Instituto Federal Catarinense
Bacharelado em Engenharia de Controle e Automação
Campus Luzerna

João Peterson Scheffer

Desenvolvimento de um método e sistema para compilação e simulação de redes de petri para utilização em controladores lógicos industriais

Luzerna
Dezembro de 2022

João Peterson Scheffer

Desenvolvimento de um método e sistema para compilação e simulação de redes de petri para utilização em controladores lógicos industriais

Trabalho de conclusão apresentado à banca examinadora do curso de Bacharelado em Engenharia de Controle e Automação, do Instituto Federal Catarinense, Campus Luzerna, como requisito para a obtenção do título de Bacharel em Engenharia de Controle e Automação, em cumprimento às exigências de sua componente curricular. Orientador: Prof. Msc. Tiago Javorani Prati

Luzerna
Dezembro de 2022

João Peterson Scheffer

Desenvolvimento de um método e sistema para compilação e simulação de redes de petri para utilização em controladores lógicos industriais

Trabalho de conclusão apresentado à banca examinadora do curso de Bacharelado em Engenharia de Controle e Automação, do Instituto Federal Catarinense, Campus Luzerna, como requisito para a obtenção do título de Bacharel em Engenharia de Controle e Automação, em cumprimento às exigências de sua componente curricular.

Luzerna (SC), 08 de dezembro de 2022:

Prof. Msc. Tiago Javorani Prati
Instituto Federal Catarinense

BANCA EXAMINADORA

Prof. Msc. XXXXXXX
Instituto Federal Catarinense

Prof. Dr. XXXXXX
Instituto Federal Catarinense

Agradecimentos

Deixo meus agradecimentos primeiramente aos meus pais, Paulo Roberto Scheffer e Maristela Sluzalla Scheffer, e minha família, que me incentivaram, apoiaram e me custodiaram durante minha formação, espero que estejam orgulhosos de mim. Agradecimento aos meus amigos, colegas e corpo do IFC Luzerna, que me deram durante minha formação um ambiente não só de educação e desenvolvimento intelectual mas também de amadurecimento, onde pude me desenvolver como pessoa e ser quem sou hoje, resguardo assim um carinho imenso por esta época da minha vida e das pessoas que me acompanharam.

Agradeço ao professor Thiago Javorani Prati, meu orientador, que em aula me apresentou aos conceitos principais sobre rede de petri e sua utilização em meio industrial, conceitos quais tomei gosto e agora culminam-se nesta obra.

Resumo

O presente trabalho propõe o desenvolvimento de um método e sistema para a simulação/execução de redes de petri e compilação para código de lista de instrução para controladores lógicos programáveis. As redes de petri são uma poderosa ferramenta para modelar e automatizar sistemas industriais, permitindo representar o comportamento de diferentes processos e eventos de forma eficiente e complementar aos métodos de programação tradicionais. No entanto, a utilização dessas redes em *software*, com ênfase em controladores lógicos programáveis, é geralmente feita de forma manual, sendo o *design* da rede realizada em uma etapa e a posteriormente convertida para *software*. Portanto, há espaço e demanda para criação de ferramentas e métodos para utilização dessa tecnologia de forma mais simples e integrada.

A base para implementação dessa tecnologia é um sistema capaz de representar e executar essas redes. Portanto, é proposto o desenvolvimento de uma biblioteca em linguagem C que apresenta as funções de representação, checagem, simulação/execução e armazenamento de redes de petri, podendo ser usada em computadores *desktop* e sistemas embarcados.

Para o uso em controladores lógicos programáveis, propõem-se um algoritmo de compilação que traduz as especificações de uma rede de petri na biblioteca C para um conjunto de instruções adequadas para o controlador lógico industrial, nesse trabalho em específico, o controlador WEG TPW04. A compilação permite que a rede seja executada diretamente pelo controlador, garantindo a funcionalidade desejada e uma integração mais trivial com o ambiente industrial.

Espera-se que o método e sistema desenvolvidos neste trabalho contribuam significativamente para o avanço da aplicação prática de redes de petri em meio industrial, provendo uma ferramenta complementar a outros métodos para os desenvolvedores de aplicações industriais e *software* em geral.

Palavras-chave: Redes de petri; Automação industrial; PLC; Linguagem C.

Abstract

The following work presents the development of a method and system for simulating/executing petri nets and compilation to instruction list code for usage on programmable logic controllers. Petri nets are a powerful tool for modeling and automating industrial systems, enabling the representation of the behavior for different processes and events in an efficient manner and complimentary to tradicional programming methods. But, the use of these nets in software, with emphasis on programmable logic controllers, is generally made by hand, in which the petri net is designed first and then converted to software. Therefore, there is space and demand for the conception of tools and methods for the utilization of this technology in a more simple and integrated manner.

The base for this implementation is a system capable of representing and executing these nets. Therefore, it is proposed the development of a C library that features the representation, checking, execution and storage of petri nets, capable of being used in desktop and embedded systems applications.

For usage with programmable logic controllers, it is proposed an compilation algorithm that translates the specifications of a given petri net in the C library to a set of instructions for the programmable logic controller, in specific for this implementation, the WEG TPW04 controller. The compilation enables the controller to execute the petri net directly, granting the desired functionality and a more trivial integration with the industrial environment.

It is hoped that the method and system developed in this work contributes significantly for the advance of the practical use of petri nets in industrial environments, providing a tool that is complementary to other programming methods for the development of industrial applications as well as in general software.

Keywords: Petri nets; Industrial automation; PLC; C language.

Lista de ilustrações

Lista de códigos

Lista de abreviaturas e siglas

PLC	*Programmable logic controller*.
CLP	Controlador lógico programável.
WYSIWYG	*What you see is what you get*.
Vscode	*Visual studio code*.
build	Processo de compilação em lote de arquivos fonte para criação de um programa executável ou biblioteca.
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos.
POSIX	Conjunto de padrões definidos pela IEEE que especifica uma interface comum para sistemas operacionais Unix-like, visando garantir a portabilidade de aplicativos entre diferentes plataformas.
Desktop	Computadores de uso pessoal como computadores de mesa e Notebooks's.
Bindings	Definições de funções, tipos e outras estruturas programáticas que mapeiam diretamente com suas contrapartidas em outro ambiente/-sistemas/linguagem de programação. Efetivamente é uma definição de intemperabilidade para que linguagens possam reutilizar código escrito em outras linguagens de programação, ou ainda outros ambientes de execução de código.
Framework	Conjunto de ferramentas base que auxiliam no desenvolvimento de aplicações.
Plugin	Conjunto de códigos que estendem a funcionalidade de um programa, como uma biblioteca.
Repositório	Neste contexto se trata do armazenamento de um projeto de *software* com seus arquivos fonte de forma versionada
Commit	Uma instância de alteração de código no versionamento de um projeto.
callback	Função programática que é executada automaticamente após uma ação ser concluída sem que o usuário chame a manualmente.
array	Do inglês, uma lista indexada de valores.

thread — Do inglês, uma linha, que representa uma linha de execução de código paralela/concorrente a execução normal de um programa.

thread safe — Representa código que é capaz de ser executado de forma paralela sem que hajam condições de acesso concorrente e possibilidade de corrupção ou alteração indevida de de memória.

mutex — Conceito que representa a capacidade de uma instância de executar código paralelo e é requisitado e liberado para diversas instâncias concorrentes, quem possui o *mutex*, pode executar, quem não possui não executa, criando um jogo de execução em turnos.

lista ligada — Do inglês, lista conectada. Uma lista onde elementos não possuem posição numérica, mas são ligados uns aos outros via um ponteiro de referência para o próximo elemento da lista.

buffer — Espaço utilizado para manipulação intermediária de algo entre o início e fim de u processo. Um armazenamento temporário.

Sumário

1 Introdução

Quando se trata de sistemas dinâmicos como um todo há sempre um interesse de criar, entender, prever e controlar estes sistemas, e no âmbito de sistemas discretos orientados a eventos [1], como alguns sistemas industriais de automação de interesse neste trabalho, há várias metodologias, práticas, tecnologias e linguagens de programação capazes de trabalhar estes sistemas de forma a alcançar resultados esperados, cada qual possuí vantagens e desvantagens.

No âmbito industrial, para realizar o controle e automação de processos, comumente são utilizados controladores lógicos programáveis, PLC's, e estes podem ser programados para realizar as funções desejadas e para isso empregam a utilização de linguagens de programação como:

- *Ladder*, que representa um processo linear de forma visual inspirada em lógica de contatos [2].
- Lista de instrução, que é uma linguagem escrita, não visual, e que representa um fluxo de operações com base em comandos de texto.
- Grafcet, uma linguagem visual que representa um processo em forma de fluxo com base em passos de um nó para outro nó do processo, onde cada nó é uma instrução de ação e cada passo é dado conforme uma transição atrelada a um evento.

Ainda há outras tecnologias que possuem outras formas abstratas e abordagens diferentes, sendo a linguagem *Ladder* um exemplo de aproveitamento de conhecimento e simplicidade, porém há casos onde certas ferramentas não apresentam melhor desempenho e eficiência dado certos tipos de especificações, em especial no caso do *Ladder*, onde controle de estado [3] e paralelismo [4] são conceitos difíceis de serem implementados.

Para resolver alguns destes problemas, condições e situações, como as mencionadas, emprega-se o uso de linguagens de modelagem lógica capazes de modelar um sistema desejado, onde pode-se empregar então métodos e técnicas para alcançar o resultado desejado. Análise, simulação e supervisão do sistemas são três características desejadas.

---

[1] Sistemas que trabalham com valores discretos como ligado e desligado por exemplo, e possuem mudança de estado conforme eventos também discretos.

[2] Uma forma simples de programação baseada na representação visual de lógica de contato, como vista em diagramas com relés e contatoras.

[3] Armazenamento e lógica do estado/situação atual de um processo

[4] Capacidade de unir dois fluxos de trabalho com razões de trabalho diferentes em um ponto definido.

Neste contexto há várias linguagens de modelagens lógicas, como autômatos, máquinas de *Moore*, dentre outras linguagens lógicas, que podem ser usadas para abstrair alguns conceitos como os citados anteriormente e nesse contexto apresenta-se então a rede de Petri (IZHIKEVICH, 2011), que é capaz de reproduzir conceitos abstratos como paralelismo, sincronia, mutualidade exclusiva, etc., de forma simples, bem como vários outros conceitos comuns de lógica e aritmética e conceitos ainda exclusivos, intrínsecos a si própria.

As redes de Petri são um instrumento de maior formalidade e podem representar conceitos abstratos, fazendo da mesma uma ótima ferramenta para automação e controle de sistemas a eventos discretos, e que cada vez mais vem sendo estudada em meio acadêmico quanto a sua utilização como mecanismo de modelagem de processos, modelagem de sistemas criticos (LEVESON; STOLZY, 1987) (GHEZZI et al., 1991), e prevenção de *deadlocks* [5] (KAID et al., 2015).

Visto o grau de interesse acadêmico, utilidade industrial e dada sua ótima representação de processos e liberdade de abstração, pode-se dizer que há interesse prático em redes de petri, porém o emprego real deste tipo de tecnologia é mínimo. Existem metodologias (MOREIRA; BASILIO, 2014) que por exemplo, propõem a funcionalidade de compilação de redes de petri para código *Ladder*, usado então em automação industrial em PLC's.

Esse tipo de função é implementada em projetos como *PetriLab* (SOUZA, 2015), porém não trazem integração real, uma boa experiência de programação ou mesmo boas práticas de design de *software*. Essas características são valorizadas e por vezes indispensáveis para o processo de desenvolvimento, integração, teste, manutenção e melhoria contínua de sistemas e controle de sistemas em meio industrial. Visto essas preocupações, há necessidade de ferramentaria adequada e atrativa para as empresas e desenvolvedores destes sistemas industriais, bem como a difusão da utilização de redes de petri como ferramenta de uso de automação industrial e *software* em geral.

1.1 Justificativa

Sendo apresentado o estado atual de disseminação de utilização de redes de petri, tanto geral como industrial, justifica-se a criação de ferramentaria necessária ao desenvolvimento de aplicações. Aplicações tais que implementam necessariamente formas de representação e execução destas redes, e ainda a funcionalidade de transformar estas redes em programas prontos para utilização em meio industrial.

[5] Singularidade em processos onde o sistema entra em um ponto de parada de forma imprevista e não possuí forma de autocorreção, permanece ou parado em falha, ou em repetição contínua de uma única instrução.

A capacidade de transformar tais redes em programas industriais, compreende-se como o processo de compilação, onde a rede de petri é traduzida através de um algoritmo proposto para o resultado de saída, sendo esta saída código puro, arquivos digitais ou programas prontos para utilização com PLC's industriais.

1.2 Objetivos

1.2.1 Objetivo geral

Desenvolvimento de um ambiente de definição, simulação e compilação e armazenamento de rede de petri voltadas para utilização em meio industrial, PLC's.

1.2.2 Objetivos específicos

- Desenvolvimento de uma biblioteca implementada em linguagem C que deve implementar os seguintes pontos:
  - Estrutura de dados.
  - serialização de dados para armazenamento.
  - Capacidade de checagem e validação.
  - Capacidade de execução normal e temporizada de forma assíncrona.
- Desenvolvimento de algoritmos de compilação de redes de petri para os seguintes alvos:
  - Lista de instrução, em formato de texto para a referência PLC WEG TPW04.

2 Fundamentação teórica

Nesta introdução teórica, será abordado o funcionamento básico das redes de petri e alguns dos principais tipos existentes. É importante compreender esses conceitos fundamentais antes de prosseguir para a implementação de *software*, pois definirá a ideia de funcionamento e implementação, onde poderá se observar a utilidade das redes, possíveis abstrações e comportamentos desejados, como os de interesse abordados na seção de introdução. Também conceitos relacionados a implementação de *software* e outros conceitos adjacentes necessários ao entendimento do desenvolvimento das partes integrantes do trabalho.

2.1 Redes de petri

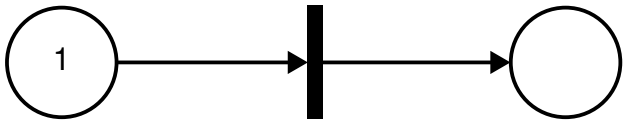
As redes de petri são uma poderosa ferramenta para a modelagem e análise de sistemas concorrentes e paralelos. Introduzidas por Carl Adam Petri em 1962 (PETRI, 1962), as redes de petri fornecem uma representação gráfica e formal para descrever a dinâmica de sistemas complexos, permitindo a análise de propriedades importantes, como comportamento temporal, concorrência e paralelismo.

Uma rede de petri é composta por lugares, transições, fichas e arcos. Cada componente tem um papel fundamental na representação e modelagem do sistema.

- Lugares: Os lugares representam estados ou condições do sistema. Eles são representados por círculos em um diagrama de rede de petri. Os lugares podem conter fichas, que são unidades discretas que refletem o estado atual do sistema.
- Transições: As transições representam eventos ou ações que podem ocorrer no sistema. Elas são representadas por retângulos em um diagrama de rede de petri. Para que uma transição seja disparada, é necessário que todos os lugares de entrada estejam marcados com pelo menos uma ficha.
- Fichas: As fichas são unidades discretas que se movem entre os lugares em resposta à ocorrência das transições. Elas refletem o estado atual do sistema e representam o fluxo de controle. Quando uma transição é disparada, ela consome as fichas dos lugares de entrada e produz novas fichas nos lugares de saída.
- Arcos: Os arcos conectam os lugares às transições e vice-versa. Eles indicam as relações de dependência entre os elementos do sistema. Existem dois tipos de arcos: arcos de entrada, que conectam lugares a transições, indicando que os lugares são precondições para a ocorrência da transição, e arcos de saída,

que conectam transições a lugares, indicando que as transições produzem fichas nos lugares de saída. Estes arcos podem ainda conter pesos, desta forma, um arco de entrada pode retirar mais de uma ficha de um lugar, ou um arco de saída pode inserir várias fichas em outro lugar.

Figura 1 – Rede de petri com dois lugares e uma transição



O funcionamento de uma rede de petri ocorre através de um processo chamado de disparo. Quando todas as condições de precondições de uma transição são satisfeitas, a transição é disparada, consumindo as fichas dos lugares de entrada e produzindo fichas nos lugares de saída correspondentes. Esse processo de disparo ocorre de forma determinística, onde as transições só podem ser disparadas quando todas as condições são atendidas.

Em mais detalhes, as fichas só se movem de um lugar ao outro caso uma transição aconteça, e ela só irá acontecer se o arco for atendido, por exemplo, o arco de peso de saída só irá permitir que a transição ocorra caso a quantidade de fichas no lugar de origem for maior ou igual ao peso, efetivamente colocando uma condição de disparo. Quando todas as condições atreladas a uma transição são atendidas, tal transição diz se sensibilizada.

2.1.1 Tipos de Redes de petri

Existem vários tipos de redes de petri, cada um com suas características e aplicabilidades específicas. Alguns dos principais tipos são:

- Redes de petri coloridas (JENSEN, 1997): Nesse tipo de rede, além das fichas, são utilizadas cores para representar diferentes propriedades ou atributos dos elementos do sistema. Isso permite uma modelagem mais expressiva, onde as cores das fichas podem influenciar o comportamento das transições.

- Redes de petri temporizadas (ZHOU; HRúZ, 2007): Nesse tipo de rede, são adicionadas informações de tempo às transições e arcos. Essas informações podem incluir atrasos, tempo de execução ou intervalos de tempo específicos para a ocorrência de eventos. Isso possibilita a análise de propriedades temporais e a simulação de sistemas baseados em tempo.

- Redes de petri estocásticas (HILLSTON, 2009): Nas redes de petri estocásticas, são incorporadas probabilidades às transições e arcos, permitindo a modelagem de sistemas com comportamento probabilístico. Essas redes são úteis para a análise de sistemas em que a ocorrência de eventos é incerta ou aleatória.

- Redes de petri priorizadas (ZHOU; HRúZ, 2007): Redes onde as transições tem a propriedade de serem priorizadas para disparo em relação ao outras, úteis em caso de disparo simultâneo de uma ou mais transições.

- Redes de petri de alto nível (ISO, 2000): As redes de petri de alto nível são uma extensão das redes de petri tradicionais, que permitem uma representação mais abstrata e simplificada dos sistemas. Elas são úteis para modelar sistemas complexos e lidar com a explosão combinatória que pode ocorrer em redes de

petri tradicionais. Abstrações como lugares compartilhados, macro transições e hierarquia de sub-redes são utilizadas para simplificar a representação e análise desses sistemas.

Esses são apenas alguns dos principais tipos de redes de petri, e cada um deles oferece recursos e abordagens específicas para a modelagem e análise de sistemas. O uso correto e adequado desses tipos de redes de petri depende das características do sistema a ser modelado e das propriedades que se deseja analisar.

2.1.2 Delimitação de uma rede para uso industrial

As redes de petri são particularmente interessantes, pois apresentam um bom grau de flexibilidade e capacidade de abstração, podendo comportar lógicas completas, simples e robustas, então é de interesse delimitar o tipo de rede que deverá ser trabalhada de forma que possamos ter as funcionalidades e características para modelagem de sistemas, como preferência para sistemas de automação industrial.

O *software* Petrilab (SOUZA, 2015) abre o caminho com uma delimitação interessante e prática para uso industrial, onde se propõe a utilização de métodos de entrada e saída para rede de petri, condições lógicas que permitem o acionamento de transições e os tipos de arcos desejados, sendo eles os de peso e arcos negados. Baseado neste trabalho podemos construir e reavaliar algumas decisões sobre a definição da rede de petri.

Particularmente no uso industrial a utilização de temporização é indispensável para qualquer tipo de processo, então esse é um pré-requisito inegociável. Em nome da implementação multi plataforma e considerando também precisão, será definido que as transições temporizadas trabalhem com milissegundos em vez de microssegundos. Um exemplo de transição temporizada pode ser visto na figura 2.

Os arcos com peso são úteis, porém é fácil encontrar situações onde algumas abstrações se fazem convenientes, como no caso de realizar a retirada de todos as fichas de um lugar de uma só vez, um comedor de fichas, figura 3, cujo trabalho é exatamente este. Conforme a lógica desejada, tal comportamento pode ser também atrelado a outro tipo de arco, o arco de *reset* (reinicialização) (DUFOURD; FINKEL; SCHNOEBELEN, 1998), não presente na implementação do Petrilab, onde após o disparo de uma transição o mesmo irá remover todos as fichas do lugar atrelado, sendo essa relação não direcional, como nos arcos de peso. Um exemplo de arco de *reset* pode ser visto na figura 4.

Ainda há mais um tipo de arco de interesse, por vezes pode ser necessário a criação de lógicas de exclusão mútua, ou de verificação simples, podendo vir a utilizar um arranjo lógico de lugares e transições dedicado, tendo a função de detectar quando não há fichas em lugar, comportamento que pode ser alcançado com um único arco,

Figura 2 – Rede de petri com transição temporizada

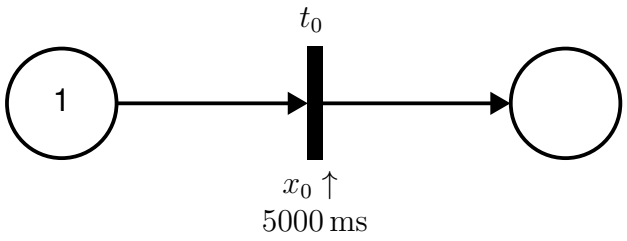
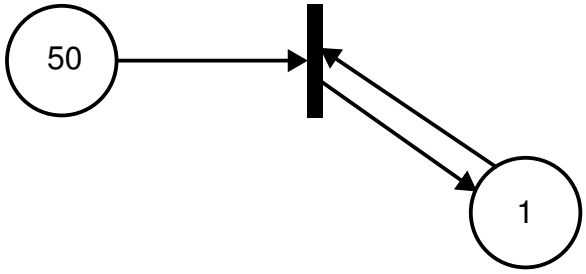


Figura 3 – Configuração comedor de fichas



o arco de negado, também chamado de arco inibidor, que somente permite o disparo de uma transição quando não há fichas em um determinado lugar.

Tal arco permite abstrações mais simples mas também possibilita que nossa rede de petri seja computacionalmente completa (ZAITSEV, 2014), ou seja, teoricamente ela pode vir a computar qualquer problema, propriedade útil quando desejamos que nossa rede de petri seja capaz de modelar qualquer tipo de sistema visando a automação industrial, e também que isso seja feito de forma conveniente.

Ainda para o uso industrial devemos nos perguntar como podemos interagir com a rede de petri em *software* com o processo/sistema atrelado, para isso necessitamos de entradas e saídas. As entradas devem ser atreladas as transições, pois são estas

Figura 4 – Arco de *reset*

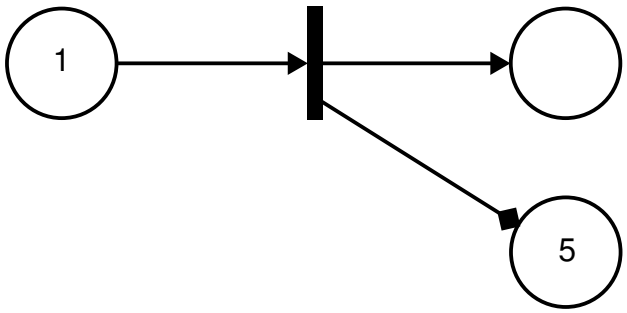
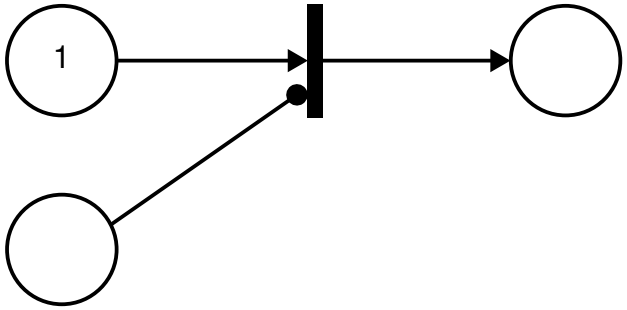


Figura 5 – Arco negado, ou inibidor



que movem o estado da rede.

A forma como as entradas são atreladas é baseada no seguinte fato, se uma rede de petri é disparada continuamente, em tempo real, toda transição é sempre executada quando possível, e se atrelarmos o estado binário de uma entrada à capacidade de disparo de uma transição, a mesma pode ser levada ao disparo várias vezes por ciclo, podendo ocasionar um comportamento indesejado, portanto se propõe a utilização de eventos instantâneos para ativação. No caso de entradas discretas, estes eventos são as bordas, de subida ou descida, que acontecem quando a entrada muda de estado e duram por apenas um ciclo de execução, garantindo disparos únicos de transições, sendo na subida, descida, ou ambos, da entrada.

No Petrilab são dadas como entradas também, de certa forma, condições lógicas não baseadas em eventos. Na implementação aqui proposta, iremos descartar essas condições em nome da simplicidade, pois esse tipo de funcionalidade pode ser alcançado por uma abstração com alguns lugares e transições extras, ficando a critério do autor da rede.

As saídas podem se referir ao estado interno da rede, ou seja, as fichas nos lugares, então é de livre escolha a forma como podemos retirar a informação da rede de como acionar as saídas. Propõe-se a utilização de comparações do tipo maior ou igual entre um número de fichas e a quantidade de fichas em determinado lugar, tal condição é binária e será atrelada a uma saída externa.

Nada impede também a utilização da própria quantidade de fichas seja a saída, caso deseje se uma saída numérica, que pode ser acessada diretamente do lugar da rede de petri.

2.1.3 Implementação de redes de petri em *software*

Não nos atrelando a definição formal matemática das redes de petri, é necessário abordar a representação dessas redes, de maneira computacional na forma de *software*, e existem dois jeitos clássicos de realizar essa representação, de forma relacional ou matricial.

A forma relacional trata lugares e transições como vértices de um gráfico direcionado, e em código há uma lista de lugares e transições e uma lista de relações, onde cada relação é atrelada a um lugar e transição e possui metainformação sobre o tipo de relação, como o próprio tipo, que pode ser um arco direcionado de peso, um arco negado ou de *reset*. Esse tipo de representação é mais compreensível e intuitiva, porém pode sofrer impactos de desempenho, pois para realização de verificações e disparos da rede é necessário percorrer estas listas para cada tipo de operação distinta.

A outra forma, matricial, declara-se uma matriz de arcos, onde as linhas são os lugares, e as colunas as transições, e o valor em determinada posição determina a quantidade de fichas a serem movidas, como no caso de arcos de peso. Este tipo de implementação é mais simples e rápida, pois não há necessidade de percorrer as relações, que é uma operação de tempo linear, porém a representação matricial cresce quadraticamente conforme mais lugares e transições são atrelados, aumentando consumo de memória e armazenamento.

Dado que o tamanho de redes de petri não são usualmente grandes, e que há prioridade de processamento sobre memória, será escolhida a representação matricial para implementação em *software* da rede de petri proposta. Tal representação acomoda tranquilamente os tipos de arcos propostos, bem como temporização e de entrada e saída.

Código 1 – Inicialização de uma rede de petri de forma matricial

```
pnet_t *pnet = pnet_new(
    pnet_arcs_map_new(1,2,
        -1,
            0
    ),
    pnet_arcs_map_new(1,2,
            0,
            1
    ),
    NULL,
    NULL,
    pnet_places_init_new(2,
        1, 0
    ),
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
);
```

Fonte: Do autor.

2.1.4 Limitações e comportamento

Seja qual for o paradigma de implementação escolhido, motor de execução a ser implementado, há algumas situações a serem abordadas, detalhes de implementação que irão afetar o comportamento geral de execução da rede e que devem ser delimitados e abordados de forma a garantir equivalência das implementações.

2.1.4.1 Disparos simultâneos

Quando duas transições estão sensibilizadas simultaneamente, no momento do disparo qual deve ser o comportamento? Há algumas formas de pensar sobre tal dilema. Uma possível solução seria a utilização de rede de petri priorizadas em que uma transição é priorizada sobre outra, garantindo assim que apenas umas delas dispare por vez.

Outra maneira é garantir a propriedade de unicidade, ou seja, que apenas uma transição seja executada por vez. Há duas formas de garantir isso, permitindo que apenas uma transição seja sensibilizada por vez, ou que várias sejam sensibilizadas mas apenas o disparo de uma seja efetuado. Visto que será realizada a implementação de transições temporizadas, deve-se pensar que várias transições podem ser

marcadas como sensibilizadas a fim de serem temporizadas e disparadas posteriormente, como discutido na seção 2.1.5.

Portanto, irá se permitir a sensibilização de várias transições, mas apenas uma poderá ser disparada por vez em um ciclo de execução. O processamento dos disparos será parado após o primeiro disparo, que será realizado na transição sensível de menor índice, o que pode por vezes causar ambiguidade sobre qual transição será disparada antes.

Pelo fato de desejarmos que a rede seja temporizada, de entrada/saída e de ser executada em tempo real, diminui-se naturalmente a probabilidade que haja colisão de disparo entre uma ou mais transições ao mesmo tempo, devido às entradas virem de um sistema físico real em tempo real e de algumas transições serem temporizadas.

Assim, deixa se um aviso de ambiguidade para este tipo de implementação, pois mesmo que que seja garantido o disparo de uma só transição, eliminando o dilema do disparo mútuo, ainda pode-se ocorrer ambiguidade sobre qual transição será disparada na rede, porém menos comum, dado o tipo de rede trabalhada. Fica assim à critério do autor da rede de petri garantir que, para que uma transição seja priorizada em uma situação de ambiguidade, que a mesma seja de menor índice ou que se empregue um mecanismo auxiliar com outros lugares e transições para garantir o comportamento desejado.

2.1.4.2 Entradas

Como discutido anteriormente, as entradas são dadas como eventos atrelados a transições, e com a mesma preocupação de disparos simultâneos existem dois casos a serem considerados, transições com múltiplos eventos de entrada, e várias transições que utilizam a mesma entrada.

Para fins de simplificação será reforçado que cada transição seja atrelada unicamente a uma entrada e vice-versa, garantindo que disparos simultâneos causados pela mesma entrada nunca ocorram. Caso a mesma entrada seja atrelada a várias transições, pode haver ambiguidade, pelo fato das transições dependerem do mesmo evento no mesmo ciclo de execução.

Casos onde deseja-se tal comportamento, pode-se fácil e seguramente ser implementado utilizando múltiplas transições e lugares auxiliares, deixando assim essa abstração de responsabilidade do autor da rede de petri.

2.1.5 Temporização

As transições temporizadas funcionam como retardos de tempo. Dado uma transição que depende de um evento de entrada e de arcos condicionais, como arcos de peso e arcos negados, está começará a contar a partir do momento que todas as condições forem satisfeitas. A contagem é realizada até o tempo determinado, onde

a transição será disparada somente se as condições dos arcos ainda permanecem verdadeiras. Se as condições permitirem e a transição for sensibilizada novamente, a contagem atual deve continuar normalmente até que a transição tenha chance de tentar disparar, momento em que será liberada para iniciar a contagem novamente se requerido.

As transições não temporizadas devem ser disparadas de forma individual, já as temporizadas podem começar a contar de forma simultânea, porém, devem ser disparadas também de forma individual. Em caso de conflito, da mesma forma que para as transições normais, a transição de menor índice nas matrizes receberá prioridade.

O razão dessa definição tem como base garantir o funcionamento correto das transições temporizadas nos seguintes casos:

- Transições temporizadas sem entrada e com arcos condicionais. Devem começar a contagem logo que os arcos condicionais permitirem o disparo.
- Transições temporizadas com entrada e sem arcos condicionais. Devem começar a contagem assim que o evento de entrada for verdadeiro.
- Transições temporizadas com entrada e com arcos condicionais. Devem começar a contagem assim que o evento de entrada for verdadeiro, porém o disparo só acontece se ao final da contagem os arcos de condição forem satisfeitos.
- Transições temporizadas sem entrada e sem arcos condicionais. Nunca serão sensibilizadas e disparadas.

Tal definição evita que, por exemplo, uma transição sem evento de entrada, fique tentando contar o tempo de retardo, situação que pode evitar que a mesma dispare mesmo que os arcos condicionais permitam, pois a mesma estará contando repetidamente, e poderá somente ser disparada de forma períodica durante um pequeno espaço de tempo.

2.2 Estruturas de dados e execução de código

Quando se trata de programação de bibliotecas e artifícios[1] de código, ou seja, código que será usado por terceiros, é necessário sempre se ter em mente que a implementação seja além de funcional, bem documentada e que seja robusta a erros, conceito que envolve, mas não é limitado a, tratamento de erros e exceções, visibilidade parcial da implementação, que esconde parte da funcionalidade evitando uso indesejado por parte do usuário, ou ainda a utilização de testes unitários, que testam partes individuais da biblioteca de forma automatizada. Então é necessário entender

[1] Pequena parte, fragmento, de um todo

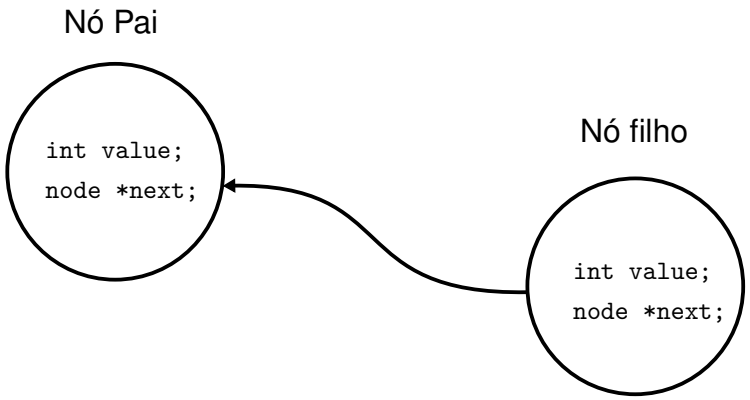
alguns conceitos de organização e execução de código por que garantirão nossas expectativas de funcionamento bem como irão garantir a robustez desejada a biblioteca.

2.2.1 Listas ligadas

Listas ligadas, ou listas encadeadas, são estruturas de dados parecidas com listas normais, que são conjuntos de dados de mesmo tipo dispostos de forma sequencial por um índice numérico, já as listas ligadas não são. Elas ganham a denominação de lista, pois cada elemento guarda dentro de si mesmo uma referência para o próximo elemento da lista, de forma encadeada, daí o nome desse tipo de lista.

Suas vantagens em relação às listas normais são que elas não tem espaço de memória predefinido e alinhado, elas podem crescer indefinidamente e de forma desorganizada com relação ao alinhamento de memória, são flexíveis, mas com o custo de terem de ser percorridas toda vez para se encontrar o elemento em dada posição, operação que tem um tempo de execução proporcional ao tamanho da lista, enquanto a lista normal faz isso de forma indexada, por isso é de tempo constante.

Figura 6 – Lista ligada com dois nós



Para implementações em C a lista é basicamente um conjunto de nós, que são estruturas que guardam dentro de si um valor de tipo desejado, e um ponteiro de memória para outro nó na lista.

2.2.2 Filas priorizadas

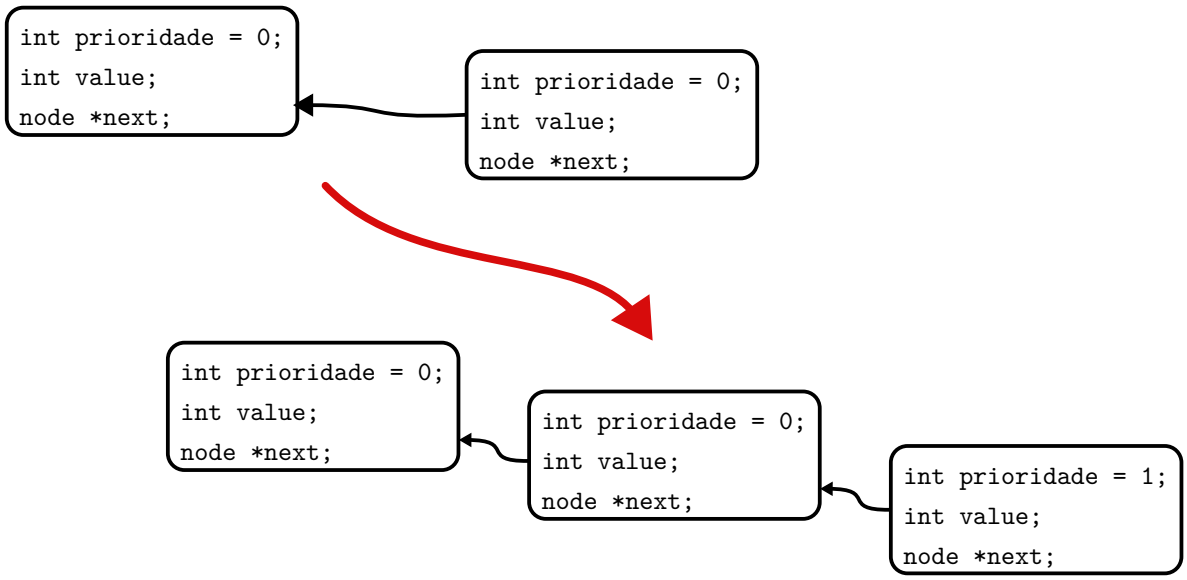
Filas priorizadas são como filas normais, listas onde o elemento a ser inserido é colocado a frente e retirado ao fim, ou seja, quem entra primeiro sai primeiro. A lista priorizada implementa a funcionalidade de inserir elementos baseados em uma

medida de privilégio que pode ser um número arbitrário ou uma medida como tempo de vida, uma escala de erro, etc. Da mesma forma que na fila normal, elementos são retirados ao fim da fila.

Podem ser implementados a partir de listas indexadas ou listas ligadas.

Um exemplo de fila priorizada seria uma onde processamos pacotes de entrega, sejam eles pacotes de protocolos de comunicação ou pacotes postais, onde temos uma lista ligada com dois pacotes de prioridade baixa, zero por exemplo, agora suponha que deseja-se processar um pacote com maior urgência, para tanto, se insere o pacote com seu valor de prioridade igual a um, assim a fila passa ter como último elemento o novo pacote, de prioridade maior.

Figura 7 – Dinâmica da lista priorizada para pacotes



2.2.3 *Threads* e *mutex locks*

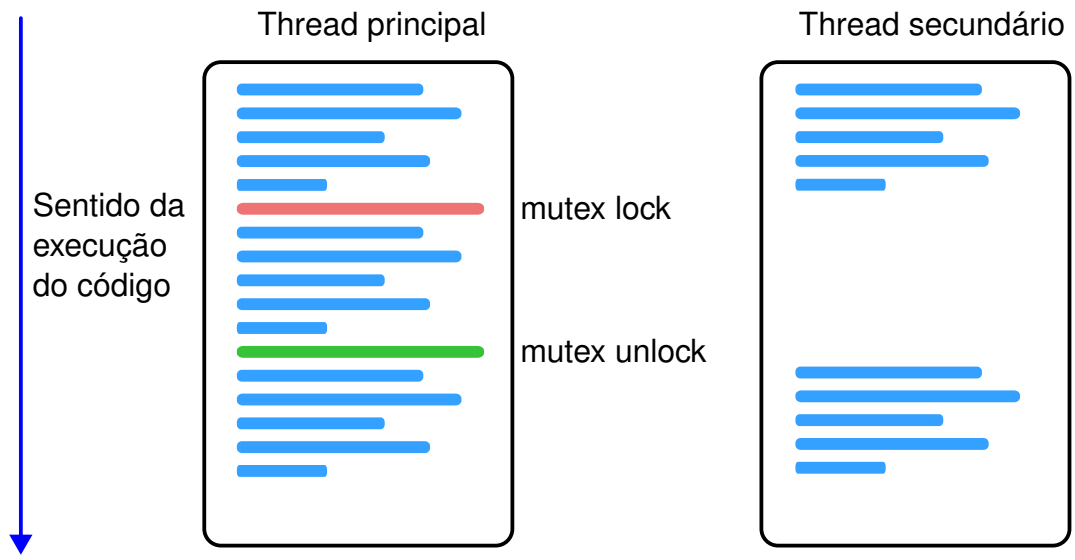
O processamento paralelo e concorrente são abordagens utilizadas para melhorar a eficiência e desempenho dos sistemas computacionais, permitindo a execução simultânea de múltiplas tarefas. No processamento paralelo, várias tarefas são executadas ao mesmo tempo, utilizando recursos computacionais simultaneamente. Isso é comumente aplicado em sistemas com múltiplos núcleos de processamento, onde diferentes tarefas podem ser executadas em paralelo, aumentando a capacidade de processamento total do sistema.

Uma das formas de alcançar o processamento paralelo é por meio do uso de *threads*. As *threads* são unidades básicas de execução em um programa, representando fluxos independentes de controle que podem executar tarefas em paralelo. Uma aplicação pode conter várias *threads*, permitindo a execução simultânea de diferentes

partes do código. As *threads* compartilham o mesmo espaço de memória, o que facilita a comunicação e a troca de dados entre elas.

A execução concorrente de *threads* também pode trazer desafios. Um problema comum é a ocorrência de *race conditions*, que acontecem quando duas ou mais *threads* acessam ou modificam uma região de memória, ou variável compartilhada simultaneamente, resultando em comportamento imprevisível do programa. Isso ocorre quando as operações das *threads* não são adequadamente sincronizadas, resultando em um conflito entre as operações concorrentes. Para evitar *race conditions*, são utilizados mecanismos de sincronização, como *mutex locks*, *mutex locks* são mecanismos de exclusão mútua, onde uma *thread* obtém um lock para bloquear o acesso de outras *threads* a um recurso compartilhado até que seja liberado. Isso garante que apenas uma *thread* possa acessar o recurso por vez, evitando *race conditions*. Os *mutex locks* são usados para proteger regiões críticas do código onde ocorrem operações compartilhadas, garantindo a consistência e prevenindo conflitos entre as *threads*.

Figura 8 – Processamento concorrente usando *mutex*



2.2.4 *Threads* e *callbacks*

Em utilização normal, quando um *thread* precisa acessar um recurso protegido, ela solicita o bloqueio do *mutex*, aguardando até que ele esteja disponível. Isso impede que outras *threads* acessem o recurso simultaneamente, evitando *race conditions*. Uma vez obtido o bloqueio do *mutex*, a *thread* pode acessar o recurso de forma segura, realizar as operações necessárias e, ao finalizar, deve desbloquear o *mutex* para permitir que outras *threads* possam acessar o recurso. Dessa forma, o uso adequado de *mutex*es permite controlar a execução e garantir a sincronização correta entre as *threads*, evitando conflitos e garantindo a integridade dos dados compartilhados.

A utilização de *callbacks* é uma forma comum de comunicação entre dois *threads* em programação concorrente. Um *callback* é uma função ou bloco de código que é passado como argumento para outra função. Ao utilizar *callbacks* para comunicação entre *threads*, um *thread* pode registrar um *callback* que será acionado quando uma determinada condição for atendida ou quando algum evento específico ocorrer. Essa abordagem é particularmente útil em situações em que um *thread* precisa notificar outro *thread* sobre algo que aconteceu. Por exemplo, considere um cenário em que um *thread* A está aguardando a conclusão de um processamento realizado pelo *thread* B. Em vez de ficar bloqueado a execução, esperando o *thread* B terminar, o *thread* A pode registrar um *callback* que será executado pelo *thread* B assim que o processamento for concluído. Dessa forma, o *thread* A pode continuar executando outras tarefas sem ficar preso esperando a finalização do *thread* B.

A utilização de *callbacks* nesse contexto permite uma abordagem assíncrona de comunicação, na qual os *threads* podem operar de forma independente e notificar uns aos outros quando necessário. Além disso, a flexibilidade dos *callbacks* permite que o código de um *thread* possa ser modificado ou estendido sem afetar diretamente o código do outro *thread*, tornando o sistema mais modular e adaptável. No entanto, é importante considerar questões de sincronização e concorrência ao utilizar *callbacks* entre *threads*. Mecanismos adequados, como *mutex locks* ou semáforos[2], devem ser empregados para garantir a correta sincronização e evitar problemas de *race conditions* ou acesso simultâneo a recursos compartilhados.

Tratando de implementações, mais específico em C, *callback* são funções e tem formatos distintos, e geralmente são passados como ponteiros de função. Normalmente *callbacks* são dados como funções genéricas que não retornam nada e recebem também informações genéricas, usado `void*`.

Código 2 – Implementação usual de callbaks em C

```
1  typedef struct{
2    int value;
3    char *name;
4  }custom_data_t;
5
6  void callback(void *args){
7    custom_data_t *data = (custom_data_t*)args;
8    printf("Name: %s, value: %d\n", data->name, data->value);
9  }
10
11 int main(){
12   custom_data_t data = {
13     .value = 42,
```

[2] Mecanismo de sincronização análogo a um semáforo de trânsito, onde só se executa código quando houver um “sinal verde”, que é dado pelo outro ator de execução

```
14      .name = "The answer"
15    };
16
17    register_callback(callback, &data);
18 }
```

Fonte: Do autor.

A implementação desses conceitos em C pode ser alcançada através da biblioteca `pthreads` (LINUX FOUNDATION, 2021), biblioteca de paralelismo/concorrência padrão em sistema que suportam POSIX e a mais difundida das implementações, geralmente presente em sistemas operacionais por padrão.

2.2.5 Temporização

Temporizadores são um tópico interessante em si, pois em sistemas embarcados geralmente têm-se acesso a temporizadores de *hardware*, que são precisos e fáceis de usar, porém, para sistemas operacionais que não são de tempo real e não possuem temporizadores de *hardware* disponíveis a nível de usuário, acabam se tornando um leve problema. A maneira como pode-se implementar um temporizador seria na criação de uma execução paralela/concorrente de código onde pode-se contar o tempo através de chamadas de função que retornam o tempo passado desde um ponto arbitrário, como a execução do sistema ou da própria aplicação, valor que pode ser impreciso em cima do fato de ainda estar sendo contado continuamente em uma rotina concorrente, que não é garantida de ser executada em tempo real. O resultado é que podemos alcançar timers com boa acurácia, na casa dos milissegundos, porém com baixa precisão, com bastante variação em torno do valor original de contagem.

A utilização de execução concorrente de código pelo uso de *threads* recaí sobre a capacidades da maioria dos sistemas, sejam computadores pessoais ou sistemas embarcados, portanto trata-se de uma boa escolha para implementação genérica. Há ainda a possibilidade de se utilizar temporizadores de *hardware* de forma condicional conforme a arquitetura envolvida, o que adiciona um grau de complexidade ao sistema de compilação e deve ser alterado pontualmente para cada sistema a ser suportado, portanto, uma boa escolha para projetos mais especializados e menos genéricos.

Para realizar a medição utiliza se a função `clock` da biblioteca padrão C. Essa função retorna o tempo de CPU decorrido desde o início do programa em ciclos de *clock*, que podem ser convertidos em unidades de tempo, como milissegundos. A função `clock` oferece uma forma padronizada de obter-se a temporização, independentemente do sistema operacional, garantindo maior portabilidade e precisão, no entanto, é importante ressaltar que a função `clock` mede o tempo de CPU decorrido e não segue de fato o tempo real.

A utilização normalmente recai sobre o paradigma de obter se o tempo inicial e calcular o tempo decorrido subtraindo chamadas seguintes a função até que a diferença entre elas seja maior ou igual ao tempo desejado.

Código 3 – Temporizador de $1\,\mathrm{s}$ em C

```c
#include <time.h>

#define CLOCK_TO_MS(x) ((int)((x) * 1000 / CLOCKS_PER_SEC))

int now = CLOCK_TO_MS(clock());

if((now - CLOCK_TO_MS(clock())) >= 1000){
  printf("1 second passed");
}
```

Fonte: Do autor.

2.3 Programação de PLC's

2.3.1 IEC 61131-3

A norma IEC 61131-3 (International Electrotechnical Commission, 2013) é um padrão internacional amplamente utilizado para a programação de controladores lógicos programáveis (PLCs) e sistemas de automação industrial. Ela foi desenvolvida pela Comissão Eletrotécnica Internacional (*International Electrotechnical Commission* - IEC) e define uma linguagem comum e um ambiente de programação para PLCs.

Uma das principais características da norma IEC 61131-3 é a definição de cinco linguagens de programação padronizadas para PLCs. Essas linguagens são: diagrama de contatos (*LD - Ladder Diagram*), lista de instruções (*IL - Instruction List*), texto estruturado (*ST - Structured Text*), diagrama de blocos de função (*FBD - Function Block Diagram*) e gráfico sequencial (*SFC - Sequential Function Chart*). Cada linguagem tem sua própria sintaxe e semântica, permitindo aos programadores escolher a linguagem mais adequada para a tarefa em questão.

Além das linguagens de programação, a norma IEC 61131-3 também estabelece diretrizes para a organização e estruturação de programas PLC, o uso de variáveis, a manipulação de tempo, a comunicação com dispositivos externos e a interface com o *hardware* do PLC. Essas diretrizes garantem a interoperabilidade entre diferentes PLCs e facilitam a portabilidade de programas entre diferentes sistemas.

Outro aspecto importante da norma IEC 61131-3 é a definição de blocos de função reutilizáveis. Esses blocos são unidades de *software* que encapsulam uma funcionalidade específica e podem ser usados em diferentes programas e projetos. Eles promo-

vem a modularidade, a reutilização de código e a manutenção simplificada, permitindo um desenvolvimento mais eficiente e flexível de sistemas de automação.

A norma também aborda aspectos relacionados à segurança funcional, especificando requisitos para garantir a integridade e confiabilidade dos sistemas de automação. Ela define categorias de segurança e diretrizes para a programação de funções de segurança, como monitoramento de entradas, lógica de falha segura e supervisão de sistemas.

2.3.2 Lista de instrução

Uma das linguagens padronizadas para programação de PLC's é a lista de instrução (MASLAR, 1996), que consiste em uma sequência de instruções que especificam as operações a serem executadas pelo PLC afim de controlar o comportamento do sistema desejado. Dentre os comandos mais comuns presentes destacam-se o `LD` (Load), `ST` (Store), `S`, `AND` dentre outros.

O comando `LD` (Load) é utilizado para carregar um valor ou estado lógico em uma variável interna ou memória do PLC. Essa instrução é fundamental para a atribuição de valores a variáveis e para a leitura de entradas do sistema.

O comando `ST` (Store) é responsável por escrever um valor lógico em uma memória, geralmente uma memória interna ou saída do PLC. Caso referencie uma saída, irá se controlar a ativação ou desativação de dispositivos externos.

O comando `S` é utilizado para definir um estado lógico específico em variáveis internas do PLC. Esse comando é útil para inicializar variáveis ou para acionar estados específicos em situações de controle. Diferente do comando `ST`, esta instrução apenas grava o valor lógico "verdadeiro" em uma variável, permanecendo assim até seja alterado por outra instrução.

O comando `AND` realiza a operação lógica "e" entre dois ou mais valores. Ele permite realizar condições de controle mais complexas ao combinar múltiplas entradas.

Estes comandos basicamente dão ordem de execução a um programa. Um programa é formado por seções, cada seção de execução individualmente e em ordem sequencial. Para cada seção, é primeiramente carregado um valor, usando por exemplo `LD`, seguido então por alguma lógica, e por fim direcionando o resultado para uma memória interna ou saída física, usando por exemplo `ST` ou `S`.

Um exemplo de seção seria abrir uma variável e escrever seu valor em uma saída.

Código 4 – Seção de entrada e saída em lista de instrução

```
1 LD var
2 ST outvar
```

Fonte: Do autor.

2.3.3 Referência WEG TPW04

A definição de lista de instrução é padronizada, porém, fabricantes optam por realizar implementações que abrangem a norma em grande parte, mas que por vezes apresentam diferenças e particularidades. Da empresa WEG, o modelo de PLC TPW04 (WEG, 2015), pode apresentar algumas diferenças em relação à definição da norma IEC 61131-3. Essas diferenças podem ocorrer devido a recursos específicos do PLC como otimizações de desempenho ou necessidades específicas do ambiente de automação em que o TPW04 é utilizado. É importante ressaltar que as diferenças podem variar dependendo da versão do firmware do TPW04. Aqui estão algumas possíveis diferenças.

- Conjunto de Instruções: O TPW04 pode ter um conjunto de instruções específico que vai além das instruções definidas na norma IEC 61131-3. A WEG pode adicionar instruções adicionais para atender a requisitos específicos do PLC ou para fornecer recursos extras aos programadores.
- Sintaxe e Semântica: Embora o TPW04 siga os conceitos e princípios gerais da norma IEC 61131-3, pode haver diferenças sutis na sintaxe e nomenclatura das instruções. Essas diferenças podem ser introduzidas para simplificar a programação ou otimizar o desempenho do PLC.
- Recursos Específicos: O TPW04 pode oferecer recursos específicos que não estão incluídos na norma IEC 61131-3. Isso pode incluir recursos avançados de temporização, contadores específicos, funções de comunicação específicas ou suporte para dispositivos periféricos exclusivos da WEG.

Para a referência WEG, existem algumas extensões e renomeações dos comandos definidos pela norma. Um exemplo de instrução específica WEG para o TPW04 seriam as bordas de subida, `LDP`, e descida `LDF`, que são utilizadas para controlar a ocorrência de eventos específicos no sistema. As bordas de subida são acionadas quando um sinal lógico muda de zero para um, enquanto as bordas de descida são acionadas quando o sinal lógico muda de um para zero. Essas bordas são fundamentais para a detecção de transições e acionamento de ações em tempo real.

Também há temporizadores, que são componentes essenciais na programação de controladores lógicos programáveis. Eles permitem a criação de atrasos programados e temporizações precisas, sendo utilizados para controlar o tempo de espera entre eventos e para executar ações em momentos específicos. Os temporizadores podem ser configurados para atrasos fixos ou variáveis, dependendo das necessidades do sistema.

Por fim temos alguns comandos que são de extrema importância, pois possibilitam a reutilização de uma mesma computação várias vezes, os comandos `MPS` (Memory

push), inserir em memória, MRD (Memory read), ler da memória, e MPP (Memory pop), retirar da memória. Os comandos funcionam basicamente como uma pilha, onde o resultado de uma operação pode ser inserido com (MPS), re-lido quantas vezes forem necessárias com (MRD), e por fim lido a última vez, ao mesmo tempo que é retirado da pilha, (MPP). Esses comandos são importantes para a coordenação e controle de programas em sistemas complexos, pois possibilitam atrelar várias saídas a um mesmo conjunto de condições.

Como este projeto visa a implementação prática para com o modelo TPW04, está será a referência a ser seguida para o desenvolvimento do compilador, respeitando as instruções específicas WEG, porém tentando utilizar as instruções mais triviais e gerais de forma a facilitar futuras adaptações para outras referências de outros modelos e fabricantes.

2.3.4 Lista de instrução e *Ladder*

Uma das características da programação de PLC's é que pelo padrão IEC 61161-3, plataforma de programação de desses PLC's implementam um conjunto dessas linguagens, sendo mais usuais a linguagem *Ladder* e de lista de instrução, e normalmente também fabricantes dão a habilidade aos programadores de alterar em tempo real entre as linguagens, isso é possível, pois todas são relativamente simples e normalmente podem ser traduzidas a partir da lista de instrução.

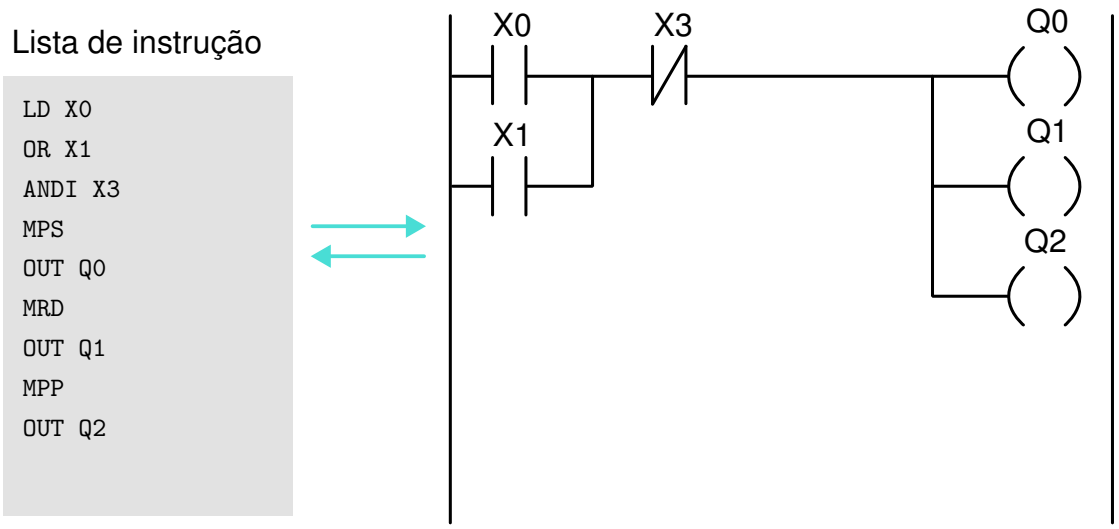
Assim, contanto que se tenha um método de tradução de uma dada linguagem para lista de instrução e vice-versa, há possibilidade do fabricante implementar essa tradução em tempo real.

A linguagem *Ladder* em especial é um ótimo candidato, pois as operações mais básicas são facilmente mapeadas. No exemplo da figura 9, temos um simples programa de PLC, onde pode-se observar um exemplo de implementação da relação entre as linguagem *Ladder* e lista de instrução. Observa-se que para as ramificações *Ladder* de entrada, utilizamos somente operações lógicas, uma operação lógica "ou" e outra "e, negada", assim, para entradas, a maior parte dos casos será uma expressão lógica em termos das entradas.

Para saída iremos utilizar as instruções de memória, inserindo o valor após a última operação lógica e depois lendo até a última entrada, onde o valor é retirado e a declaração do programa termina.

Podem haver outras implementações e variações desta apresentada, porém, serve de ilustração para a capacidade de se converter entre as linguagens, provando em torno também que quando se tem um programa em lista de instrução, podemos facilmente alcançar outros paradigmas e linguagens, se tornando um ótimo alvo base para compilação de programas.

Figura 9 – Exemplo de tradução entre *Ladder* e lista de instrução referência WEG TPW04



3 Desenvolvimento

3.1 Metodologia

Este trabalho se trata de um processo de desenvolvimento pelo do autor de maneira autônoma, dado que os tópicos de programação e alguns detalhes de implementação são por natureza de livre implementação, tornando este trabalho por grande parte um trabalho exploratório.

3.1.1 A base do projeto

Para implementação do sistema proposto, é necessário entender que deseja-se que os resultados do sistema, as redes de petri, sejam acessíveis em várias plataformas, visando-se generalização. Entende-se assim que deve-se haver um motor capaz de executar estas redes e seu funcionamento em suas respectivas plataformas. Em computadores de uso geral, e em sistemas embarcados torna-se atrativo uma implementação na forma de uma biblioteca em linguagem C, pois a maioria dos sistemas embarcados funcionam com *toolchains*[1] em C, e também pois abre-se as portas para que outros ambientes e linguagens de programação utilizem esta biblioteca. Isto é possível pelo fato de C ser de baixo nível e seguir padrões estabelecidos de arquivos e execução de código no geral, tornando possível que linguagens de mais alto nível possam fazer *bindings*[2] para com a biblioteca.

Mais ainda, outro motor importante é a capacidade de executar tais redes em plataformas distintas, onde a portabilidade da biblioteca em C torna-se difícil, como os PLC's comentados anteriormente, onde a lista de instrução é mais comum do que a linguagem C. A lista de instrução, tanto pela difusão quanto pelo baixo nível de abstração, um candidato preferido para alvo de compilação, assim a rede pode ser editada, simulada e testada em um computador de uso geral, por exemplo, e então compilada para lista de instrução, que deve ser gerada de forma a garantir funcionalidade igual a da biblioteca em C.

O baixo nível de abstração da lista de instrução ainda possibilita que esta seja usada como fonte para compilação posterior por ferramentas de terceiros para suas plataformas alvo, abrindo mais funcionalidade para esse tipo de sistema de trabalho. Por exemplo, lista de instrução é comumente compilada para *Ladder* de forma intercambiável, ou seja, é um processo bidirecional, comportamento desejado por

[1] Conjunto de ferramentas que possibilitam a programação e manuseio de programas para uma plataforma e/ou arquitetura de computador específica.

[2] Sistema onde se mapeia de forma diretas funções, variáveis e definições de código entre sistemas/ambientes de programação diferentes, possibilitando interoperabilidade entre elas

desenvolvedores, dado que *Ladder* é uma linguagem visual simples e de mais fácil desenvolvimento do que a lista de instrução pura.

Para implementação da representação das redes, da dinâmica e da compilação será utilizada a linguagem C bem como um sistema de compilação para a biblioteca usando ferramentas básicas no padrão POSIX, em especial do projeto GNU (GNU, 2023) sendo elas o compilador *gcc*, e para *build*[3] o *Make*.

3.1.2 Rede de petri

Quanto a definição do tipo de rede de petri adotada neste trabalho, serão adotadas redes de petri com extensões específicas e utilidade geral de e industrial, garantindo maior flexibilidade no design. A definição destas extensões e os detalhes de implementação serão discutidos e embasados conforme trabalhos anteriores, bem como a experiência prática do autor.

3.1.3 Compilador de lista de instrução

Lista de instrução é um tipo de programação relativamente difundida e portanto bem generalizada, mas ainda assim há diferenças entre fabricantes. Em vista disto, a implementação do compilador de rede de petri para lista de instrução proposta neste trabalho irá utilizar uma implementação específica, sendo esta a referência da fabricante WEG para o PLC TPW04 (WEG, 2015), sendo este um modelo amplamente utilizado em meio industrial e também de fácil acesso em educacional e acadêmico. Futuros trabalhos podem partir da mesma referência para implementação de compiladores para mais arquiteturas de PLC's, dada que as diferenças de implementação são pequenas entre tipos de plataformas e fabricantes diferentes devido a norma IEC 61161-3 (International Electrotechnical Commission, 2013).

3.1.4 Publicação

Todo o trabalho desenvolvido nesta obra será versionado e disponibilizado no repositório "pnet" (PETERSON, 2023) via Github (GITHUB, 2023), sob a licença domínio público MIT (MIT, 2023).

3.2 Biblioteca C

No desenvolvimento da biblioteca em C, como qualquer outro tipo de programa, uma das primeiras coisa a serem feitas é a definição de uma estrutura de dados, como discutido previamente, iremos implementar a rede de petri em código utilizando

[3] Processo organizado de construção de um programa ou biblioteca a partir de diferentes arquivos fonte, que são compilados e ligados conforme a necessidade do projeto.

a representação matricial. Implementada a funcionalidade das matrizes podemos definir a estrutura da nossa rede de petri e por fim, definir os algoritmos que irão realizar a funcionalidade desejada da biblioteca.

Como padrão serão definidas para cada estruturas de dados um `struct` que será inicializado a partir de uma função de criação de forma dinâmica, retornando um ponteiro para a estrutura.

3.2.1 Representação matricial

Primeiramente define-se a estrutura primitiva da matriz, a qual irá comportar o tamanho e *array* dinâmico de memória de duas dimensões. Escolhe se `size_t` pois este comporta o maior valor de tamanho disponível na arquitetura a ser compilada para e `int` para os valores da matriz, visto que todas as matrizes que irã comportar a representação matricial podem ter valores positivos e negativos.

Código 5 – Estrutura C da matriz

```
1 typedef struct{
2     size_t x;
3     size_t y;
4     int **m;
5 }pnet_matrix_t;
```

Fonte: Do autor.

Com a definição da matriz, pode-se pensar nas definições dos arcos, para isso em uma matriz A, n por m, com n sendo a quantidade de linhas e lugares, e m sendo a quantidade de colunas e transições, arcos de tipo peso podem ser dados como no seguinte exemplo.

$$A = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 2 \end{bmatrix}$$

Onde pode-se ver três lugares e duas transições, onde a primeira transição retira uma ficha do primeiro lugar e a coloca no segundo lugar, e a segunda transição retira do segundo lugar e coloca duas fichas no terceiro lugar.

Uma das problemáticas que irá surgir é a que a rede de petri permite dois arcos de peso de um mesmo lugar para uma mesma transição, e nossa representação não a comporta, pois para retirar e colocar uma ficha, devemos marcar como zero, mas zero significa nenhum arco, para tanto, divide-se a matriz em duas, uma matriz A_p para arcos de peso positivo, que colocam fichas em lugares e outra matriz A_n para arcos negativos, que retiram fichas de lugares. Assim teremos para o mesmo exemplo:

$$A_p = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 0 & 0 \end{bmatrix}$$

$$A_n = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 2 \end{bmatrix}$$

Assim temos a distinção entre arcos não existentes e dois arcos distintos entre um lugar e uma transição.

Para os arcos negados e de *reset* a representação é mais simples, pois devemos apenas marcar com um, para um arco, ou zero, para sem arco. Exemplo de um arco negado da primeira transição para com o terceiro lugar.

$$A_i = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}$$

E um exemplo de um arco *reset* que após o disparo da primeira transição zera todos os lugares.

$$A_r = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

Tendo já os arcos, podemos agora definir o estado inicial da rede de petri, que são as fichas iniciais que alguns lugares possuem conforme o design do autor da rede. É relativamente simples, apenas uma matriz unidimensional com os valores para cada lugar. Exemplo de lugar inicial de um de três lugares.

$$P_i = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

Também para as transições é necessário definir o valor da temporização dos arcos, que também é uma matriz unidimensional. Exemplo de um tempo de $5\,\mathrm{s}$ para a primeira de duas transições.

$$T = \begin{bmatrix} 5000 & 0 \end{bmatrix}$$

Para as entradas da rede define-se, como nos arcos, uma matriz de relação entre entradas e transições. Assim define-se uma matriz de n linhas, as entradas, por m colunas, as transições, e também deve se definir uma forma de codificar em um número inteiro, devido à implementação da matriz, o tipo de evento a ser usado, para isso define-se um enumerador C com as três possíveis bordas propostas, de subida, descida e ambas.

Código 6 – Codificação dos eventos de entrada

```
typedef enum{
    pnet_event_none          = 0x00,
    pnet_event_pos_edge      = 0x01,
    pnet_event_neg_edge      = 0x02,
    pnet_event_any_edge      = 0x03,
    pnet_event_t_max
}pnet_event_t;
```

Fonte: Do autor.

Observa-se que definimos também um evento nulo, zero, que simboliza nenhuma relação, e um evento `max`, que tem como função apenas a checagem, que será vista mais a frente. Note também como a borda de ambos, `any_edge`, é a soma do valor das outras duas bordas, assim que utilizar a biblioteca para criação de redes de petri pode utilizar do operador lógico "ou" para indicar verbosamente ambas as bordas, `pnet_event_any_edge = pnet_event_pos_edge | pnet_event_neg_edge`.

Um exemplo de matriz onde deseja-se que a transição um seja acionada somente quando haja uma borda de subida da entrada dois.

$$I_{in} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

As saídas são definidas como a quantidade de fichas maior ou igual necessárias a ativação, portanto nenhuma codificação é necessária, apenas o valor numérico, assim tempos uma matriz de n linhas, lugares, or m colunas, saídas. Um exemplo caso desejarmos que a saída um seja acionada quando o número de fichas no lugar três seja maior ou igual a quatro.

$$O_{out} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 4 & 0 \end{bmatrix}$$

Com estas definições podemos então definir a nossa estrutura em C da rede de petri, listagem de código 7.

Código 7 – Estrutura C da rede de petri

```
struct pnet_t{
    // size
    size_t num_places;
    size_t num_transitions;
    size_t num_inputs;
    size_t num_outputs;

    // maps
```

```
9     pnet_matrix_t *neg_arcs_map;
10    pnet_matrix_t *pos_arcs_map;
11    pnet_matrix_t *inhibit_arcs_map;
12    pnet_matrix_t *reset_arcs_map;
13    pnet_matrix_t *places_init;
14    pnet_matrix_t *transitions_delay;
15    pnet_matrix_t *inputs_map;
16    pnet_matrix_t *outputs_map;
17
18    // validation
19    bool valid;
20
21    // net state
22    pnet_matrix_t *places;
23    pnet_matrix_t *sensitive_transitions;
24
25    // input edges state
26    pnet_matrix_t *inputs_last;
27
28    // output values
29    pnet_matrix_t *outputs;
30
31    // async
32    pnet_callback_t function;
33    void *user_data;
34    pthread_t thread;
35    pthread_mutex_t lock;
36    transition_queue_t *transition_to_fire;
37 };
```

Fonte: Do autor.

Nota-se a presença de mapas, estes são as matrizes até então mencionadas, mesmo as fichas inicias e temporização, o restante dos valores são pertinentes ao processamento e execução de rede e serão explorados posteriormente.

3.2.2 Checagem

Dada a estrutura da rede de petri a primeira coisa a ser pensada é que as matrizes são de livre preenchimento e portanto podem haver definições de redes de petri inválidas, comportamento que deve ser devidamente checado para que a rede de petri possa ser minimamente executável, para tanto implementa-se um algoritmo de checagem baseado nas definições propostas, na capacidade de execução e viabilidade do ponto de vista de *software*.

Destaca-se um problema em particular, que seria a mínima rede válida permitida, pois diz respeito as capacidades permitidas para execução da rede sem erros e garan-

tindo a comportamento desejado dadas as especificações. Para isto vamos considerar quais seriam as matrizes opcionais da rede de petri. Entradas e saídas são opcionais por não interferem no funcionamento da rede. Temporização também não é requerida para todas as transições.

A quantidade de lugares e transições é dada pelos tamanhos das matrizes de arcos e da marcação inicial das fichas, assim ao menos uma dessas deve ser não nula, e mesmo que não hajam arcos, uma rede pode ainda ser válida, com somente lugares e transições, somente um lugar e uma transição ou ainda um único lugar ou única transição. Para uma rede de petri com uma única transição devemos ter ao menos uma das quatro matrizes de arcos, e dado que estas também definem a existência de lugares, pois não podem haver zero linhas em uma matriz, então uma rede de apenas uma transição não é possível neste tipo de implementação, sendo assim a menor rede viável é a de apenas um lugar, a qual pode ser definida apenas pela matriz de marcação inicial.

É interessante ainda expandir essa definição se escolhermos que nossa rede de petri seja passível de execução, sendo necessário assim uma transição e um tipo de arco capaz de alterar a quantidade de fichas, arcos de peso ou arcos de *reset*. Assim então definimos a rede mínima para esta implementação uma rede com no mínimo a matriz de marcação inicial e ao menos umas das duas matrizes de arco, de *reset* ou de peso, as quais podem ser matrizes de minimamente de um por um, sendo assim as redes mínimas as redes de um lugar e uma transição com um arco de peso ou arco de *reset*.

Dadas as especificações e a análise da rede mínima viável, os seguintes pontos devem ser reforçados para criação das redes:

- Consistência na quantidade de lugares e transições nas matrizes. Erro se qualquer matriz não for adequada.
- Existência de somente números positivos na matriz de arcos positivos. Outros valores serão zerados.
- Existência de somente números negativos na matriz de arcos negativos. Outros valores serão zerados.
- Existência de somente um ou zero na matriz de arcos negados e arcos de *reset*. Outros valores serão considerados como um.
- Existência da matriz de marcações iniciais. Erro se não detectado.
- Números negativos de fichas na inicialização dos lugares, interpretado como erro.

- Existência de pelo menos um arco de pelo menos um tipo um de arco, garantindo dinâmica na rede. Erro se não detectado.
- Números negativos na temporizações das transições, interpretado como erro.
- Existência de apenas eventos válidos de entrada. Corrigido para evento nulo, zero, caso contrário.
- Existência de apenas um evento de entrada por uma transição. Erro caso contrário.

Ao final da checagem, uma variável binária é marcada como um dentro da estrutura da rede de petri, indicando que esta pode-se executada normalmente.

É importante ressaltar que a criação da rede e sua checagem é referente a esta implementação em específico dada as necessidades de *software*, tipo de rede proposto e limitações, não necessariamente refletindo os conceitos formais de rede de petri mínima e implicações relacionadas.

Também referente a implementação são realizados os seguintes testes práticos na implementação da rede de petri para garantir funcionalidade não só de execução mínima mas também das outras capacidades que irão serão posteriormente desenvolvidas nas seções seguintes.

- Teste para retorno não nulo e código de erro ok
- Teste para criação com todos os argumentos
- Teste para todos os argumentos nulos
- Teste para apenas lugares, sem arcos
- Teste para autocorreção de valores
- Teste para arcos sem peso ou negados
- Teste para múltiplas entradas em uma única transição
- Teste de sensibilização
- Teste de disparo sem arcos de peso ou *reset*
- Teste de disparo com entradas para matriz de entrada nulo
- Teste de disparo com entradas nulas para matriz de entrada não nula
- Teste de disparo de uma única transição com evento de entrada do tipo `none`
- Teste de falha no disparo

- Teste de transição com arco de peso, negados e *reset* e eventos de entrada
- Teste de arco de peso e *reset*
- Teste para sensibilização de transições sem matriz de entradas e entrada nula para `pnet_fire`
- Teste de disparo manual
- Teste para estado de saída no início
- Teste para múltiplas transições mútuas e um arco bidirecional
- Teste para estado de saída no final
- Teste de rede temporizada sem passar função de *callback*
- Teste de acurácia de transições temporizada
- Teste de petri multi temporizadas
- Testes com redes de petri de demonstração
- Testes de serialização de matriz
- Testes de desserialização da matriz
- Comparação se matriz é igual a desserialização da serialização da matriz
- Teste para criação de matrizes grandes
- Teste matriz de um por um é igual a desserialização da serialização
- Teste de tamanho máximo factível da desserialização da serialização
- Teste de desserialização da serialização da rede de petri
- Teste de compilação de rede de petri para lista de instrução

3.2.3 Sensibilização

Dado uma rede petri válida, para que antes de a executarmos em tempo real, precisamos executá-la uma única vez, um disparo, e para isso necessitamos antes verificar quais transições estão sensibilizadas dado o estado atual, dado pela matriz `places`, visto no código 7. Para isso utilizaremos um simples algoritmo que irá varrer e comparar os mapas dos arcos de condição, que são os arcos de peso negativos e arcos negados, com o estado atual da rede.

Código 8 – Sensibilização das transições

```
for(size_t transition = 0; transition < pnet->num_transitions;
    transition++){
    // set transition to sensibilized
    pnet->sensitive_transitions->m[0][transition] = 1;
    for(size_t place = 0; place < pnet->num_places; place++){

        // check desensibilization
        if(
            // negative arcs
            (
                // when there are negative arcs
                (pnet->neg_arcs_map != NULL) &&
                // when there is a neg arc for this transition/place
                (pnet->neg_arcs_map->m[place][transition] != 0) &&
                // and there is not enough tokens
                ((pnet->neg_arcs_map->m[place][transition] + pnet->
    places->m[0][place]) < 0)
            ) ||
            // inhibit arcs
            (
                // when there are inhibit arcs
                (pnet->inhibit_arcs_map != NULL) &&
                // if there is a inhibit arc
                (pnet->inhibit_arcs_map->m[place][transition] == 1) &&
                // and place has token, then do not trigger, otherwise
    trigger
                (pnet->places->m[0][place] != 0)
            )
        ){
            // desensibilize
            pnet->sensitive_transitions->m[0][transition] = 0;
            break;
        }
    }
}
```

Fonte: Do autor.

Na implementação primeiramente dizemos que, para determinada transição, ela se encontra sensibilizada por padrão, então verificamos se existem as matrizes dos arcos de condição e para cada tipo de arco, se existe relação entre o lugar e transição atual, e ainda, se o lugar possui a condição necessária, seja de fichas necessárias para os arcos de peso negativo ou zero fichas para o arco negado. Chegando ao final de todas as das condições, se algo não for verdadeiro, a transição será marcada como não sensibilizada.

Note que a matriz das transições `sensitive_trasitions` é interna a rede de petri e que a cada verificação de sensibilidade retorna várias transições sensibilizadas, que está em contradição com nossa especificação da seção 2.1.4, porém será tratada posteriormente na realização de disparos.

O ponto de entrada para realização análise de sensibilização será uma função distinta denominada `pnet_sense`.

3.2.4 Disparo

Dada uma matriz com as transições a serem sensibilizadas é necessário ainda verificar se as transições possuem relação de entrada, caso no qual deverá ser feita uma verificação das bordas de entrada. A verificação das bordas de entrada é feita comparando se o valor atual é diferente do anterior, para isso é necessário guardar o estado anterior das entradas, seria a matriz `inputs_last` dada no código 7.

O ponto de entrada para realização dos disparos será uma função distinta denominada `pnet_fire`, a qual dentro si realiza uma chamada a função `pnet_sense`, incorporando assim a lógica completa de disparos, exceto pelos disparos temporizados, que irão ocorrer de forma assíncrona, discutidos na próxima seção. Na função `pnet_fire`, também é repassado o valor atual das entradas externas, assim pode-se verificar se houve mudança e portanto, se houver um evento ou uma borda, e se o evento irá contribuir para o disparo de alguma transição. Para isso verifica-se a entrada com o estado anterior e armazena se o resultado na matriz temporária `edges`.

Código 9 – Processamento dos eventos de borda de entrada

```
1 // run for every input given
2 for(size_t input = 0; input < pnet->num_inputs; input++){
3     // check for pos edges
4     if(pnet->inputs_last->m[0][input] == 0 && inputs->m[0][input] ==
      1){
5         edges->m[0][input] = pnet_event_pos_edge;
6     }
7     // check for neg edges
8     else if(pnet->inputs_last->m[0][input] == 1 && inputs->m[0][input]
      == 0){
9         edges->m[0][input] = pnet_event_neg_edge;
10    }
11 }
12
13 // store last inputs
14 pnet_matrix_copy(pnet->inputs_last, inputs);
```

Fonte: Do autor.

Primeiramente para cada transição, se assume que ela está sensibilizada, então se verifica para cada uma se existe um mapa de entrada, caso não ela permanece sensibilizada, caso não, verifica-se se a mesma possui um evento nulo atrelado, caso qual pulamos e a transição continua sensibilizada, caso contrário, se a borda for a mesma especificada pela matriz de entrada então a mesma continua sensibilizada. Caso nenhuma dessas condições não for verdadeira, a transição é marcada como dessensibilizada.

Código 10 – Sensibilização de transições pelos evetnos de entrada

```
for(size_t transition = 0; transition < pnet->num_transitions; transition++){
    transitions->m[0][transition] = 0;

    // if input map is null all transitions can occurr
    if(pnet->inputs_map == NULL){
        transitions->m[0][transition] = 1;
        continue;
    }

    for(size_t input = 0; input < pnet->num_inputs; input++){


        // if event type is none mark as firable, run until the end of inputs
        if(pnet->inputs_map->m[input][transition] == pnet_event_none){
            transitions->m[0][transition] = 1;
        }
        // if the transitions has an event. When a single event is found then this event must be satisfied,
        // otherwise the transition stay desensibilized, so we exit the loop when we reach it
        else{
            // using the & operator to check edge type, see pnet_event_t for why
            if(pnet->inputs_map->m[input][transition] & edges->m[0][input]){
                transitions->m[0][transition] = 1;
            }
            else{
                transitions->m[0][transition] = 0;
            }

            break;
        }
    }
```

```
31 }
```

Fonte: Do autor.

É interessante notar como no código 6, as bordas de subida e descida são distintas, mas a borda ambos é a soma do valor delas, basta realizar a operação lógica “e” para poder determinar o resultado, se o valor da matriz de entrada for igual ao detectado, então “e” será verdadeiro, e se o valor da matriz for para ambos, se a detectada for, por exemplo, de subida, então teremos `0x03 & 0x01`, que será também verdadeiro, como pode ser visto na linha 21 do código 10.

Tendo agora duas matrizes, a matriz de transições sensibilizadas pelos arcos condicionais e a matriz de transições sensibilizadas pelas entradas, portanto as transições que devem ser disparadas serão o resultado lógico “e” de ambas as matrizes.

Código 11 – Disparo das transições

```
1  // transitions that are sensibilized and got the event
2  pnet_matrix_t *transitions_able_to_fire = pnet_matrix_and(
       input_event_transitions, pnet->sensitive_transitions);
3
4  // fire transitions
5  for(size_t transition = 0; transition < pnet->num_transitions;
       transition++){
6      if(transitions_able_to_fire->m[0][transition] == 1){
                 // firable transition
7          if(
8              (pnet->transitions_delay == NULL) ||
                 // not timed
9              (
10                 (pnet->transitions_delay != NULL) &&
                 // timed
11                 (pnet->transitions_delay->m[0][transition] == 0)
                 // but instant
12             )
13         ){
14             // move and callback
15             pnet_move(pnet, transition);
16             if(pnet->function != NULL) pnet->function(pnet, transition
       , pnet->user_data);
17             break;
                 // only one instant transitions
18         }
19         else{
20             // add to queue
21             transition_queue_push(pnet->transition_to_fire, transition
       , pnet->transitions_delay->m[0][transition]);
```

```
22            }
23
24        }
25 }
```

Fonte: Do autor.

No código 11 linha 2 vemos a operação lógica “e”, e temos então a matriz de transições para executar, porém, conforme mencionado na seção 2.1.4, destas, podemos executar ao máximo uma transição, então a partir do momento que temos uma transição não temporizada que está apta, esta será executada e o irá quebrar o laço de repetição, assim outras transições não serão executadas. Aqui vê se a ambiguidade de que a primeira transição que for declara tem prioridade, devido ao laço de repetição, essa é a justificativa para implementação de prioridade de transições, que vem do tipo de rede de petri priorizada, porém, novamente, a escolha deste tipo de implementação é justificada pela finalidade de aplicação, assumindo que esse tipo de ambiguidade não será de grande problema.

A função `pnet_move` realiza a movimentação das fichas conforme os arcos atuadores, os arcos de peso negativo e positivo e o arco *reset*, a forma como a movimentação é alcançada pode ser dada em termos matemáticos, para os arcos de peso, de uma única matriz A com arcos positivos e negativos, e uma matriz T que representa a transição a ser disparada, o resultado da movimentação das fichas pode ser dado como:

$$T_{t=0} = \begin{bmatrix} 1 & 0 & \dots \end{bmatrix}$$

$$P_n = P + T \cdot A^T$$

Onde a é uma célula, m o índice da coluna, A^T a transposta de, P a matriz dos lugares atuais e P_n os novos lugares após o disparo.

Esse algoritmo funciona de forma adequada, exceto pelo fato da multiplicação matricial ser extremamente custosa para grandes matrizes. Como se sabe o tipo de matriz que temos, podemos começar extraindo a coluna da matriz de arcos que representa a única transição a ser disparada, diminuindo drasticamente o cálculo envolvido. Com essa nova matriz unidimensional, pode-se simplesmente realizar a soma diretamente aos lugares da rede de petri.

Para a implementação desse trabalho necessitamos primeiramente extrair as colunas da matriz de arcos positiva e negativa e então realizar a soma.

$$a = A(m = t)$$

Onde A_n são os arcos negativos e $(m = t)$ representa a coluna da transição t, sendo a_n um vetor de m elementos e uma única linha.

$$P_n = \left(P + a_n^T + a_p^T\right) \odot \neg a_r$$

Onde a_n são a coluna transição dos arcos negativos, p positivos e r de *reset*, $\odot$ é a multiplicação de elemento a elemento, diferente da multiplicação matricial normal, e $\neg$ a negação lógica dos elementos matriciais. Em código temos:

Código 12 – Movimentação das fichas

```
1  // for weighted arcs, only if at least one of them is not null
2  if(pnet->pos_arcs_map != NULL || pnet->neg_arcs_map != NULL){
3
4      pnet_matrix_t *weighted_matrix = NULL;
5      pnet_matrix_t *pos_weighted_matrix = NULL;
6      pnet_matrix_t *neg_weighted_matrix = NULL;
7
8      // extract the values from the arcs for the transition
9      if(pnet->pos_arcs_map != NULL)
10         pos_weighted_matrix = pnet_matrix_extract_col(pnet->
       pos_arcs_map, transition);
11     if(pnet->neg_arcs_map != NULL)
12         neg_weighted_matrix = pnet_matrix_extract_col(pnet->
       neg_arcs_map, transition);
13
14     // sum the pos and neg values or returning only the non null maps
15     if(pos_weighted_matrix != NULL && neg_weighted_matrix != NULL){
16         weighted_matrix = pnet_matrix_add(pos_weighted_matrix,
       neg_weighted_matrix);
17         pnet_matrix_delete(pos_weighted_matrix);
18         pnet_matrix_delete(neg_weighted_matrix);
19     }
20     else if(neg_weighted_matrix != NULL)
21         weighted_matrix = neg_weighted_matrix;
22     else
23         weighted_matrix = pos_weighted_matrix;
24
25     // finnaly adding the difference in tokens for the places,
       effectly moving the tokens
26     pnet_matrix_t *buffer = pnet_matrix_add(places_res,
       weighted_matrix);
27     pnet_matrix_copy(places_res, buffer);
28     pnet_matrix_delete(weighted_matrix);
29     pnet_matrix_delete(buffer);
30 }
31
```

```
32 // for reset arcs
33 if(pnet->reset_arcs_map != NULL){
34     // get places to reset for a given transition
35     pnet_matrix_t *places_reset = pnet_matrix_extract_col(pnet->
       reset_arcs_map, transition);
36 
37     // negate so its a 0 for a place to be reset, that way we can then
       multiply it with the places element by element, zeroing out the
       places where needed
38     pnet_matrix_t *places_reset_neg = pnet_matrix_neg(places_reset);
39     pnet_matrix_delete(places_reset);
40 
41     pnet_matrix_t *buffer = pnet_matrix_mul_by_element(places_res,
       places_reset_neg);
42     pnet_matrix_copy(places_res, buffer);
43 
44     pnet_matrix_delete(places_reset_neg);
45     pnet_matrix_delete(buffer);
46 }
```

Fonte: Do autor.

3.2.5 Temporização e assincronia

Nota-se que no código 11 que a transição só é de fato disparada caso não haja matriz de temporização ou quando a temporização para a mesma for zero, porém quando há temporização, a mesma deve iniciar o processo de contagem juntamente com outras transições que possam começar a contar, isso pelo fato de que o início da contagem pode ocorrer de forma simultânea, somente o disparo das transições deve ser único, comportamento que pode-se tornar complexo, para tanto deve se visionar um sistema de contagem que garanta o mesmo.

Dentro deste *thread* código erá executado para efetuar a contagem das transições temporizadas quando forem sensibilizadas, porém, como visto, isso pode levar a situações onde podem haver múltiplas transições simultâneas. Para lidar com esse aspecto e facilitar a organização e contagem destas transições, é implementado uma estrutura de dados específica, uma fila priorizada.

A implementação da fila é feita da seguinte forma, uma lista ligada que guarda elementos especiais, responsáveis pela metainformação de cada transição temporizada, e que trabalha a inserção e retirada dos elementos conforme a prioridade da contagem atual, de forma que os elementos com menor quantidade de tempo restante fiquem ao fim da lista, pois tem maior prioridade, assim podemos apenas verificar o último elemento da lista para saber se o mesmo deve ser disparado ou não, sem necessidade de checar o restante dos elementos.

Os elementos da lista ligada são transições, que são da seguinte forma:

Código 13 – Elemento de transição temporizado

```
1 typedef struct{
2   size_t transition;
3   int start;
4   int delay;
5 }transition_t;
```

Fonte: Do autor.

Onde guarda-se o número da transição, o tempo inicial t_0 onde a transição foi ativada e o tempo total de temporização t. Para realizar a inserção na lista teremos uma função `transition_queue_push`, qual insere os elementos de menor $t_0 + t$.

Código 14 – Inserção de elemento na lista priorizada por tempo

```
1 typedef struct{
2   size_t transition;
3   int start;
4   int delay;
5 }transition_t;
```

Fonte: Do autor.

Percorre-se a lista com um laço de repetição, começando ao final, onde estão as transições mais próximas de executar, em direção ao início. Verifica-se então se a transição já está presente na lista, se sim retornar da chamada, pois não há necessidade de inserção. Se o tempo final $t_0 + t$ for menor que o da transição atual, inserir o elemento a frente daquela transição, e caso se chegar ao início da lista sem que não seja inserido nada, inserir ao início da lista como a transição mais longe de ser executada. Tal processo é assegurado pelo uso de um `mutex` definido para a instância da lista, evitando possíveis erros por *race conditions*.

Essa chamada foi usada no código 11 linha 21, onde verifica-se que a inserção é realizada para quantas transições forem sensibilizadas naquele momento.

Para retirada da transição da lista implementa-se a chamada `transition_queue_pop`, que é definida como:

Código 15 – Retirada de elemento na lista priorizada por tempo

```
1 int now = CLOCK_TO_MS(clock());
2
3 transition_t node_value = queue_value(queue->q->last, transition_t);
4 if((now - node_value.start) >= node_value.delay){
5   *transition = node_value;
6
```

```
7   queue_node_t *node = queue->q->last;
8   queue->q->last = node->prev;
9
10  if(node->prev != NULL)
11    node->prev->next = NULL;
12
13  queue->q->size--;
14
15  if(queue->q->first == node)
16    queue->q->first = NULL;
17
18  free(node->value);
19  free(node);
20
21  queue_unlock();
22  return true;
23 }
```

Fonte: Do autor.

Na linha 1 obtém se o tempo atual em milissegundos usando a função `clock`, que nos dá a quantidade de tempo passado desde o início do programa até o momento atual. Então verifica-se se o último elemento da lista é passível de ser retirado, então compara-se o tempo passado até o momento desde a inserção da transição com o tempo de temporização total. Caso tenha sido alcançado, a função retorna sucesso e o elemento pode ser acessado pelo ponteiro de referência passado como parâmetro, caso contrário, o retorno é zero.

Assim, a contagem no *thread*, dado pela própria estrutura da rede de petri, pode ser feita com a seguinte rotina de execução:

Código 16 – Rotina de execução do *thread* temporizador

```
1 while(1){
2     pthread_testcancel();
3
4     transition_t transition;
5     if(transition_queue_pop(pnet->transition_to_fire, &transition)){
6         pnet_sense(pnet);
7         if(pnet->sensitive_transitions->m[0][transition.transition] ==
      1){       // re check sensibility
8             pnet_move(pnet, transition.transition);
                // FIRE!! move tokens and call callback
9             if(pnet->function != NULL)
10                pnet->function(pnet, transition.transition, pnet->
      user_data);
11        }
```

```
12      }
13 }
```

Fonte: Do autor.

Observa-se que a rotina é executa de forma infinita, sempre tentando retirar elementos da fila priorizada, caso não consiga, o laço simplesmente se repete. Lembrando que este processo ocorre de forma paralela ao código da rede. Caso o retorno seja verdadeiro podemos verificar se a transição ainda encontra se sensibilizada, comportamento desejado, pois se ela deixar de ser sensibilizada então a mesma não deve ser disparada. Caso se encontrar, então disparamos a rede e executamos o *callback* de retorno, deixando o usuário da rede atento que uma transição foi executada de forma assíncrona, fora to tempo de execução da chamada `pnet_fire`.

Tal *callback* é definido na criação da rede, visto no código 7 juntamente a fila priorizada, uma simples função que é executada de forma assíncrona e recebe como parâmetros os dados do usuário, dados no momento da criação, e o número da transição que foi executada, que pode ser utilizado para lógicas auxiliares de escolha do usuário.

As funções de checagem de sensibilização e de movimentação de fichas são *thread* safe também, graças a um outro *mutex* localizado também na estrutura rede de petri, assim, quando as transições temporizadas ocorrerem, elas não entrarão em conflito com as transições instantâneas da execução normal.

Com essa última definição chegamos ao final da estrutura dada no código 7, onde cada elemento definido além das matrizes de definição, os mapas, é necessário para que a rede funcione como um todo.

3.2.6 Serialização de matrizes

A serialização de dados é bastante simples quando têm-se acesso ao baixo nível, como é o caso da linguagem C, onde podemos ler dados da memória e interpretá-los de maneira livre. Uma das formas de fazer isso é utilizando uma técnica chamada *pointer casting*, onde uma estrutura é definida e então atrelada como ponteiro a uma região de memória, funcionando para leitura e escrita, isso se torna uma forma conveniente de se serializar e desserializar dados.

As matrizes que compõe nossa rede de petri ocupam um espaço relevante de memória, portanto guardar uma matriz n por m de números inteiros ocuparia minimamente $n \cdot m \cdot 4$ *bytes*, tamanho que cresce exponencialmente, porém, como a maioria das matrizes possui muitos valores nulos, devido às relações de arcos e marcação inicial, podemos tomar vantagem disso e gravar apenas os valores não nulos, usando marcadores de linha e coluna.

Define-se assim então para cada matriz um cabeçalho de memória, onde serão contidas a quantidade de linhas e colunas e então os valores não nulos.

Código 17 – Cabeçalho da serialização de matriz

```
typedef struct{
    uint32_t x;
    uint32_t y;
    uint8_t first_byte;
}pnet_matrix_header_t;

uint32_t data[] = {
    4, 4,
    0x80000000, 3, 1,
    0x80000001, 0, 64, 3, 1,
    0x80000003, 2, 0x7FFFFFFF, 3, 16
};

pnet_matrix_header_t *header = (pnet_matrix_header_t*)data;
```

Fonte: Do autor.

Como estamos usando matrizes de número inteiro, será fixado o tamanho em *bits* das variáveis como `uint32_t` para índices e `int32_t` para os valores.

Os valores serão escritos da seguinte forma, marca-se primeiro a linha, caso haja no mínimo um valor não nulo nela, usando o índice da linha e marcando o último *bit* da representação binária do número, como forma de diferenciar que este número se trata do índice da linha. Assim o número a ser gravado será de `n = c | 0x80000000`, e a leitura pode ser feita como `c = n & ~0x80000000`.

Dada a linha, cada valor da linha será um par de dois números, o índice da coluna e o valor.

Para seguinte matriz de exemplo teremos:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 64 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \text{0x7FFFFFFF} & 16 \end{bmatrix}$$

$$Ser = \left\{4, 4, \text{0x80000000}, 3, 1, \text{0x80000001}, 0, 64, 3, 1, \text{0x80000003}, 2, \text{0x7FFFFFFF}, 3, 16\right\}$$

Assim, dado um *array* de memória de tipo `uint32_t *` de tamanho `size_t bytes`, podemos realizar um casting conforme o código 17 para encontrar o tamanho da matriz e assim pré alocar a matriz para ler novamente os dados.

3.2.7 Serialização da Rede de petri

Dada a serialização das matrizes, a rede de petri torna-se fácil, pois é composta de várias matrizes, deixando somente a definição de alguns metadados necessários. O cabeçalho do arquivo pode ser dado como:

Código 18 – Cabeçalho de arquivo da rede de petri

```
#pragma pack(push,1)
typedef struct{
    char magic[4];
    uint16_t version;
    uint8_t valid;
    uint8_t matrix_size;
    uint32_t size;
    uint32_t crc32;
    uint32_t num_places;
    uint32_t num_transitions;
    uint32_t num_inputs;
    uint32_t num_outputs;

    uint32_t neg_arcs_map_size;
    uint8_t neg_arcs_map_first_byte;
    /**
     * ...
     * neg_arcs_map
     * pos_arcs_map
     * inhibit_arcs_map
     * reset_arcs_map
     * places_init
     * transitions_delay
     * inputs_map
     * outputs_map
     * places
     * sensitive_transitions
     * outputs
     * inputs_last
    */
}pnet_file_header_t;
#pragma pack(pop)
```

Fonte: Do autor.

Nota-se que o primeiro membro, *magic*, trata-se de uma identificação visual, este guarda quatro caracteres de texto de valor sempre “PNET”, de forma a indicar que este arquivo é um arquivo de tipo rede de petri, seguido do número de versão do arquivo, que têm-seu caso de uso caso a partir deste valor no cabeçalho ocorra mudanças no

futuro, os programas podem alterar a leitura e escrita com base no número de versão, dando flexibilidade e opções de versionamento.

Temos o valor binário de validez da rede de petri, seguido do valor de tamanho das matrizes, que como discutido anteriormente foi dado como de trinta e dois *bits*, `0x20`, mas poderia ser dado como matrizes de dezesseis *bits*, `0x10`, ou outros tamanhos caso a necessidade de armazenamento de memória.

O campo `size` armazena o valor total de *bytes* ocupado pela serialização rede excepto pelo cabeçalho fixo, ou seja, a partir do campo `neg_arcs_map_size`. Existe também a checagem CRC32 (GNU, 2012) dos dados serializados, exceto pelo cabeçalho, para que se possa detectar erros por corrupção de memória. E finalmente as quantidades de elementos, transições, lugares, entradas e saídas, e então as matrizes e seus tamanhos em *bytes*, que compõem nossa rede de petri.

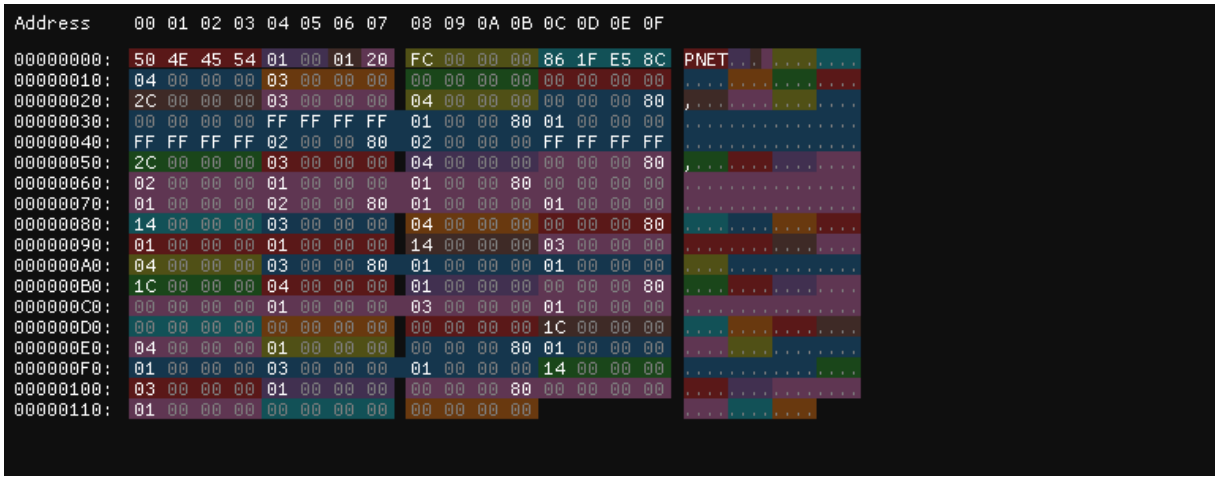
Matrizes serializadas

- Matriz de arcos de peso negativos
- Matriz de arcos de peso positivos
- Matriz de arcos negados
- Matriz de arcos de *reset*
- Matriz de marcação inicial
- Matriz de temporização das transições
- Matriz de entrada
- Matriz de saída
- Matriz de estado atual dos lugares
- Matriz de transições sensíveis
- Matriz de estados das saídas
- Matriz do estado anterior das entradas

Como temos um número fixo de matrizes, basta realizar a leitura do tamanho de cada uma, e então a leitura da matriz em si.

Para este arquivo, dado o nome da biblioteca e do indicador no cabeçalho, sua extensão será denominada `.pnet`. Um arquivo preenchido tem o seguinte formato visto sobre um analisador hexadecimal:

Figura 10 – Visão do arquivo pnet em um analisador hexadecimal



3.3 Algoritmo de compilação para lista de instrução

A compilação para lista de instrução foi baseada no método de compilação para *Ladder* apresentado no trabalho Petrilab (SOUZA, 2015), porém aqui, para lista de instrução. Fato que é possível devido à característica de que programação *Ladder* pode ser gerada a partir de uma lista de instrução e vice-versa.

Como está sendo implementando um compilador para lista de instrução, sabe-se que funcionalidades de borda de eventos e temporizadores são de fácil acesso, dispensando-se a criação de estruturas adicionais para implementação dos mesmos, e portanto, podemos compilar a rede de petri de forma mais simples, mapeando estas funcionalidades diretamente conforme necessário. Diferente da biblioteca C, onde foi desenvolvido o funcionamentos dessas funcionalidades.

Para a compilação serão definidas as seções assim como implementados na biblioteca em C.

- Marcação inicial
- Sensibilização
- Disparo
- Lógica de saída

Cada passo será desenvolvido de forma sequencial, para ter-se esta ordem de execução de código no PLC dado.

Para compilação para código C usaremos uma função facilitadora, `string_cat_fmt` , que é uma função feita com base na biblioteca C padrão, parecida com a função `printf` mas que recebe uma string de formatação, a expande e concatena o resultado

em um buffer de saída. Assim temos ao início um buffer vazio, e vamos inserindo as instruções até que ao final tenhamos o programa compilado.

Também iremos utilizar as constantes `INITIAL_RE`, que se refere a memória `M8002` que é ativa por apenas um ciclo quando o PLC é inicializado, e `ALWAYS`, a memória especial `M8000`, que é sempre ativa durante a execução do código.

Como parte do algoritmo de compilação devemos definir algumas memórias temporárias e memórias para o estado da rede. Será definido que as transições e sua sensibilização via temporizador e entradas serão feitas e então armazenadas em uma memória simples, no PLC, uma memória `M`. Transições temporizadas utilizaram uma memória auxiliar extra. Para os lugares necessitamos armazenar a quantidade de fichas atual para cada um, então utilizaremos uma unidade de memória não binária, uma memória `word` de 16 bits, no PLC, memória tipo `D`. As entradas e saídas serão respectivamente `X` e `Y`.

Ainda iremos ter um pulo condicional usado para satisfazer a condição definida na seção 2.1.4, dado no PLC pela instrução `J`, para marcação, e `CJ` para o pulo condicional.

Também iremos considerar a posição inicial das memórias, também as entradas, saídas e o índice do pulo condicional. Todas essas variáveis internas necessitam de um índice numérico e, portanto, o algoritmo de compilação deverá ser ciente do índice inicial para cada uma delas, para que o possa incrementar conforme a quantidade de transições, lugares, entradas e saídas.

Temos assim a seguinte chamada de função para realização da compilação da rede de petri para lista de instrução. Percebe-se que para compilação, basta que tenhamos as informações da rede de petri e das posições iniciais.

Código 19 – Definição da função de compilação para lista de instrução

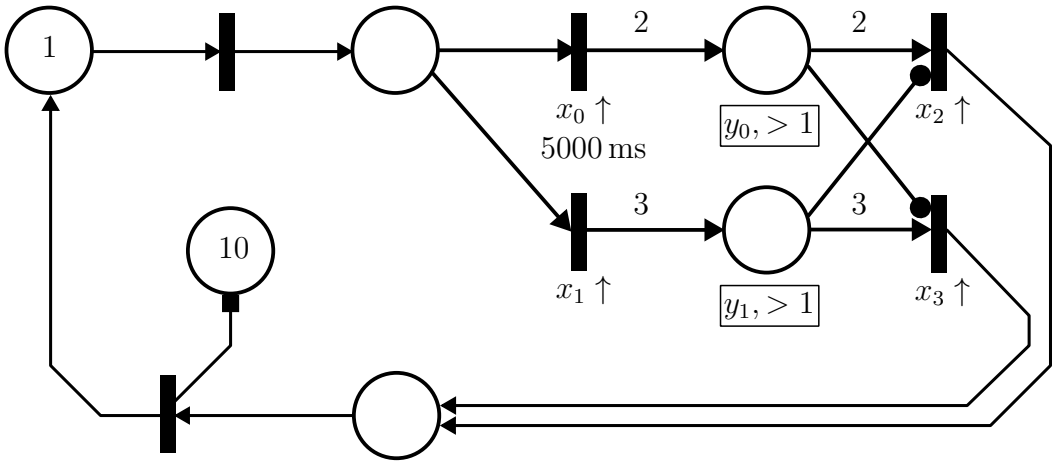
```
1 char *pnet_compile_il_weg_tpw04(pnet_t *pnet, int input_offset, int
    output_offset, int transition_offset, int place_offset, int
    timer_offset, int timer_min, int jump_offset);
```

Fonte: Do autor.

Dada uma rede de petri exemplo, figura 11, vamos analisar o processo de compilação, mostrando a implementação das diferentes seções, a lista de instrução para cada componente e a implementação em código C. A rede apresenta todas as funcionalidades propostas, exceto pelo arco de *reset*, e demonstra uma implementação simples de um processo de exclusão mútua utilizando arcos negados, redundante, pois há apenas uma ficha em cada lugar por vez, mas demonstra um processo lógico simples.

Na rede de petri, abaixo das transições temos o evento de borda relacionado a entrada, abaixo dos lugares temos a legenda da saída e a condição lógica de ativação

Figura 11 – Rede de petri exemplo



da mesma.

Em lista de instrução, começaremos a compilação pela marcação inicial, onde é utilizada a memória especial `M8001` e a instrução `MOV` para mover as fichas para os lugares iniciais, que são no PLC memórias do tipo *word*.

A compilação é feita basicamente pela varredura da matriz de marcação inicial, ou seja, um laço de repetição que irá iterar pelos lugares. Assim também será realizada a implementação para os arcos, entradas e saídas, sempre utilizando as matrizes de definição, os mapas, para que possamos traduzir em código de lista de instrução.

Código 20 – Exemplo de lista de instrução - Marcação inicial

```
1 LD M8002
2 MOV K1 D200
```

Fonte: Do autor.

Em código C isso pode ser implementado por um laço de repetição pelos lugares da rede de petri, onde para cada lugar, se há marcação, concatena-se a expressão em formato de lista de instrução no texto de saída final.

Código 21 – Compilação da marcação dos lugares iniciais

```
1 for(size_t place = 0; place < pnet->num_places; place++){
2   if(pnet->places_init->m[0][place]){
3     string_cat_fmt(buffer, "LD M%u\nMOV K%u D%u\n", BUFFER_SIZE,
    INITIAL_RE, pnet->places_init->m[0][place], place + place_offset);
4   }
5 }
```

Fonte: Do autor.

A sensibilização das transições é relativamente fácil, e será dada como os eventos de entrada juntamente as condições de disparo, mapeadas para uma memória auxiliar que irá ser usada para movimentar as fichas. Para entradas normais temos:

Código 22 – Exemplo de lista de instrução - Sensibilização

```
1 LDP X1
2 AND>= D201 K1
3 OUT M32
```

Fonte: Do autor.

Para transições temporizadas, devemos além de ter se as condições de sensibilização e eventos de entrada, ter-se um temporizador. Na referência WEG TPW04, tal temporizador é dado em código como `OUT T# K#`, onde `T` é o índice do temporizador e `K` a quantidade de tempo. Para transições com entrada, gera-se apenas um pulso para ativação do temporizador, por isso devemos utilizar uma memória auxiliar extra para garantir que após um pulso, o temporizador continue contando, utilizando um arranjo chamado de selo.

O Selo é basicamente uma memória auxiliar que é ativada juntamente a saída original, no nosso caso, o temporizador. E ainda mais, utiliza-se o estado desta memória como entrada auxiliar a entrada de pulso original utilizando a operação lógica “ou”. O resultado é que quando o pulso de entrada ativa a saída e a memória auxiliar, esta mesma mantém as saídas acionadas, mesmo sem o pulso de entrada.

Para desativar o selo devemos adicionar utilizar a operação lógica “e” negada com as entradas e uma memória externa, que enquanto estiver em estado desligada, permitirá que a entrada acione as saídas e o selo aconteça, porém quando acionada, negará o valor das entradas, inclusive da memória selo, desativando o selo e também o temporizador. Como estamos interessados em um pulso desse temporizador para indicar sensibilização da transição, será utilizado a memória `T` do próprio temporizador, que se torna ativa quando o mesmo termina a contagem.

Ainda mais, a memória `T` será utilizada para com uma memória `M`, análoga a memória auxiliar de uma transição normal, para indicar que a transição temporizada encontra-se agora sensibilizada. Assim temos o efeito geral de, quando ocorrer um pulso de entrada e as condições de disparo forem verdadeiras, um selo acontece, habilitando a contagem do temporizador que após chegar ao final da contagem irá, romper o selo, se auto desligando, e acionando com um pulso a memória auxiliar da transição.

Código 23 – Exemplo de lista de instrução - Sensibilização com temporizador

```
1 LDP X0
2 OR M37
3 ANI T1
4 AND>= D201 K1
5 MPS
6 OUT T1 K50
7 MPP
8 OUT M37
```

Fonte: Do autor.

Nota-se que, para a memória auxiliar do selo, será utilizada a mesma região de memória das memórias auxiliares das transições, porém deslocada n transições acima do deslocamento dado ao compilador. Assim, para uma rede de seis transições, se a segunda transição for temporizada, e o deslocamento inicial das memórias for de trinta, então temos que as memórias das transições serão `M30`, `M31`, `M32` e a memória auxiliar de selo da terceira transição será `M37`, dado à soma de deslocamento mais quantidade de transições mais índice da transição, $30 + 6 + 1 = 37$.

A compilação das entradas, condições e timers pode ser dada por um laço de repetição pelas transições e entradas. Caso não houverem entradas, a transição sempre dispara, e caso houverem, verifica-se se são temporizadas ou não. Caso não sejam temporizadas, implementamos o código de lista de instrução como no código 22, caso sejam temporizadas, como no código 23. Varremos assim, as matrizes de temporização e de entrada e verificamos se para dada transição a entrada e temporização correspondente, se houver. Em código C temos:

Código 24 – Verificação de evento de entrada e de temprização para uma transição

```
1 size_t input;
2 pnet_event_t input_evt = pnet_event_none;
3 if(pnet->inputs_map != NULL){
4   for(input = 0; input < pnet->num_inputs; input++){
5     if(pnet->inputs_map->m[input][transition] != pnet_event_none){
6       input_evt = pnet->inputs_map->m[input][transition];
7       break;
8     }
9   }
10 }
11
12 size_t delay =
13   pnet->transitions_delay != NULL ?
14   pnet->transitions_delay->m[0][transition] :
15   0;
16
```

```
17 switch(input_evt){
18   case pnet_event_pos_edge:
19     string_cat_fmt(buffer, "LDP X%u\n", BUFFER_SIZE, input +
     input_offset);
20   break;
21
22   case pnet_event_neg_edge:
23     string_cat_fmt(buffer, "LDF X%u\n", BUFFER_SIZE, input +
     input_offset);
24   break;
25
26   case pnet_event_any_edge:
27     string_cat_fmt(buffer, "LDF X%u\nORP X%u\n", BUFFER_SIZE, input +
     input_offset, input + input_offset);
28   break;
29
30   default:
31   case pnet_event_none:
32     string_cat_fmt(buffer, "LD M%u\n", BUFFER_SIZE, ALWAYS);
33   break;
34 }
```

Fonte: Do autor.

Note que na linha 17 em diante, com base no evento de entrada, já pode-se inserir a condição de entrada no código compilado.

Com as informações da transição, pode-se então compilar o código para mesma.

Código 25 – Compilação da sensibilização da transição

```
1 // add retentive memory to count after event and reset after count
2 if(delay){
3   string_cat_fmt(buffer, "OR M%u\nANI T%u\n", BUFFER_SIZE, transition
     + pnet->num_transitions + transition_offset, transition +
     timer_offset);
4 }
5
6 for(size_t place = 0; place < pnet->num_places; place++){
7   if(
8     pnet->inhibit_arcs_map != NULL &&
9     pnet->inhibit_arcs_map->m[place][transition]
10  ){
11    string_cat_fmt(buffer, "AND= D%u K0\n", BUFFER_SIZE, place +
     place_offset);
12  }
13
14  if(
15    pnet->neg_arcs_map != NULL &&
```

```
16      pnet->neg_arcs_map->m[place][transition]
17    ){
18      string_cat_fmt(buffer, "AND>= D%u K%u\n", BUFFER_SIZE, place +
       place_offset, -pnet->neg_arcs_map->m[place][transition]);
19    }
20 }
21
22 // add timer, retentive memory and load timer output
23 if(delay){
24   string_cat_fmt(buffer, "MPS\nOUT T%u K%u\n", BUFFER_SIZE, transition
       + timer_offset, delay / timer_min);
25   string_cat_fmt(buffer, "MPP\nOUT M%u\n", BUFFER_SIZE, transition +
       pnet->num_transitions + transition_offset);
26   string_cat_fmt(buffer, "LD T%u\n", BUFFER_SIZE, transition +
       timer_offset);
27 }
28
29 string_cat_fmt(buffer, "OUT M%u\n", BUFFER_SIZE, transition +
     transition_offset);
```

Fonte: Do autor.

Como a entrada já foi inserida, basta apenas inserir as condições dos arcos, arcos de peso negativo com a instrução AND>= e os arcos negados com a instrução AND=, linhas 6 à 20 do código acima. E também a saída da sensibilização da transição, linha 29.

Nota-se nas linhas 2 e 23, o tratamento especial caso houver temporização para a transição, onde irá ser adicionado o código relevante a temporização, conforme o exemplo do código 23.

Para o disparo e movimentação das fichas, deve-se adicionar as instruções de aritméticas conforme o tipo de arco. Arcos de peso negativos e positivos, que irão usar as instruções de subtração e adição para movimentação das fichas, SUB e ADD respectivamente, e o arco de *reset*, que usa a instrução de movimentação MOV com constante zero.

Para cada transição, se carrega o valor de sensibilização da mesma, e adicionam-se as instrução de movimentação. Caso hajam várias, utiliza-se a instrução MPS e relacionadas para se ativar as diversas saídas. Agora para cada lugar, adicionam-se as instruções de ADD para os arcos positivos, SUB para os arcos negativos e de MOV para os arcos de reset. Entre cada instrução devemos ler o valor da sensibilização da transição usando MRD. Na última instrução de movimentação usamos MPP para limpar o valor da sensibilização da pilha e escrevemos a última instrução.

Como visto na seção 2.1.4, deve-se garantir o disparo de apenas uma transição, para isso irá se utilizar um pulo condicional, que irá impedir a execução de outros

disparos. O pulo será para uma marcação de execução, por exemplo `P0`, que é localizada na instrução a frente da última instrução da última movimentação, efetivamente pulando todas os disparos. A instrução para o pulo é `CJ`.

Código 26 – Exemplo de lista de instrução - Disparo das transições

```
1 LD M35
2 MPS
3 ADD D200 K1 D200
4 MRD
5 SUB D204 K1 D204
6 MRD
7 MOV K0 D205
8 MPP
9 CJ P0
```

Fonte: Do autor.

Em código C temos:

Código 27 – Compilação da movimentação das fichas

```
1 // pnet move tokens
2 // LD (M00 | M8000) MPS ADD D000 K000 D000 MRD ... MPP SUB D000 K000
    D000
3 arc_t *arcs = malloc(pnet->num_places * 3 * sizeof(arc_t));
4 size_t arc = 0;
5 for(size_t transition = 0; transition < pnet->num_transitions;
    transition++){
6
7   // run through places and save arcs
8   for(size_t place = 0; place < pnet->num_places; place++){
9     if(
10      pnet->neg_arcs_map != NULL &&
11      pnet->neg_arcs_map->m[place][transition]
12    ){
13      arcs[arc].type = arc_type_weighted_neg;
14      arcs[arc].place = place;
15      arcs[arc].weight = pnet->neg_arcs_map->m[place][transition];
16      arc++;
17    }
18
19    if(
20      pnet->pos_arcs_map != NULL &&
21      pnet->pos_arcs_map->m[place][transition]
22    ){
23      arcs[arc].type = arc_type_weighted_pos;
24      arcs[arc].place = place;
```

```
25        arcs[arc].weight = pnet->pos_arcs_map->m[place][transition];
26        arc++;
27      }
28
29      if(
30        pnet->reset_arcs_map != NULL &&
31        pnet->reset_arcs_map->m[place][transition]
32      ){
33        arcs[arc].type = arc_type_reset;
34        arcs[arc].place = place;
35        arc++;
36      }
37    }
38
39    // write arcs
40    if(arc > 0){
41      string_cat_fmt(buffer, "LD M%u\n", BUFFER_SIZE, transition +
      transition_offset);
42
43      for(size_t fire = 0; fire < arc; fire++){
44        if(arc > 1){
45          if(fire == 0)
46            string_cat_raw(buffer, "MPS\n");
47          // else if(fire == (arc - 1))
48          //   string_cat_raw(buffer, "MPP\n");
49          else
50            string_cat_raw(buffer, "MRD\n");
51        }
52
53        switch(arcs[fire].type){
54          case arc_type_weighted_neg:
55            string_cat_fmt(buffer, "SUB D%u K%u D%u\n", BUFFER_SIZE,
      arcs[fire].place + place_offset, -arcs[fire].weight, arcs[fire].
      place + place_offset);
56          break;
57
58          case arc_type_weighted_pos:
59            string_cat_fmt(buffer, "ADD D%u K%u D%u\n", BUFFER_SIZE,
      arcs[fire].place + place_offset, arcs[fire].weight, arcs[fire].
      place + place_offset);
60          break;
61
62          case arc_type_reset:
63            string_cat_fmt(buffer, "MOV K%u D%u\n", BUFFER_SIZE, 0, arcs
      [fire].place + place_offset);
64          break;
65        }
```

```
66      }
67
68      string_cat_fmt(buffer, "MPP\nCJ P%u\n", BUFFER_SIZE, jump_offset);
69    }
70
71    arc = 0;
72  }
```

Fonte: Do autor.

Nas linhas 8 à 37 iremos percorrer as matrizes de arcos de peso negativos, positivos e de arco de *reset*, necessário pois assim saberemos a quantidade de instruções à serem inseridos para uma transição.

Na linha 41, começamos carregando o valor de sensibilidade da transição verificado anteriormente. Na linha 43, um laço de repetição por todos os arcos computados anteriormente, verificando se há necessidade de inserção das instruções de memória `MPS, MRD, MPP`.

Na linhas 53 verificamos o tipo do arco a ser adicionado, adicionando as movimentações das fichas, para arco de peso negativo, a instrução `SUB`, arco positivo a instrução `ADD` e arco de *reset*, instrução `MOV`.

Por fim temos as saídas, as quais só utilizam a comparação entre os lugares e uma constante dada pela matriz de saída, e após a comparação, acionam a saída física.

Código 28 – Exemplo de lista de instrução - Saídas

```
1 P0
2 LD>= D202 K1
3 OUT Y0
```

Fonte: Do autor.

Nota-se na linha 1 a definição da marcação para o pulo condicional, necessário para garantir o disparo único da transições.

Implementando, faremos um laço de repetição entre os lugares e as saídas, para cada lugar, verifica-se se há uma saída atrelada, se houver, verifica-se se o lugar possui a mesma quantidade ou mais fichas do que necessárias, usando a instrução `LD>=` para com a memória do lugar. Então, em seguida usamos a instrução `OUT` para redirecionar o valor da comparação para a saída atrelada.

Em código C temos:

Código 29 – Compilação para as codições de saída

```
1 if(pnet->outputs_map != NULL){
2   for(size_t place = 0; place < pnet->num_places; place++){
3     for(size_t output = 0; output < pnet->num_outputs; output++){
```

```
4       if(pnet->outputs_map->m[place][output])
5         string_cat_fmt(buffer, "LD>= D%u K%u\nOUT Y%u\n", BUFFER_SIZE,
      place + place_offset, pnet->outputs_map->m[place][output], output
     + output_offset);
6      }
7   }
8 }
```

Fonte: Do autor.

Na linha 2 e 3, temos os laços que irão varrer os lugares e saídas, onde verificaremos a quantidade de fichas e acionaremos as saídas, na linha 4 vemos a concatenação da instrução inteira usando `LD>=` e `OUT`.

4 Conclusão

Dados os objetivos e aspirações deste projeto, pode-se afirmar que tanto a biblioteca em C quanto o algoritmo de compilação alcançaram o resultado esperado. Para biblioteca C (PETERSON, 2023) pode-se listar as seguintes estatísticas de projeto:

- Commits realizados: 42
- Arquivos: 21
- Linhas comentadas de código: 947
- Linhas de código escrito: 3368
- Período de trabalho: Maio 2022 - Junho 2023

Dada as especificações de projeto delimitadas, a checagem e testes deixam a biblioteca C com aspecto robusto e de pronto uso. A execução e temporização funcionam com base na biblioteca padrão C e a biblioteca `pthread`, e portanto, irão trazer performance e generalidade para aplicações desktop e compatibilidade com sistemas embarcados, através da implementação genérica a base de *threads*.

Tendo opção de serialização da rede de petri em um formato de arquivo robusto, com versionamento e checagem de erro, capaz também de guardar estado, traz-se a possibilidade de um opção extra a compilação, onde dispositivos podem embarcar essas redes em formato de arquivo e carregá-las em memória no momento de execução.

A própria estrutura traz um aspecto trivial para utilização da mesma, seja na utilização das entradas e saídas de forma direta, no acesso ao estado interno, no aspecto assíncrono e também na parte da compilação, onde para maioria dos casos de utilização, dois laços de repetição, um para os lugares e outro para as transições, podem ser usados para varrer toda rede de petri, como por exemplo, utilizado na compilação para lista de instrução.

Trabalhos futuros podem ser feitos sobre a parte assíncrona de temporização visto que temos o objectivo de levar essa biblioteca para sistemas embarcados. A compilação condicional de um sistema assíncrono baseado em timers de *hardware* traria toda precisão e acurácia que se espera do sistema embarcado, com a rede de petri nele embutida.

O algoritmo de compilação para rede de petri foi desenvolvido de maneira bastante simples, dado a estrutura da rede em mãos e o fato da lista de instrução já suportar alguns aspectos de arquitetura já de forma nativa, como eventos de borda e temporização, abrindo então portas para implementação de outras plataformas que também

usam a lista de instrução, por vezes sendo extremamente similar a referência WEG TPW04 (WEG, 2015). Podendo ainda também possuir conversão da lista para outras linguagens como o próprio *Ladder*, aumentando o alcance dessa implementação além da própria lista de instrução.

Em sumo, os objetivos foram alcançados e espera-se que este trabalho sirva como semente para outras implementações e avanços, bem como sirva para difusão da utilização de redes de petri para modelagem de sistemas a eventos discretos, em especial a automação de sistemas industriais.

Referências

DUFOURD, C.; FINKEL, A.; SCHNOEBELEN, P. Reset nets between decidability and undecidability. In: LARSEN, K. G.; SKYUM, S.; WINSKEL, G. (Ed.). *Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 103–115. ISBN 978-3-540-68681-1.

GHEZZI, C. et al. A unified high-level petri net formalism for time-critical systems. *IEEE Transactions on software engineering*, IEEE Computer Society, v. 17, n. 2, p. 160, 1991.

GITHUB. *Github*. 2023. Disponível em: https://github.com. Acesso em: 17 jun. 2023.

GNU. *libiberty - CRC32*. 2012. Disponível em: https://github.com/gcc-mirror/gcc/blob/master/libiberty/crc32.c. Acesso em: 17 jun. 2023.

GNU. *GNU is not Unix*. 2023. Disponível em: https://www.gnu.org. Acesso em: 17 jun. 2023.

HILLSTON, J. *Stochastic Petri Nets*. 2009. Disponível em: https://www.inf.ed.ac.uk/teaching/courses/pm/handouts/stochasticpetrinets.pdf. Acesso em: 20 jun. 2023.

IEEE. *POSIX*. 2023. Disponível em: http://get.posixcertified.ieee.org. Acesso em: 17 jun. 2023.

International Electrotechnical Commission. *IEC 61131-3: Programmable Controllers - Part 3: Programming Languages*. Geneva, Switzerland, 2013.

ISO. *High-level Petri Nets - Concepts, Definitions and Graphical Notation*. 2000. Disponível em: http://www.petrinets.info/docs/pnstd-4.7.1.pdf. Acesso em: 20 jun. 2023.

IZHIKEVICH, E. M. *Petri Net*. 2011. Disponível em: http://www.scholarpedia.org/article/Petri_net.

JENSEN, K. *Coloured Petri Nets : Basic Concepts, Analysis Methods and Practical Use*. 2. ed. [S.l.]: Springer Berlin Heidelberg, 1997. ISBN 3-540-60943-1.

KAID, H. et al. Applications of petri nets based models in manufacturing systems: A review. In: *Proceedings of the International Conference on Operations Excellence & Service Engineering, Orlando, FL*. [S.l.: s.n.], 2015. p. 516–28.

LEVESON, N.; STOLZY, J. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, SE-13, n. 3, p. 386–397, 1987.

LINUX FOUNDATION. *pthreads(7) Linux manual page*. 2021. Disponível em: https://man7.org/linux/man-pages/man7/pthreads.7.html. Acesso em: 20 jun. 2023.

MASLAR, M. Plc standard programming languages: Iec 1131-3. In: *Conference Record of 1996 Annual Pulp and Paper Industry Technical Conference*. [S.l.: s.n.], 1996. p. 26–31.

MIT. *MIT License*. 2023. Disponível em: https://mit-license.org. Acesso em: 17 jun. 2023.

MOREIRA, M. V.; BASILIO, J. C. Bridging the gap between design and implementation of discrete-event controllers. *IEEE Transactions on Automation Science and Engineering*, v. 11, n. 1, p. 48–65, 2014.

PETERSON, J. *pnet - a petri net library for C/C++*. 2023. Disponível em: https://github.com/Joao-Peterson/pnet. Acesso em: 17 jun. 2023.

PETRI, C. A. *Kommunikation mit Automaten*. Bonn, 1962.

SOUZA, A. L. de. *Petrilab: Uma plataforma para simulação e geração de diagramas ladder de controladores a eventos discretos modelados por redes de petri*. Tese (Doutorado) — Departamento de Engenharia Elétrica da Escola Politécnica, Universidade, 2015.

WEG. *Série TPW-04 - Manual de Programação*. 2015. Disponível em: https://static.weg.net/medias/downloadcenter/hd8/h4d/WEG-controlador-logico-programavel-tpw04-manual-de-programacao-10003853205-manual-portugues-br.pdf. Acesso em: 17 jun. 2023.

ZAITSEV, D. A. Toward the minimal universal petri net. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, v. 44, n. 1, p. 47–58, 2014.

ZHOU, M.; HRúZ, B. *Modeling and Control of Discrete-event Dynamic Systems*. 1. ed. [S.l.]: Springer London, 2007. ISBN 978-1-84628-872-2.