teoria e prática. Rio de Janeiro, 2002.

da Silva Barreto, R. A Time Petri Net-basead Methology for Embedded Hard Real-Time Software Synthesis. PhD thesis, Universidade Federal de Pernambuco, Centro de Informatica, 2005.

Dezani, H. Geração automática de código para microcontroladores aplicada a um ambiente de co-projeto de hardware e software. Master's thesis, Faculdade de Engenharia de Ilha Solteira – UNESP/FEIS, 2006.

Jensen, K. Coloured petri net: Basic concepts, Berlin: Springer 2nd edition, 1997.

Lakos, C. & Keen, C. LOOP++: A new language for object-oriented Petri net. In European Simulation Multiconference, 1994.

Oliveira, C. A. L., Simulação de Redes de Petri em Ambiente Java. Pernambuco, 2006.

Rangel, G. S. Protool: uma ferramenta de prototipação de software para o Ambiente PROSOFT. Master's thesis, Universidade Federal do Rio Grande do Sul, 2006.

Sibertin-Blanc, C. Cooperative objects: principles, use and implementation. 2001.

Xavier, C. S. C. L., Uma Especificação Executável Usando uma Linguagem de Redes de Petri no Domínio de Sistemas Embarcados. Manaus, 2011.

Ziviani, N. Projeto de Algoritmos: com implementações em Java e C++. São Paulo, 2007.