Universidade Federal de Ouro Preto
Escola de Minas
CECAU - Colegiado do Curso de
Engenharia de Controle e Automação

Douglas Meneses Barbosa

DESENVOLVIMENTO DE UMA APLICAÇÃO WEB COM O OBJETIVO DE CONSTRUIR E SIMULAR REDES DE PETRI

Monografia de Graduação

Ouro Preto, 2025

Douglas Meneses Barbosa

DESENVOLVIMENTO DE UMA APLICAÇÃO WEB COM O OBJETIVO DE CONSTRUIR E SIMULAR REDES DE PETRI

Trabalho apresentado ao Colegiado do Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como parte dos requisitos para a obtenção do Grau de Engenheiro de Controle e Automação.

Universidade Federal de Ouro Preto

Orientador: Prof. Dr. Danny Augusto Vieira Tonidandel

Coorientador: Não definido

Ouro Preto

2025

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
ESCOLA DE MINAS
DEPARTAMENTO DE ENGENHARIA CONTROLE E AUTOMACAO

UFOP
Universidade Federal de Ouro Preto

FOLHA DE APROVAÇÃO

Santos Dumont

Como construir um avião?

Monografia apresentada ao Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Engenharia de Controle e Automação

VERSÃO DE DEMONSTRAÇÃO

Aprovada em 26 de fevereiro de 2023

Membros da banca

[Doutor] - Carlos Chagas - Orientador (Universidade Federal de Ouro Preto)

[Doutora] - Nise da Silveira - Orientador (Universidade Federal da Bahia)

[Doutor] - Leopoldo Nachbin - (Instituto Nacional de Matemática Pura e Aplicada)

[Doutora] - Ruth Sonntag Nussenzweig - (Universidade de São Paulo)

Nise da Silveira, coorientadora do trabalho, aprovou a versão final e autorizou seu depósito na Bilbioteca de Trabalhos de Conclusão de Curso da UFOP em 04/07/2023.

Documento assinado eletronicamente por Nise da Silveira, **PROFESORA DE MAGISTÉRIO SUPERIOR**, em 04/07/2023, às 10:16, conforme horário oficial de Brasília, com fudamento no art. 6º, § 1º, do Decreto no 8.539, de 8 de outubro de 2015.

A autenticidade desde documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador 26021982 e o código CRC X3DF4F4

Agradecimentos

Os agradecimentos [são opcionais, e] vem aqui...

Júpiter leva 4332 dias para fazer uma revolução.

— Oliver Lodge.

Resumo

O resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (…) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chaves: latex. abntex. editoração de texto.

Abstract

This is the english abstract.

Key-words: latex. abntex. text editoration.

Lista de ilustrações

Sumário

1 Introdução

nome da aplicação: Online Petri Net Simulator

Em 1962, Carl Adam Petri, por meio de sua dissertação, mostrou para o mundo a sua criação, as redes de Petri (PETRI, 1962). Podemos definir uma rede de Petri como sendo uma ferramenta matemática para modelagem de sistemas concorrentes. Além da modelagem matemática, as redes de Petri podem ser ilustradas graficamente por meio de seus elementos, os lugares, as transições e os arcos. Além disso, com a evolução da industria e da tecnologia, as redes de Petri ganharam ainda mais relevância, uma vez que elas permitem a modelagem de diversos tipos de sistemas, em diferentes áreas.

A evolução da industria e da tecnologia também culminou com a popularização da internet (LINS, 2013). O surgimento do World Wide Web, em 1990, por Tim Berners-Lee, permitiu aos primeiros usuários da internet, como conhecemos hoje, a interação com um sistema de hipertexto. Com o passar dos anos, as aplicações web se tornaram cada vez mais comuns e sofisticadas. O que antes começou com páginas estáticas evoluiu para aplicações dinâmicas, interativas e de fácil acesso para a maioria das pessoas. Atualmente, se consegue ter uma experiência muito próxima as funcionalidades de um computador pessoal, com aplicações desktop, apenas manipulando abas em um navegador.

Diante da facilidade de acesso a aplicativos web por meio dos navegadores, como Google Chrome, Firefox, Safari, Edge, entre outros, surge a seguinte ideia: desenvolver uma aplicação web que possibilite aos usuários a criação e simulação do comportamento de redes de Petri, de forma simples e intuitiva. Para tal, torna-se necessário o conhecimento de tecnologias voltadas para o desenvolvimento web, como HTML5, CSS3 e JavaScript.

Além do conhecimento em desenvolvimento, será necessário compreender os conceitos e práticas de infraestrutura, viabilizando a disponibilidade e o acesso contínuo à aplicação web. Uma vez desenvolvida, a aplicação pode ser disponibilizada para acesso por qualquer usuário que possua conexão com a internet, facilitando a modelagem de redes de Petri.

1.1 Justificativas e Relevância

As redes de Petri representam uma poderosa ferramenta gráfica e matemática para a modelagem e análise de sistemas concorrentes e distribuídos. No entanto, o seu entendimento pode ser um desafio, especialmente para aqueles sem familiaridade com os conceitos e experiência matemática.

Atualmente, existem algumas ferramentas capazes de construir e simular redes de

Petri. Entretanto, muitas delas requerem um conhecimento mínimo de computação, para que possam ser instaladas em sistemas operacionais como Linux, Windows e MacOS. Além disso, algumas delas não são multiplataforma, restringindo o acesso dessas ferramentas pelas pessoas.

Nesse contexto, uma aplicação web, simples e intuitiva, atenderia as necessidades, tanto de usuários comuns, como o de estudantes e pesquisadores interessados em entender o funcionamento das redes de Petri. Através de uma aplicação web, o acesso é simplificado e facilitado, pois elimina a necessidade de instalações complicadas e pré-conhecimento técnico avançado.

1.2 Objetivo Geral

O desenvolvimento de uma aplicação web capaz de criar e simular o comportamento de redes de Petri.

1.3 Objetivos Específicos

- Desenvolvimento de um motor de simulação, capaz de simular o comportamento das redes de Petri criadas, possibilitando a visualização, por parte do usuário, do comportamento em diferentes cenários;
- Desenvolvimento de uma interface simples e intuitiva;
- Aprendizado de tecnologias voltadas para o desenvolvimento web;
- Criação de uma alternativa simples e de fácil acesso para o aprendizado de redes de Petri;
- Alocação da aplicação em um servidor, permitindo o acesso público.

1.4 Metodologia

Inicialmente, para o desenvolvimento da aplicação web, capaz de construir e simular o comportamento de redes de Petri, será necessário entender os requisitos e características mínimas para o funcionamento da aplicação. Analisando softwares já existentes que cumprem essa função, serão desenvolvidas as seguintes funcionalidades básicas:

- Área da tela em que a rede de Petri será renderizada;
- Botões para inserção de lugares, arcos, tokens e transições;

- Botão de simulação em que, ao ser acionado, irá permitir a análise do comportamento da rede de Petri criada;
- Opção de definir labels para os lugares, arcos e posições;
- Alteração de cor da transição quando houver os requisitos mínimos satisfeitos;
- Opção de importação e exportação de projetos.

Com as características básicas da aplicação definidas, será desenvolvido um diagrama de fluxo, ilustrado na figura 1, evidenciando o passo a passo de funcionamento da aplicação.

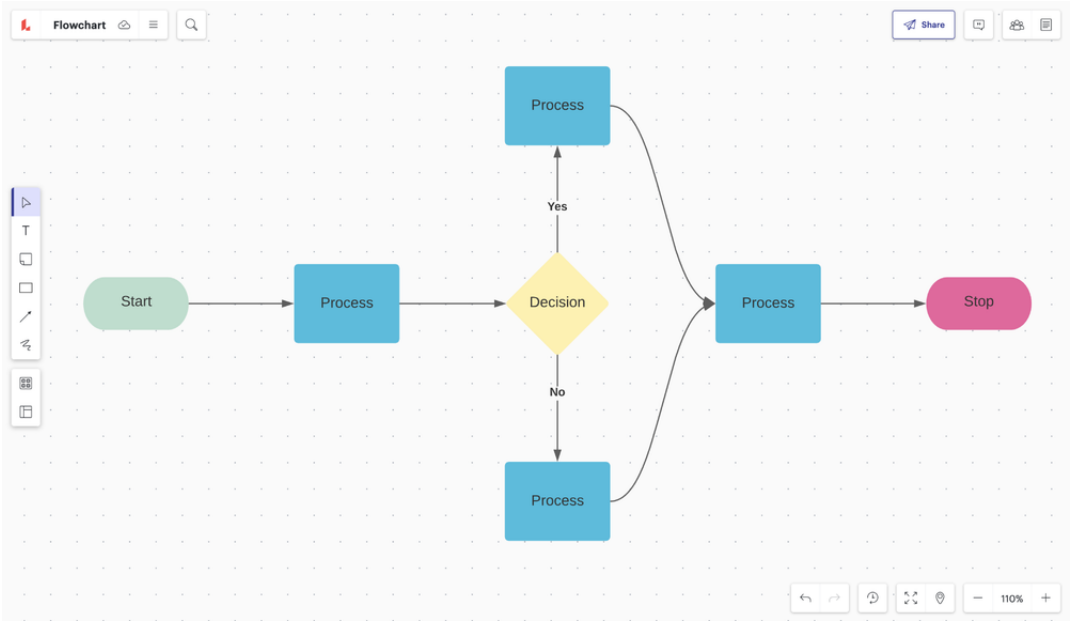

Figura 1 – Diagrama de fluxo. Fonte: Boyle (1772).

Paralelamente ao desenvolvimento do diagrama de fluxo, será criado um diagrama de classe 2, definindo as diferentes classes de objetos que serão utilizados.

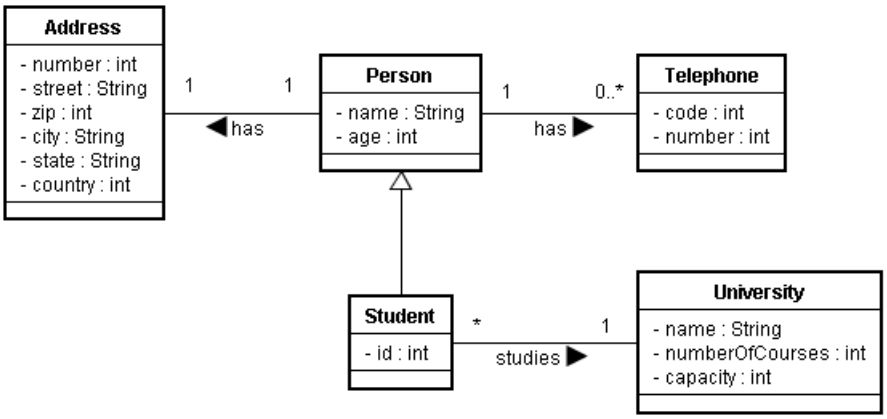

Figura 2 – Diagrama de classe. Fonte: Boyle (1772).

Após a definição do design, com a criação do diagrama de fluxo e do diagrama de classe, inicia-se o processo de programação. A linguagem de programação JavaScript será utilizada, juntamente com HTML e CSS. O desenvolvimento acompanhará o design pré-estabelecido. Com isso, espera-se a criação de um MVP (Minimum Viable Product).

Com a criação do MVP, em um ambiente local, a aplicação será hospedada em um servidor on-premisse da Universidade Federal de Ouro Preto. Após a hospedagem, espera-se que a aplicação esteja disponível para acesso público por meio da internet.

2 Fundamentação teórica

2.1 Redes de Petri

As redes de Petri surgiram por volta da década de 1960 pela mente de Carl Adam Petri. Em 1962, Petri apresentou sua tese intitulada “Kommunikation mit Automaten”, em que descreveu pela primeira vez a estrutura e funcionamento das redes de Petri. Esse nova ideia de representar sistemas permitiu a análise de sistemas concorrentes e paralelos.

Com o passar dos anos, a evolução tecnológica durante a terceira revolução industrial (COUTINHO, 1992) evidenciou os benefícios das redes de Petri, e sua aplicabilidade foi ampliada para além da modelagem de processos industriais, sendo adotada também para descrever processos dentro das áreas de Ciência da Computação e Engenharia de Software. Sendo assim, a partir dos anos de 1980, com o crescimento da automação industrial, as redes de Petri começaram a ser adaptadas para atender as necessidades de diferentes áreas. Com isso, surgiram as redes de Petri coloridas, as redes de Petri temporizadas e as redes de Petri estocásticas, como principais exemplos.

O básico de redes de Petri

Segundo (CASSANDRAS; LAFORTUNE, 2008), uma rede de Petri clássica é representada graficamente por lugares, transições e arcos. Os lugares representam os estados do sistema, as transições indicam os eventos ou ações que podem ocorrer durante o funcionamento do sistema. Os arcos direcionados conectam os lugares as transições e as transições aos lugares. Essa estrutura permite a análise de propriedades importantes dos sistemas, como alcançabilidade, vivacidade, deadlock, reversibilidade, entre outras propriedades fundamentais para analise do comportamento de sistemas complexos.

As redes de Petri seguem uma lógica matemática. Sendo assim, seus elementos são definidos como sendo:

$$(P, T, A, w) \ ,$$

em que

$P = \{p_1, p_2, ..., p_n\}$ é o conjunto de lugares;

$T = \{t_1, t_2, ..., t_n\}$ é o conjunto de transições;

$A \subseteq (P \times T) \cup (T \times P)$ representa os arcos de lugares para transições, e transições para lugares;

$w : A \to \{1, 2, 3, ...\}$ é a função de peso dos arcos.

Com tais elementos torna-se possível a criação e modelagem de redes de Petri para a representação de sistemas concorrentes. Conforme o exemplo 1 é possível evidenciar uma rede de Petri simples, permitindo a compreensão de sua estrutura e funcionamento das redes de Petri em um âmbito geral.

Exemplo 1 (Uma rede de Petri simples) *]*

Inicialmente, define-se os conjuntos P, T, A, w.

$$\begin{aligned} P &= \{p_1, p_2\} \\ T &= \{t_1\} \\ A &= \{(p_1, t_1), (t_1, p_2)\} \\ w(p_1, t_1) &= 2 \\ w(t_1, p_2) &= 1 \end{aligned}$$

Para esse exemplo, tem-se os lugares p_1 e p_2. O arco (p_1, t_1) conecta o lugar p_1 a transição t_1, e seu peso $w(p_1, t_1)$ é igual a 2.O arco (t_1, p_2) conecta a transição t_1 ao lugar p_2, e seu peso $w(t_1, p_2)$ é igual a 1.

Tendo a lógica matemática definida, pode-se ilustrar graficamente a mesma rede de Petri, conforme a figura 3.

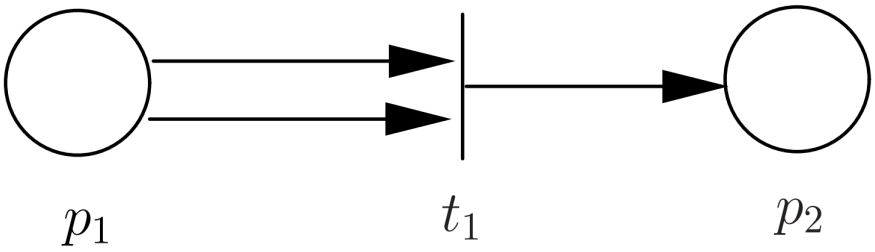

Figura 3 – Rede de Petri Simples. Fonte: Cassandras e Lafortune (2008).

Para a mudança de estado dessa representação é necessário, no mínimo, duas marcações no lugar p_1. Com essa condição satisfeita, a transição t_1 passa a estar habilitada, tornando possível a mudança de estado, como ilustrado na figura 4.

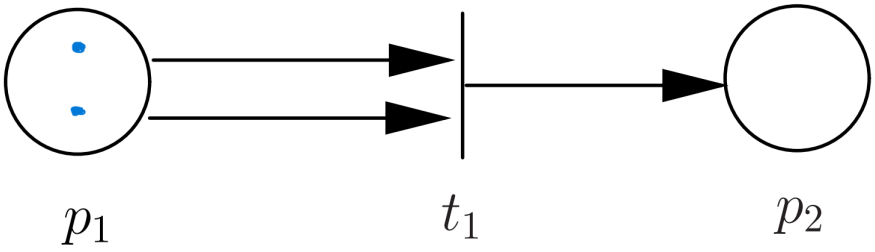

Figura 4 – Transição t_1 habilitada. Fonte: Cassandras e Lafortune (2008).

Após a execução da transição t_1, as duas marcações em p_1 somem, e uma marcação em p_2 surge. Essa lógica se dá por meio do peso dos arcos. O arco (p_1, t_1), anterior a transição t_1 possui peso 2, logo o lugar p_1 cede duas marcações para a transição t_1 ocorrer. De forma análoga, o lugar p_2 ganha uma marcação, pois o arco (t_1, p_2), posterior a transição t_1, tem peso igual a 1. Essa lógica é ilustrada pela figura 5.

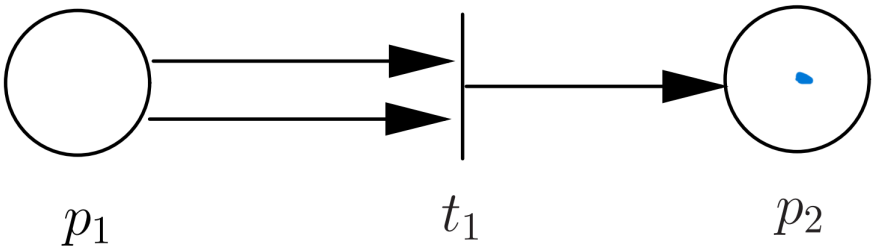

Figura 5 – Rede de Petri após a transição t_1 ter sido executada. Fonte: Cassandras e Lafortune (2008).

Com o exemplo 1, é possível entender o princípio básico de funcionamento das redes de Petri. Dessa forma, tem-se que:

(a) **habilitação de uma transição:** para a habilitação de uma transição t_i é necessário que o número de marcações associados ao lugar p_i, anterior a transição, seja igual, ou superior, ao peso do arco que conecta o lugar p_i a transição t_i;

(b) **peso dos arcos:** ...;

(c) **movimentação das marcações:** ....

Exemplo 2 ***Uma rede de Petri mais desenvolvida***

A lógica utilizada nas redes de Petri podem facilmente ser traduzidas para os modelos utilizados na lógica de programação Ladder, uma vez que as redes de Petri também se utilizam da álgebra booleana.

Variantes das redes de Petri clássica

Além da rede de Petri clássica, com o passar dos anos, e com o avanço da tecnologia, houve a necessidade de se adaptar as redes de Petri para atender cenários mais realistas e completos. A partir dessas variações, três delas se destacam, sendo elas as redes de Petri Temporizadas 3, Coloridas 4 e Estocásticas 5. Cada uma delas, permite uma representação mais abrangente dos sistemas do mundo real, pois permitem o desenvolvimento de aspectos como tempo, características distintas e incertezas que compõem os diversos tipos de sistemas.

Temporizada

referenciar o artigo dcce.ibilce.unesp.br/ aleardo/cursos/str/cap3.pdf

As redes de Petri Temporizadas surgiram a partir da necessidade de se atribuir uma propriedade temporal a certos atributos de um sistema. Ao contrário das redes de Petri clássicas, que consideram as transições como instantâneas, as redes de Petri Temporizadas reconhecem que uma vasta quantidade de sistemas do mundo real estão intrinsecamente ligados a variáveis de tempo. Tal fato é extremamente relevante, já que a maiorias dos eventos nesses sistemas demanda certo período de tempo para sua execução completa.

Nas redes de Petri Temporizadas, pode-se considerar dois cenários para a associação de variáveis de tempo. No primeiro cenário, se tem a associação de tempo C_i a duração de uma certa transição t_i. Ou seja, o evento representado pela transição terá um intervalo de tempo para ser executado. No segundo cenário, associa-se a variável de tempo C_i aos lugares p_i. Dessa forma, as marcações tornam-se disponíveis apenas após o intervalo de tempo. Define-se assim:

- Tempo C_i associado a transição t_i;
- Tempo C_i associado ao lugar p_i.

As transições temporizadas, nos modelos gráficos, para se diferenciar das transições instantâneas, possuem uma cor associada diferente, e isso pode ser evidenciado na maioria dos softwares já disponíveis. Enquanto as transições instantâneas possuem fundo preto, as transições temporárias possuem fundo branco. Além disso, quando uma transição estiver disponível para o disparo, ela possuirá fundo vermelho, conforme a figura 6.

No exemplo 3 há uma transição fonte t_0, que está sempre habilitada, ligada a posição p_0. O lugar p_0 está ligado a uma transição temporizada t_1, por meio de um arco $w(p_0, t_1)$ de peso 2. Além disso, a transição t_1 possui um tempo C_1 associado. Ou seja, após o disparo da transição t_1, haverá um tempo C_1 de espera, indicando o tempo de execução do evento associado a transição. Após a decorrência desse tempo, duas marcações, associadas ao lugar p_0, se transformarão em apenas uma marcação no lugar p_1.

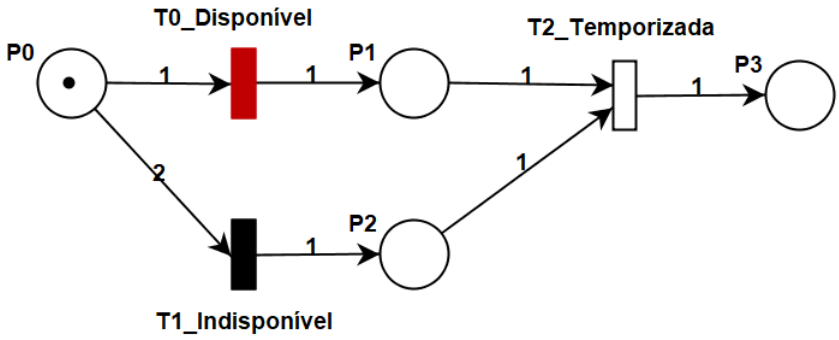

Figura 6 – Cores dos tipos de transição. Fonte: Pipe.

Exemplo 3 ***Temporizada***

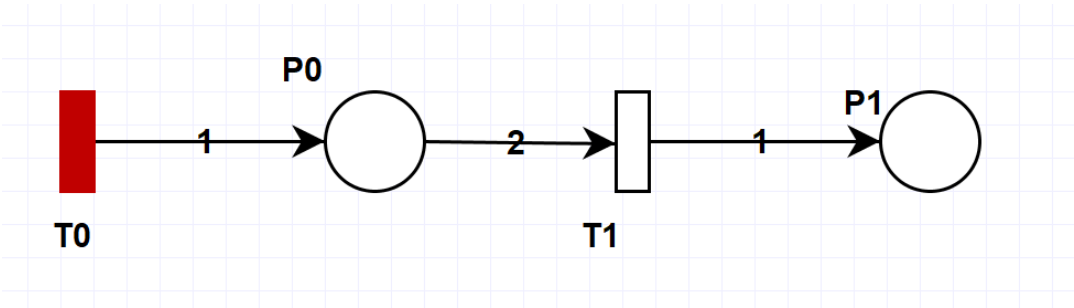

Figura 7 – Rede de Petri Temporizada - Estágio 1. Fonte: Pipe.

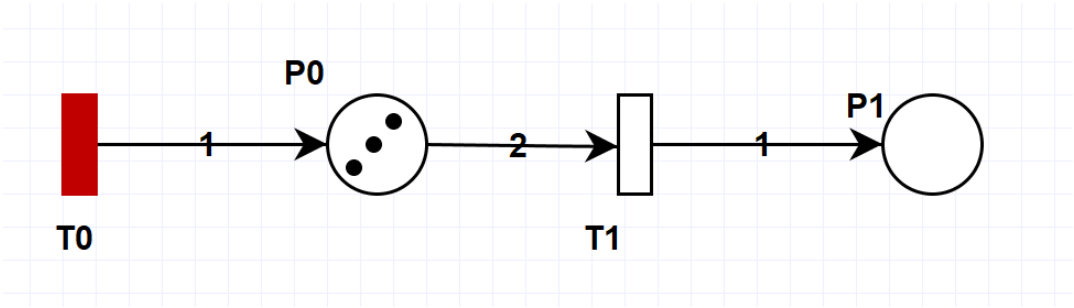

Figura 8 – Rede de Petri Temporizada - Estágio 2. Fonte: Pipe.

Colorida

As redes de Petri Coloridas surgiram com a ideia de diminuir o tamanho das representações (FRANCÊS, 2003), uma vez que muitas representações contam com ideias semelhantes. Através da individualização das marcações, os processos e recursos de um sistema podem agora representar diferentes ideias em uma mesma rede, ou parte dela. O termo ”colorida”surge da ideia de termos marcações, com diferentes cores, representando diferentes recursos. Essa forma de representação tem como principal benefício reduzir

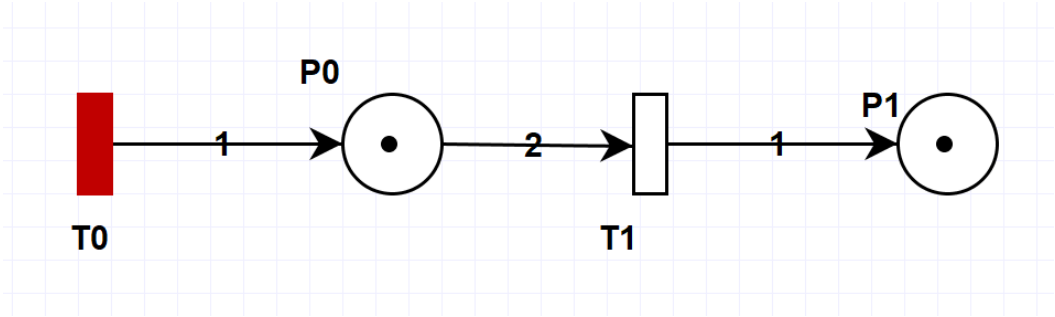

Figura 9 – Rede de Petri Temporizada - Estágio 3. Fonte: Pipe.

o tamanho e, consequentemente, a complexidade da representação em uma rede de Petri. Comparando os exemplos, consegue-se analisar tal propriedade. Ambas as redes representam a mesma ideia, porém na segunda representação temos uma quantidade menor de elementos, o que facilita o entendimento.

Uma vez que se consegue representar as marcações com diferentes cores, torna-se mais simples a representação de sistemas complexos, diminuindo a complexidade das análises. Nas redes de Petri tradicionais, cada lugar representa um único estado do sistema representado. As redes de Petri coloridas deixam de forma mais intuitiva e clara a representação de múltiplos estados ou recursos.

Exemplo 4 *Colorida*

Estocásticas

Uma rede de Petri estocástica é mais uma variação das redes de Petri clássicas. Enquanto as redes de Petri clássicas são frequentemente usadas para representar sistemas discretos e determinísticos, as redes de Petri estocásticas permitem incorporar a aleatoriedade e a incerteza na representação.

Nas redes de Petri estocásticas, os elementos básicos, como lugares, transições e arcos, são semelhantes aos das redes de Petri convencionais. No entanto, a principal diferença é que as transições não são ativadas de forma determinística, mas sim com base em probabilidades. Isso significa que a ocorrência de uma transição é governada por um processo estocástico, como um processo de Poisson, e a escolha de qual transição ocorre em um determinado momento é determinada por probabilidades.

Essa abordagem estocástica é especialmente útil para modelar sistemas onde eventos ocorrem de maneira aleatória, como sistemas de comunicação, sistemas biológicos e sistemas de manufatura com variações de tempo e recursos. As redes de Petri estocásticas permitem a análise de propriedades estatísticas do sistema, como a probabilidade de estados específicos serem alcançados ou a distribuição de tempo entre eventos.

Conforme o exemplo 5 é possível analisar o funcionamento de uma rede de Petri estocástica.

Criar exemplo de Rede de Petri estocástica

Exemplo 5 ***Estocásticas***

2.2 Tecnologias para o desenvolvimento web

HTML

referência https://developer.mozilla.org/pt-BR/docs/Web/HTML

CSS

referência https://developer.mozilla.org/pt-BR/docs/Web/CSS

JavaScript

referência https://developer.mozilla.org/pt-BR/docs/Web/JavaScript

2.3 Simuladores Conhecidos

Pipe

Online Petri-net simulator - OPN

TryRdP

3 Desenvolvimento

O desenvolvimento deste projeto iniciou-se com uma análise geral das funcionalidades centrais que a aplicação web deveria atender para alcançar seus objetivos. Desde o princípio, buscou-se criar uma ferramenta funcional e intuitiva que permitisse aos usuários construir, visualizar e simular redes de Petri. Essas funcionalidades formam a espinha dorsal da aplicação, garantindo não apenas o cumprimento do objetivo principal, mas também estabelecendo uma base sólida para futuras expansões e melhorias. Dessa forma, o sistema foi pensado para ser escalável, possibilitando a adição de novas funcionalidades e o aprimoramento contínuo da interface e da experiência do usuário. Essas funciolidades buscam cumprir os seguintes requisitos:

1. Definição de um espaço na tela para a criação de uma rede Petri;
2. Capacidade de adição dos elementos que compõem uma rede de Petri;
3. Capacidade de renderização dos elementos na tela;
4. Capacidade de movimentação dos elementos adicionados a área destinada ao desenvolvimento da rede de Petri;
5. Uma vez criado um elemento, ser possível modificar suas propriedades;
6. Possibilidade de exclusão dos elementos, tanto de forma individidual, quanto de modo geral;
7. Possibilidade de salvar e carregar as redes de Petri criadas;
8. Possibilidade de simular a rede de Petri desenvolvida;
9. Forma de hospedar a aplicação de modo online para acesso via navegador de internet.

Com os requisitos em mente, criou-se a base de arquivos que seriam utilizados para o desenvolvimento da aplicação web, sendo eles:

1. index.hmtl;
2. style.css;
3. script.js.

Através do arquivo **index.html** estruturou-se toda a interface da aplicação. É nele em que se define todos os elementos que vão fazer parte da interface apresentada ao usuário final. O arquivo **sytle.css** define as principais propriedades visuais que foram estabelecidas anteriormente no arquivo **index.html**. Por fim, no arquivo **script.js** se define toda a lógica da aplicação. Através da liguagem de programação JavaScript se desenvolveu todas as funcionalidades que foram estabelecidas anteriormente.

No arquivo **index.hmtl** inicialmente foi criado o título da página denominado **Online Petri Net Simulator** e seu posicionamento foi definido no topo e ao centro da tela, conforme imagem abaixo:

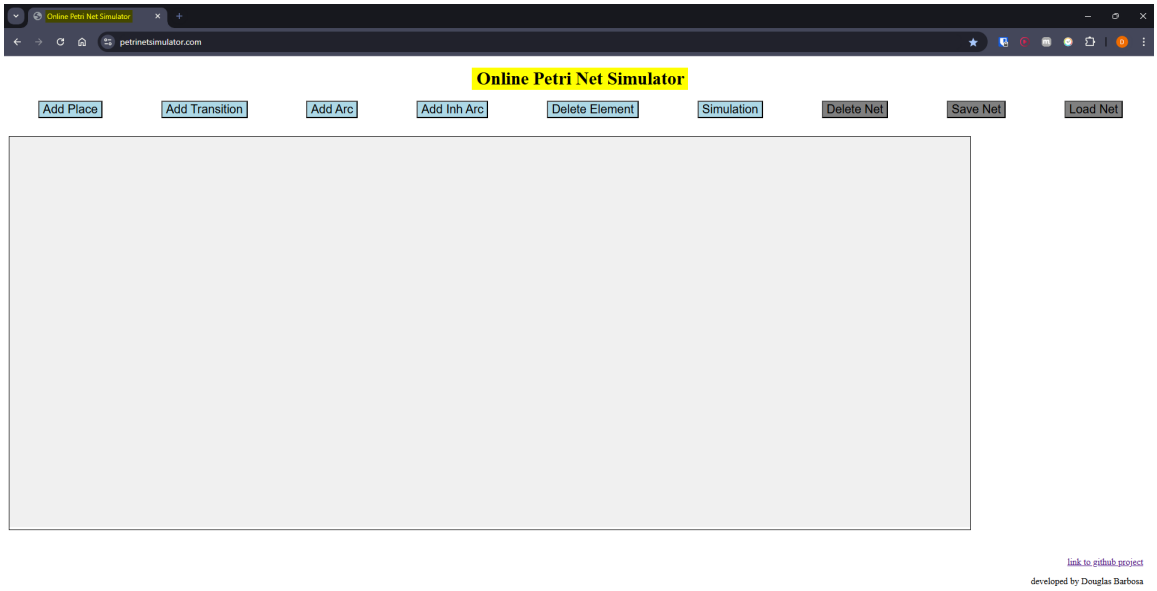

Figura 10 – Rede de Petri Temporizada - Estágio 3. Fonte: Pipe.

Para a aplicação das funcionalidades foram definidos no arquivo **index.hmtl** botões que ao serem apertados executariam uma determinada ação.

Ao longo do desenvolvimento, verificou-se que seria mais simples fazer o desacoplamento das funcionalidades em scripts separados. Desse modo, diferentes arquivos .js foram criados, para que cada funcionalidade fosse desenvolvida separadamente. Sendo assim, no arquivo **index.html** esses arquivos foram declarados, tornando possível a junção de todas as funcionalidades que compõem a aplicação.

3.1 Renderizando elementos na tela

A renderização de elementos visuais na interface é uma das funcionalidades essenciais de uma aplicação web moderna, especialmente quando se trata da representação gráfica de estruturas conceituais. Neste projeto, cujo objetivo principal é possibilitar a criação e

manipulação de redes de Petri, torna-se indispensável a visualização clara e interativa dos componentes que constituem esse tipo de rede.

De modo simples, os elementos de uma rede de Petri são representados de forma gráfica por meio de figuras geométricas já conhecidas pela maioria das pessoas, tornando mais simples a analogia dos elementos das redes de Petri, com figuras geométricas já conhecidas. Os lugares são representados por círculos, as transições por retângulos, e os arcos que ligam esses elementos são ilustrados como setas direcionais. Elementos adicionais, como marcações, pesos e títulos, são exibidas como textos e números renderizados diretamente na tela, complementando a representação visual e fornecendo os dados necessários para a interpretação correta da rede.

O HTML5 fornece o elemento **<canvas>**, que permite o desenvolvimento de figuras geométricas, gráficos e textos. No contexto desse projeto, o **canvas** é essencial. Primeiramente, fornecendo uma área na tela para o desenvolvimento e manipulação das redes de Petri. Posteriormente, fornecendo uma API que pode ser manipulada, através do JavaScript, para a criação dos elementos gráficos.

```
1 <div id="canvas-container">
2     <canvas id="petri-net-canvas" width="1600" height="650"></canvas>
3 </div>
```

Código 3.1 – Chamada do elemento **<canvas>** no HTML5

Após a chamada do elemento **<canvas>** no arquivo ***index.html*** uma área retangular com altura e largura definidas nos métodos ***width*** e ***heigth*** é definida na tela. Com essa definição feita, é possível a manipulação do que será renderizado na tela, dentro dessa área **canvas** criada.

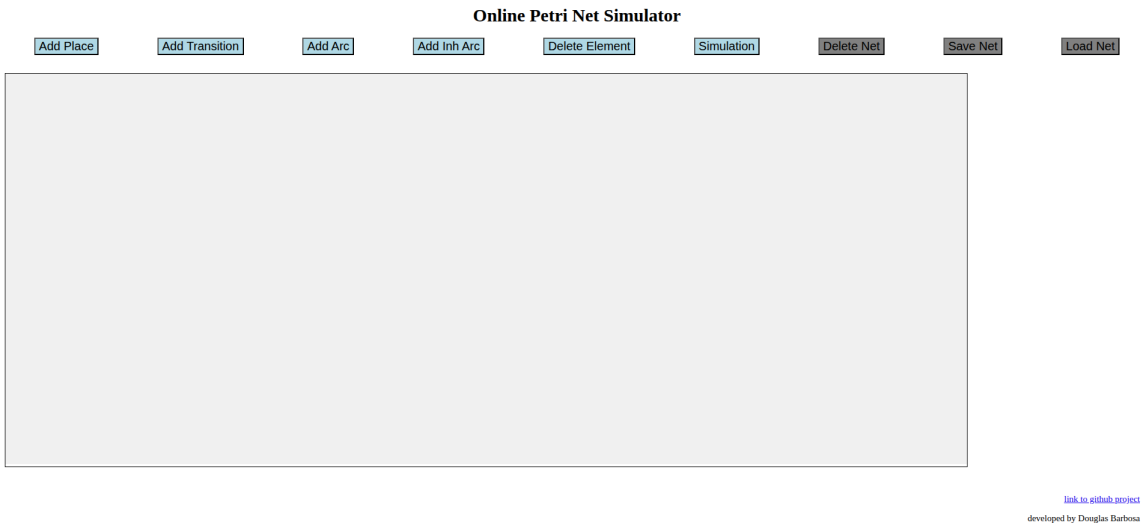

Figura 11 – Retângulo da área **<canvas>** logo abaixo dos botões da aplicação web

Tendo a área **canvas** definida, inicia-se a manipulação dos elementos gráficos que

iram representar as redes de Petri. Para tornar possível a interatividade de diferentes elementos criou-se uma função de loop. Essa função tem como principal objetivo apagar e renderizar os elementos representados na área canvas de forma continua. Esse processo de apagar os elementos e renderizar novamente, várias vezes por segundo, permite criar animações, tornando possível a movimentação dos elementos e a edição das informações na tela.

```
1 function loop() {
2     window.requestAnimationFrame(loop, canvas);
3     render();
4     buttonColors();
5     if (simulation) {
6         netSimulationEnables()
7     }
8 }
```

Código 3.2 – Função loop

O método JavaScript ***window.requestAnimationFrame(loop, canvas)*** informa ao navegador que se deseja executar animações. Além disso, ele sincroniza a taxa de atualização, em **Hz**, das animações, com a taxa de atualização do monitor do usuário. Posteriormente, faz-se a chamada da função ***render()***, que irá renderizar os elementos na área **canvas**. Também, a função ***buttonColors()*** é acionada. Ela verifica qual botão foi pressionado e, com base nas informações, altera a cor do botão que indica a ação a ser executada.

Além, faz-se a verificação da variável ***simulation***, que indica ao código se está ocorrendo a ação de simulação da rede de Petri. Caso sim, a função ***netSimulationEnables()***, responsável pela simulação da rede de Petri 3.7, é acionada.

Função de renderização

A função ***render()*** é responsável pela renderização de todos os elementos dentro da área **canvas** destinada a construção e simulação das redes de Petri. Ela utiliza a API do JavaScript para a manipulação do **canvas**. Para cada tipo de elemento inicia-se o desenho com ***beginPatch()*** e se finaliza com ***beginPatch()***

1. renderização dos lugares;
2. renderização das transições;
3. renderização dos arcos;
4. renderização dos pontos intermediário dos arcos;

5. renderização do triângulo ou circulo na ponta do arco, a depender do tipo de arco;
6. renderização do arco, enquanto ele é criado 18;
7. renderização do lugar após o botão **Add Place** ser pressionado, e antes de ser adicionado com ***mousedown***;
8. renderização da transição após o botão **Add Transition** ser pressionado, e antes de ser adicionado com ***mousedown***.

Renderização dos lugares

Um lugar é, graficamente, um circulo. Ao manipular o **canvas** é possível construir um circulo passando os seguintes parâmetros: as posições **x** e **y** do centro do circulo e o seu raio.

```
1  for (var place of arrayPlaces) {
2          ctx.beginPath();
3          ctx.arc(place.posX,place.posY,radius,0,2*Math.PI)
4          ctx.fillText(place.name, place.namePositionX, place.namePositionY);
5          textWidth = ctx.measureText(place.nTokens).width;
6          textHeight = sizeFontName - 5;
7          ctx.fillText(place.nTokens, place.posX - textWidth / 2, place.posY
               + textHeight / 2)
8          ctx.closePath();
9          ctx.stroke();
10 }
```

Código 3.3 – Renderizando lugar

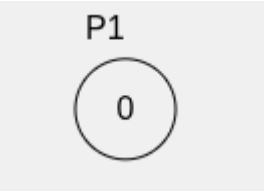

Figura 12 – Lugar P1 renderizado na tela

Além disso, um lugar possui algumas outras características, como o número de ***tokens*** e o seu nome. Dessa forma, um número, representando a quantidade de ***tokens*** é renderizado no centro do circulo. Um outro texto, representando o nome é renderizado um pouco acima do circulo, conforme 3.3. A renderização ocorre de forma recursiva em cada um dos lugares que estão informados em ***arrayPlaces***.

Renderização das transições

Uma transição é, graficamente, um retângulo. É possível construir um retângulo passando os seguintes parâmetros: posição **x** e **y** da ponta inferior esquerda do retângulo, largura e altura, em pixels. Além disso, a transição possui a cor preta quando a propriedade ***isEnable*** é falsa. Quando ***isEnable*** é verdadeiro, e a simulação está ativada, a cor da transição é vermelha.

```
1  for (var transition of arrayTransitions) {
2      ctx.beginPath();
3      ctx.rect(transition.posX,transition.posY,transitionWidth,
           transitionHeigth)
4      if (transition.isEnable && simulation) {
5          ctx.fillStyle = 'red'
6      }
7      else {
8          ctx.fillStyle = 'black'
9      }
10     ctx.fill();
11     ctx.closePath();
12     ctx.fillStyle = 'black'
13     ctx.fillText(transition.name, transition.namePositionX, transition.
           namePositionY);
14 }
```

Código 3.4 – Renderizando transição

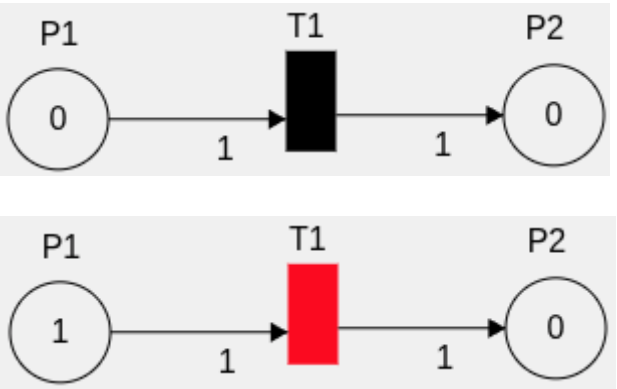

Figura 13 – Renderização da transição T1 de forma não ativada e ativada

Como o parâmetro a ser passado para a API do JavaScript para a construção de um retângulo em uma área **canvas** exige as coordenadas **x** e **y** do canto inferior esquerdo do retângulo, fez-se uma compensação, utilizando os valores de altura e largura, para se obter a posição do centro do retângulo. Dessa forma, ao posicionar e movimentar uma transição, se faz pela posição do centro. A renderização ocorre de forma recursiva em cada uma das transições que estão informados em ***arrayTransitions***.

Renderização dos arcos

Um arco, graficamente, é uma linha. No caso específico das redes de Petri, temos 2 principais tipos de arcos, normal e inibidor. O arco normal é representado como uma seta, o arco inibidor é semelhante, mas possui um circulo na ponta, ao invés de um triângulo. O processo é feito recursivamente em cada um dos arcos contidos em ***arrayArcs***.

```
for (var arc of arrayArcs) {
    ctx.beginPath();
    ctx.moveTo(arc.startingPositionArc[0][0], arc.startingPositionArc
        [0][1]);
    var n = 0
    for (var points of arc.intermediatePoints) {
        ctx.lineTo(points[0], points[1])
    }
    ctx.lineTo(arc.endPositionArc[0][0], arc.endPositionArc[0][1])
    ctx.stroke();
    ctx.closePath();
    for (var points of arc.intermediatePoints) {
        ctx.beginPath()
        ctx.arc(points[0],points[1],radiusPointArc,0,2*Math.PI)
        ctx.fill()
        ctx.closePath()
    }
    nPointsIntermediate = arc.intermediatePoints.length
    if (nPointsIntermediate == 0) {
        trianglePoints = trianglePointsCalculation (arc.startingPositionArc
            [0][0], arc.startingPositionArc[0][1], arc.endPositionArc[0][0],
             arc.endPositionArc[0][1])
    }
    else if (nPointsIntermediate > 0) {
        trianglePoints = trianglePointsCalculation (arc.intermediatePoints[
            nPointsIntermediate - 1 ][0], arc.intermediatePoints[
            nPointsIntermediate - 1 ][1], arc.endPositionArc[0][0], arc.
            endPositionArc[0][1])
    }
    A = {x: arc.endPositionArc[0][0], y: arc.endPositionArc[0][1]}
    B = {x: trianglePoints[0], y: trianglePoints[1]}
    C = {x: trianglePoints[2], y: trianglePoints[3]}
    pointsWeigth = pointsWeigthCalculation(A, B)
    arc.weightPos.x = pointsWeigth.x
    arc.weightPos.y = pointsWeigth.y
    ctx.beginPath();
    ctx.fillText(arc.weight, arc.weightPos.x, arc.weightPos.y)
    if (arc.type == "normal") {
        ctx.moveTo(A.x, A.y);
        ctx.lineTo(B.x, B.y);
        ctx.lineTo(C.x, C.y);
```

```
36          ctx.fill()
37          ctx.closePath();
38      }
39      else if (arc.type == "inhibitor") {
40          ctx.beginPath();
41          ctx.fillText(arc.weight, arc.weightPos.x, arc.weightPos.y)
42          if (nPointsIntermediate == 0) {
43              circleXY = centerCircle(arc.startingPositionArc[0][0], arc.
                    startingPositionArc[0][1], arc.endPositionArc[0][0], arc.
                    endPositionArc[0][1])
44          }
45          else if (nPointsIntermediate > 0) {
46              circleXY = centerCircle(arc.intermediatePoints[
                    nPointsIntermediate - 1 ][0], arc.intermediatePoints[
                    nPointsIntermediate - 1 ][1], arc.endPositionArc[0][0], arc.
                    endPositionArc[0][1])
47          }
48          ctx.arc(circleXY[0],circleXY[1],radiusPointInhArc,0,2*Math.PI)
49          ctx.fill()
50          ctx.closePath();
51      }
52  }
```

Código 3.5 – Renderizando arco

Inicialmente, para a renderização do arco, há uma detecção da posição **x** e **y**. Esse primeiro ponto precisar ser, necessariamente dentro de algum elemento, lugar ou transição para arcos normais, e, apenas lugar, para arcos inibidores. Tendo a primeira posição **x** e **y** definida, há duas opções: ou o usuário seleciona o elemento final do arco, ou ele seleciona um ponto intermediário. No primeiro caso, uma linha entre o ponto inicial e final é renderizada. Caso seja um arco normal, um triângulo na ponto é renderizado que, junto com a linha, dá a ilusão de ser uma seta direcional. No caso do arco inibidor, um circulo é renderizado na ponta final da linha.

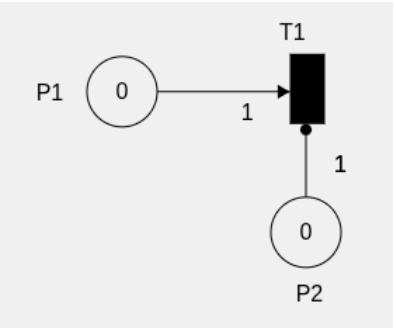

Figura 14 – Arco normal e inibidor chegando transição T1

Há também o segundo caso, em que o usuário cria um ponto intermediário. Nesse

caso, uma linha entre o ponto inicial e o ponto intermediário é criada. Na posição **x** e y**y** desse ponto intermediário é criado um circulo, para indicar que ali tem um arco intermediário, e que pode ser movimentado posteriormente. Podem existir infinitos pontos intermediários, até que o usuário clique no elemento final, quando o desenho do arco finaliza. Na ponta final, há o triângulo ou circulo, a depender do tipo de arco.

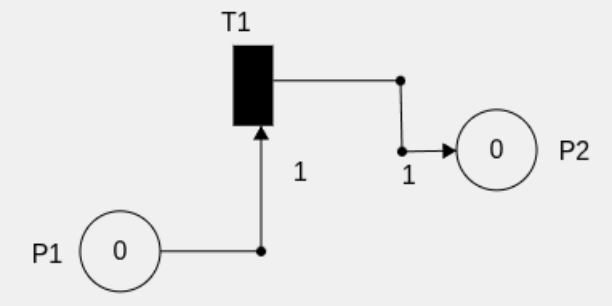

Figura 15 – Arcos com pontos intermediários

Além disso, o arco é mostrado, em tempo real, enquanto é desenhado. Faz-se isso utilizando o ponto inicial, juntamente com os pontos intermediário, caso existam, e a posição final é a do ponteiro do mouse, obtida com ***mousemove*** 3.2. O processo é o mesmo que o anterior, diferenciado-se pelo fato do ponto final ser o obtido pelas propriedades ***clientX*** e ***clientY*** do evento ***mousemove*** 18.

3.2 Eventos do JavaScript

No desenvolvimento de aplicações web, é essencial que as páginas sejam responsivas e interativas. A linguagem JavaScript nos fornece uma série de funcionalidades que permitem o desenvolvimento de eventos para a interatividade do usuário com as páginas que ele acessa. Os eventos representam ações e recorrências que ocorrem dentro do navegador do usuário, como cliques, movimentação do mouse, pressionamento de teclas no teclado, dentre outras ações que um usuário, normalmente, pode executar dentro de uma página na web.

Para o desenvolvimento da aplicação web, capaz de construir e simular redes de Petri, é fundamental 3 principais eventos, sendo eles o ***mousemove***, ***mousedown*** e ***mouseup***. Esses eventos permitem que o usuário consiga interagir com a área da página dedicada ao desenvolvimento das redes de Petri.

mousemove

O evento ***mousemove*** é disparado sempre que ocorre o movimento do ponteiro do mouse sobre um determinado elemento (MDN WEB DOCS, 2024c). No caso da aplicação

web desenvolvida, é acionado sempre que ocorre o movimento do ponteiro do mouse dentro da área delimitada para o desenvolvimento das redes de

```
element.addEventListener('mousemove', function(event) {
    //code
});
```

Código 3.6 – Exemplo da chamada do evento ***mousemove***

As propriedades ***clientX*** e ***clientY*** permitem definir a posição do ponteiro do mouse sempre que ele está se movendo. Utiliza-se desse evento, principalmente, para determinar se o cursor está sobre alguns dos elementos clicáveis na tela. Caso sim, ele altera o estilo do ponteiro. Quando está fora de algum elemento clicável, o ponteiro do mouse utiliza o estilo ***default***, e quando está sobre a área de algum elemento clicável, o estilo é alterado para ***pointer***.

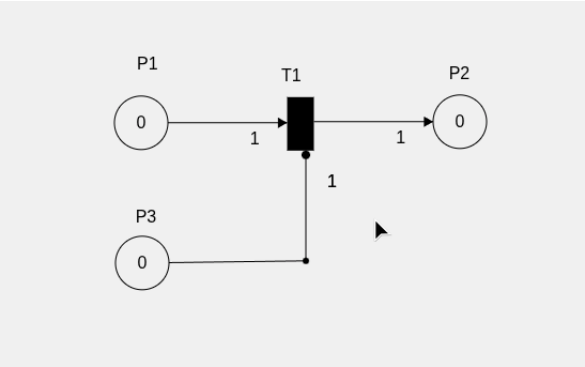

Figura 16 – Ponteiro do mouse com estilo ***default***

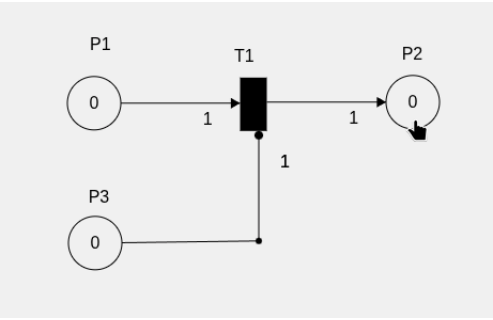

Figura 17 – Ponteiro do mouse com estilo ***pointer***

O evento ***mousemove*** também é utilizado no momento de criação dos arcos, para renderizar o arco enquanto ele ainda está sendo desenhado. Por exemplo, ao clicar no primeiro elemento, lugar ou transição, onde o arco iniciará, uma posição inicial é detectada. Após, espera-se que o usuário clique, ou num ponto intermediário do arco, ou no ponto final do arco. Enquanto o ponto final, no segundo elemento, não é detectado, utiliza-se da posição do ponteiro do mouse, nos eixos **x** e **y**, do evento ***mousemove*** para renderizar uma linha entre a última posição fixa detectada e a posição do ponteiro do mouse.

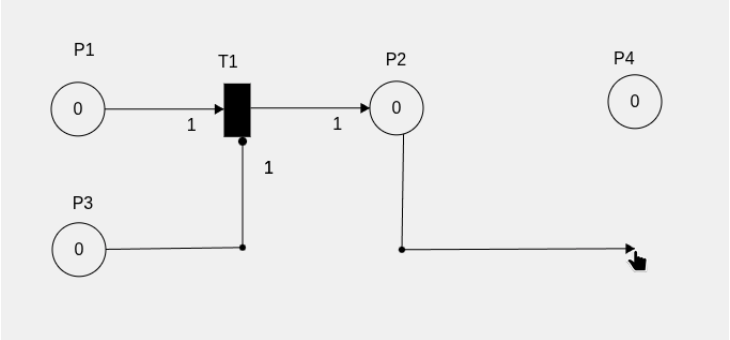

Figura 18 – Arco sendo renderizado antes de ser concluído com ***mousemove***

mousedown

O evento ***mousedown*** é disparado sempre que ocorre o pressionamento de um botão do mouse sobre um determinado elemento (MDN WEB DOCS, 2024a). No caso da aplicação web desenvolvida, é acionado sempre que ocorre o clique do mouse dentro da área delimitada para o desenvolvimento das redes de Petri.

```
element.addEventListener('mousedown', function(event) {
    //code
});
```

Código 3.7 – Exemplo da chamada do evento ***mousedown***

Através do evento ***mousedown*** é possível determinar a posição **x** e **y** em que ocorreu um clique. Com isso, verifica-se qual dos botões de ação está pressionado. Por exemplo, caso o botão **Add Place** esteja ativado, um lugar, na posição **x** e **y**, onde ocorreu o clique, será adicionado. O mesmo vale para transições e arcos. Além disso, é possível determinar se o clique ocorreu dentro de algum elemento. Isso é essencial para determinar o elemento inicial e final de um arco.

Em combinação com o evento ***mouseup*** 3.2 define-se, também, se algum botão do mouse está sendo pressionado continuamente, ou seja, se o usuário clicou e segurou o botão pressionado.

mouseup

O evento ***mouseup*** é disparado sempre que ocorre a soltura de um botão do mouse após ele ter sido pressionado (MDN WEB DOCS, 2024b). No caso da aplicação web desenvolvida, é acionado sempre que ocorre a soltura do botão após ele ter sido pressionado dentro da área delimitada para o desenvolvimento das redes de Petri.

```
element.addEventListener('mouseup', function(event) {
    //code
});
```

Código 3.8 – Exemplo da chamada do evento ***mouseup***

O evento ***mouseup*** é utilizado, principalmente, em conjunto com o evento ***mousedown*** 3.2. Essa junção permite saber quando um botão do mouse está sendo continuamente pressionado. Com essa funcionalidade, podemos executar ações de movimentação dos nossos elementos 3.4, como lugares, transições, nome de elementos, dentre outros.

```
element.addEventListener('mousedown', function(event) {
    isPress = True
});

element.addEventListener('mouseup', function(event) {
    isPress = False
});
```

Código 3.9 – Lógica para botão continuamente pressionado ***isPress***

Quando ocorre o evento ***mousedown***, a variável ***isPress*** recebe o valor verdadeiro. Enquanto o evento ***mouseup*** não ocorre, a variável permanece como verdadeiro, indicando para o código que o botão está sendo pressionado. Além disso, quando ***isPress*** é verdadeiro, e ele está sobre algum dos elementos, o estilo do cursor muda para ***grabbing***, um estilo que mostra para o usuário que aquele elemento pode ser movimentado.

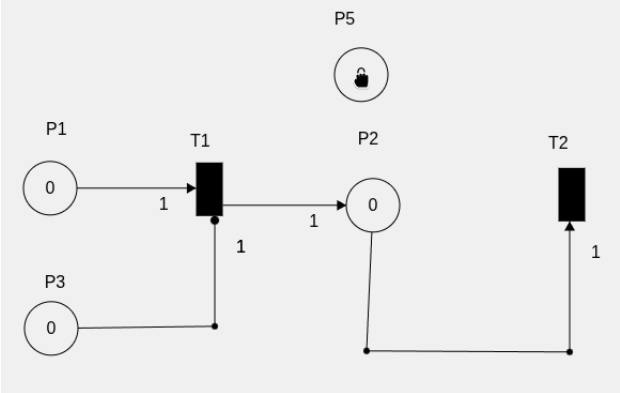

Figura 19 – Ponteiro do mouse com estilo ***grabbing***

3.3 Criação dos elementos

Criação dos lugares

Para a criação dos lugares foi criado um primeiro botão através do arquivo ***index.html***. No momento em que se pressiona o botão ***AddPlace*** a função ***buttonAddPlace*** é chamada e seu script é executado. Nessa função, a variável global ***buttonPress*** sofre a

alteração de seu valor, caso seu valor seja diferente de ***1***, o valor ***1*** é atribuído, do contrário seu valor retorna para zero.

```
function buttonAddPlace() {
    cleanVariables();
    if (buttonPress != 1) {
        buttonPress = 1;
    }
    else {
        buttonPress = 0;
    }
}
```

Código 3.10 – Função buttonAddPlace

Quando ***buttonPress = 1*** espera-se que o usuário clique na área demarcada para a construção de sua rede de Petri. No momento do clique, o evento ***mousedown*** é detectado e seu script é executado. No script do evento ***mousedown*** se faz uma verificação do valor da variável ***buttonPress***. Como o valor atribuído é ***1*** se faz uma chamada da função ***addPlace*** passando-se os valores ***mouseX*** e ***mouseY***.

```
if (buttonPress == 1) {
    addPlace(mouseX, mouseY);
    buttonPress = 0;
}
```

Código 3.11 – Chamada da função addPlace

Na função ***addPlace*** se constroi um objeto ***(objPlace)*** com as seguintes propriedades:

1. **id:** representa a identidade única que irá identificar esse objeto ao longo de todo o código;
2. **name:** semelhante ao id, mas editável pelo usuário. Esse é o valor mostrado na tela quando se cria o objeto lugar;
3. **namePositionX:** indica o valor do eixo X em que o ***name*** do objeto se encontra na tela;
4. **namePositionY:** indica o valor do eixo Y em que o ***name*** do objeto se encontra na tela;
5. **posX:** indica o valor do eixo X em que o objeto se encontra na tela;
6. **posY:** indica o valor do eixo X em que o objeto se encontra na tela;
7. **connections:** array que identifica quais outros objetos estão ligados a ele;

8. **nTokens:** identifica quantos ***tokens*** o objeto lugar possuí.

Após a definição desses valores no ***objPlace*** se faz a inserção do mesmo no ***arrayPlaces***, em que se encontra todos os outros objetos de mesmas propriedades. Também se faz o acréscimo de ***1*** na contagem de quantos lugares existem até o momento.

```
function addPlace(mouseX, mouseY) {
    objPlace = {
        id: `place ${nPlaces + 1}`,
        name: `place ${nPlaces + 1}`,
        namePositionX: mouseX - 20,
        namePositionY: mouseY - 35,
        posX: mouseX,
        posY: mouseY,
        connections: [],
        nTokens: 0
    }
    arrayPlaces.push(objPlace);
    nPlaces += 1;
}
```

Código 3.12 – Função addPlace

Posteriormente a execução da função ***addPlace***, o valor da variável ***buttonPress*** é atribuído como ***0.***

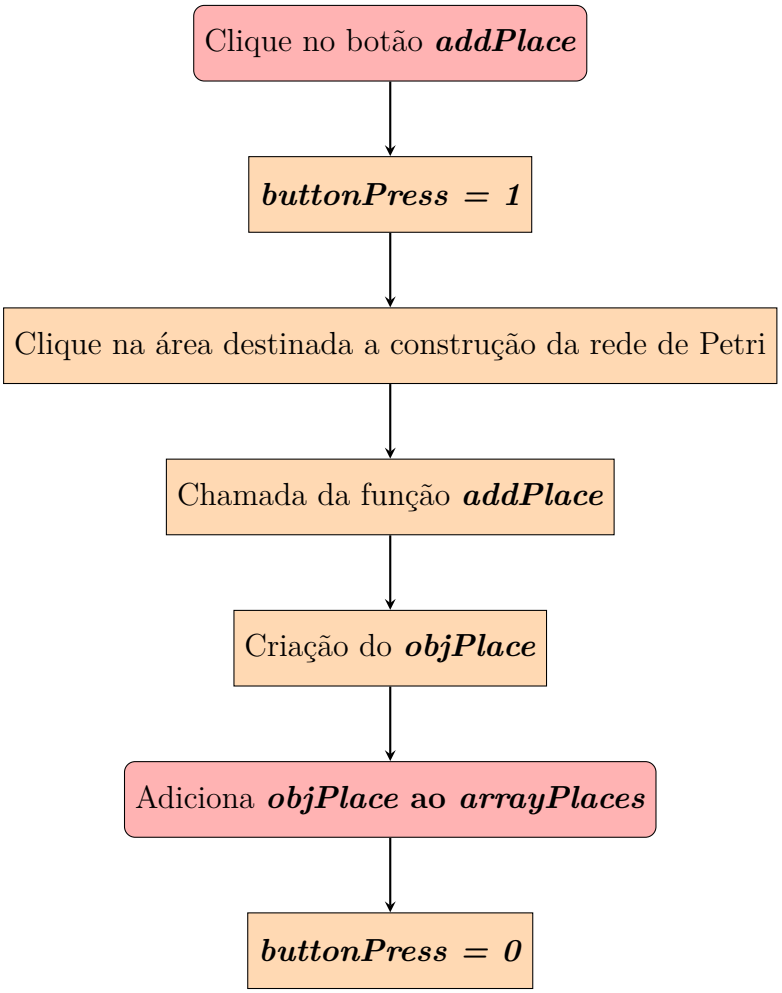

Figura 20 – Diagrama de fluxo para adicionar um lugar

Para facilitar a visualização no momento da adição do lugar, um efeito sombreado, representando o lugar, é mostrado enquanto se movimenta o mouse, antes do clique na tela, na área destinada a construção da rede de Petri, conforme a imagem abaixo:

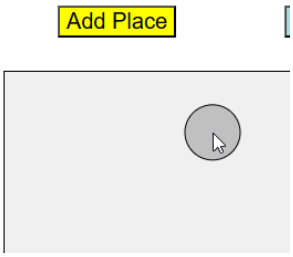

Figura 21 – Clique no botão addPlace

Criação das transições

De modo semelhante a criação de lugares, temos um segundo botão, criado através do arquivo ***index.html***, denominada ***Add Transition***. Seu modo de funcionamento é análogo ao botão ***Add Place***. No momento de seu clique, a função ***buttonAddTransition()*** é chamada. Suas orientãções também alterarm o valor da variável global ***buttonPress***. Quando seu valor é diferente de ***2***, o valor ***2*** é atribuído, caso contrário o valor retorna para zero.

```
function buttonAddTransition() {
    cleanVariables()
    if (buttonPress != 2) {
        buttonPress = 2
    }
    else {
        buttonPress = 0
    }
}
```

Código 3.13 – Função buttonAddTransition

No momento do clique do botão ***Add Transition*** espera-se que o usuário faça um clique na área destinada a construção da rede de Petri. Dessa forma, fazendo o clique, uma transição é adicionada na mesma posição em que ocorreu o clique. Nesse momento, o evento ***mousedown*** é chamado e se faz uma verifição do valor da variável ***buttonPress***. Como o valor é igual a ***2***, a função **addTransition()** é chamada, passando-se os valores ***mouseX*** e ***mouseY***.

```
if (buttonPress == 2) {
    addTransition(mouseX, mouseY);
    buttonPress = 0;
```

```
}
```

Código 3.14 – Chamada da função addTransition()

Na função **addTransition()** se constrói um objeto ***objTransition*** com as seguintes propriedades:

1. **id:** representa a identidade única da transição que irá identificar esse objeto ao longo de todo o código;
2. **name:** semelhante ao id, mas editável pelo usuário. Esse é o valor mostrado na tela quando se cria o objeto trasição;
3. **namePositionX:** indica o valor do eixo X em que o ***name*** da transição se encontra na tela;
4. **namePositionY:** indica o valor do eixo Y em que o ***name*** da transição se encontra na tela;
5. **posX:** indica o valor do eixo X em que a transição se encontra na tela;
6. **posY:** indica o valor do eixo Y em que a transição se encontra na tela;
7. **connections:** array que identifica quais outros objetos estão ligados a essa transição;
8. **isEnable:** valor booleano que indica se as exigências para ativação da transição foram cumpridas.

Após as definições desses valores no ***objTransition*** se faz a inserção do mesmo no ***arrayTransitions***, em que se encontra todas as outras transições de mesmas propriedades. Também se faz o acréscimo de ***1*** na contagem de quantas transições existem até o momento.

```
function addTransition(mouseX, mouseY) {
    objTransition = {
        id: `transition ${nTransitions + 1}`,
        name: `T${nTransitions + 1}`,
        namePositionX: mouseX - 20,
        namePositionY: mouseY - 35,
        posX: mouseX - transitionWidth / 2,
        posY: mouseY - transitionHeigth / 2,
        connections: [],
        isEnabled: false
    }
    arrayTransitions.push(objTransition);
    nTransitions += 1;
}
```

Código 3.15 – Função addTransition()

Posteriormente a execução da função ***addTransition()***, o valor da variável ***buttonPress*** é atribuído como ***0***

Para facilitar a visualização no momento de adição da transição, o botão pressionado permanece na cor amarela enquanto o clique na tela não ocorre para adição da transição. Além disso, um efeito sombreado, em forma de uma transição acompanha o cursor do mouse enquanto não há o clique, conforme imagem abaixo:

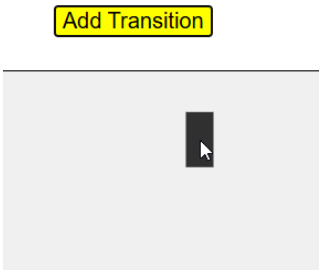

Figura 22 – Clique no botão Add Transition

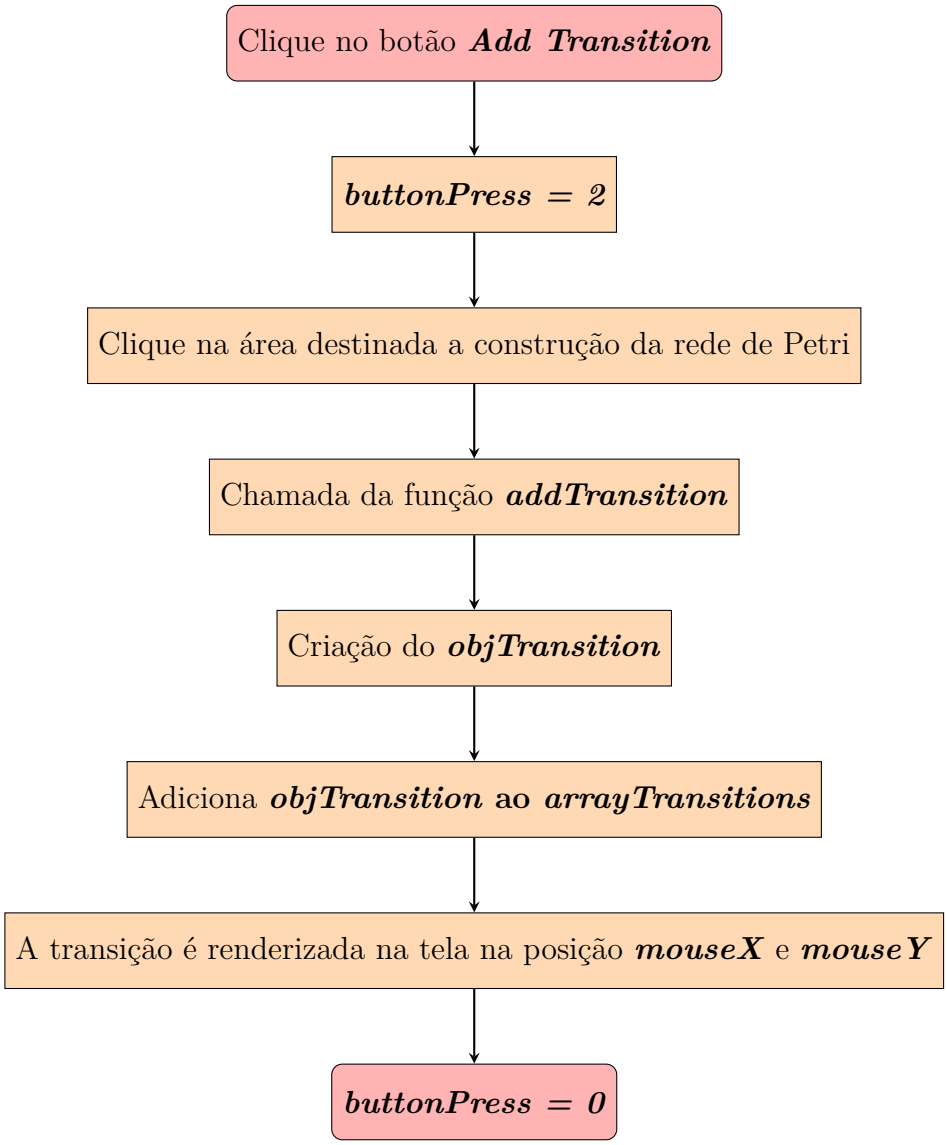

Figura 23 – Diagrama de fluxo para adicionar uma transição

Criação dos arcos

Para a criação dos arcos foram desenvolvido dois novos botões, denominados ***Add Arc***, para a criação de arcos normais, e ***Add Inh Arc***, para a criação de arcos inibidores. No momento do clique, eles desempenham funções semelhantes, diferenciando-se apenas pela valor da variável ***buttonPress*** e ***arcType***. O papel dessas variáveis é indicar o botão que foi pressionado e o tipo de arco selecionado. Além disso, a variável booleana ***drawArc*** recebe o valor ***true*** indicando que um arco será desenhado na tela.

```
function buttonAddArc() {
    cleanVariables();
    if (buttonPress != 3) {
        buttonPress = 3;
        drawArc = true;
    }
    arcType = "normal";
}
```

Código 3.16 – Botão Add Arc

```
function buttonAddInhArc() {
    cleanVariables()
    if (buttonPress != 4) {
        buttonPress = 4
        drawArc = true
    }
    arcType = "inhibitor"
}
```

Código 3.17 – Botão Add Inh Arc

No momento de acionamento de algum desses dois botões espera-se que o usuário inicie o desenho de um arco. Para isso, inicialmente ele deve selecionar o ponto inicial do desenho desse arco. Para arcos normais, o ponto inicial pode ser tanto um lugar, quanto por uma transição. Para os arcos inibidores, o ponto inicial deve ser obrigatoriamente um lugar, não é possível iniciar o desenho de um arco inibidor a partir de uma transição. Quando o usuário faz o primeiro clique, posterior ao clique do botão, o evento ***mousedown*** é ativado e a verificação da variável ***buttonPress*** é verificado. Caso ***buttonPress*** seja igual a ***3*** ou ***4*** a função ***addArc*** é chamada. Como o valor da variável ***buttonPress*** não se altera ao longo do desenho do arco, sempre que há um clique na tela, há a chamada da função ***addArc***.

A cada chamada da função ***addArc*** se faz uma verificação de algumas variáveis temporárias que auxiliam a construção dos arcos, são elas:

1. ***startingPositionArc:*** array que indica o ponto inicial, nos eixos X e Y da tela, do arco, além de contter o id do item em que o arco se inicia;

2. ***intermediatePoints:*** array que indica todos os pontos, nos eixos X e Y da tela, depois do ponto inicial e antes do ponto final;

3. ***endPositionArc:*** array que indica a ponto final, nos eixos X e Y da tela, do arco, além de contter o id do item em que o arco termina.

A primeira verificação que se faz na função ***addArc*** é a existência de pontos intermediários. Para essa vericação analisamos o tamanho do array ***startingPositionArc*** e ***endPositionArc***. Se o primeiro for maior que ***0***, e o último igual a ***0***, significa que o clique foi feito para a construção de um ponto intermediário arco. Dessa forma, se faz uma inserção no array ***intermediatePoints*** da posição dos eixos X e Y, passando-se os valores das variáveis ***mouseX*** e ***mouseY***.

```
1 if (startingPositionArc.length > 0 && endPositionArc.length == 0) {
2     intermediatePoints.push([mouseX, mouseY]);
3 }
```

Código 3.18 – Verificação de pontos intermediários

Caso a condição anterior não seja atendida, se entende que o ponto ***X*** e ***Y*** não representa um ponto intermediário do arco. Posteriormente, se percorre os arrays ***arrayPlaces*** e ***arrayTransition***. Para cada uma das posições de ambos os arrays se faz a verificação, através das funções ***isInsidePlace*** e ***isInsideTransition***, se o clique do mouse ocorreu dentro de algum dos lugares ou transições que estão dispostas na tela. Caso o clique ocorra dentro de algum desses dois elementos, o ponto ***X*** e ***Y*** será gravado na variável ***startingPositionArc***, juntamente com o ***id*** do elemento em que o arco se iniciou.

Para lugares

Caso a chamada da função ***isInsidePlace*** retorne ***true*** e a variável ***startingPositionArc*** não contenha nenhum valor, o ponto inicial, com os eixos ***X*** e ***Y*** e o id do lugar, são gravados nas propriedades do arco. Além disso, no lugar em quem o arco se iniciou, insere-se a informação de que aquele arco se iniciou ali. A variável auxiliar ***typeElement*** também recebe o valor ***"place"***, para indicar que aquele arco se iniciou num lugar.

Caso a verificação anterior não seja atendida, se faz uma nova verificação. Se a chamada da função ***isInsidePlace*** retornar ***true*** e a variável ***typeElement*** contiver o valor ***"transition"***, quer dizer que o arco se iniciou em uma transição, e agora está

finalizando em um lugar. Com isso, se adiciona o valor ***X*** **e** ***Y*** do clique e o ***id*** do lugar em que o arco está finalizando na variável ***endPositionArc*** que, posteriormente, será adicionado nas propriedades do arco. O lugar recebe a informação de que esse arco finalizou nele.

Com a finalização do arco, a variável ***drawArc*** recebe o valor ***false*** e a ***typeElement*** recebe ***null***

```
for (var place of arrayPlaces) {
    isInsidePlace = insidePlace(mouseX, mouseY, place.posX, place.posY);
    posEdge = adjustedPositionArcPlace(mouseX, mouseY, place.posX, place.
        posY);
    posEdge.push(place.id);
    if (isInsidePlace && startingPositionArc.length == 0) {
        startingPositionArc.push(posEdge);
        place.connections.push(`Start Arc ${nArcs + 1}`);
        start = place.id;
        typeElement = "place";
    }
    else if (isInsidePlace && typeElement == "transition") {
        endPositionArc.push(posEdge);
        place.connections.push(`Finish Arc ${nArcs + 1}`);
        end = place.id;
        drawArc = false;
        typeElement = null;
    }
}
```

Código 3.19 – Construção de arco verificando os lugares

Para transições

Caso a chamada da função ***isInsideTransition*** retorne ***true*** e a variável ***startingPositionArc*** não contenha nenhum valor, o ponto inicial, com os eixos ***X*** e ***Y*** e o id da transição, são gravados nas propriedades do arco. Além disso, na transição em quem o arco se iniciou, insere-se a informação de que aquele arco se iniciou ali. A variável auxiliar ***typeElement*** também recebe o valor ***"transition"***, para indicar que aquele arco se iniciou numa transição.

Caso a verificação anterior não seja atendida, se faz uma nova verificação. Se a chamada da função ***isInsideTransition*** retornar ***true*** e a variável ***typeElement*** contiver o valor ***"place"***, quer dizer que o arco se iniciou em um lugar, e agora está finalizando em uma transição. Com isso, se adiciona o valor ***X*** **e** ***Y*** do clique e o ***id*** da transição em que o arco está finalizando na variável ***endPositionArc*** que, posteriormente, será adicionado nas propriedades do arco. A transição recebe a informação de que esse arco finalizou nele.

Com a finalização do arco, a variável ***drawArc*** recebe o valor ***false*** e a ***typeElement*** recebe ***null***

```
for (var transition of arrayTransitions) {
    isInsideTransition = insideTransition(mouseX, mouseY, transition.posX,
        transition.posY);
    mouseXY = [mouseX, mouseY, transition.id];
    if (isInsideTransition && startingPositionArc.length == 0 && arcType ==
         "normal") {
        startingPositionArc.push(mouseXY);
        transition.connections.push(`Start Arc ${nArcs + 1}`);
        start = transition.id;
        typeElement = "transition";
    }
    else if (isInsideTransition && typeElement == "place") {
        endPositionArc.push(mouseXY);
        transition.connections.push(`Finish Arc ${nArcs + 1}`);
        end = transition.id;
        drawArc = false;
        typeElement = null;
    }
}
```

Código 3.20 – Construção de arco verificando as transições

Criando o arco com as propriedades

Quando as variáveis temporárias ***startingPositionArc*** e ***endPositionArc*** apresentam valores atribuídos significa que todas as propriedades para se construir um arco estão satisfeitas, são elas:

1. **id:** representa a identidade única do arco que irá identificar esse objeto ao longo de todo o código;
2. **name:** identificador visual e textual do arco na tela;
3. **type:** indica o tipo do arco, se ele é do tipo normal ou inibidor;
4. **startingPositionArc:** indica o valor dos eixos X e Y em que o arco se inicia;
5. **endPositionArc:** indica o valor dos eixos X e Y em que o arco finaliza;
6. **start:** indica o id do elemento em que o arco se inicia;
7. **end:** indica o id do elemento em que o arco finaliza;
8. **intermediatePoints:** cada posição representa o valor dos eixos X e Y de um ponto intermediário do arco;

9. **isEnable:** valor booleano que indica se as exigências para ativação daquele arco estão satisfeitas.
10. **weight:** indica o peso do arco.
11. **weightPos:** indica o valor dos eixos X e Y em que o peso do arco será mostrado na tela.

Tendo essas propriedades definidas se faz a adição do arco no ***arrayArcs***. Por fim, as variáveis temporárias são atribuídas com valores nulos para que o próximo arco possa ser construído.

```
1
2  if (startingPositionArc.length > 0 && endPositionArc.length > 0) {
3      intermediatePoints.pop();
4      objArc = {
5          id: `Arc ${nArcs + 1}`,
6          name: `A${nArcs + 1}`,
7          type: arcType,
8          startingPositionArc: startingPositionArc,
9          endPositionArc: endPositionArc,
10         start: start,
11         end: end,
12         intermediatePoints: intermediatePoints,
13         isEnable: false,
14         weight: 1,
15         weightPos: {x: null, y: null}
16     }
17     arrayArcs.push(objArc);
18     nArcs += 1;
19     startingPositionArc = [];
20     endPositionArc = [];
21     intermediatePoints = [];
22     start = null;
23     end = null;
24 }
```

Código 3.21 – Criando o arco com as propriedades

3.4 Movimentando elementos na tela

A movimentação de elementos é essencial para a construção de uma rede de Petri. A partir do momento em que os elementos são dispostos na tela, eles necessitam ser movimentados para uma melhor visualização e compreensão da rede. Os elementos, por vezes, podem ficar sobrepostos e mal distribuídos no momento de sua criação, gerando desconforto visual. Os elementos que podem ser movidos são:

1. lugares;
2. transições;
3. nome dos elementos;
4. pontos intermediários dos arcos.

Outros elementos, como número de ***tokens***, peso dos arcos, além dos pontos **x** e **y**, iniciais e finais dos arcos, acompanham a movimentação dos elementos principais, que podem ser movidos. Ou seja, não é possível movê-los diretamente.

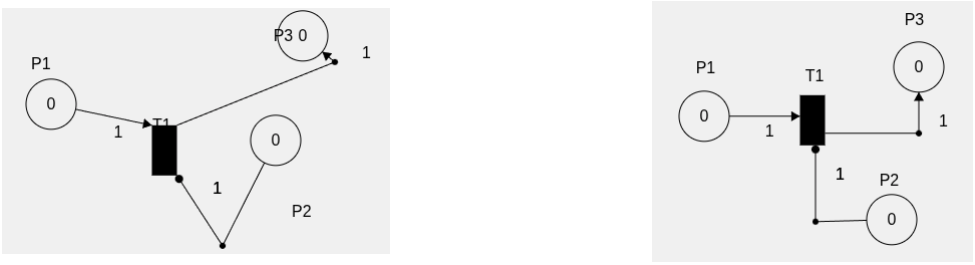

Figura 24 – Elementos mal distribuídos e movimentados para uma melhor visualização.

Além disso, alguns critérios precisam ser atendidos para que o movimento do elemento ocorra. É necessário que o cursor do mouse seja pressionado e segurado dentro da área do elemento a ser movimentado. Para isso, se torna fundamental descobrir se um clique ocorreu dentro da área dos elementos que podem se movimentar. No caso dos lugares e pontos intermediários, se calcula se o clique ocorreu dentro da área do circulo que os formam. No caso das transições e nome dos elementos, se calcula com base no retângulo que os formam.

Verificando clique em círculos

Ao clicarmos na área **canvas** obtemos o ponto **x** e **y** onde ele ocorreu. Assim, temos 2 possibilidades: ocorreu dentro do circulo, no caso, lugar ou ponto intermediário, ou fora. Para determinar se o clique ocorreu dentro de algum dos circulos distribuidos na área **canvas** calcula-se o comprimento entre o ponto **x** e **y** do clique e o ponto **x** e **y** central de cada circulo (MACORATTI, 2014), de forma recursiva. Caso esse comprimento seja inferior ou igual ao raio de algum desses circulos, podemos afirmar que ele ocorreu dentro de algum elemento. Caso contrário, ocorreu fora.

O cálculo é feito utilizando-se da fórmula para distância entre 2 pontos.

$$d_{AB}^2 = (x_B - x_A)^2 + (y_B - y_A)^2 \tag{3.1}$$

```
function insidePlace(mouseX,mouseY,placeX,placeY) {
    var a = mouseX - placeX;
    var b = mouseY - placeY;
    var c = (Math.pow(a,2) + Math.pow(b,2) <= Math.pow(radius,2))
    return c;
}
for (var place of arrayPlaces) {
    isInsidePlace = insidePlace(mouseX, mouseY, place.posX, place.posY)
    if (isInsidePlace && isPress){
        place.posX = mouseX;
        place.posY = mouseY;
    }
}
```

Código 3.22 – Verificando clique e atualizando posição do circulo

Tendo o retorno da função e a variável ***isPress*** 3.9 como verdadeiro, há a satisfação das condições necessárias para movimentar o elemento. Dessa forma, a nova posição do centro do circulo é a posição do ponteiro do mouse, obtida com ***mousemove*** 3.2. A movimentação ocorrerá até que o botão do mouse deixe de ser pressionado.

Verificando clique em retângulos

As transições e nome de elementos podem ser caracterizados como retângulos dentro da área **canvas**. Ao clicar na área **canvas** obtemos as posições **x** e **y** do clique. Para determinarmos se o clique ocorreu dentro de algum dos retângulos verifica-se e comparara-se o ponto **x** e **y** do clique com o ponto inferior esquerdo do retângulo (MANZONI, 2013). Faz-se a verificação da seguinte forma:

1. $mouseX \geq transitionX$;
2. $mouseX \leq transitionX + transitionWidth$;
3. $mouseY \geq transitionY$;
4. $mouseY \leq transitionY + transitionHeigth$.

```
function insideTransition(mouseX, mouseY, transitionX, transitionY) {
    var a = (mouseX >= transitionX) && (mouseX <= transitionX +
        transitionWidth) && (mouseY >= transitionY) && (mouseY <=
        transitionY + transitionHeigth)
    return a;
}

 for (var transition of arrayTransitions) {
```

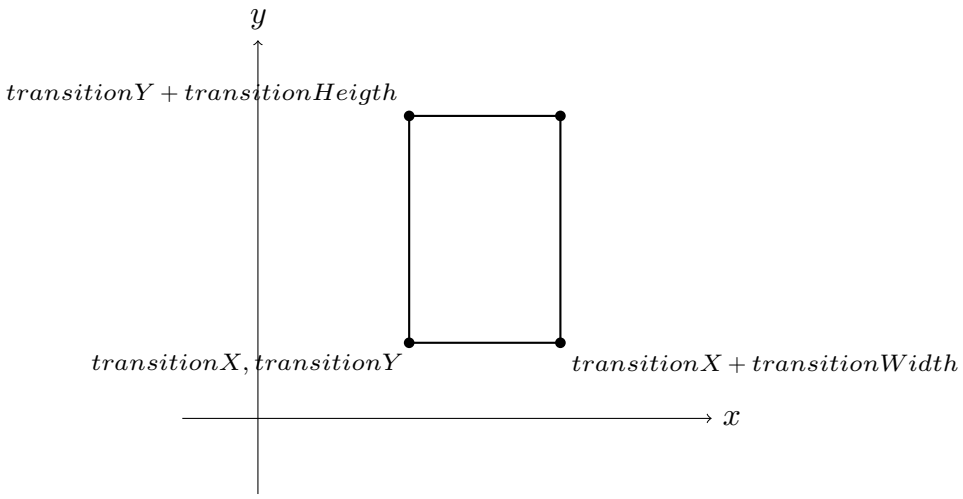

Figura 25 – Transição representada no plano cartesiano.

```
7       isInsideTransition = insideTransition(mouseX, mouseY, transition.posX,
            transition.posY)
8       if (isPress && isInsideTransition) {
9           transition.posX = mouseX - transitionWidth / 2;
10          transition.posY = mouseY - transitionHeigth / 2;
11      }
12  }
```

Código 3.23 – Verificando clique e atualizando posição do retângulo

Caso todas essas comparações sejam verdadeiras, pode-se concluir que o clique ocorreu dentro de algum retângulo. Com isso, tendo ***isPress*** verdadeira, move-se o retângulo. A posição central do retângulo será a mesma do ponteiro do mouse.

3.5 Exclusão

A partir do momento em que elementos são criados, há a necessidade de poder excluí-los, seja para corrigir algum erro, testar novas possibilidades, ou seja para deletar tudo e iniciar de novo. Na aplicação web desenvolvida buscou-se focar na exclusão tanto dos elementos individualmente, no caso, lugares, transições e arcos, quanto também pela exclusão da rede por completo, excluindo tudo de uma única vez.

criar fluxo para explicar o processo

Figura 26 – Diagrama de fluxo para exclusão

Deletando elementos

Para deletar um elemento, de forma individual, é necessário clicar no botão ***Delete Element***. A partir dessa ação há 3 possibilidades:

1. exclusão de um lugar;
2. exclusão de uma transição;
3. exclusão de um arco.

Para deletar um lugar, ou transição, o processo é semelhante. Além de deletar o elemento em si, há a necessidade de deletar os arcos que chegam e que saem, visto que, os arcos são dependentes desses elementos. Após o clique no botão ***Delete Element*** há a mudança do valor da variável ***buttonPress*** para **5** e, em seguida, quando ocorrer o evento ***mousedown*** 3.2, há a chamada da função ***deleteElements()***.

A função percorre os 3 arrays principais de elementos, ***arrayPlaces***, ***arrayTransitions*** e ***arrayArcs***. Dessa forma, há a verificação da ocorrência de um clique dentro da área de algum dos elementos que podem ser excluídos: lugar, transição ou arco. No caso dos arcos, verifica-se se o clique ocorreu na área de algum dos pontos intermediários, ou na área da ponta do arco, sendo a área circular ou triangular, a depender do tipo de arco, normal ou inibidor. Caso a condição seja verdadeira, exclui-se o elemento do seu ***array*** correspondente.

```
canvas.addEventListener('mousedown', (event) => {

    const rect = canvas.getBoundingClientRect();
    const mouseX = event.clientX - rect.left;
    const mouseY = event.clientY - rect.top;

    if (buttonPress == 5) {
        deleteElements(mouseX, mouseY)
    }

    function deleteElements(mouseX, mouseY) {
```

```
12
13          for (var place of arrayPlaces) {
14              isInsidePlace = insidePlace(mouseX, mouseY, place.posX, place.
                    posY)
15              if (isInsidePlace) {
16                  if (place.connections.length > 0) {
17                      for (var connectionPlace of place.connections) {
18                          arcIdPlace = `${connectionPlace.substring(
                                connectionPlace.indexOf('␣') + 1)}`
19                          for (var transition of arrayTransitions) {
20                              for (var connectionTransition of transition.
                                    connections) {
21                                  arcIdTransition = `${connectionTransition.
                                        substring(connectionTransition.indexOf('
                                        ␣') + 1)}`
22                                  if (arcIdTransition == arcIdPlace) {
23                                      indexConection = transition.connections
                                            .indexOf(connectionTransition)
24                                      transition.connections.splice(
                                            indexConection, 1)
25                                  }
26                              }
27                          }
28                          for (var arc of arrayArcs) {
29                              index = arrayArcs.indexOf(arc)
30                              if (arcIdPlace == arc.id) {
31                                  arrayArcs.splice(index, 1)
32                              }
33                          }
34                      }
35                  }
36                  index = arrayPlaces.indexOf(place)
37                  arrayPlaces.splice(index, 1)
38                  buttonPress = 0
39              }
40          }
41      }
42
43  })
```

Código 3.24 – Função ***deleteElements()*** simplificada para excluir um lugar

Na exclusão de lugares e transições, verifica-se quais arcos chegam ou saem desses elementos através da propriedade ***connections*** 7. Tendo as conexões, faz-se também a exclusão desses arcos. Ao deletar um lugar e os arcos que chegam ou que saem, por exemplo, também exclui-se na transição a menção, na propriedade ***connections***, que há uma conexão com o arco excluído. O mesmo vale para quando se exclui uma transição.

Por fim, quando exclui-se um arco, de forma individual, verifica-se o elemento inicial e final, ou seja, onde o arco inicia e onde o arco finaliza. Tendo esses 2 elementos, faz-se a exclusão da menção de conexão na propriedade ***connections*** de cada um desses elementos.

Deletando a rede por completo

A exclusão da rede por completo é uma funcionalidade de extrema importância. Uma vez que se deseja iniciar um novo projeto é necessário limpar a área de desenvolvimento das redes de Petri. A forma como a aplicação web funciona leva em consideração os valores das diferentes variáveis. Ou seja, a rede de Petri, nada mais é, que a interpretação das várias variáveis dentro do código. Inicialmente, quando não se tem nenhum elemento da rede de Petri, as variáveis são todas nulas. Conforme os elementos vão sendo criados, as variáveis vão recebendo valores, e esses valores vão sendo interpretados ao longo de todo o código. Com a exclusão da rede como um todo, se deseja voltar ao estado inicial, em que todas as variáveis são nulas. Dessa forma, para excluir a rede por completo, basta limpar as variáveis, atribuindo valores vazios, zeros e nulos.

```
1  function deleteNet() {
2      localStorage.clear();
3      arrayPlaces = [];
4      arrayTransitions = [];
5      arrayArcs = [];
6      buttonPress = 0
7      nPlaces = 0;
8      nTransitions = 0;
9      nArcs = 0;
10     drawArc = false;
11 }
```

Código 3.25 – Função ***deleteNet()***

A função ***deleteNet()*** retorna todas as variáveis, que compõem a rede de Petri, para seu valor inicial, zero ou vazio. Além disso, ela limpa o ***localStorage*** 3.6, eliminando os dados salvos na memória do navegador.

Entretanto, para executar a função ***deleteNet()***, é necessário que o usuário confirme tal solicitação, visto que, excluir toda a rede é uma operação extrema, em que se excluirá todos os dados referentes a rede de Petri que está sendo desenvolvida até o momento. Para isso, no momento em que o usuário faz o clique no botão **Delete Net**, uma caixa de confirmação aparece, pedindo ao usuário para confirmar a ação de exclusão da rede, ou para cancelar a solicitação.

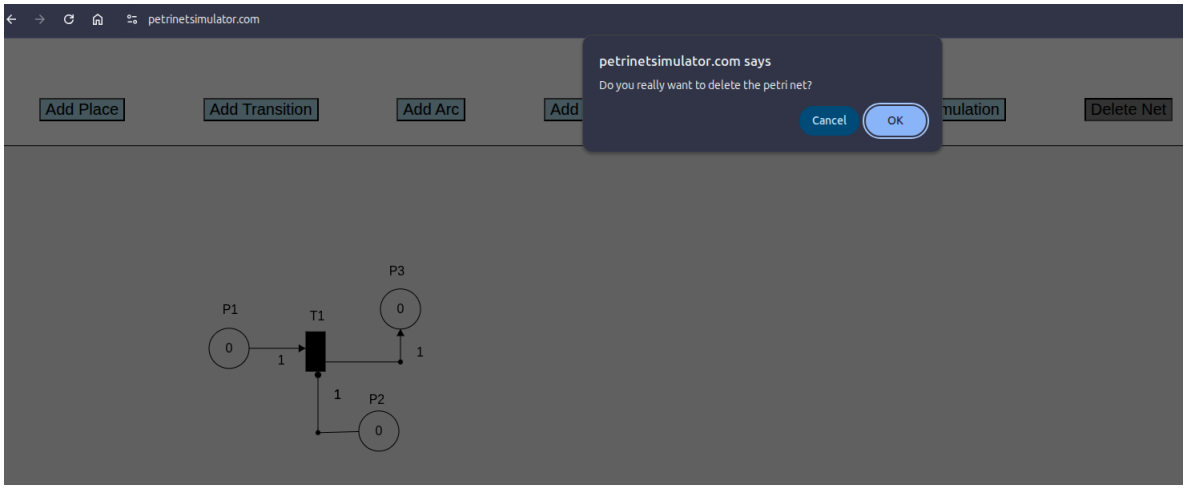

Figura 27 – Caixa de confirmação para exclusão da rede por completo

```
document.getElementById('buttonDeleteNet').addEventListener('click',
    function() {
    var confirmation = confirm("Do you really want to delete the petri net?
        ");
    if (confirmation) {
        deleteNet();
    } else {
        buttonPress = 0;
    }
});
```

Código 3.26 – Código para caixa de confirmação de exclusão da rede por completo

Após a confirmação, por parte do usuário, faz-se a exclusão da rede por completo.

3.6 Salvando e carregando a rede

Com a rede de Petri já construída, torna-se essencial disponibilizar mecanismos que permitam seu salvamento e posterior carregamento, garantindo assim a persistência dos dados e a continuidade do trabalho do usuário. Na aplicação web desenvolvida, essas funcionalidades foram implementadas por meio da exportação e importação de arquivos no formato **JSON**, o que possibilita a portabilidade da rede entre diferentes dispositivos ou sessões. Adicionalmente, a aplicação realiza o salvamento automático no ***localStorage*** do navegador, permitindo a recuperação da rede de Petri mesmo após o fechamento ou recarregamento da página, evitando perdas acidentais de progresso.

Como citado em capítulos anteriores, a rede de Petri, nada mais é, que a interpretação das várias variáveis ao longo de todo o código 3.5. Dessa forma, para que ocorra o salvamento da rede de Petri, é necessário persistir os dados dessas variáveis de maneira

simples e portável. Com isso, um arquivo ***.json***, contendo todos os dados necessários para a replicação da rede de Petri é exportado sempre que ocorre uma ação de salvamento, através do botão ***Save Net*** da rede de Petri e, para o carregamento, é feito uma importação, com o botão ***Load Net***, desse arquivo ***.json***. O formato ***JSON*** é leve e portável, facilitando o compartilhamento da rede de Petri entre os vários usuários.

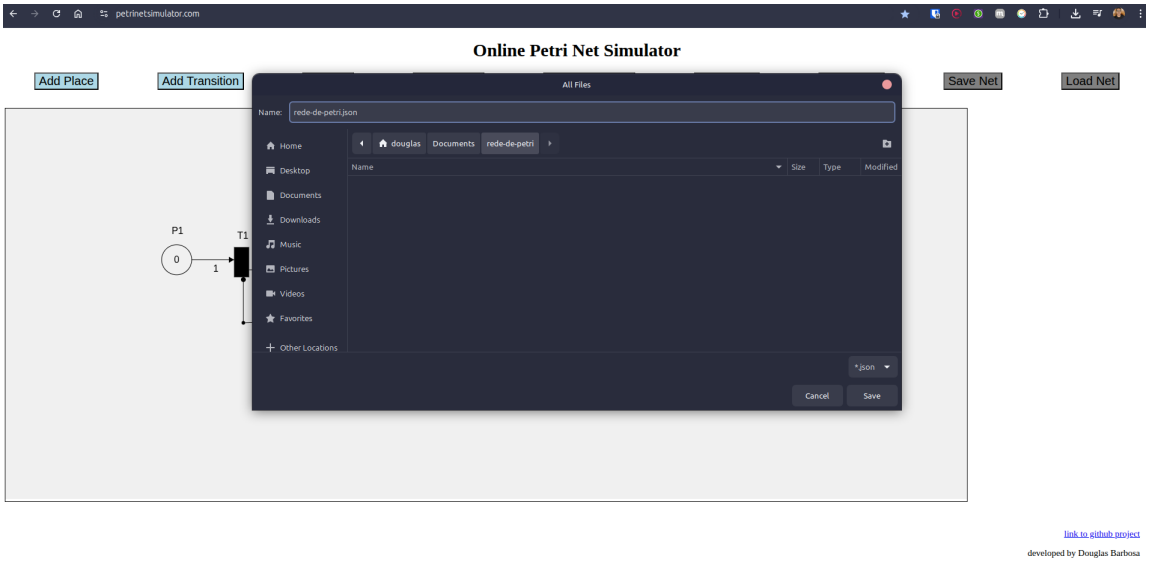

Figura 28 – Tela para exportação da rede de Petri

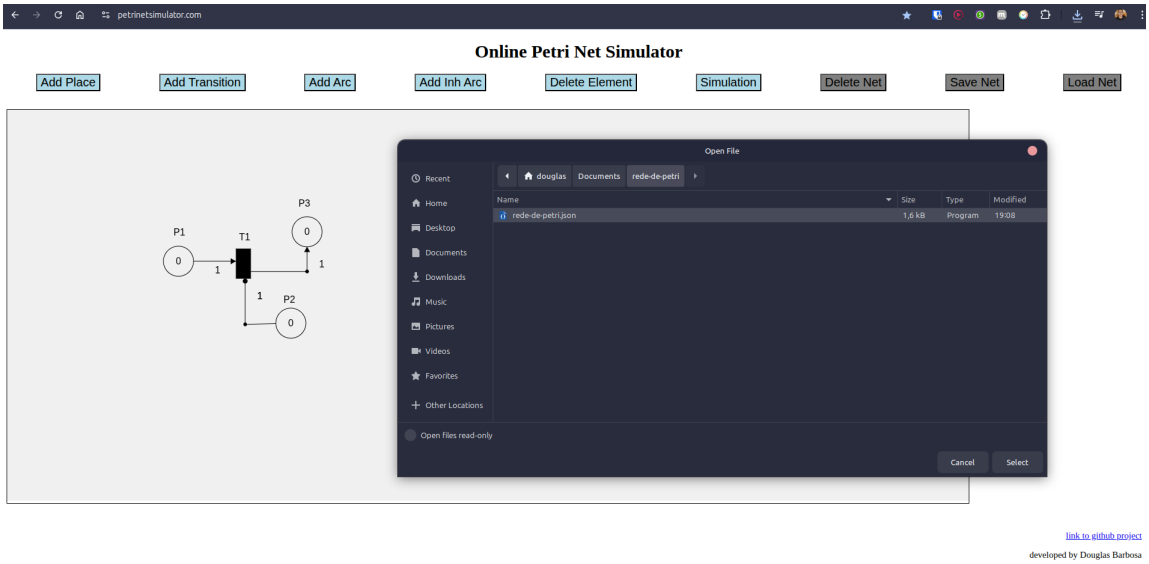

Figura 29 – Tela para importação da rede de Petri

Para a exportação, após o clique do botão ***Save Net***, ocorre a chamada da função ***saveJSON()***. Ela chama outra função, **saveVariables()**, que salva as variáveis. Após, ocorre a exportação dos dados no arquivo ***.json***

```
1 function saveVariables() {
2     variables = {
```

```
        places: arrayPlaces,
        transitions: arrayTransitions,
        arcs: arrayArcs,
        nPlaces: nPlaces,
        nTransitions: nTransitions,
        nArcs: nArcs,
    };
    return variables;
}

function saveJSON() {
    variables = saveVariables();
    const dataStr = JSON.stringify(variables);
    const blob = new Blob([dataStr], { type: "application/json" });
    const url = URL.createObjectURL(blob);
    const a = document.createElement("a");
    a.href = url;
    a.download = "data.json";
    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);
}

document.getElementById("buttonSaveNet").addEventListener("click", saveJSON
    );
document.getElementById("buttonLoadNet").addEventListener("click", () => {
    document.getElementById("fileInput").click();});
```

Código 3.27 – Código para exportação da rede num arquivo ***.json***

Para a importação, após o clique do botão ***Load Net***, ocorre a chamada da função ***loadJSON()***. Ela chama outra função, **loadVariables()**, que carrega as variáveis. Após, ocorre a importação dos dados do arquivo ***.json***

```
function loadVariables(data) {
    arrayPlaces = data.places;
    arrayTransitions = data.transitions;
    arrayArcs = data.arcs;
    nPlaces = data.nPlaces;
    nTransitions = data.nTransitions;
    nArcs = data.nArcs;
}

function loadJSON(event) {
    const file = event.target.files[0];
    const reader = new FileReader();
    reader.onload = function(e) {
        const data = JSON.parse(e.target.result);
```

```
        loadVariables(data);
        };
    reader.readAsText(file);
}

document.getElementById("fileInput").addEventListener("change", loadJSON);
```

Código 3.28 – Código para importação da rede num arquivo ***.json***

localStorage

Além da exportação e importação de arquivos ***.json***, há também o salvamento através do ***localStorage***. O ***localStorage*** é uma funcionalidade nativa dos navegadores modernos que permite o armazenamento de dados localmente no dispositivo do usuário, de forma persistente e sem necessidade de conexão com o servidor (MDN WEB DOCS, 2024d).

```
window.addEventListener('beforeunload', () => {
    variables = JSON.stringify(saveVariables());
    localStorage.setItem('data', variables);
});

window.addEventListener('load', () => {
    data = JSON.parse(localStorage.getItem('data'));
    if (data) {
        loadVariables(data);
    }
});
```

Código 3.29 – Código para persistência dos dados com ***localStorage***

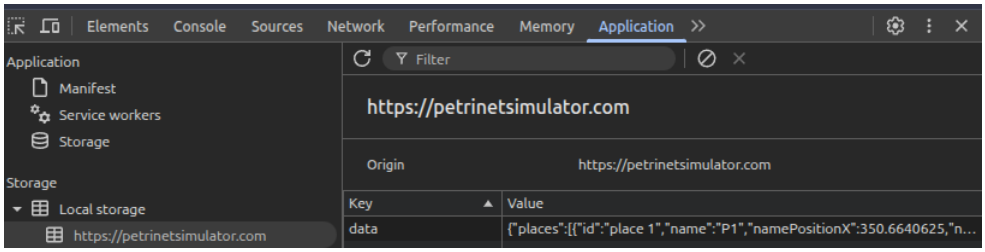

Figura 30 – Dados persistidos no ***localStorage***

Na aplicação desenvolvida, o ***localStorage*** é utilizado como mecanismo complementar de persistência de dados. Sempre que a rede de Petri é modificada, o seu estado atual é convertido para uma representação em formato ***JSON*** e armazenado localmente. Dessa forma, ao retornar à aplicação, o usuário pode retomar seu trabalho a partir do ponto em que parou, sem a necessidade de realizar uma exportação manual. Esta abordagem contribui significativamente para a usabilidade e robustez da ferramenta, proporcionando uma experiência mais fluida e confiável.

3.7 Simulação da rede

A simulação da rede de Petri desenvolvida pelo usuário desempenha um papel fundamental no funcionamento do sistema. Após a criação da rede, torna-se necessário permitir sua manipulação e execução. Como as redes de Petri possuem uma representação matemática bem definida, a simulação pode ser realizada de forma eficiente. Para isso, é necessário comparar, modificar e atualizar as variáveis que compõem a estrutura da rede, como número de marcações dos lugares, peso dos arcos e verificando se uma transição está ativada ou não.

Para simular a rede de Petri, duas funções principais foram desenvolvidas. A primeira, denominada ***netSimulationEnables()***, analisa todas as transições, verificando quais delas estão ativadas. Cada transição é analisada individualmente, verificando se as condições para ativação estão sendo cumpridas. Caso sim, a propriedade ***isEnable*** 8 da transição é definida como ***true*** e a cor da transição na tela se altera para vermelho 3.1.

```
function netSimulationEnables() {
    for (var transition of arrayTransitions) {
        var arrayIsEnable = []
        for (var connection of transition.connections) {
            var point = connection.substring(0, connection.indexOf('␣'))
            if (point == "Finish") {nu
                var arcId = `${connection.substring(connection.indexOf('␣')
                    + 1)}`
                for (var arc of arrayArcs) {
                    if (arc.id == arcId) {
                        weightArc = arc.weight
                        placeStart = arc.startingPositionArc[0][2]
                        for (var place of arrayPlaces) {
                            if (place.id == placeStart) {
                               if (arc.type == "normal") {
                                    if (place.nTokens >= arc.weight) {
                                            arc.isEnable = true
                                    }
                                    else {
                                            arc.isEnable = false
                                    }
                                }
                                else if (arc.type == "inhibitor") {
                                    if (place.nTokens < arc.weight) {
                                        arc.isEnable = true
                                    }
                                    else {
                                        arc.isEnable = false
                                    }
                                }
```

```
                                arrayIsEnable.push(arc.isEnable)
                            }
                        }

                    }

                }
            }
        }
        allTrue = arrayIsEnable.every(value => value === true);
        if (allTrue) {
            transition.isEnable = true
        }
        else {
            transition.isEnable = false
        }
    }
}
```

Código 3.30 – Função ***netSimulationEnables()***

Inicialmente, percorre-se todas as transições analisando quais arcos estão chegando. Essa etapa é feita observando a propriedade ***connections*** de cada transição. Analisando o peso do arco, em comparação com o número de marcações do lugar em que ele está saindo, determina-se se o arco está ativado ou não.

Arcos normais: número de marcações $\geq$ peso do arco;

Arcos inibidores: número de marcações $<$ peso do arco.

Após cada verificação de arco, o resultado é armazenado no ***arrayIsEnable***. Caso todas as verificações sejam verdadeiras, a propriedade ***isEnable*** da transição é alterada para ***true***, caso contrário, como ***false***.

A segunda função, denominada ***netSimulationMove()***, faz a movimentação das marcações, conforme o usuário clica na transição que está ativada. A função é chamada sempre que ocorre o evento ***mousedown*** e a variável ***simulation*** é definida como***true***, o que ocorre sempre que o botão ***Simulation*** é pressionado.

```
function buttonNetSimulation() {
    cleanVariables()
    if (!simulation) {
        buttonPress = 6
        simulation = true
    }
    else if (simulation) {
```

```
        buttonPress = 0
        simulation = false
    }
}
canvas.addEventListener('mousedown', (event) => {
    const rect = canvas.getBoundingClientRect();
    const mouseX = event.clientX - rect.left;
    const mouseY = event.clientY - rect.top;
    if (simulation) {
        netSimulationMove(mouseX, mouseY)
    }
})
```

Código 3.31 – Código para ativação da simulação

Tendo a simulação ativada, o usuário deve fazer o clique nas transições que deseja, simulando o comportamento da rede e movimentando as marcações do lugar de origem para o lugar de destino. A função ***netSimulationMove()*** executa duas ações:

1. subtrair do lugar de origem o número de marcações com base no peso do arco que sai;
2. adicionar ao lugar de destino o número de marcações com base no peso do arco que chega.

```
function netSimulationMove(mouseX, mouseY) {
    for (var transition of arrayTransitions) {
        isInsideTransition = insideTransition(mouseX, mouseY, transition.
            posX, transition.posY)
        if (isInsideTransition && transition.isEnable) {
            for (var connection of transition.connections) {
                var point = connection.substring(0, connection.indexOf('␣')
                    )
                if (point == "Start") {
                    var arcId = `${connection.substring(connection.indexOf(
                        '␣') + 1)}`
                    for (var arc of arrayArcs) {
                        if (arc.id == arcId) {
                            placeEnd = arc.endPositionArc[0][2]
                            for (var place of arrayPlaces) {
                                if (place.id == placeEnd && arc.type == "
                                    normal") {
                                    place.nTokens = place.nTokens + arc.
                                        weight
                                }
                            }
```

```
                        }
                    }
                }
            }
        }
        if (isInsideTransition && transition.isEnable) {
            for (var connection of transition.connections) {
                var point = connection.substring(0, connection.indexOf('␣')
                    )
                if (point == "Finish") {
                    var arcId = `${connection.substring(connection.indexOf(
                        '␣') + 1)}`
                    for (var arc of arrayArcs) {
                        if (arc.id == arcId) {
                            placeStart = arc.startingPositionArc[0][2]
                            for (var place of arrayPlaces) {
                                if (place.id == placeStart && arc.type == "
                                    normal") {
                                    place.nTokens = place.nTokens - arc.
                                        weight
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Código 3.32 – Função ***netSimulationMove()***

O primeiro passo da função verifica se o clique ocorreu dentro de alguma transição e se ela está ativada. Caso sim, as duas ações ocorrem. A primeira subtrai do lugar de origem o valor correspondente ao peso do arco que está saindo. A segunda soma ao lugar de destino o valor correspondente ao peso de arco. Também, é necessário que os arcos para essas ações sejam do tipo normal. O peso dos arcos inibidores não interferem na subtração ao adição de marcações, apenas na ativação dos arcos.

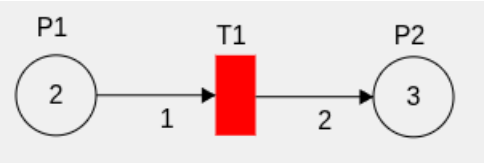

Figura 31 – Movimentação das marcações - Parte 1

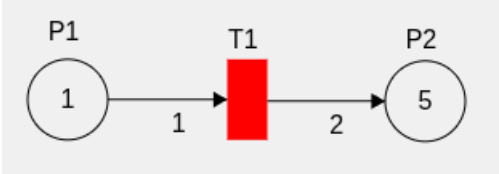

Figura 32 – Movimentação das marcações - Parte 2

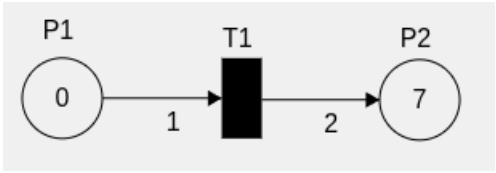

Figura 33 – Movimentação das marcações - Parte 3

Através da função ***netSimulationMove()*** simula-se o comportamento da rede de Petri.

3.8 Publicação do projeto online

4 Considerações Finais

Escrever aqui sobre as expectativas futuras. Pode-se colocar aquilo que ainda pode ser desenvolvido na aplicação, como: redes coloridas, estocásticas, desenvolvimento de uma interface mais agradável, dentre outras melhorias

Falar aqui sobre criação de uma tela de login, e banco de dados para armazenamento das redes de petri criadas. Melhorias de interface como adicionar um texto explicando o que o usuário deve fazer quando executa uma ação.

Referências

BOYLE, Robert. *The works of the Honourable Robert Boyle*. Edição: Thomas Birch. London: J. e F. Rivington, 1772. 6 v. 1062 p. Disponível em: http://bit.ly/boyle-works. Citado 0 vez nas páginas 11, 12.

CASSANDRAS, Christos G.; LAFORTUNE, Stéphane. *Introduction to Discrete Event Systems*. 2. ed. New York: Springer, 2008. ISBN 978-0-387-33332-8. DOI: 10.1007/978-0-387-68612-7. Citado 1 vez nas páginas 13–15.

COUTINHO, Luciano. A terceira revolução industrial e tecnológica. As grandes tendências das mudanças. *Economia e sociedade*, Universidade Estadual de Campinas (UNICAMP), Instituto de Economia, v. 1, n. 1, p. 69, 1992. Citado 1 vez na página 13.

FRANCÊS, Carlos Renato Lisboa. Introdução às redes de petri. *Laboratório de Computação Aplicada, Universidade Federal do Pará*, 2003. Citado 1 vez na página 17.

LINS, Bernardo Felipe Estellita. A evolução da Internet: uma perspectiva histórica. *Cadernos Aslegis*, v. 48, p. 11–45, 2013. Citado 1 vez na página 9.

MACORATTI, José Carlos. *Verificando a relação de um ponto e uma circunferência*. 2014. https://www.macoratti.net/14/05/c_vpc1.htm. Acessado em 28 de maio de 2025. Disponível em: https://www.macoratti.net/14/05/c_vpc1.htm. Citado 1 vez na página 44.

MANZONI, Raymond. *How to check if a point is inside a rectangle?* 2013. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/190373 (version: 2023-02-14). eprint: https://math.stackexchange.com/q/190373. Disponível em: https://math.stackexchange.com/q/190373. Citado 1 vez na página 45.

MDN WEB DOCS. *Element: mousedown event - Web APIs | MDN*. 2024. Accessed: 2025-05-22. Disponível em: https://developer.mozilla.org/en-US/docs/Web/API/Element/mousedown_event. Citado 1 vez na página 30.

MDN WEB DOCS. *Element: mousedown event - Web APIs | MDN*. 2024. Accessed: 2025-05-22. Disponível em: https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseup_event. Citado 1 vez na página 30.

MDN WEB DOCS. *Element: mousemove event - Web APIs | MDN*. 2024. Accessed: 2025-05-22. Disponível em: https://developer.mozilla.org/en-US/docs/Web/API/Element/mousemove_event. Citado 1 vez na página 28.

MDN WEB DOCS. *Element: Window.localStorage - Web APIs | MDN*. 2024. Accessed: 2025-05-22. Disponível em: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage. Citado 1 vez na página 53.

PETRI, Carl. *Kommunikation mit Automaten*. 1962. Tese (Doutorado) – TU Darmstadt. Citado 1 vez na página 9.