

CARLOS EDUARDO ALVES DA SILVA

**DESENVOLVIMENTO DE BIBLIOTECA PARA
APLICAÇÕES DE PNRD E PNRD INVERTIDA
EMBARCADAS EM ARDUINO**

UBERLÂNDIA, MINAS GERAIS

2017

CARLOS EDUARDO ALVES DA SILVA

DESENVOLVIMENTO DE BIBLIOTECA PARA APLICAÇÕES DE PNRD E PNRD INVERTIDA EMBARCADAS EM ARDUINO

Monografia de Conclusão de Curso
apresentada no curso de graduação em
Engenharia Mecatrônica da Universidade
Federal de Uberlândia, como parte dos
requisitos para obtenção de título de
**BACHAREL EM ENGENHARIA
MECATRÔNICA.**

Área de concentração: Engenharia Mecatrônica.

Orientador: Prof. Dr. José Jean-Paul Zanlucchi
de Souza Tavares

UBERLÂNDIA, MINAS GERAIS

2017

Dedico esse trabalho de conclusão de curso à minha família, que sempre me apoiou de todas as formas possíveis, e aos integrantes e frequentadores do MAPL que ajudaram na solução de vários problemas encontrados no caminho.

Agradecimentos

Aos meus pais e família, que sempre priorizaram a minha formação e educação, me apoiando nos mais diversos sentidos.

Ao professor Dr. José Jean-Paul Zanlucchi de Souza Tavares, pela orientação ativa e paciente, sempre incentivando e facilitando a realização desse trabalho, fornecendo inclusive, os materiais para o seu desenvolvimento.

Ao doutorando Alexandre Rodrigues de Souza por estar sempre disposto a me auxiliar nos aspectos relativos à programação em C++.

Ao doutorando Hiroshi Murofushi, que acompanhou esse projeto e foi essencial na resolução de questões práticas, principalmente no seu início.

Ao doutorando João Paulo da Silva Fonseca, que me ajudou na compreensão da PNRD e na análise de requisitos do sistema.

Aos demais frequentadores do MAPL, que de alguma forma ajudaram no desenvolvimento do projeto. Seja ajudando na soldagem das antenas RFID, seja explicando questões específicas da tecnologia RFID ou simplesmente tendo uma conversa agradável.

Aos meus colegas de turma e professores que foram essenciais para a minha formação profissional e pessoal.

SILVA, Carlos Eduardo Alves da. **Desenvolvimento de Biblioteca para Aplicações de PNRD e PNRD Invertida Embarcadas em Arduino**. 2017. 101 f. Trabalho de Conclusão de Curso (Graduação em Engenharia Mecatrônica) – Universidade Federal de Uberlândia, Uberlândia, 2017.

Resumo

A indústria metal-mecânica está em transformação, denominada indústria 4.0, que se baseia na comunicação entre os dispositivos e na internet das coisas. A complexidade e mutabilidade desses processos geram desafios na área de controle e modelagem de sistemas. Nesse contexto, esse trabalho tem como objetivo auxiliar na implementação de uma abordagem inovadora chamada PNRD (Redes de Petri Elementares inseridas em base de dados RFID) e sua variação, a PNRD invertida. Nessa abordagem, o modelo teórico amplamente estudado e desenvolvido das redes de Petri serve como base para a definição de uma estrutura de dados formal a ser gravada em etiquetas RFID. O resultado é um sistema de controle altamente adaptável, distribuído, escalável e de grande aplicabilidade. Assim, esse trabalho descreve o projeto, a implementação e a validação de uma biblioteca para a plataforma Arduino® que simplifica o desenvolvimento de novas aplicações que utilizam tanto a abordagem de PNRD como a de PNRD invertida.

Palavras-chaves: RFID, PNRD, redes de Petri, Arduino, sistema embarcado, controle discreto, modelagem de sistemas.

SILVA, C. E. A. **Development of a Library for PNRD and Inverted PNRD Applications Embedded in Arduino.** Monograph of the Mechatronics Engineering Course Completion, Federal University of Uberlândia, Uberlândia. Pages: 101.

Abstract

The metal-mechanic industry is in transformation, called industry 4.0, which is based on the communication between devices and on the Internet of Things. The complexity and changeability of those processes creates challenges in the field of systems modeling and control. In this context, this work has the purpose of assist the implementation of an innovative approach called PNRD (Elementary Petri Net inside a RFID distributed Database) and its variation, the inverted PNRD. In this approach, the theoretical model widely studied and developed of Petri Nets is used as a basis to define a data structure to be recorded on RFID tags. The result is a highly adaptive, distributed, scalable and applicable control system. Therefore, this work describes the design, implementation and validation of a library for the Arduino® platform which simplifies the development of new applications that use the PNRD approach as well as the inverted PNRD.

Keywords: RFID, PNRD, Petri nets, Arduino, embedded systems, discrete control, systems modeling.

Lista de Figuras

Figura 1 - Esquema de funcionamento da tecnologia RFID	18
Figura 2 - Modelos de etiquetas RFID utilizadas no desenvolvimento da biblioteca.	19
Figura 3 - Antena monoestática	21
Figura 4 – Radiação Antena Direcional.....	21
Figura 5 – Radiação Antena Omnidirecional.....	21
Figura 6 – Feixe de antenas direcionais de diferentes ganhos.	22
Figura 7 – Polarização linear versus polarização circular.	22
Figura 8 - Exemplo de Rede de Petri	24
Figura 9 - Exemplo de modelagem de um semáforo com marcação no estado verde	25
Figura 10 - Exemplo de tratamento de concorrência com Redes de Petri.	26
Figura 11 - Representação gráfica das possíveis marcações no modelo de dois semáforos.....	27
Figura 12 - Representação gráfica de um exemplo de disparo	31
Figura 13 - Exemplo de Rede de Petri na verificação de mancais.	32
Figura 14 - Arduinos utilizados no desenvolvimento da biblioteca, Arduino® Uno à esquerda e a Arduino® Mega à direita.....	34
Figura 15 - Arquitetura dos módulos da biblioteca	37
Figura 16- Exemplo de fragmentação de memória.	39
Figura 12 - Leitor RFID NFC PN532	46
Figura 18 - Esquema do formato NDEF	53
Figura 19 - Representação simplificada de um processo de verificação de mancais usando Redes de Petri.	58

Figura 20 - Montagem física do simulador de uma máquina verificadora de peça. O botão vermelho indica que a peça foi reprovada, o amarelo indica que ela foi aprovada.	58
Figura 21 - Funcionamento da simulação de aplicação usando PNRD	59
Figura 22 - Ilustração do teste com PNRD invertida	59
Figura 23 - Visualizador de histórico de uma etiqueta.....	60

Lista de Tabelas

Tabela 1 - Tipos de etiquetas RFID de acordo com sua frequência.....	20
Tabela 2 – Isolamento de cabos comumente usados e suas perdas.....	23
Tabela 3 - As características de sistemas não-sequenciais e a sua representação com Redes de Petri.....	28
Tabela 4 - Estruturas de representativas de arcos em Redes de Petri	42
Tabela 5 - Lista de métodos públicos da classe PetriNet.....	48
Tabela 6 - Métodos Públicos exclusivos da classe Pnrd.....	51
Tabela 7 - Relação entre o tipo de dado e o bit de referência no cabeçalho de informação presente.....	54

Lista de Abreviaturas e Siglas

HF – Alta frequência (em inglês: *High Frequency*);

I²C - Circuito Inter Integrado (em inglês: *Inter-Integrated Circuit*)

LF – Baixa frequência (em inglês: *Low Frequency*);

MAPL – Laboratório de Planejamento Automático de Manufatura;

NDEF – Formato de troca de informações em comunicações por campo de proximidade (em inglês: *NFC Exchange Format*);

NFC – Comunicação por campo de proximidade (em inglês: *Near Field Communication*);

PNRD – Redes de Petri Elementares inseridas em base de dados RFID (em inglês: *Elementary Petri Net inside a RFID distributed Database*);

RF – Radiofrequência;

RFID – Identificação por Radiofrequência (em inglês: *Radio-Frequency Identification*);

RP – Rede de Petri;

SHF – Super Alta Frequência (em inglês: *Super High Frequency*);

SO – Sistema Operacional;

SPI – Interface Serial Periférica (em inglês: *Serial Peripheral Interface*);

UFU – Universidade Federal de Uberlândia.

UHF – Ultra alta frequência (em inglês: *Ultra High Frequency*);

USB – Barramento Serial Universal (em inglês: *Universal Serial Bus*);

Lista de Símbolos

E_n – Lugar de índice n em uma rede de Petri;

T_n – Transição de índice n em uma rede de Petri;

P – Conjunto dos Lugares em uma rede de Petri;

T – Conjunto das Transições em uma rede de Petri;

A – Conjunto dos arcos em uma rede de Petri;

M – Função de marcações em uma rede de Petri;

M_k – Vetor de marcações;

A^t – Matriz de incidência;

t – Quantidade de transições em uma rede de Petri;

e – Quantidade de lugares em uma rede de Petri;

max_{in} – Quantidade máxima de entradas de uma transição que faz parte do conjunto de transições em uma rede de Petri;

max_{out} – Quantidade máxima de saída de uma transição que faz parte do conjunto de transições em uma rede de Petri;

M_{LA} – Quantidade de memória em *bytes* utilizada pelas listas de adjacência;

M_{MI} – Quantidade de memória em *bytes* utilizada pela matriz de incidência;

k_{in} – Quantidade de arcos na lista de entradas de uma determinada transição;

k_{out} – Quantidade de arcos na lista de saídas de uma determinada transição;

SUMÁRIO

Agradecimentos	4
INTRODUÇÃO	15
1.1 OBJETIVOS	16
1.2 JUSTIFICATIVA	17
FUNDAMENTAÇÃO TEÓRICA.....	18
2.1 TECNOLOGIA RFID	18
2.1.1 – Etiquetas	19
2.1.2 – Antenas	20
2.1.3 – Leitores	23
2.1.4 – Infraestrutura de comunicação	23
2.1.5 – Aplicação.....	23
2.2 REDES DE PETRI	24
2.2.1 – Conceito de disparo	25
2.2.1 – Representatividade de Sistemas.....	26
2.2.3 – Definição formal	29
2.3 PNRD	31
2.3 PNRD INVERTIDA.....	33
2.4 ARDUINO.....	34
METODOLOGIA E DESENVOLVIMENTO.....	35
3.1 DEFINIÇÃO DE REQUISITOS.....	35
3.2 PROJETO DE SOFTWARE	36
3.2.1 – Considerações sobre as estruturas de dados utilizadas	38
3.2.2 – Vetor de marcações e vetor de disparos	39
3.2.3 – Estados e transições	40
3.2.4 – Representação dos arcos da rede de Petri	41
3.2.5 – Condições	44

3.2.6 – Histórico da etiqueta.....	45
3.3 IMPLEMENTAÇÃO	46
3.3.1 – Módulo Redes de Petri.....	47
3.3.2 – Módulo PNRD	49
3.3.3 – Módulo Leitor PN532.....	52
3.3.4 – Exemplo de código de aplicação.....	54
3.4 Testes	56
3.4.1 Testes do módulo de Rede de Petri.....	56
3.4.2 Testes do módulo de PNRD.....	56
3.4.3 Testes do módulo do leitor PN532	57
3.4.5 Testes gerais	57
RESULTADOS E CONCLUSÕES.....	61
4.1 Resultados	61
4.3 Conclusão	61
4.2 Trabalhos futuros	62
BIBLIOGRAFIA	63
Apêndices.....	65
Apêndice A: Tabela de Comparação entre as Estruturas de Dados para armazenagem da informação dos arcos em uma rede de Petri.	66
Apêndice B: códigos da simulação de aplicação utilizando abordagem PNRD	67
Máquina 1: Gravação inicial das etiquetas.....	67
Máquina 2: Verificação de peças	68
Máquina 3 : Gerenciamento de estoque	70
Apêndice C : códigos da simulação de aplicação utilizando abordagem PNRD invertida	72
Robô com movimentação em sentido horário	72
Robô com movimentação em sentido anti-horário	73
Visualizador de Histórico da Etiqueta.....	75

Apêndice D : arquivos de código da biblioteca desenvolvida	76
PetriNet.h	76
PetriNet.cpp	78
Pnrd.h.....	90
Pnrd.cpp.....	92
Pn532NfcReader.h.....	96
Pn532NfcReader.cpp	96

INTRODUÇÃO

A identificação por radiofrequência ou RFID é uma tecnologia baseada na utilização de campos eletromagnéticos para a escrita e leitura de dados em dispositivos chamados etiquetas ou *tags* através de uma antena.

Essas etiquetas podem ser anexadas a objetos de forma a acompanhar o seu trajeto físico, disponibilizando seus dados ao longo de uma cadeia produtiva. Ou podem ser associadas a lugares ou objetos físicos imóveis, permitindo que os seus dados estejam acessíveis naquela determinada localidade.

Apesar de a tecnologia RFID existir desde a segunda guerra mundial (DOMDOUZI, KUMAR, ANUMBA, 2007), somente nas últimas duas décadas essa tecnologia se tornou atrativa comercialmente devido a diminuição do valor das etiquetas e da adoção de padrões para as antenas e etiquetas RFID (NATH, REYNALDS, WANT, 2006).

Atualmente, as aplicações da tecnologia RFID são diversas. Na indústria, ela pode ser usada no controle da distribuição e armazenamento de materiais, peças e produtos (WANT, 2006). No setor de transportes, exemplos de utilização seriam a etiquetagem de bagagens no transporte aéreo e a cobrança automática em pedágios e estacionamentos. No setor hoteleiro e empresarial, ela é utilizada como forma de controle de acesso. No comércio, a tecnologia RFID permite a detecção de furtos e controle de estoques. Na área da medicina, ela pode ser utilizada no gerenciamento de amostras de sangue e na utilização de pulseiras etiquetadas com informações dos pacientes. Em bibliotecas, são utilizadas para catalogação e controle de acervo. Enfim, são bastante variados os usos do RFID no presente. Todavia, a maioria dessas aplicações se baseia na substituição dos códigos de barras por etiquetas RFID.

A rede de Petri (RP), por sua vez, é uma técnica de modelagem que permite a representação de sistemas, especialmente sistemas concorrentes, utilizando como alicerce uma forte base matemática (MACIEL, LINS, CUNHA, 1996). Utilizada como uma ferramenta de análise e controle de sistemas discretos, ela é capaz de modelar casos de concorrência, paralelismo, assincronismo, conflitos e *deadlocks* (ZHANG, WANG, WU, 2016). Tal versatilidade faz com que as redes de Petri sejam aplicáveis

na descrição e visualização de sistemas com fluxo de trabalho complexo, que é o caso de vários processos de produção da atualidade.

Tavares e Saraiva (2010) propuseram um método de integração do modelo de redes de Petri com sistemas de controle RFID, ao definir uma estrutura formal de dados para as etiquetas RFID. Essa organização foi denominada de “Redes de Petri Elementares inseridas em base de dados RFID” ou PNRD. A adoção dessa estrutura permite que a etiqueta contenha a informação do estado do produto em tempo real e o processo no qual ele está inserido. Dessa forma, as etiquetas se tornam um banco de dados distribuído cuja atualização é efetuada em tempo de execução. Além disso, leitores de etiqueta localizados ao longo da cadeia produtiva possibilitam a identificação automática de eventos inesperados na linha de produção em comparação com o processo pré-programado na etiqueta.

No entanto, em determinadas aplicações, como a descrita por Fonseca e Tavares (2017), a estrutura de dados da PNRD precisa ser modificada. Essa nova estrutura será denominada aqui como PNRD invertida. Na PNRD invertida a etiqueta pertence a um local fixo, enquanto o leitor RFID está presente em um agente ativo móvel, como uma pessoa ou um robô.

Para realizar a implementação de um software baseado na PNRD ou na PNRD invertida, é necessário um conjunto de rotinas. Algumas rotinas necessárias em ambos os casos são: a leitura e gravação de etiquetas, a atualização do estado do objeto e a verificação de erros. Essas rotinas em comum indicam a possibilidade da construção de uma biblioteca que facilite a implementação de ambos os casos.

1.1 OBJETIVOS

Os objetivos primários desse trabalho são projetar e desenvolver o código de uma biblioteca que implemente as lógicas de PNRD e PNRD invertida, de modo a facilitar o desenvolvimento futuro de aplicações em sistemas embarcados baseadas nesses modelos. A plataforma de desenvolvimento escolhida foi o Arduino®. Dessa forma, pode-se listar como objetivos específicos desse projeto:

- Desenvolver uma interface de desenvolvimento que permita a fácil implementação da PNRD e a PNRD invertida para sistemas embarcados, em especial, o Arduino®;
- Disponibilizar ferramentas que permitam a adaptabilidade da biblioteca de acordo com a aplicação;
- Realizar testes modulares e gerais para validação do software criado;

1.2 JUSTIFICATIVA

A complexidade dos processos produtivos da atualidade, juntamente com a volatilidade de demanda e as constantes mudanças tecnológicas, fazem com que a típica abordagem que separa técnicas de modelagem, validação e controle de processos seja muito dispendiosa, exigindo constante retrabalho na análise e organização da produção.

Nesse contexto, o trabalho realizado vem auxiliar na aplicação e disseminação de uma solução que permite não só a reunião de estratégias de controle e modelagem, como também vai de encontro com tendências da indústria 4.0, especificamente a comunicação máquina-máquina e Internet das Coisas.

A PNRD e a PNRD invertida promovem a descentralização da informação relacionada não só ao produto, mas também ao próprio processo produtivo. As etiquetas RFID permitem a comunicação de uma máquina para outra, sem o uso de um servidor central, em uma arquitetura típica da Internet das Coisas. Tal estrutura facilita a integração de novas máquinas ou módulos de produção, permitindo a independência e escalabilidade dos sistemas de produção.

Esse trabalho apresenta a fundamentação teórica no capítulo 2, seguindo da descrição e da metodologia e desenvolvimento no capítulo 3. Por fim, os resultados e conclusões são apresentados no capítulo 4, seguidos pelas referências bibliográficas. O código-fonte da biblioteca desenvolvida se encontra em apêndice.

FUNDAMENTAÇÃO TEÓRICA

2.1 TECNOLOGIA RFID

A tecnologia de identificação por radiofrequência, ou RFID, utiliza ondas eletromagnéticas entre 125 kHz e 5,8 GHz, para armazenar e ler informações de pequenos dispositivos chamados etiquetas ou *tags*. Um sistema RFID é basicamente estruturado em cinco componentes: etiqueta, antena, leitor, infraestrutura de comunicação e aplicação (AHSAN, SHAH, KINGTON, 2010). A Figura 1 mostra como esses componentes se interagem.

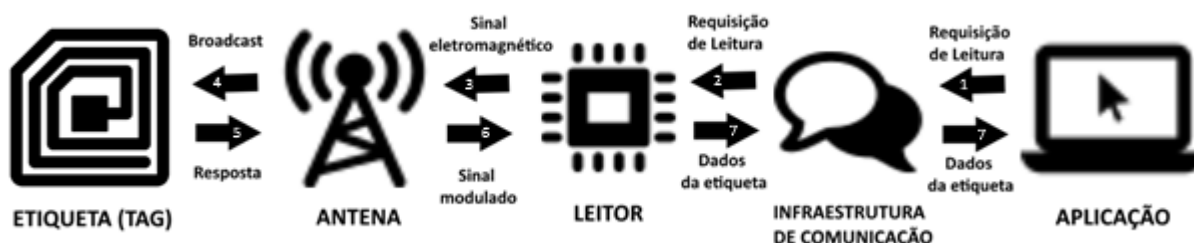


Figura 1 - Esquema de funcionamento da tecnologia RFID. Fonte dos ícones:

<<https://icons8.com/>>, acesso em 07/12/2017

Na maioria das aplicações de RFID, o fluxo de dados ocorre da seguinte forma:

- 1) A aplicação realiza uma requisição de leitura;
- 2) Essa requisição é informada ao leitor por meio da infraestrutura de comunicação;
- 3) O leitor gera um sinal de radiofrequência (RF) para detectar etiquetas próximas;
- 4) A antena transmite, via broadcast, o sinal do leitor;
- 5) Ao receber o sinal, a etiqueta envia uma resposta à antena através da modulação do sinal recebido;
- 6) O leitor recebe o sinal RF modulado da antena e decodifica a informação presente.
- 7) Utilizando a infraestrutura de comunicação, o leitor informa os dados recebidos para a aplicação.

Uma vantagem na utilização do RFID para a identificação de objetos é a capacidade que as etiquetas RFID possuem de alterar dinamicamente a sua informação interna. Além disso, se comparada ao código de barras, a tecnologia RFID facilita a leitura, pois o sensor não precisa apontar diretamente para a etiqueta, só precisa estar em seu alcance. Também é possível ler várias etiquetas de uma única vez, aumentando a rapidez do processo.

2.1.1 – Etiquetas

Basicamente, a etiqueta pode ser caracterizada como um transponder de ondas RF. Em sua maioria, seu principal componente é um microchip capaz de armazenar informação. Existem etiquetas capazes de guardar somente seu número de identificação, enquanto outras armazenam até 64 KB de informação (FUJITSU LTD., 2014). Existem etiquetas de diferentes formas e tamanhos, como indicado na Figura 2.

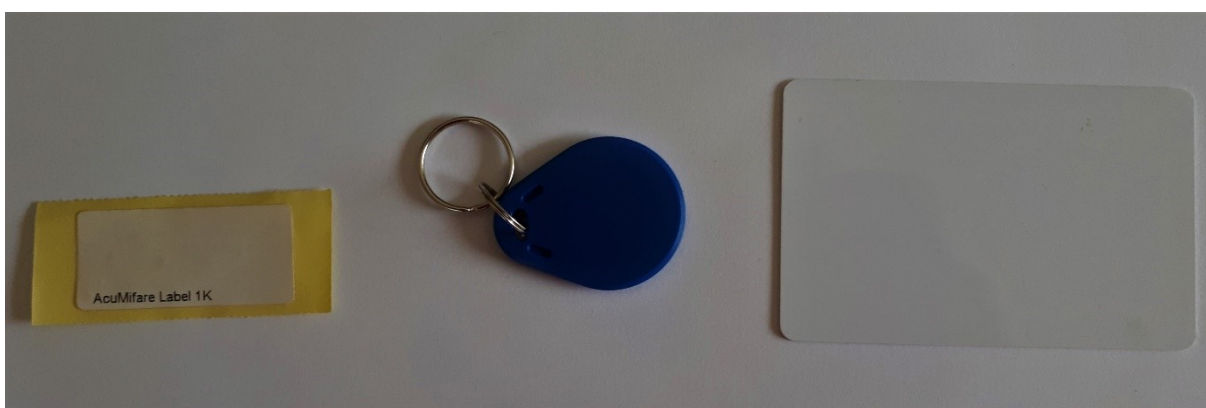


Figura 2 - Modelos de etiquetas RFID utilizadas no desenvolvimento da biblioteca.

Em etiquetas do tipo *rewrite*, é possível modificar a informação salva por meio de ondas RF, enquanto que em etiquetas *read-only*, essa alteração só é possível reprogramando eletronicamente o microchip da etiqueta (AHSAN, SHAH, KINGTON, 2010). Em alguns casos, as etiquetas podem ser mistas, apresentando áreas de memória do tipo *read-only* e do tipo *rewrite*.

As etiquetas podem ser classificadas como passivas, ativas ou semipassivas (MUROFUSHI, 2011). As etiquetas passivas não necessitam de fonte de energia própria, elas utilizam a energia proveniente das ondas produzidas pelas antenas RFID para energizar o chip. Por sua vez, as ativas possuem uma fonte de energia própria,

como uma bateria ou a energia solar, o que propicia a leitura de uma maior distância. No entanto, se comparadas com as passivas, seu custo é mais elevado, possuem um tamanho maior e tem uma vida útil reduzida. As etiquetas semipassivas também possuem fonte de energia própria, mas dependem da energização do sinal para a sua ativação.

Outra forma de classificação tanto das etiquetas quanto dos leitores RFID é quanto à frequência do sinal. A tabela 1 apresenta as faixas de frequência dos sistemas RFID.

Tabela 1 - Tipos de etiquetas RFID de acordo com sua frequência. (LI, MENG, 2016)

Tipo da Etiqueta	Frequência
Baixa frequência (LF)	125 -134,3 kHz
Alta frequência (HF)	13,56 MHz
Ultra alta frequência (UHF)	860 - 960 MHz
Super alta frequência (SHF)	2,45 – 5,8 GHz

2.1.2 – Antenas

A antena é o componente responsável pela transmissão e recebimento do campo magnético usado na transferência de informação entre o leitor e as etiquetas. Dependendo das suas características construtivas, ela pode favorecer a leitura de etiquetas em certas direções e fornecer um maior alcance de leitura.

A antena opera como uma ponte entre as *tags* e o leitor, realizando a modulação dos sinais de radiofrequência enviados pelo leitor, e a demodulação dos sinais recebidos das *tags*. A figura 3 apresenta uma antena monoestática, ou seja, que realiza as operações de leitura e escrita nas *tags*. Existem antenas que só realizam uma das operações, sendo chamadas de biestáticas.



Figura 3 - Antena monoestática

As antenas são classificadas quanto à diretividade, ganho, polarização e frequência de operação. Quanto à diretividade a antena, esta pode ser direcional, irradiando para apenas uma direção em que a antena esteja apontada, como na Figura 4, ou omni-direcional, em que irradia para todas as direções, como na Figura 5.

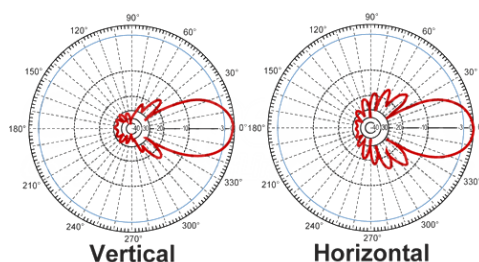


Figura 4 – Radiação Antena Direcional. Fonte: <<http://blog.atlasrfidstore.com>>, acesso em 26/09/2016

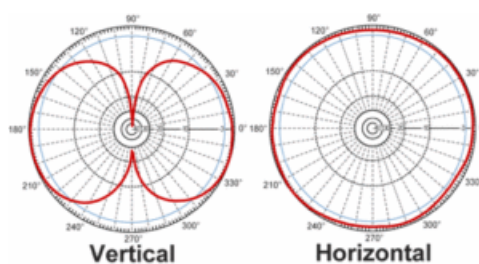


Figura 5 – Radiação Antena Omnidirecional. Fonte: <<http://blog.atlasrfidstore.com>>, acesso em 26/09/2016

O ganho determina quantas vezes a antena ampliará o sinal do leitor, é dado em decibéis (dB) e é atrelado à largura do feixe emitido por uma antena direcional, como ilustra a Figura 6. Sendo que uma antena de menor ganho implica numa maior largura de feixe, e vice-versa.

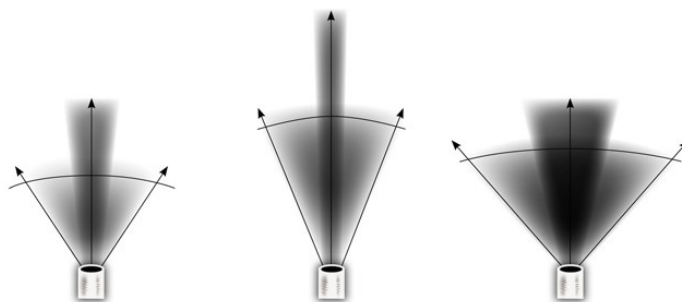


Figura 6 – Feixe de antenas direcionais de diferentes ganhos. Fonte: <http://blog.atlasrfidstore.com>, acesso em 26/09/2016

A polarização da antena define como o sinal RF viaja pelo meio depois de enviado pela antena, podendo esta ser linear ou circular, como mostra a Figura 7.

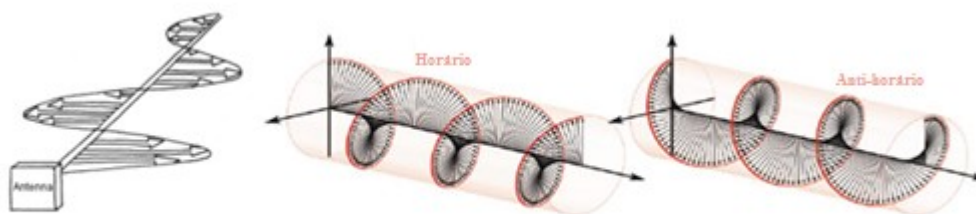


Figura 7 – Polarização linear versus polarização circular. Fonte: <http://blog.atlasrfidstore.com>, acesso em 26/09/2016.

A polarização linear, é caracterizada pela emissão em apenas um plano, podendo ser vertical ou horizontal. Este tipo de antena exige que as *tags* a serem lidas sempre estejam em uma orientação fixa para uma leitura confiável. Como este tipo de antena apresenta irradiação concentrada em apenas um plano, o sinal consegue se propagar por maiores distâncias com maior potência.

Já as antenas de polarização circular, emitem seu sinal de maneira helicoidal, podendo ser no sentido horário ou anti-horário, como ilustra a Figura 7. Desta maneira, transmitindo o sinal em vários planos simultaneamente, a leitura das *tags* pode ser feita em diversas orientações, porém, ao custo da distância de leitura ser menor que sua contraparte.

Também deve-se levar em consideração a conexão entre a antena e o leitor, realizada por um cabo coaxial que gera também uma perda em dB em função do comprimento do cabo e seu isolamento. São dados quatro exemplos de cabos com diferentes isolamentos e, conseqüentemente, diferentes perdas na tabela 2.

Tabela 2 – Isolamento de cabos comumente usados e suas perdas. Fonte: <http://blog.atlasrfidstore.com>, acesso em 26/09/2016

Cabo	LMR-195	LMR-240	LMR-400	LMR-600
Perda/30m (dB)	11.1	7.6	3.9	2.5

2.1.3 – Leitores

O leitor é um sistema eletrônico que gera e interpreta os sinais recebidos pela antena. Ele embute algoritmos contra colisões e pode, por vezes, operar com mais de uma frequência de sinal (AHSAN, SHAH, KINGTON, 2010). Alguns leitores tem a capacidade de conectar-se com múltiplas antenas, enquanto outros estão acoplados a uma única antena, formando um dispositivo único.

O leitor RFID controla diretamente a antena, de maneira a modular os comandos de leitura e escrita a serem enviados para as *tags* via RF, e demodular as respostas recebidas pela antena advindas da *tag*.

2.1.4 – Infraestrutura de comunicação

A infraestrutura de comunicação é o canal pelo qual o leitor transmitirá a informação recebida para a aplicação de alto nível. Entre as formas de comunicação possíveis pode-se citar os protocolos seriais como o RS-232, RS-485, I²C e o SPI, ou ainda protocolos sem fio como o WiFi e o Bluetooth.

2.1.5 – Aplicação

A aplicação é o software que vai tratar a informação recebida pelo leitor. Ela pode estar ligada a um banco de dados de modo a sincronizar a informação do banco com a leitura das informações da etiqueta e/ou fornecer uma interface para o usuário interpretar os dados lidos.

2.2 REDES DE PETRI

As redes de Petri (RP) são um modelo conceitual gráfico e matemático introduzido por Carl Adam Petri em 1962 (MURATA, 1989). Amplamente utilizada e estudada ao longo dos anos, essa forma de modelagem consiste em definir dois tipos de elementos: lugares e transições. Os lugares são tipicamente representados por circunferências e as transições por traços ou retângulos.

Em uma RP, lugares são ligados a transições por meio de arcos direcionados. Um arco obrigatoriamente liga um lugar a uma transição. Portanto, não é possível a ligação direta entre duas transições ou dois lugares. A seguir, na Figura 8, pode-se observar um exemplo simples de uma RP com 4 transições e 5 lugares.

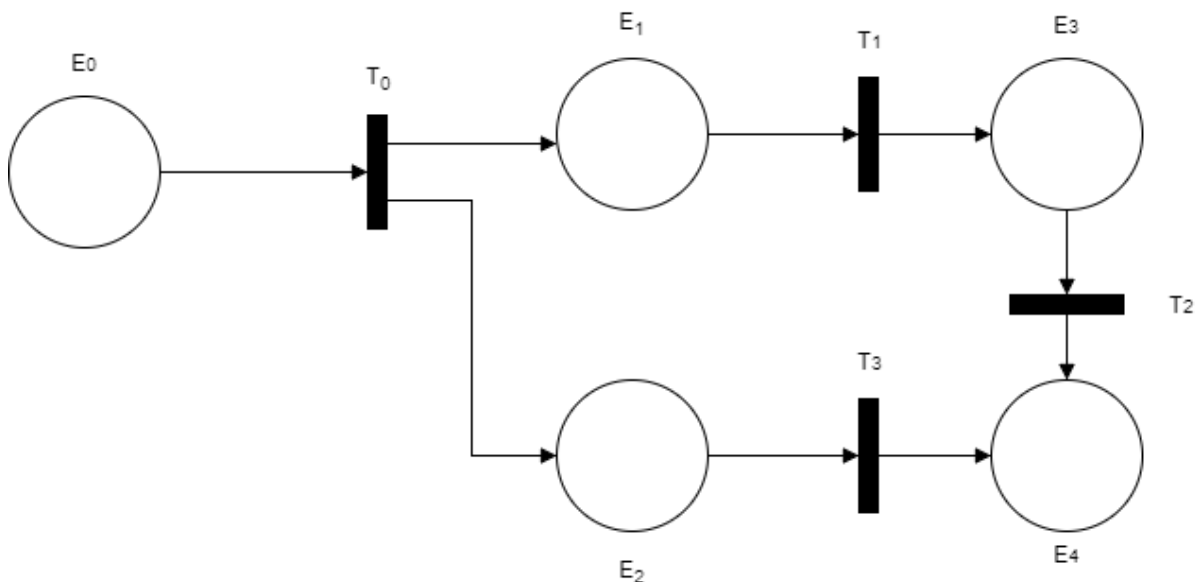


Figura 8 - Exemplo de Rede de Petri

Se o arco apontar para a transição, o lugar é definido como uma entrada da transição. Caso contrário, este é uma saída da transição. Dessa forma, no exemplo anterior, o lugar E₁ é uma entrada da transição T₁ e os lugares E₁ e E₂ são saídas da transição T₀.

Além dos lugares e transições, uma RP é definida com um conjunto de fichas ou *tokens* iniciais. Essas fichas indicam em que estado ou em que conjunto de estados o sistema modelado se encontra atualmente. Elas são representadas por um ponto dentro do lugar correspondente. O conjunto das fichas é denominado como marcação.

Por exemplo, podemos representar um semáforo utilizando as redes de Petri. Para isso, cria-se três lugares (vermelho, verde, amarelo) e três transições entre esses

lugares, como representado na Figura 9. Uma marcação é utilizada para indicar que o semáforo está atualmente no estado verde, amarelo ou vermelho.

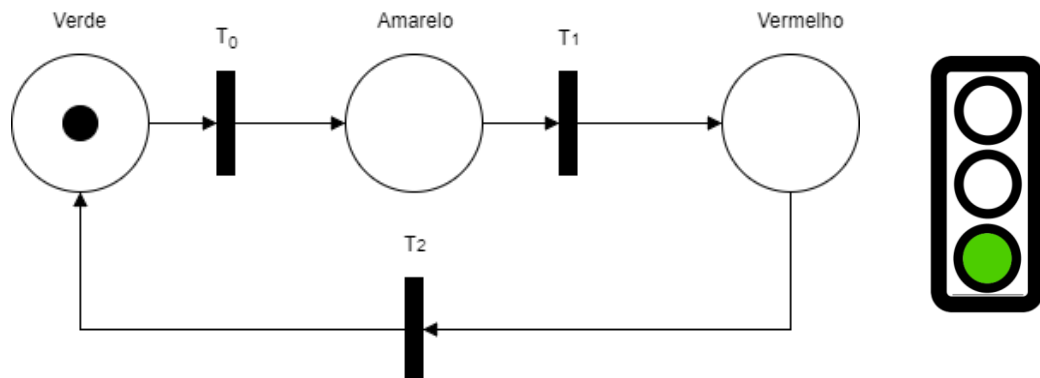


Figura 9 - Exemplo de modelagem de um semáforo com uma ficha no estado verde

É importante ressaltar que o estado amarelo não é indicado por uma transição, mas sim por um estado. A característica efêmera do estado amarelo pode dar a impressão que ele é uma transição. No entanto, no modelo de redes de Petri, a transição é indivisível, ou seja, sem gradação visível.

Ao observarmos um sistema em determinado instante de tempo, ele estará em um estado ou em outro. O semáforo pode estar verde, vermelho ou amarelo; não existe estado perceptível entre dois desses estados. Ao realizar a afirmação que o semáforo amarelo é uma transição, o modelo se torna incapaz de representar uma dessas possibilidades. Pois as fichas só podem estar contidas dentro de lugares, não dentro de transições.

2.2.1 – Conceito de disparo

Um conceito importante nas redes de Petri é o disparo. Um disparo acontece quando uma transição ou um conjunto de transições é acionado de forma a alterar a marcação de uma RP. No exemplo do semáforo, podemos disparar a transição T_0 para alterar o estado do semáforo de verde para amarelo.

Uma transição somente pode ser disparada se existirem marcações suficientes nos seus lugares de entrada. Por exemplo, a transição T_1 só é disparada se houver pelo menos uma ficha no lugar amarelo. Caso essa regra não seja satisfeita, será dito que o disparo resultou em uma exceção.

Em algumas redes de Petri os arcos entre transições e lugares possuem pesos. Se o peso estiver associado a uma entrada, ele indica a quantidade de fichas

consumidas no disparo da transição. Se ele estiver em uma saída, o mesmo indicará a quantidade de fichas criadas após o disparo. No restante desse trabalho, restringiremos as redes de Petri para redes com pesos unitários.

Em redes de Petri não-autônomas, o disparo pode ser habilitado por eventos externos (DAVID, ALLA, 2010). Dessa forma, podemos associar cada transição a uma condição específica do sistema. No caso do semáforo, por exemplo, podemos estabelecer uma condição de tempo mínimo para disparo de cada transição. Dessa forma, mesmo que o semáforo se encontre no verde, ele passará para o estado amarelo só após decorrer um tempo mínimo de espera.

2.2.1 – Representatividade de Sistemas

A grande vantagem da rede de Petri é a sua capacidade de representar sistemas caracterizados como sendo concorrentes, assíncronos, distribuídos, paralelos, não determinísticos e/ou estocásticos (MURATA, 1989). Pode-se ilustrar essa capacidade ao se ampliar o exemplo anterior para dois semáforos que controlam a passagem de carros em um cruzamento. Nesse caso, temos um exemplo de concorrência de recursos e de paralelismo. A Figura 10 ilustra uma possível representação do sistema descrito em redes de Petri, baseada na apresentação de AALST (201-) sobre redes de Petri.

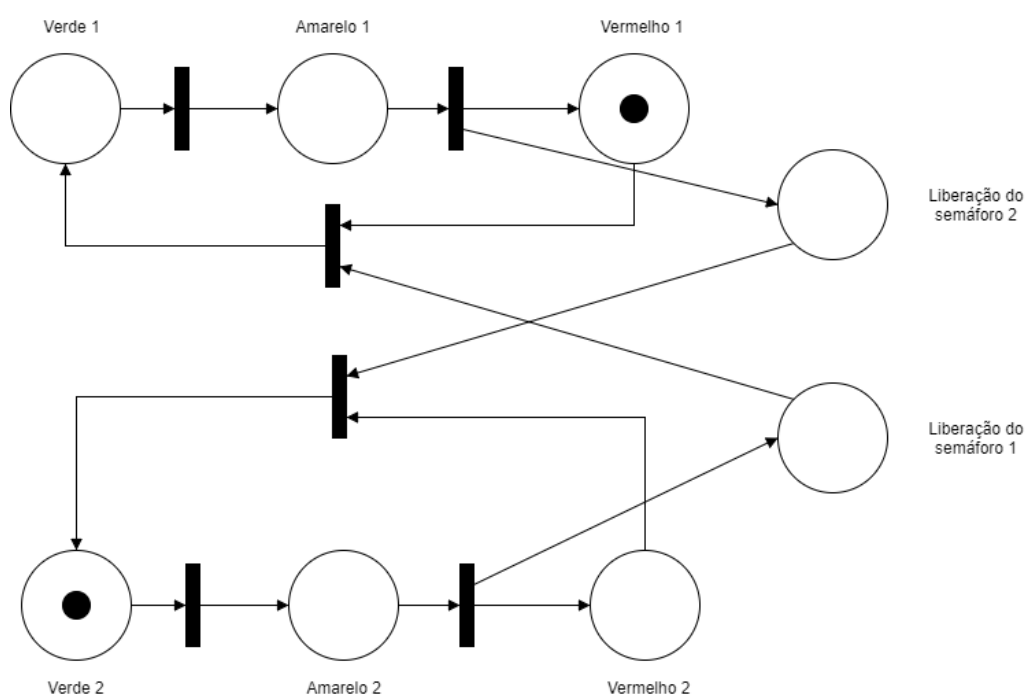


Figura 10 - Exemplo de tratamento de concorrência com Redes de Petri.

Nessa solução, cada semáforo é representado por uma marcação que percorre os estados verde, amarelo e vermelho. No entanto, para passar do estado vermelho para o verde, cada semáforo necessita de uma marcação de liberação, criada somente quando o outro semáforo passa da cor amarela para a vermelha. Na Figura 11 podemos ver graficamente o ciclo do modelo criado.

Caso a marcação inicial for igual a qualquer uma das representadas na Figura 6, a Rede de Petri nunca indicará que ambos os semáforos estão verdes. Dessa forma, o próprio modelo é capaz de impor as restrições relativas à característica de concorrência do sistema modelizado.

A Tabela 3 mostra exemplos clássicos de sistemas não-sequenciais e representações possíveis desses sistemas usando Redes de Petri, ilustrando a alta capacidade de representação desse modelo.

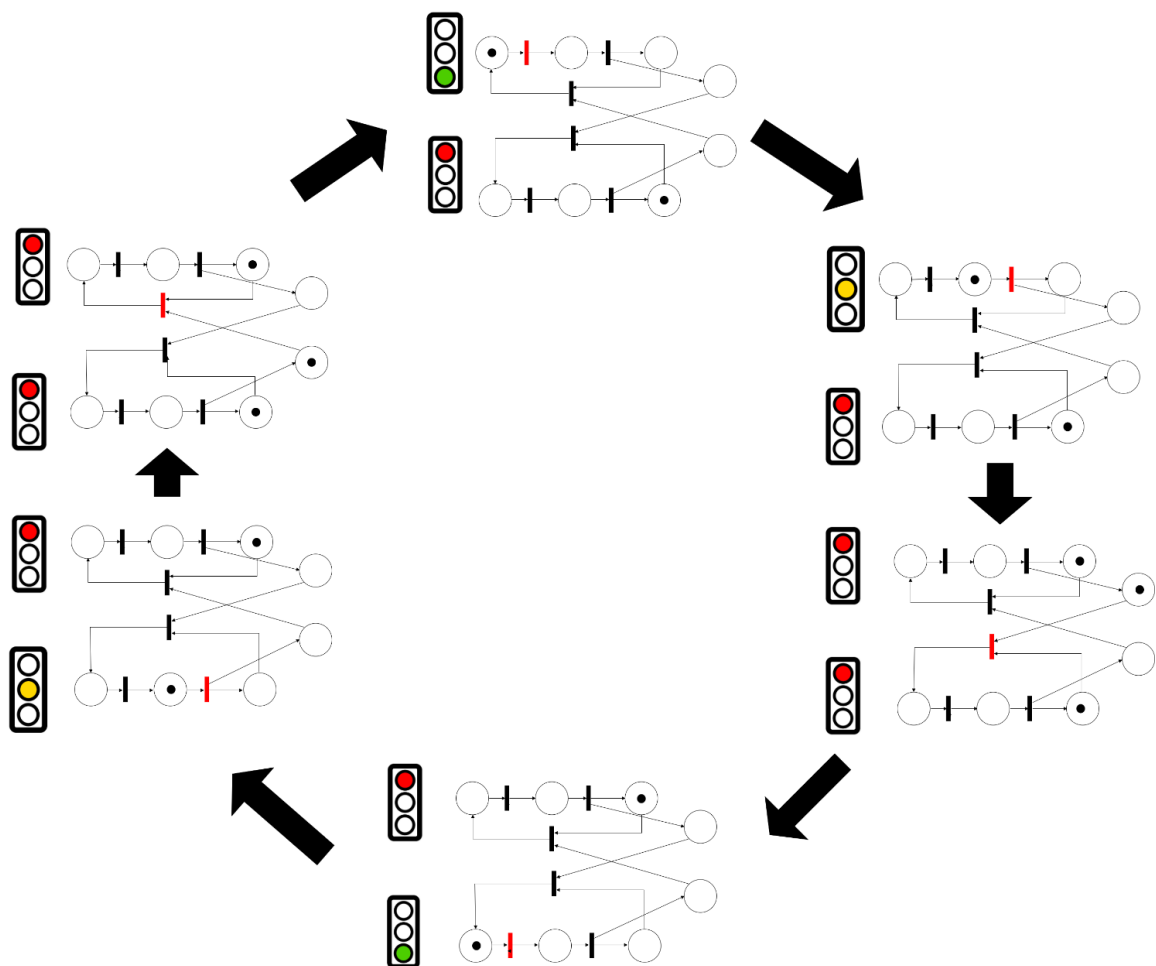
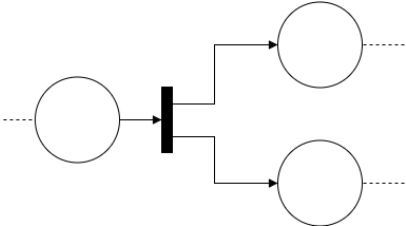
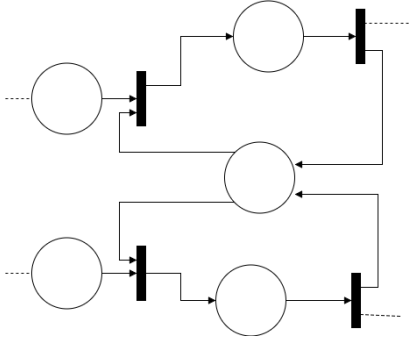
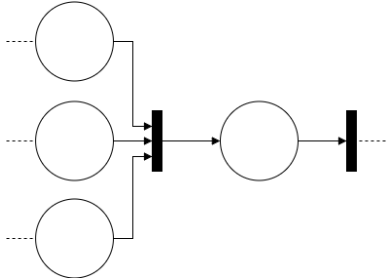
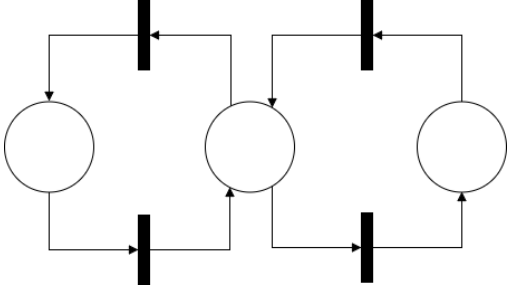
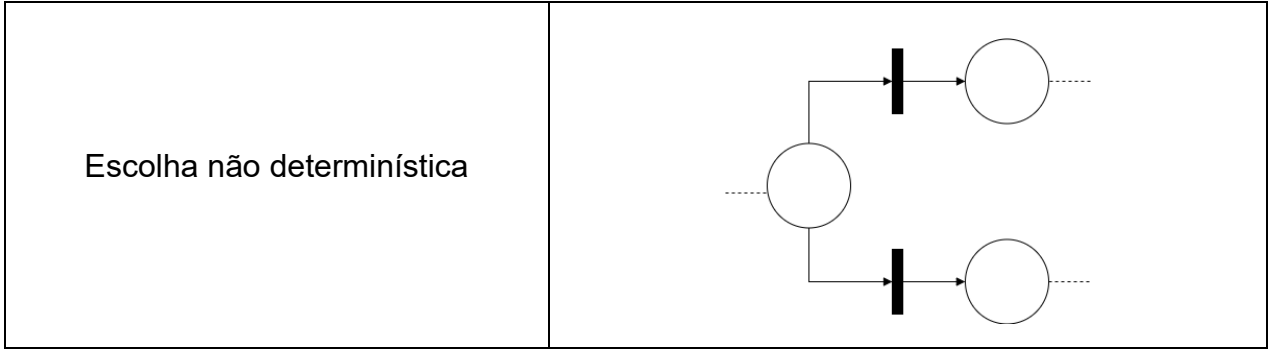


Figura 11 - Representação gráfica das possíveis marcações no modelo de dois semáforos. As transições em vermelho indicam que elas estão habilitadas, podendo ser disparadas.

Além do seu poder de representação, as redes de Petri também permitem a análise do sistema modelado, identificando conflitos, ciclos, comportamentos inesperados e *deadlocks*.

Tabela 3 - As características de sistemas não-sequenciais e a sua representação com Redes de Petri

Característica do Sistema	Possível representação
Paralelismo	
Concorrência	
<i>Rendez-vous</i>	
Máquina de estados	



2.2.3 – Definição formal

Após apresentadas as suas principais características, a rede de Petri com marcação será definida formalmente como sendo uma tupla de 4 elementos (P, T, A, M) . Sendo que:

P é um conjunto finito não-nulo de lugares.

T é um conjunto finito não-nulo de transições.

A é o conjunto de arcos direcionados de estados para transições e de transições para estados, tal que $A \subseteq (P \times T) \cup (T \times P)$.

M é a função do número de marcações presentes em cada estado, de modo que $M : P \rightarrow IN$.

Uma vez definida a RP. Um disparo pode ser caracterizado pela equação 1 a seguir:

$$M_{k+1} = M_k + A^T \times u_k \quad k = 0, 1, 2 \dots n \quad (1)$$

Em que:

M_k é um vetor coluna cujo elemento da enésima linha corresponde a quantidade de marcações no enésimo estado, em determinado instante de tempo k . Ele será denominado como vetor de marcações.

M_{k+1} é o vetor de marcações resultante do disparo k ;

A^T é a matriz de incidência, que representa o conjunto de arcos da RP. O elemento da matriz de incidência a_{ij} tem o valor de -1 quando o estado i for uma entrada da transição j e tem o valor de 1 quando for uma saída. Caso não exista

relação entre o lugar e a transição, o elemento terá valor 0. Nessa matriz o número de colunas é igual à quantidade de transições na RP e o número de linhas é igual à quantidade de lugares na RP.

u_k é um vetor coluna cujo elemento da n -ésima linha corresponde a quantidade vezes que a transição n foi realizada no disparo. Será aqui denominado vetor de disparos.

Voltando ao exemplo da RP da Figura 6, podemos representar a matriz de incidência como:

$$A^T = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Note que a primeira coluna da matriz representa os arcos que ligam a transição T_0 , a segunda coluna representa os arcos da transição T_1 e assim por diante. Da mesma forma, há uma relação entre cada linha da matriz e um estado da RP. Assim, podemos indicar que a matriz de incidência é de tamanho $m \times n$, sendo que n é o número de estados e m o número de transições na RP.

Considerando um conjunto de marcações iniciais, uma no estado E_0 e duas no estado E_3 , pode-se construir o seguinte vetor de marcações:

$$M_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \\ 0 \end{bmatrix}$$

Estabelecendo ainda que no instante 0 há o disparo das transições T_0 e T_2 , tem-se o seguinte vetor de disparos:

$$u_0 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

De acordo com a equação 1, o resultado do disparo é o seguinte vetor de marcações:

$$M_1 = M_0 + A^T \times u_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Como se pode perceber, a equação resultou em um vetor de marcações válido cujos elementos representam a quantidade de marcações em cada estado. Se o vetor de marcações final possuir números negativos, o disparo resultou em uma exceção. Ou seja, não havia marcações suficientes nos estados de entrada para acionar as transições descritas no vetor de disparos.

A Figura 12 mostra a descrição gráfica do que aconteceu durante o exemplo de disparo descrito.

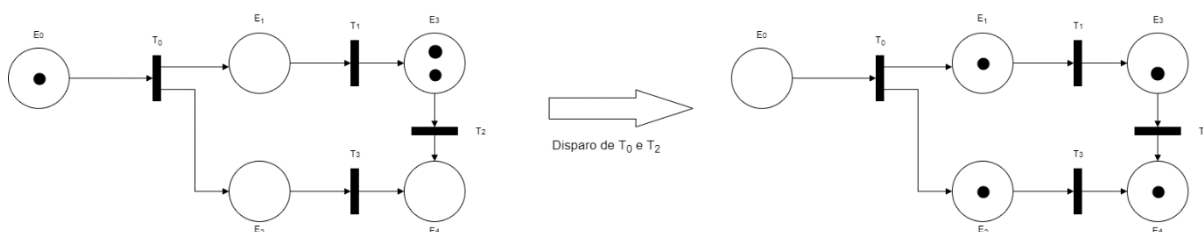


Figura 12 - Representação gráfica de um exemplo de disparo

2.3 PNRD

Redes de Petri Elementares inseridas em base de dados RFID é uma estrutura de dados formal a ser armazenada na etiqueta RFID de forma a integrá-la com sistema de controle (TAVARES; SARAIVA; 2010).

Ela pressupõe que o processo sofrido pela etiqueta possa ser modelado por meio de redes de Petri, de modo que o fluxo de trabalho da etiqueta é definido por uma matriz de incidência e o seu estado atual por um vetor de marcações.

Essas duas informações, juntamente com o ID da etiqueta e uma indicação de tempo, caracterizam a estrutura típica da PNRD. Sendo que a indicação de tempo é incluída de forma a possibilitar o sistema a realizar análises quantitativas de performance.

As antenas e leitores, por sua vez, realizam os disparos, armazenam e fornecem o vetor de disparos adequado a cada situação.

Por exemplo, pensando em uma fábrica que produza mancais. Consideremos que após a usinagem de uma peça, seja necessária a inspeção de dois tipos de requisitos: a cilindricidade e a rugosidade do seu furo. Caso o teste de cilindricidade falhar, a peça é enviada para a reciclagem. Se o teste de rugosidade falhar ela é colocada em uma fila para ser retrabalhada. Caso os dois testes estiverem de acordo com os padrões, a peça é armazenada em estoque. Esse processo pode ser representado pela RP da Figura 13.

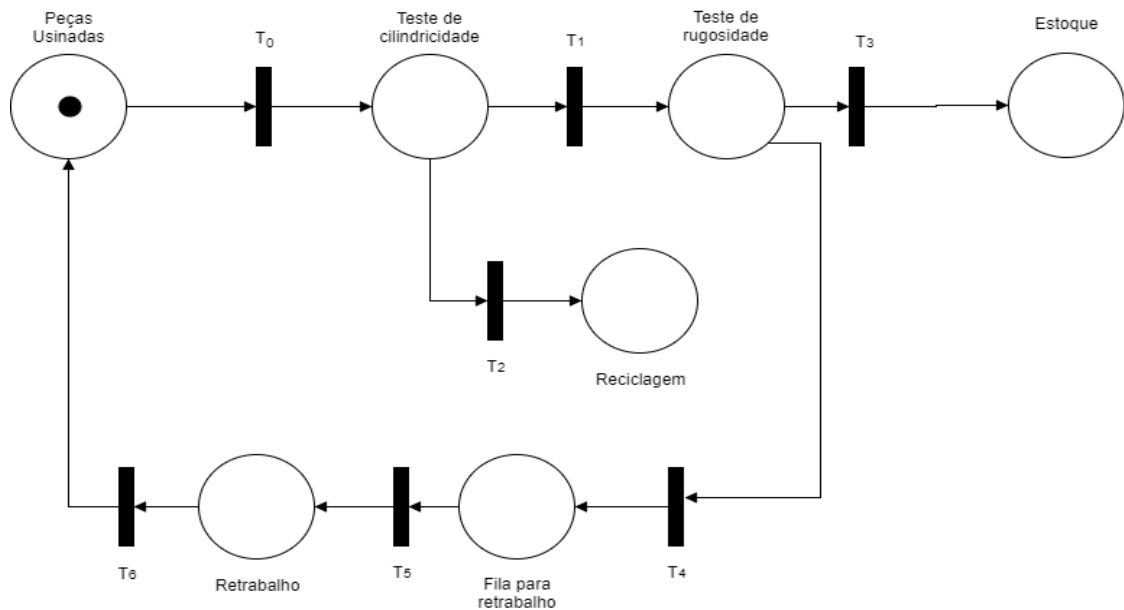


Figura 13 - Exemplo de Rede de Petri na verificação de mancais.

Considerando uma abordagem por PNRD, cada peça usinada é dotada de uma etiqueta RFID. Essa etiqueta contém a informação da matriz de incidência da RP e o atual vetor de marcações. No começo do processo, a marcação inicial é igual à mostrada na Figura 13.

Dessa forma, a máquina que verifica a cilindricidade do furo é dotada de uma antena RFID. O leitor ligado a essa antena, ao perceber uma nova peça, adquire as informações da mesma, isto é, sua marcação e sua matriz de incidência, realiza o disparo da transição T_0 , calcula o próximo estado e o grava na etiqueta. Em seguida, em concordância com o novo estado da etiqueta (teste de cilindricidade), envia um comando para começar o teste. Dependendo do resultado do teste, o sistema envia uma mensagem ao aplicativo RFID que dispara a transição T_1 ou a T_2 e atualiza o novo estado da etiqueta.

Perceba que a passagem da etiqueta em uma determinada antena não define a transição disparada. É necessário um conjunto de informações, que podem ou não incluir: dados externos, marcação inicial, a identificação da etiqueta e a identificação da antena, para identificar uma transição disparada.

De forma semelhante, uma transição não está necessariamente ligada a uma única antena. No caso apresentado, poderia haver mais de uma máquina verificadora de medidas. No entanto, vale a pena constar que independentemente do número de máquinas, o fluxo de trabalho não muda. Isso permite que o sistema seja capaz de crescer sem sofrer uma grande modificação no seu funcionamento. Nesse caso, a abordagem PNRD torna o sistema de controle da fábrica facilmente escalável.

Continuando o exemplo retratado, após do disparo da transição T_1 ou T_2 a peça é enviada para a reciclagem ou para o teste de rugosidade. Ambos os lugares são dotados de antenas RFID que leem a etiqueta e realizam um novo disparo. Se o resultado desse disparo for uma exceção, percebe-se que a peça não foi enviada para o lugar correto. De igual forma, pode-se detectar peças não-conformes ou não-testadas mandadas para o estoque. Essa análise de exceções permite que a PNRD detecte e trate erros automaticamente em tempo de execução.

Por fim, uma outra característica da PNRD é a distribuição dos dados do processo nas etiquetas, tornando desnecessária a consulta em um banco de dados central, agilizando o fluxo de dados e diminuindo custos de implementação de rede locais.

2.3 PNRD INVERTIDA

A PNRD invertida é uma variação da PNRD convencional. Nessa abordagem, as etiquetas contêm, dentro da sua memória, o vetor de disparos no lugar do vetor de marcações e a matriz de incidência.

Desenvolvida por Fonseca e Tavares (2017) no contexto de busca e salvamento de pessoas em trilhas de caminhada, essa abordagem busca viabilizar a utilização do conceito da PNRD em aplicações cuja a instalação de antenas RFID nos pontos de transição seja demasiadamente difícil.

A PNRD invertida utiliza antenas RFID móveis para ler etiquetas posicionadas em pontos de transição fixos. Assim, o leitor RFID é o responsável pela armazenagem

do estado do sistema, enquanto a etiqueta determina a sua transição. Uma exceção, nesse contexto, indica que o agente portador da antena RFID está seguindo por um caminho errado, ou realizando uma ação que ele não deveria fazer.

Essa nova abordagem também trouxe uma necessidade de armazenar um histórico de visitas de leitores na etiqueta. Na aplicação de busca e salvamento, por exemplo, seria essencial visualizar quais pessoas passaram por determinada rota.

2.4 ARDUINO

O Arduino® é uma plataforma eletrônica aberta de fácil utilização (ARDUINO, 2017a). Ela possui as características de um sistema embarcado multiuso, fornecendo suporte para a interação com diversos dispositivos eletrônicos, como LEDs, sensores, placas de comunicação, *drivers* de motores e leitores RFID.

Desprovido de um sistema operacional, os sistemas Arduino® podem ser reprogramados via cabo USB. Tal característica os torna extremamente populares na prototipagem de sistemas embarcados.

Algumas vantagens na utilização dessa plataforma são: uma grande comunidade de desenvolvedores, a variedade de sistemas compatíveis, a facilitação no desenvolvimento do *software* e de *hardware*, a disponibilidade de bibliotecas e o suporte para programação orientada a objeto.

Existem diferentes modelos de Arduinos comerciais, que variam em tamanho, capacidade de processamento, memória e número de portas, conforme ilustrado na Figura 14.

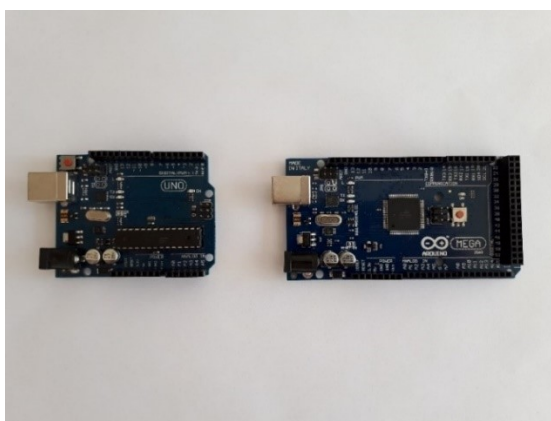


Figura 14 - Arduinos utilizados no desenvolvimento da biblioteca, Arduino® Uno à esquerda e a Arduino® Mega à direita.

METODOLOGIA E DESENVOLVIMENTO

Primeiramente, definiram-se e analisaram-se os requisitos da biblioteca. Depois, realizou-se um projeto de *software*, estabelecendo a sua arquitetura básica. Durante a realização do projeto, foi constatada a possibilidade de divisão da biblioteca em três módulos. Cada um dos módulos foi implementado e testado individualmente, em uma sequência adequada. No final, foram feitos testes integrando os diferentes módulos.

Existem várias metodologias utilizadas para desenvolvimento de *software*. A metodologia utilizada nesse projeto foi baseada no modelo incremental definido por Pressman (2011). Dessa forma, o software foi feito de modo escalonado. Sendo que cada módulo foi desenvolvido em uma sequência linear de planejamento, modelagem, construção e teste.

3.1 DEFINIÇÃO DE REQUISITOS

Partindo dos objetivos especificados no item 1.1 e analisando as aplicações da PNRD e PNRD invertidas descritas por Tavares e Saraiva (2010) e Fonseca e Tavares (2017). Foi definida a seguinte lista de requisitos:

1. É necessário a criação de sub-rotinas de realização e verificação de disparos em redes de Petri;
2. A biblioteca deve ser capaz de ler e gravar na etiqueta as estruturas de dados típicas da PNRD e da PNRD invertida.
3. A configuração dos dados concernentes à RP de cada aplicação deve ser feita de forma intuitiva;
4. A implementação de outros tipos de leitores deve ser facilitada, de modo a aproveitar o código desenvolvido para aplicações que utilizem outros modelos de leitores RFID;
5. É preciso haver suporte para a gravação do histórico da etiqueta, contendo informação de identificação do leitor e do instante de tempo inicial de cada estado;
6. É necessário haver um suporte para redes de Petri não-autônomas, em que um disparo só pode ocorrer com a satisfação de determinadas condições.

Esses requisitos apontam três funcionalidades básicas da biblioteca a ser desenvolvida: a base de cálculo das redes de Petri, a abordagem PNRD ou PNRD invertida e a troca de informação com as etiquetas.

3.2 PROJETO DE SOFTWARE

O projeto desenvolvido leva em conta uma abordagem orientada a objetos. Essa abordagem melhora a clareza da biblioteca e permite o reaproveitamento do código em diferentes aplicações, que vai de encontro aos objetivos iniciais do trabalho e dos requisitos 3 e 4 descritos anteriormente.

Alguns desenvolvedores afirmam que a abordagem por programação estruturada é preferível à orientação de objetos no caso de sistemas embarcados. Essa afirmação se baseia em dois fatores:

- A abstração em objetos gera desvios de código que tornam a execução do código mais lenta em comparação com a execução sequencial;
- O programador pode perder a noção da utilização da memória, tendo em vista que cada objeto possui diferentes estruturas no seu interior, inclusive outros objetos.

Ambos esses fatores são relevantes em sistemas computacionais de processamento e memória reduzidos, como é o caso de sistemas embarcados.

Portanto, como forma de minimizar esses efeitos indesejáveis, evitou-se ao máximo a utilização de objetos dentro das rotinas da biblioteca. Em seu lugar, foram utilizados tipos e estruturas de dados clássicos da linguagem C. Os objetos criados são utilizados como forma de interface com o desenvolvedor da aplicação, facilitando a sua compreensão e diminuindo o tempo de desenvolvimento.

A biblioteca desenvolvida foi dividida em três módulos: a rede de Petri, a PNRD e o leitor de etiquetas conforme mostrado na Figura 15.

O módulo da rede de Petri é o responsável pela base matemática da biblioteca. Ele permite a definição da matriz de incidência, do vetor de marcações e do vetor de disparos, bem como o cálculo de disparos e a verificação de exceções. Além disso, fornece suporte para o controle de disparos através de condições externas à RP, conforme relatado no requisito número 6.

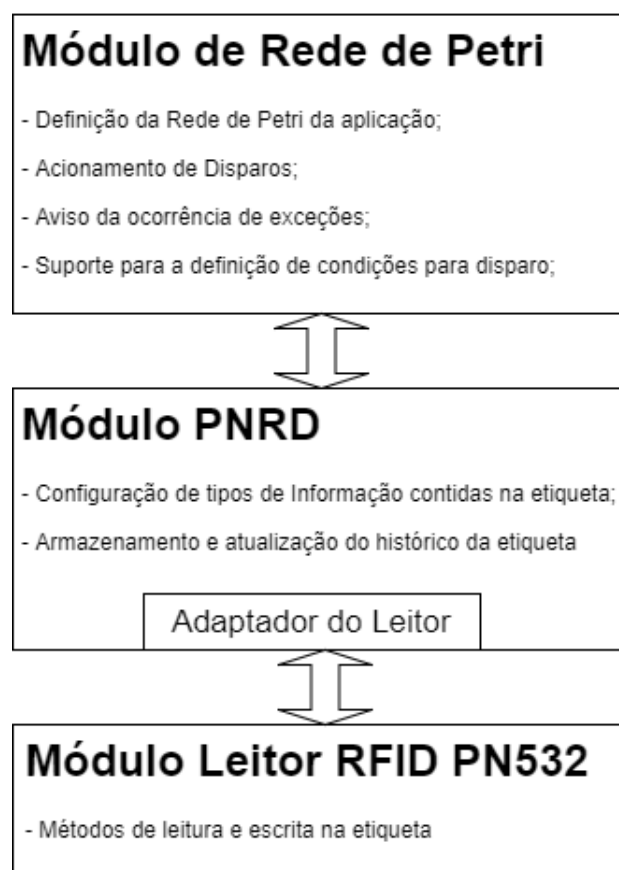


Figura 15 - Arquitetura dos módulos da biblioteca

O módulo da PNRD é o responsável pela integração entre o sistema RFID e o módulo da Rede de Petri, permitindo o desenvolvimento de aplicações tanto da PNRD clássica como a da PNRD invertida.

A principal diferença entre a aplicação da PNRD clássica e a abordagem da PNRD invertida é o conjunto de estruturas a serem salvas nas etiquetas. A solução encontrada foi a criação de rotinas de configuração, que determinam quais os dados são guardados nas etiquetas e quais são de responsabilidade do controlador. Como resultado, é possível criar aplicações que misturam as estruturas da PNRD e PNRD invertida ou que incluem outros tipos de dados na etiqueta, como o histórico da etiqueta.

Para possibilitar a compatibilidade entre esse módulo e múltiplos tipos de leitores RFID, foi criada uma classe adaptadora chamada *Reader* ou leitor. Esse objeto contém dois métodos virtuais de leitura e escrita de etiqueta. Para a utilização de novos tipos de leitores, é necessário somente reescrever esses dois métodos em uma classe filha da classe *Reader*.

O terceiro módulo é o do leitor PN532. Ele contém os métodos de leitura e escrita de etiquetas utilizando a antena de modelo PN532.

Note que caso o usuário precise de trocar o modelo de antena, o único módulo a ser substituído é o terceiro. O código dos outros dois módulos pode ser reaproveitado.

3.2.1 – Considerações sobre as estruturas de dados utilizadas

Um bom projeto de software passa pela definição das estruturas de dados a serem utilizadas. A escolha dessas estruturas influencia na quantidade de memória alocada para a aplicação, no seu tempo de execução e no seu tempo de desenvolvimento. Além disso, essas estruturas podem restringir ou abranger a utilidade da biblioteca.

Para a definição dessas estruturas foram avaliados os seguintes parâmetros: a memória alocada, as suas possíveis restrições à aplicação e o tempo de execução das principais rotinas. Além disso, foi considerado o problema de fragmentação da memória do Arduino®.

A fragmentação ocorre ao se alocar e desalocar memória múltiplas vezes durante a execução do programa. Ao longo do tempo, a memória se torna dividida em blocos não sequenciais, conforme ilustrado na Figura 16. Esse processo acontece até o ponto de ocorrer uma falha devido ao fato de nenhum desses blocos ter memória suficiente para um pedido de alocação, apesar de existir uma quantidade total de memória disponível suficiente.

Em sistemas computacionais, esse problema é normalmente tratado pelo sistema operacional (SO). No entanto, o Arduino® é um sistema embarcado desprovido de SO. Assim, podem ser aplicadas duas diferentes soluções: a alocação previa de memória a ser utilizada durante todo o programa durante a configuração do sistema, ou a implementação de rotinas que permitam a desfragmentação da memória. Como rotinas de desfragmentação possuem uma complexidade relativamente grande e fogem do escopo dos objetivos estabelecidos, optou-se pela primeira opção.

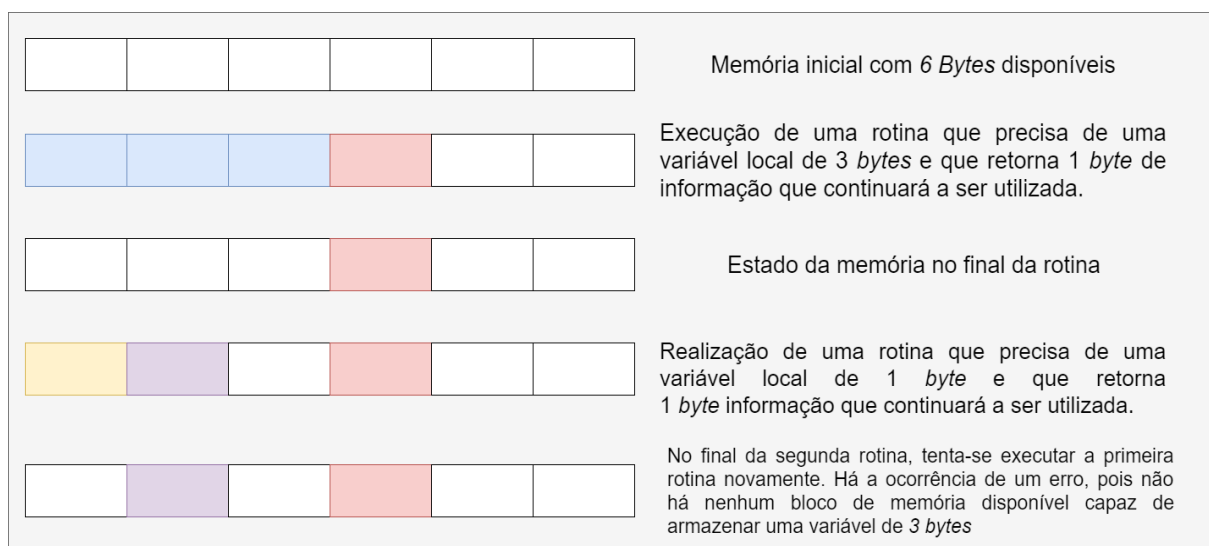


Figura 16- Exemplo de fragmentação de memória.

Dessa forma, todas as estruturas criadas alocam um espaço de memória na sua criação sem que tal espaço sofra modificações ao longo da execução do programa. Evitou-se, portanto, a utilização de estruturas dinâmicas, como árvores ou listas encadeadas, que podem causar a fragmentação da memória.

Desse modo, ao declarar um objeto rede de Petri ou PNRD, é necessário definir o seu número de estados e de transições, de modo a alocar adequadamente a quantidade de memória para cada uma de suas estruturas.

3.2.2 – Vetor de marcações e vetor de disparos

Os vetores de marcação e disparos foram definidos como dois *arrays* do tipo inteiro de 16 bits, sem sinal.

No caso do vetor de marcações, o elemento com índice n no *array* indica o número de fichas no n ésimo estado.

No caso do vetor de disparos, o elemento com índice n no *array* indica a quantidade de vezes que a n ésima transição foi disparada.

Dessa forma, o tamanho do *array* das marcações é igual ao número de lugares e o tamanho do *array* de disparos é equivalente ao número de transições da rede de Petri utilizada.

Tanto a quantidade de marcações em um lugar quanto a quantidade de transições em um disparo são definidos como números inteiros não negativos nas redes de Petri ordinárias. Por isso, a adoção de um tipo inteiro sem sinal.

O tamanho de 16 bits foi definido por dois motivos: o padrão de processador dos Arduinos é 16 bits, o que significa que a performance de operações com números de 16 bits é otimizada. Além disso, considerou-se muito improvável que uma aplicação PNRD necessite de uma quantidade de marcações e transições superiores a capacidade de representação de 16 bits, que é de 65.536 valores distintos.

3.2.3 – Estados e transições

Conforme comentado, inicialmente há a necessidade de definição da quantidade de lugares e transições da RP para alocar a memória adequada. Para a representar a quantidade de estados e transições foram utilizados inteiros de 8 bits, sem sinal.

Optou-se por utilizar inteiros de 8 bits e não 16 bits, porque a armazenagem dos *arrays* correspondentes ao vetor de marcações e ao vetor de disparos com 65.535 elementos seria muito dispendioso em termos de memória.

Para se ter uma ideia quantitativa, um vetor de marcações com esse número de lugares ocuparia cerca de 131 KB de memória. Valor que ultrapassa a capacidade de armazenamento de qualquer etiqueta RFID disponível no mercado atualmente e a capacidade de armazenamento da memória SRAM dos Arduino® Uno e Arduino® Mega, que segundo site oficial da marca (2017) são de respectivamente 2 KB e 8 KB.

Dessa forma, a biblioteca possui uma limitação de um número máximo de 255 lugares e 255 transições. Pois apesar do número 0 indicar um lugar ou transição válidos, o número 255 é indicativo de um lugar ou transição nulos, e é utilizado como terminador de listas. No caso de maior quantidade de estados e transições possível, o *array* de disparos e o *array* de marcações ocupariam, cada um, um máximo de 508 Bytes.

3.2.4 – Representação dos arcos da rede de Petri

No item 2.2.3 introduziu-se uma forma de representação dos arcos em uma RP chamada matriz de incidência. No entanto, o fato de redes de Petri com grande quantidade de transições e estados usualmente apresentarem matrizes de incidência cuja a maioria dos elementos é zero, levou ao questionamento da possibilidade de existência de estruturas de dados mais adequadas à armazenagem dessa informação.

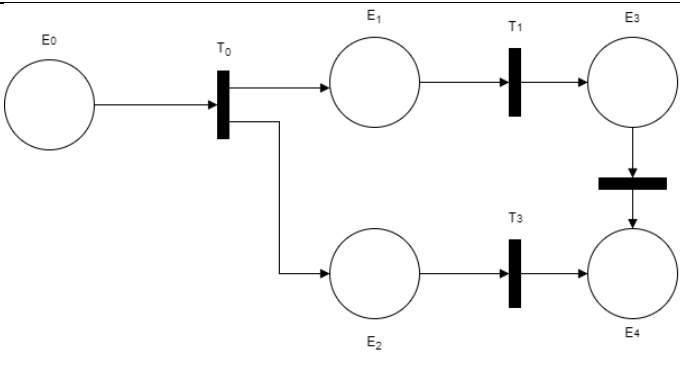
Dessa forma, abriu-se a possibilidade da representação dos arcos por três tipos diferentes de estruturas. Essas possibilidades foram baseadas em estruturas clássicas usadas na representação de grafos, uma vez que segundo Murata (1989, p. 542, tradução nossa) as redes de Petri são “um tipo de grafo orientado”.

As três estruturas são as seguintes:

- Matriz de Incidência, conforme especificada no item 2.2.3;
- Matrizes de Adjacência. Nesse caso, existem duas matrizes diferentes: as que definem os arcos de entrada e outra que define os arcos de saída. Se existir um arco apontado do estado i para a transição j . A matriz de entradas apresenta valor "1" no seu elemento a_{ij} , e o valor "0" caso contrário. Se existir um arco apontado da transição j para o estado i . A matriz de saídas apresenta valor "1" no seu elemento a_{ij} , e o valor "0" caso contrário.
- Listas de Adjacência. Nessa estrutura, cada transição possui duas listas distintas. A primeira é a lista entrada, que contém a numeração dos estados que são entrada da transição. A segunda é a lista de saída, que contém a numeração dos estados que são saídas daquela transição.

Na Tabela 4, observa-se como o exemplo inicial da seção 2.2 pode ser representado utilizando essas estruturas. Note que, devido a impossibilidade de utilização de listas dinâmicas é necessário alocar uma certa quantidade de elementos para a lista de cada transição. Essa quantidade sempre deve ser maior ou igual ao número máximo de entradas ou de saídas de uma transição.

Tabela 4 - Estruturas de representativas de arcos em Redes de Petri

Representação gráfica																										
Matriz de Incidência	$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$																									
Matrizes de Adjacência	Entradas: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$	Saídas: $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$																								
Listas de Adjacência	Lista de Entradas: <table border="1" data-bbox="703 1227 999 1451"><tr><td>T₀</td><td>E₀</td><td><i>null</i></td></tr><tr><td>T₁</td><td>E₁</td><td><i>null</i></td></tr><tr><td>T₂</td><td>E₂</td><td><i>null</i></td></tr><tr><td>T₃</td><td>E₃</td><td><i>null</i></td></tr></table>	T ₀	E ₀	<i>null</i>	T ₁	E ₁	<i>null</i>	T ₂	E ₂	<i>null</i>	T ₃	E ₃	<i>null</i>	Lista de Saídas: <table border="1" data-bbox="1098 1227 1377 1451"><tr><td>T₀</td><td>E₁</td><td>E₃</td></tr><tr><td>T₁</td><td>E₂</td><td><i>null</i></td></tr><tr><td>T₂</td><td>E₄</td><td><i>null</i></td></tr><tr><td>T₃</td><td>E₄</td><td><i>null</i></td></tr></table>	T ₀	E ₁	E ₃	T ₁	E ₂	<i>null</i>	T ₂	E ₄	<i>null</i>	T ₃	E ₄	<i>null</i>
T ₀	E ₀	<i>null</i>																								
T ₁	E ₁	<i>null</i>																								
T ₂	E ₂	<i>null</i>																								
T ₃	E ₃	<i>null</i>																								
T ₀	E ₁	E ₃																								
T ₁	E ₂	<i>null</i>																								
T ₂	E ₄	<i>null</i>																								
T ₃	E ₄	<i>null</i>																								

Essas três estruturas foram comparadas usando os seguintes parâmetros:

1. Tamanho da memória necessária;
2. A complexidade das principais operações relacionadas à rede de Petri;
3. Outras características;

Basicamente, pode-se dizer que as listas de adjacência facilitam a verificação e cálculo de disparos. Enquanto as estruturas matriciais facilitam a definição de arcos e verificação de relação entre um estado e uma transição.

Considerando o disparo de uma transição, nas estruturas matriciais, é necessário verificar valor por valor de toda uma coluna, testando se há ou não relação

de entrada ou de saída. Nas listas de adjacência, no entanto, todos esses testes não são necessários, pois estão na lista somente os estados relacionados à transição. Consequentemente, o cálculo do disparo se torna mais rápido.

Entretanto, ao analisar o algoritmo necessário para definir um arco nessas estruturas, percebe-se que nas listas de adjacência é necessário percorrer a lista dos valores de estados, até chegar em um cujo valor seja nulo. Enquanto que nas outras estruturas o arco pode ser definido apenas modificando o valor em um endereço específico.

Considerando que a definição da rede de Petri e dos arcos é feita somente em fase de configuração, a velocidade de execução dessa rotina não é muito importante. No entanto, os disparos são realizados múltiplas vezes e em tempo de execução, o que os fazem necessitar de um algoritmo rápido e eficaz. Dessa forma, a lista de adjacência é a estrutura mais indicada pensando no quesito de velocidade.

No quesito de memória, não é possível afirmar a superioridade da representação por matriz de incidência ou por listas de adjacência. Cada elemento da matriz de adjacência ocupa, no mínimo, 2 bits, pois pode apresentar 3 tipos diferentes de valor (1, 0, -1). Portanto, quantidade de memória, em Bytes, da matriz de incidência pode ser definida pela equação 2:

$$M_{MI} = \left\lceil \frac{t \times e}{4} \right\rceil \quad (2)$$

Sendo t o número de transições e e o número de estados.

Como cada elemento da lista de adjacência ocupa um Byte de memória, pois contém o identificador de um lugar da RP. A quantidade de memória da lista de incidência com um número máximo de lugares de saídas max_{in} e máximo de lugares de entrada max_{out} pode ser definida pela equação 3:

$$M_{LA} = t \times (max_{in} + max_{out}) \quad (3)$$

Considerando que $\left\lceil \frac{t \times e}{4} \right\rceil \cong \frac{t \times e}{4}$, pode-se afirmar que a lista de incidência ocupa menos memória quando:

$$M_{LA} < M_{MI}$$

$$t \times (max_{in} + max_{out}) < \frac{t \times e}{4}$$

Sendo um número positivo não-nulo, tira-se que:

$$e > 4 \times (max_{in} + max_{out})$$

Ou seja, quando o número de lugares da RP for maior que quatro vezes a soma do número máximo de entradas e de saídas, a lista de adjacência ocupará menos memória.

Se pensarmos de ponto de vista prático, a maioria dos modelos de Redes de Petri possuem transições com pouca quantidade de entradas e de saídas. Além disso, o problema de armazenagem de dados só ocorre quando se há um número grande de estados e transições. Assim, a lista de adjacência, em termos práticos, é uma representação mais adequada.

Outra vantagem da estrutura de lista de adjacências é a possibilidade de modelizar transições com *self-loop*, em que um estado é uma entrada e saída de uma transição ao mesmo tempo. No entanto, uma desvantagem é que essa estrutura tende a limitar a quantidade máxima de lugares ligados a cada transição.

A estrutura de matrizes de adjacência também é capaz de representar transições impuras e não possui limitações quanto a quantidade de arcos de uma transição. No entanto, ela perde nos quesitos de memória e velocidade em relação às outras estruturas.

O Apêndice A mostra o resumo das comparações entre estruturas. Devido ao melhor desempenho em casos práticos de PNRD, a estrutura de lista de adjacências foi a escolhida para representar os arcos de uma PN.

No entanto, devido ao uso difundido das matrizes de incidência na literatura de redes de Petri, foi estabelecida a necessidade de promover uma interface capaz de converter essas duas estruturas, indo de acordo com o requisito número 3.

3.2.5 – Condições

Conforme estabelecido no requisito número 6, é necessário oferecer suporte para PN não-autônomas utilizando o conceito de condições. As condições são uma ferramenta para o usuário lidar com casos de concorrência na RP.

Inicialmente, pensou-se em estabelecer, para cada transição, um conjunto de condições, que poderiam ser mensagens, intervalos de tempo ou entradas booleanas. No entanto, percebeu-se uma enorme complexidade relativa à configuração de tais condições, uma vez que o usuário necessitava definir o tipo da condição, a transição na qual ela estaria associada e como ela estaria relacionada ao resto do conjunto de condições daquela transição. Tal complexidade não é condizente com o principal objetivo do trabalho, que é desenvolver uma interface para facilitar o desenvolvimento de aplicações PNRD.

Dessa forma, essa lógica foi simplificada de modo que o módulo de rede de Petri armazena unicamente uma condição booleana para cada transição. Além de diminuir a quantidade de memória utilizada, essa abordagem faz com que a lógica por trás da habilitação do disparo da transição esteja completamente a caráter da aplicação, permitindo variações dos tipos de condição inicialmente pensados.

3.2.6 – Histórico da etiqueta

Modelizou-se uma entrada do histórico da etiqueta como sendo uma estrutura com as seguintes informações:

- Identificador do leitor, como um inteiro sem sinal de 16 bits;
- Número do lugar, como um inteiro sem sinal de 8 bits;
- Quantidade de fichas naquele lugar, como um inteiro sem sinal de 16 bits;
- Um indicativo do tempo do sistema como um inteiro sem sinal de 32 bits;

No total, cada entrada necessita de 9 bytes de informação. Sendo que o número máximo de entradas no histórico da etiqueta deve ser informado anteriormente, para a devida alocação de memória.

Assim, o histórico foi modelizado como uma lista não-sequencial e estática de entradas de histórico, sendo que, uma vez que todas suas posições estão ocupadas ela substitui automaticamente a primeira entrada inserida.

3.3 IMPLEMENTAÇÃO

A escrita do software foi realizada utilizando *notepad++* e o *Visual Studio 2017*. Sendo que testes funcionais foram feitos a partir da IDE do Arduino® versão 1.8. Para controle de versão, utilizou-se o *TortoiseSVN*.

Inicialmente foi utilizado o Arduino® Uno como plataforma de desenvolvimento e posteriormente o Arduino® Mega devido a necessidade múltiplas portas seriais e de uma maior memória.

Foram utilizados também o leitor RFID NFC PN532 ilustrado na Figura 12 e as etiquetas RFID Mifare de 1 KB mostradas na Figura 2. Segundo o manual de utilização feito pela *Electrohouse* (Electrohouse, 2017), esse leitor utiliza a frequência de 13,56 MHz, sendo caracterizado como HF. Seu funcionamento se baseia no padrão de comunicação por campo de proximidade (NFC), sendo compatível com aparelhos celulares. Esse módulo possui bibliotecas compatíveis com a plataforma Arduino®, sendo que foi utilizada a biblioteca disponível em <<https://github.com/elechouse/PN532>> (acessado em 11/12/2017) para ajudar no desenvolvimento das rotinas de leitura e escrita de etiquetas.

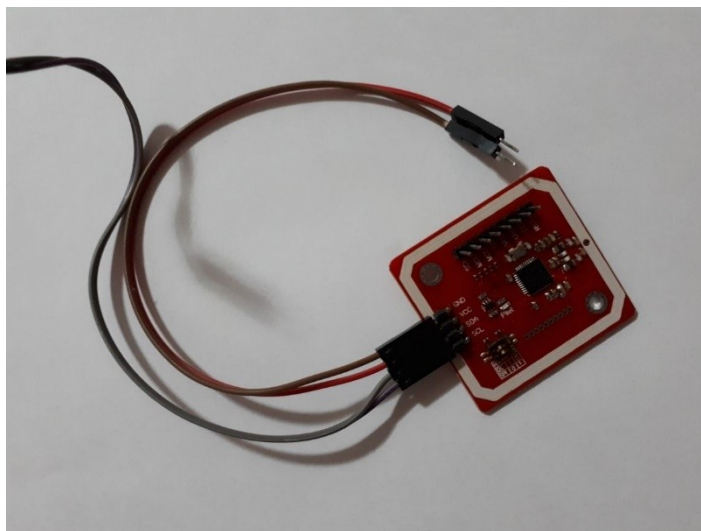


Figura 17 - Leitor RFID NFC PN532

Os módulos foram desenvolvidos na seguinte ordem cronológica: rede de Petri, PNRD e leitor PN532. Cada um dos módulos apresenta um arquivo de cabeçalho e de código fonte separados. Assim, caso a aplicação necessite apenas do modelo de redes de Petri, ou utilize outro leitor que não seja o PN532, seja possível importar apenas as rotinas necessárias à aplicação.

3.3.1 – Módulo Redes de Petri

O módulo de redes de Petri foi construído tendo como base a classe *PetriNet*. Com essa classe cria-se um modelo matemático de uma RP.

Durante a criação do objeto dessa classe são definidas as seguintes informações: a quantidade de transições e de lugares, número máximo de entradas e de saídas por transição e a presença de testes de condições, de modo que a quantidade de memória adequada seja alocada na sua iniciação.

Em seguida, pode-se definir o vetor de marcações presentes na RP, os arcos entre estados e transições e o vetor de disparos.

Por fim, pode-se realizar ou verificar um disparo. O retorno da realização ou verificação do disparo é chamado de *FireError*, que pode ser:

- *NO_ERROR* – No contexto da verificação de disparo, significa que o disparo descrito não gera erros. No contexto de realização de disparos, significa que o disparo foi feito e o vetor de marcações foi atualizado;
- *CONDITIONS_ARE_NOT_APPLIED* – Indica que as condições associadas às transições do disparo não estão satisfeitas;
- *PRODUCE_EXCEPTION* – Indica que o disparo resulta em exceção, ou seja, não existem marcações suficientes nos lugares de entrada para efetuar as transições;
- *NOT_KNOWN* – Quando o programa segue por um fluxo inesperado, usualmente é indício de um *bug*;

Dessa forma, pode-se dividir os métodos públicos da classe *PetriNet* em cinco categorias:

- Construtor: são métodos que criam um objeto *PetriNet*, eles definem a quantidade de memória alocada para as suas estruturas internas;
- Configuração: são métodos que definem as informações básicas da rede de Petri, como o seu vetor de marcação, o seu vetor de disparos, os arcos entre transições e estados, e se a condição de determinada transição está satisfeita ou não.
- Disparo: são métodos que verificam ou realizam disparos;

- Desenvolvimento e *Debug*: são métodos que facilitam o desenvolvimento da aplicação, mostrando, de uma maneira intuitiva, as informações da RP;

- Acesso: são métodos que permitem o acesso direto nas estruturas de dados internas do objeto rede de Petri. Devem ser usados com cautela em operações em que os métodos anteriores não satisfazem;

A Tabela 5 apresenta a listagem de todos os métodos públicos desenvolvidos para a classe *PetriNet*, divididos nessas categorias.

Tabela 5 - Lista de métodos públicos da classe *PetriNet*.

Categoria	Métodos
Construtor	▪ PetriNet(uint8_t num_places, uint8_t num_transitions, uint8_t num_max_of_inputs, uint8_t num_max_of_outputs, bool hasConditions); ▪ PetriNet(uint8_t num_places, uint8_t num_transitions, uint8_t num_max_of_inputs, uint8_t num_max_of_outputs); ▪ PetriNet(uint8_t num_places, uint8_t num_transitions, bool hasConditions); ▪ PetriNet(uint8_t num_places, uint8_t num_transitions);
Configuração	▪ void setFireVector(uint16_t* vector); ▪ void setTokenVector(uint16_t* vector); ▪ bool setIncidenceMatrix(int8_t* matrix); ▪ bool addInput(uint8_t place, uint8_t transition); ▪ bool deleteInput(uint8_t place, uint8_t transition); ▪ bool addOutput(uint8_t place, uint8_t transition); ▪ bool deleteOutput(uint8_t place, uint8_t transition); ▪ bool setInputs(uint8_t transition, uint8_t* inputs, uint8_t inputsCount); ▪ bool setOutputs(uint8_t transition, uint8_t* outputs, uint8_t outputsCount); ▪ void conditionIsSatisfied(uint8_t transition); ▪ void conditionIsNotSatisfied(uint8_t transition); ▪ void conditionUpdate(uint8_t transition, bool isSatisfied);
Disparo	▪ FireError fire(); ▪ FireError fire(uint8_t transition); ▪ FireError isTriggerable(uint8_t transition); ▪ FireError isTriggerable();

Desenvolvimento e <i>Debug</i>	▪ uint8_t getNumberOfPlaces(); ▪ uint8_t getNumberOfTransitions(); ▪ uint8_t getNumberMaxOfInputs(); ▪ uint8_t getNumberMaxOfOutputs(); ▪ void getFireVector(uint16_t* vector); ▪ void printFireVector(); ▪ void getTokenVector(uint16_t* vector); ▪ void printTokenVector(); ▪ bool setIncidenceMatrix(int8_t* matrix); ▪ void getIncidenceMatrix(int8_t* matrix); ▪ int8_t getMatrixElement(int8_t place, int8_t transition); ▪ void printIncidenceMatrix(); ▪ bool isConditionSatisfied(uint8_t transition); ▪ void printConditions();
Acesso	▪ uint16_t* getFireVectorPointer(); ▪ uint16_t* getTokenVectorPointer(); ▪ uint8_t* getInputsPointer(uint8_t transition); ▪ uint8_t* getOutputsPointer(uint8_t transition); ▪ uint8_t* getAdjacencyListPointer(); ▪ uint8_t* getConditionsPointer();

3.3.2 – Módulo PNRD

O módulo PNRD foi construído tendo como base a classe *Pnrd*. Essa classe é filha da classe *PetriNet* e herda os seus métodos públicos. Na Tabela 6, tem-se os métodos públicos exclusivos da classe *Pnrd*.

Além disso, ela possui métodos que auxiliam a definir quais os tipos de dados serão gravados na etiqueta. Os tipos de dados que podem ser gravados na *Pnrd* são definidos pela classe enumerada *PetriNetInformation* e podem ser:

- TOKEN_VECTOR – Vetor de marcações;
- ADJACENCY_LIST – Lista de Adjacências;
- FIRE_VECTOR – Vetor de disparos;
- CONDITIONS – Condições;
- TAG_HISTORY – Histórico da etiqueta;

- OTHER – Outro;

Foram definidos dois tipos de configuração para a *Pnrd*, descritos como *PnrdPolicy*:

- *DEFAULT_SETUP* – Nessa configuração, a *PNRD* somente salva e lê os dados previamente definidos como pertencentes às etiquetas;
- *TAG_SETUP* – Nessa configuração, os dados pertencentes à etiqueta serão definidos no momento de leitura de uma etiqueta, de modo que todas as estruturas presentes na etiqueta serão definidas como pertencentes à etiqueta;

Basicamente, a configuração *DEFAULT_SETUP* precisa ser utilizada em etiquetas inicialmente sem informação, enquanto a *TAG_SETUP* é necessária no caso de aplicações mistas de PNRD e PNRD invertida. Nos demais casos, a configuração dependerá da preferência do desenvolvedor da aplicação.

É nesse módulo também que há a implementação do histórico da etiqueta. O histórico é composto por um *array* não-ordenado de estruturas do tipo *TagHistoryEntry* e um indicador que aponta para o índice da última entrada a ser inserida.

O tamanho desse *array* é definido por uma constante macro chamada *MAX_NUM_OF_TAG_HISTORY_ENTRIES*, cujo valor padrão é de 10. Para modificar esse tamanho, basta definir, na aplicação, essa macro como sendo a quantidade desejada.

O histórico da etiqueta, caso presente, é atualizado automaticamente após a realização de um disparo. Quando o *array* do histórico estiver cheio, ele sobrescreverá a nova entrada no lugar da entrada mais antiga.

Para escrever ou obter os dados em uma etiqueta são utilizados os métodos *getData()* e *setData()*, que retornam respectivamente um *ReadError* e um *WriteError*. Que podem ser:

- *NO_ERROR* – A operação de escrita ou leitura foi efetuada com sucesso;
- *TAG_NOT_PRESENT* – Não existe nenhuma etiqueta RFID compatível com o leitor na sua proximidade;

- *INFORMATION_NOT_SAVED* – Exclusivo no caso de escrita. Indica que ocorreu um erro no ato de gravação dos dados na etiqueta;
- *INFORMATION_NOT_PRESENT* – Exclusivo no caso de leitura e da configuração *TAG_SETUP*. Indica que nem todas as informações configuradas com sendo das etiquetas estão presentes na etiqueta lida.
- *DATA_SIZE_NOT_COMPATIBLE* – Exclusivo no caso de leitura. Significa que o objeto *Pnrd* criado não alocou uma quantidade de memória adequada para representar as informações contidas na etiqueta;
- *NOT_ENOUGH_SPACE* – Exclusivo no caso de gravação. Indica que a etiqueta não tem memória interna suficiente para gravar os dados;
- *NOT_AUTORIZED* – significa que o ato de escrita ou leitura não foi autorizado pela etiqueta;
- *VERSION_NOT_SUPPORTED* – quer dizer que o leitor não suporta a versão de dados presentes na etiqueta;
- *ERROR_UNKNOWN* – significa a ocorrência de um erro genérico não especificado.

Tabela 6 - Métodos Públicos exclusivos da classe *Pnrd*

Categoria	Métodos
Construtor	▪ <i>Pnrd</i>(Reader* readerPointer, uint8_t num_places, uint8_t num_transitions, uint8_t num_max_of_inputs, uint8_t num_max_of_outputs, bool hasConditions, bool hasTagHistory); ▪ <i>Pnrd</i>(Reader* readerPointer, uint8_t num_places, uint8_t num_transitions, bool hasConditions, bool hasTagHistory);
Configuração	▪ void setAsTagInformation(PetriNetInformation info); ▪ void setAsDeviceInformation(PetriNetInformation info); ▪ ReadError getData(); ▪ WriteError saveData(); ▪ void setTagId(uint32_t tagId); ▪ void setDeviceId(uint32_t deviceId); ▪ uint32_t getDeviceId(); ▪ bool setTagHistory(TagHistoryEntry* vector, uint8_t numberOfEntries); ▪ void addTagHistoryEntry(TagHistoryEntry entry);

Desenvolvimento e <i>Debug</i>	▪ <code>bool isTagInformation(PetriNetInformation info);</code> ▪ <code>uint8_t getDataInTag();</code> ▪ <code>uint32_t getTagId();</code> ▪ <code>void setDeviceId(uint32_t deviceId);</code> ▪ <code>uint32_t getDeviceId();</code> ▪ <code>uint8_t getTagHistory(TagHistoryEntry* vector);</code> ▪ <code>void printTagHistory();</code>
Acesso	▪ <code>TagHistoryEntry* getTagHistoryPointer();</code> ▪ <code>uint8_t* getTagHistoryIndexPointer();</code> ▪ <code>void removeLastTagHistoryEntry();</code>

3.3.3 – Módulo Leitor PN532

O módulo de Leitor PN532 oferece suporte à leitura e escrita de etiquetas pelo leitor PN532. A implementação faz o uso do padrão NDEF (*NFC Exchange Format*) para a interação com a etiqueta. Este padrão é amplamente utilizado em dispositivos que comportam a tecnologia NFC, como celulares e algumas de cartão.

Além de permitir a troca entre sistemas compatíveis, esse padrão faz com que os múltiplos blocos de memória de característica *rewrite* presentes na etiqueta e que normalmente precisam ser acessados um a um, sejam abstraídos em uma única estrutura chamada *NDEF message*. No entanto, por se tratar de um encapsulamento de informação, ele aumenta um pouco a quantidade de memória utilizada na etiqueta.

Uma *NDEF message* pode conter múltiplos *NDEF records*. Um *NDEF record* é uma estrutura composta de cabeçalho e dados. Sendo que o cabeçalho é composto pelas seguintes informações básicas: o tamanho dos dados úteis, o seu tipo e um identificador (NFC Forum, 2006). A Figura 18 demonstra como é estruturada uma *NDEF message*.

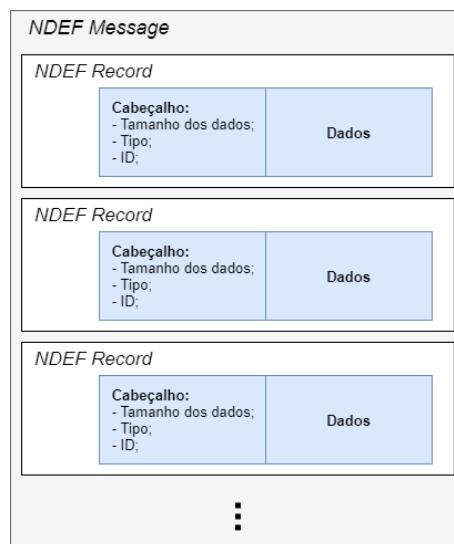


Figura 18 - Esquema do formato NDEF

Dessa forma, foi definida uma *NDEF Record* do tipo PNRD com as seguintes informações:

- **Número de versão:** determina de que forma os dados relativos à PNRD estão organizados. Ela ocupa um espaço de um *byte*. A versão nº 1 foi definida de forma que os dados a serem gravados nas etiquetas sejam exatamente iguais às estruturas de dados utilizadas pela biblioteca, sendo que a ordem dos dados é a seguinte: vetor de marcações, listas de adjacências, vetor de disparos, condições, histórico da etiqueta. Outras versões podem ser futuramente desenvolvidas com o intuito de realizar a compactação dos dados.
- **Cabeçalho de informação presente:** ocupando um *byte*, ele indica quais tipos de informação estão presentes na etiqueta. Sendo que cada bit indica se uma informação está ou não presente, de acordo com a Tabela 7. Conforme pode ser observado, há espaço para a definição de novos tipos de dados.
- **Número de estados e número de transições:** ocupando dois *bytes* e contendo a informação da quantidade de estados e de transições da Rede de Petri associada à etiqueta, essa informação pode ser usada para identificar se o objeto *Pnrd* é capaz de lidar com a quantidade de informação da etiqueta.

- **Dados:** contém a informação útil da estrutura, seu tamanho é variável.

Tabela 7 - Relação entre o tipo de dado e o *bit* de referência no cabeçalho de informação presente

Tipo de Dado	<i>Bit</i> indicativo (do menos para o mais significativo)
Vetor de marcações	1º
Listas de adjacência	2º
Vetor de disparos	3º
Condições	4º
Histórico da etiqueta	5º
Não definido	6º
Não definido	7º
Não definido	8º

3.3.4 – Exemplo de código de aplicação.

Em seguida, há um exemplo de código de aplicação, que realiza uma verificação de disparos simples usando a biblioteca desenvolvida. Nele, configura-se uma rede PNRD com 3 estados e 3 transições. São realizadas as seguintes etapas de configuração, nessa ordem:

1. Definição do tipo de comunicação utilizada no PN532. Nesse caso, foi usada a Serial 1 do Arduino® Mega;
2. Criação dos objetos do leitor e da PNRD.
3. Inicialização do leitor e da comunicação com o computador;
4. Definição das estruturas de dado que serão gravadas na etiqueta;

Depois de realizada a configuração do sistema, um laço infinito é executado. Dentro desse laço é feita uma tentativa de leitura de uma etiqueta. Caso a leitura for bem-sucedida e a etiqueta for diferente da última lida, o programa segue adiante. Se não, ele tenta realizar uma nova leitura.

Dessa forma, a aplicação tenta realizar o disparo da primeira transição na PNRD da etiqueta lida. Se o disparo for bem-sucedido, o valor do vetor de marcações resultante é mostrado e a nova informação é gravada na etiqueta.

```

1 #include <Pn532NfcReader.h>
2 #include <PN532_HSU.h>
3

```

```

4  //Definição da comunicação com o leitor RFID PN532
5  PN532_HSU pn532hsu(Serial1);
6  NfcAdapter nfc = NfcAdapter(pn532hsu);
7
8  //Criação dos objetos de leitor e PNRD
9  Pn532NfcReader* reader = new Pn532NfcReader(&nfc);
10 Pnrd pnrd = Pnrd(reader, 3, 3);
11
12 uint32_t tagId = 0xFF;
13
14 void setup() {
15     //Iniciação do leitor e da UART
16     Serial.begin(9600);
17     reader->initialize();
18
19     //Configuração para a abordagem PNRD clássica
20     pnrd.setAsTagInformation(PetriNetInformation::TOKEN_VECTOR);
21     pnrd.setAsTagInformation(PetriNetInformation::ADJACENCY_LIST);
22 }
23
24 void loop() {
25
26     //Tentativa de leitura da etiqueta
27     ReadError readError = pnrd.getData();
28
29     if (readError == ReadError::NO_ERROR) {
30
31         //Verifica se é uma nova etiqueta
32         if (tagId != pnrd.getTagId()) {
33             tagId = pnrd.getTagId();
34             Serial.print('\n');
35             Serial.print("Leitura da etiqueta de código: ");
36             Serial.println(tagId, HEX);
37         }
38         else {
39             return;
40         };
41
42         //Realização do disparo da transição de índice 0
43         FireError fireError = pnrd.fire(0);
44
45         switch (fireError) {
46             case FireError::NO_ERROR:
47                 Serial.println("Disparo bem sucedido.");
48
49                 //Mostrar o vetor de marcações resultante do disparo
50                 pnrd.printTokenVector();
51
52                 //Salvar a nova informação na etiqueta
53                 if (pnrd.saveData() == WriteError::NO_ERROR) {
54                     Serial.println("Informação atualizada.");
55                 } else {
56                     Serial.println("Erro na atualização da etiqueta.");
57                 };
58                 break;
59
60             case FireError::PRODUCE_EXCEPTION:
61                 Serial.println("Erro: disparo gerou exceção.");
62                 break;
63

```

```

64         case FireError::CONDITIONS_ARE_NOT_APPLIED:
65             Serial.println("Erro: condições não são satisfeitas.");
66             break;
67     }
68 }
69 }

```

3.4 Testes

Cada módulo foi desenvolvido e testado separadamente, de modo sequencial. Essa abordagem facilitou o desenvolvimento, pois permitiu uma melhor identificação das fontes de erro.

No final de todas as etapas de desenvolvimento foram realizados testes simulando dois sistemas, um que utiliza PNRD e outro que utiliza PNRD invertida.

Todos os testes realizados foram de carácter exclusivamente funcional. Não houve a avaliação de desempenho de nenhuma das rotinas desenvolvidas.

3.4.1 Testes do módulo de Rede de Petri

Neste módulo foram testadas as rotinas de configuração, de disparo e de *debug*. Foram utilizados os seguintes casos de uso:

- Configuração e disparo de uma rede simples, com 3 transições e 3 lugares;
- Configuração e disparo de uma rede com 3 transições e 6 lugares, com múltiplos estados de entrada e de saída em cada transição;
- Configuração e disparo de uma rede com 250 transições e 250 lugares;
- Realização de um disparo composto de múltiplas transições;
- Realização de um disparo que resulte em exceção;
- Configuração de uma rede com condições associadas e a realização de disparos com e sem as condições habilitadas;

3.4.2 Testes do módulo de PNRD

No módulo PNRD foram testadas as rotinas de determinam quais as informações estão presentes na etiqueta e a criação automática de histórico da etiqueta, em um caso de uso único.

3.4.3 Testes do módulo do leitor PN532

Neste módulo foram testadas a leitura e gravação de dados nas etiquetas RFID, usando os seguintes passos:

- Primeiramente, executava-se um programa que, por meio da biblioteca desenvolvida, gravava os dados da PNRD em 5 etiquetas diferentes;
- Em seguida, em outro programa, lia-se os dados de cada etiqueta *byte a byte*, verificando se os mesmos estão condizentes com a informação pretendida.
- Por último, executava-se um programa de leitura e visualização utilizando a biblioteca desenvolvida;

Foram verificadas as configurações típicas da abordagem PNRD e da PNRD invertida, além do caso em que todas as estruturas de dados são gravadas na mesma etiqueta.

3.4.5 Testes gerais

Para a realização de testes de integração entre os módulos, optou-se pela simulação de dois sistemas, um que utiliza a abordagem de PNRD e um outro que utiliza a abordagem da PNRD invertida.

O primeiro sistema desenvolvido é uma simplificação do exemplo dado no item 2.3, que descreve a utilização da abordagem PNRD em um processo de verificação de mancais.

No lugar de dois testes separados, o sistema simulado só realiza um teste que define se a peça será destinada para o estoque ou para a fila de retrabalho. Dessa forma, o modelo de redes de Petri pode ser descrito conforme a Figura 19.

Para a simulação desse sistema, foram escritos os códigos de três máquinas diferentes. Tais códigos são encontrados no Apêndice B.

A primeira máquina realiza a gravação inicial das etiquetas, salvando a informação da Rede de Petri e o vetor de marcações inicial. Ele pressupõe que as etiquetas já estejam no formato NDEF.

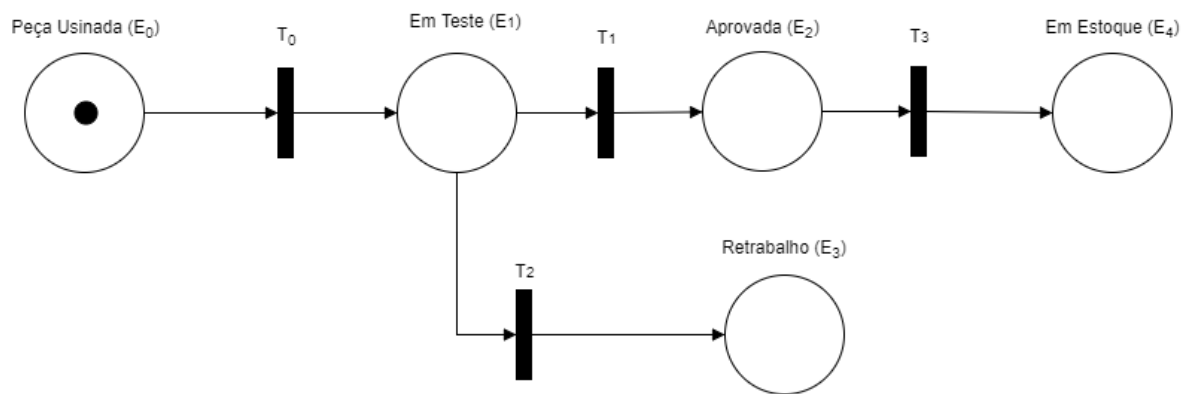


Figura 19 - Representação simplificada de um processo de verificação de mancais usando redes de Petri.

A segunda máquina é a verificadora de peças. Ao chegar uma peça nova, ela ativa a transição T_0 e, em seguida, realiza o teste de conformação com os padrões estabelecidos. De acordo com esse teste, ela ativará a transição T_1 ou T_2 . Na simulação, associou-se o resultado do teste com dois botões. Sendo que um significava que a peça foi aprovada e outro, que ela foi rejeitada, conforme visto na Figura 20.

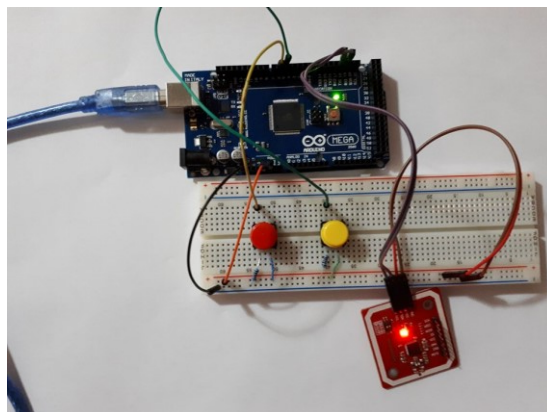


Figura 20 - Montagem física do simulador de uma máquina verificadora de peça. O botão vermelho indica que a peça foi reprovada, o amarelo indica que ela foi aprovada.

A terceira máquina é a gestora de estoque. Ela verifica se a peça que chegou no estoque realmente passou no teste e ativa a transição T_3 .

Na Figura 21, mostra-se as mensagens de *log* do segundo e terceiro elementos, no caso da aprovação ou reprovação da peça. Demonstrando o funcionamento do sistema desenvolvido.

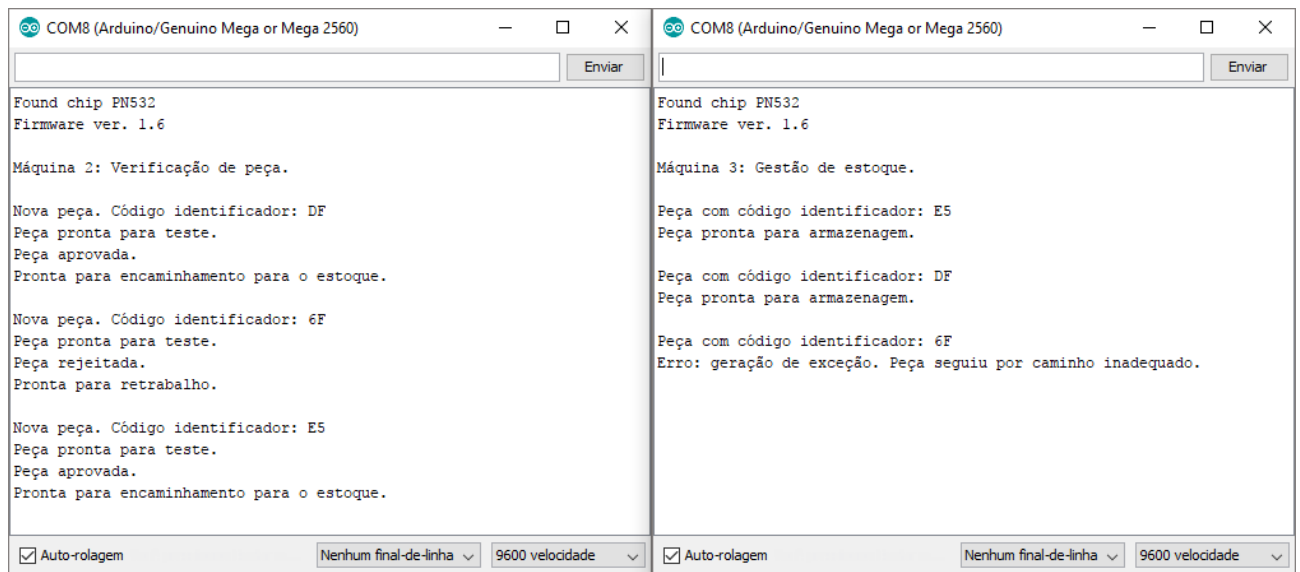


Figura 21 - Funcionamento da simulação de aplicação usando PNRD

O outro sistema desenvolvido, baseado na PNRD invertida, foi composto de um círculo com 5 etiquetas. Cada uma dessas etiquetas contém a informação de um vetor de transições diferente e o seu próprio histórico.

Foram escritos os códigos simulando um robô que percorre o caminho horário no círculo de etiquetas e outro que percorre o caminho anti-horário. A única mudança entre esses dois códigos está na definição da matriz de incidência.

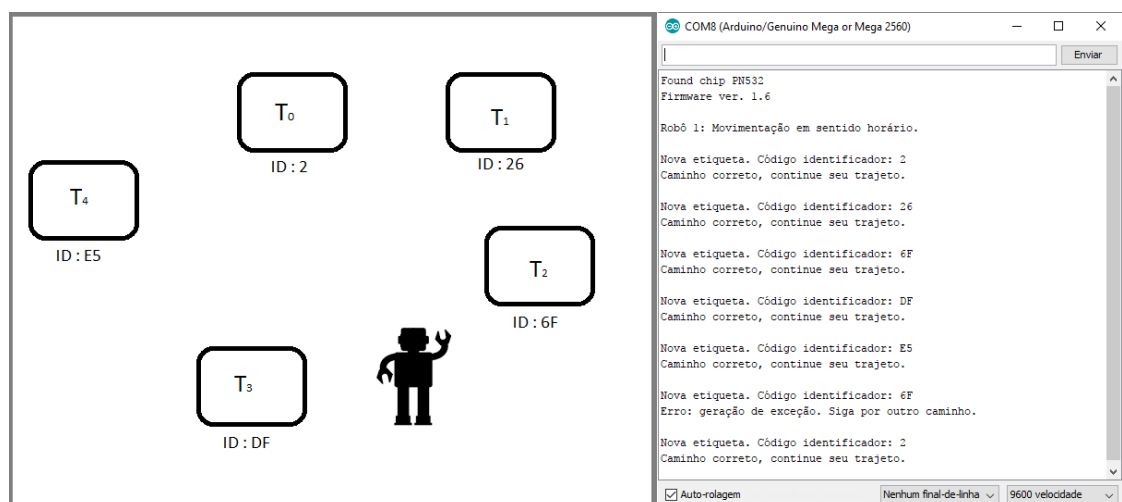


Figura 22 - Ilustração do teste com PNRD invertida

O sistema é capaz de verificar automaticamente se o robô está ou não no caminho correto, conforme ilustrado na Figura 22. No exemplo dado, o robô percorre uma volta nas etiquetas, mas depois segue diretamente para a etiqueta de transição T_2 , causando uma exceção. Depois, ele corrige seu caminho indo para a etiqueta de transição T_0 .

Ainda foi escrito um código que coleta as informações de histórico de cada etiqueta e as exibe para o usuário, conforme mostrado na Figura 23. Os códigos relativos a esse segundo sistema simulado estão no Apêndice C.

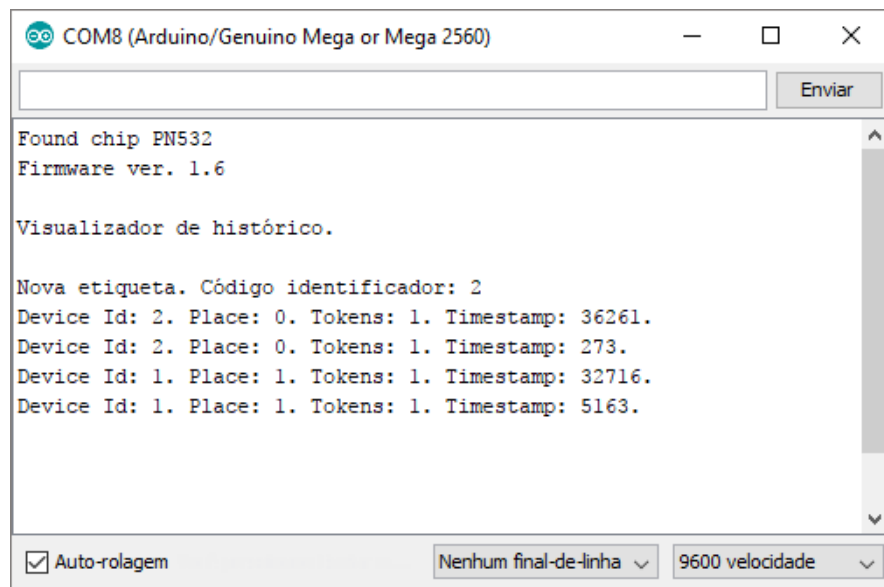


Figura 23 - Visualizador de histórico de uma etiqueta

RESULTADOS E CONCLUSÕES

A implementação da biblioteca foi realizada cumprindo os requisitos e objetivos estabelecidos anteriormente. Além disso, ampliou-se a possibilidade da sua utilização para além das abordagens da PNRD ou PNRD invertida clássicas. Pois, com a configuração de quais tipos de informações devem ou não ser gravadas na etiqueta, é possível a definição de outros tipos de base de dados RFID baseada em redes de Petri.

4.1 Resultados

Depois da realização de diversos testes, a biblioteca desenvolvida foi validada e está apta a ser utilizada em aplicações reais que utilizam a abordagem de PNRD e PNRD invertida. Além disso, ela também permite a utilização de estruturadas de dado que misturam as duas abordagens.

A relativa rapidez no desenvolvimento dos programas de teste demonstra que o objetivo principal do trabalho, que é a facilitação da aplicação do conceito da PNRD, foi cumprido. Além disso, avalia-se que todos os requisitos definidos por projeto foram apropriadamente cumpridos.

No entanto, durante a execução dos testes, ficou perceptível a geração de mensagens de erro no momento em que as etiquetas são removidas do alcance do leitor. Tais mensagens são provenientes da biblioteca de terceiros que serviu como base na implementação dos métodos de leitura e escrita de etiquetas. No caso de grandes frequências de leitura e escrita, esses erros podem chegar a travar ou reiniciar o programa do Arduino®. Futuramente, é aconselhável a realização de modificações nessa biblioteca ou a sua substituição de modo a garantir uma maior confiabilidade dos sistemas desenvolvidos.

4.3 Conclusão

A realização deste trabalho abrangeu diversas áreas de estudo do curso de engenharia mecatrônica. Foi necessário aplicar o conhecimento visto em disciplinas como eletrônica digital, sistemas embarcados, simulação de sistemas automatizados, algoritmos e programação de computadores, álgebra linear, sistemas operacionais, arquitetura de redes de computadores, banco de dados e redes industriais. Além

disso, foi necessário o aprofundamento de conhecimentos em assuntos como engenharia de *software*, estruturas de dado, tecnologia RFID e Redes de Petri.

Dessa forma, conclui-se que o trabalho realizado vem a contribuir de maneira efetiva tanto na formação do discente envolvido, quanto na geração de futuras pesquisas. Apesar de várias melhorias e implementações de funcionalidades serem possíveis, o *software* desenvolvido já pode começar a ser utilizado de forma prática como uma ferramenta eficiente de desenvolvimento de aplicações baseadas em PNRD e PNRD invertida.

4.2 Trabalhos futuros

Os trabalhos a serem realizados tendo como base a biblioteca produzida podem ser classificados como:

- Implementação de funcionalidades: algumas das possíveis melhorias a serem desenvolvidas são: a implementação de novos tipos de leitores, a criação de versões mais compactas de dados para as etiquetas, o suporte a redes de Petri com pesos nos arcos, a especificação de capacidades máximas de marcações em cada estado e a modelagem por redes de Petri coloridas.

- Desenvolvimento de aplicações: atualmente planeja-se utilizar a biblioteca realizada para a programação de uma célula de flexível de manufatura e de um sistema de busca e salvamento de pessoas no MAPL (Laboratório de Planejamento Automático de Manufatura) na UFU. Além disso, ela pode ser futuramente utilizada na lógica de exames de robôs e no planejamento em tempo real de sistemas de produção.

- Documentação: a biblioteca pode ser documentada e disponibilizada de forma a difundir a sua utilização.

- Utilização em outros sistemas: por ser baseada em C++ e feita de forma generalizada, essa biblioteca também pode servir como base na escrita de métodos para a implementação da lógica PNRD em celulares e outros sistemas embarcados ou até em sistemas computacionais complexos.

BIBLIOGRAFIA

AALST, W. Petri nets (1/2). Apresentação. Universidade de Tecnologia de Eindhoven. 201-.

AHSAN, K.; SHAH, H.; KINGTON P. RFID Applications: An Introductory and Exploratory Study. **IJCSI International Journal of Computer Science Issues**, Vol. 7, Issue 1, Nº 3, 2010.

ARDUINO. Arduino®. What is Arduino? Disponível em: <<https://www.arduino.cc/en/Guide/Introduction#>>. Acesso em 09/12/2017a.

ARDUINO. Site oficial do Arduino®. Disponível em: <<https://www.arduino.cc/>>. Acesso em 09/12/2017b.

DAVID, R.; ALLA, H. Discrete, Continuous, and Hybrid Petri Nets. Springer Science & Business Media, 2010.

ELECTROHOUSE. PN532 NFC RFID Module User Guide. Versão 3. Disponível em <http://www.elechouse.com/elechouse/images/product/PN532_module_V3/PN532_%20Manual_V3.pdf>. Acesso em 11/12/2017.

FONSECA, J. P. S.; TAVARES, J.J.P.Z.S. Petri Net with RFID Distributed Database for aous Search and Rescue in Trails and Crossings. **Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'17)**, p. 229-230, 2017.

FUJITSU LTD. Datasheet World's Largest-Capacity 64KByte FRAM Metal Mount RFID Tag. 2014. Disponível em: <<http://www.fujitsu.com/downloads/AIT/ait-downloads-64kbttag.pdf>>. Acesso em 08/12/2017.

MUROFUSHI, R. H., Desenvolvimento de um sistema de posicionamento interno baseado na potência do sinal de resposta de um sistema RFID. 2016. Dissertação de Mestrado, Universidade Federal de Uberlândia, Uberlândia.

LI Z., MENG Z. RFID Tag as a Sensor - A Review on the Innovative Designs and Applications. **Measurement Science Review**, Volume 16, Issue 6, p.305-315. 2016.

MACIEL, P.R.M.; LINS, R.D.; CUNHA, P.R.F. Introdução às Redes de Petri e Aplicações, **10ª Escola de Computação**, Campinas, 1996.

MURATA, T. 1989. Petri Nets: Properties, Analysis and Applications. **Proceedings of the IEEE**, vol. 77, nº 4, p. 541-580.

NATH, B.; REYNALDS F.; WANT, R. RFID Technology and Applications. **IEEE Pervasive Computing**, vol. 5, nº 1, p. 22-24, 2006.

NFC Forum™. NFC Data Exchange Format (NDEF) – Technical Specification. 2006. Disponível em <<http://archive.eet-china.com/www.eet-china.com/ARTICLES/2006AUG/PDF/NFCForum-TS-NDEF.pdf?SOURCES=DOWNLOAD>>. Acesso em 13/12/2017.

PRESSMAN, R. S. Engenharia de Software - Uma abordagem profissional. 7ª Edição. AMGH Editora Ltda. 2011.

TAVARES, J.J.P.Z.S., SARAIVA, T.A. Elementary Petri Nets inside RFID Database (PNRD). **Journal International Journal of Production Research**, vol. 48, 2010 – n. 9: RFID Technology and Applications in Production and Supply Chain Management.

WANT, R. An Introduction to RFID Technology. **IEEE Pervasive Computing**, vol. 5, nº 1, p. 25-33, 2006.

ZHANG, Y.; WANG, W.; WU, N. IoT-Enabled Real-Time Production Performance Analysis in Exception Diagnosis Model. **IEEE Transaction on Automation Science and Engineering**, Vol. 13, nº 3, p. 1318-1332, 2016.

Apêndices

Apêndice A: Tabela de Comparação entre as Estruturas de Dados para armazenagem da informação dos arcos em uma rede de Petri.

			Comparação de Estruturas			Grau de Importância	Estrutura de melhor performance
			Matriz de Incidência	Matrizes de Adjacência	Listas de Adjacência		
Parâmetros de Comparação	Memória alocada*		$[t * e / 4]$	$2 * [t * e / 8]$	$t * (max_{in} + max_{out})$	Grande	Depende da densidade da rede
	Complexidade das Operações básicas	Definir um arco	$O(1)$	$O(2)$	$O(k_{in})$ ou $O(k_{out})$	Pequeno	Matriz de Incidência
		Determinar uma relação entre estado e transição	$O(1)$	$O(2)$	$O(k_{in} + k_{out})$	Médio	Matriz de Incidência
		Verificar se o disparo é possível	$O(e)$	$O(e)$	$O(k_{in} + 1)$	Grande	Lista de Adjacência
		Realizar o disparo de uma transição	$O(e)$	$O(2e)$	$O(k_{in} + k_{out})$	Grande	Lista de Adjacência
Vantagens não mensuráveis	O espaço alocado independentemente do número de entradas/saídas		Sim	Sim	Não	Médio	Matrizes de Adjacência
	Permite a modelagem de redes impuras		Não	Sim	Não	Pequeno	

Símbolos:

t = número de transições;

e = número de estados;

k_{in} = quantidade de arcos na lista de entradas;

k_{out} = quantidade de arcos na lista de saídas;

max_{in} = quantidade máxima de entradas;

max_{out} = quantidade máxima de saídas;

Apêndice B: códigos da simulação de aplicação utilizando abordagem PNRD

Máquina 1: Gravação inicial das etiquetas

```
1  #include <Pn532NfcReader.h>
2  #include <PN532_HSU.h>
3
4  //Definição das informações da etiqueta
5  int8_t mIncidenceMatrix[] = { -1,  0,  1,  0,
6                                1, -1, -1,  0,
7                                0,  1,  0, -1,
8                                0,  0,  1,  0,
9                                0,  0,  0,  1 };
10
11  uint16_t mStartingTokenVector[] = { 1, 0, 0, 0, 0 };
12
13  //Definição da comunicação com o leitor RFID PN532
14  PN532_HSU pn532hsu(Serial1);
15  NfcAdapter nfc = NfcAdapter(pn532hsu);
16
17  //Criação dos objetos de leitor e PNRD
18  Pn532NfcReader* reader = new Pn532NfcReader(&nfc);
19  Pnrd pnrd = Pnrd(reader, 5, 4);
20
21  void setup() {
22      //Iniciação do leitor e da UART
23      Serial.begin(9600);
24      reader->initialize();
25
26      pnrd.setIncidenceMatrix(mIncidenceMatrix);
27      pnrd.setTokenVector(mStartingTokenVector);
28
29      //Configuração para a abordagem PNRD clássica
30      pnrd.setAsTagInformation(PetriNetInformation::TOKEN_VECTOR);
31      pnrd.setAsTagInformation(PetriNetInformation::ADJACENCY_LIST);
32
33      Serial.print("\nMáquina 1: Gravação inicial das etiquetas.");
34  }
35
36  void loop() {
37      delay(1000);
38      Serial.println("Aproxime uma etiqueta a ser configurada.");
39
40      //Tentativa de escrita na etiqueta
41      if (pnrd.saveData() == WriteError::NO_ERROR) {
42          Serial.println("Etiqueta configurada com sucesso.");
43      };
44
45      Serial.print('\n');
46  }
```

Máquina 2: Verificação de peças

```
1  #include <Pn532NfcReader.h>
2  #include <PN532_HSU.h>
3
4  //Rotinas relacionadas à comunicação com o leitor RFID PN532
5  PN532_HSU pn532hsu(Serial1);
6  NfcAdapter nfc = NfcAdapter(pn532hsu);
7
8  //Criação dos objetos de leitor e PNRD
9  Pn532NfcReader* reader = new Pn532NfcReader(&nfc);
10 Pnrd pnrd = Pnrd(reader, 5, 4, true, false);
11
12 uint32_t tagId = 0xFF;
13
14 const int buttonAccept = 6;
15 const int buttonReject = 5;
16
17 bool tagReadyToContinue = false;
18
19 void setup() {
20     //Iniciação do leitor e da serial
21     Serial.begin(9600);
22     reader->initialize();
23
24     //Iniciação dos botões
25     pinMode(buttonAccept, INPUT);
26     pinMode(buttonReject, INPUT);
27
28     //Configuração para a abordagem PNRD clássica
29     pnrd.setAsTagInformation(PetriNetInformation::TOKEN_VECTOR);
30     pnrd.setAsTagInformation(PetriNetInformation::ADJACENCY_LIST);
31
32     Serial.println("\nMáquina 2: Verificação de peça.");
33 }
34
35 void loop() {
36     delay(1000);
37
38     //Leitura da etiqueta
39     ReadError readError = pnrd.getData();
40
41     //Caso a leitura for realizada sem erros
42     if (readError == ReadError::NO_ERROR) {
43         FireError fireError;
44
45         //Verifica se é uma nova etiqueta
46         if (tagId != pnrd.getTagId()) {
47             tagId = pnrd.getTagId();
48             Serial.print("\nNova peça. Código identificador: ");
49             Serial.println(tagId, HEX);
50             tagReadyToContinue = false;
51
52             //Realização do disparo da transição de índice 0
53             FireError fireError = pnrd.fire(0);
54
55             switch (fireError) {
56                 case FireError::NO_ERROR:
57                     //Salvar a nova informação na etiqueta
58                     if (pnrd.saveData() == WriteError::NO_ERROR) {
```

```

59         Serial.println("Peça pronta para teste.");
60         return;
61     } else {
62         Serial.println("Erro na atualização da etiqueta.");
63         return;
64     };
65
66     case FireError::PRODUCE_EXCEPTION:
67         Serial.println("Erro: geração de exceção. Peça seguiu por caminho
68 inadequado.");
69         return;
70
71     case FireError::CONDITIONS_ARE_NOT_APPLIED:
72         break;
73 }
74
75 } else {
76     //Habilitação das transições no caso da aprovação ou reprovação da peça
77     pnrđ.conditionUpdate(1, digitalRead(buttonAccept) && !tagReadyToContinue);
78     pnrđ.conditionUpdate(2, digitalRead(buttonReject) && !tagReadyToContinue);
79
80     //Realização do disparo da transição de índice 1
81     fireError = pnrđ.fire(1);
82     switch (fireError) {
83     case FireError::NO_ERROR:
84         Serial.println("Peça aprovada.");
85         //Salvar a nova informação na etiqueta
86         if (pnrđ.saveData() == WriteError::NO_ERROR) {
87             Serial.println("Pronta para encaminhamento para o estoque.");
88             tagReadyToContinue = true;
89         } else {
90             Serial.println("Erro na atualização da etiqueta.");
91         };
92         return;
93
94     case FireError::PRODUCE_EXCEPTION:
95         Serial.println("Erro: geração de exceção. Peça seguiu por caminho
96 inadequado.");
97         return;
98
99     case FireError::CONDITIONS_ARE_NOT_APPLIED:
100         break;
101     }
102
103     //Realização do disparo da transição de índice 2
104     fireError = pnrđ.fire(2);
105     switch (fireError) {
106     case FireError::NO_ERROR:
107         Serial.println("Peça rejeitada.");
108         //Salvar a nova informação na etiqueta
109         if (pnrđ.saveData() == WriteError::NO_ERROR) {
110             Serial.println("Pronta para retrabalho.");
111             tagReadyToContinue = true;
112         } else {
113             Serial.println("Erro na atualização da etiqueta.");
114         };
115         return;
116
117     case FireError::PRODUCE_EXCEPTION:
118         Serial.println("Erro: geração de exceção. Peça seguiu por caminho
119 inadequado.");
120         return;

```

```

121
122         case FireError::CONDITIONS_ARE_NOT_APPLIED:
123             break;
124     }
125 }
126 }
127 }

```

Máquina 3 : Gerenciamento de estoque

```

1  #include <Pn532NfcReader.h>
2  #include <PN532_HSU.h>
3
4  //Rotinas relacionadas à comunicação com o leitor RFID PN532
5  PN532_HSU pn532hsu(Serial1);
6  NfcAdapter nfc = NfcAdapter(pn532hsu);
7
8  //Criação dos objetos de leitor e PNRD
9  Pn532NfcReader* reader = new Pn532NfcReader(&nfc);
10 Pnrd pnrdr = Pnrd(reader, 5, 4);
11
12 uint32_t tagId = 0xFF;
13
14 void setup() {
15     //Iniciação do leitor e da serial
16     Serial.begin(9600);
17     reader->initialize();
18
19     //Configuração para a abordagem PNRD clássica
20     pnrdr.setAsTagInformation(PetriNetInformation::TOKEN_VECTOR);
21     pnrdr.setAsTagInformation(PetriNetInformation::ADJACENCY_LIST);
22
23     Serial.println("\nMáquina 3: Gestão de estoque.");
24 }
25
26 void loop() {
27     delay(1000);
28
29     //Leitura da etiqueta
30     ReadError readError = pnrdr.getData();
31
32     //Caso a leitura for realizada sem erros
33     if (readError == ReadError::NO_ERROR) {
34         FireError fireError;
35
36         //Verifica se é uma nova etiqueta
37         if (tagId != pnrdr.getTagId()) {
38             tagId = pnrdr.getTagId();
39             Serial.print("\nPeça com código identificador: ");
40             Serial.println(tagId, HEX);
41
42             //Realização do disparo da transição de índice 3
43             FireError fireError = pnrdr.fire(3);
44
45             switch (fireError) {
46                 case FireError::NO_ERROR:
47
48                     //Salvar a nova informação na etiqueta

```

```

49         if (pnrd.saveData() == WriteError::NO_ERROR) {
50             Serial.println("Peça pronta para armazenagem.");
51             return;
52         }else {
53             Serial.println("Erro na atualização da etiqueta.");
54             return;
55         };
56         return;
57
58         case FireError::PRODUCE_EXCEPTION:
59             Serial.println("Erro: geração de exceção. Peça seguiu por caminho
60 inadequado.");
61             break;
62
63         case FireError::CONDITIONS_ARE_NOT_APPLIED:
64             break;
65     }
66 }
67 }
68 }

```

Apêndice C : códigos da simulação de aplicação utilizando abordagem PNRD invertida

Robô com movimentação em sentido horário

```
1  #include <Pn532NfcReader.h>
2  #include <PN532_HSU.h>
3
4  int8_t mIncidenceMatrix[] = { -1,  0,  0,  0,  1,
5                                1, -1,  0,  0,  0,
6                                0,  1, -1,  0,  0,
7                                0,  0,  1, -1,  0,
8                                0,  0,  0,  1, -1 };
9
10 uint16_t mStartingTokenVector[] = { 1,0,0,0,0 };
11
12 //Rotinas relacionadas à comunicação com o leitor RFID PN532
13 PN532_HSU pn532hsu(Serial1);
14 NfcAdapter nfc = NfcAdapter(pn532hsu);
15
16 //Criação dos objetos de leitor e PNRD
17 Pn532NfcReader* reader = new Pn532NfcReader(&nfc);
18 Pnrd pnrd = Pnrd(reader, 5, 5, false, true);
19
20 uint32_t tagId = 0xFF;
21
22 void setup() {
23     //Iniciação do leitor e da serial
24     Serial.begin(9600);
25     reader->initialize();
26
27     //Iniciação dos dados relativos ao modelo de RP do robô
28     pnrd.setTokenVector(mStartingTokenVector);
29     pnrd.setIncidenceMatrix(mIncidenceMatrix);
30
31     //Configuração para a abordagem PNRD invertida
32     pnrd.setAsTagInformation(PetriNetInformation::FIRE_VECTOR);
33     pnrd.setAsTagInformation(PetriNetInformation::TAG_HISTORY);
34
35     //Configuração do ID do robô
36     pnrd.setDeviceId(1);
37     Serial.println("\nRobô 1: Movimentação em sentido horário.");
38 }
39
40 void loop() {
41     delay(1000);
42
43     //Leitura da etiqueta
44     ReadError readError = pnrd.getData();
45
46     //Caso a leitura for realizada sem erros
47     if (readError == ReadError::NO_ERROR) {
48         FireError fireError;
49
50         //Verifica se é uma nova etiqueta
51         if (tagId != pnrd.getTagId()) {
```

```

52         tagId = pnrđ.getTagId();
53         Serial.print("\nNova etiqueta. Código identificador: ");
54         Serial.println(tagId, HEX);
55
56         //Realização do disparo contido na etiqueta
57         FireError fireError = pnrđ.fire();
58
59         switch (fireError) {
60         case FireError::NO_ERROR:
61             //Salvar a nova informação na etiqueta
62             if (pnrđ.saveData() == WriteError::NO_ERROR) {
63                 Serial.println("Caminho correto, continue seu trajeto.");
64                 return;
65             } else {
66                 Serial.println("Erro na atualização da etiqueta.");
67                 return;
68             };
69             return;
70
71         case FireError::PRODUCE_EXCEPTION:
72             Serial.println("Erro: geração de exceção. Siga por outro
73 caminho.");
74             break;
75         }
76     }
77 }
78 }

```

Robô com movimentação em sentido anti-horário

```

#include <Pn532NfcReader.h>
#include <PN532_HSU.h>

int8_t mIncidenceMatrix[] = { 1, 0, 0, 0, -1,
                             -1, 1, 0, 0, 0,
                             0, -1, 1, 0, 0,
                             0, 0, -1, 1, 0,
                             0, 0, 0, -1, 1 };

uint16_t mStartingTokenVector[] = { 1,0,0,0,0 };

//Rotinas relacionadas à comunicação com o leitor RFID PN532
PN532_HSU pn532hsu(Serial1);
NfcAdapter nfc = NfcAdapter(pn532hsu);

//Criação dos objetos de leitor e PNRD
Pn532NfcReader* reader = new Pn532NfcReader(&nfc);
Pnrđ pnrđ = Pnrđ(reader, 5, 5, false, true);

uint32_t tagId = 0xFF;

void setup() {
    //Iniciação do leitor e da serial
    Serial.begin(9600);
    reader->initialize();
}

```

```

//Iniciação dos dados relativos ao modelo de RP do robô
pnrd.setTokenVector(mStartingTokenVector);
pnrd.setIncidenceMatrix(mIncidenceMatrix);

//Configuração para a abordagem PNRD invertida
pnrd.setAsTagInformation(PetriNetInformation::FIRE_VECTOR);
pnrd.setAsTagInformation(PetriNetInformation::TAG_HISTORY);

//Configuração do ID do robô
pnrd.setDeviceId(2);
Serial.println("\nRobô 2: Movimentação em sentido anti-horário.");
}

void loop() {
    delay(1000);

    //Leitura da etiqueta
    ReadError readError = pnrd.getData();

    //Caso a leitura for realizada sem erros
    if (readError == ReadError::NO_ERROR) {
        FireError fireError;

        //Verifica se é uma nova etiqueta
        if (tagId != pnrd.getTagId()) {
            tagId = pnrd.getTagId();
            Serial.print("\nNova etiqueta. Código identificador: ");
            Serial.println(tagId, HEX);

            //Realização do disparo contido na etiqueta
            FireError fireError = pnrd.fire();

            switch (fireError) {
                case FireError::NO_ERROR:

                    //Salvar a nova informação na etiqueta
                    if (pnrd.saveData() == WriteError::NO_ERROR) {
                        Serial.println("Caminho correto, continue seu trajeto.");
                        return;
                    } else {
                        Serial.println("Erro na atualização da etiqueta.");
                        return;
                    }
                };
                return;

            case FireError::PRODUCE_EXCEPTION:
                Serial.println("Erro: geração de exceção. Siga por outro
caminho.");
                break;
            }
        }
    }
}

```

Visualizador de Histórico da Etiqueta

```
#include <Pn532NfcReader.h>
#include <PN532_HSU.h>

//Rotinas relacionadas à comunicação com o leitor RFID PN532
PN532_HSU pn532hsu(Serial1);
NfcAdapter nfc = NfcAdapter(pn532hsu);

//Criação dos objetos de leitor e PNRD
Pn532NfcReader* reader = new Pn532NfcReader(&nfc);
Pnrd pnrd = Pnrd(reader, 5, 5, false, true);

uint32_t tagId = 0xFF;

void setup() {
    //Iniciação do leitor e da serial
    Serial.begin(9600);
    reader->initialize();

    //Configuração para a abordagem PNRD invertida
    pnrd.setAsTagInformation(PetriNetInformation::FIRE_VECTOR);
    pnrd.setAsTagInformation(PetriNetInformation::TAG_HISTORY);

    Serial.println("\nVisualizador de histórico.");
}

void loop() {
    delay(2000);
    //Leitura da etiqueta
    ReadError readError = pnrd.getData();

    //Caso a leitura for realizada sem erros
    if (readError == ReadError::NO_ERROR) {
        if (tagId != pnrd.getTagId()) {
            tagId = pnrd.getTagId();
            Serial.print("\nNova etiqueta. Código identificador: ");
            Serial.println(tagId, HEX);

            //Visualização do histórico
            pnrd.printTagHistory();
        }
    }
}
```

Apêndice D : arquivos de código da biblioteca desenvolvida

PetriNet.h

```
1  #ifndef PETRI_NET_H
2  #define PETRI_NET_H
3
4  #include <Arduino.h>
5
6  enum class FireError {
7      NO_ERROR = 0,
8      CONDITIONS_ARE_NOT_APPLIED = 1, //One or more of the conditions necessary for
9  the transition aren't available
10     PRODUCE_EXCEPTION = 2,           //The result of the fire is an exception
11     NOT_KNOWN = 3
12 };
13
14 /*
15 PlatformInterface class helps to implement the PetriNet library across platforms.
16 In most of the cases, the only modifications you will have to make is to extend this
17 class
18 to match your plaform and replace it on the Platforminterface of the PetriNet class.
19 */
20 class PlatformInterface {
21 public:
22     virtual void print(char character) = 0;
23     virtual uint32_t getTimeStamp() = 0;
24 };
25
26 //Implementation of Arduino-based functions
27 class Arduino : public PlatformInterface {
28 public:
29     void print(char character) {
30         Serial.print(character);
31     };
32
33     uint32_t getTimeStamp() {
34         return millis();
35     }
36 };
37
38 class PetriNet {
39 protected:
40     uint8_t* AdjacencyList;
41     uint16_t* TokenVector;
42     uint16_t* FireVector;
43     uint8_t* Conditions;
44
45     bool hasConditions = false;
46
47     uint8_t NumberOfPlaces;
48     uint8_t NumberOfTransitions;
49     uint8_t NumberMaxOfInputs;
50     uint8_t NumberMaxOfOutputs;
51
52     PlatformInterface* platformInterface = new Arduino();
53
54     uint8_t* OutputListAuxiliarPointer;
```

```

55
56     void print(char toPrint);
57     void print(uint8_t number);
58     void print(uint16_t toPrint);
59     void print(uint32_t toPrint);
60     void print(char * toPrint);
61
62 private:
63     void prepareMemoryStack();
64
65 public:
66     PetriNet(uint8_t num_places, uint8_t num_transitions, uint8_t num_max_of_inputs,
67     uint8_t num_max_of_outputs, bool hasConditions);
68     PetriNet(uint8_t num_places, uint8_t num_transitions, uint8_t num_max_of_inputs,
69     uint8_t num_max_of_outputs);
70     PetriNet(uint8_t num_places, uint8_t num_transitions, bool hasConditions);
71     PetriNet(uint8_t num_places, uint8_t num_transitions);
72     ~PetriNet();
73
74     uint8_t getNumberOfPlaces();
75     uint8_t getNumberOfTransitions();
76     uint8_t getNumberMaxOfInputs();
77     uint8_t getNumberMaxOfOutputs();
78
79     void setFireVector(uint16_t* vector);
80     void getFireVector(uint16_t* vector);
81     uint16_t* getFireVectorPointer();
82     void printFireVector();
83
84     void setTokenVector(uint16_t* vector);
85     void getTokenVector(uint16_t* vector);
86     uint16_t* getTokenVectorPointer();
87     void printTokenVector();
88
89     bool setIncidenceMatrix(int8_t* matrix);
90     void getIncidenceMatrix(int8_t* matrix);
91     int8_t getMatrixElement(int8_t place, int8_t transition);
92     void printIncidenceMatrix();
93
94     bool addInput(uint8_t place, uint8_t transition);
95     bool deleteInput(uint8_t place, uint8_t transition);
96     bool addOutput(uint8_t place, uint8_t transition);
97     bool deleteOutput(uint8_t place, uint8_t transition);
98
99     uint8_t getInputs(uint8_t transition, uint8_t* inputs);
100     bool setInputs(uint8_t transition, uint8_t* inputs, uint8_t inputsCount);
101     uint8_t getOutputs(uint8_t transition, uint8_t* outputs);
102     bool setOutputs(uint8_t transition, uint8_t* outputs, uint8_t outputsCount);
103
104     uint8_t* getInputsPointer(uint8_t transition);
105     uint8_t* getOutputsPointer(uint8_t transition);
106
107     uint8_t* getAdjacencyListPointer();
108
109     void conditionIsSatisfied(uint8_t transition);
110     void conditionIsNotSatisfied(uint8_t transition);
111     void conditionUpdate(uint8_t transition, bool isSatisfied);
112
113     bool isConditionSatisfied(uint8_t transition);
114     uint8_t* getConditionsPointer();
115     void printConditions();
116

```

```

117         FireError fire();
118         FireError fire(uint8_t transition);
119         FireError isTriggerable(uint8_t transition);
120         FireError isTriggerable();
121     };
122 #endif

```

PetriNet.cpp

```

1  #include "PetriNet.h"
2
3  //Constructors
4  PetriNet::PetriNet(uint8_t num_places, uint8_t num_transitions, uint8_t
5  num_max_of_inputs, uint8_t num_max_of_outputs, bool hasConditions) {
6      this->hasConditions = hasConditions;
7
8      this->NumberOfPlaces = num_places;
9      this->NumberOfTransitions = num_transitions;
10
11      this->NumberMaxOfInputs = num_max_of_inputs;
12      this->NumberMaxOfOutputs = num_max_of_outputs;
13
14      prepareMemoryStack();
15  }
16
17  PetriNet::PetriNet(uint8_t num_places, uint8_t num_transitions, uint8_t
18  num_max_of_inputs, uint8_t num_max_of_outputs) {
19      this->NumberOfPlaces = num_places;
20      this->NumberOfTransitions = num_transitions;
21
22      this->NumberMaxOfInputs = num_max_of_inputs;
23      this->NumberMaxOfOutputs = num_max_of_outputs;
24
25      prepareMemoryStack();
26  }
27
28  PetriNet::PetriNet(uint8_t num_places, uint8_t num_transitions, bool hasConditions) {
29      this->hasConditions = hasConditions;
30
31      this->NumberOfPlaces = num_places;
32      this->NumberOfTransitions = num_transitions;
33
34      this->NumberMaxOfInputs = num_places;
35      this->NumberMaxOfOutputs = num_places;
36
37      prepareMemoryStack();
38  }
39
40  PetriNet::PetriNet(uint8_t num_places, uint8_t num_transitions) {
41      this->NumberOfPlaces = num_places;
42      this->NumberOfTransitions = num_transitions;
43
44      this->NumberMaxOfInputs = num_places;
45      this->NumberMaxOfOutputs = num_places;
46
47      prepareMemoryStack();

```

```

48     }
49
50     //Destructor
51     PetriNet::~PetriNet() {
52         free(TokenVector);
53         free(FireVector);
54         free(AdjacencyList);
55         free(Conditions);
56     }
57
58     //Public methods
59     uint8_t PetriNet::getNumberOfPlaces() {
60         return NumberOfPlaces;
61     }
62
63     uint8_t PetriNet::getNumberOfTransitions() {
64         return NumberOfTransitions;
65     }
66
67     uint8_t PetriNet::getNumberMaxOfInputs()
68     {
69         return NumberMaxOfInputs;
70     }
71
72     uint8_t PetriNet::getNumberMaxOfOutputs()
73     {
74         return NumberMaxOfOutputs;
75     }
76
77     void PetriNet::setFireVector(uint16_t* vector)
78     {
79         for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
80             FireVector[transition] = vector[transition];
81         }
82     }
83
84     void PetriNet::getFireVector(uint16_t* vector) {
85         for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
86             vector[transition] = FireVector[transition];
87         }
88     }
89
90     uint16_t* PetriNet::getFireVectorPointer() {
91         return FireVector;
92     }
93
94     void PetriNet::printFireVector() {
95         print("Fire Vector:\n\n");
96
97         for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
98             print(FireVector[transition]);
99             print('\n');
100         }
101
102         print('\n');
103     }
104
105     void PetriNet::setTokenVector(uint16_t* vector) {
106         for (uint8_t place = 0; place < NumberOfPlaces; place++) {
107             TokenVector[place] = vector[place];
108         }
109     }

```

```

110
111 void PetriNet::getTokenVector(uint16_t* vector) {
112     for (uint8_t place = 0; place < NumberOfPlaces; place++) {
113         vector[place] = TokenVector[place];
114     }
115 }
116
117 uint16_t * PetriNet::getTokenVectorPointer()
118 {
119     return TokenVector;
120 }
121
122 void PetriNet::printTokenVector() {
123     print("Token Vector:\n\n");
124
125     for (int32_t place = 0; place < NumberOfPlaces; place++) {
126         print(TokenVector[place]);
127         print('\n');
128     }
129
130     print('\n');
131 }
132
133 bool PetriNet::setIncidenceMatrix(int8_t* matrix) {
134     bool noError = true;
135
136     uint8_t sizeOfAdjacencyList = (NumberMaxOfInputs + NumberMaxOfOutputs) *
137     NumberOfTransitions;
138     for (uint8_t listIndex = 0; listIndex < sizeOfAdjacencyList; listIndex++) {
139         AdjacencyList[listIndex] = 0xFF;
140     }
141
142     for (uint8_t place = 0; place < NumberOfPlaces; place++) {
143         for (uint8_t transition = 0; transition < NumberOfTransitions;
144 transition++) {
145             if (matrix[place*NumberOfTransitions + transition] == 1) {
146                 noError &= addOutput(place, transition);
147             }
148             else if (matrix[place*NumberOfTransitions + transition] == -1) {
149                 noError &= addInput(place, transition);
150             }
151         }
152     }
153
154     return noError;
155 }
156
157 void PetriNet::getIncidenceMatrix(int8_t* matrix) {
158     uint16_t index;
159
160     for (uint8_t place = 0; place < NumberOfPlaces; place++) {
161         for (uint8_t transition = 0; transition < NumberOfTransitions;
162 transition++) {
163             index = place * NumberOfTransitions + transition;
164             matrix[index] = 0;
165         }
166     }
167
168     for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
169         for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs;
170 inputIndex++) {
171             index = transition * NumberMaxOfInputs + inputIndex;

```

```

172         if (AdjacencyList[index] == 0xFF) {
173             break;
174         }
175         else {
176             matrix[AdjacencyList[index] * NumberOfTransitions +
177 transition] = 1;
178         }
179     }
180 }
181
182     for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
183         for (uint8_t outputIndex = 0; outputIndex < NumberMaxOfInputs;
184 outputIndex++) {
185             index = transition * NumberMaxOfOutputs + outputIndex;
186             if (OutputListAuxiliarPointer[index] == 0xFF) {
187                 break;
188             }
189             else {
190                 matrix[OutputListAuxiliarPointer[index] *
191 NumberOfTransitions + transition] = -1;
192             }
193         }
194     }
195 }
196
197 int8_t PetriNet::getMatrixElement(int8_t place, int8_t transition) {
198     uint16_t index;
199
200     for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs; inputIndex++) {
201         index = transition * NumberMaxOfInputs + inputIndex;
202         if (AdjacencyList[index] == 0xFF) {
203             break;
204         }
205         else if (AdjacencyList[index] == place) {
206             return -1;
207         }
208     }
209
210     for (uint8_t outputIndex = 0; outputIndex < NumberMaxOfInputs; outputIndex++) {
211         index = transition * NumberMaxOfOutputs + outputIndex;
212         if (OutputListAuxiliarPointer[index] == 0xFF) {
213             break;
214         }
215         else if (OutputListAuxiliarPointer[index] == place) {
216             return 1;
217         }
218     }
219
220     return 0;
221 }
222
223 void PetriNet::printIncidenceMatrix() {
224     print("Incidence Matrix: \n\n");
225
226     for (uint8_t place = 0; place < NumberOfPlaces; place++) {
227         for (uint8_t transition = 0; transition < NumberOfTransitions;
228 transition++) {
229             switch (getMatrixElement(place, transition)) {
230                 case 1:
231                     print("1 ");
232                     break;
233                 case -1:

```

```

234             print("-1 ");
235             break;
236         case 0:
237             print("0 ");
238             break;
239     }
240 }
241 print('\n');
242 }
243
244 print('\n');
245 }
246
247 bool PetriNet::addInput(uint8_t place, uint8_t transition) {
248     uint16_t index = transition * NumberMaxOfInputs;
249
250     for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs; inputIndex++,
251 index++) {
252         if (AdjacencyList[index] == 0xFF) {
253             AdjacencyList[index] = place;
254             return true;
255         }
256         else if (AdjacencyList[index] == place) {
257             return false;
258         }
259     }
260     return false;
261 }
262
263 bool PetriNet::deleteInput(uint8_t place, uint8_t transition) {
264     uint16_t index = transition * NumberMaxOfInputs;
265
266     for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs; inputIndex++,
267 index++) {
268         if (AdjacencyList[index] == 0xFF) {
269             return false;
270         }
271         else if (AdjacencyList[index] == place) {
272             for (; inputIndex < NumberMaxOfInputs - 1; inputIndex++, index++)
273 {
274                 AdjacencyList[index] = AdjacencyList[index] + 1;
275             }
276             AdjacencyList[index] = 0xFF;
277             return true;
278         }
279     }
280     return false;
281 }
282
283 bool PetriNet::addOutput(uint8_t place, uint8_t transition) {
284     uint16_t index = transition * NumberMaxOfOutputs;
285
286     for (uint8_t outputIndex = 0; outputIndex < NumberMaxOfOutputs; outputIndex++,
287 index++) {
288         if (OutputListAuxiliarPointer[index] == 0xFF) {
289             OutputListAuxiliarPointer[index] = place;
290             return true;
291         }
292         else if (OutputListAuxiliarPointer[index] == place) {
293             return false;
294         }
295     }

```

```

296         return false;
297     }
298 }
299
300 bool PetriNet::deleteOutput(uint8_t place, uint8_t transition) {
301     uint16_t index = transition * NumberMaxOfInputs;
302
303     for (uint8_t outputIndex = 0; outputIndex < NumberMaxOfInputs; outputIndex++,
304 index++) {
305         if (OutputListAuxiliarPointer[index] == 0xFF) {
306             return false;
307         }
308         else if (OutputListAuxiliarPointer[index] == place) {
309             for (; outputIndex < NumberMaxOfInputs - 1; outputIndex++,
310 index++) {
311                 OutputListAuxiliarPointer[index] =
312 OutputListAuxiliarPointer[index] + 1;
313             }
314             OutputListAuxiliarPointer[index] = 0xFF;
315             return true;
316         }
317     }
318     return false;
319 }
320
321 uint8_t PetriNet::getInputs(uint8_t transition, uint8_t * inputs) {
322     uint8_t counter = 0;
323     uint16_t index = transition * NumberMaxOfInputs;
324
325     for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs; inputIndex++,
326 index++) {
327         if (AdjacencyList[index] == 0xFF) {
328             return counter;
329         }
330         else {
331             inputs[counter] = AdjacencyList[index];
332         }
333     }
334
335     return counter;
336 }
337
338 bool PetriNet::setInputs(uint8_t transition, uint8_t * inputs, uint8_t inputsCount) {
339     bool noError = true;
340     uint16_t index = transition * NumberMaxOfInputs;
341
342     if (inputsCount > NumberMaxOfInputs) {
343         inputsCount = NumberMaxOfInputs;
344         noError = false;
345     }
346
347     uint8_t inputIndex = 0;
348     for (; inputIndex < inputsCount; inputIndex++, index++) {
349         AdjacencyList[index] = inputs[inputIndex];
350     }
351
352     for (; inputIndex < NumberMaxOfInputs; inputIndex++, index++) {
353         AdjacencyList[index] = 0xFF;
354     }
355
356     return noError;
357 }

```

```

358
359 uint8_t PetriNet::getOutputs(uint8_t transition, uint8_t * outputs) {
360     uint8_t counter = 0;
361     uint16_t index = transition * NumberMaxOfOutputs;
362
363     for (uint8_t outputIndex = 0; outputIndex < NumberMaxOfOutputs; outputIndex++,
364 index++) {
365         if (OutputListAuxiliarPointer[index] == 0xFF) {
366             return counter;
367         }
368         else {
369             outputs[counter] = OutputListAuxiliarPointer[index];
370         }
371     }
372
373     return counter;
374 }
375
376 bool PetriNet::setOutputs(uint8_t transition, uint8_t * outputs, uint8_t outputsCount)
377 {
378     bool noError = true;
379     uint16_t index = transition * NumberMaxOfOutputs;
380
381     if (outputsCount > NumberMaxOfOutputs) {
382         outputsCount = NumberMaxOfOutputs;
383         noError = false;
384     }
385
386     uint8_t outputIndex = 0;
387     for (; outputIndex < outputsCount; outputIndex++, index++) {
388         OutputListAuxiliarPointer[index] = outputs[outputIndex];
389     }
390
391     for (; outputIndex < NumberMaxOfOutputs; outputIndex++, index++) {
392         OutputListAuxiliarPointer[index] = 0xFF;
393     }
394
395     return noError;
396 }
397
398 uint8_t * PetriNet::getInputsPointer(uint8_t transition) {
399     return AdjacencyList + transition * NumberMaxOfInputs;
400 }
401
402 uint8_t * PetriNet::getOutputsPointer(uint8_t transition) {
403     return OutputListAuxiliarPointer + transition * NumberMaxOfOutputs;
404 }
405
406 uint8_t * PetriNet::getAdjacencyListPointer()
407 {
408     return AdjacencyList;
409 }
410
411 void PetriNet::conditionIsSatisfied(uint8_t transition) {
412     Conditions[transition / 8] |= 0b00000001 << transition % 8;
413 }
414
415 void PetriNet::conditionIsNotSatisfied(uint8_t transition) {
416     Conditions[transition / 8] &= ~(0b00000001 << transition % 8);
417 }
418
419 void PetriNet::conditionUpdate(uint8_t transition, bool isSatisfied) {

```

```

420         if (isSatisfied) {
421             conditionIsSatisfied(transition);
422         }
423         else {
424             conditionIsNotSatisfied(transition);
425         }
426     }
427
428     bool PetriNet::isConditionSatisfied(uint8_t transition) {
429         return Conditions[transition / 8] & (0b00000001 << transition % 8);
430     }
431
432     uint8_t* PetriNet::getConditionsPointer() {
433         return Conditions;
434     }
435
436     void PetriNet::printConditions() {
437         print("Conditions: \n\n");
438
439         for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
440             print(transition);
441             if (isConditionSatisfied(transition)) {
442                 print(": Satisfied\n");
443             }
444             else {
445                 print(": Not satisfied\n");
446             }
447         }
448
449         print('\n');
450     }
451     FireError PetriNet::fire() {
452
453         if (hasConditions) {
454             for (uint8_t transition = 0; transition < NumberOfTransitions;
455 transition++) {
456                 if (FireVector[transition] > 0) {
457                     if (!isConditionSatisfied(transition)) {
458                         return FireError::CONDITIONS_ARE_NOT_APPLIED;
459                     }
460                 }
461             }
462         }
463
464         for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
465             uint16_t fireQuantity = FireVector[transition];
466             uint16_t index = transition * NumberMaxOfInputs;
467
468             for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs;
469 inputIndex++, index++) {
470                 if (AdjacencyList[index] == 0xFF) {
471                     break;
472                 }
473                 else {
474                     if (TokenVector[AdjacencyList[index]] < fireQuantity) {
475                         //Return the token vector to its initial value when
476 an exception occurs
477
478                     for (inputIndex--; inputIndex < 0; inputIndex--,
479 index--) {
480                         TokenVector[AdjacencyList[index]] +=
481 fireQuantity;
482                     }
483                 }
484             }
485         }
486     }

```

```

482
483         for (transition--; transition < 0; transition--) {
484             fireQuantity = FireVector[transition];
485             index = transition * NumberMaxOfInputs;
486
487             for (uint8_t inputIndex = 0; inputIndex <
488 NumberMaxOfInputs; inputIndex++, index++) {
489                 if (AdjacencyList[index] == 0xFF) {
490                     break;
491                 }
492                 else {
493
494                     TokenVector[AdjacencyList[index]] -= fireQuantity;
495                 }
496             }
497         }
498
499         return FireError::PRODUCE_EXCEPTION;
500     }
501     else {
502         TokenVector[AdjacencyList[index]] -= fireQuantity;
503     }
504 }
505 }
506 }
507
508 for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
509     uint16_t fireQuantity = FireVector[transition];
510     uint16_t index = transition * NumberMaxOfOutputs;
511
512     for (uint8_t outputIndex = 0; outputIndex < NumberMaxOfOutputs;
513 outputIndex++, index++) {
514         if (OutputListAuxiliarPointer[index] == 0xFF) {
515             break;
516         }
517         else {
518             TokenVector[OutputListAuxiliarPointer[index]] +=
519 fireQuantity;
520         }
521     }
522 }
523
524 return FireError::NO_ERROR;
525 }
526
527 FireError PetriNet::fire(uint8_t transition) {
528
529     if (hasConditions) {
530         if (!isConditionSatisfied(transition)) {
531             return FireError::CONDITIONS_ARE_NOT_APPLIED;
532         }
533     }
534
535     uint16_t index = transition * NumberMaxOfInputs;
536     for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs; inputIndex++,
537 index++) {
538         if (AdjacencyList[index] == 0xFF) {
539             break;
540         }
541         else {
542             if (TokenVector[AdjacencyList[index]] < 1) {

```

```

543                                     //Return the token vector to its initial value when an
544 exception occurs
545                                     for (transition--; transition < 0; transition--) {
546                                         index = transition * NumberMaxOfInputs;
547
548                                         for (uint8_t inputIndex = 0; inputIndex <
549 NumberMaxOfInputs; inputIndex++, index++) {
550                                             if (AdjacencyList[index] == 0xFF) {
551                                                 break;
552                                             }
553                                             else {
554                                                 TokenVector[AdjacencyList[index]] ++;
555                                             }
556                                         }
557                                     }
558
559                                     return FireError::PRODUCE_EXCEPTION;
560                                 }
561                                 else {
562                                     TokenVector[AdjacencyList[index]]--;
563                                 }
564                             }
565                         }
566
567                         index = transition * NumberMaxOfOutputs;
568                         for (uint8_t outputIndex = 0; outputIndex < NumberMaxOfOutputs; outputIndex++,
569 index++) {
570                             if (OutputListAuxiliarPointer[index] == 0xFF) {
571                                 break;
572                             }
573                             else {
574                                 TokenVector[OutputListAuxiliarPointer[index]] ++;
575                             }
576                         }
577
578                         return FireError::NO_ERROR;
579                     }
580
581     FireError PetriNet::isTriggerable(uint8_t transition) {
582
583         if (hasConditions) {
584             for (uint8_t transition = 0; transition < NumberOfTransitions;
585 transition++) {
586                 if (FireVector[transition] > 0) {
587                     if (!isConditionSatisfied(transition)) {
588                         return FireError::CONDITIONS_ARE_NOT_APPLIED;
589                     }
590                 }
591             }
592         }
593
594         uint16_t index = transition * NumberMaxOfInputs;
595         for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs; inputIndex++,
596 index++) {
597             if (AdjacencyList[index] == 0xFF) {
598                 break;
599             }
600             else {
601                 if (TokenVector[AdjacencyList[index]] < 1) {
602                     return FireError::PRODUCE_EXCEPTION;
603                 }
604             }

```

```

605     }
606
607     return FireError::NO_ERROR;
608 }
609
610 FireError PetriNet::isTriggerable() {
611     if (hasConditions) {
612         for (uint8_t transition = 0; transition < NumberOfTransitions;
613 transition++) {
614             if (FireVector[transition] > 0) {
615                 if (!isConditionSatisfied(transition)) {
616                     return FireError::CONDITIONS_ARE_NOT_APPLIED;
617                 }
618             }
619         }
620     }
621 }
622
623 for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
624     uint16_t fireQuantity = FireVector[transition];
625     uint16_t index = transition * NumberMaxOfInputs;
626
627     for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs;
628 inputIndex++, index++) {
629         if (AdjacencyList[index] == 0xFF) {
630             break;
631         }
632         else {
633             if (TokenVector[AdjacencyList[index]] < fireQuantity) {
634                 //Return the token vector to its initial value when
635 an exception occurs
636                 for (inputIndex--; inputIndex < 0; inputIndex--,
637 index--) {
638                     TokenVector[AdjacencyList[index]] +=
639 fireQuantity;
640                 }
641                 for (transition--; transition < 0; transition--) {
642                     fireQuantity = FireVector[transition];
643                     index = transition * NumberMaxOfInputs;
644
645                     for (uint8_t inputIndex = 0; inputIndex <
646 NumberMaxOfInputs; inputIndex++, index++) {
647                         if (AdjacencyList[index] == 0xFF) {
648                             break;
649                         }
650                         else {
651                             TokenVector[AdjacencyList[index]] -= fireQuantity;
652                         }
653                     }
654                 }
655             }
656             return FireError::PRODUCE_EXCEPTION;
657         }
658     }
659     else {
660         TokenVector[AdjacencyList[index]] -= fireQuantity;
661     }
662 }
663 }
664 }
665 }
666

```

```

667         for (uint8_t transition = 0; transition < NumberOfTransitions; transition++) {
668             uint16_t fireQuantity = FireVector[transition];
669             uint16_t index = transition * NumberMaxOfInputs;
670
671             for (uint8_t inputIndex = 0; inputIndex < NumberMaxOfInputs;
672 inputIndex++, index++) {
673                 if (AdjacencyList[index] == 0xFF) {
674                     break;
675                 }
676                 else {
677                     TokenVector[AdjacencyList[index]] += fireQuantity;
678                 }
679             }
680         }
681
682         return FireError::NO_ERROR;
683     }
684
685     //Private methods
686     void PetriNet::prepareMemoryStack() {
687         TokenVector = (uint16_t*)malloc(sizeof(uint16_t)*NumberOfPlaces);
688         FireVector = (uint16_t*)malloc(sizeof(uint16_t)*NumberOfTransitions);
689
690         uint16_t sizeOfAdjacencyList = (NumberMaxOfInputs + NumberMaxOfOutputs) *
691 NumberOfTransitions;
692         AdjacencyList = (uint8_t*)malloc(sizeof(uint8_t)* sizeOfAdjacencyList);
693         OutputListAuxiliarPointer = AdjacencyList + (NumberMaxOfInputs *
694 NumberOfTransitions);
695
696         for (uint8_t listIndex = 0; listIndex < sizeOfAdjacencyList; listIndex++) {
697             AdjacencyList[listIndex] = 0xFF;
698         }
699
700         if (hasConditions) {
701             uint8_t sizeOfConditions = (NumberOfTransitions + 7) / 8;
702             Conditions = (uint8_t*)malloc(sizeof(uint8_t)*sizeOfConditions);
703             for (uint8_t conditionsIndex = 0; conditionsIndex < sizeOfConditions;
704 conditionsIndex++) {
705                 Conditions[conditionsIndex] = 0xFF; //Set all conditions to true;
706             }
707         }
708     }
709
710     //Protected Methods
711     void PetriNet::print(char toPrint) {
712         platformInterface->print(toPrint);
713     }
714
715     void PetriNet::print(uint8_t toPrint) {
716         for (uint8_t base = 100; base >= 1; base = base / 10) {
717             uint8_t num = toPrint / base;
718             if (base == 1 || num != 0) {
719                 platformInterface->print('0' + num);
720             }
721             toPrint = toPrint % base;
722         }
723     }
724
725     void PetriNet::print(uint16_t toPrint) {
726         for (uint16_t base = 10000; base >= 1; base = base / 10) {
727             uint16_t num = toPrint / base;

```

```

729         if (base == 1 || num != 0) {
730             platformInterface->print('0' + num);
731         }
732         toPrint = toPrint % base;
733     }
734 }
735
736 void PetriNet::print(uint32_t toPrint) {
737     for (uint32_t base = 100000000; base >= 1; base = base / 10) {
738         uint32_t num = toPrint / base;
739         if (base == 1 || num != 0) {
740             platformInterface->print('0' + num);
741         }
742         toPrint = toPrint % base;
743     }
744 }
745
746 void PetriNet::print(char* toPrint) {
747     uint8_t counter = 0;
748
749     while (toPrint[counter] != '\0') {
750         platformInterface->print(toPrint[counter]);
751         counter++;
752     }
753 }

```

Pnrd.h

```

1  #ifndef PNRD_H
2  #define PNRD_H
3
4  #include "PetriNet.h"
5
6  #ifndef MAX_NUM_OF_TAG_HISTORY_ENTRIES
7  #define MAX_NUM_OF_TAG_HISTORY_ENTRIES 10
8  #endif
9
10 enum class PetriNetInformation {
11     TOKEN_VECTOR = 0,
12     ADJACENCY_LIST = 1,
13     FIRE_VECTOR = 2,
14     CONDITIONS = 3,
15     TAG_HISTORY = 4,
16     OTHER = 5
17 };
18
19 enum class ReadError {
20     NO_ERROR = 0,
21     TAG_NOT_PRESENT = 1,
22     INFORMATION_NOT_PRESENT = 2,
23     DATA_SIZE_NOT_COMPATIBLE = 3,
24     NOT_AUTORIZED = 4,
25     VERSION_NOT_SUPPORTED = 5,
26     ERROR_UNKNOWN = 6
27 };
28
29 enum class WriteError {

```

```

30         NO_ERROR = 0,
31         TAG_NOT_PRESENT = 1,
32         INFORMATION_NOT_SAVED = 2,
33         NOT_ENOUGH_SPACE = 3,
34         NOT_AUTORIZED = 4,
35         VERSION_NOT_SUPPORTED = 5,
36         ERROR_UNKNOWN = 6
37     };
38
39
40     enum class PnrdPolicy {
41         DEFAULT_SETUP = 0,
42         TAG_SETUP = 1
43     };
44
45     struct TagHistoryEntry {
46         uint16_t DeviceId = 0;
47         uint8_t Place = 255;
48         uint16_t Tokens = 0;
49         uint32_t TimeStamp = 0;
50     };
51
52
53     class Reader;
54
55     class Pnrd : public PetriNet {
56     private:
57         Reader* reader;
58         uint8_t dataInTag;
59
60         uint32_t tagId;
61         uint32_t deviceId;
62
63         TagHistoryEntry* tagHistory;
64         uint8_t tagHistoryIndex = 0;
65
66         bool hasTagHistory;
67
68
69     private:
70         void preparePnrdMemoryStack();
71
72         void printTagHistoryEntry(TagHistoryEntry entry);
73         void saveTagHistory();
74
75     public:
76         Pnrd(Reader* readerPointer, uint8_t num_places, uint8_t num_transitions, uint8_t
77 num_max_of_inputs, uint8_t num_max_of_outputs, bool hasConditions, bool
78 hasTagHistory);
79         Pnrd(Reader* readerPointer, uint8_t num_places, uint8_t num_transitions, uint8_t
80 num_max_of_inputs, uint8_t num_max_of_outputs);
81         Pnrd(Reader* readerPointer, uint8_t num_places, uint8_t num_transitions, bool
82 hasConditions, bool hasTagHistory);
83         Pnrd(Reader* readerPointer, uint8_t num_places, uint8_t num_transitions);
84         ~Pnrd();
85
86         void setAsTagInformation(PetriNetInformation info);
87         void setAsDeviceInformation(PetriNetInformation info);
88         bool isTagInformation(PetriNetInformation info);
89         uint8_t* getDataInTag();
90
91         ReadError getData();

```

```

92     WriteError saveData();
93
94     void setTagId(uint32_t tagId);
95     uint32_t getTagId();
96
97     void setDeviceId(uint32_t deviceId);
98     uint32_t getDeviceId();
99
100    bool setTagHistory(TagHistoryEntry* vector, uint8_t numberOfEntries);
101    uint8_t getTagHistory(TagHistoryEntry* vector);
102    TagHistoryEntry* getTagHistoryPointer();
103    uint8_t* getTagHistoryIndexPointer();
104    void printTagHistory();
105
106    void addTagHistoryEntry(TagHistoryEntry entry);
107    void removeLastTagHistoryEntry();
108
109    FireError fire();
110    FireError fire(uint8_t transition);
111
112    PnrdPolicy setup = PnrdPolicy::DEFAULT_SETUP;
113 };
114
115
116 class Reader {
117 public:
118     virtual ReadError read(Pnrd* pnrd) = 0;
119     virtual WriteError write(Pnrd* pnrd) = 0;
120 };
121 #endif

```

Pnrd.cpp

```

1  #ifndef PNRD_CPP
2
3  #define PNRD_CPP
4
5
6  #include "Pnrd.h"
7
8  //Constructors
9  Pond::Pond(Reader * readerPointer, uint8_t nameplates, uint8_t num_transitions,
10 uint8_t num_max_of_inputs, uint8_t num_max_of_outputs, bool hasConditions, bool
11 hasTagHistory) : PetriNet(num_places, num_transitions, num_max_of_inputs,
12 num_max_of_outputs, hasConditions) {
13     this->hasTagHistory = hasTagHistory;
14     reader = readerPointer;
15     preparePnrdMemoryStack();
16 }
17
18 Pond::Pond(Reader * readerPointer, uint8_t num_places, uint8_t num_transitions,
19 uint8_t num_max_of_inputs, uint8_t num_max_of_outputs) : PetriNet(num_places,
20 num_transitions, num_max_of_inputs, num_max_of_outputs) {
21     reader = readerPointer;
22     preparePnrdMemoryStack();
23 }
24

```

```

25 Pnrd::Pnrd(Reader * readerPointer, uint8_t num_places, uint8_t num_transitions, bool
26 hasConditions, bool hasTagHistory) : PetriNet(num_places, num_transitions,
27 hasConditions) {
28     this->hasTagHistory = hasTagHistory;
29     reader = readerPointer;
30     preparePnrdMemoryStack();
31 }
32
33 Pnrd::Pnrd(Reader* readerPointer, uint8_t num_of_places, uint8_t num_of_transitions)
34 :PetriNet(num_of_places, num_of_transitions) {
35     reader = readerPointer;
36     preparePnrdMemoryStack();
37 }
38
39 //Destructor
40 Pnrd::~Pnrd() {
41     if (hasTagHistory) {
42         free(tagHistory);
43     }
44 }
45
46 void Pnrd::setAsTagInformation(PetriNetInformation info) {
47     dataInTag |= 0b1 << ((int)info);
48 }
49
50 void Pnrd::setAsDeviceInformation(PetriNetInformation info) {
51     dataInTag &= ~(0b1 << ((int)info));
52 }
53
54 bool Pnrd::isTagInformation(PetriNetInformation info) {
55     return dataInTag & (0b1 << ((int)info));
56 }
57
58 uint8_t* Pnrd::getDataInTag() {
59     return &dataInTag;
60 }
61
62 ReadError Pnrd::getData() {
63     return reader->read(this);
64 }
65
66 WriteError Pnrd::saveData() {
67     return reader->write(this);
68 }
69
70 void Pnrd::setTagId(uint32_t tagId) {
71     this->tagId = tagId;
72 }
73
74 uint32_t Pnrd::getTagId() {
75     return tagId;
76 }
77
78 void Pnrd::setDeviceId(uint32_t deviceId) {
79     this->deviceId = deviceId;
80 }
81
82 uint32_t Pnrd::getDeviceId() {
83     return deviceId;
84 }
85
86 bool Pnrd::setTagHistory(TagHistoryEntry * vector, uint8_t numberOfEntries) {

```

```

87     bool noError = true;
88     if (numberOfEntries > MAX_NUM_OF_TAG_HISTORY_ENTRIES) {
89         numberOfEntries = MAX_NUM_OF_TAG_HISTORY_ENTRIES;
90         noError = false;
91     }
92
93     for (uint8_t index = 0; index < numberOfEntries; index++) {
94         tagHistory[index] = vector[index];
95     }
96
97     return noError;
98 }
99
100 uint8_t Pnrd::getTagHistory(TagHistoryEntry * vector) {
101     uint8_t index = 0;
102
103     for (uint8_t counter = tagHistoryIndex + 1; counter <
104 MAX_NUM_OF_TAG_HISTORY_ENTRIES; counter++) {
105         vector[index] = tagHistory[counter];
106         if (!(tagHistory[counter].Place == 0xFF)) {
107             vector[index] = tagHistory[counter];
108             index++;
109         }
110     }
111
112     for (uint8_t counter = 0; counter < tagHistoryIndex; index++, counter++) {
113         if (!(tagHistory[counter].Place == 0xFF)) {
114             vector[index] = tagHistory[counter];
115             index++;
116         }
117     }
118
119     return index;
120 }
121
122 TagHistoryEntry* Pnrd::getTagHistoryPointer() {
123     return tagHistory;
124 }
125
126 uint8_t* Pnrd::getTagHistoryIndexPointer() {
127     return &tagHistoryIndex;
128 }
129
130 void Pnrd::printTagHistory() {
131     for (uint8_t counter = tagHistoryIndex; counter > 0; counter--) {
132         printTagHistoryEntry(tagHistory[counter]);
133     }
134
135     if (tagHistory[0].Place != 0xFF) {
136         printTagHistoryEntry(tagHistory[0]);
137     }
138
139     for (uint8_t counter = MAX_NUM_OF_TAG_HISTORY_ENTRIES - 1; counter >
140 tagHistoryIndex; counter--) {
141         if (tagHistory[counter].Place == 0xFF) {
142             return;
143         }
144         printTagHistoryEntry(tagHistory[counter]);
145     }
146 }
147
148 void Pnrd::addTagHistoryEntry(TagHistoryEntry entry) {

```

```

149         if (tagHistoryIndex == MAX_NUM_OF_TAG_HISTORY_ENTRIES) {
150             tagHistoryIndex = 0;
151         }
152         else {
153             tagHistoryIndex++;
154         }
155
156         tagHistory[tagHistoryIndex] = entry;
157     }
158
159     void Pnrd::removeLastTagHistoryEntry() {
160         tagHistory[tagHistoryIndex].Place = 0xFF;
161
162         if (tagHistoryIndex == 0) {
163             tagHistoryIndex = MAX_NUM_OF_TAG_HISTORY_ENTRIES;
164         }
165         else {
166             tagHistoryIndex--;
167         }
168     }
169
170     FireError Pnrd::fire() {
171         FireError fireError = this->PetriNet::fire();
172         saveTagHistory();
173         return fireError;
174     }
175
176     FireError Pnrd::fire(uint8_t transition) {
177         FireError fireError = this->PetriNet::fire(transition);
178         saveTagHistory();
179         return fireError;
180     }
181
182     //Private Methods
183     void Pnrd::preparePnrdMemoryStack() {
184         if (hasTagHistory) {
185             tagHistory = (TagHistoryEntry*)malloc(sizeof(TagHistoryEntry)*
186 MAX_NUM_OF_TAG_HISTORY_ENTRIES);
187             for (uint8_t count = 0; count < MAX_NUM_OF_TAG_HISTORY_ENTRIES; count++)
188             {
189                 TagHistoryEntry entry;
190                 entry.Place = 0xFF;
191                 tagHistory[count] = entry;
192             }
193             tagHistoryIndex = 0;
194         }
195     }
196
197     void Pnrd::printTagHistoryEntry(TagHistoryEntry entry) {
198         print("Device Id: ");
199         print(entry.DeviceId);
200         print(". ");
201
202         print("Place: ");
203         print(entry.Place);
204         print(". ");
205
206         print("Tokens: ");
207         print(entry.Tokens);
208         print(". ");
209
210         print("Timestamp: ");

```

```

211         print(entry.TimeStamp);
212         print(".\n");
213     }
214
215     void Pnrd::saveTagHistory() {
216         if (hasTagHistory) {
217             for (uint8_t place = 0; place < getNumberOfPlaces(); place++) {
218                 if (TokenVector[place] > 0) {
219                     TagHistoryEntry entry;
220                     entry.Place = place;
221                     entry.DeviceId = getDeviceId();
222                     entry.Tokens = TokenVector[place];
223                     entry.TimeStamp = platformInterface->getTimeStamp();
224                     addTagHistoryEntry(entry);
225                 }
226             }
227         }
228     }
229
230 #endif

```

Pn532NfcReader.h

```

1  #ifndef PN532_NFC_READER_H
2
3  #define PN532_NFC_READER_H
4
5  #include <Arduino.h>
6  #include "Pnrd.h"
7
8  #include <PN532.h>
9  #include <NfcAdapter.h>
10
11 class Pn532NfcReader : public Reader {
12 private:
13     NfcAdapter* nfcAdapter;
14     ReadError getInformation(byte* payload, Pnrd* pnrd);
15     WriteError setInformation(byte* payload, Pnrd* pnrd);
16
17 public:
18     void initialize();
19
20     Pn532NfcReader(NfcAdapter* adapter);
21
22     ReadError read(Pnrd* pnrd);
23     WriteError write(Pnrd* pnrd);
24 };
25 #endif

```

Pn532NfcReader.cpp

```

1  #ifndef PN532_NFC_READER_CPP

```

```

2
3 #define PN532_NFC_READER_CPP
4
5
6 #include "Pn532nfcReader.h"
7
8 static const byte comparisonByteVector[] = { 'P','N','R','D' };
9 String comparisonString = "PNRD";
10
11 Pn532NfcReader::Pn532NfcReader(NfcAdapter* adapter) :Reader() {
12     nfcAdapter = adapter;
13 }
14
15 void Pn532NfcReader::initialize() {
16     nfcAdapter->begin();
17 };
18
19 ReadError Pn532NfcReader::read(Pnrd* pnrd) {
20     if (nfcAdapter->tagPresent()) {
21         NfcTag tag = nfcAdapter->read();
22         if (tag.hasNdefMessage()) {
23             NdefMessage message = tag.getNdefMessage();
24             int recordCount = message.getRecordCount();
25             NdefRecord record;
26
27             for (int recordNumber = 0; recordNumber < recordCount;
28 recordNumber++) {
29                 record = message.getRecord(recordNumber);
30
31
32
33                 if (record.getType().equals(comparisonString)) {
34                     int payloadLength = record.getPayloadLength();
35                     byte payload[payloadLength];
36                     record.getPayload(payload);
37
38                     int uidLength = tag.getUidLength();
39                     byte uid[uidLength];
40                     tag.getUid(uid, uidLength);
41                     //Stores only the two last hex values of the uid as
42 the tag id;
43                     pnrd->setTagId(uid[uidLength - 2]);
44
45                     return getInformation(payload, pnrd);
46                 }
47             }
48
49             return ReadError::INFORMATION_NOT_PRESENT;
50         }
51     }
52     else {
53         return ReadError::INFORMATION_NOT_PRESENT;
54     }
55
56 }
57
58 else {
59     return ReadError::TAG_NOT_PRESENT;
60 }
61
62 return ReadError::ERROR_UNKNOWN;
63 };

```

```

64
65
66 WriteError Pn532NfcReader::write(Pnrd* pnr) {
67     if (nfcAdapter->tagPresent()) {
68         NdefRecord record = NdefRecord();
69         int typeSize = 4;
70         NdefMessage writeMessage = NdefMessage();
71
72         record.setType(comparisonByteVector, typeSize);
73         uint16_t size = 4;
74
75         if (pnrd->isTagInformation(PetriNetInformation::TOKEN_VECTOR)) {
76             size += sizeof(uint16_t) * pnrd->getNumberOfPlaces();
77         }
78
79         if (pnrd->isTagInformation(PetriNetInformation::ADJACENCY_LIST)) {
80             size += 2;
81             size += (pnrd->getNumberMaxOfInputs() + pnrd->
82 >getNumberMaxOfOutputs()) * pnrd->getNumberOfTransitions();
83         }
84
85         if (pnrd->isTagInformation(PetriNetInformation::FIRE_VECTOR)) {
86             size += sizeof(uint16_t) * (pnrd->getNumberOfTransitions());
87         }
88
89         if (pnrd->isTagInformation(PetriNetInformation::CONDITIONS)) {
90             size += (pnrd->getNumberOfTransitions() + 7) / 8;
91         }
92
93         if (pnrd->isTagInformation(PetriNetInformation::TAG_HISTORY)) {
94             size += 2;
95             size += sizeof(TagHistoryEntry) * MAX_NUM_OF_TAG_HISTORY_ENTRIES;
96         }
97
98         if (size > 1024) {
99             return WriteError::NOT_ENOUGH_SPACE;
100         }
101
102         byte pointer[size];
103
104         setInformation(pointer, pnrd);
105
106         record.setPayload(pointer, size);
107         writeMessage.addRecord(record);
108
109         if (nfcAdapter->write(writeMessage)) {
110             return WriteError::NO_ERROR;
111         }
112         else {
113             free(pointer);
114             return WriteError::INFORMATION_NOT_SAVED;
115         }
116     }
117     else {
118         return WriteError::TAG_NOT_PRESENT;
119     }
120
121     return WriteError::ERROR_UNKNOWN;
122 };
123
124 ReadError Pn532NfcReader::getInformation(byte* payload, Pnrd * pnrd) {
125

```

```

126     uint16_t index = 0;
127
128     byte version = payload[index];
129     index++;
130
131     if (version != (byte)1) {
132         return ReadError::VERSION_NOT_SUPPORTED;
133     }
134
135     if (pnrd->setup == PnrdPolicy::TAG_SETUP) {
136         pnrd->getDataInTag()[0] = payload[index];
137         index++;
138     }
139     else {
140         if (payload[index] != pnrd->getDataInTag()[0]) {
141             return ReadError::INFORMATION_NOT_PRESENT;
142         }
143         index++;
144     }
145
146     uint8_t size = (uint8_t)payload[index];
147     index++;
148     if (size != pnrd->getNumberOfPlaces()) {
149         return ReadError::DATA_SIZE_NOT_COMPATIBLE;
150     }
151
152     size = (uint8_t)payload[index];
153     index++;
154     if (size != pnrd->getNumberOfTransitions()) {
155         return ReadError::DATA_SIZE_NOT_COMPATIBLE;
156     }
157
158     if (pnrd->isTagInformation(PetriNetInformation::TOKEN_VECTOR)) {
159         size = sizeof(uint16_t) * pnrd->getNumberOfPlaces();
160         for (uint16_t counter = 0; counter < size; counter++) {
161             ((byte*)pnrd->getTokenVectorPointer())[counter] = payload[index];
162             index++;
163         }
164     }
165
166     if (pnrd->isTagInformation(PetriNetInformation::ADJACENCY_LIST)) {
167         size = (uint8_t)payload[index];
168         index++;
169         if (size != pnrd->getNumberMaxOfInputs()) {
170             return ReadError::DATA_SIZE_NOT_COMPATIBLE;
171         }
172
173         size = (uint8_t)payload[index];
174         index++;
175         if (size != pnrd->getNumberMaxOfOutputs()) {
176             return ReadError::DATA_SIZE_NOT_COMPATIBLE;
177         }
178
179         size = sizeof(uint8_t) * (pnrd->getNumberMaxOfInputs() + pnrd-
180 >getNumberMaxOfOutputs()) * pnrd->getNumberOfTransitions();
181         for (uint16_t counter = 0; counter < size; counter++) {
182             ((byte*)pnrd->getAdjacencyListPointer())[counter] =
183 payload[index];
184             index++;
185         }
186     }
187

```

```

188     if (pnrd->isTagInformation(PetriNetInformation::FIRE_VECTOR)) {
189         size = sizeof(uint16_t) * pnrd->getNumberOfTransitions();
190         for (uint16_t counter = 0; counter < size; counter++) {
191             ((byte*)pnrd->getFireVectorPointer())[counter] = payload[index];
192             index++;
193         }
194     }
195
196     if (pnrd->isTagInformation(PetriNetInformation::CONDITIONS)) {
197         size = sizeof(uint8_t) * (pnrd->getNumberOfTransitions() + 7) / 8;
198         for (uint16_t counter = 0; counter < size; counter++) {
199             ((byte*)pnrd->getConditionsPointer())[counter] = payload[index];
200             index++;
201         }
202     }
203
204     if (pnrd->isTagInformation(PetriNetInformation::TAG_HISTORY)) {
205         uint8_t* tagIndex = pnrd->getTagHistoryIndexPointer();
206
207         size = sizeof(uint8_t);
208         for (uint16_t counter = 0; counter < size; counter++) {
209             ((byte*)tagIndex)[counter] = payload[index];
210             index++;
211         }
212
213         size = sizeof(TagHistoryEntry) * MAX_NUM_OF_TAG_HISTORY_ENTRIES;
214         for (uint16_t counter = 0; counter < size; counter++) {
215             ((byte*)pnrd->getTagHistoryPointer())[counter] = payload[index];
216             index++;
217         }
218     }
219
220     return ReadError::NO_ERROR;
221 }
222
223 WriteError Pn532NfcReader::setInformation(byte* payload, Pnrd *pnrd) {
224     uint8_t version = 1;
225     uint16_t index = 0;
226     uint16_t size = 0;
227
228
229     payload[index] = version;
230     index++;
231
232     payload[index] = pnrd->getDataInTag()[0];
233     index++;
234     payload[index] = pnrd->getNumberOfPlaces();
235     index++;
236     payload[index] = pnrd->getNumberOfTransitions();
237     index++;
238
239     if (pnrd->isTagInformation(PetriNetInformation::TOKEN_VECTOR)) {
240         size = sizeof(uint16_t) * pnrd->getNumberOfPlaces();
241         for (uint16_t counter = 0; counter < size; counter++) {
242             payload[index] = ((byte*)pnrd->getTokenVectorPointer())[counter];
243             index++;
244         }
245     }
246
247     if (pnrd->isTagInformation(PetriNetInformation::ADJACENCY_LIST)) {
248         payload[index] = pnrd->getNumberMaxOfInputs();
249         index++;

```

```

250         payload[index] = pnr->getNumberMaxOfOutputs();
251         index++;
252         size = sizeof(uint8_t) * (pnr->getNumberMaxOfInputs() + pnr->
253 >getNumberMaxOfOutputs()) * pnr->getNumberOfTransitions();
254         for (uint16_t counter = 0; counter < size; counter++) {
255             payload[index] = ((byte*)pnr->
256 >getAdjacencyListPointer())[counter];
257             index++;
258         }
259     }
260
261     if (pnr->isTagInformation(PetriNetInformation::FIRE_VECTOR)) {
262         size = sizeof(uint16_t) * pnr->getNumberOfTransitions();
263         for (uint16_t counter = 0; counter < size; counter++) {
264             payload[index] = ((byte*)pnr->getFireVectorPointer())[counter];
265             index++;
266         }
267     }
268
269     if (pnr->isTagInformation(PetriNetInformation::CONDITIONS)) {
270         size = (pnr->getNumberOfTransitions() + 7) / 8;
271         for (uint16_t counter = 0; counter < size; counter++) {
272             payload[index] = ((byte*)pnr->getConditionsPointer())[counter];
273             index++;
274         }
275     }
276
277     if (pnr->isTagInformation(PetriNetInformation::TAG_HISTORY)) {
278         uint8_t* tagIndex = pnr->getTagHistoryIndexPointer();
279         byte* pointer = (byte*)pnr->getTagHistoryPointer();
280
281         payload[index] = tagIndex[0];
282         index++;
283         size = sizeof(TagHistoryEntry) * MAX_NUM_OF_TAG_HISTORY_ENTRIES;
284         for (uint16_t counter = 0; counter < size; counter++) {
285             payload[index] = pointer[counter];
286             index++;
287         }
288     }
289
290     return WriteError::NO_ERROR;
291 }
292 #endif

```