UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MESTRADO ACADÊMICO EM COMPUTAÇÃO

FLÁVIO YURI DE SOUSA

SIMPLIFICAÇÃO DE MODELOS DE PROCESSOS UTILIZANDO PROPRIEDADES DE AUTÔMATOS E MODULARIZAÇÃO

QUIXADÁ

2023

FLÁVIO YURI DE SOUSA

SIMPLIFICAÇÃO DE MODELOS DE PROCESSOS UTILIZANDO PROPRIEDADES DE AUTÔMATOS E MODULARIZAÇÃO

Dissertação apresentada ao Curso de Mestrado Acadêmico em Computação do Programa de Pós-Graduação em Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Computação. Área de Concentração: Algoritmos e Teoria da Computação

Orientador: Prof. Dr. Davi Romero de Vasconcelos

QUIXADÁ

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S696s Sousa, Flávio Yuri de.
Simplificação de modelos de processos utilizando propriedades de autômatos e modularização / Flávio Yuri de Sousa. – 2023.
135 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Campus de Quixadá, Programa de Pós-Graduação em Computação, Quixadá, 2023.
Orientação: Prof. Dr. Davi Romero de Vasconcelos.

1. Gestão de Processos de Negócios. 2. Mineração de Processos. 3. Teoria dos Autômatos. I. Título.
CDD 005

FLÁVIO YURI DE SOUSA

SIMPLIFICAÇÃO DE MODELOS DE PROCESSOS UTILIZANDO PROPRIEDADES DE AUTÔMATOS E MODULARIZAÇÃO

Dissertação apresentada ao Curso de Mestrado Acadêmico em Computação do Programa de Pós-Graduação em Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Computação. Área de Concentração: Algoritmos e Teoria da Computação

Aprovada em: 20 de novembro de 2023

BANCA EXAMINADORA

Prof. Dr. Davi Romero de Vasconcelos (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Edward Hermann Haeusler
Pontíficia Universidade Católica do Rio de Janeiro
(PUC-RIO)

Prof. Dr. Régis Pires Magalhães
Universidade Federal do Ceará (UFC)

Prof. Dr. Criston Pereira de Souza
Universidade Federal do Ceará (UFC)

Aos meus pais, Maria Sueli de Sousa Costa e José Flávio de Sousa por tudo que fizeram por mim.

AGRADECIMENTOS

À minha família, em especial aos meus pais, Maria Sueli de Sousa Costa e José Flávio de Sousa por todo apoio e pelo incentivo quando necessário. Vocês sempre fizeram de tudo por mim. Amo vocês mais que tudo, simples assim. Aos meus primos que são o mesmo que serem meus irmãos, Lucas Sousa, Carlos Magno, Amanda Beatriz, Antônio Ednei, Ellis Fernanda, Carliane Sousa, Edenisie e Edenilson Filho, Thiago e Diego, Hevily, Havila (acho que é assim o nome de vocês) e Ísis, meu afilhado Luis Eduardo, e todos os outros primos (a família é enorme). Aos meus tios e tias, avôs e avós por tudo.

Aos meus amigos que fiz no colégio e que me aguentam até hoje, Sávio, Beatriz, Jêsca, Segundo, Vinícius. Aos amigos que fiz na graduação e que também me aguentam até hoje, Alex, Leo, Juliana, Pedro, Pedro, Tutu, Décio, e todos os outros da turma, em especial ao Ronildo, ao Leonardo, o jogador, e ao Mamá, por me ajudarem bastante nesses tempos nem que seja apenas pela descontração (no caso do Ronildo foi ajuda também na parte de suportar o mestrado em meio a uma pandemia e na parte de estudar junto pras disciplinas). Agradeço também aos meus amigos Mateus Felipe, Manu (que no momento sofre pelo BTS), Matheus Cavalcante por tudo, só por vocês existirem eu já estaria feliz. Sem os citados, eu não teria conseguido.

Ao professor Davi Romero de Vasconcelos, por não ter desistido de mim apesar de ter motivos, pelos puxões de orelha quando necessários e por toda a orientação do trabalho. Um farol no meio do nada pra guiar o marinheiro perdido.

Aos professores da Universidade Federal do Ceará que contribuíram pra minha formação (sintam-se culpados), em especial aos professores Régis Pires e Criston Souza que participaram do projeto e da minha banca, junto do professor Edward Hermann, a quem também agradeço, e ao professor Paulo de Tarso pela ajuda nos momentos finais do trabalho apesar de estar altamente atarefado.

Agradeço à Secretaria da Fazenda do Ceará, a FUNCAP e ao laboratório Ísis pelo financiamento do trabalho e pelos dados cedidos para que utilizássemos nos testes.

Agradeço à minha banda, que não existe mais, mas está aí nas plataformas da internet (escuta aí vai ACHADOS E PERDIDOS) e aos caras que fizeram parte dela comigo. À galera dos rachas pelos momentos de diversão e pela constatação que eu não poderia confiar nos meus dotes futebolísticos pra ganhar a vida. Ao Santos Futebol Clube, por me mostrar que se alguma coisa pode piorar, ela vai piorar (MAS VAI VOLTAR PRA PRIMEIRA DIVISÃO COMO UMA

FÊNIX ALADA DO NORTE DO ZIMBÁBUE). Ao Monty Python e o Nas Garras da Patrulha. Ao Neil Gaiman por ter criado Sandman e ter escrito um episódio muito bom de doctor who (a quem também agradeço). Aos medabots e os cavaleiros dos zodíaco. Ao Ednaldo Pereira. Ao John Lennon por ter criado os Beatles. Aos alienígenas de Quixadá e a todo mundo que tenho pelo menos um mínimo de simpatia.

“Às vezes é difícil separar a coragem da burrice
e a persistência da falta do que fazer.”

(Craque Daniel)

RESUMO

No contexto da ciência da computação, podemos definir um processo como um conjunto de atividades que acontecem em uma sequência lógica afim de se obter resultados. A gestão de processos de negócio (*Business Process Management* ou BPM) é o conjunto de técnicas, ferramentas e métodos usados para amparar as fases do ciclo da vida de um processo de negócio, trazendo diversos benefícios para seu usuário, tais como simulação, documentação e uma maior compreensão dos processos. Mineração de processos se utiliza de *logs* de eventos, ou seja, conjuntos de atividades, para extrair conhecimento, podendo ser representado de várias formas, como por *business process management notation* (BPMN), redes de Petri ou sistemas de transições, como autômatos, por exemplo. Quando representamos *logs* de eventos complexos, com muitos processos, por meio de autômatos, verifica-se que há problemas de explosão de estados, gerando um modelo ininteligível. Este trabalho propõe uma abordagem que utiliza propriedades de autômatos e conceitos de decomposição de autômatos para modularizar processos e simplificar os modelos em sistemas de transições. Uma análise da abordagem é apresentada, com foco nas vantagens e desvantagens de utilizar sistemas de transições na modelagem de processos, mais especificamente no número de componentes e na acurácia dos modelos. Os resultados obtidos mostram que a abordagem proposta consegue diminuir o número de estados e transições de um modelo em sistema de transições com sucesso, tornado-o assim mais legível, mas os modelos gerados pelos algoritmos de descoberta *inductive miner* e *heuristic miner* normalmente possuem uma maior acurácia para comportamentos não vistos.

Palavras-chave: gestão de processos de negócios; mineração de processos; teoria dos autômatos; decomposição de autômatos.

ABSTRACT

In the context of computer science, we can define a process as a set of activities that takes place in a logic sequence to obtain results. The business process management (BPM) is the set of tecnics, tool and methods used to support the phases of life cycle of a business process, bringing several benefits to its user, such as simulation, documentation and a greater understanding of the processes. Process mining uses event logs, i.e, sets of activities, to extract knowledge, being able to be represented in many ways, such as business process management notation (BPMN), Petri nets or transitions systems, such as automata, for example. When we represent complex event logs, with a lot of process, through automata, it is verified that there are state explosion problem, generating an intelligible model. This work proposes an approach that uses automata decomposition concepts to modularise processes. In the preliminary experiments, we present promising results in the modularisation of a simple automata, but in the future, minimal automata generated from a real event log provided by the Ceará State Finance Department (SEFAZ-CE) will be used.

Keywords: business process management; process mining; automata theory; automata decomposition.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

No contexto da ciência da computação, *processos* são conjuntos de atividades que acontecem numa sequência lógica a fim de obter-se resultados. De acordo com Gomes (2016), geralmente os processos apresentam uma estrutura em comum: uma entrada, que são os dados necessários para que o processo seja computado, um processamento, que irá manipular a entrada de forma que seja obtida uma saída para o usuário, e uma saída, que será um dado obtido através do processamento.

Chamamos de gestão de processos de negócios (*Business Process Management* ou BPM) o "conjunto de técnicas, ferramentas e métodos usados para amparar as fases de *design*, execução, gestão e análise de processos operacionais de negócios" (HOFSTEDE; WESKE, 2003), ou seja, o BPM é usado para dar suporte a todas as fases do ciclo da vida de um processo de negócio.

O BPM surgiu devido a uma mudança de foco nas empresas em torno dos processos, antes, o foco estava no fluxo de trabalho de áreas funcionais, porém, foi sendo substituído pelo foco nos processos de negócios.

Segundo Gomes (2016), a utilização de BPM traz diversos benefícios, tais como a simulação, a documentação, a execução, o controle e uma maior compreensão dos processos por parte do leitor, além de agregar valor à organização e gerar valo aos seus clientes.

Segundo Aalst *et al.* (2011) a ideia de mineração de processos (*process mining*) é descobrir, monitorar e melhorar processos reais extraindo conhecimentos a partir de um *log* de eventos, providenciando uma conexão entra a mineração de dados e a análise e modelagem de processos de negócios. Mineração de processos inclui descoberta de processos automatizado, verificação de conformidade, construção de modelos de simulação, extensão do modelo, reparo do modelo, previsão de caso e recomendações baseadas em histórico. Um *log* de eventos é um conjunto de *traces*, onde cada *trace* é uma sequência finita de atividades, também chamadas de eventos. De acordo com Aalst (2012), os *logs* de eventos são o ponto de partida para a mineração de processos. A Figura 1 nos mostra um exemplo de um trecho de um *log* de eventos onde cada linha corresponde a um evento, cada evento possui algumas propriedades, representadas nas colunas da tabela, são elas: *Case id*, que indica a qual *trace* o evento está associado, *Event id*, que mostra um identificador único para cada evento, *Timestamp*, que diz o momento em que o evento aconteceu, *Activity*, que diz qual atividade é realizada no evento, *Resource*, que indica quem está realizado a atividade e *Cost* que indica o custo que a atividade terá. Para cada *log* de

evento as colunas *Case id*, *Timestamp* e *Activity* serão essenciais, enquanto que as outras colunas serão consideradas como informações adicionais que podem serem usadas para análise.

Figura 1 – Exemplo de trecho de *log* de eventos

Case id	Event id	Properties				
		Timestamp	Activity	Resource	Cost	...
1	35654423	30-12-2010:11.02	register request	Pete	50	...
	35654424	31-12-2010:10.06	examine thoroughly	Sue	400	...
	35654425	05-01-2011:15.12	check ticket	Mike	100	...
	35654426	06-01-2011:11.18	decide	Sara	200	...
	35654427	07-01-2011:14.24	reject request	Pete	200	...
2	35654483	30-12-2010:11.32	register request	Mike	50	...
	35654485	30-12-2010:12.12	check ticket	Mike	100	...
	35654487	30-12-2010:14.16	examine casually	Pete	400	...
	35654488	05-01-2011:11.22	decide	Sara	200	...
	35654489	08-01-2011:12.05	pay compensation	Ellen	200	...
3	35654521	30-12-2010:14.32	register request	Pete	50	...
	35654522	30-12-2010:15.06	examine casually	Mike	400	...
	35654524	30-12-2010:16.34	check ticket	Ellen	100	...
	35654525	06-01-2011:09.18	decide	Sara	200	...
	35654526	06-01-2011:12.18	reinitiate request	Sara	200	...
	35654527	06-01-2011:13.06	examine thoroughly	Sean	400	...
	35654530	08-01-2011:11.43	check ticket	Pete	100	...
	35654531	09-01-2011:09.55	decide	Sara	200	...
	35654533	15-01-2011:10.45	pay compensation	Ellen	200	...
4	35654641	06-01-2011:15.02	register request	Pete	50	...
	35654643	07-01-2011:12.06	check ticket	Mike	100	...
	35654644	08-01-2011:14.43	examine thoroughly	Sean	400	...
	35654645	09-01-2011:12.02	decide	Sara	200	...
	35654647	12-01-2011:15.44	reject request	Ellen	200	...
...	...	...	...	...	...	...

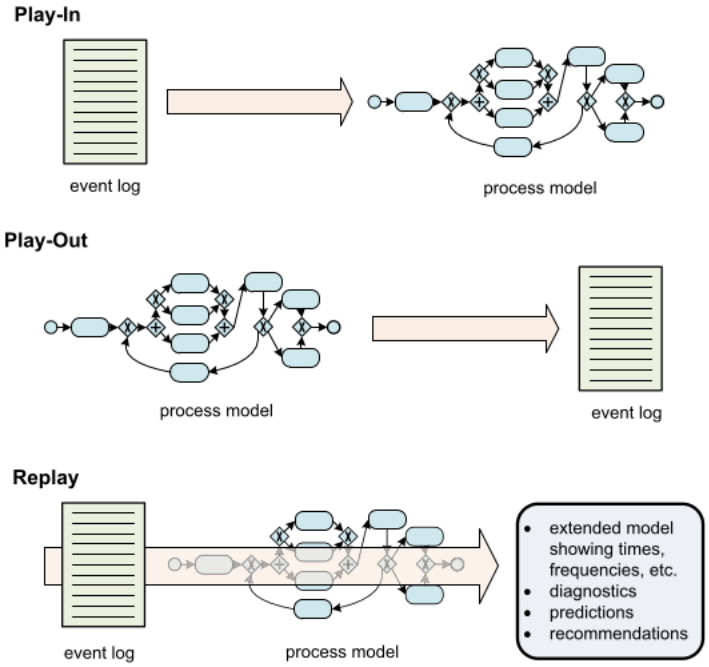
Fonte: (AALST, 2016)

De acordo com Aalst (2016), um elemento chave da mineração de processos é o estabelecimento de um forte laço entre um modelo de processo e a "realidade". Para representar essa relação, utilizamos os termos *play-in*, onde dado um *log* de eventos como entrada o objetivo é construir um modelo que o represente, *play-out*, onde dado um modelo de entrada o objetivo é gerar *log* de eventos, e *replay*, que utiliza um *log* de eventos e um modelo de processo como entrada e faz um *replay*, ou seja, verifica se o modelo de processos corresponde ao *log* de eventos para verificar conformidade, estender o modelo com novas informações, entre outras utilidades. A Figura 2 ilustra essas noções.

Existem diferentes formas de representar visualmente os modelos de processos de negócios, como por exemplo o *Business Process Management Notation* (BPMN) (WHITE, 2004), redes de Petri (AALST, 2019) e sistemas de transições (AALST, 2016), como autômatos.

Uma das formas mais tradicionais de representação de processos é por meio de autômatos, uma das suas principais características é ser uma modelagem mais simples por conter apenas estados e transições. Apesar de o modelo ser simples, uma grande dificuldade encontrada em representações por sistemas de transições é que quando o processo é muito complexo, o sistema gerado pode ter baixa legibilidade devido ao problema conhecido como

Figura 2 – *Play-in, play-out, replay*



Fonte: (AALST, 2016)

"explosão de estados" (AALST *et al.*, 2006) devido ao paralelismo dos processos. Para evitar esse problema, será utilizada uma abordagem de simplificação e de modularização, baseada nas ideias de decomposição de autômatos (KROHN *et al.*, 1968), dos modelos.

Decomposição de autômatos tem sido estudada há anos como no trabalho de Krohn e Rhodes (1965). O que se quer ao decompor autômatos é encontrar autômatos mais simples que o original que, quando conectados, sejam capazes de realizar as mesmas computações do autômato original. Tais conexões podem ser feitas em forma de cascada, ou seja, um processamento só acontece quando o processo anterior acabar, de forma paralela, ou então pode se utilizar alguma propriedade de fechamento, como interseção, para gerar um novo autômato.

Trabalhos como os de Aalst *et al.* (2006) e Kalenkova *et al.* (2014) tratam de sistemas de transições no contexto de mineração de processos, sendo complementares entre si. O primeiro trabalho apresenta uma abordagem que utiliza teoria das regiões para sintetizar uma rede de Petri a partir de um sistema de transições gerado de um *log* de eventos. O segundo trabalho apresenta com mais detalhes como é feita a decomposição do sistema de transição para a aplicação da teoria das regiões para gerar uma rede de Petri a fim de descobrir modelos mais legíveis. Outros trabalhos como os de Gaži e Rovan (2008) e Rovan e Sádovskỳ (2018) trabalham ideias de decomposição de autômatos para resolver o problema de soluções assistidas, que é quando tem-se um processo resolvedor para um problema e um processo "conselheiro" que irá fazer

um pré-processamento da entrada, tornando o processo mais simples. Enquanto o primeiro faz a decomposição de um DFA utilizando a noção de partições para encontrar um autômato resolvedor e um autômato "conselheiro", o segundo estende essas ideias a nível de NFA.

O objetivo geral deste trabalho é simplificar modelos de processos representados por autômatos utilizando propriedades de autômatos e modularização de processos a partir da decomposição de autômatos, aumentando assim a sua legibilidade. Para isso, definimos os objetivos específicos desta forma: primeiro é gerado um sistema de transições a partir de um log de eventos, que, para fins de testagem, utilizaremos três *logs* de eventos reais disponibilizados pela Secretaria da Fazenda do Estado do Ceará (SEFAZ-CE) e dois *logs* de eventos utilizados em competições como a *BPI Challenge*, logo após utilizamos propriedades de equivalências de autômatos para simplificá-los, depois o autômato é modularizado e, por fim, comparamos a nossa solução com os modelos obtidos utilizando algoritmos de descoberta bastante utilizados na mineração de processos, os critérios de comparação serão o número de componentes e acurácia dos modelos.

O restante deste trabalho está assim organizado: no Capítulo 2 serão aprofundados os conceitos presentes no trabalho. O Capítulo 3 apresenta trabalhos relacionados ao tema proposto; o Capitulo 4 detalha a proposta deste trabalho; o Capitulo 5 apresenta os resultados obtidos pelos experimentos e, no Capítulo 6 serão apresentadas algumas considerações sobre o trabalho e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, serão apresentados os conceitos principais utilizado por este trabalho. A Seção 2.1 aborda o tema de teoria dos autômatos pois nossa proposta utiliza diferentes tipos de autômatos e suas propriedades de fechamento. Na Seção 2.2 é mostrado os conceitos de gerência de processos de negócios, *bussiness process management*, com foco na mineração de processos e na representação de processos por meio de sistemas de transições.

2.1 Teoria dos Autômatos

Em teoria dos autômatos (HOPCROFT *et al.*, 2006), um **problema** é definido pela questão de decidir se, dada uma palavra, ela pertence a alguma linguagem particular. O conceito de linguagem é definido da forma que se segue:

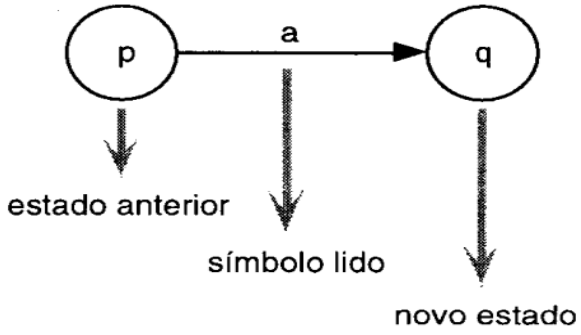
Definição 1 *(HOPCROFT et al., 2006) Seja* Σ *um conjunto finito e não vazio de símbolos de entradas, chamado de* ***alfabeto****. Uma* ***palavra*** *é uma sequência de símbolos do alfabeto. A palavra vazia, representada por* ε*, é a palavra que não contém símbolos.* Σ^* *é o conjunto de todas as palavras possíveis de um alfabeto* Σ*. Uma linguagem L é um conjunto de palavras sobre um alfabeto* Σ*.*

De acordo com Menezes (1998), um autômato finito pode ser visto como uma máquina composta por três partes:

- **Fita:** Dispositivo finito que contém a informação a ser processada (entrada);
- **Unidade de controle:** Reflete o estado atual da máquina. Possui uma unidade de leitura (cabeça da fita) que irá ler um símbolo de entrada por vez, movendo-se sempre para a direita;
- **Função de transição:** Comanda as leituras e define o estado da máquina, ou seja, a partir do estado atual, determina qual o próximo estado considerando o símbolo que está sendo lido;

A função de transição pode ser representada por um grafo direcionado, como mostrado na Figura 3. No grafo, chamamos de **estado inicial** o estado da máquina no momento em que irá se iniciar a computação e de **estado final** ou **estado de aceitação** o estado que irá determinar a aceitação da entrada. Graficamente, o estado inicial é representado com uma seta apontada para ele, enquanto que os estados finais são representados por círculos duplos.

Figura 3 – Função de transição como um grafo



Fonte: (MENEZES, 1998)

Diz-se que um autômato aceita uma palavra se, a partir de um estado inicial do autômato, há transições que levam até um estado final lendo todos os símbolos da palavra. Os autômatos podem ser autômatos finitos determinísticos (*Deterministic finite automata* ou DFA) (HOPCROFT *et al.*, 2006) ou autômatos finitos não-determinísticos (*Non-deterministic finite automata* ou NFA) (HOPCROFT *et al.*, 2006). Suas definições serão mostradas a seguir.

De acordo com Hopcroft *et al.* (2006), um DFA pode ser definido pela quíntupla:

$$\mathrm{M} = (\mathrm{Q}, \Sigma, \delta, q_0, F) \tag{2.1}$$

onde:

- Q é o conjunto dos estados;
- Σ é o alfabeto de entrada;
- δ é a função de transição onde: $Q \times \Sigma \rightarrow Q$;
- q_0 é o estado inicial;
- F é o conjunto de estados finais.

A utilização do termo *determinístico* acontece pelo motivo de que enquanto houver um símbolo de entrada a ser lido sempre será possível determinar o próximo estado que o autômato irá assumir e esse estado será único em todas as situações.

A função de transição estendida é útil para realizar a computação de uma palavra e é definida da seguinte forma:

Definição 2 *(HOPCROFT et al., 2006) Uma* ***função de transição estendida*** $\overline{\delta}_D : Q_D \times \Sigma^* \rightarrow Q_D$ *é definida por indução da forma:*

$$\overline{\delta}_D(q, \varepsilon) = q \tag{2.2}$$

$$\overline{\delta}_D(q,xa) = \delta_D(\overline{\delta}_D(q,x),a) \tag{2.3}$$

Define-se uma **linguagem** de um DFA da seguinte forma:

Definição 3 *(HOPCROFT et al., 2006) Seja o DFA $A = (Q,\Sigma,\delta,q_0,F)$, então definimos a linguagem de A, L(A), assim:*

$$L(A) = \{w|\overline{\delta}(q_0,w)\} \in F \tag{2.4}$$

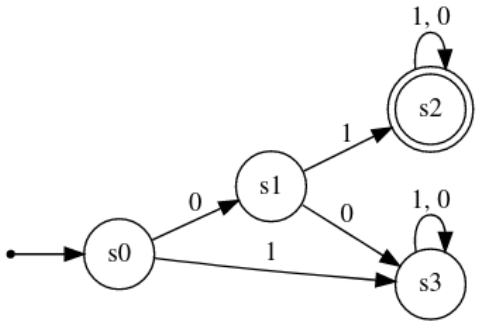
Para representar um DFA de maneira gráfica, podemos usar uma **diagrama de transições** ou uma **tabela de transições**. Um diagrama de transições é definido da forma que se segue:

- Para cada estado $q \in Q$, existe um nó q correspondente;
- Para cada estado $q \in Q$ e para cada símbolo da entrada $a \in \Sigma$ tal que $\delta(q,a) = p$, o diagrama tem um arco que sai do nó q e vai para o nó p rotulado por a;
- Existe uma seta apontando para o q_0 indicando-o como estado inicial;
- Os nós que representam os estados finais possuem um círculo duplo, diferentemente dos demais nós que possuem apenas um.

Um exemplo de DFA que tem como alfabeto $\Sigma = \{0,1\}$ e aceita palavras começadas por 01 é mostrado na Figura 4. O estado inicial "s0" tem duas possibilidades de caminhos, quando se lê 0 na entrada, que vai para o estado "s1", ou quando se lê 1 na entrada, que vai para o estado "s3". Quando estamos no estado "s1" e se lê 1, vai para o estado "s2", ou se o símbolo lido for 0, vai para o estado "s3". Quando está no estado "s2", a entrada pode acabar que será aceita, caso não acabe, qualquer símbolo lido irá levar para o mesmo estado "s2", pois o que deve ser observado é se a cadeia inicial é "01" enquanto o resto pode ser de qualquer maneira. O estado "s3" é o estado da máquina que irá rejeitar a entrada, pois quando se lê um símbolo não esperado, a transição irá levar até ele, que não é final e todas as suas transições continuam em "s3".

Dizemos que um DFA A_1 é equivalente ao DFA A_2 se $L(A_1) = L(A_2)$, ou seja, se as palavras reconhecidas pelos dois autômatos são as mesmas. Uma importante parte da teoria dos autômatos é que a partir de um DFA qualquer podemos encontrar um DFA com o número mínimo de estados.

Figura 4 – Exemplo de DFA



Fonte: Próprio autor (2023)

Teorema 1 *(HOPCROFT et al., 2006) Todo DFA A tem um autômato equivalente DFA A^{min} que tem o menor número de estados possíveis.*

O teorema se dá a partir do seguinte definição e do seguinte teorema:

Definição 4 *(HOPCROFT et al., 2006) Dois estados p e q são equivalentes se, para cada entrada w, $\overline{\delta}(p,w)$ é um estado de aceitação se, e somente se, $\overline{\delta}(q,w)$ é um estado de aceitação.*

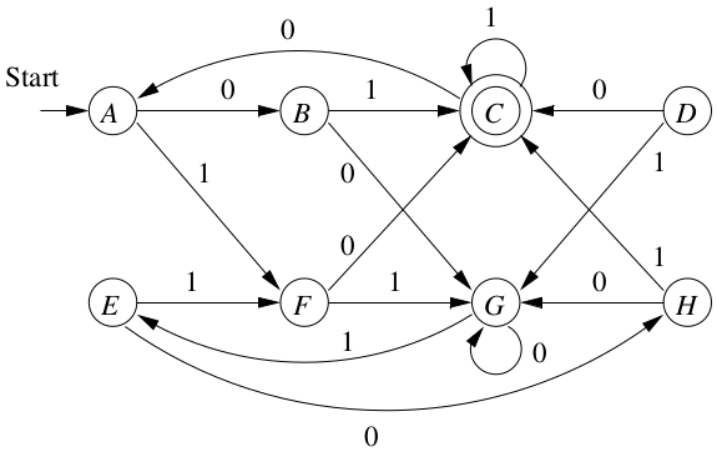
Teorema 2 *(HOPCROFT et al., 2006) Se criarmos para cada estado q do DFA um bloco consistindo em q e todos os seus estados equivalentes, então cada bloco será uma partição do conjunto de estados. Todos os membros do bloco são equivalentes e não há par de estados de diferentes blocos que sejam equivalentes entre si.*

De acordo com Hopcroft *et al.* (2006), para encontrar estados equivalentes é preciso verificar para cada par se eles são distinguíveis. Para um melhor entendimento, é apresentado um exemplo de autômato e sua minimização na Figura 5.

Inicialmente, podemos dizer que alguns pares de estados claramente não são equivalentes, como por exemplo, os estados C e H, pois C é um estado de aceitação e H não é, ou seja, $\overline{\delta}(C,\varepsilon)$ é um estado de aceitação enquanto que $\overline{\delta}(H,\varepsilon)$ não. Agora, analisamos os estados E e G e vemos que a entrada 1 não os distinguem, pois os levam para os estados F e E que não são estados de aceitação. Da mesma forma, a entrada 0 não os distinguem, pois os levam para os estados G e H que não são estados de aceitação. Todavia, a entrada 10 os distinguem pois $\overline{\delta}(E,10) = C$, que é estado de aceitação enquanto que $\overline{\delta}(G,10) = H$, que não é estado de aceitação.

Agora consideremos os estados A e E. A entrada 1 não os distinguem pois os levam para o estado F, de forma que toda entrada iniciada por 1 também não os distinguirá, do mesmo

Figura 5 – Exemplo de DFA com estados equivalentes

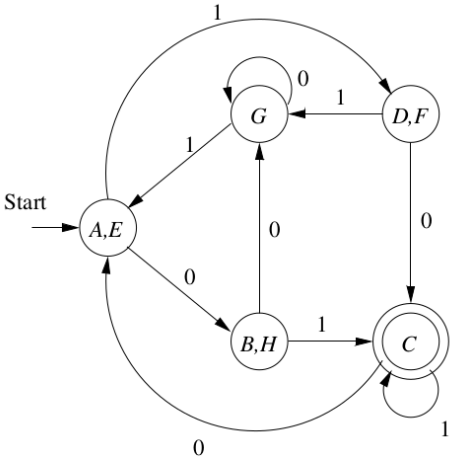


Fonte: (HOPCROFT *et al.*, 2006)

modo que a entrada 0 também não os distinguem pois os levam para H e B, que não são estados de aceitação. A entrada 00 não os distinguem pois $\overline{\delta}(A,00) = G$ e $\overline{\delta}(E,00) = G$, de forma que toda entrada iniciada por 00 também não os distinguirá. A entrada 01 também não os distinguem pois $\overline{\delta}(A,01) = C$ que é um estado de aceitação e $\overline{\delta}(E,01) = C$ também.

Ao realizar esse processo para todos os pares de estados, e juntando os estados equivalentes em blocos, conseguimos o autômato minimizado da Figura 6.

Figura 6 – Exemplo de DFA minimizado



Fonte: (HOPCROFT *et al.*, 2006)

Autômatos finitos não-determinísticos podem ser vistos como uma espécie de autômatos onde é realizada uma computação paralela onde múltiplos processos são realizados concorrentemente. O autômato finito não-determinístico se divide para seguir a diversas escolhas de caminhos que ocorrerão paralelamente, logo é gerada uma árvore de computação onde, se pelo menos um caminho chegar em um estado final após ler todos os símbolos de entrada, então

toda palavra será aceita.

De acordo com Hopcroft *et al.* (2006), um NFA difere de um DFA pelo fato de sua função de transição δ receber um estado e um símbolo de entrada como argumentos e retornar um conjunto de estados.

Segundo Hopcroft *et al.* (2006), um NFA pode ser definido da maneira que se segue:

$$M = (Q, \Sigma, \delta, q_0, F) \tag{2.5}$$

onde:

- Q é o conjunto dos estados;
- Σ é o alfabeto de entrada;
- δ é a função de transição onde: $Q \times \Sigma \rightarrow 2^Q$;
- q_0 é o estado inicial;
- F é o conjunto de estados finais.

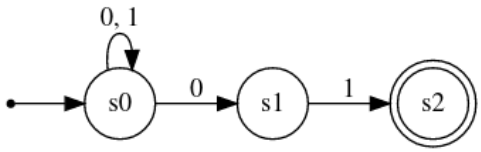
A linguagem de um NFA é definida da seguinte forma:

Definição 5 *(HOPCROFT et al., 2006) Seja $A = (Q, \Sigma, \delta, q_0, F)$ um NFA, então:*

$$L(A) = \{w | \overline{\delta}(q_0, w)\} \cap F \neq \emptyset\} \tag{2.6}$$

A Figura 7 nos mostra um exemplo de NFA que aceita palavras que terminam em 01.

Figura 7 – Exemplo de NFA



Fonte: Próprio autor (2023)

Uma variação do NFA permite que sejam realizadas transições sobre o caractere vazio, representado pela letra ε. Um autômato finito com transições vazias, ou ε-NFA, é definido da forma $A_\varepsilon = (Q, \Sigma, \delta_\varepsilon, q_0, F)$, onde:

- Q é o conjunto de estados;
- Σ é o alfabeto de entrada;
- q_0 é o estado inicial;

- F é o conjunto de estados finais;
- $\delta_\varepsilon : Q \times \Sigma \cup \{\varepsilon\} \rightarrow 2^Q$.

Para entendermos como é feita a computação de um ε-NFA é necessário nos atentarmos para o conceito de ε-fechamento. O ε-fechamento de um nó q, representado por $S_\varepsilon(q)$ diz que:

- $q \in S_\varepsilon(q)$;
- Se $p \in S_\varepsilon(q)$ e existe uma transição vazia a partir de p, então $\delta(p,\varepsilon) \subseteq S_\varepsilon(q)$, ou seja, se p é alcançável a partir de q, lendo ε, então todos os estados alcançados a partir de p, lendo ε, também estão no ε-fechamento de q.

Assim, para definir uma computação, temos que a função de transição estendida, representada por $\overline{\delta}_\varepsilon$, se dá da forma: $\overline{\delta}_\varepsilon : Q \times \Sigma^* \rightarrow 2^Q$. Formalmente, temos:

-

$$\overline{\delta}_\varepsilon(q,\varepsilon) = S_\varepsilon(q) \tag{2.7}$$

-

$$\overline{\delta}_\varepsilon(q,xa) = \bigcup_{r \in R} S_\varepsilon(r) \text{ onde } R = \bigcup_{p \in \overline{\delta}_\varepsilon(p,x)} \delta_\varepsilon(p,a) \tag{2.8}$$

A linguagem de um ε-NFA é definida da seguinte forma:

Definição 6 *(HOPCROFT et al., 2006) Seja um ε-NFA $E = (Q,\Sigma,\delta,q_0,F)$ então:*

$$L(E) = \{\overline{\delta}_\varepsilon(q_0,w) \cap F \neq \emptyset\} \tag{2.9}$$

DFAs e NFAs reconhecem a mesma classe de linguagens, uma vez que existe uma equivalência entre os dois tipos de autômatos. Sipser (1996) diz que duas máquinas são equivalentes se elas reconhecem a mesma linguagem. Tal equivalência se mostra útil pois, dependendo do caso, é mais fácil construir um NFA do que um DFA.

Teorema 3 *(SIPSER, 1996) Todo autômato finito não-determinístico tem um autômato finito determinístico equivalente.*

Prova: Seja $N = (Q,\Sigma,\delta,q_0,F)$ um NFA que reconhece uma linguagem L(N). Construímos um DFA $M = (Q',\Sigma,\delta',q_0',F')$ da forma:

- $Q' = 2^Q$;

- $q'_0 = \{q_0\}$;
- $F' = \{R \in Q' | R$ é um subconjunto e contém algum estado de aceitação de N $\}$,
- Para cada estado $S \subseteq Q$ e para cada entrada $a \in \Sigma$, temos que:

$$\delta'(S,a) = \bigcup_{r \in R} S_\varepsilon(r) \text{ onde } R = \bigcup_{p \in S} \delta(p,a) \tag{2.10}$$

A demonstração de que o DFA M simula o processamento do NFA N se dá por indução do tamanho da palavra. Deve-se mostrar que:

$$\delta'(q'_0, w) = \langle q_1, \cdots, q_x \rangle \text{sse} \delta(q_0, w) = \{q_1, \cdots, q_x\} \tag{2.11}$$

onde w é uma palavra qualquer da linguagem Σ.

1. *Base da indução:* Seja w tal que |w| = 0. Então:

   $$\delta'(q'_0, \varepsilon) = \langle q_0 \rangle \text{sse} \delta(q_0, \varepsilon) = \{q_0\} \tag{2.12}$$

   o que é verdade pela função de transição estendida.

   Hipótese de indução: Seja w, tal que $|w| = n$ e $n \geq 1$. Suponha que seja verdade:

   $$\delta'(q'_0, w) = \langle q_1, \cdots, q_x \rangle \text{sse} \delta(q_0, w) = \{q_1, \cdots, q_x\} \tag{2.13}$$

2. *Passo indutivo:* Seja w, tal que $|wa| = n+1$ e $n \geq 1$.

   $$\delta'(q'_0, wa) = \langle p_1, \cdots, p_v \rangle \text{sse} \delta(q_0, wa) = \{p_1, \cdots, p_v\} \tag{2.14}$$

   o que equivale, por hipótese de indução, a:

   $$\delta'(\langle q_1, \cdots, q_x \rangle, a) = \langle p_1, \cdots, p_v \rangle \text{sse} \delta(\{q_1, \cdots, q_x\}, a) = \{p_1, \cdots, p_v\} \tag{2.15}$$

   o que é verdadeiro, por definição de δ'.

Logo M simula N para qualquer entrada $w \in \Sigma^*$. **Fim da prova.**

2.1.1 Composição de autômatos

Podemos criar novos autômatos a partir de outros autômatos utilizando as **propriedades de fechamento** sobre as operações regulares: união, interseção e complemento. Tais propriedades também servem para verificar se uma linguagem L qualquer é regular.

Definição 7 *(SIPSER, 1996) Uma linguagem é chamada de uma linguagem regular se algum autômato finito a reconhece.*

Teorema 4 *(SIPSER, 1996) Se L_1 e L_2 são linguagens regulares, então $L_1 \cup L_2$ também o é.*

Prova: Suponha uma máquina M_1 que reconhece a linguagem L_1, onde $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$. Suponha também uma outra máquina M_2 que reconhece a linguagem L_2, onde $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Construímos a máquina $M = (Q, \Sigma, \delta, q_0, F)$ que reconhecerá $L_1 \cup L_2$ da forma:

- $Q = \{(r_1, r_2)$, onde $r_1 \in Q_1$ e $r_2 \in Q_2\}$ ou seja, o produto cartesiano dos conjuntos Q_1 e Q_2;
- Σ é o mesmo de M_1 e M_2;
- δ é a função de transição definida da forma:

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)) \text{onde} a \in \Sigma \tag{2.16}$$

- q_0 é o par (q_1, q_2);
- F é conjunto dos pares onde pelo menos um dos membros é um estado de aceitação de M_1 ou de M_2.

Seja w uma palavra de $L_1 \cup L_2$. É possível assumir que $w \in L_1$.

Seja $w = x_0, x_1, x_2, \cdots, x_m$ onde $m \geq 0$. Como M_1 aceita w, existem $s_0, s_1, s_2, \cdots, s_m \in Q_1$ de forma que:

$$\overline{\delta}(q_1, \varepsilon) = s_0 \tag{2.17}$$

$$\overline{\delta}(s_0, x_1) = s_1 \tag{2.18}$$

$$\overline{\delta}(s_1, x_2) = s_2 \tag{2.19}$$

...

$$\overline{\delta}(s_{m-1}, x_m) = s_m \tag{2.20}$$

onde $s_m \in F_1$.

Assim, como $\forall q \in Q_1$ e $\forall x \in \Sigma$, $\delta_1(q, x) = \delta(q, x)$, então $\forall s \in \overline{\delta}_1(q_1, x)$ significa que $\forall s \in \overline{\delta}(q_1, x)$.

Desta forma, podemos substituir δ_1 por δ. O caminho acima fica desta forma:

$$\overline{\delta}(q_1,\varepsilon) = s_0 \tag{2.21}$$

$$\overline{\delta}(s_0,x_1) = s_1 \tag{2.22}$$

$$\overline{\delta}(s_1,x_2) = s_2 \tag{2.23}$$

$$\cdots$$

$$\overline{\delta}(s_{m-1},x_m) = s_m \tag{2.24}$$

onde $s_m \in F_1 \cup F_2$ e $s_0, s_1, \cdots, s_m \in Q$. Além disso:

$$\delta(q_0,\varepsilon) = \{q_1,q_2\} \text{de forma que} q_1 \in \overline{\delta}(q_0,\varepsilon) \tag{2.25}$$

logo

$$\overline{\delta}(q_0,\varepsilon) = q_1, \tag{2.26}$$

$$\overline{\delta}(q_1,\varepsilon) = s_0, \tag{2.27}$$

então, $\overline{\delta}(q_0,\varepsilon) = s_0$ (2.28)

Portanto, o caminho acima também pode ser escrito desta forma:

$$\overline{\delta}(q_0,\varepsilon) = s_0 \tag{2.29}$$

$$\overline{\delta}(s_0, x_1) = s_1 \tag{2.30}$$

$$\overline{\delta}(s_1, x_2) = s_2 \tag{2.31}$$

...

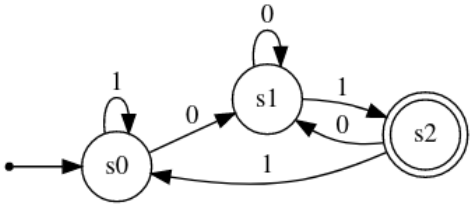
$$\overline{\delta}(s_{m-1}, x_m) = s_m \tag{2.32}$$

onde $s_m \in F_1 \cup F_2$ e $s_0, s_1, \cdots, s_m \in Q$.

Assim, M aceita w. **Fim da prova.**

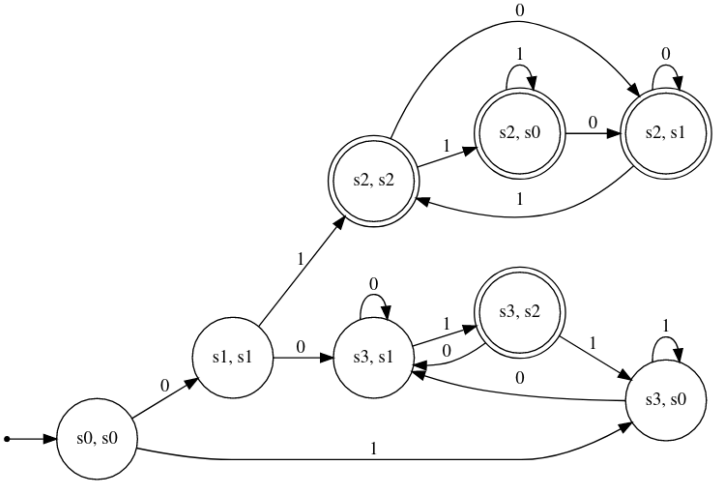
A Figura 9 mostra o resultado da união do autômato da Figura 4 com o autômato da Figura 8. O resultado, é o autômato para a linguagem que inicia ou termina com 01.

Figura 8 – Autômato que lê entradas que terminam com 01



Fonte: Próprio autor (2023)

Figura 9 – União de autômatos



Fonte: Próprio autor (2023)

Partindo deste teorema, podemos definir:

Teorema 5 *Se L_1 e L_2 são linguagens regulares, então $L_1 \cap L_2$ também o é.*

Prova: Suponha uma máquina M_1 que reconhece a linguagem L_1, onde $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$. Suponha também uma outra máquina M_2 que reconhece a linguagem L_2, onde $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Construímos a máquina $M = (Q, \Sigma, \delta, q_0, F)$ que reconhecerá $L_1 \cap L_2$ da forma:

- $Q = \{(r_1, r_2)$, onde $r_1 \in Q_1$ e $r_2 \in Q_2\}$ ou seja, o produto cartesiano dos conjuntos Q_1 e Q_2;
- Σ é o mesmo de M_1 e M_2;
- δ é a função de transição definida da forma:

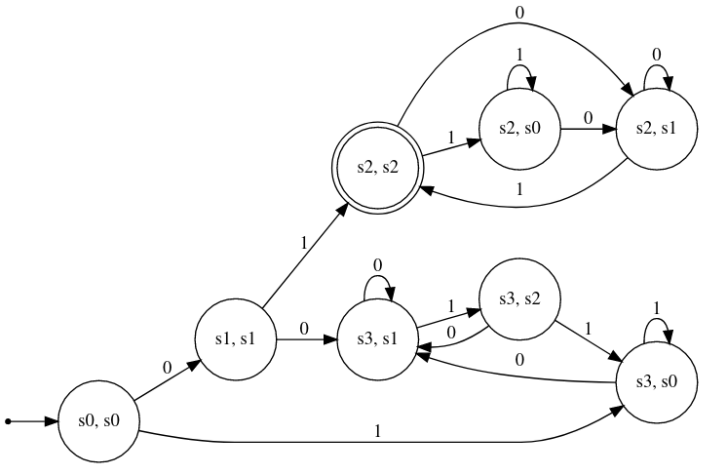
$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)) \text{ onde } a \in \Sigma \tag{2.33}$$

- q_0 é o par (q_1, q_2);
- F é conjunto dos pares onde os dois membros são estados de aceitação de M_1 e de M_2.

Fim da prova.

Na Figura 10, temos um exemplo de autômato resultado da interseção dos autômatos mostrados nas Figuras 4 e 8. Neste caso, o autômato só irá aceitar palavras que iniciam e terminam com 01.

Figura 10 – Interseção de autômatos



Fonte: Próprio autor (2023)

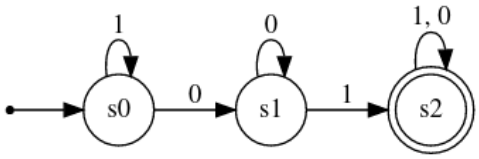
Já o complemento de uma linguagem L_1 será a linguagem que aceita tudo o que L_1 não aceita e não aceita tudo o que L_1 aceita.

Teorema 6 *(SIPSER, 1996) Dada uma linguagem regular L_1, então a linguagem complemento de L_1, $\overline{L_1}$, também será regular.*

Prova: Seja $M_1 = (Q, \Sigma, \delta, q_0, F)$ um autômato que reconheça a linguagem L_1. A linguagem complemento $\overline{L_1}$ será reconhecida pelo autômato $\overline{M_1} = (Q, \Sigma, \delta, q_0, Q - F)$ invertendo os estados de aceitação e não aceitação de M_1. **Fim da prova.**

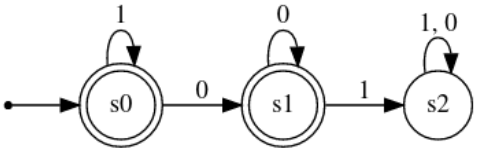
Na Figura 11, temos um exemplo de autômato para a linguagem que aceita palavras que contém 01 enquanto que na Figura 12, temos um autômato para a linguagem complemento, ou seja, que não contém 01.

Figura 11 – Exemplo de autômato que lê entradas que contenham 01



Fonte: Próprio autor (2023)

Figura 12 – Complemento do autômato que lê entradas que contenham 01



Fonte: Próprio autor (2023)

Agora a propriedade de concatenação é apresentada:

Teorema 7 *(SIPSER, 1996) Dada duas linguagens regulares L_1 e L_2, então a concatenação entre as duas linguagens, L, também será regular.*

Prova: Sejam $M_1 = (Q_1, \Sigma_1, \delta_1, q0_1, F_1)$ e $M_2 = (Q_2, \Sigma_2, \delta_2, q0_2, F_2)$ dois NFAs que reconhecem as linguagens $L(M_1)$ e $L(M_2)$ respectivamente. Suponha um autômato $M = (Q, \Sigma, \delta, q0, F)$ onde:

- Q = $Q_1 \cup Q_2$;
- $q0 = q0_1$;
- $F = F_2$;
- Define-se δ, para qualquer $q \in Q$ e qualquer $a \in \Sigma$ desta forma:

$$\delta(q,a) = \delta_1(q,a) \text{ se } q \in Q_1 \text{ e } q \notin F_1; \tag{2.34}$$

$$\delta(q,a) = \delta_2(q,a) \text{ se } q \in F_1 \text{ e } a \neq \varepsilon; \tag{2.35}$$

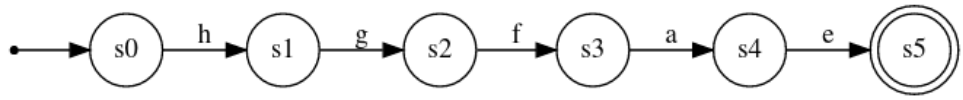
$$\delta(q,a) = \delta_1(q,a) \cup q0_2 \text{ se } q \in F_1 \text{ e } a = \varepsilon; \tag{2.36}$$

$$\delta(q,a) = \delta_2(q,a) \text{ se } q \in Q_2. \tag{2.37}$$

Fim da prova.

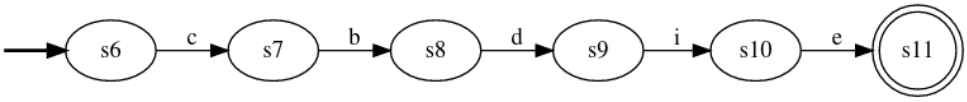
Nas Figuras 13 e 14 temos exemplos de autômatos que lêem as cadeias “hgfae” e “cbdie”, respectivamente, e na Figura 15 temos a concatenação entre os dois, ou seja, um autômato que aceita quando a entrada pode ser dividida em duas partes, a primeira aceita por M_1 e a segunda por M_2.

Figura 13 – Autômato M1



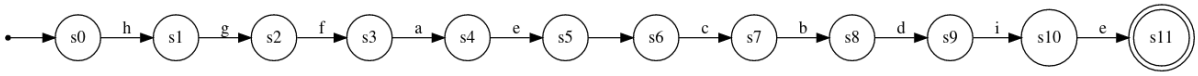
Fonte: Próprio autor (2023)

Figura 14 – Autômato M2



Fonte: Próprio autor (2023)

Figura 15 – Autômato M



Fonte: Próprio autor (2023)

Agora veremos a propriedade operação estrela. Seja uma linguagem L(N), a operação estrela irá juntar um número qualquer, incluindo 0, de cadeias de L(N) de forma que será obtida uma nova linguagem:

Teorema 8 *(SIPSER, 1996) Dada uma linguagem regular L, então a operação estrela, L*, também será regular.*

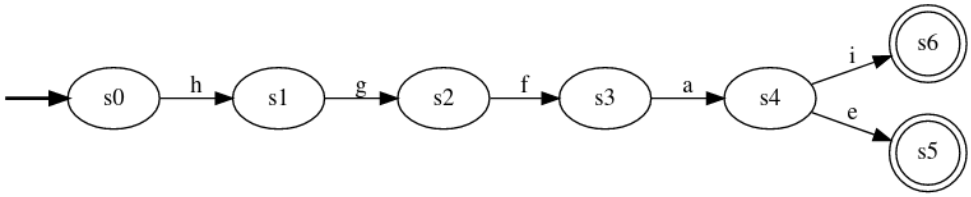
Prova: Seja uma máquina $M_1 = (Q_1, \Sigma, \delta_1, q0_1, F_1)$ que reconhece L. Então, para reconhecer L^*, construímos uma máquina $M = (Q, \Sigma, \delta, q0, F)$ tal que:

- $Q = \{q0\} \cup Q_1$;
- $F = \{q0\} \cup F_1$;
- para quaisquer $q \in Q$ e $a \in \Sigma$, $\delta(q,a)$ é definido desta forma:
  - $\delta_1(q,a)$ se q está em Q_1 mas não é estado final;
  - $\delta_1(q,a)$ se q é estado final e a não é vazio;
  - $\delta_1(q,a) \cup \{q0_1\}$ se q é estado final e a é vazio;
  - $\{q0_1\}$ se $q = q_0$ e a é vazio;
  - $\emptyset$ se $q = q0$ e a não for vazio.

Fim da prova.

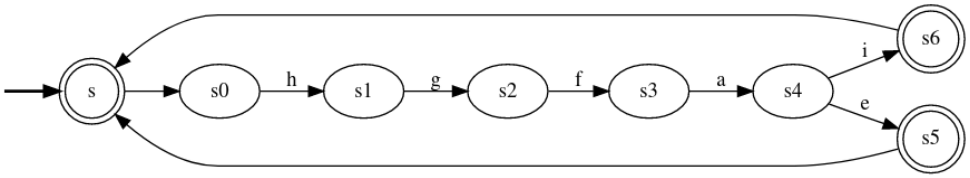
Na Figura 16 é mostrado um exemplo de autômatos que reconhece as cadeias “hgfae” e “hgfai”. Na Figura 17 temos a operação estrela do autômato, ou seja, um autômato que aceita quando a entrada pode ser dividida em cadeias da entrada sendo que cada cadeia é aceita pelo autômato da Figura 16.

Figura 16 – Autômato N



Fonte: Próprio autor (2023)

Figura 17 – Autômato N^*



Fonte: Próprio autor (2023)

2.2 Gestão de Processos de Negócios

Segundo Gomes (2016), os processos, dentro do contexto computacional, têm por característica a presença de uma entrada, um processamento e uma saída, onde a entrada são os dados e informações necessárias para que o processamento seja computado, o processamento são

ações que irão ocorrer sobre esses dados, gerando assim uma saída, e a saída é uma informação obtida dos dados processuais.

Ao conjunto de técnicas, métodos e ferramentas computacionais utilizadas para dar suporte às diferentes fases do ciclo da vida de um processo de negócio, chamamos de gestão de processos de negócios (*Business Process Manager* ou BPM) (AALST, 2016).

Segundo Paim *et al.* (2009), os processos de negócios são usados principalmente para diminuir a perda de tempo na identificação de um problema. Logo, para que o tempo seja realmente reduzido, as ações de modelagem devem estar estruturadas de uma forma que permita o diagnóstico dos processos e identifique as soluções de uma forma fácil, fazendo com que, dessa forma, os processos sejam implementados no menor intervalo de tempo e custos.

Alguns ainda desconhecem o gerenciamento de processos de negócio, chegando a acreditar que é apenas desperdício de tempo e de esforço e pulam esta etapa. Todavia, segundo Gomes (2016), gerenciar processos de negócios é importante pois eles precisam ser projetados, documentados e desenhados tanto para facilitar possíveis aperfeiçoamentos e para facilitar a compreensão por todo o público que tem interesse no assunto.

Para realizarmos melhorias em determinado processo, é necessário que seja elaborada uma modelagem, pois ela irá auxiliar na descoberta de inconsistências, trazendo benefícios para todos os envolvidos.

2.2.1 Modelos para BPM

Para representar os modelos de processos de negócios podemos utilizar diversas modelagens. A seguir, apresentamos algumas formas de representação.

2.2.1.1 Sistemas de transições

Segundo Aalst (2016), sistemas de transições são as formas mais básicas de modelar processos de negócios por serem compostos apenas de **estados** e de **transições**.

Definimos um sistema de transições como uma tripla ST = (S, A, T), onde S é o conjunto de estados, A é o conjunto de ações e T é o conjunto de transições que irão ligar dois estados através das ações. Também definimos dois conjuntos que estão contidos no conjunto S, são eles: S^{inicio} e S^{final} que representam os *estados iniciais* e *estados finais*, também referidos como *estados de aceitação*, respectivamente. De acordo com Aalst (2016), na teoria, o conjunto S pode ser um conjunto infinito, mas na prática o espaço dos estados é finito, sendo assim,

os sistemas de transições podem ser chamados de Máquinas de Estados Finitas (*Finite State Machines* ou FSM) ou de autômatos de estados finitos.

Em um sistema de transição qualquer caminho que se inicie em um estado do conjunto S^{inicio} corresponde a uma possível *sequência de execução*, em um mesmo sistema pode haver infinitas sequências de execução. Dizemos que um caminho termina com sucesso se ele termina em um dos estados do conjunto S^{final}. Quando o caminho chega em um estado que não está no conjunto S^{final} diz-se que não é aceito.

Segundo Aalst (2016), qualquer modelo de processo com semântica executável pode ser mapeado em um sistema de transição, assim, muitas das noções definidas para sistemas de transições podem ser traduzidas para outros tipos de modelagem como BPMN (GOMES, 2016) ou redes de Petri (DONGEN *et al.*, 2009), desde que os modelos sejam expressos em uma linguagem confiável. Por exemplo, a noção de equivalência de *traces* considera dois sistemas equivalentes quando suas sequências de execução são as mesmas e pode ser aplicada para diferentes modelagens.

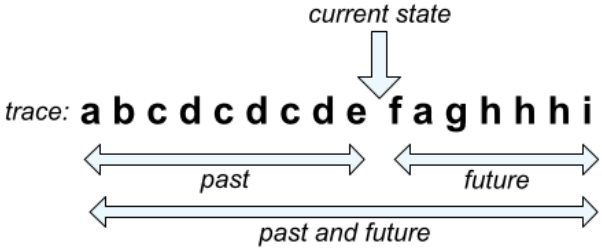
Quando se representa *log* de eventos por meio de sistemas de transições, é necessário definir o que é um estado. Para resolver esse problema podemos escolher três diferentes abordagens que poderão gerar diferentes resultados dependendo da escolha. Na primeira abordagem, o estado atual é determinado pelo passado do *trace*, ou seja, as atividades que já aconteceram; na segunda abordagem, o estado atual é determinado pelas atividades futuras, ou seja, dependendo do caminho escolhido, quais atividades poderão acontecer ainda; a terceira abordagem é uma união de ambas.

As atividades passadas de um caso são chamadas de prefixo do *trace* completo e as atividades futuras de um caso são chamados de sufixo de um *trace completo*. No momento da construção do sistema de transição pode ser escolhido a abordagem que considera o **prefixo/sufixo completo**, a que considera o **prefixo/sufixo parcial**. Na Figura 18, vemos como é construída a noção de passado e futuro de um *trace*. Segundo Aalst (2016), nesse caso, vemos que o *trace* $\sigma =< a,b,c,d,c,d,c,d,e,f,a,g,h,h,h,i >$, teria o seu passado representado pelo *trace* parcial $\sigma_1 =< a,b,c,d,c,d,c,d,e >$ e o futuro pelo *trace* parcial $\sigma_2 =< f,a,g,h,h,h,i >$.

Dada uma sequência σ e um número k, definimos uma função de representação de estado $l^{estado}()$ onde k indica o número de eventos que ocorrem em σ para produzir o estado. De acordo com Aalst (2016), essa função pode ter quatro formas, são elas:

- $l_1^{estado}(\sigma,k) = hd^k(\sigma) =< a_1,a_2,\ldots,a_k >$ onde $hd^k(\sigma)$ é uma função que retorna uma

Figura 18 – Exemplo de passado e futuro de um estado



Fonte: (AALST, 2016)

sequência com os k primeiros elementos da sequência σ. Por exemplo, seja $\sigma' = \langle a,b,c,d,c,d,e\rangle \in L$ um *trace* de um *log* de eventos, $l_1^{estado}(\sigma',3) = \langle a,b,c\rangle$;

- $l_2^{estado}(\sigma,k) = tl^{n-k}(\sigma) =< a_{k+1}, a_{k+2}, \ldots, a_n >$ onde $tl^{n-k}(\sigma)$ é uma função que retorna uma sequência com os últimos elementos a partir do k-ésimo elemento na sequência σ. Por exemplo, seja $\sigma' = \langle a,b,c,d,c,d,e\rangle \in L$ um *trace* de um *log* de eventos, $l_2^{estado}(\sigma',3) = \langle d,c,d,e\rangle$;
- $l_3^{estado}(\sigma,k) = \partial_{multiconjunto}(hd^k(\sigma)) = [a_1,a_2,\ldots,a_k]$ que converte o passado completo em um multiconjunto, ou seja, não importa a ordem dos eventos, apenas a sua frequência. Por exemplo, seja $\sigma' = \langle a,b,c,d,c,d,e\rangle \in L$ um *trace* de um *log* de eventos, $l_3^{estado}(\sigma',5) = [a^1,b^1,c^2,d^1]$;
- $l_4^{estado}(\sigma,k) = \partial_{conjunto}(hd^k(\sigma)) = \{a_1,a_2,\ldots,a_k\}$ que faz um conjunto representando o passado completo. Nele, o que importa são as atividades que são executadas pelo menos uma vez não importando a ordem dos eventos nem a sua frequência. Por exemplo, seja $\sigma' = \langle a,b,c,d,c,d,e\rangle \in L$ um *trace* de um *log* de eventos, $l_4^{estado}(\sigma',5) = \{a,b,c,d\}$;

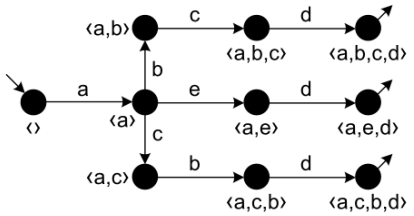
Assim, podemos definir um sistema de transições baseado em um log de eventos dessa forma:

Definição 8 *(AALST, 2016) Seja L um log de eventos e* $l^{estado}()$ *uma função de representação de estado.* $ST_{L,l^{estado}()} = (S,A,T)$ *onde:*

- $S = \{l^{estado}(\sigma,k) | \sigma \in L \; e \; 0 \leq k \leq |\sigma|\}$ *é o conjunto de estados;*
- $A = \{\sigma(k) | \sigma \in L \; e \; 1 \leq k \leq |\sigma|\}$ *é o conjunto de atividades;*
- $T = \{(l^{estado}(\sigma,k), \sigma(k+1), l^{estado}(\sigma,k+1)) | \sigma \in L \; e \; 0 \leq k \leq |\sigma|\}$ *é o conjunto de transições;*
- $S^{inicio} = \{l^{estado}(\sigma,0) | \sigma \in L\}$ *é o conjunto de estados iniciais e* $S^{final} = \{l^{estado}(\sigma,|\sigma|) | \sigma \in L\}$ *é o conjunto de estados finais.*

Por exemplo, a Figura 19 nos mostra o sistema de transições gerado utilizando a função $l_1^{estado}(\sigma,4)$ no *log* $L = \langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^2, \langle a,e,d\rangle$.

Figura 19 – Exemplo de sistema de transições



Fonte: (AALST, 2016)

Mas, embora os sistemas de transições sejam simples, é problemática a forma de expressar concorrências de forma sucinta. Por exemplo, suponha que haja n atividades para serem executadas em paralelo, ou seja, todas as atividades precisam ser executadas mas qualquer ordem de execução é válida, logo, teríamos n! possíveis sequências de execução. Para representar tudo isso em forma de autômatos finitos seriam necessários 2^n estados e $n \times 2^{n-1}$ transições. Esse problema é denominado problema da *explosão de estados*. Tomemos como exemplo um *log* de eventos real, disponibilizado pela SEFAZ, que contém 258 processos (*traces*), onde cada processo, em média, dura 170 dias e tem mais de 27 eventos.

Quando é gerado um NFA para representá-lo, temos um resultado ininteligível por ser muito grande e ter muitas transições. Após a determinização e a minimização do autômato, temos uma redução considerável no número de estados e transições. Tal redução pode ser melhor compreendida analisando a imagem da Tabela 1, onde são apresentados os dados a respeito das representações mostradas. É possível perceber que o número de estados da máquina não-determinística é maior que o da máquina determinística que, por sua vez, tem quase o dobro de estados da máquina determinística mínima. O mesmo acontece para o número de transições. Em relação ao NFA, vê-se que a máquina determinística tem $30,6\%$ de estados a menos, enquanto que a determinística mínima tem $63,04\%$ de estados a menos.

Tabela 1 – Tabela Comparativa das Máquinas

Máquina de estados	Estados	Transições	% de estados a menos do que o NFA	% de transições a menos do que o NFA
Não-Determinística	7071	6813	-	-
Determinística	4909	4908	30,6	27,07
Determinística mínima	2550	2799	63,04	58,02

Fonte: Próprio autor (2021)

Por esse motivo e por causa da natureza concorrente dos processos de negócios, o uso de modelagens mais expressivas, como redes de Petri, por exemplo, se torna necessário para

representar os processos de maneira mais adequada.

2.2.1.2 Redes de Petri

Redes de Petri (ou *Petri nets*) (AALST, 2016) são estruturas utilizadas para diversos fins, mais recentemente, elas tem tido um papel fundamental no BPM e em campos de estudos relacionados, como na mineração de processos.

De acordo com Aalst (2016), uma rede de Petri é um grafo bipartido composto de lugares, ou *places* (representados por círculos) e transições (representados por quadrados). A sua estrutura é estática, embora os *tokens* (representados por pontos pretos) possam fluir por toda a rede. A distribuição de *tokens* pelos lugares determina o estado de uma rede de Petri e é referido como sua marcação. As transições são responsáveis por movimentar os *tokens* de uma marcação para outra

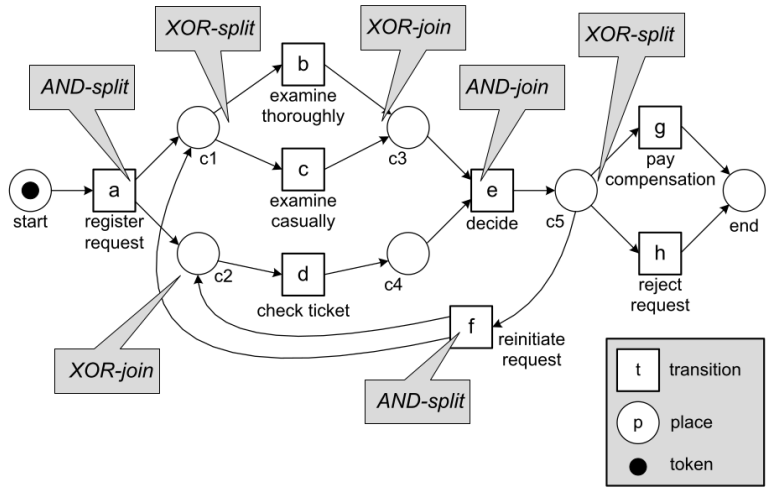
Definição 9 *(AALST, 2016) Uma rede de Petri é uma tripla N = (P, T, F), onde P é um conjunto finito de lugares, T é um conjunto finito de transições e F é o conjunto de setas direcionados chamados de relação de fluxo; a interseção entre P e T é vazia; F está contido na união do produto de P e T com o produto de T e P.*

A Figura 20 nos mostra um exemplo de rede de Petri onde $P = \{start, c1, c2, c3, c4, c5, end\}$, $T = \{a, b, c, d, e, f, g, h\}$, e $F = \{(start, a), (a, c1), (a, c2), (c1, b), (c1, c), (c2, d), (b, c3), (c, c3), (d, c4), (c3, e), (c4, e), (e, c5), (c5, f), (f, c1), (f, c2), (c5, g), (c5, h), (g, end), (h, end)\}$. Interessante notar que ela também mostra as construções da rede de Petri.

Quando uma rede de Petri tem um estado inicial e um estado final, é chamada de **rede de Petri de aceitação**. De acordo com (AALST, 2019), dizemos que uma transição é ativada quando em todos os lugares de entrada há um token, ela pode disparar consumindo um token de cada lugar de entrada e produzindo um token em cada lugar de saída. O comportamento de uma rede de Petri de aceitação é descrito por todos os seus *traces* da sua marcação inicial até a sua marcação final. A marcação do modelo da Figura 20 é $\{start\}$ o que habilita a transição a que, ao ser disparada, consome um *token* e produz dois *tokens*, um para c1 e outro para c2, como é visto na Figura 21.

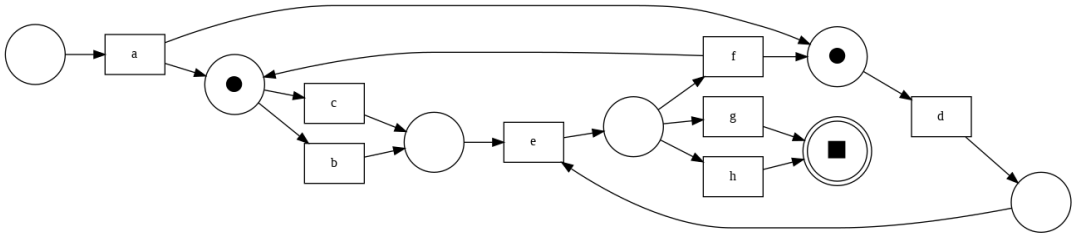
Agora a marcação do modelo está $\{c1, c2\}$, habilitando as transições b, c e d. Em c1, há dois caminhos possíveis: um que lê b e outro que lê c, assim, apenas um *token* será consumido e apenas um *token* será gerado de forma que as duas atividades não ocorrem em paralelo. A

Figura 20 – Exemplo de rede de Petri e seus componentes



Fonte: (AALST, 2016)

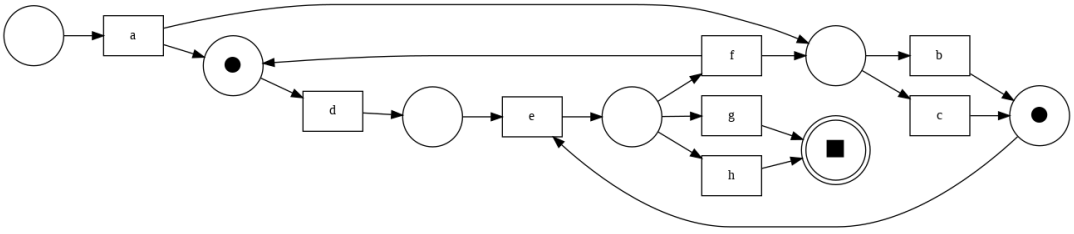
Figura 21 – Exemplo de rede de Petri



Fonte: Próprio autor (2023)

Figura 22 nos mostra o modelo após disparar a transição c. Nota-se que a transição a não está mais habilitada.

Figura 22 – Exemplo de rede de Petri



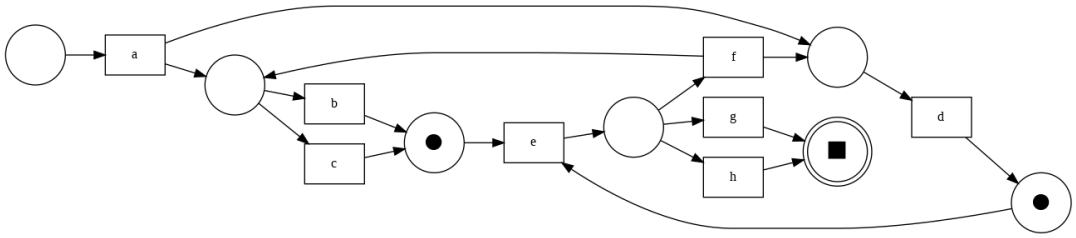
Fonte: Próprio autor (2023)

Agora a marcação do modelo está $\{c2, c3\}$ e a transição habilitada é d, pois e só pode ser disparada se houver um *token* em c3 e um *token* em c4. Ao disparar a transição d, temos a configuração mostrada na Figura 23.

Desta forma, temos a seguinte marcação: $\{c3, c4\}$, habilitando a transição e. Ao dispará-la, consumimos dois *tokens* e produzimos um *token* em c5, como mostra a Figura 24.

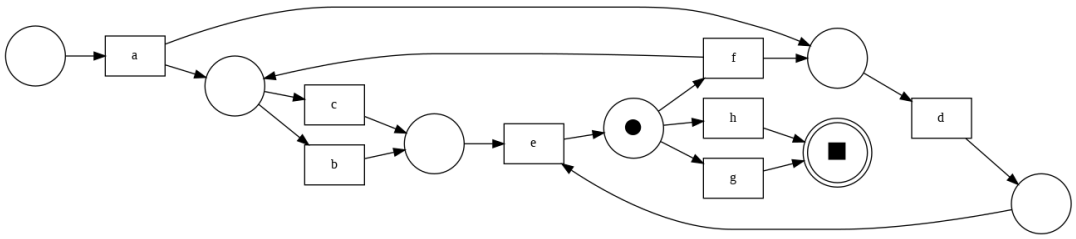
Por fim, temos a marcação $\{c5\}$ que habilita as transições f, g e h, mas assim como

Figura 23 – Exemplo de rede de Petri



Fonte: Próprio autor (2023)

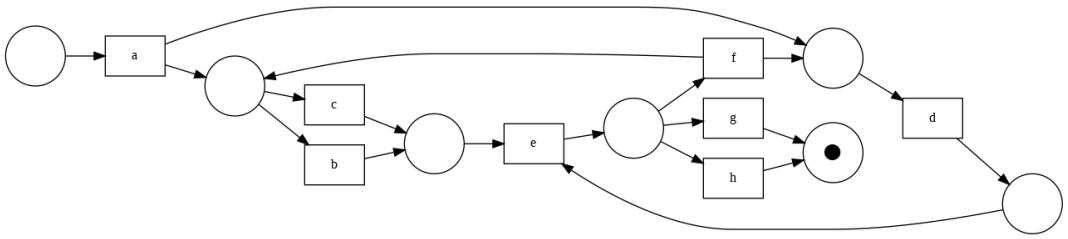
Figura 24 – Exemplo de rede de Petri



Fonte: Próprio autor (2023)

no *place* c1, apenas uma transição pode ser escolhida, consumindo um *token* e produzindo um *token*. Ao dispararmos a transição *g*, por exemplo, chegamos no *place end* que marca o fim do processo, como mostra a Figura 25.

Figura 25 – Exemplo de rede de Petri

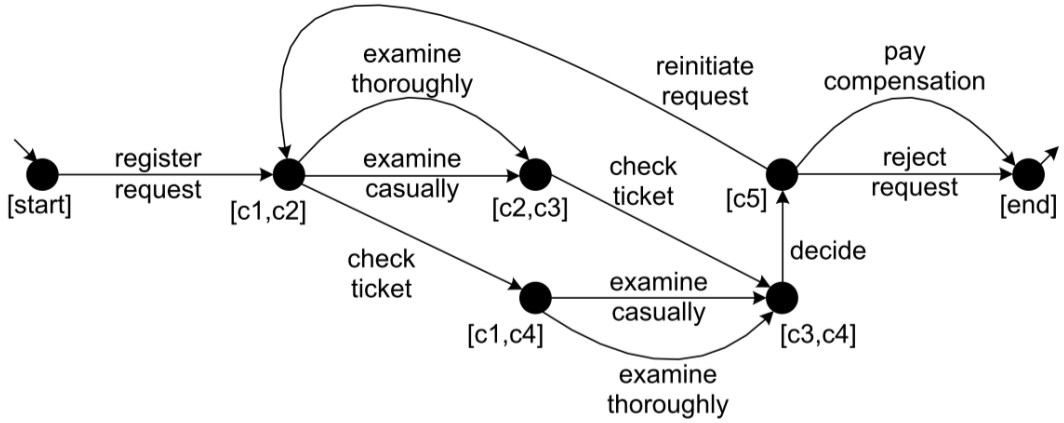


Fonte: Próprio autor (2023)

Segundo Aalst (2016), uma rede de Petri rotulada é uma tupla N = (P, T, F, A, l), onde (P, T, F) é uma rede de Petri como definida na Definição 9. A é o conjunto de rótulos de atividades, e l é a função de rotulação que irá rotular as transições.

É possível transformar uma rede de Petri rotulada em um sistema de transições onde os estados representariam as marcações alcançáveis, o estado inicial e o conjunto de estados finais do sistema de transições é a marcação inicial e as marcações finais da rede de Petri, respectivamente. Uma transição do sistema de transições pode representar tanto uma transição da rede de Petri ou uma seta. Por exemplo, para o modelo da Figura 20, conseguimos o sistema de transições da Figura 26.

Figura 26 – Exemplo de sistema de transições da rede de Petri da Figura 20



Fonte: (AALST, 2016)

2.2.1.3 Business process management notation

Outra notação utilizada para modelar processos é a *Business Process Management Notation* (BPMN) (GOMES, 2016), que é um padrão que visa oferecer uma noção gráfica para auxiliar no entendimento e gerenciamento dos processos.

Segundo White (2004), a definição de cada símbolo do diagrama em BPMN é semelhante a de outros diagramas que modelam fluxos de processos, como os tradicionais fluxogramas. Como tal notação está se referindo a um padrão que representa processos de negócios, ela pode ser adotada por empresas que exerçam atividades diversas de segmentos diversos.

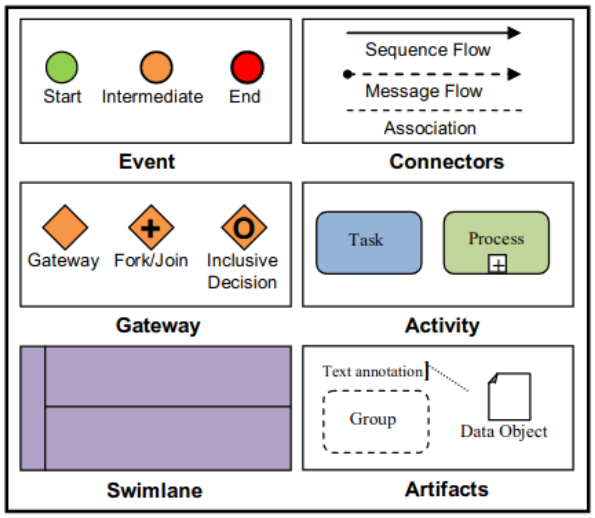
Gomes (2016) diz que os elementos que compõem um diagrama em BPMN são classificados em 5 categorias básicas: objetos de fluxo, objetos de conexão, dados, agrupamentos e artefatos. Eles podem ser encontrados na figura 27.

Os objetos de fluxo servem para definir o comportamento de um processo e são divididas em três categorias: evento, atividade e *gateway*.

Os eventos são usados para identificar ou informar algo que acontece no processo e são representados por círculos. Existem três categorias de eventos, que são os eventos de início, que servem para marcar o início de um processo e é caracterizado pela cor verde; eventos intermediários, que marcam algo que acontece durante o processo, ou seja, depois do início e antes do fim; e eventos de fim, que marcam o fim de um processo e são caracterizados pela cor vermelha. Cada categoria pode se dividir em diversas subcategorias.

Segundo Gomes (2016), as atividades identificam uma ação que pode ser executada e são representadas por verbos. Por exemplo, na frase *a coordenação assinou o documento*,

Figura 27 – Categorias de elementos BPMN



Fonte: (KHERBOUCHE *et al.*, 2013)

vemos que existe uma atividade, **assinar**, feita por um agente, **a coordenação**.

Os *gateways* são usados para identificar tanto divergências quanto convergências, ou seja, quando um fluxo se divide em dois ou mais fluxos ou quando dois ou mais fluxos convergem em um só fluxo, e são representados no formato de um losango. É muito utilizado para representar diferentes fluxo que divergem entre si, como, por exemplo, sim ou não e aprovado ou reprovado.

Os objetos de conexão servem para ligar elementos, sejam eles, atividades, *gateways* ou eventos. Sem eles o processo ficaria desordenado pois eles determinam a direção e o sentido do fluxo.

De acordo com Gomes (2016), artefatos são elementos que nos ajudam a adicionar informações importantes na modelagem do processo de negócio, como anotações, objetos de dados e agrupamentos.

Os elementos que ficam no fundo do processo são chamados de piscinas (*pool*), onde dentro dela será colocado todo o fluxo com *gateways*, atividades, entre outros. Para identificar quem executa as atividades de um processo, utiliza-se raias dentro da piscina.

A sintaxe de um processo BPMN central é definido da seguinte forma:

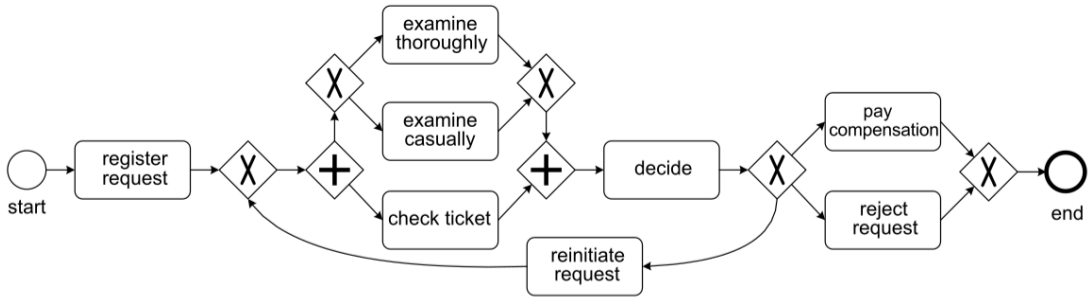
Definição 10 *(DIJKMAN et al., 2007) Um processo BPMN central é uma tupla* $P = (O, A, E, G, T, S, T^R, E^I, E^{I_M}, E^{I_T}, E^{I_R}, G^F, G^J, G^X, G^V, G^M,$ *COND, EXCP) onde:*

- *O é um conjunto de objetos que podem ser particionados em conjuntos disjuntos de atividades A, eventos E e gateways G;*
- *A pode ser particionado em conjuntos disjuntos de tarefas T e atividades de invocações de subprocessos S;*

- $T^R \subseteq T$ *é um conjunto de tarefas recebidas;*
- *E pode ser particionado em conjuntos disjuntos de eventos iniciais,* E^S*, eventos intermediários,* E^I*, e eventos finais,* E^F*;*
- E^I *pode ser particionado em conjuntos disjuntos de eventos de mensagens intermediárias* E^{I_M}*, eventos temporais intermediários* E^{I_T}*, e eventos de erro intermediários* E^{I_R}*;*
- *G pode ser particionado em conjuntos disjuntos de gateways de bifurcação paralela* G^F*, gateways de junção paralela* G^J*, gateways de decisão XOR baseado em dados* G^X*, gateways de decisão XOR baseado em eventos* G^V *e gateways de fusão XOR* G^M*;*
- $F \subseteq O \times O$ *é a relação de controle de fluxo;*
- *COND:* $F \cap (G^X \times O) \to C$ *é uma função que mapeia sequencias de fluxo emanadas de gateways XOR baseados em dados para condições;*
- *EXCP:* $E^I \nrightarrow A$ *é uma função que faz a correspondência entre um evento intermediário para uma atividade onde a ocorrência do evento sinaliza uma exceção e interrompe a performance da atividade.*

A Figura 28 nos mostra um exemplo de modelo de processos utilizando BPMN.

Figura 28 – Exemplo de modelo de processo utilizando BPMN



Fonte: (AALST, 2016)

2.2.2 Mineração de processos

Mineração de processos (*process mining*) (AALST, 2016) é o campo de pesquisa onde são analisados processos usando dados de eventos. Seu objetivo é, a partir de dados extraídos de logs de eventos, descobrir, monitorar e melhorar processos reais.

Segundo Aalst *et al.* (2006), a ideia básica da mineração de processos é aprender com as execuções observadas de um processo e:

- Descobrir novos modelos;

- Verificar a conformidade de um modelo;
- Estender um modelo existente projetando informações extraídas dos logs em algum modelo inicial.

De acordo com Aalst (2012), os *logs* de eventos são o ponto de partida para a mineração de processos. Cada evento em um *log* refere-se a uma *atividade* e é relacionado a um *caso* em particular. Os *logs* de eventos podem também apresentar informações extras tais como quem está realizando a atividade ou a data e a hora que o evento está sendo realizado. A definição formal de um *log* de eventos se dá da seguinte forma:

Definição 11 *(AALST, 2016) Seja A um conjunto finito de atividades. Um trace* $\sigma \in A^*$ *é uma sequência ordenada de atividades (*$\sigma = \langle a_1, a_2, \cdots, a_n \rangle \in A^*$*). Um log de eventos L será um multi-conjunto de traces sobre A (*$L = \langle \sigma^1, \sigma^2, \cdots, \sigma^m \rangle$*).*

Logs de eventos podem ser utilizados por três tipos de mineração de processos: *descoberta*, que é a técnica mais utilizada e serve para produzir um modelo; *conformidade*, que verifica se um modelo está de acordo com a realidade; *aprimoramento* que visa utilizar informações de processos reais em logs de eventos para melhorar um modelo.

Aalst (2016) nos mostra um exemplo de um log de eventos na Tabela 2, onde cada linha corresponde a um evento. Na primeira coluna temos o *id* do *trace*, na segunda coluna, o *id* do evento, na terceira coluna temos o *timestamp*, ou seja, um carimbo com a data e a hora que o evento acontece, na quarta coluna temos a atividade que está sendo realizada, as outras duas colunas são formadas por informações adicionais: quem está realizando a atividade e o custo de cada atividade na quinta e na sexta coluna respectivamente.

Na Tabela 3, temos uma representação mais compacta do *log* de eventos da Tabela 2, na qual consideramos apenas as atividades e a ordem que elas ocorrem, onde a = *register request*, b = *examine thoroughly*, c = *examine casually*, d = *check ticket*, e = *decide*, f = *reinitiate request*, g = *pay compensation* e h = *reject request*.

De acordo com Aalst (2016), um elemento chave da mineração de processos é o estabelecimento de um forte laço entre um modelo de processo e a "realidade", para representar essa relação, utilizamos os termos *play-in*, onde, dado um comportamento como entrada, o objetivo é construir um modelo que o represente, *play-out*, onde dado um modelo de entrada, o objetivo é gerar comportamento, e *replay*, que utiliza um *log* de eventos e um modelo de processo como entrada e faz um *replay*, como o próprio nome indica, para verificar conformidade, estender o modelo com novas informações, entre outras utilidades.

Tabela 2 – Exemplo de *log* de eventos

Case id	Event id	Timestamp	Activity	Resource	Cost
1	35654423	30-12-2010:11.02	Register request	Pete	50
	35654424	31-12-2010:10.06	examine thoroughly	Sue	400
	35654425	05-01-2011:15.12	check ticket	Mike	100
	35654426	06-01-2011:11.18	decide	Sara	200
	35654427	07-01-2011:14.24	reject request	Pete	200
2	35654483	30-12-2010:11.32	register request	Mike	50
	35654485	30-12-2010:12.12	check ticket	Mike	100
	35654487	30-12-2010:14.16	examine casually	Pete	400
	35654488	05-01-2011:11.22	decide	Sara	200
	35654489	08-01-2011:12.05	pay compensation	Ellen	200
3	35654521	30-12-2010:14.32	register request	Pete	50
	35654522	30-12-2010:15.06	examine casually	Mike	400
	35654524	30-12-2010:16.34	check ticket	Ellen	100
	35654525	06-01-2011:09.18	decide	Sara	200
	35654526	06-01-2011:12.18	reinitiate request	Sara	200
	35654527	06-01-2011:13.06	examine thoroughly	Sean	400
	35654530	08-01-2011:11.43	check ticket	Pete	100
	35654531	09-01-2011:09.55	decide	Sara	200
	35654533	15-01-2011:10.45	pay compensation	Ellen	200
4	35654641	06-01-2011:15.02	register request	Pete	50
	35654643	07-01-2011:12.06	check ticket	Mike	100
	35654644	08-01-2011:14.43	examine thoroughly	Sean	400
	35654645	09-01-2011:12.02	decide	Sara	200
	35654647	12-01-2011:15.44	reject request	Ellen	200
5	35654711	06-01-2011:09.02	register request	Ellen	50
	35654712	07-01-2011:10.16	examine casually	Mike	400
	35654714	08-01-2011:11.22	check ticket	Pete	100
	35654715	10-01-2011:13.28	decide	Sara	200
	35654716	11-01-2011:16.18	reinitiate request	Sara	200
	35654718	14-01-2011:14.33	check ticket	Ellen	100
	35654719	16-01-2011:15.50	examine casually	Mike	400
	35654720	19-01-2011:11.18	decide	Sara	200
	35654721	20-01-2011:12.48	reinitiate request	Sara	200
	35654722	21-01-2011:09.06	examine casually	Sue	400
	35654724	21-01-2011:11.34	check ticket	Pete	100
	35654725	23-01-2011:13.12	decide	Sara	200
	35654726	24-01-2011:14.56	reject request	Mike	200
6	35654871	06-01-2011:15.02	register request	Mike	50
	35654873	06-01-2011:16.06	examine casually	Ellen	400
	35654874	07-01-2011:16.22	check ticket	Mike	100
	35654875	07-01-2011:16.52	decide	Sara	200
	35654877	16-01-2011:11.47	pay compensation	Mike	200
...	...	...	...	...	...

Fonte: (AALST, 2016)

Tabela 3 – Representação compacta do *log* de eventos da Tabela 2

Case id	**Trace**
1	$\langle a,b,d,e,h\rangle$
2	$\langle a,d,c,e,g\rangle$
3	$\langle a,c,d,e,f,b,d,e,g\rangle$
4	$\langle a,d,b,e,h\rangle$
5	$\langle a,c,d,e,f,c,d,e,h\rangle$
6	$\langle a,c,d,e,g\rangle$
...	...

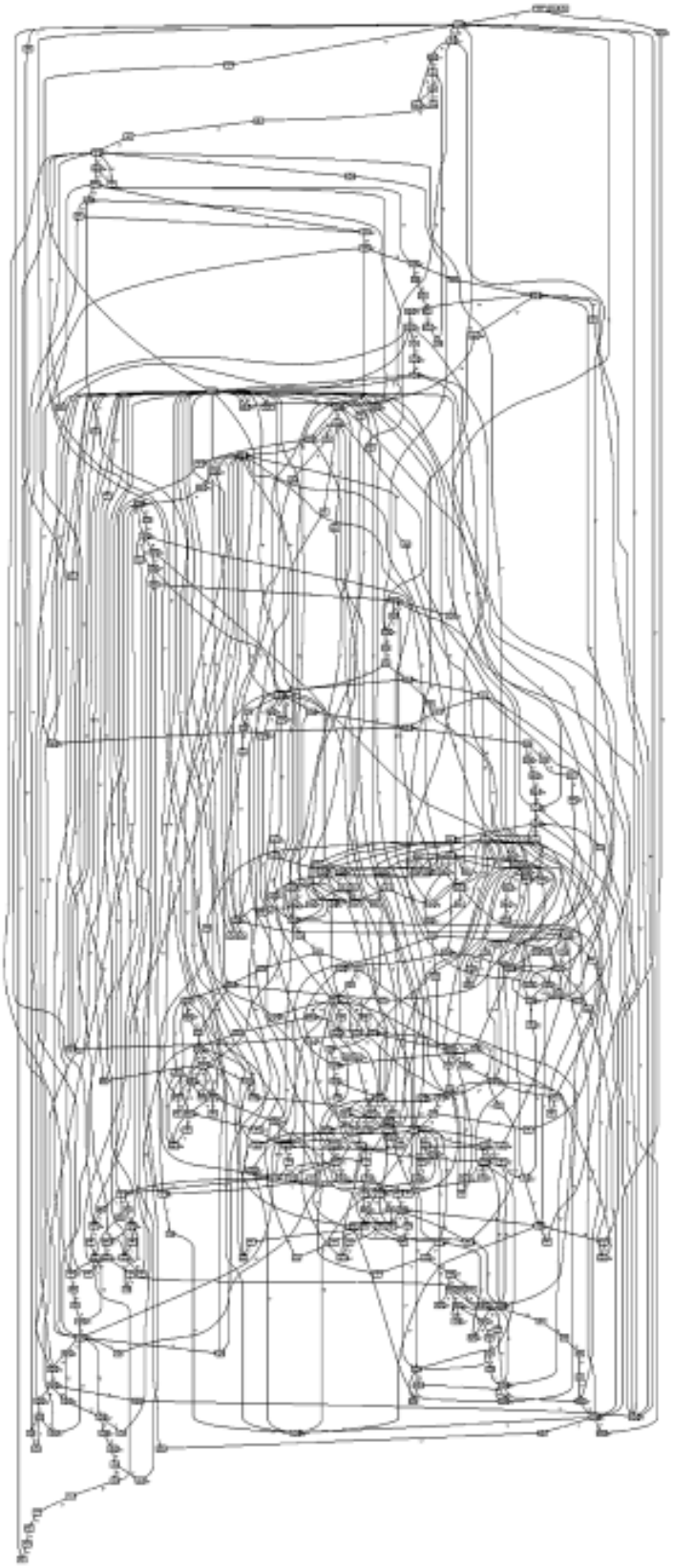
Fonte: (AALST, 2016)

Segundo Aalst *et al.* (2011), a mineração de processos pode abranger diferentes perspectivas, como a perspectiva de controle de fluxo (*the control-flow perspective*), a perspectiva organizacional (*the organisational perspective*), a perspectiva dos casos (*the case perspective*) e a perspectiva do tempo (*the time perspective*). A perspectiva de controle de fluxo concentra-se na ordenação das atividades para encontrar uma boa caracterização de todos os caminhos possíveis e expressá-los em alguma notação, como uma rede de Petri, por exemplo. A perspectiva organizacional concentra-se em saber quais atores estão envolvidos e como são os seus relacionamentos para estruturar a organização. A perspectiva dos casos foca nas propriedades dos casos como o caminho no processo ou os seus atores. Por fim, a perspectiva do tempo preocupa-se com o tempo e a frequência dos eventos.

Na área de descoberta de processos, podemos encontrar vários algoritmos com diferenças entre cada um. Segundo Aalst (2016) temos: algoritmo α (AALST, 2016), *heuristic miner* (WEIJTERS; RIBEIRO, 2011), *inductive miner* (AALST, 2016), *genetic process miner* (MEDEIROS *et al.*, 2007), entre outros.

Representar graficamente processos muito complexos é um problema encontrado pois tais processos podem gerar sistemas sem uma estrutura organizada, conhecidos como **processos espaguetes**, que dificultam o entendimento do comportamento do processo. Um exemplo de processo espaguete é mostrado na Figura 29, onde é possível perceber a dificuldade de interpretar o processo.

Figura 29 – Exemplo de processo espaguete



Fonte: (AALST, 2016)

2.2.3 Critérios de avaliação de processos

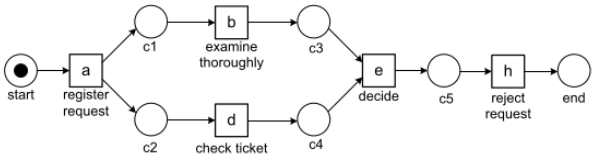
Segundo Aalst (2016), os principais critérios de qualidade de um modelo são:

- Acurácia, ou *fitness*: O modelo descoberto deve permitir o comportamento visto no *log* de eventos. É o critério mais relacionado com conformidade. *Fitness* pode ser calculado tanto para um *trace* específico do *log* ou então para todo o conjunto de *traces* do *log*.
- Precisão, ou *precision*: O modelo descoberto não deve permitir um comportamento completamente diferente do que foi visto no *log* de eventos.
- Generalização, ou *generalization*: O modelo descoberto deve generalizar o comportamento do exemplo visto no *log* de eventos.
- Simplicidade, ou *simplicity*: O modelo descoberto deve ser o mais simples possível, ou seja, que tenha o menor número de componentes, como transições, por exemplo.

Segundo Aalst (2016), um dos principais desafios encontrados quando se modelam negócios é garantir o balanceamento entre os quatro critérios. Por exemplo, um modelo muito simples pode ter problemas de precisão ou de aptidão. É dito que um modelo é *overfitting* quando ele é tão específico que só permite que haja um, ou poucos comportamentos, sendo assim, um modelo que não generaliza, do mesmo modo, é dito que um modelo é *underfitting* quando ele é tão geral a ponto de permitir comportamentos que não têm relação nenhuma com os comportamentos observados, ou seja, é um modelo que peca em precisão. É fácil ver que o sobreajuste (*overfitting*) e o sob ajuste (*underfitting*) são de difícil balanceamento.

Aalst (2016) nos mostra um exemplo, onde temos apenas dois *traces*, são eles $\langle a,b,d,e,h\rangle$ e $\langle a,d,b,e,h\rangle$, baseado nisso, o algoritmo α irá gerar o modelo de processo apresentado na imagem 30:

Figura 30 – Exemplo de modelo de processo descoberto pelo algoritmo α



Fonte: (AALST, 2016)

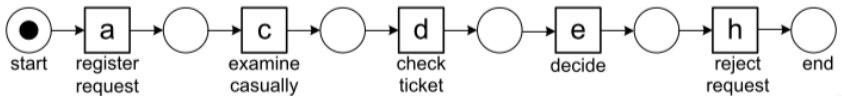
Podemos notar que o modelo descoberto aceita apenas dois comportamentos, que são justamente os comportamentos observados nos *traces* que serviram de base, logo, temos um caso de *overfitting*. Para casos mais complexos, é recomendável que se evite o *overfitting*, descobrindo concorrência entre as atividades, por exemplo, pois, segundo Aalst (2016), a não

descoberta de concorrência pode gerar modelos espaguetes, ou então generalizando o modelo descobrindo *loops* ou ciclos.

Para exemplificar como funciona o balanceamento entre os quatro critérios, Aalst (2016) mostra um exemplo. Considere o *log* de eventos $L = [\langle a,c,d,e,h\rangle, \langle a,b,d,e,g\rangle, \langle a,d,c,e,h\rangle, \langle a,b,d,e,h\rangle, \langle a,c,d,e,g\rangle, \langle a,d,c,e,g\rangle, \langle a,d,b,e,h\rangle, \langle a,c,d,e,f,d,b,e,h\rangle, \langle a,c,d,e,f,b,d,e,h\rangle, \langle a,c,d,e,f,d,b,e,g\rangle, \langle a,d,c,e,f,c,d,e,h\rangle, \langle a,d,c,e,f,d,b,e,h\rangle, \langle a,d,c,e,f,b,d,e,g\rangle, \langle a,c,d,e,f,b,d,e,f,d,b,e,g\rangle, \langle a,d,c,e,f,d,b,e,g\rangle, \langle a,d,c,e,f,b,d,e,f,b,d,e,g\rangle, \langle a,d,c,e,f,d,b,e,f,b,d,e,h\rangle, \langle a,d,b,e,f,b,d,e,f,d,b,e,g\rangle, \langle a,d,c,e,f,d,b,e,f,c,d,e,f,d,b,e,g\rangle]$. É possível construir pelo menos quatro modelos diferentes baseados em L.

O primeiro modelo, pode ser observado na Figura 31.

Figura 31 – Exemplo 1 de modelo de processo para o *log* L

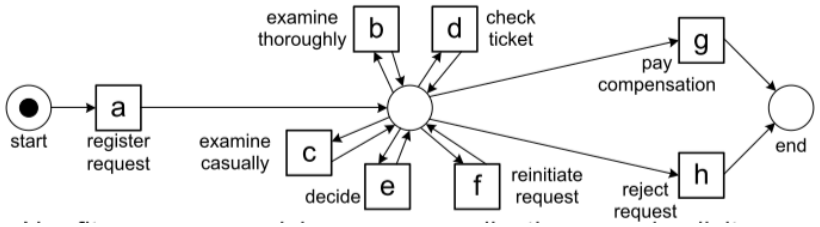


Fonte: (AALST, 2016)

Observa-se que o modelo descoberto tem uma boa precisão, pois não permite comportamentos diferentes dos comportamentos vistos no *log*, e é bastante simples. Entretanto, tal modelo não possui um bom *fitness*, pois há comportamentos do *log* que não são reproduzíveis no modelo, nem uma boa generalização, pois ele não permite comportamentos não vistos no *log*.

O próximo modelo, pode ser observado na Figura 32.

Figura 32 – Exemplo 2 de modelo de processo para o *log* L



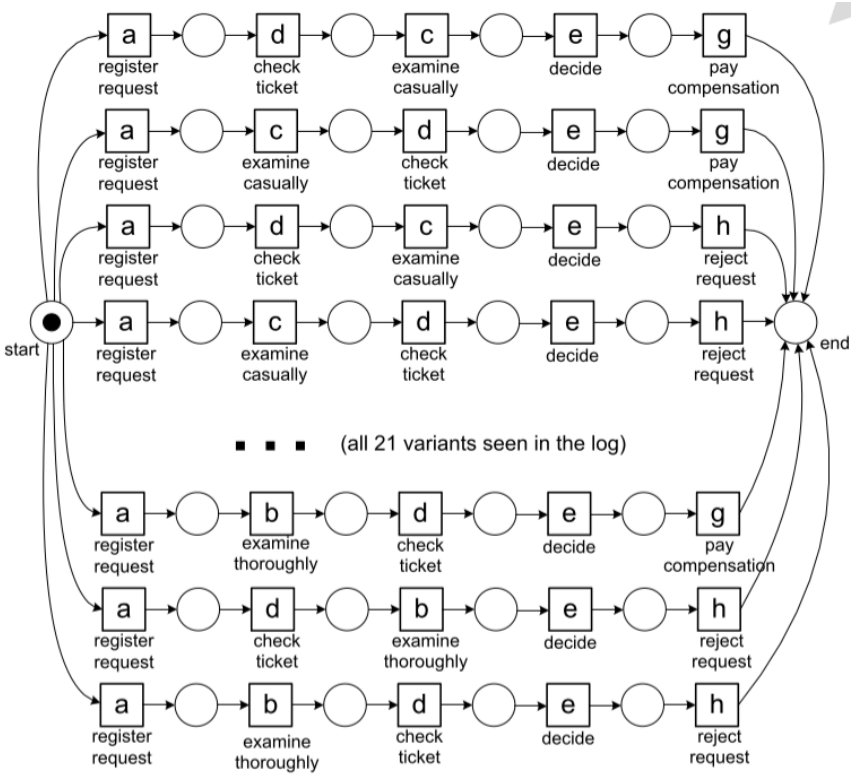
Fonte: (AALST, 2016)

Observa-se que o modelo tem uma boa generalização, permitindo mais comportamentos do que os vistos no *log*, um bom *fitness*, pois permite os comportamentos vistos no *log*, e é bem simples, mas peca na precisão, pois permite diversos comportamentos muito diferentes do que os observados no *log*. Modelos desse tipo são conhecidos por "*flower models*" ou "modelos flores" e têm característica a generalização exacerbada, permitindo praticamente qualquer

comportamento do *log*.

O próximo modelo pode ser observado na Figura 33.

Figura 33 – Exemplo 3 de modelo de processo para o *log* L



Fonte: (AALST, 2016)

O modelo em questão possui um bom *fitness* e uma boa precisão, pois permite os comportamentos observados no *log*, mas como o modelo só permite esses comportamentos, nota-se que há uma baixa generalização. Como o modelo é representado *trace* por *trace*, e o *log* é grande, vemos também uma baixa simplicidade.

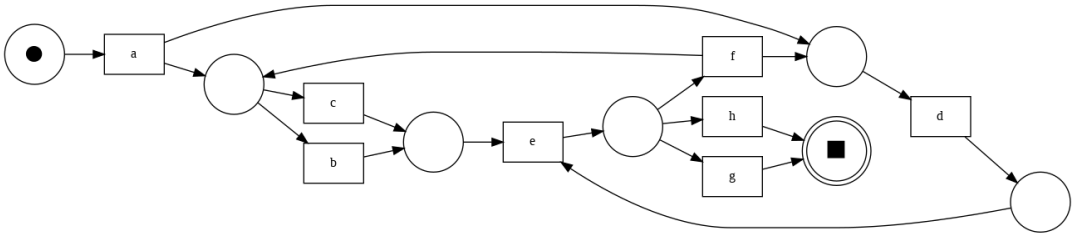
Por fim, um modelo que possui um bom balanceamento para o mesmo *log* já foi apresentado na Figura 20. Iremos reapresentá-lo na Figura 34 a seguir. Observa-se que o modelo permite todos os comportamentos vistos no *log*, além de permitir mais alguns comportamentos não vistos e é construído de uma forma bastante simples.

Dos 4 modelos gerados a partir de L, este é o que melhor equilibra os critérios.

2.2.3.1 Token Based Replay

Para quantificarmos a noção de *fitness*, podemos utilizar estratégias como o *Replay* baseado em *tokens* (ou *Token based replay*). De acordo com Aalst (2016), a principal característica deste método é que a noção de *fitness* é aplicada a nível de eventos, ao invés de aplicá-la a

Figura 34 – Exemplo de rede de Petri e seus componentes



Fonte: (AALST, 2016)

nível de *traces*. Desta forma, quando se encontra um problema, o *replay* não é interrompido pois força-se as transições a serem executadas.

Ao final, registra-se a quantidade de *tokens* que foram produzidos, representada por **p**, a quantidade de *tokens* que foram consumidos, representada por **c**, a quantidade de *tokens* que foram inseridos à força, representada por **m**, e a quantidade de *tokens* que, ao final do *replay*, não atingiram o estado final, representada por **r**. Cada valor registrado será utilizado na fórmula que calcula o *fitness*:

- Para apenas um trace:

$$0,5*(1-(m/c))+0,5*(1-(r/p)) \tag{2.38}$$

- Para o *log* completo:

$$\begin{aligned} &0,5*(1-(\Sigma_{\sigma\in L}L(\sigma)\times m_{N,\sigma}/\Sigma_{\sigma\in L}L(\sigma)\times c_{N,\sigma}))+\\ &\quad 0,5*(1-(\Sigma_{\sigma\in L}L(\sigma)\times r_{N,\sigma}/\Sigma_{\sigma\in L}L(\sigma)\times p_{N,\sigma})) \end{aligned} \tag{2.39}$$

onde $L(\sigma)$ é a frequência do *trace* σ.

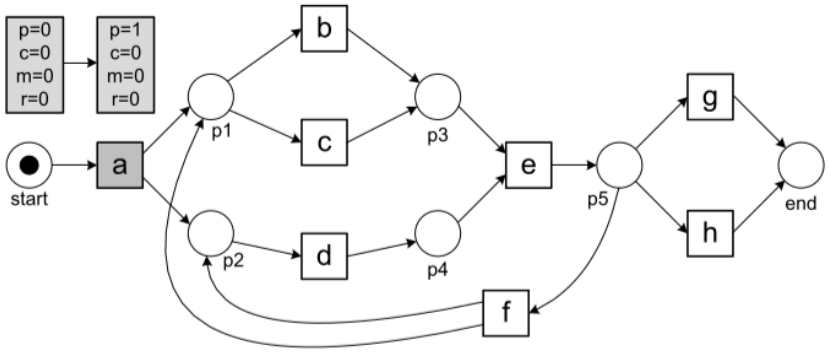
O resultado da conta será entre 0 e 1, onde 0 quer dizer que os *traces* do *log* são totalmente irreconhecíveis pelo modelo, e 1 significa que o modelo reconhece completamente os *traces* do *log*.

Aalst (2016) nos mostra um exemplo de um modelo representado por uma Rede de Petri e o passo a passo realizado pelo *replay*. A Figura 35 nos mostra o primeiro passo do *replay* do *trace* $\langle a,c,d,e,h\rangle$. Nota-se que ao iniciar o modelo, o ambiente já produz um *token* para a marcação inicial.

Agora, tenta-se ativar a transição a. É possível e a consome um *token* e produz dois *tokens*, como mostrado na Figura 36.

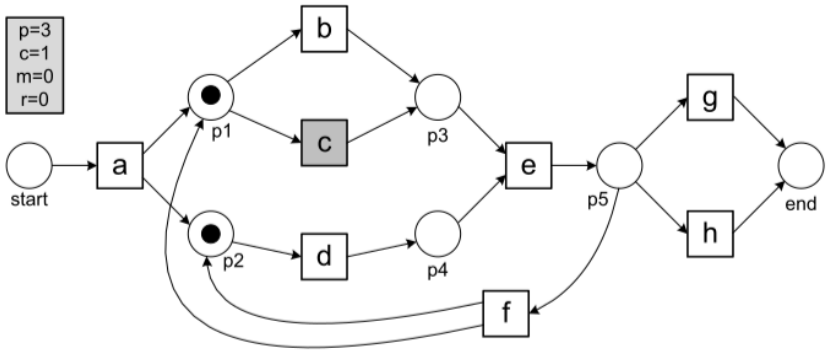
Então faz-se o *replay* do segundo evento: c, que irá consumir um *token* e produzir um *token*, como mostrado na Figura 37.

Figura 35 – Primeiro passo de *replay* do *trace* $\langle a,c,d,e,h \rangle$ em modelo



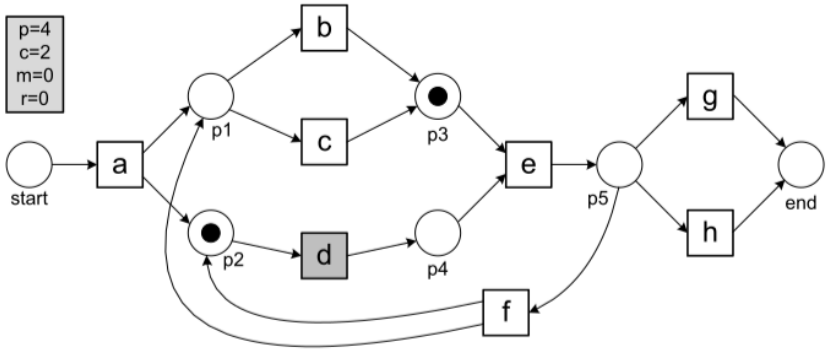
Fonte: (AALST, 2016)

Figura 36 – Segundo passo de *replay* do *trace* $\langle a,c,d,e,h \rangle$ em modelo



Fonte: (AALST, 2016)

Figura 37 – Terceiro passo de *replay* do *trace* $\langle a,c,d,e,h \rangle$ em modelo



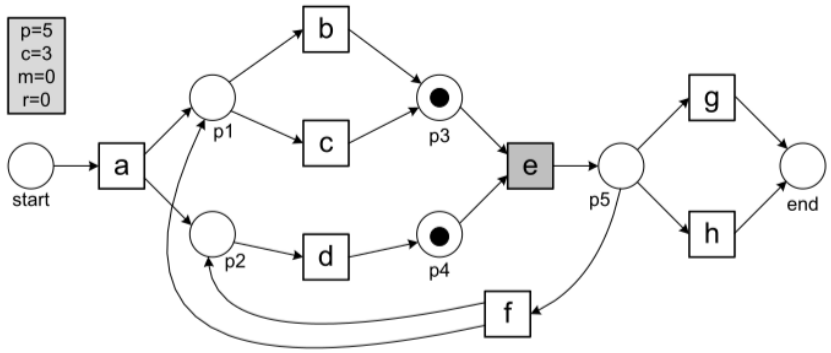
Fonte: (AALST, 2016)

Após isso é feito o *replay* do terceiro evento: *d*, que, assim como o evento anterior irá consumir um *token* e produzir um *token* do modelo. Na Figura 38 temos a configuração do modelo após a ativação da transição *d*.

Nas Figuras 39 e 40 são mostrados os resultados após fazer o *replay* dos eventos *e* e *h*, respectivamente. Nota-se que o evento *e* consome dois *tokens* e produz apenas um e que *h* consome um *token* e produz um *token*, mas que ao chegar na marcação final do modelo, também se é consumido um *token*.

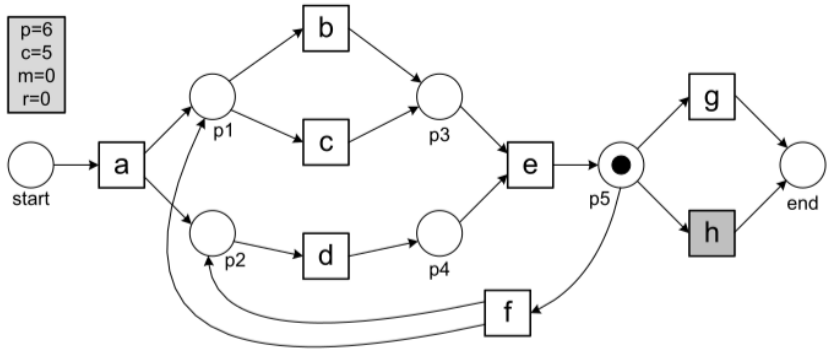
Assim, ao aplicar a fórmula que calcula o *fitness* de um *trace*, tem-se:

Figura 38 – Quarto passo de *replay* do *trace* $\langle a,c,d,e,h\rangle$ em modelo



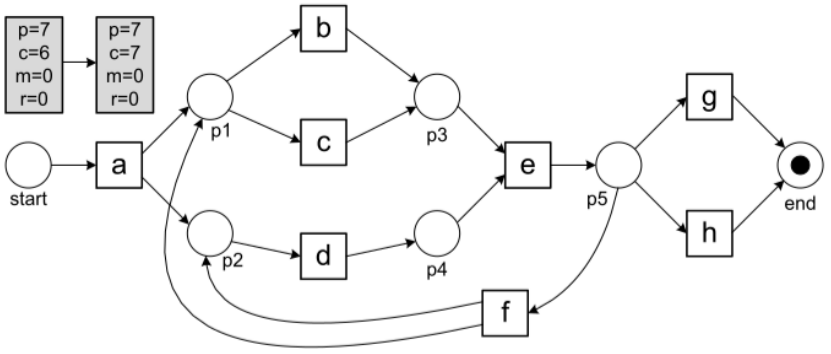
Fonte: (AALST, 2016)

Figura 39 – Quinto passo de *replay* do *trace* $\langle a,c,d,e,h\rangle$ em modelo



Fonte: (AALST, 2016)

Figura 40 – Sexto passo de *replay* do *trace* $\langle a,c,d,e,h\rangle$ em modelo



Fonte: (AALST, 2016)

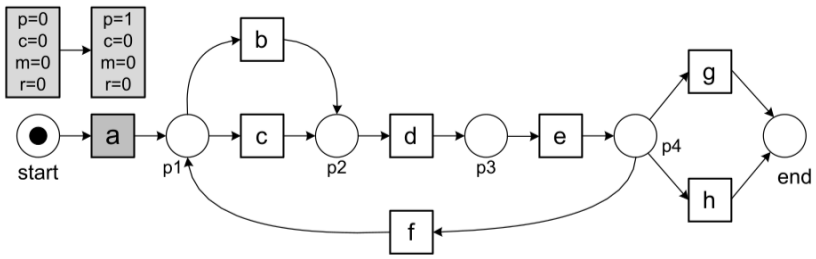
$$0,5*(1-(0/7))+0,5*(1-(0/7))=1.$$

Que quer dizer que o *trace* foi completamente reconhecido pelo modelo, sem precisar forçar nenhuma transição.

Aalst (2016) também mostra um exemplo onde o resultado não é igual a 1. Considere o *trace* $\sigma_2 = \langle a,d,c,e,h\rangle$. Será feito o *replay* de σ_2 no modelo N_2, mostrado na Figura 41.

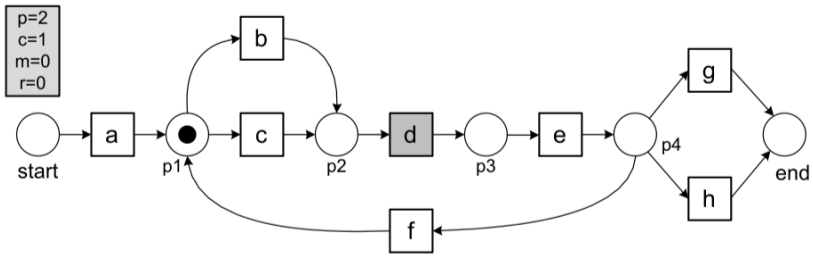
Primeiramente é produzido o *token* inicial e depois faz-se o *replay* de "a". Vê-se que é possível e que "a" consome um *token* e produz um *token*, como mostrado na Figura 42.

Figura 41 – Primeiro passo de *replay* do *trace* $\langle a,d,c,e,h\rangle$ em modelo N_2



Fonte: (AALST, 2016)

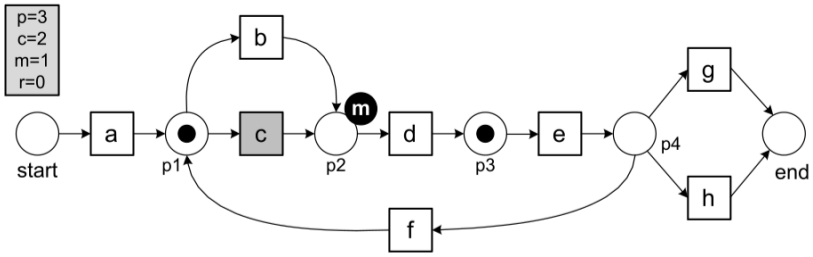
Figura 42 – Segundo passo de *replay* do *trace* $\langle a,d,c,e,h\rangle$ em modelo N_2



Fonte: (AALST, 2016)

Agora, tenta-se fazer o *replay* de "d", que não é diretamente alcançável a partir do *place* p1. Assim, será inserido um *token* de forma que seja possível acionar "d" . O marcador "m", que indica o número de *tokens* que foram inseridos forçadamente, é acrescido em 1 e um *token* é produzido e um *token* é consumido, como mostra a Figura 43.

Figura 43 – Terceiro passo de *replay* do *trace* $\langle a,d,c,e,h\rangle$ em modelo N_2



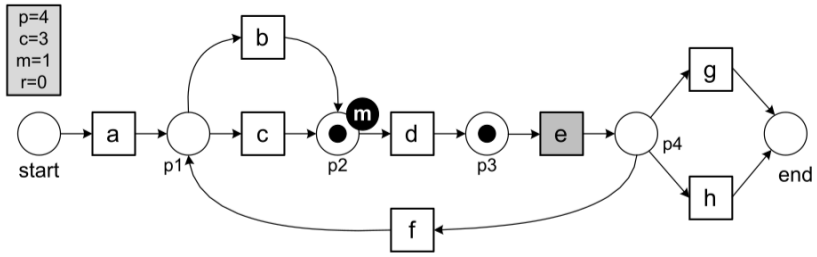
Fonte: (AALST, 2016)

Após isso, faz-se o *replay* do terceiro evento do *trace* que é "c", que é um evento alcançável e produz e consome um *token*, como mostra a Figura 44

A Figura 45 mostra que ao fazer o *replay* de "e", é produzido e consumido um *token* e, na Figura 46, nota-se que ao fazer o *replay* de "h" é produzido e consumido um *token*, e ao chegar na marcação final, é consumido um *token*.

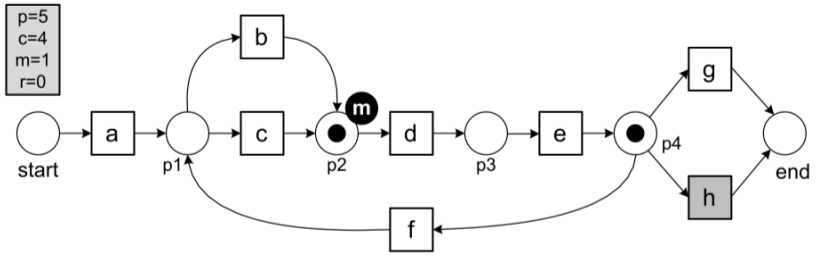
Ao fim, como mostra a Figura 46, tem-se seis *tokens* produzidos e seis *tokens* consumidos, logo, $p = c = 6$, mas nota-se que temos um *token* que foi inserido forçadamente, assim $m = 1$, e um *token* ainda restou no modelo sem ser consumido, dessa forma $r = 1$. Desta

Figura 44 – Quarto passo de *replay* do *trace* $\langle a,d,c,e,h\rangle$ em modelo N_2



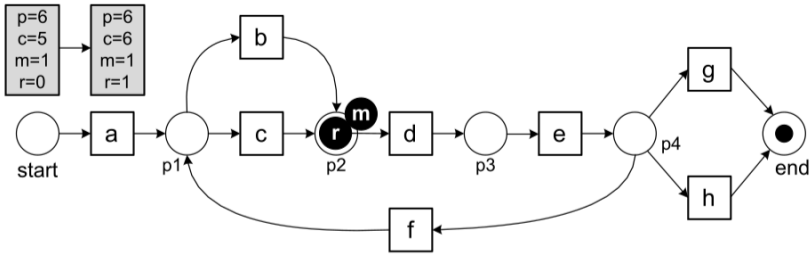
Fonte: (AALST, 2016)

Figura 45 – Quinto passo de *replay* do *trace* $\langle a,d,c,e,h\rangle$ em modelo N_2



Fonte: (AALST, 2016)

Figura 46 – Sexto passo de *replay* do *trace* $\langle a,d,c,e,h\rangle$ em modelo N_2



Fonte: (AALST, 2016)

forma, ao aplicar a fórmula que calcula o *fitness* de um *trace*, obtem-se:

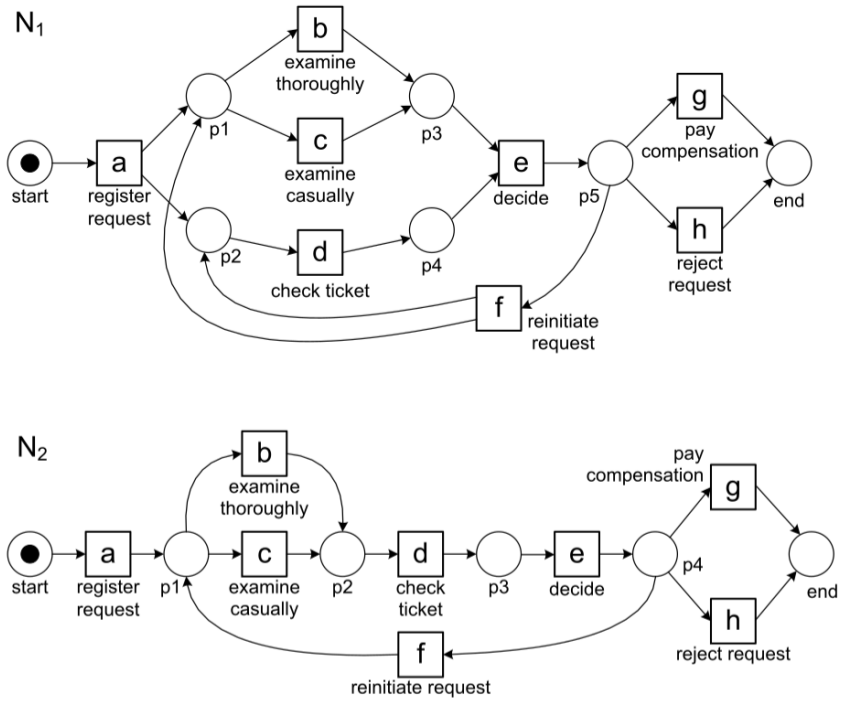
$$0,5*(1-(1/6))+0,5*(1-(1/6))=0,8333.$$

Como Berti *et al.* (2019) ressalta, é importante notar que esta abordagem tem problemas, como por exemplo, o problema da inundação de *tokens*, que ocorre quando o comportamento do *log* difere consideravelmente do modelo, pois são inseridos vários *tokens* para realizar o *replay*, o que também acarreta em vários *tokens* permanecendo no modelo sem serem consumidos. Desta forma, o resultado da medida do *fitness* pode ser muito alto, pois vários eventos serão ativados mesmo sendo inalcançáveis, subsequentemente permitindo praticamente qualquer comportamento. Outros problemas, levantados por Aalst (2016) são a abordagem ser muito específica para redes de Petri e, caso se deseje utilizá-la para um modelo de outro tipo, será necessário realizar uma conversão para rede de Petri, e também a dificuldade em tratar modelos com transições invisíveis e eventos com o mesmo *label*.

2.2.3.2 *Alignments*

De acordo com Aalst (2016), a abordagem *alignments* (ou alinhamentos) surgiu para lidar com essas limitações do *token based replay*. A melhor correspondência entre um trace e um modelo é chamada de alinhamento ótimo. Para exemplificar, Aalst (2016) nos mostra os seguintes modelos da Figura 47.

Figura 47 – Exemplo de modelos



Fonte: (AALST, 2016)

Agora, consideramos o *trace* $\sigma = \langle a,d,b,e,h \rangle$. O alinhamento ótimo entre o *trace* e o modelo pode ser representado por uma tabela, onde a primeira linha corresponde ao *trace* e a segunda linha corresponde ao caminho da marcação inicial até a marcação final do modelo. No caso do modelo N_1 temos apenas um alinhamento ótimo, que é mostrado na Figura 48.

Figura 48 – Alinhamento do *trace* $\langle a,d,b,e,h \rangle$ em N_1

$$\gamma_1 = \begin{array}{|c|c|c|c|c|} a & d & b & e & h \\ \hline a & d & b & e & h \end{array}$$

Fonte: (AALST, 2016)

No modelo N_2 temos vários alinhamentos ótimos, como mostrados na Figura 49.

Onde » significa que houve um desalinhamento, como em γ_{2a}, por exemplo, que há

Figura 49 – Alinhamentos do *trace* $\langle a,d,b,e,h\rangle$ em N_2

$$\gamma_{2a}=\begin{vmatrix} a & \gg & d & b & e & h \\ a & b & d & \gg & e & h \end{vmatrix} \qquad \gamma_{2b}=\begin{vmatrix} a & \gg & d & b & e & h \\ a & c & d & \gg & e & h \end{vmatrix} \qquad \gamma_{2c}=\begin{vmatrix} a & d & b & \gg & e & h \\ a & \gg & b & d & e & h \end{vmatrix}$$

Fonte: (AALST, 2016)

um “b” no modelo antes de “b” acontecer no *log*, assim, o *log* também faz um movimento que é impossível no modelo também. Neste caso, vemos que todos os alinhamentos têm dois »’s.

De acordo com Aalst (2016), um movimento é um par (x, (y,t)), onde o primeiro elemento representa a ocorrência do *log* enquanto o segundo elemento representa o que ocorre no modelo. Por exemplo, (a, (a,t1)) significa que o *log* faz o movimento “a”, que o modelo também faz o movimento “a” e este movimento do modelo é causado pela transição t1. Desta forma, Aalst (2016) nos mostra os quatro casos de movimentos legalizados e a definição de alinhamentos:

- x = y, onde y é o *label* visível da transição t;
- x = » e y é o *label* visível da transição t;
- x = », y é τ e t é a transição invisível;
- x $\neq$ » e (y, t) = ».

Um alinhamento é uma sequência de movimentos legais onde, ao removermos todos os », a primeira linha corresponde ao *trace* do *log* e a segunda linha representa uma sequência de ações que partem da marcação inicial até a marcação final.

Como o mesmo modelo pode ter diversos alinhamentos possíveis, como por exemplo na Figura 49, devemos escolher um modelo que seja o mais apropriado. Assim, são designados custos para os movimentos indesejados, de forma que o alinhamento com o menor custo será o alinhamento escolhido. Quando os movimentos do *trace* e do modelo são legais, então não há custos, quando não são, como por exemplo haver um movimento no modelo que não está previsto no *trace*, um custo é associado a esse movimento, a depender do movimento e de sua importância no processo. Para simplificar, assumiremos que todos os custos diferentes de 0 serão 1. Aalst (2016) nos diz que um alinhamento é ótimo quando não há nenhum outro alinhamento possível com um custo menor.

O *fitness* de um *trace* σ pode ser calculado da seguinte forma:

$$fitness(\sigma,N) = 1 - (\delta(\lambda^N_{opt}(\sigma)/(\delta(\lambda^N_{worst}(\sigma)). \tag{2.40}$$

Onde $\lambda^N_{worst}(\sigma)$ é o pior caso possível, ou seja, um caso onde todos os movimentos são des-

sincronizados, e $\lambda^{N}_{opt}(\sigma)$ é o melhor caso encontrado. O resultado estará entre 0, significando que o modelo não corresponde ao *trace*, e 1, significando que o modelo pode repetir o *trace* completamente.

De acordo com Aalst (2016), esta noção também pode ser estendida para todo o *log*, desta forma:

$$1 - (\Sigma_{\sigma \in L} L(\sigma) \times \delta(\lambda^{N}_{opt}(\sigma)) / \Sigma_{\sigma \in L} L(\sigma) \times \delta(\lambda^{N}_{worst}(\sigma))) \tag{2.41}$$

3 TRABALHOS RELACIONADOS

Neste capítulo serão apresentadas algumas abordagens que utilizam sistemas de transições para representar logs de eventos e também trabalhos sobre decomposição de autômatos.

3.1 Mineração de processos: Uma abordagem em dois passos usando sistemas de transições e regiões

O trabalho de Aalst *et al.* (2006) utiliza uma abordagem em dois passos para resolver problemas de descoberta de processos com dependências muito complicadas. O primeiro passo da abordagem é transformar um log de eventos em um sistema de transições. Tendo em vista isso, os autores determinaram o estado do log focando nas atividades passadas e nas futuras do *trace*. Baseado nessa definição de estados é criado o sistema de transições onde o conjunto de atividades será o conjunto de ações do autômato, dessa forma, dependendo da escolha do tipo de representação de estado, poderá ser construído diferentes autômatos. Logo após é utilizada algumas estratégias, como por exemplo adicionar ou remover transições, juntar estados para formar um novo, "matar *loops*", entre outros, para evitar alguns problemas, como a falta de generalização ou informações desnecessárias.

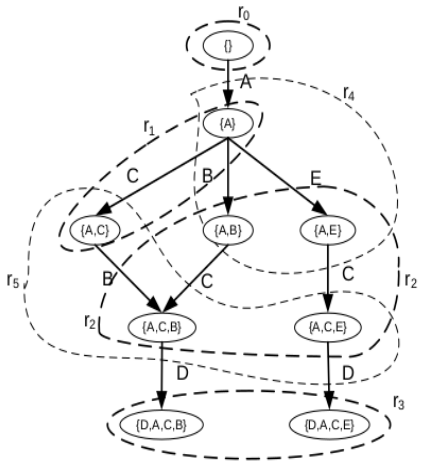
Com o sistema de transições construído, é utilizada uma abordagem que irá usar teoria das regiões para transformá-lo em uma rede de Petri. Segundo os autores, uma região *S'*, onde *S'* é um subconjunto de estados, é definida se, para cada evento e, ou seja, para cada transição, uma das condições a seguir é satisfeita:

- Todas as transições $s_1 \rightarrow^e s_2$ entram em *S'*, ou seja, $s_1 \notin S'$ e $s_2 \in S'$;
- Todas as transições $s_1 \rightarrow^e s_2$ saem *S'*, ou seja, $s_1 \in S'$ e $s_2 \notin S'$;
- Todas as transições $s_1 \rightarrow^e s_2$ não cruzam *S'*, ou seja, $s_1, s_2 \in S'$ ou $s_1, s_2 \in S'$.

Um exemplo de regiões em um sistema de transições é apresentado na Figura 50.

Baseado nisso, os autores utilizam algoritmos mostrados nos trabalhos Cortadella *et al.* (1995) e Cortadella *et al.* (1998) com algumas modificações para criar uma rede de Petri sem lugares redundantes. Para isso, são definidos custos para as diferentes soluções geradas baseadas nas regiões mínimas criadas. Os autores implementaram suas ideias no contexto da ferramenta ProM (VERBEEK *et al.*, 2010) e compararam sua abordagem com dois algoritmos de mineração bastante utilizados: o algoritmo α (AALST *et al.*, 2004) e o algoritmo multi-fase (DONGEN; AALST, 2004). Em comparação, a abordagem utilizada é mais complexa e mais lenta, mas em

Figura 50 – Exemplo de sistema de transições e suas regiões



Fonte: (AALST *et al.*, 2006)

compensação, tem uma maior acurácia no resultado e captura melhor dependências de longo alcance, além de poder gerar resultados diferentes de acordo com suas configurações. Embora o trabalho compartilhe o fato de utilizar sistemas de transição no contexto de mineração de processos com a abordagem do presente trabalho, ele difere por causa das estratégias usadas para "melhorar" o sistema de transições gerado e por transformá-lo em uma rede de Petri ao final.

3.2 Descoberta de modelos de processos: Um método baseado em decomposição de sistemas de transições

O trabalho de Kalenkova *et al.* (2014) apresenta uma nova abordagem de decomposição para descobrir modelos mais legíveis de logs de eventos com base no conhecimento preeliminar sobre a estrutura do log de eventos: casos regulares e especiais da execução do processo são tratados separadamente. O método de decomposição proposto pelos autores visa explorar a modularidade em processos para obter modelos de processo mais legíveis. Usando técnicas existentes, primeiro é produzido um sistema de transição. Este sistema de transição é decomposto em uma parte regular e uma parte especial com base no conhecimento que já se tem sobre um registro de eventos. Em seguida, são aplicados algoritmos de descoberta baseados em região em cada parte do sistema de transição para descobrir modelos de subprocessos que no final serão combinados em um modelo unificado. Depois, os autores comprovam que as propriedades estruturais e comportamentais dos modelos de subprocesso são herdados pelo modelo de processo unificado. Após separar o sistema de transição, é criada uma rede de Petri baseada nos novos sistemas de transição, nelas, será aplicado o algoritmo A para obter um fluxo

de processo regular e um especial, correspondendo as partições do sistema de transições original, então as redes de Petri separadas são reunificadas, formando assim uma nova rede de Petri. Essa abordagem é semelhante à apresentada neste trabalho por utilizar decomposição de autômatos no contexto de BPM, mas difere por transformar o sistema de transições em uma rede de Petri no fim, também não utilizamos apenas decomposição de autômatos na simplificação das máquinas, e, por fim, o objetivo entre os trabalhos diferem.

3.3 Resolução assistida de problemas e decomposições de autômatos finitos e sobre a utilidade da informação: *Framework* e caso de nfa

Outros trabalhos também utilizam a abordagem de decomposição de autômatos para resolução de problemas. O trabalho de Gaži e Rovan (2008) estuda a resolução assistida de problemas utilizando decomposição de autômatos. O trabalho parte do princípio que resolver um problema significa construir um autômato A para uma linguagem L e a "assistência" é dada por informação adicional sobre a entrada. Por exemplo, ao invés de olha para um autômato A onde L = L(A) nós podemos olhar para um autômato B possivelmente mais simples tal que $L = L(B) \cap L'$ onde L' ou o seu autômato correspondente (A' onde $L' = L(A')$) é chamada de "conselheira" pois irá fazer uma espécie de pré-processamento da entrada e será útil se conseguir que A' seja menos complicado que A e que B seja mais simples que A também. A medida de complexidade de um autômato neste trabalho é dada pelo número de estados do autômato. Logo o problema pode ser reformulado como um problema de decomposição de autômatos: dado um DFA A, encontrar dois DFAs A1 (resolvedor) e A2 (conselheiro) onde $w \in L(A)$ pode ser determinado pelas computações de A1 e A2. A decomposição é feita a partir de partições do autômato que irão agrupar estados que irão fazer uma sub-computação da entrada. O trabalho de Rovan e Sádovskỳ (2018) estende o trabalho de Gaži e Rovan (2008) para NFAs.

O trabalho de Rovan e Sádovskỳ (2018) apresenta uma abordagem que utiliza decomposição de autômatos e outros formalismo das linguagens formais e teoria dos autômatos para criar um *framework* que lida com a noção de utilidade da informação. Os autores definem a essência da abordagem da seguinte forma: "A informação é útil se pode nos ajudar a resolver um problema mais facilmente", em outras palavras, querem providenciar conselhos para possivelmente simplificar o problema. Em resumo, o problema é dado por uma linguagem L e resolvê-lo significa encontrar um NFA A onde L = L(A) e como os autores estão interessados numa solução simples, consideram A como um autômato mínimo. Se existe informações adicionais sobre a

palavra de entrada providenciadas pela linguagem $L(A_{adv})$, pode ser possível encontrar uma solução mais simples ainda, que seria um autômato A_{new} onde $L = L(A_{new}) \cap L(A_{adv})$. Os autores reformulam esta questão em termos puramente de teoria dos autômatos da seguinte forma: Existe uma decomposição paralela (não trivial) de um NFA mínimo para L? Ou seja, dado um NFA A, existem dois NFAs A_1 e A_2 onde $L(A) = L(A_1) \cap L(A_2)$ e que o número de estados de A_1 e de A_2 é menor que o de A. Os autores concluem que é possível definir formalmente um *framework* para formalizar e estudar alguns atributos de informação. Estes trabalhos se assemelham ao presente trabalho por utilizarem decomposição de autômatos, mas, nesses casos, a decomposição é utilizada para encontrar soluções mais simples para problemas utilizando uma espécie de autômato conselheiro, enquanto que nossa abordagem visa modularizar processos em BPM.

3.4 Decomposição de autômatos

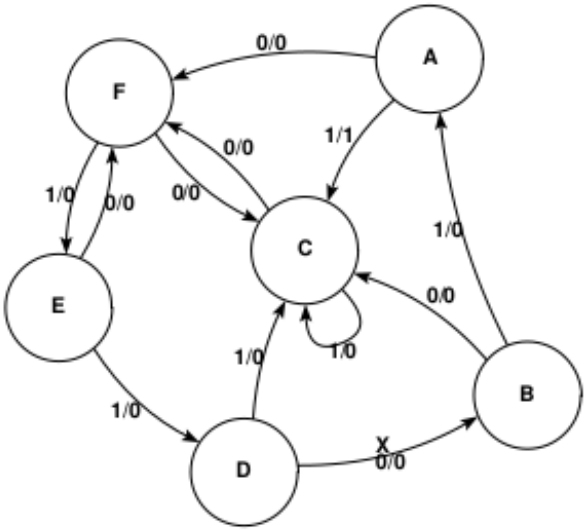
A decomposição de autômatos tem sido estudada há muito tempo, como nos trabalhos de Talwar (1997) e Rovan e Sádovskỳ (2018). Diversos outros trabalhos nos mostram formas de realizar a decomposição de máquinas de estados finitos, sendo a mais famosa apresentada pelo teorema de Krohn-Rhodes (KROHN; RHODES, 1965). Nesta seção serão apresentadas algumas formas de decompor um autômato em partes menores encontradas na literatura.

O teorema de Krohn-Rhodes, também conhecido como a decomposição algébrica hierárquica de automatos de estados finitos, é uma das principais contribuições da teoria dos semigrupos finitos e foi apresentado nos anos 60. Ele diz respeito a uma forma de decomposição de semigrupos finitos, como autômatos, por exemplo. Segundo Diekert *et al.* (2012) o teorema afirma que um semigrupo finito pode ser decomposto em um produto de coroa de grupos simples finitos bem controlados e semigrupos que não contenham nenhum subgrupo trivial. Em termo de autômatos, segundo Talwar (1997), o teorema diz que cada autômato de estado finito pode ser simulado por uma conexão de um autômato de grupo simples e dois autômatos de redefinição de estados.

Outra forma de decompor autômatos é apresentada no trabalho de Monteiro e Oliveira (1998). Nele, os autores dividem o autômato original arbitrariamente em dois, logo após criam estados auxiliares que irão receber as transições que cruzam os dois semiautômatos. O método é mostrado nas Figuras 51 e 52. O trabalho utiliza autômatos de Mealy (BRITO *et al.*, 2003), que são autômatos determinísticos onde cada transição gera uma saída. Desta forma, cada transição indica qual símbolo de entrada está sendo lido, de qual estado vem a transição e o que está sendo

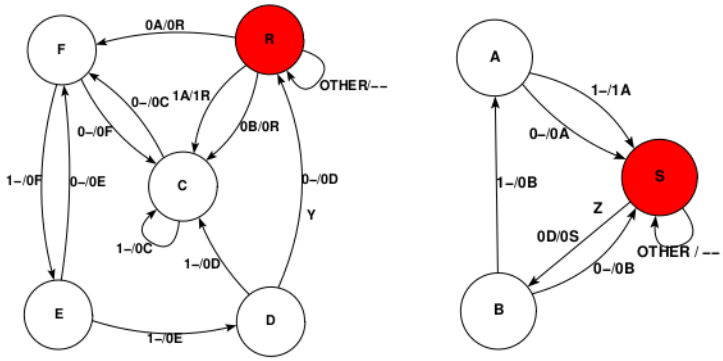
escrito na saída. É possível perceber que na Figura 51 a transição marcada com X se torna duas transições Y e Z na Figura 52 e que os estados auxiliares são marcados com a cor vermelha.

Figura 51 – Exemplo de autômato



Fonte: (MONTEIRO; OLIVEIRA, 1998)

Figura 52 – Autômato divido em 2



Fonte: (MONTEIRO; OLIVEIRA, 1998)

Outros trabalhos mostram diferentes formas de decompor autômatos como por exemplo os trabalhos de Rovan e Sádovskỳ (2018) e Gaži e Rovan (2008) que utilizam a estratégia de decomposição de autômatos para descobrir dois autômatos mais simples que o original onde o primeiro seria um resolvedor do problema enquanto o segundo seria um conselheiro (*advicer*) que pode dar informações úteis para que o resolvedor não precise ser tão complexo.

Definição 12 *(GAŽI; ROVAN, 2008) Um par de DFAs (A1, A2), onde* $A1 = (K_1, \Sigma, \delta_1, q_1, F_1)$ *e* $A2 = (K_2, \Sigma, \delta_2, q_2, F_2)$ *forma uma decomposição de identificação de aceitação (AI-Decomposition)*

de um DFA $A = (K, \Sigma, \delta, q, F)$*, se* $L(A) = L(A_1) \cap L(A_2)$.

Definição 13 *(GAŽI; ROVAN, 2008) Um par de DFAs (A1, A2), onde* $A1 = (K_1, \Sigma, \delta_1, q_1, F_1)$ *e* $A2 = (K_2, \Sigma, \delta_2, q_2, F_2)$ *forma uma decomposição de identificação de estados (SI-Decomposition) de um DFA* $A = (K, \Sigma, \delta, q_0, F)$*, se existe um mapeamento* $\beta : K_1 \times K_2 \to K$*, onde* $\beta(\delta_1(q_1, w), \delta_2(q_2, w)) = \delta(q_0, w)$ *para todo* $w \in \Sigma^*$.

Definição 14 *(GAŽI; ROVAN, 2008) Um par de DFAs (A1, A2), onde* $A1 = (K_1, \Sigma, \delta_1, q_1, F_1)$ *e* $A2 = (K_2, \Sigma, \delta_2, q_2, F_2)$ *forma uma decomposição de identificação de aceitação fraca (wAI-Decomposition) de um DFA* $A = (K, \Sigma, \delta, q, F)$*, se existe uma relação* $R \subseteq K_1 \times K_2 \to K$*, onde* $R(\delta_1(q_1, w), \delta_2(q_2, w)) \iff w \in L(A)$ *para todo* $w \in \Sigma^*$.

Definição 15 *(GAŽI; ROVAN, 2008) Uma conexão paralela de dois DFAs* $A1 = (K_1, \Sigma, \delta_1, q_1, F_1)$ *e* $A2 = (K_2, \Sigma, \delta_2, q_2, F_2)$ *é o DFA* $A = (K_1 \times K_2, \Sigma, \delta, (q_1, q_2), F_1 \times F_2)$ *onde* $\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$.

O trabalho de Rovan e Sádovskỳ (2018) estende essas definições para NFAs.

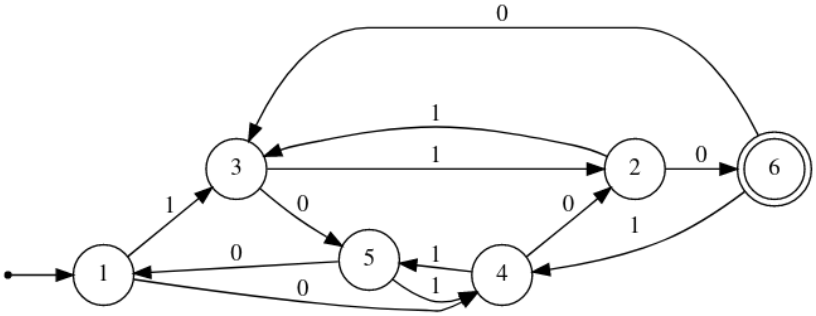
A abordagem utilizada é que: dada uma linguagem L, onde A seja um autômato tal que L = L(A), podemos dividir A em partições possivelmente mais simples que irão ser equivalentes ao autômato original.

Uma partição π de um conjunto não-vazio A é um conjunto de subconjuntos de A onde cada elemento de A pertença a um e somente dos conjuntos de π e cada elemento de π é não vazio. De acordo com Gaži e Rovan (2008), uma partição π de um conjunto de estados de um DFA $A = (K, \Sigma, \delta, q_0, F)$ é dita ter **propriedade de substituição** se:$\forall p, q \in K$:

$$p \equiv_\pi q \Rightarrow (\forall a \in \Sigma; \delta(p, a) \equiv_\pi \delta(q, a)), \tag{3.1}$$

onde $\equiv_\pi$ indica que dois elementos pertencem à mesma parte da partição. Ou seja uma partição π tem a propriedade de substituição se, para toda entrada, é possível mapear estados de π em estados de π. Por exemplo, na Figura 53 é possível ver que as partições $\pi_1 = \{\langle 1, 2, 3\rangle; \langle 4, 5, 6\rangle\}$, por exemplo, tem a propriedade de substituição, pois $\delta(1,1) = 3$, $\delta(3,1) = 2$, $\delta(2,1) = 3$ e $\delta(1,0) = 5$, $\delta(3,0) = 6$ e $\delta(2,0) = 6$, logo ao separar o autômato nessas duas partições, se uma transição sai de um estado do primeiro conjunto para outro estado do primeiro conjunto, todos os estados do mesmo conjunto, ao lerem a mesma entrada, irão pra um destino do primeiro conjunto, se ela sair do conjunto, as transições de todos os estados lendo o mesmo conjunto também saem. Um autômato pode ter mais de uma partição que tenha propriedade de substituição.

Figura 53 – Autômato exemplo - partições



Fonte: Adaptado de (MULLIN, 1969)

Percebe-se que cada máquina A_{π_1} e A_{π_2}, que são as máquinas imagem de π_1 e π_2 respectivamente, faz apenas uma parte da computação de A e que operadas em conjunto elas formam justamente a máquina A pois, segundo Mullin (1969), cada bloco de π_1 tem pelo menos um estado de A em comum com π_2.

3.5 Tabela comparativa

Tabela 4 – Tabela Comparativa

Trabalho	Abordagem	Decomposição de autômatos	Mineração de Processos
Aalst *et al.* (2006)	Transforma logs de eventos em sistemas de transições e utilizando teoria das regiões constroem uma rede de Petri equivalente	Não	Sim
Kalenkova *et al.* (2014)	Transforma logs de eventos em sistemas de transições e decompõe o sistema em uma parte regular e uma parte especial para depois construir uma rede de Petri com teoria das regiões	Sim	Sim
Gaži e Rovan (2008)	Utiliza decomposição de autômatos finitos determinísticos para resolver o problema de resolução assistida	Sim	Não
Rovan e Sádovskỳ (2018)	Utiliza decomposição de autômatos finitos não-determinísticos para resolver o problema de resolução assistida	Sim	Não
Monteiro e Oliveira (1998)	Utiliza decomposição de autômatos finitos não-determinísticos para reduzir a mudança de atividades em circuitos de sequências lógicas	Sim	Não
Este trabalho	Simplificação de autômatos gerados a partir de processos utilizando propriedades de autômatos e modularização de processos	Sim	Sim

Fonte: Próprio autor (2023)

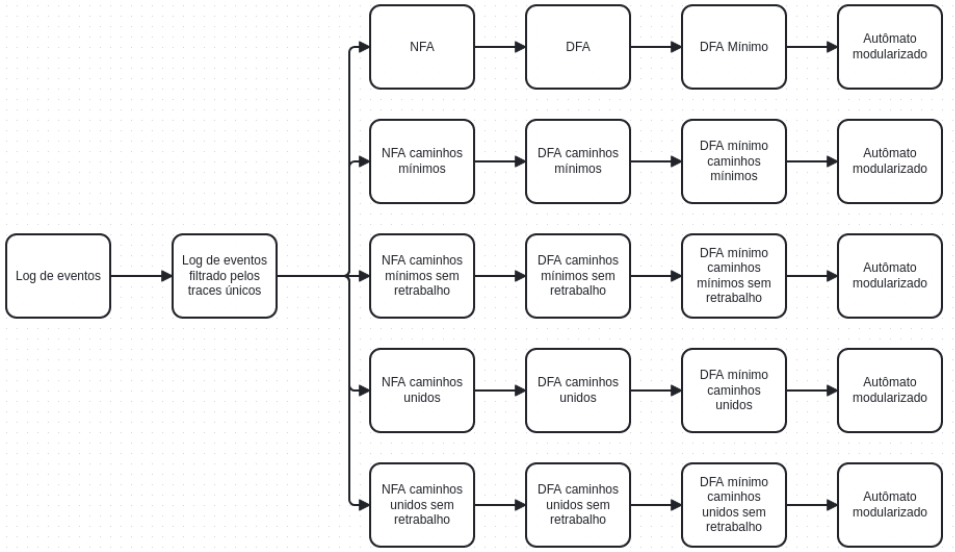
4 SIMPLIFICAÇÃO DE MODELOS DE PROCESSOS UTILIZANDO PROPRIEDADES DE AUTÔMATOS E MODULARIZAÇÃO

Neste trabalho, são propostas algumas formas de simplificação de modelos de processos representados por autômatos. Tais formas são as seguintes:

- Filtra-se o *log* de eventos pelos seus *traces* únicos e o representamos como um NFA, encontra-se seu DFA equivalente, minimiza-se o DFA encontrado e, por fim, modulariza-se suas sequências unitárias;
- Filtra-se o *log* de eventos pelos seus *traces* únicos e o representamos como um NFA com caminhos mínimos, encontra-se seu DFA equivalente, minimiza-se o DFA encontrado e, por fim, modulariza-se suas sequências unitárias;
- Filtra-se o *log* de eventos pelos seus *traces* únicos e o representamos como um NFA com caminhos mínimos sem retrabalho, encontra-se seu DFA equivalente, minimiza-se o DFA encontrado e, por fim, modulariza-se suas sequências unitárias;
- Filtra-se o *log* de eventos pelos seus *traces* únicos e o representamos como um NFA com caminhos mínimos unidos, encontra-se seu DFA equivalente, minimiza-se o DFA encontrado e, por fim, modulariza-se suas sequências unitárias;
- Filtra-se o *log* de eventos pelos seus *traces* únicos e o representamos como um NFA com caminhos mínimos unidos sem retrabalho, encontra-se seu DFA equivalente, minimiza-se o DFA encontrado e, por fim, modulariza-se suas sequências unitárias;

A Figura 54 demonstra o passo-a-passo feito.

Figura 54 – Passo-a-passo da proposta deste trabalho



Fonte: Próprio autor (2023)

Também é proposta uma forma de conversão de autômatos para modelos em BPMN. Nas subseções abaixo estão descritas como realizamos a proposta.

4.1 Representando sistemas de transições como autômatos finitos

Os sistemas de transições podem ser representados como autômatos finitos. Mostraremos como transformar um *log* de eventos em um autômato finito não-determinístico para representar o controle de fluxo de um processo. A abordagem codifica um *log* em um sistema de transições utilizando a estratégia de prefixo completo.

Definição 16 *Seja L um log de eventos para um dado conjunto de atividades A.*

- *Definimos $s_{i,j}$ como a posição de σ_j^i no trace σ^i em L como o número*

$$s_{i,j} = \left(\sum_{t=0}^{i-1} |\sigma^t| \right) + j \tag{4.1}$$

- *Definimos Θ_j como o conjunto de eventos na j-ésima posição em cada trace em L*

$$\Theta_j = \bigcup_{i=1}^{k} \sigma_j^i \tag{4.2}$$

Podemos codificar qualquer log de eventos em autômatos finitos não-determinísticos com ε-transições $N_L = (Q_L, A, \delta_L, 0, F_L)$ como se segue:

- *$Q_L = \{0, 1, \cdots, n\}$ é o conjunto de estados onde n é o tamanho do log de eventos L;*
- *A é o conjunto de atividades;*
- *0 é o estado inicial ($0 \in Q_L$);*
- *Para cada $\sigma^i = \sigma_0^i, \sigma_1^i, \cdots, \sigma_m^i$ adiciona-se o estado $s_{i,m}$ como estado final ($s_{i,m} \in F_L$);*
- *A função de transição δ_L se dá da forma que se segue:*
  - *Para cada $a \in \Theta_1$, temos:*

$$\delta_L(0, a) = \bigcup_{i=1}^{k} s_{i,1} \tag{4.3}$$

onde k representa o tamanho de Θ_1

  - *Para cada $\sigma = \sigma_0^i, \sigma_1^i, \cdots, \sigma_m^i$ em L e para cada $m > j \geq 1$, temos:*

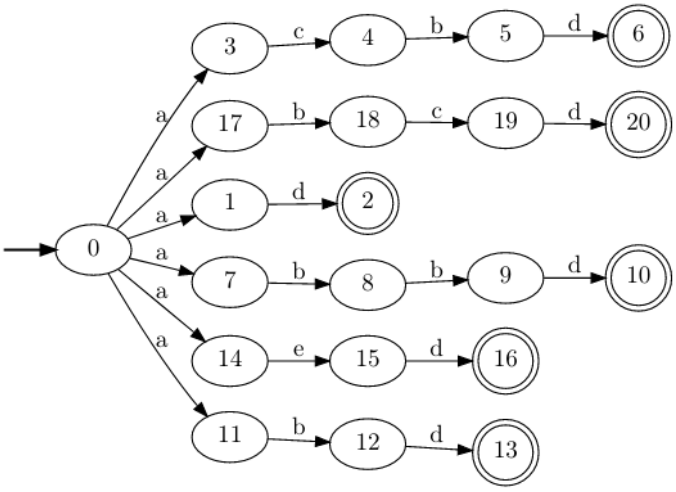
$$\delta_L(s_{i,j}, \sigma_j^i) = \{s_{i,j} + 1\} \tag{4.4}$$

.

Por exemplo, seja $A = \{a,b,c,d,e\}$ um conjunto de atividades. Seja um log de eventos $L = \langle\langle a,b\rangle\langle a,c,b,d\rangle\langle d,e\rangle\rangle$. Temos que $\sigma^1 = \langle a,b\rangle$, $\sigma^2_3 = b$, $s_{2,3} = 2$ e $\theta_2 = \{b,c,e\}$.

Na Figura 55, temos um exemplo de NFA construído para o *log* de eventos $L_1 = \{\langle a,d\rangle, \langle a,c,b,d\rangle, \langle a,b,d\rangle, \langle a,e,d\rangle, \langle a,b,c,d\rangle, \langle a,b,b,d\rangle\}$.

Figura 55 – NFA gerado a partir de L_1



Fonte: Próprio autor (2021)

Teorema 9 *Seja L um log de eventos e N_L um ε-NFA para L. A linguagem de $L(N_L)$ é o conjunto de traces de L.*

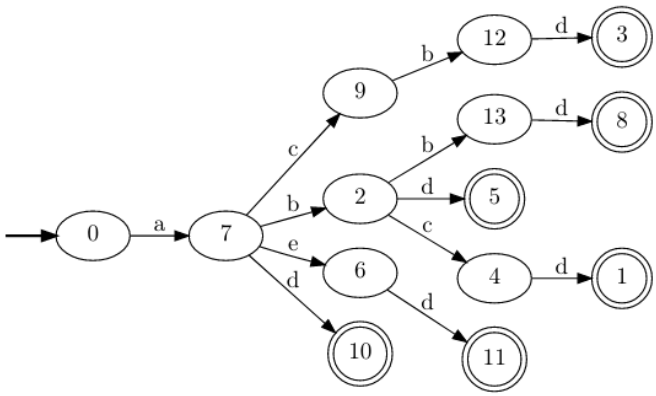
A prova deste teorema se dá diretamente pela codificação e pode ser mostrada por indução no tamanho de um *trace*.

Desde que um *log* de eventos pode ser codificado como um NFA e todo NFA possui um DFA equivalente, podemos encontrar um DFA mínimo, ou seja, um DFA que contém o menor número de estados possíveis, que seja equivalente ao NFA, como diz o Colorário abaixo.

Colorário 1 *Seja $\mathscr{L}$ um log de eventos. Existe um autômato finito determinístico mínimo $D^{min}_{\mathscr{L}}$ na qual $L(D^{min}_{\mathscr{L}})$ é o conjunto de traces de $\mathscr{L}$.*

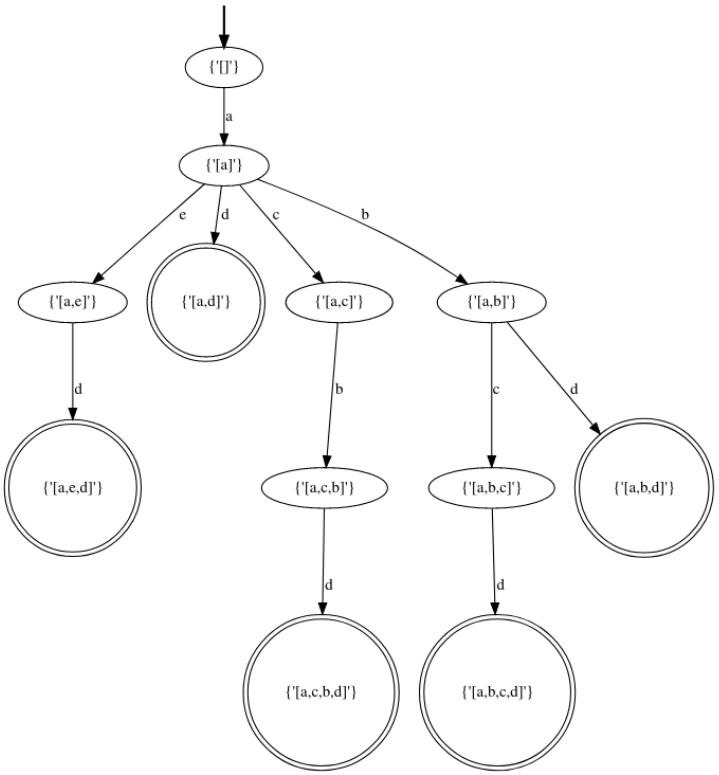
A Figura 56 nos mostra o DFA equivalente ao NFA da Figura 55. Nota-se que o DFA gerado é equivalente ao DFA gerado utilizando a abordagem de Aalst (2016), que define um estado baseado no seu histórico completo, apresentada na Sub-seção 2.2.1. A Figura 57 apresenta o DFA gerado utilizando o histórico.

Figura 56 – DFA de L_1



Fonte: Próprio autor (2021)

Figura 57 – DFA gerado utilizando histórico completo



Fonte: Próprio autor (2021)

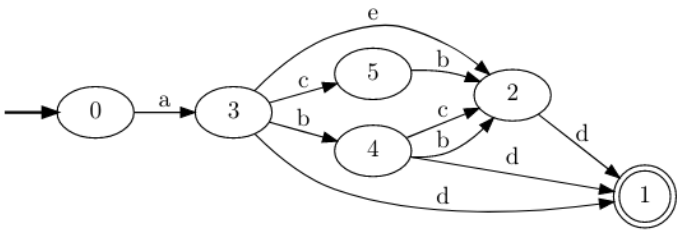
Nos dois casos, o DFA mínimo gerado é igual e é mostrado na Figura 58

4.1.1 Caminhos felizes

Para entendermos o conceito de caminhos felizes, é importante entendermos primeiro o que é **retrabalho** e o que é **repetição**. A Figura 59 exemplifica melhor esses conceitos.

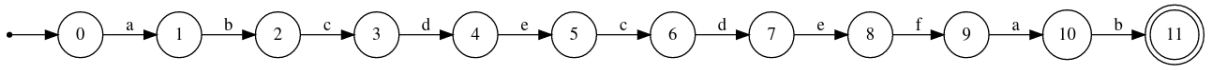
Tanto retrabalho quanto repetição são sequências de atividades repetidas, porém, o que difere um conceito do outro é quando a repetição acontece. Vemos na Figura 59 que a

Figura 58 – DFA mínimo de L_1



Fonte: Próprio autor (2021)

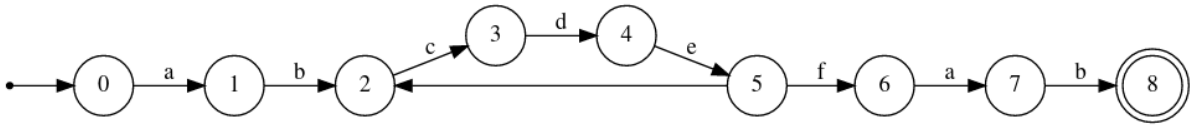
Figura 59 – Exemplo de processo com repetições e retrabalho



Fonte: Próprio autor (2023)

sequência de atividades “ab” aparece duas vezes no processo, mas como aparece no começo e no fim do processo, com várias outras atividades entre ela, dizemos que “ab” é uma repetição, pois é essencial para o funcionamento do processo. Quando encontramos uma sequência que se repete consecutivamente, ou seja, após a sua ocorrência, ela já ocorre novamente no processo, podemos considerar que houve um retrabalho neste processo e que o número de vezes que a sequência se repete é a quantidade de retrabalho explícito no *log*. Podemos ver que, neste exemplo, a sequência “cde” se repete consecutivamente sendo assim considerada retrabalho. Podemos representar o mesmo processo da maneira mostrada na Figura 60, pois o modelo ainda irá representar o retrabalho, além de permitir que ele ocorra mais de uma vez ou nenhuma vez, o que melhora sua generalização.

Figura 60 – Exemplo de processo com repetições e retrabalho



Fonte: Próprio autor (2023)

Para remover as sequências de retrabalho de um *trace* l, realizamos os seguintes passos:

- Para cada atividade x_i, onde i representa a posição da atividade no *trace*, verificamos se existe uma atividade y_k, onde k representa a posição da atividade no *trace*, na qual $x_i = y_k$ mas $i \neq k$;
- Se sim, então verificamos se para cada atividade nas posições $i+1, i+2, \cdots, k-1$ são

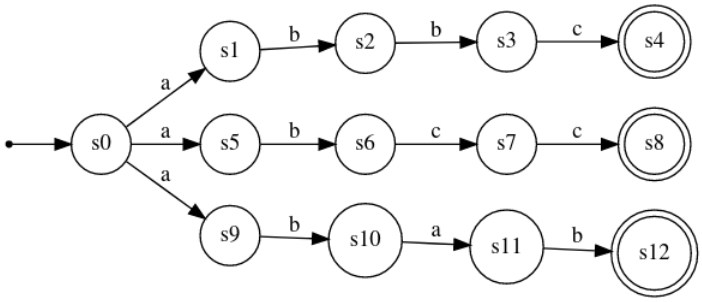
exatamente iguais às atividades das posições $k+1, k+2, \cdots, k+(k-i)$;

- Se sim, o novo *trace* será formado pelas atividades das posições menores que k e maiores que $k+(k-i)$.

Definição 17 *No contexto de processos, chama-se de "caminho feliz" o caminho do modelo construído a partir de um trace sem retrabalho, ele indicará a sequência de atividades que irão ocorrer sem que haja alguma repetição consecutiva de sequências tarefas.*

Por exemplo, seja um *log* de eventos $L = \langle a,b,b,c\rangle$, $\langle a,b,c,c\rangle$, $\langle a,b,a,b\rangle$. O NFA gerado a partir de L é mostrado na Figura 61.

Figura 61 – NFA referente ao *log* L

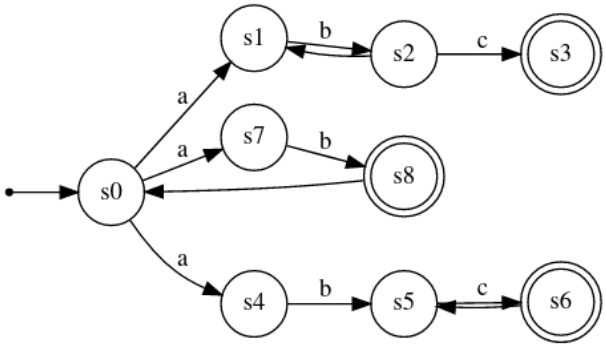


Fonte: Próprio autor (2023)

Cada *trace* é analisado procurando repetições consecutivas. Começando pelo *trace* $\langle a,b,b,c\rangle$, observa-se que "a" não se repete em nenhum momento, então parte-se para próxima atividade, "b", e é possível notar que ela é repetida consecutivamente uma vez, o que caracteriza retrabalho, após essa repetição, observa-se a última atividade "c" que não se repete em nenhum momento, desta forma, a única sequência encontrada é a sequência de "b". Assim, é possível representar esse *trace* no modelo de forma que, ao invés de explicitar a sua repetição, adiciona-se uma transição vazia do estado que recebe o final da cadeia para o estado que provê o início da cadeia enquanto que o estado que recebe o fim da cadeia continua provendo as transições necessárias para a modelagem do *trace*. Para o *trace* $\langle a,b,c,c\rangle$, a abordagem é a mesma. Para o *trace* $\langle a,b,a,b\rangle$ verifica-se que a primeira atividade "a" é repetida no *trace*, mas não consecutivamente, então analisa-se se a próxima atividade após o primeiro "a" e sua repetição são iguais, nos dois casos após "a", temos "b" e após o primeiro "b", já tem-se a repetição de "a", o que caracteriza uma sequência consecutiva de repetição, assim, ao modelar este *trace*, é adicionada uma transição vazia do estado que recebe o final da primeira cadeia para o estado que

provê o início da primeira cadeia, como não há mais atividades após a repetição, o estado que recebe o fim da cadeia será um estado final. A Figura 62 mostra o resultado desta operação.

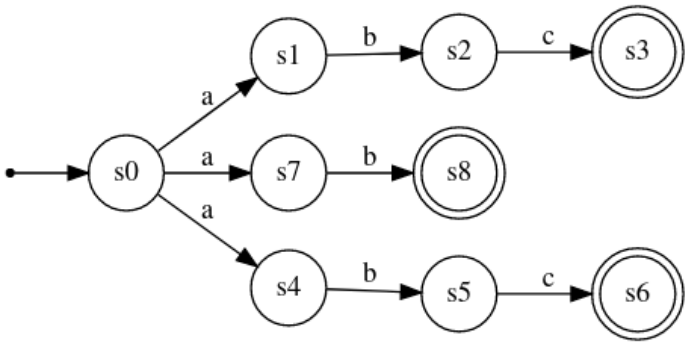
Figura 62 – NFA referente ao *log* L com caminhos mínimos



Fonte: Próprio autor (2023)

Nota-se que há uma redução no número de estados e uma generalização do modelo, que aceita muitos comportamentos a mais do que os observados no *log* original. Ainda assim, pode-se querer representar apenas o comportamento do processo sem que haja nenhuma repetição de ação, ou seja, o "caminho feliz", para isso, são removidos os retrabalhos do *log* de forma que o novo *log* L será formado pelos *traces* $L = \langle a,b,c\rangle, \langle a,b,c\rangle, \langle a,b\rangle$. O modelo que representa L é apresentado na Figura 63.

Figura 63 – NFA referente ao *log* L com caminhos mínimos sem retrabalho

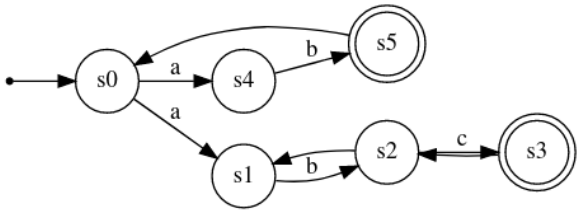


Fonte: Próprio autor (2023)

Também pode-se ver que dois *traces* de L, ao remover o retrabalho, se tornam idênticos. São eles $\langle a,b,b,c\rangle$ e $\langle a,b,c,c\rangle$, que se tornam $\langle a,b,c\rangle$. Assim, é possível juntá-los em um único caminho do modelo, aumentando ainda mais a generalização, como mostra a Figura 64.

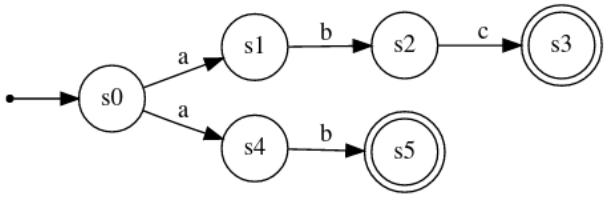
E então é possível remover todo o retrabalho, com o mesmo procedimento realizado para remover o retrabalho do *log* que gera o modelo da Figura 63. O *log* resultante é $L = \langle a,b,c\rangle, \langle a,b\rangle$, e o modelo o representando é apresentado na Figura 65.

Figura 64 – NFA referente ao *log* L com os caminhos mínimos iguais unidos



Fonte: Próprio autor (2023)

Figura 65 – NFA referente ao *log* L com os caminhos mínimos iguais unidos sem retrabalho



Fonte: Próprio autor (2023)

4.1.2 Building Blocks

Building blocks (RODGER *et al.*, 2006) são máquinas que têm um propósito específico e que podem ser usadas como componentes de outras máquinas, agindo como sub-máquinas. De acordo com Rodger *et al.* (2006), *building blocks* são muito utilizadas em Máquinas de Turing (HOPCROFT *et al.*, 2006), mas neste trabalho iremos estender o seu uso para máquinas que não possuem saídas, como é o caso dos autômatos já mostrados na Seção 2.1. Desta forma, a nova definição de um NFA_BB se dá desta forma:

$$M = (Q, \Sigma, \delta, q_0, F, l, n_p, BB)$$

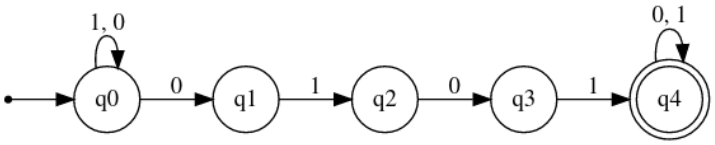
onde:

- Q é um conjunto finito de estados;
- Σ é um conjunto finito representando o alfabeto;
- q_0 é o estado inicial e $q_0 \in Q$;
- F é o conjunto finito de estados finais e $F \subseteq Q$;
- $\delta = Q \times \Sigma \rightarrow 2^Q$ é uma função de transição que recebe como argumento um estado e uma entrada do alfabeto e retorna um conjunto de estados;
- l é um nome (*label*) do NFA_BB;
- n_p é um NFA_BB na qual este NFA_BB pertence. Dizemos que n_p é o autômato pai de M;
- BB é um conjunto finito de NFA_BB, na qual para cada $bb \in BB$, temos que o *label* de $bb \in Q$.

Quando a computação de uma entrada atinge um estado que representa um sub-máquina, ela vai para a sub-máquina a partir do caractere que foi lido para chegar no estado, se a computação chegar em um estado de aceitação da sub-máquina então ela para a computação, voltando assim para a máquina pai. Tal recurso será necessário para representar os subprocessos condensados que serão encontrados, tornando o autômato mais legível.

A Figura 66 nos mostra um exemplo de NFA que não possui *building blocks* e que reconhece a palavra "x0101y", onde $x \in \Sigma^*$ e $y \in \Sigma^*$.

Figura 66 – NFA sem *Building-Blocks* que reconhece a palavra x0101y



Fonte: Próprio autor (2023)

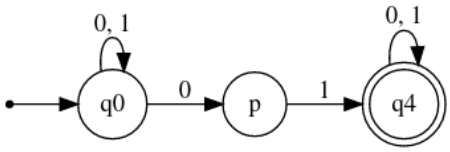
Utilizar *building blocks* permite construir outro autômato que reconhece a mesma linguagem, e que é definido desta forma:

- $Q = \{q0, p, q4\}$;
- $\Sigma = \{0, 1\}$
- q_0 é o estado inicial;
- $F = \{q_4\}$;
- $\delta(q_0, 0) = \{q_0, p\}$, $\delta(q_0, 1) = \{q_0\}$, $\delta(p, 0) = \emptyset$, $\delta(p, 1) = \{q_4\}$, $\delta(q_4, 0) = \{q_4\}$, $\delta(q_4, 1) = \{q_4\}$;
- n é o nome;
- $BB = \{p\}$, onde p é definido por:
  - $Q_p = \{q1, q2, q3\}$;
  - $\Sigma_p = \{0, 1\}$;
  - $q1$ é o estado inicial;
  - $F_p = q3$;
  - $\delta_p(q1, 0) = \emptyset,\ \delta_p(q1, 1) = \{q2\},\ \delta_p(q2, 0) = \{q3\},\ \delta_p(q2, 1) = \emptyset,\ \delta_p(q3, 0) = \emptyset,\ \delta_p(q3, 1) = \emptyset$;
  - p é o nome;
  - $BB_p = \emptyset$.

Tal autômato e sua sub-máquina são mostrados nas Figuras 67 e 68. Podemos ver que o estado "p" representa uma sub-máquina que lê a sequência "10". Ao alcançar o estado "p", a computação

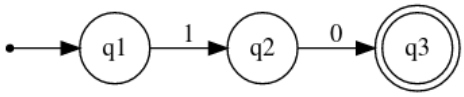
dará início à sub-máquina referente, e ao acabar a computação da sub-máquina, irá retornar ao processo principal. Nota-se que a sequência lida por "p" é uma subsequência da palavra reconhecida pelo NFA da Figura 67.

Figura 67 – NFA com *Building-Blocks* que reconhece a palavra x0101y



Fonte: Próprio autor (2023)

Figura 68 – *Building-Blocks* referente ao estado "p"



Fonte: Próprio autor (2023)

4.2 Concatenação

Uma das operações de autômatos que podem ser utilizadas para gerar um novo autômato é a concatenação. Na proposta deste trabalho, será feita uma decomposição de um autômato baseada na concatenação de autômatos menores, ou seja, quando se encontra os autômatos que foram concatenados para formar o autômato original, os consideramos como subprocessos. Para isso, serão consideradas apenas as concatenações de cadeias unitárias, que são cadeias de estados onde cada estado terá no máximo uma transição chegando nele e uma transição saindo dele.

Seja A um NFA, onde $D = (Q, \Sigma, \delta, q_0, F)$, NE(q) uma função que retorna o número de transições que saem do estado q, e NC(q) o número de transições que chegam no estado q. Definimos um **corte** como um subconjunto de estados onde a interseção entre cada corte será vazia. Seja A um conjunto de cortes, onde cada corte começa vazio:

- Para cada $q_n \in Q$:
  - Se $0 \leq \text{NE}(q_n) < 2$, $0 \leq \text{NC}(q_n) < 2$, e q_n não está incluído em nenhum corte, então adiciona q_n em A_z, onde $A_z \in A$ e, se $\text{NE}(q_n) > 0$, verifica para o estado q_m, onde $q_m = \delta(q_n, x)$, sendo x um caractere qualquer de Σ:
    - Se q_m também satisfaz a condição, então o estado também é incluído em A_z, e verifica para o estado de destino de q_m, caso haja;

        * Senão, q_n será o estado final do corte A_z. Assim, a computação de A_z se encerra e se inicia a computação de A_{z+1};
    - Se o estado q_n for incluído em um corte vazio, ele se torna o estado inicial deste corte;
    - Ao fim, o autômato original irá ser formado pela concatenação dos cortes, da forma que $L(D) = L(A_0)L(A_1)...L(A_N)$. Tal concatenação se dá ligando cada corte com ε-transições.

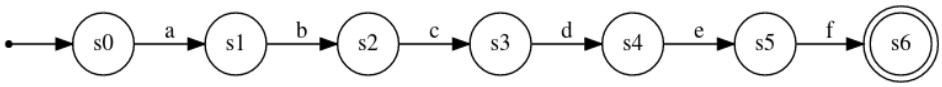
Por fim, para não ocorrer casos em que um número muito grande de estados sejam incluídos em um mesmo corte, tornando o resultado ainda ilegível, ou que um número muito pequeno de estados sejam incluídos em um corte, são criados um valor *min* e *max*, que representam o número mínimos e o número máximo de estados que um corte deve possuir, respectivamente. Assim, seja NA(a) o número de estados contidos no corte a:

- Para cada corte $A_n \in A$, verifica-se se NA(A_n) > *max*, se sim:
    - Partindo do primeiro estado do corte, para cada estado $s \in A_n$, adiciona-se s em um novo corte A_m
        * Se NA(A_m) = *max*, encerramos a computação de A_m e iniciamos a computação de A_{m+1}. Se A_n ainda possui estados que não estão em A_m, continua-se do estado s_{prox}, onde s_{prox} é o primeiro estado de A_n que não foi incluído em A_m;
        * Senão, se s é o último estado de A_n, verificamos se NA(A_m) $\geq$ *min*, senão descartamos esse corte.
        * Senão, verifica-se o estado s_{prox};
- Se NA(A_n) < *min*, descartamos esse corte.

Por exemplo, na Figura 69 temos um autômato que lê a sequência “abcdef”. Para encontrar os cortes deste autômatos, começamos a analisar a partir do primeiro estado, $s0$, como este estado é estado inicial, não o incluímos em nenhum corte, como $s0$ só tem uma transição partindo dele, analisaremos o seu único estado de destino, $s1$. Como $s1$ só tem uma transição chegando nele e uma transição partindo dele, o incluímos no corte $A_0 = \{s1\}$ e analisamos o seu destino, $s2$. $s2$ também cumpre todos os requisitos, então o incluímos no corte, $A_0 = \{s1, s2\}$ e analisamos o seu destino. $s3$ cumpre os requisitos, então o adicionamos no corte, $A_0 = \{s1, s2, s3\}$ e analisamos o seu destino. $s4$ cumpre os requisitos, então o adicionamos no corte, $A_0 = \{s1, s2, s3, s4\}$ e analisamos o seu destino. $s5$ cumpre os requisitos, então o adicionamos no corte, $A_0 = \{s1, s2, s3, s4, s5\}$ e analisamos o seu destino. Quando chegamos

no estado $s6$, verificamos que este estado é um estado de aceitação, logo ele não é incluído no corte e como $s5$ só possui uma transição partindo dele, não há mais estados alcançáveis desta sequência, então fechamos o corte.

Figura 69 – Exemplo de autômato com sequência



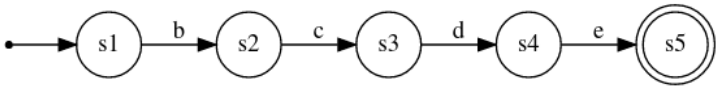
Fonte: Próprio autor (2023)

As figuras 70 e 71 nos mostram o autômato modularizado e a sua submáquina, respectivamente.

Figura 70 – Exemplo de autômato com sequência modularizada

Fonte: Próprio autor (2023)

Figura 71 – Sequência “Seq0” expandida



Fonte: Próprio autor (2023)

Um outro exemplo utilizando o mesmo autômato: diremos que o máximo e o mínimo de estados em cada corte é 2, então verificamos que o número de estados de A_0 é 5, que é maior que o tamanho máximo. Logo cria-se um novo corte $A_{0_1} = \{\}$, adicionamos o primeiro estado, encontrado da esquerda para a direita, da lista, $s1$, assim, $A_{0_1} = \{s1\}$; depois verificamos o próximo estado da lista, $s2$, como A_{0_1} só contém 1 estado, então podemos adicionar $s2$ em A_{0_1}. Dessa forma, $A_{0_1} = \{s1, s2\}$, e como A_{0_1} atingiu o tamanho máximo de estados, paramos a sua computação e iniciamos a de A_{0_2} continuando do próximo estado de A_0, $S3$. Incluímos $s3$ em A_{0_2}, assim, $A_{0_2} = \{s3\}$ e partimos para o próximo estado de A_0, $s4$; incluímos $s4$ em A_{0_2}, assim $A_{0_2} = \{s3, s4\}$, e como A_{0_2} atingiu o tamanho máximo, paramos a sua computação e iniciamos a de A_{0_3}, continuando do próximo estado de A_0, $s5$. Incluímos $s5$ em A_{0_3}, como não há mais estados em A_0 e A_{0_3} não atingiu o tamanho mínimo, descartamos esse corte. Dessa forma, temos $Q = \{s0, A_{0_1}, A_{0_2}, s5, s6\}$, onde:

- $\delta(s0, a) = A_{0_1}$;

- $\delta(A_{0_1}, c) = A_{0_2}$;
- $\delta(A_{0_2}, e) = s5$;
- $\delta(s5, f) = s6$;

4.3 Representando autômatos finitos como BPMN

Nesta Seção, será mostrada uma forma de traduzir os autômatos finitos gerados a partir de *logs* de eventos para modelos em BPMN. Nossa definição para um modelo em BPMN é uma simplificação da definição apresentada na Subseção 2.2.1. Definimos um modelo em BPMN da seguinte forma:

$$BPMN = \langle A, G, e_i, E_t, S, Z \rangle$$

onde:

- A é um conjunto de atividades;
- G é um conjunto de *gateways*;
- e_i é o evento inicial e $e_i \subseteq E$;
- E_t é o conjunto de eventos finais (terminais) e $\forall e_k \in E_t, e_k \subseteq E$;
- S é um conjunto de fluxos de sequências, onde $S \subseteq O \times O$, onde $O = A \cup \{e_i\} \cup E_t \cup G$. Representaremos o conjunto de fluxos utilizando a letra S para evitar confusão com o conjunto de estados finais de um autômato;
- Z é o conjunto de subprocessos.

Definição 18 *Seja um DFA $D = \langle Q, \Sigma, \delta, q_0, F \rangle$ e um BPMN $B = \langle A, G, e_i, E_t, S, Z \rangle$, a tradução de D para B se dá da seguinte forma:*

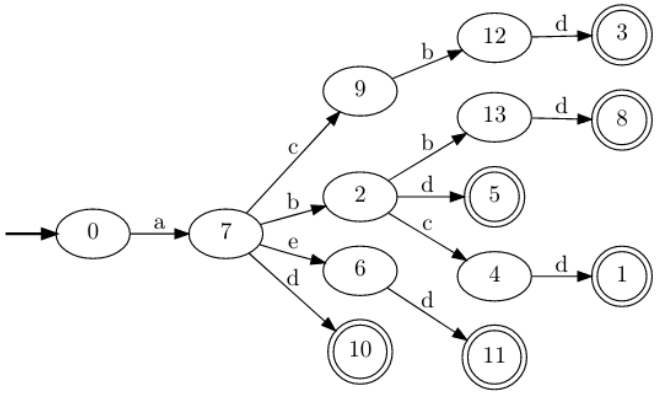
- $A = \Sigma$*;*
- G = $\{q \in Q$*, onde q é um estado onde há mais de uma transição saindo ou mais de uma transição chegando nele*$\}$*;*
- $\forall q, p \in Q$ *e* $\forall a \in \Sigma$*, se* $\delta(q,a) = p$ *então* $\langle q, a \rangle \in S$ *e* $\langle a, p \rangle \in S$
  - *No caso de haver apenas uma transição saindo e uma transição chegando no estado, então pode-se ligar as duas tarefas, ou seja, se* $\exists q, p, r \in Q$ *e* $\delta(q,a) = p$, $\delta(p,b) = r$ *sendo as únicas transições de p, então temos:* $\langle q, a \rangle$, $\langle a, b \rangle$*;*
- *Para o estado inicial, temos que* $\langle e_i, q_0 \rangle \in S$*;*
- $\forall q \in F$, $\langle q, e_q \rangle \in S$ *onde* e_q *é o evento final*

– *No caso de haver mais de um estado final, cria-se um evento final,* e_{qAux} *para cada estado final, onde* $\forall f \in F, \langle f, e_qAux \rangle$ *sendo* $e_{qAux} \in G$*;*

- $Z = \emptyset$.

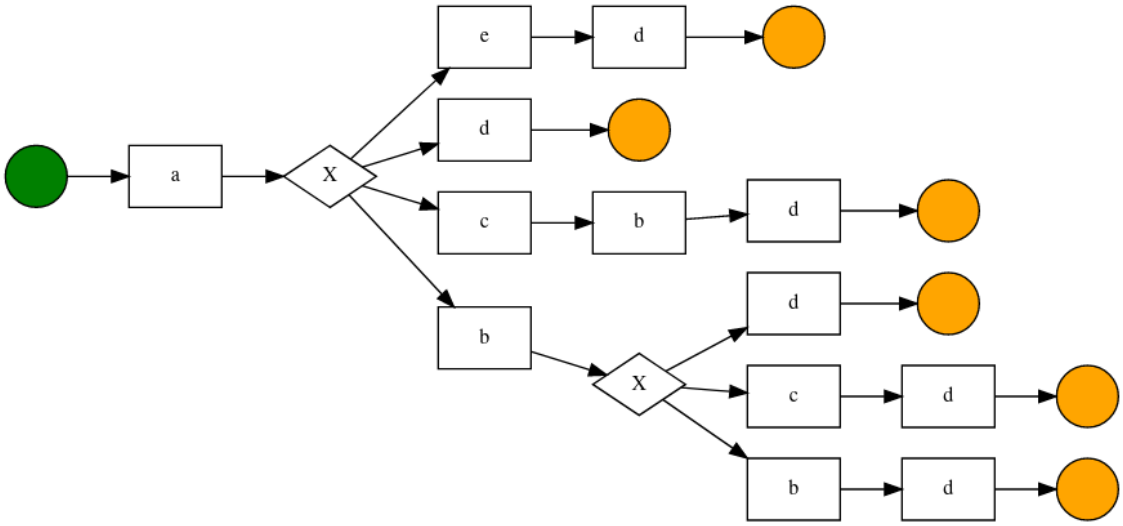
Utilizando essa definição, conseguimos realizar uma tradução dos DFAs mostrados na Seção 4.1, para modelos em BPMN. Por exemplo, para o autômato da Figura 72, temos o modelo apresentado na Figura 73.

Figura 72 – DFA da Figura 56



Fonte: Próprio autor (2021)

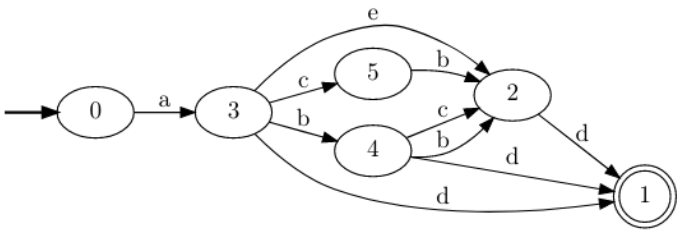
Figura 73 – Modelo em BPMN correspondente ao DFA da Figura 72



Fonte: Próprio autor (2023)

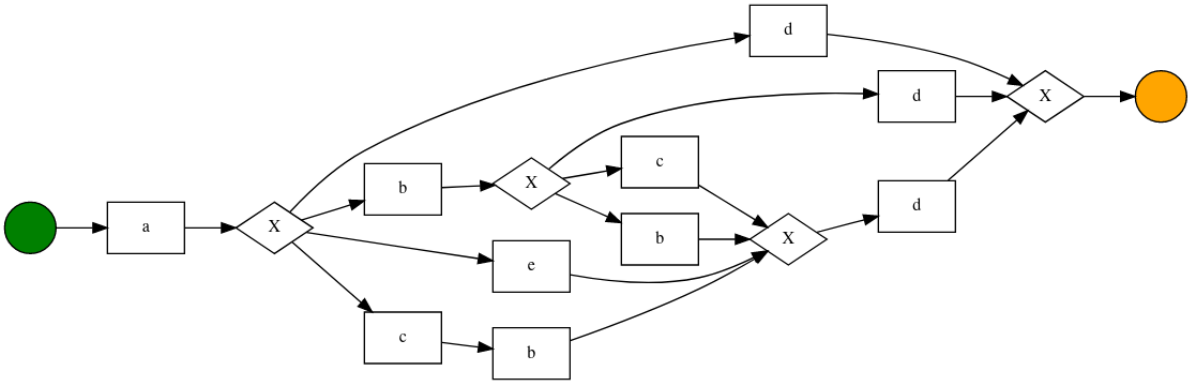
Para o modelo mostrado na Figura 74, temos o modelo em BPMN da Figura 75.

Figura 74 – DFA da Figura 58



Fonte: Próprio autor (2021)

Figura 75 – Modelo em BPMN correspondente ao DFA da Figura 58



Fonte: Próprio autor (2023)

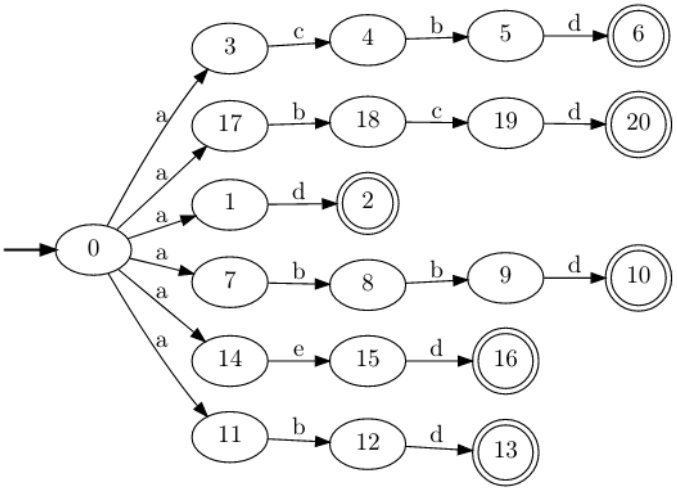
Definição 19 *Seja um NFA* $N = \langle Q, \Sigma, \delta, q_0, F, NFAs \rangle$ *e um BPMN* $B = \langle A, G, e_i, E_t, S, Z \rangle$*, a tradução de N para B se dá da seguinte forma:*

- $A = \Sigma$*;*
- G = $\{q \in Q$*, onde q é um estado onde há mais de uma transição saindo ou mais de uma transição chegando nele*$\}$*;*
- $\forall q, p \in Q$ *e* $\forall a \in \Sigma$*, se* $\delta(q,a) = p$ *então* $\langle q,a \rangle \in S$ *e* $\langle a,p \rangle \in S$
  - *No caso de haver apenas uma transição saindo e uma transição chegando no estado, então pode-se ligar as duas tarefas, ou seja, se* $\exists q, p, r \in Q$ *e* $\delta(q,a) = p$*,* $\delta(p,b) = r$ *sendo as únicas transições de p, então temos:* $\langle q,a \rangle \in S$*,* $\langle a,b \rangle \in S$*;*
  - *No caso de haver mais de um estado de destino para a mesma transição, então cria-se um gateway exclusivo auxiliar que irá ligar a task aos destinos. Por exemplo: se* $\exists q, p, r \in Q$ *e* $\delta(q,a) = p, r$*, então teremos* $\langle q,a \rangle \in S$*,* $\langle a, q_{aux} \rangle \in S$*, onde* $q_{aux} \in G$*,* $\langle q_{aux}, p \rangle \in S$ *e* $\langle q_{aux}, r \rangle \in S$*,*
- *Para o estado inicial, temos que* $\langle e_i, q_0 \rangle \in S$*;*
- $\forall q \in F$*,* $\langle q, e_q \rangle \in S$ *onde* e_q *é o evento final*

  - *No caso de o estado final q ter apenas uma transição de entrada, então faz-se* $\langle x, e_q \rangle \in S$*, onde* $\delta(w,x) = q$ *sendo* $w \in Q$ *e* $x \in \Sigma$*;*
  - *No caso de haver mais de um estado final, cria-se um gateway exclusivo auxiliar, onde* $\forall f \in F, \rangle f, q_{aux} \langle\ e\ \rangle q_{aux}, e_q \langle$*, sendo* $q_{aux} \in G$*.*
- *Cada submáquina em NFAs, será representada como um subprocesso de BPMN, representado por uma bolinha branca. Os componentes da submáquina serão representado, utilizando as mesmas regras descritas acima, no mesmo processo do BPMN principal.*

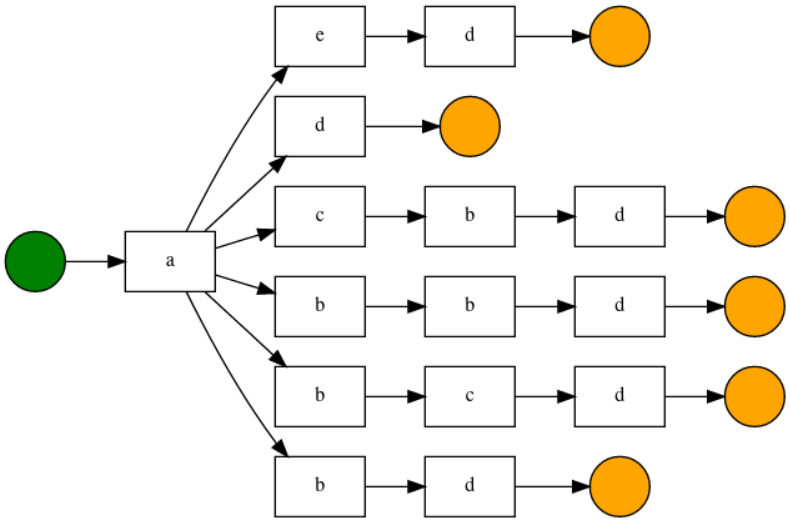
Utilizando essa definição, conseguimos realizar uma tradução do NFA mostrado na Figura 76, para o modelo em BPMN da Figura 81.

Figura 76 – NFA da Figura 55



Fonte: Próprio autor (2021)

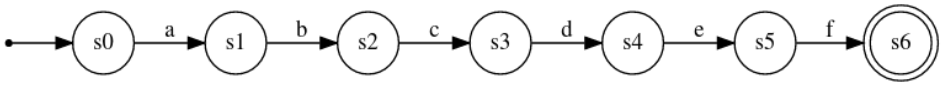
Figura 77 – BPMN correspondente ao NFA da Figura 76



Fonte: Próprio autor (2023)

Agora mostraremos o exemplo do autômato modularizado das Figuras 79 e 80, que, por sua vez, representam a modularização do autômato da Figura 78, traduzidos para o modelo BPMN da Figura 81. O autômato original, lê a cadeia “abcdef”, ao modularizarmos na Figura 81, criamos uma submáquina que lê a cadeia “bcdef”, desta forma o autômato modularizado, ou seja, utilizando essa submáquina, continua lendo a mesma linguagem.

Figura 78 – Autômato da Figura 69

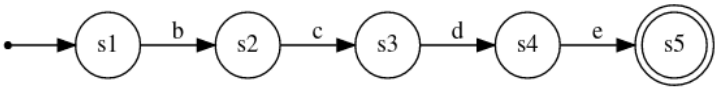


Fonte: Próprio autor (2023)

Figura 79 – Modularização da sequência do autômato da Figura 78

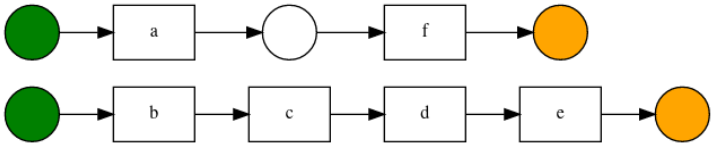
Fonte: Próprio autor (2023)

Figura 80 – Sequência “Seq0” expandida da Figura 79



Fonte: Próprio autor (2023)

Figura 81 – BPMN correspondente aos NFAs das Figuras 70 e 71



Fonte: Próprio autor (2023)

Na Figura 81, o modelo de cima representa o autômato modularizado, com o círculo representando o subprocesso. O modelo de baixo representa o subprocesso expandido.

5 EXPERIMENTOS

Neste capítulo serão apresentados os resultados dos experimentos feitos. Na Seção 5.1, é mostrado como são criados os autômatos utilizando a linguagem *Python* (MENEZES, 2016). Na Seção 5.2 apresenta-se o resultado da proposta com as sequências máximas tendo 25 estados, as mínimas tendo 3 estados e utilizando os dados das bases *Finale* (LIBRARY, 2022), *Prepaid Travel Costs* (LIBRARY, 2020), que são bases de dados utilizadas em competições de mineração de processos como a *BPI Challenge* (BPI..., 2020) e também utilizando dados de uma base de dados disponibilizada pela Secretaria da Fazenda do Ceará (SEFAZ-CE).

5.1 Representando autômatos finitos em *Python*

Para este trabalho, é utilizada a linguagem *Python* (MENEZES, 2016) e suas bibliotecas, tanto para criação e manipulação de autômatos, quanto para a mineração de processos.

Para representar autômatos finitos, utilizamos uma biblioteca chamada "teocomp" (VASCONCELOS; GUERRA, 2023). Nela temos classes para representar cada tipo de autômato, sendo eles DFA, NFA, ε-NFA (NFA com transições vazias) e NFA_BB (NFA com *building blocks*). Também temos funções como, por exemplo, *dfa.minimization()*, que retorna um DFA mínimo correspondente ao DFA que já temos e *nfa.determinization()*, que retorna um DFA correspondente ao NFA que já temos.

Para a construção de autômatos a partir de *logs* de eventos, utilizamos a biblioteca "*automaton2bpmn*", que pode ser encontrada no *link*: ⟨*https://github.com/daviromero/automaton2bpmn*⟩. Nela temos funções que constroem autômatos a partir de *logs* de eventos, como, por exemplo, *automaton2bpmn.to_automaton.to_nfa()* que irá construir um NFA baseado no *log* de eventos, *automaton2bpmn.to_automaton.to_nfa_minimum_path()* que irá construir um NFA com caminhos mínimos a partir do *log* de eventos (baseado na Seção 4.1.1). Esta biblioteca também é útil pois podemos converter os autômatos para modelos BPMN, usando as funções *automaton2bpmn.to_bpmn. dfa_to_bpmn()* e *automaton2bpmn.to_bpmn. nfa_to_bpmn()* que irão converter DFAs e NFAs em modelos BPMN, respectivamente, baseadas nos conceitos apresentados na Seção 4.1.2.

O algoritmo criado para construir um NFA a partir de um *log* de eventos está descrito no Algoritmo 1.

Figura 82 – Algoritmo 1

Algoritmo 1: *Log* de eventos para NFA

Input : Um *log* de eventos na forma de uma lista de *traces* e um *booleano* que diz se o NFA terá *building blocks*

Output : Dicionário representando um NFA

```
states = () acceptingStates = () alphabet = () transitions = dict()
initialState ← '0'
states.add(initialState)
i ← 1
foreach j ← 0 até |log| do
    foreach k ← 0 até |log[j]| do
        states.add(i)
        if k = |log[j]| then
            acceptingStates.add(i)
        end
        alphabet.add(log[j][k])
        if k=0 then
            transitions.add((0, log[j][k]), set(i))
        end
        else
            transitions.add((i-1, log[j][k]), set(i))
        end
        i ← i + 1
    end
end
```

Fonte: Próprio autor (2023)

O Algoritmo 2 descreve como acontece a determinimização de um NFA, utilizando o conceito de equivalência entre NFAs e DFAs apresentado na Seção 2.1.

Figura 83 – Algoritmo 2

Algoritmo 2: Determinização de NFA

```
Input  : Um NFA
Output : Dicionário representando um DFA que lê a mesma linguagem da entrada

initialStates = NFA.startState
if |initialStates| > 0 then
    states.add(NFA.initialStates)
end
queue.add(initialStates)
setStates.add(initialStates)
if setStates[0]∪NFA.acceptStates then
    acceptStates.add(setStates[0])
end
while queue nao esta vazia do
    atual ← queue.pop()
    foreach a ∈ NFA.alphabet do
        foreach state ∈ atual do
            if (state, a) ∈ NFA.transitions then
                proximos.add(destinos da transicao)
            end
        end
        if proximos nao esta incluida em setStates then
            setStates.add(proximos)
            queue.add(proximos)
            states.add(proximos)
            if proximos ∪ NFA.acceptStates then
                acceptStates.add(proximos)
            end
        end
        transition[atual, a] ← proximos
    end
end
```

Fonte: Próprio autor (2023)

O Algoritmo 3 irá construir um NFA mínimo que represente o *log* de eventos utilizando as noções de caminhos felizes apresentadas na Subseção 4.1.1.

Figura 84 – Algoritmo 3

Algoritmo 3: *Log* de eventos para NFA minimo

Input : Um *log* de eventos na forma de uma lista de *traces*, um *booleano* que diz se o NFA terá *building blocks*, um *booleano* que diz se haverá retrabalho ou nao

Output : Dicionário representando um NFA minimo

```
states = () acceptingStates = () alphabet = () transitions = dict()
initialState ← '0' states.add(initialState)
i ← 1
foreach j ← 0 até |log| do
    pos ← i
    trace ← remove todas as repeticoes em log[j]
    foreach k ← 0 até |trace| do
        states.add(i)
        if k = |trace − 1| then
            acceptingStates.add(i)
        end
        alphabet.add(log[j][k])
        if k=0 then
            transitions.add((0, log[j][k]), set(i))
        end
        else
            transitions.add((i-1, log[j][k]), set(i))
        end
        i ← i + 1
    end
    if retrabalho then
        destino ← pos + posicao do ultimo elemento repetido no trace
        if primeiro elemento repetido for o primeiro elemento do trace then
            transition[destino, ' '] ← set(0)
        end
        else
            posIni = pos + (posicao do primeiro elemento repetido -1)
            transition[destino, ' '] ← set(posIni)
        end
    end
end
```

Fonte: Próprio autor (2023)

Para nossa proposta de encontrar subprocessos, primeiramente, encontraremos as maiores sequências unitárias possíveis em NFAs, ou seja, sequências de estados que só tem uma transição chegando nele e uma transição saindo dele. Para isso utilizamos o Algoritmo 4.

Figura 85 – Algoritmo 4

Algoritmo 4: Encontra sequências unitárias em NFAs

```
Input  : Um NFA
Output : Lista de listas representando as sequências de estados unitários

foreach estado ∈ NFA.states do
    cor[estado] ← branco
end
pilha.add(NFA.startState)
while pilha não for vazia do
    estado ← topo da pilha
    cores[estado] ← cinza
    if estado tiver apenas uma saída e não for final nem inicial then
        sequencia.add(estado)
        proximos ← estados brancos destino da transição de estado
        if |proximos| > 0 then
            pilha.add(proximos)
        end
        else
            sequencias.add(sequencia)
            sequencia ← []
        end
    end
    else
        proximos ← estados brancos destino da transição de estado
        if |proximos| > 0 then
            pilha.add(proximos)
        end
        if |sequencia| > 0 then
            sequencias.add(sequencia)
            sequencia ← []
        end
    end
end
```

Fonte: Próprio autor (2023)

Então separamos as sequências de acordo com os parâmetro de tamanho máximo e mínimo das sequencias, como mostrado no Algoritmo 5.

Figura 86 – Algoritmo 5

Algoritmo 5: Separa sequências unitárias em NFAs

Input : Uma lista de listas representando as sequências unitárias, um inteiro representando o tamanho mínimo que as sequências devem ter, um inteiro representando o tamanho máximo que as sequências devem ter

Output : Lista de listas representando as novas sequências

```
foreach sequencia ∈ lista de sequencias do
    aux ← []
    if |sequencia| > MAX then
        foreach estado ∈ sequencia do
            aux.add(estado)
            if |aux| = MAX then
                Remove estados em aux que já representam sub-máquinas
                if |aux| ≥ MIN then
                    sequenciasSeparadas.add(aux)
                    aux ← []
                end
            end
        end
        if |aux| ≥ MIN then
            Remove estados em aux que já representam sub-máquinas
            if |aux| ≥ MIN then
                sequenciasSeparadas.add(aux)
                aux ← []
            end
        end
    end
end
```

Fonte: Próprio autor (2023)

Logo após, iremos colapsar essas sequências em sub-máquinas como mostrado no Algoritmo 6, utilizando o conceito de *building blocks* apresentado na Subseção 4.1.2.

Figura 87 – Algoritmo 6

Algoritmo 6: Colapsa sequências unitárias em NFAs

Input : Um NFA, uma lista de listas representando as sequências unitárias, um inteiro representando o tamanho mínimo que as sequências devem ter, um inteiro representando o tamanho máximo que as sequências devem ter

Output : NFA com as sequências colapsadas em sub-máquinas

```
sequenciasSeparadas ← separaSequencias(sequencias)
i ← 0
foreach sequencia ∈ sequenciasSeparadas do
    sub ← "seq"+i
    i ← i + 1
    estados.add(sub)
    foreach estado ∈ sequencia do
        estados.remove(estado)
        remove transições que saem de estado fazendo elas sairem de sub com os
          mesmos destinos lendo as mesmas coisas
        remove transições que chegam em estado fazendo elas chegarem em sub com os
          mesmos sorvedouros lendo as mesmas coisas
    end
    Np ← novo automato com os estados e transições que foram removidos
    Np.pai ← NFA
    Np.label ← sub
    Adiciona NP no dicionário de sub-máquinas de NFA com a chave sendo sub
end
NFAs ← maquinasIguais(NFAs)
```

Fonte: Próprio autor (2023)

A função do Algoritmo 7 irá verificar se há alguma submáquina contida em outra submáquina, ou seja, se a sequência lida por uma submáquina é uma subsequência lida por outra submáquina, ou então se as submáquinas são iguais, isto é, se a sequência lida por uma submáquina é exatamente a mesma lida por outra submáquina.

Figura 88 – Algoritmo 7

Algoritmo 7: Submáquinas iguais ou contidas

Input : submáquinas de NFA
Output : Dicionário de submáquinas de NFA

```
j ← 0
foreach sub ∈ sub-máquinas NFA do
    foreach sub2 ∈ sub-máquinas NFA do
        if sub = sub2 e sub != sub2 then
            sub2.estados ← sub.estados
            sub2.transições ← sub.transições
            sub2.label ← sub+j
            j ← j+1
        end
        else if sub ⊂ sub2 e sub != sub2 then
            remove os estados e transições iguais em sub2
            sub2.estados.add(sub+j)
            A transição que chegava no primeiro estado igual e a que saía do último
              estado igual agora vão para o estado sub+j
            Nj ← novo automato com os estados e transições de sub
            Nj.pai ← sub2
            Nj.label ← sub+j
            Adiciona Nj no dicionário de sub-máquinas da sub-máquina sub2 com a
              chave sendo sub+j
            j ← j+1
        end
    end
end
```

Fonte: Próprio autor (2023)

5.2 Experimentos

Nesta Seção serão abordados os resultados obtidos utilizando a proposta deste trabalho nas bases de dados *Finale* (LIBRARY, 2022), *Prepaid Travel Costs* (LIBRARY, 2020) e de um *log* disponibilizado pela Secretaria da Fazenda do estado do Ceará (SEFAZ - CE) (CEARá, 2017). Logo após iremos mostrar os resultados de outros algoritmos de descoberta na mesma base de dados, a fim de compará-los. Para comparar os diferentes modelos obtidos, iremos construir modelos BPMN equivalentes aos sistemas de transições e depois os transformaremos em redes de Petri que sejam equivalentes aos modelos gerados, os modelos de Rede de Petri serão convertidos utilizando a função `pm4py.convert_to_petri_net` da biblioteca *pm4py*, enquanto que os autômatos serão convertidos utilizando as funções `nfa_to_bpmn`, `dfa_to_bpmn`

e `nfaBB_to_bpmn` da biblioteca *automaton2bpmn* e que têm como base os conceitos mostrados na Seção 4.3. Também será feita uma comparação dos nossos autômatos com os autômatos obtidos a partir das redes de Petri geradas pelos algoritmos, para isso, utilizaremos os grafos de alcançabilidade de cada modelo. Os objetos de análise serão: número de componentes e a acurácia de cada modelo.

Os testes foram realizados em um computador *Intel(R) Core(TM) i3-10110U CPU @ 2.10GHz* com o sistema operacional Ubuntu 22.04.2. Utilizamos o *google colab* (GOOGLE, 2023) que nos permite usar uma espécie de máquina virtual com 12,7GB de memória RAM e 107,7GB de disco. Para testar as acurácias dos modelos gerados, separamos a base de dados por *traces* em base de treino e base de teste, com uma proporção de oito para dois respectivamente, para isso, utilizamos a função `train_test_split` da biblioteca `sklearn.model_selection`, onde sempre embaralhamos os *traces* gerando bases de treino e de testes diferentes para cada *log*. Para o cálculo da acurácia dos modelos gerados sem retrabalho, realizamos a operação de remoção de retrabalho da base de teste baseado na descrição da Subseção 4.1.1, pois de outra forma o resultado sempre seria muito baixo e não representaria o objetivo de se ter um modelo construído sem retrabalho. Para todos os testes, utilizamos 100 por cento da base de treino para construir o modelo, ou seja, construímos o modelo baseado nos cem por cento mais frequentes *traces*.

5.2.1 Finale

Nesta Subseção serão abordados os resultados obtidos utilizando a proposta deste trabalho na base de dados *Finale*. A base de dados *Finale* possui 14 atividades, mostradas junto com suas respectivas frequências na Tabela 5.

Tabela 5 – Frequência das atividades da base de dados *Finale*

Atividades	**Frequência**
Assign seriousness	4938
Take in charge ticket	5060
Resolve ticket	4983
Closed	4574
Wait	1463
Create SW anomaly	67
Insert ticket	118
Schedule intervention	5
RESOLVED	2
INVALID	2
VERIFIED	3
Resolve SW anomaly	13
Require upgrade	119
DUPLICATE	1

Fonte: Próprio autor (2023)

O *log* de eventos também possui estas características:

- Número de *traces*: 4580;
- Media de tamanho dos *traces*: 4,66;
- Desvio padrão do tamanho dos *traces* 1,17;
- Número de *traces* únicos: 226;
- N° eventos: 21348.

Para termos uma noção da distribuição dos *traces* no *log* de eventos, a Tabela 6 nos mostra os *traces* únicos com a maior frequência no *log* e os *traces* com a menor frequência no *log*.

Tabela 6 – Frequência dos *traces* da base de dados *Finale*

Trace	**Frequência**	%
Trace 0	2366	51,65
Trace 1	552	12,05
Trace 2	228	4,97
Trace 3	213	4,65
Trace 4	164	3,58
...	...	...
Trace 221	1	0,02
Trace 222	1	0,02
Trace 223	1	0,02
Trace 224	1	0,02
Trace 225	1	0,02

Fonte: Próprio autor (2023)

Nota-se que a soma das frequências de todos os *traces* irá resultar no número total de *traces*.

Para construir o modelo e calcular a sua acurácia, separamos o *log* de eventos em base de treino e base de teste. A base de treino vai conter as atividades mostradas na Tabela 7.

Tabela 7 – Atividades da base de dados *Finale* - Treino

Atividades	**Frequência**
Assign seriousness	3612
Take in charge ticket	3899
Resolve ticket	3712
Closed	3662
Wait	794
Create SW anomaly	37
Insert ticket	107
Schedule intervention	4
RESOLVED	2
INVALID	2
VERIFIED	2
Resolve SW anomaly	4
Require upgrade	7

Fonte: Próprio autor (2023)

O número de traces da base de treino é 3664; a média de tamanho dos traces é 4,32; o desvio padrão do tamanho dos traces é 0,86; o número de traces únicos é 46 e o número de eventos: 15844.

A Tabela 8 nos mostra as atividades da base de testes.

Tabela 8 – Atividades da base de dados *Finale* - Teste

Atividades	**Frequência**
Assign seriousness	886
Take in charge ticket	869
Resolve ticket	920
Closed	897
Wait	452
Insert ticket	11
Require upgrade	96
Create SW anomaly	23
VERIFIED	1
Resolve SW anomaly	4
DUPLICATE	1
Schedule intervention	1

Fonte: Próprio autor (2023)

O número de traces da base de testes é 916; a média de tamanho dos traces é 4,54; o desvio padrão do tamanho dos traces é 0,80; o número de traces únicos é 49 e o número de eventos: 4161.

Para um melhor entendimento, apresentamos os dados das bases na Tabela 9.

Tabela 9 – Comparação das bases de dados

Base	**N° traces**	**Média tam traces**	**Eventos**	**Traces únicos**	**Eventos únicos**
Log original	4580	4,66	21348	226	1594
Base treino	3664	4,32	15844	46	1162
Base teste	916	4,54	4161	49	658

Fonte: Próprio autor (2023)

A Tabela 10 mostra os dados das máquinas de estados geradas a partir da base de dados após filtrarmos os *traces* únicos.

Tabela 10 – Máquinas de estados

N° de Traces únicos: 3664 Frequência: 1,0 N° Eventos: 16937						
Máquina de estados	**Atividades**	**Estados**	**Transições**	**Estados de Aceitação**	**Sub-Autômatos**	**Estados totais**
NFA	13	1163	1162	160	0	1163
DFA	13	535	534	160	-	-
DFA min	13	115	245	4	-	-
OP Seq 3-25	13	110	240	4	2	117
NFA CM	13	879	1044	160	0	879
DFA CM	13	469	821	142	-	-
DFA min CM	13	163	489	7	-	-
OP Seq 3-25	13	158	484	7	2	165
NFA sRet	13	879	878	160	0	879
DFA sRet	13	158	157	46	-	-
DFA min sRet	13	41	77	3	-	-
OP Seq 3-25	13	33	69	3	3	44
NFA join	13	276	346	46	0	276
DFA join	13	276	296	46	-	-
DFA min join	13	121	487	6	-	-
OP Seq 3-25	13	116	482	6	2	123
NFA sRet join	13	276	275	46	0	276
DFA sRet join	13	158	157	46	-	-
DFA min sRet join	13	41	77	3	-	-
OP Seq 3-25	13	33	69	3	3	44
Legenda:						
	CM: Caminhos mínimos	sRet: Sem retrabalho	OP Seq: Operação sequências			

Fonte: Próprio autor (2023)

Nesta tabela, cada linha é referente à uma máquina de estados. Importante destacar que geramos as máquinas não-determinísticas das linhas 0, 4, 8, 12 e 16 e que, para cada máquina não-determinística, as próximas três linhas são referentes às operações de determinização, minimização e modularização das máquinas. Na primeira coluna temos um identificador, na

segunda coluna temos o nome da máquina de estados, na terceira coluna temos o número de atividades do modelo, na quarta coluna temos o número de estados da máquina, na quinta coluna apresentamos o número de transições da máquina, na sexta coluna apresentamos o número de estados de aceitação da máquina, na sexta coluna apresentamos o número de Sub-Autômatos da máquina (quando a máquina é não determinística, quando é determinística será representada por um traço, indicando que não há modularização possível na máquina), por fim, na sétima coluna vemos o número de estados totais, ou seja, o número de estados da máquina somados com o número de estados das suas submáquinas (novamente, contando apenas nas máquinas não-determinísticas. Nota-se que na separação entre treino e teste, uma das atividades aparece exclusivamente na base de teste, cremos que seja a atividade *DUPLICATE* que aparece apenas uma vez no *log* original. No cabeçalho da tabela, temos representados os dados referentes ao em número de *traces* únicos, a porcentagem de *traces* utilizados e o número de eventos do *log*, considerando os *traces* únicos.

A Tabela 11 nos mostra o resultado das acurácias dos modelos. Para calcular a acurácia dos modelos, utilizamos três abordagens diferentes, a primeira abordagem, apresentada na segunda coluna da Tabela, consiste em transformar os *traces* da base de testes em uma lista de palavras de entrada e verificar se são aceitos pelos autômatos gerados a partir da base de treino e calculamos a porcentagem de aceitação. A segunda abordagem, representada na terceira coluna, é a abordagem *token based replay*, descrita na Subseção 2.2.3.1 e, na quarta coluna, é apresentado os valores da acurácia utilizando a abordagem *alignments*, apresentada na Subseção 2.2.3.2.

Utilizando a acurácia dos autômatos como base, podemos ver que os autômatos das linhas 11 e 19 são as máquinas com os menores números de componentes encontradas e também possuem a mesma acurácia e que tal acurácia é aceitável. Interessante notar que as operações de caminhos mínimos e de junção de caminhos (linhas 4 e 12, respectivamente) trazem uma generalização maior que o autômato não-determinístico da linha 0; também devemos perceber que os modelos sem repetição obtiveram as melhores acurácias.

Para as acurácias utilizando *token based replay* e *alignments* não foram calculadas as acurácias para os modelos modularizados pois as redes de Petri geradas não possuem nenhuma estrutura equivalente a uma sub-máquina. Nota-se que as acurácias diferem entre máquinas equivalentes e isso se deve por alguns motivos: os autômatos são transformados em modelos em BPMN, que por sua vez são transformados em redes de Petri, podendo gerar diferenças

Tabela 11 – Acurácias das máquinas de estados geradas a partir da base de dados *Finale*

Referente à	**Acurácia Autômatos**	**Token based Replay**	**Alignments**
Não-Determinística	88,864629	53,3071	97,4677
Determinística	88,864629	97,0183	97,4672
Determinística min	88,864629	71,4950	97,4657
Operação Sequências min/max:3-25 estados DFA min	88,864629	-	-
Não-Determinística caminho mínimo	91,484716	64,8966	98,0903
Determinística caminho mínimo	91,484716	20,7656	98,0902
Determinística min caminho mínimo	91,484716	32,0531	98,0886
Operação Sequências min/max:3-25 estados DFA min caminho mínimo	91,484716	-	-
Não-Determinística sem retrabalho	92,903930	63,4166	98,3466
Determinística sem retrabalho	92,903930	61,8439	98,3466
Determinística min sem retrabalho	92,903930	85,8224	98,3444
Operação Sequências min/max:3-25 estados DFA min sem retrabalho	92,903930	-	-
Não-Determinística caminhos unidos	92,467249	55,7882	98,2775
Determinística caminhos unidos	92,467249	34,8772	98,2809
Determinística min caminhos unidos	92,467249	55,8762	98,2801
Operação Sequências min/max:3-25 estados DFA min caminhos unidos	92,467249	-	-
Não-Determinística caminhos unidos sem retrabalho	92,903930	59,9256	98,3466
Determinística caminhos unidos sem retrabalho	92,903930	97,3517	98,3466
Determinística min caminhos unidos sem retrabalho	92,903930	92,4679	98,3444
Operação Sequências min/max:3-25 estados DFA min caminhos unidos sem retrabalho	92,903930	-	-

Fonte: Próprio autor (2023)

entre os modelos, pois dizemos que a conversão de um modelo A para um modelo B é aceitável se $L(A) \subseteq L(B)$. Outro motivo que alguns dos modelos não determinísticos tendem a ter uma acurácia menor é justamente o fato das redes de Petri não considerarem o não-determinísmo, de forma que em um NFA, se o último símbolo da entrada leva a um estado final, então a palavra é aceita, enquanto que nas redes de Petri, utilizando *token based replay* por exemplo, vão se inserindo *tokens* para cada caminho de uma mesma transição e, se ao final, esses *tokens* não forem consumidos, o nível de acurácia do modelo diminui. Por fim, na abordagem *token based replay*, há ainda o que se chama de inundação de *tokens*, que força o modelo a reconhecer o *trace* mesmo que diminua a acurácia um pouco, desta forma, alguns resultados podem estar sendo representados como melhores do que o que realmente são. Com a abordagem *alignments*,

percebemos que as acurácias das máquinas já são mais uniformes e desta vez sem a interferência da inundação de *tokens*, podemos ter um resultado mais confiável. Os modelos gerados tem uma boa aceitação dos dados da base de teste.

A fim de comparação, utilizamos os algoritmos de descoberta *alpha miner*, *heuristic miner* e *inductive miner* na mesma base de treino e verificamos a acurácia utilizando a função `pm4py.fitness_token_based_replay` da biblioteca *pm4py* nos casos de teste com o modelo gerado. O resultado pode ser conferido na Tabela 12.

Tabela 12 – Tabela de comparação dos modelos acurácia *token based replay - Finale*

Modelo	**Places**	**Transições**	**Arcos**	**Componentes**	**Acurácia**
Alpha miner	4	13	13	17	53,6480
Heuristic miner	21	34	80	55	85,7668
Inductive miner	47	64	136	111	99,8530
DFA min caminhos unidos sem retrabalho	46	84	168	130	92,4669

Fonte: Próprio autor (2023)

Neste caso, podemos notar que nosso menor modelo transposto para uma rede de Petri pode ser tão simples quanto os outros, seu número de componentes não sendo muito maior que o número de componentes do modelo gerado pelo *inductive miner*, porém, em questão de acurácia, nosso menor modelo tem uma acurácia maior que as dos modelos gerados pelo *heuristic miner* e pelo *alpha miner*, sendo menor apenas do que o modelo gerado pelo *inductive miner*.

Podemos verificar também utilizando a abordagem *alignments*, para determinar se o modelo gerado tem realmente uma boa acurácia ou é apenas um problema de inundação de *tokens*. O resultado é mostrado na Tabela 13 e corrobora a ideia que nosso modelo é mais complexo que os demais apesar de ter uma acurácia melhor que o *alpha miner* e o *heuristic miner*. Podemos dizer que a acurácia do nosso modelo tende a estar próxima à acurácia do modelo gerado pelo *heuristic miner* na maioria dos casos, mas neste caso em específico, nosso modelo obteve resultado mais aproximado ao do modelo do *inductive miner*.

Por fim, fizemos o procedimento reverso ao transformar as redes de Petri geradas pelos algoritmos em autômatos finitos. Para isto, usamos o grafo de alcançabilidade das redes de Petri e os comparamos com o menor modelo gerado pelo nosso algoritmo. A Tabela 14 nos mostra a comparação entre os modelos.

Podemos verificar que para os modelos gerados pelos algoritmos *alpha miner* e

Tabela 13 – Tabela de comparação dos modelos acurácia *alignments - Finale*

Modelo	**Places**	**Transições**	**Arcos**	**Componentes**	**Acurácia**
Alpha miner	4	13	13	17	53,2570
Heuristic miner	21	34	80	55	66,5392
Inductive miner	47	64	136	111	99,1478
DFA min caminhos unidos sem retrabalho	46	84	168	130	98,3444

Fonte: Próprio autor (2023)

Tabela 14 – Tabela de comparação dos modelos acurácia *autômatos - Finale*

Modelo	**Estados**	**Alfabeto**	**Transições**	**Acurácia**
Alpha miner	0	0	0	Estouro de tempo
Heuristic miner	0	0	0	Estouro de tempo
Inductive miner	265	13	1079	Estouro de memória
DFA min caminhos unidos sem retrabalho	41	13	77	92,9039

Fonte: Próprio autor (2023)

heuristic miner não foi possível obter o grafo de alcançabilidade no tempo limite estabelecido de 2 horas, então, na tabela, representamos seus dados como 0, enquanto que, para o modelo do *inductive miner*, conseguimos o sistema de transições correspondente, mas ao verificar a sua acurácia, houve um estouro de memória devido à sua complexidade.

5.2.2 Prepaid Travel Costs

Nesta Subseção serão abordados os resultados obtidos utilizando a proposta deste trabalho na base de dados *Prepaid Travel Cost*. O *log* de eventos possui 19 atividades, mostradas com suas frequências na Tabela 15; 2099 *traces*, onde a média de tamanho dos *traces* é 8,69 e o desvio padrão do tamanho dos *traces* 2,25; o número de *traces* únicos é 206; o número de eventos é 18246. A Tabela 16 nos mostra os *traces* únicos mais e menos frequentes.

Tabela 15 – Atividades do *log Prepaid Travel Cost* e suas respectivas frequências

Atividades	**Frequência**
Permit SUBMITTED by EMPLOYEE	1952
Permit FINAL_APPROVED by SUPERVISOR	1478
Request For Payment SUBMITTED by EMPLOYEE	2279
Rquest For Payement FINAL_APPROVED by SUPERVISOR	1958
Request For Payment REJECTED by MISSING	9
Permit REJECTED by MISSING	22
Request Payment	1990
Payment Handled	1990
Permit APPROVED by SUPERVISOR	390
Permit FINAL_APPROVED by DIRECTOR	390
Request For Payment APPROVED by PRE_APPROVER	160
Permit APPROVED by PRE_APPROVER	158
Request For Payment REJECTED by EMPLOYEE	18
Request For Payment REJECTED by EMPLOYEE	271
Request For Payment APPROVED by SUPERVISOR	41
Request For Payment FINAL_APPROVED by DIRECTOR	41
Permit REJECTED by PRE_APPROVER	16
Permit REJECTED by EMPLOYEE	84
Request For Payment REJECTED by SUPERVISOR	24
Permit REJECTED by SUPERVISOR	36
Request For Payment SAVED by EMPLOYEE	24
Request For Payment APPROVED by ADMINISTRATION	1726
Request For Payment APPROVED by BUDGET OWNER	705
Request For Payment REJECTED by ADMINISTRATION	230
Permit APPROVED by ADMINISTRATION	1603
Permit APPROVED by BUDGET OWNER	612
Permit REJECTED by ADMINISTRATION	15
Request For Payment REJECTED by BUDGET OWNER	7
Permit REJECTED by BUDGET OWNER	17

Fonte: Próprio autor (2023)

Para construir o modelo e calcular a sua acurácia, separamos o *log* de eventos em base de treino e base de teste. A base de treino vai conter as atividades mostradas na Tabela 17. O número de *traces* da base de treino é 1679; a média de tamanho dos *traces* é 8,74; o desvio padrão do tamanho dos *traces* é 2,16; o número de *traces* únicos é 176 e o número de eventos: 14678.

Tabela 16 – Frequência dos *traces* da base de dados *Prepaid Travel Cost*

Trace	**Frequência**	%
Trace 0	569	27,10
Trace 1	309	14,72
Trace 2	126	6,00
Trace 3	100	4,76
Trace 4	92	4,38
...	...	...
Trace 201	1	0,04
Trace 202	1	0,04
Trace 203	1	0,04
Trace 204	1	0,04
Trace 205	1	0,04

Fonte: Próprio autor (2023)

A Tabela 18 nos mostra as atividades da base de testes. O número de *traces* da base de testes é 420; a média de tamanho dos *traces* é 8,49; o desvio padrão do tamanho dos *traces* é 2,57; o número de *traces* únicos é 71 e o número de eventos: 3568.

Tabela 17 – Atividades do *log Prepaid Travel Cost* - Treino

Atividades	**Frequência**
Permit SUBMITTED by EMPLOYEE	1615
Permit FINAL_APPROVED by SUPERVISOR	1212
Request For Payment SUBMITTED by EMPLOYEE	1817
Request For Payement FINAL_APPROVED by SUPERVISOR	1578
Request For Payment REJECTED by MISSING	9
Permit REJECTED by MISSING	22
Request Payment	1602
Payment Handled	1601
Permit APPROVED by SUPERVISOR	338
Permit FINAL_APPROVED by DIRECTOR	338
Request For Payment APPROVED by PRE_APPROVER	160
Permit APPROVED by PRE_APPROVED	158
Request For Payment REJECTED by PRE_APPROVER	18
Request For Payment REJECTED by EMPLOYEE	200
Request For Payment APPROVED by SUPERVISOR	32
Request For Payment FINAL_APPROVED by DIRECTOR	32
Permit REJECTED by PRE_APPROVER	16
Permit REJECTED by EMPLOYEE	65
Request For Payment REJECTED by SUPERVISOR	15
Permit REJECTED by SUPERVISOR	28
Request For Payment SAVED by EMPLOYEE	18
Request For Payment APPROVED by ADMINISTRATION	1324
Request For Payment APPROVED by BUDGET OWNER	544
Request For Payment REJECTED by ADMINISTRATION	170
Permit APPROVED by ADMINISTRATION	1273
Permit APPROVED by BUDGET OWNER	469
Permit REJECTED by ADMINISTRATION	8
Request For Payment REJECTED by BUDGET OWNER	3
Permit REJECTED by BUDGET OWNER	13

Fonte: Próprio autor (2023)

Para um melhor entendimento, apresentamos os dados das bases na Tabela 19.

Tabela 18 – Atividades do *log Prepaid Travel Cost* - Teste

Atividades	**Frequência**
Permit SUBMITTED by EMPLOYEE	337
Permit APPROVED by ADMINISTRATION	330
Request For Payment SUBMITTED by EMPLOYEE	462
Request For Payment APPROVED by ADMINISTRATION	402
Permit APPROVED by BUDGET OWNER	143
Request For Payment APPROVED by BUDGET OWNER	161
Request For Payement FINAL_APPROVED by SUPERVISOR	380
Permit FINAL_APPROVED by SUPERVISOR	266
Request Payment	388
Payment Handled	389
Request For Payment REJECTED by ADMINISTRATION	60
Request For Payment REJECTED by EMPLOYEE	71
Permit APPROVED by SUPERVISOR	52
Permit FINAL_APPROVED by DIRECTOR	52
Permit REJECTED by SUPERVISOR	8
Permit REJECTED by EMPLOYEE	19
Request For Payment APPROVED by SUPERVISOR	9
Request For Payment FINAL_APPROVED by DIRECTOR	9
Permit REJECTED by ADMINISTRATION	7
Permit REJECTED by BUDGET OWNER	4
Request For Payment SAVED by EMPLOYEE	6
Request For Payment REJECTED by SUPERVISOR	9
Request For Payment REJECTED by BUDGET OWNER	4

Fonte: Próprio autor (2023)

Tabela 19 – Comparação das bases de dados

Base	**N° traces**	**Média tam traces**	**Eventos**	**Traces únicos**	**Eventos únicos**
Log original	2099	8,69	18246	206	2109
Base treino	1679	8,74	14678	176	1787
Base teste	420	8,49	3568	71	721

Fonte: Próprio autor (2023)

As Tabelas 20 e 21 nos mostram tabelas com os dados das máquinas de estados geradas a partir da base de dados. Para calcular a acurácia dos autômatos, verificamos se os *traces* da base de testes são aceitos pelos autômatos gerados a partir da base de treino e calculamos a porcentagem de aceitação. Para os resultados das acurácias com as abordagens *token based replay*, é interessante notar que nas máquinas não-determinísticas as acurácias são bem menores que o restante. Isso se deve ao mesmo motivo apresentado na Subseção 5.2.1. Nos resultados das acurácias utilizando a abordagem *alignments*, todos os modelos gerados possuem uma acurácia em torno de noventa e sete por cento para comportamentos não observados, o que é um resultado muito bom.

Tabela 20 – Máquinas de estados

N° de Traces únicos: 1679 Frequência: 1.0 N° Eventos: 14678						
Máquina de estados	**Atividades**	**Estados**	**Transições**	**Estados de Aceitação**	**Sub-Autômatos**	**Estados totais**
NFA	29	1788	1787	176	0	1788
DFA	29	937	936	176	-	-
DFA min	29	302	453	14	-	-
OP Seq 3-25	29	219	370	14	21	323
NFA CM	29	1742	1755	176	0	1742
DFA CM	29	895	965	173	-	-
DFA min CM	29	286	457	15	-	-
OP Seq 3-25	29	227	398	15	16	302
NFA sRet	29	1742	1741	176	0	1742
DFA sRet	29	864	863	165	-	-
DFA min sRet	29	277	419	12	-	-
OP Seq 3-25	29	208	350	12	19	296
NFA join	29	1633	1645	165	0	1633
DFA join	29	1633	1483	165	-	-
DFA min join	29	286	457	16	-	-
OP Seq 3-25	29	227	398	16	16	302
NFA sRet join	29	1633	1632	165	0	1633
DFA sRet join	29	864	863	165	-	-
DFA min sRet join	29	277	419	12	-	-
OP Seq 3-25	29	208	350	12	19	296
Legenda:						
	CM: Caminhos mínimos	sRet: Sem retrabalho	OP Seq: Operação sequências			

Fonte: Próprio autor (2023)

Neste caso, todos os modelos gerados possuem uma acurácia em torno de noventa e sete por cento para comportamentos não observados, o que é um resultado muito bom.

Tabela 21 – Acurácias das máquinas de estados geradas a partir da base de dados *Prepaid Travel Cost*

Referente à	**Acurácia Autômatos**	**Token based Replay**	**Alignments**
Não-Determinística	91,66	55,5165	97,4162
Determinística	91,66	61,8606	97,4162
Determinística min	91,66	68,0875	97,4152
Operação Sequências min/max:3-25 estados DFA min	91,66	-	-
Não-Determinística caminho mínimo	92,3809	45,5366	97,6158
Determinística caminho mínimo	92,3809	58,6431	97,6144
Determinística min caminho mínimo	92,3809	65,8201	97,6129
Operação Sequências min/max:3-25 estados DFA min caminho mínimo	92,3809	-	-
Não-Determinística sem retrabalho	91,9047	62,0577	97,3892
Determinística sem retrabalho	91,9047	93,9954	97,3891
Determinística min sem retrabalho	91,9047	76,3707	97,3882
Operação Sequências min/max:3-25 estados DFA min sem retrabalho	91,9047	-	-
Não-Determinística caminhos unidos	92,3809	56,3190	97,6158
Determinística caminhos unidos	92,3809	57,0892	97,6143
Determinística min caminhos unidos	92,3809	66,9395	97,6129
Operação Sequências min/max:3-25 estados DFA min caminhos unidos	92,3809	-	-
Não-Determinística caminhos unidos sem retrabalho	91,9047	58,1681	97,3892
Determinística caminhos unidos sem retrabalho	91,9047	93,7803	97,3891
Determinística min caminhos unidos sem retrabalho	91,9047	75,6081	97,3882
Operação Sequências min/max:3-25 estados DFA min caminhos unidos sem retrabalho	91,9047	-	-

Fonte: Próprio autor (2023)

As Tabelas 22 e 23 nos mostram as tabelas comparativas dos nossos menores modelos com os modelos gerados pelos algoritmos *alpha miner*, *heuristic miner* e *inductive miner*, utilizando as abordagens *token based replay* e *alignments*, respectivamente.

Neste caso, podemos notar que nosso modelo é mais complexo que os outros, pois possui muitos componentes a mais, mas em relação à acurácia, nosso modelo tem um melhor desempenho do que o modelo gerado pelo algoritmo *alpha miner* nas duas abordagens, mmelhor que o modelo do *heuristic miner* utilizando a abordagem *alignments*, e do que o modelo do *inductive miner* utilizando a abordagem *token based replay*. Interessante notar também que não foi possível calcular a acurácia utilizando a abordagem baseada em *alignments* para o modelo

Tabela 22 – Tabela de comparação dos modelos acurácia *token based replay - Prepaid Travel Cost*

Modelo	Places	Transições	Arcos	Componentes	Acurácia
Alpha miner	74	29	307	103	50,5338
Heuristic miner	54	100	222	154	90,9170
Inductive miner	98	132	290	230	46,8504
DFA min caminhos unidos sem retrabalho	98	132	290	230	46,8504

Fonte: Próprio autor (2023)

Tabela 23 – Tabela de comparação dos modelos acurácia *alignments - Prepaid Travel Cost*

Modelo	Places	Transições	Arcos	Componentes	Acurácia
Alpha miner	74	29	307	103	0
Heuristic miner	54	100	222	154	80,1085
Inductive miner	98	132	290	230	Estouro de memória
DFA min caminhos unidos sem retrabalho	98	132	290	230	97,3882

Fonte: Próprio autor (2023)

gerado pelo *inductive miner*, pois houve um estouro de memória, e para o modelo gerado pelo *alpha miner*, pois o modelo gerado é dito não ser *sound*.

Por fim, fizemos o procedimento reverso ao transformar as redes de Petri geradas pelos algoritmos em autômatos finitos. Para isto, usamos o grafo de alcançabilidade das redes de Petri e os comparamos com o menor modelo gerado pelo nosso algoritmo. A Tabela 24 nos mostra a comparação entre os modelos.

Tabela 24 – Tabela de comparação dos modelos acurácia *autômatos - Prepaid Travel Cost*

Modelo	Estados	Alfabeto	Transições	Acurácia
Alpha miner	0	0	0	Estouro de memória
Heuristic miner	0	0	0	Estouro de tempo
Inductive miner	0	0	0	Estouro de memória
DFA min caminhos unidos sem retrabalho	277	29	419	91,9047

Fonte: Próprio autor (2023)

Podemos verificar que para os modelos gerados pelos algoritmos *alpha miner* e *inductive miner* não foi possível obter os grafos de alcançabilidade por haver um estouro de memória durante a conversão, enquanto que, para o modelo do *heuristic miner*, não conseguimos obter o sistema de transições correspondente no tempo limite estabelecido de 2 horas.

Acreditamos que os resultados de nossa proposta para este *log* de eventos seja o melhor, comparado com os outros experimentos, pois este é o *log* de eventos com o menor

número de atividades que aparecem apenas uma vez, fazendo com que elas apareçam em nosso modelo mesmo separando o *log* em treino e teste.

5.2.3 Processo 1

Nesta Subseção serão abordados os resultados obtidos utilizando a proposta deste trabalho na base de dados que chamaremos de *Processo 1*, gentilmente cedido pela Secretaria da Fazenda do Ceará (SEFAZ-CE). Nele, fizemos uma filtragem dos dados para considerar apenas os processos que ocorreram em 2018. O *log* de eventos filtrado pelo ano de 2018 possui 22 atividades, mostradas com suas frequências na Tabela 25; também possui 167 *traces*, onde a média de tamanho dos *traces* é 14,26 e o desvio padrão do tamanho dos *traces* 10,39; o número de *traces* únicos é 143; o número de eventos é 2382. A Tabela 26 nos mostra os *traces* únicos mais e menos frequentes.

Tabela 25 – Atividades do *log* Processo 1 e suas respectivas frequências

Atividades	**Frequência**
Atividade 0	167
Atividade 1	150
Atividade 2	502
Atividade 3	87
Atividade 4	63
Atividade 5	542
Atividade 6	180
Atividade 7	220
Atividade 8	25
Atividade 9	93
Atividade 10	67
Atividade 11	99
Atividade 12	77
Atividade 13	77
Atividade 14	1
Atividade 15	4
Atividade 16	1
Atividade 17	1
Atividade 18	6
Atividade 19	10
Atividade 20	3
Atividade 21	7

Fonte: Próprio autor (2023)

Tabela 26 – Frequência dos *traces* da base de dados Processo 1 - 2018

Trace	**Frequência**	%
Trace 0	8	4,7904
Trace 1	5	2,9940
Trace 2	4	2,3952
Trace 3	4	2,3952
Trace 4	4	2,3952
...	...	...
Trace 138	1	0,5988
Trace 139	1	0,5988
Trace 140	1	0,5988
Trace 141	1	0,5988
Trace 142	1	0,5988

Fonte: Próprio autor (2023)

Para construir o modelo e calcular a sua acurácia, separamos o *log* de eventos em base de treino e base de teste. A base de treino vai conter as atividades mostradas na Tabela 27.

Tabela 27 – Atividades do *log* Processo 1 e suas respectivas frequências - Treino

Atividades	**Frequência**
Atividade 0	133
Atividade 1	117
Atividade 2	395
Atividade 3	69
Atividade 4	48
Atividade 5	411
Atividade 6	145
Atividade 7	170
Atividade 8	22
Atividade 9	93
Atividade 10	50
Atividade 11	78
Atividade 12	59
Atividade 13	59
Atividade 14	1
Atividade 15	3
Atividade 16	1
Atividade 17	1
Atividade 18	6
Atividade 19	7
Atividade 20	3
Atividade 21	6

Fonte: Próprio autor (2023)

O número de traces da base de treino é 133; a média de tamanho dos traces é 14,11;

o desvio padrão do tamanho dos traces é 9,53; o número de traces únicos é 114 e o número de eventos: 1877.

A Tabela 28 nos mostra as atividades da base de testes.

Tabela 28 – Atividades do *log* Processo 1 e suas respectivas frequências - Treino

Atividades	**Frequência**
Atividade 0	34
Atividade 1	33
Atividade 2	107
Atividade 21	1
Atividade 7	50
Atividade 6	35
Atividade 12	18
Atividade 3	18
Atividade 4	15
Atividade 5	131
Atividade 13	18
Atividade 10	17
Atividade 11	21
Atividade 8	3
Atividade 19	3
Atividade 15	1

Fonte: Próprio autor (2023)

O número de traces da base de testes é 34; a média de tamanho dos traces é 14,85; o desvio padrão do tamanho dos traces é 13,22; o número de traces únicos é 34 e o número de eventos: 505.

Para um melhor entendimento, apresentamos os dados das bases na Tabela 29.

Tabela 29 – Comparação das bases de dados

Base	**N° traces**	**Média tam traces**	**Eventos**	**Traces únicos**	**Eventos únicos**
Log original	167	14,26	2382	143	2275
Base treino	133	14,11	1877	114	1798
Base teste	34	14,85	505	34	505

Fonte: Próprio autor (2023)

A Tabela 30 mostra os dados das máquinas de estados geradas a partir da base de dados.

Tabela 30 – Máquinas de estados

N° de Traces únicos: 133 Frequência: 1.0 N° Eventos: 1877						
Máquina de estados	**Atividades**	**Estados**	**Transições**	**Estados de Aceitação**	**Sub-Autômatos**	**Estados totais**
NFA	22	1799	1798	114	0	1799
DFA	22	1180	1179	114	-	-
DFA min	22	620	730	3	-	-
OP Seq 3-25	22	213	323	3	52	672
NFA CM	22	1190	1397	114	0	1190
DFA CM	22	722	1013	111	-	-
DFA min CM	22	333	634	4	-	-
OP Seq 3-25	22	282	583	4	17	350
NFA sRet	22	1190	1189	114	0	1190
DFA sRet	22	623	622	104	-	-
DFA min sRet	22	241	339	2	-	-
OP Seq 3-25	22	155	253	2	20	261
NFA join	22	1110	1308	104	0	1110
DFA join	22	1110	1202	104	-	-
DFA min join	22	333	634	4	-	-
OP Seq 3-25	22	282	583	4	17	350
NFA sRet join	22	1110	1109	104	0	1110
DFA sRet join	22	623	622	104	-	-
DFA min sRet join	22	241	339	2	-	-
OP Seq 3-25	22	155	253	2	20	261
Legenda:						
	CM: Caminhos mínimos	sRet: Sem retrabalho	OP Seq: Operação sequências			

Fonte: Próprio autor (2023)

Para calcular a acurácia dos autômatos, verificamos se os *traces* da base de testes são aceitos pelos autômatos gerados a partir da base de treino e calculamos a porcentagem de aceitação. A Tabela 31 nos mostra os resultados dos cálculos das acurácias dos modelos gerados.

Podemos ver que os autômatos das linhas 11 e 19 são as máquinas com os menores números de componentes encontradas. É notável que as máquinas geradas utilizando os métodos de caminho mínimo e junção de caminhos, linhas 4, 8, 12 e 16, conseguem uma acurácia maior do que a máquina da linha 1, pois a maneira que são construídos os permite aceitar mais

Tabela 31 – Acurácias das máquinas de estados geradas a partir da base de dados Processo 1

Referente à	**Acurácia Autômatos**	**Token based Replay**	**Alignments**
Não-Determinística	17,4670	55,10	Estouro de memória
Determinística	17,4670	72,9630	Estouro de memória
Determinística min	17,4670	75,3333	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min	17,4670	-	-
Não-Determinística caminho mínimo	32,3529	65,9686	Estouro de memória
Determinística caminho mínimo	32,3529	69,3475	Estouro de memória
Determinística min caminho mínimo	32,3529	53,4539	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminho mínimo	32,3529	-	-
Não-Determinística sem retrabalho	32,3529	55,0914	Estouro de memória
Determinística sem retrabalho	32,3529	81,2500	Estouro de memória
Determinística min sem retrabalho	32,3529	86,4745	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min sem retrabalho	32,3529	-	-
Não-Determinística caminhos unidos	32,3529	71,6733	Estouro de memória
Determinística caminhos unidos	32,3529	67,5896	Estouro de memória
Determinística min caminhos unidos	32,3529	56,1258	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminhos unidos	32,3529	-	-
Não-Determinística caminhos unidos sem retrabalho	32,3529	56,3969	Estouro de memória
Determinística caminhos unidos sem retrabalho	32,3529	87,0813	Estouro de memória
Determinística min caminhos unidos sem retrabalho	32,3529	79,1111	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminhos unidos sem retrabalho	32,3529	-	-

Fonte: Próprio autor (2023)

comportamentos do que aqueles previstos, aumentando a sua generalização. Quando calculamos as acurácias utilizando a abordagem *token based replay*, o resultado é bem maior que utilizando

a abordagem das entradas dos autômatos, mas como não há um padrão de acurácia para os modelos, mesmo aqueles gerados a partir da determinização de um NFA ou a minização de um DFA, é levado em consideração que pode haver uma inundação de *tokens* gerando resultados de acurácia melhores do que o que realmente são. Não foi possível obter o resultado da acurácia utilizando a abordagem *alignments* pois houve estouro de memória devido a complexidade dos modelos gerados.

As Tabelas 32 e 33 nos mostram as tabelas comparativas dos nossos menores modelos com os modelos gerados pelos algoritmos *alpha miner*, *heuristic miner* e *inductive miner*, utilizando as abordagens *token based replay* e *alignments*, respectivamente.

Tabela 32 – Tabela de comparação dos modelos acurácia *token based replay* - Processo 1

Modelo	**Places**	**Transições**	**Arcos**	**Componentes**	**Acurácia**
Alpha miner	7	22	15	29	62,4672
Heuristic miner	32	71	153	103	92,2278
Inductive miner	72	103	216	175	99,9795
DFA min caminhos unidos sem retrabalho	244	343	686	587	78,8546

Fonte: Próprio autor (2023)

Tabela 33 – Tabela de comparação dos modelos acurácia *alignments* - Processo 1

Modelo	**Places**	**Transições**	**Arcos**	**Componentes**	**Acurácia**
Alpha miner	7	22	15	29	0
Heuristic miner	32	71	153	103	83,5496
Inductive miner	72	103	216	175	99,7546
DFA min caminhos unidos sem retrabalho	244	343	686	587	86,0297

Fonte: Próprio autor (2023)

Neste caso, podemos notar que nosso modelo é mais complexo que os outros, pois possui muitos componentes a mais e, em relação à acurácia, nosso modelo tem um pior desempenho do que os modelos gerados pelos algoritmos *inductive miner* e *heuristic miner* com a abordagem *token based replay*, enquanto que na abordagem *alignments*, a acurácia do nosso modelo só não é maior do que a acurácia do modelo gerado pelo *inductive miner*.

Por fim, fizemos o procedimento reverso ao transformar as redes de Petri geradas pelos algoritmos em autômatos finitos. Para isto, usamos o grafo de alcançabilidade das redes de Petri e os comparamos com o menor modelo gerado pelo nosso algoritmo. A Tabela 34 nos mostra a comparação entre os modelos.

Tabela 34 – Tabela de comparação dos modelos acurácia *autômatos* - Processo 1

Modelo	**Estados**	**Alfabeto**	**Transições**	**Acurácia**
Alpha miner	0	0	0	Estouro de tempo
Heuristic miner	0	0	0	Estouro de tempo
Inductive miner	0	0	0	Estouro de memória
DFA min caminhos unidos sem retrabalho	241	22	339	32,352941

Fonte: Próprio autor (2023)

Podemos verificar que para os modelos gerados pelos algoritmos *alpha miner* e *heuristic miner* não foi possível obter o grafo de alcançabilidade no tempo limite estabelecido de 2 horas, enquanto que, para o modelo do *inductive miner*, conseguimos obter o sistema de transições correspondente, mas o autômato gerado possui muitos componentes a mais que o nosso, assim, ao calcular a acurácia do modelo para dados não vistos, houve um estouro de memória devido a sua complexidade. A baixa acurácia do nosso modelo se deve ao *log* possuir atividades que acontecem poucas vezes e também ao fato de autômatos não conseguirem lidar com atividades em paralelo como outros tipos de modelos.

5.2.4 Processo 2

Nesta Subseção serão abordados os resultados obtidos utilizando a proposta deste trabalho na base de dados que chamaremos de *Processo 2*, gentilmente cedido pela Secretaria da Fazenda do Ceará (SEFAZ-CE). Neste *log*, fizemos uma filtragem afim de obter os processos que ocorreram no ano de 2018. O *log* de eventos filtrado pelo ano de 2018 possui 24 atividades, mostradas com suas frequências na Tabela 35; também possui 1353 *traces*, onde a média de tamanho dos *traces* é 13,69 e o desvio padrão do tamanho dos *traces* 10,32; o número de *traces* únicos é 422; o número de eventos é 18536. A Tabela 36 nos mostra os *traces* únicos mais e menos frequentes.

Tabela 35 – Atividades do *log* Processo 2 e suas respectivas frequências

Atividades	**Frequência**
Atividade 0	1505
Atividade 1	1505
Atividade 2	3772
Atividade 3	4918
Atividade 4	1562
Atividade 5	1492
Atividade 6	402
Atividade 7	1492
Atividade 8	1625
Atividade 9	1584
Atividade 10	1284
Atividade 11	11
Atividade 12	3
Atividade 13	54
Atividade 14	14
Atividade 15	147
Atividade 16	10
Atividade 17	4
Atividade 18	3
Atividade 19	11
Atividade 20	7
Atividade 21	3
Atividade 22	1
Atividade 23	5
Atividade 24	15
Atividade 25	1
Atividade 26	49
Atividade 27	5
Atividade 28	1

Fonte: Próprio autor (2023)

Para construir o modelo e calcular a sua acurácia, separamos o *log* de eventos em base de treino e base de teste. A base de treino vai conter as atividades mostradas na Tabela 37.

O número de traces da base de treino é 1204; a média de tamanho dos traces é 14,28; o desvio padrão do tamanho dos traces é 6,30; o número de traces únicos é 433 e o número de eventos: 17200.

Tabela 36 – Frequência dos *traces* da base de dados Processo 1 - 2018

Trace	**Frequência**	%
Trace 0	133	9,83
Trace 1	86	6,3562
Trace 2	74	5,4693
Trace 3	58	4,2867
Trace 4	56	4,1389
...	...	...
Trace 417	1	0,0739
Trace 418	1	0,0739
Trace 419	1	0,0739
Trace 420	1	0,0739
Trace 421	1	0,0739

Fonte: Próprio autor (2023)

Tabela 37 – Atividades do *log* Processo 2 - Treino

Atividades	**Frequência**
Atividade 0	1204
Atividade 1	1204
Atividade 2	3145
Atividade 3	3878
Atividade 4	1258
Atividade 5	1198
Atividade 6	402
Atividade 7	1195
Atividade 8	1239
Atividade 9	1283
Atividade 10	1047
Atividade 11	11
Atividade 12	3
Atividade 13	53
Atividade 14	14
Atividade 15	26
Atividade 16	10
Atividade 17	2
Atividade 18	2
Atividade 19	11
Atividade 20	6
Atividade 21	3
Atividade 22	1
Atividade 23	5

Fonte: Próprio autor (2023)

A Tabela 38 nos mostra as atividades da base de testes.

O número de *traces* da base de testes é 301; a média de tamanho dos *traces* é 14,23; o desvio padrão do tamanho dos *traces* é 5,26; o número de *traces* únicos é 155 e o número de

Tabela 38 – Atividades do *log* Processo 2 - Treino

Atividades	**Frequência**
Atividade 0	301
Atividade 1	301
Atividade 3	1040
Atividade 5	294
Atividade 7	297
Atividade 4	304
Atividade 8	386
Atividade 10	237
Atividade 9	301
Atividade 2	627
Atividade 15	121
Atividade 24	15
Atividade 25	1
Atividade 17	2
Atividade 18	1
Atividade 26	49
Atividade 20	1
Atividade 27	5
Atividade 28	1
Atividade 13	1

Fonte: Próprio autor (2023)

eventos: 4285.

Para um melhor entendimento, apresentamos os dados das bases na Tabela 39.

Tabela 39 – Comparação das bases de dados

Base	**N° traces**	**Média tam traces**	**Eventos**	**Traces únicos**	**Eventos únicos**
Log original	1353	13.69	18536	422	9797
Base treino	1204	14.28	17200	433	7616
Base teste	301	14.23	4285	155	2429

Fonte: Próprio autor (2023)

A Tabela 40 nos mostra os dados das máquinas de estados geradas a partir da base de dados.

Tabela 40 – Máquinas de estados

N° de Traces únicos: 1204 Frequência: 1.0 N° Eventos: 17200						
Máquina de estados	**Atividades**	**Estados**	**Transições**	**Estados de Aceitação**	**Sub-Autômatos**	**Estados totais**
NFA	24	7617	7616	433	0	7617
DFA	24	3650	3649	433	-	-
DFA min	24	947	1305	14	-	-
OP Seq 3-25	24	563	921	14	94	1051
NFA CM	24	5357	5889	433	0	5357
DFA CM	24	1960	2272	296	-	-
DFA min CM	24	457	848	10	-	-
OP Seq 3-25	24	406	797	10	17	474
NFA sRet	24	5357	5356	433	0	5357
DFA sRet	24	1260	1259	184	-	-
DFA min sRet	24	279	431	9	-	-
OP Seq 3-25	24	218	370	9	14	293
NFA join	24	2397	2446	184	0	2397
DFA join	24	2397	2446	184	-	-
DFA min join	24	467	876	10	-	-
OP Seq 3-25	24	418	827	10	16	483
NFA sRet join	24	2397	2396	184	0	2397
DFA sRet join	24	1260	1259	104	-	-
DFA min sRet join	24	279	431	9	-	-
OP Seq 3-25	24	218	370	9	14	293
Legenda:						
	CM: Caminhos mínimos	sRet: Sem retrabalho	OP Seq: Operação sequências			

Fonte: Próprio autor (2023)

Para calcular a acurácia dos autômatos, verificamos se os *traces* da base de testes são aceitos pelos autômatos gerados a partir da base de treino e calculamos a porcentagem de aceitação. A Tabela 41 nos mostra os resultados dos cálculos das acurácias para cada modelo.

Podemos ver que os autômatos das linhas 11 e 19 são as máquinas com os menores números de componentes encontradas. É notável que as máquinas geradas utilizando os métodos de caminho mínimo e junção de caminhos, linhas 4, 8, 12 e 16, conseguem uma acurácia maior do que a máquina da linha 1, pois a maneira que são construídos os permite aceitar mais

Tabela 41 – Acurácias das máquinas de estados geradas a partir da base de dados Processo 2

Referente à	**Acurácia Autômatos**	**Token based Replay**	**Alignments**
Não-Determinística	23,2558	69,8117	Estouro de memória
Determinística	23,2558	75,4817	Estouro de memória
Determinística min	23,2558	62,5354	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min	23,2558	-	-
Não-Determinística caminho mínimo	29,2358	55,7475	Estouro de memória
Determinística caminho mínimo	29,2358	66,6134	Estouro de memória
Determinística min caminho mínimo	29,2358	70,1309	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminho mínimo	29,2358	-	-
Não-Determinística sem retrabalho	36,5448	53,8622	Estouro de memória
Determinística sem retrabalho	36,5448	87,2918	Estouro de memória
Determinística min sem retrabalho	36,5448	87,1806	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min sem retrabalho	36,5448	-	-
Não-Determinística caminhos unidos	29,2358	59,9349	Estouro de memória
Determinística caminhos unidos	29,2358	74,1612	Estouro de memória
Determinística min caminhos unidos	29,2358	63,1924	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminhos unidos	29,2358	-	-
Não-Determinística caminhos unidos sem retrabalho	36,5448	55,4280	Estouro de memória
Determinística caminhos unidos sem retrabalho	36,5448	72,1033	Estouro de memória
Determinística min caminhos unidos sem retrabalho	36,5448	82,2022	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminhos unidos sem retrabalho	36,5448	-	-

Fonte: Próprio autor (2023)

comportamentos do que aqueles previstos, aumentando a sua generalização. Quando utilizamos a abordagem *token based replay*, a acurácia das máquinas é maior que utilizando a abordagem

das entradas dos autômatos, mas como não há um padrão de acurácia para os modelos, mesmo aqueles gerados a partir da determinização de um NFA ou a minização de um DFA, é levado em consideração que pode haver uma inundação de *tokens* gerando resultados de acurácia melhores do que o que realmente são. Não foi possível calcular a acurácia das máquinas usando a abordagem baseada em *alignments*, pois houve um estouro de memória durante o processo de cálculo.

As Tabelas 42 e 43 nos mostram as tabelas comparativas dos nossos menores modelos com os modelos gerados pelos algoritmos *alpha miner*, *heuristic miner* e *inductive miner*, utilizando as abordagens *token based replay* e *alignments*, respectivamente.

Tabela 42 – Tabela de comparação dos modelos acurácia *token based replay*

Modelo	**Places**	**Transições**	**Arcos**	**Componentes**	**Acurácia**
Alpha miner	10	24	29	34	73,9639
Heuristic miner	36	68	148	104	89,5620
Inductive miner	54	81	166	135	99,9826
DFA min caminhos unidos sem retrabalho	283	443	886	726	81,9519

Fonte: Próprio autor (2023)

Tabela 43 – Tabela de comparação dos modelos acurácia *alignments*

Modelo	**Places**	**Transições**	**Arcos**	**Componentes**	**Acurácia**
Alpha miner	10	24	29	34	0
Heuristic miner	36	68	148	104	61,6720
Inductive miner	54	81	166	135	98,3699
DFA min caminhos unidos sem retrabalho	283	443	886	726	92,3416

Fonte: Próprio autor (2023)

Neste caso, podemos notar que nosso modelo é mais complexo que os outros, pois possui muitos componentes a mais e, em relação à acurácia, nosso modelo tem um pior desempenho do que os modelos gerados pelos algoritmos *inductive miner* e *heuristic miner*, embora a diferença não seja tão grande assim, precisamos considerar a questão que nosso modelo é maior e mesmo assim possui uma acurácia menor na abordagem *token based replay*, mas como na abordagem *alignments* a acurácua do nosso modelo é melhor que a acurácia do modelo do *heuristic miner*, podemos dizer que houve uma inundação de *tokens* no cálculo da acurácia do segundo. Perecebe-se que as acurácias baseadas em *alignments* e *token based replay* têm resultados melhores do que a acurácia diretamente do autômato, que pode ser acarretada pelo

paralelismo dos eventos não ser bem aceito em um sistema de transições, enquanto que as redes de Petri lidam melhor com este fator, além do fato que este *log* possui muitas atividades que só ocorrem uma vez e quando verificamos a acurácia do autômato, caso essa atividade esteja no conjunto de teste e não esteja no conjunto de treino, ou vice-versa, a entrada é considerada como não-aceita.

Por fim, fizemos o procedimento reverso ao transformar as redes de Petri geradas pelos algoritmos em autômatos finitos. Para isto, usamos o grafo de alcançabilidade das redes de Petri e os comparamos com o menor modelo gerado pelo nosso algoritmo. A Tabela 44 nos mostra a comparação entre os modelos.

Tabela 44 – Tabela de comparação dos modelos acurácia *autômatos*

Modelo	**Estados**	**Alfabeto**	**Transições**	**Acurácia**
Alpha miner	0	0	0	Estouro de tempo
Heuristic miner	0	0	0	Estouro de tempo
Inductive miner	66	24	133	Estouro de memória
DFA min caminhos unidos sem retrabalho	279	24	431	36,5448

Fonte: Próprio autor (2023)

Podemos verificar que para os modelos gerados pelos algoritmos *alpha miner* e *heuristic miner* não foi possível obter o grafo de alcançabilidade no tempo limite estabelecido de 2 horas, enquanto que, para o modelo do *inductive miner*, conseguimos obter o sistema de transições correspondente, mas ao calcular a acurácia do modelo para dados não vistos, houve um estouro de memória.

5.2.5 Processo 3

Nesta Subseção serão abordados os resultados obtidos utilizando a proposta deste trabalho na base de dados que chamaremos de *Processo 3*, gentilmente cedido pela Secretaria da Fazenda do Ceará (SEFAZ-CE). Fizemos uma filtragem afim de obter os processos ocorridos em 2018. O *log* de eventos filtrado pelo ano de 2018 possui 33 atividades, mostradas com suas frequências na Tabela 45; também possui 248 *traces*, onde a média de tamanho dos *traces* é 26,60 e o desvio padrão do tamanho dos *traces* 12,09; o número de *traces* únicos é 245; o número de eventos é 6598. A Tabela 46 nos mostra os *traces* únicos mais e menos frequentes.

Tabela 45 – Atividades do *log* Processo 3 e suas respectivas frequências

Atividades	**Frequência**
Atividade 0	248
Atividade 1	238
Atividade 2	1228
Atividade 3	342
Atividade 4	326
Atividade 5	198
Atividade 6	259
Atividade 7	1776
Atividade 8	27
Atividade 9	188
Atividade 10	283
Atividade 11	136
Atividade 12	253
Atividade 13	328
Atividade 14	275
Atividade 15	132
Atividade 16	55
Atividade 17	103
Atividade 18	53
Atividade 19	31
Atividade 20	19
Atividade 21	10
Atividade 22	2
Atividade 23	3
Atividade 24	11
Atividade 25	15
Atividade 26	46
Atividade 27	5
Atividade 28	1
Atividade 29	3
Atividade 30	2
Atividade 31	1
Atividade 32	1

Fonte: Próprio autor (2023)

Para construir o modelo e calcular a sua acurácia, separamos o *log* de eventos em base de treino e base de teste. A base de treino vai conter as atividades mostradas na Tabela 47.

Tabela 46 – Frequência dos *traces* da base de dados Processo 3 - 2018

Trace	**Frequência**	%
Trace 0	2	0,8064
Trace 1	2	0,8064
Trace 2	2	0,8064
Trace 3	1	0,4032
Trace 4	1	0,4032
...	...	...
Trace 240	1	0,4032
Trace 241	1	0,4032
Trace 242	1	0,4032
Trace 243	1	0,4032
Trace 244	1	0,4032

Fonte: Próprio autor (2023)

O número de *traces* da base de treino é 198; a média de tamanho dos *traces* é 26,31; o desvio padrão do tamanho dos *traces* é 11,29; o número de *traces* únicos é 197 e o número de eventos: 5210.

Tabela 47 – Atividades do *log* Processo 3 -Treino

Atividades	**Frequência**
Atividade 0	198
Atividade 1	187
Atividade 2	958
Atividade 3	272
Atividade 4	258
Atividade 5	169
Atividade 6	206
Atividade 7	1394
Atividade 8	27
Atividade 9	157
Atividade 10	223
Atividade 11	130
Atividade 12	201
Atividade 13	269
Atividade 14	219
Atividade 15	99
Atividade 16	50
Atividade 17	65
Atividade 18	28
Atividade 19	19
Atividade 20	16
Atividade 21	9
Atividade 22	1
Atividade 23	1
Atividade 24	7
Atividade 25	11
Atividade 26	27
Atividade 27	2
Atividade 28	1
Atividade 29	3

Fonte: Próprio autor (2023)

A Tabela 48 nos mostra as atividades da base de testes.

O número de *traces* da base de testes é 50; a média de tamanho dos *traces* é 27,76; o desvio padrão do tamanho dos *traces* é 14,78; o número de *traces* únicos é 49 e o número de eventos: 1388.

Tabela 48 – Atividades do *log* Processo 3 -Teste

Atividades	**Frequência**
Atividade 0	50
Atividade 1	51
Atividade 7	382
Atividade 2	270
Atividade 4	68
Atividade 17	38
Atividade 6	53
Atividade 3	70
Atividade 9	31
Atividade 10	60
Atividade 11	6
Atividade 12	52
Atividade 13	59
Atividade 14	56
Atividade 26	19
Atividade 15	33
Atividade 18	25
Atividade 5	29
Atividade 19	12
Atividade 20	3
Atividade 32	1
Atividade 21	1
Atividade 16	5
Atividade 22	1
Atividade 23	2
Atividade 27	3
Atividade 24	4
Atividade 25	4

Fonte: Próprio autor (2023)

Para um melhor entendimento, apresentamos os dados das bases na Tabela 49.

Tabela 49 – Comparação das bases de dados

Base	**N° traces**	**Média tam traces**	**Eventos**	**Traces únicos**	**Eventos únicos**
Log original	248	26,60	6598	245	6547
Base treino	198	26,31	5210	197	5185
Base teste	50	27,76	1388	49	1371

Fonte: Próprio autor (2023)

A Tabela 50 nos mostra os dados das máquinas de estados geradas a partir da base de dados.

Tabela 50 – Máquinas de estados

N° de Traces únicos: 198 Frequência: 1.0 N° Eventos: 5210						
Máquina de estados	**Atividades**	**Estados**	**Transições**	**Estados de Aceitação**	**Sub-Autômatos**	**Estados totais**
NFA	32	5186	5185	197	0	5186
DFA	32	3480	3479	197	-	-
DFA min	32	1999	2194	1	-	-
OP Seq 3-25	32	504	699	1	235	2197
NFA CM	32	3724	4095	197	0	3724
DFA CM	32	2538	3018	197	-	-
DFA min CM	32	1308	1924	2	-	-
OP Seq 3-25	32	835	1451	2	100	1408
NFA sRet	32	3724	3723	197	0	3724
DFA sRet	32	2219	2218	188	-	-
DFA min sRet	32	1032	1209	2	-	-
OP Seq 3-25	32	363	540	2	112	1136
NFA join	32	3581	3943	188	0	3581
DFA join	32	3581	3727	188	-	-
DFA min join	32	1309	1928	2	-	-
OP Seq 3-25	32	836	1455	2	100	1409
NFA sRet join	32	3581	3580	188	0	3581
DFA sRet join	32	2219	2218	188	-	-
DFA min sRet join	32	1032	1209	2	-	-
OP Seq 3-25	32	363	540	2	112	1136
Legenda:						
	CM: Caminhos mínimos	sRet: Sem retrabalho	OP Seq: Operação sequências			

Fonte: Próprio autor (2023)

Para calcular a acurácia dos autômatos, verificamos se os *traces* da base de testes são aceitos pelos autômatos gerados a partir da base de treino e calculamos a porcentagem de aceitação. A Tabela 51 nos mostra os resultados dos cálculos das acurácias para cada modelo.

Tabela 51 – Acurácias das máquinas de estados geradas a partir da base de dados Processo 3

Referente à	**Acurácia Autômatos**	**Token based Replay**	**Alignments**
Não-Determinística	23,2558	47,1120	Estouro de memória
Determinística	23,2558	66,9450	Estouro de memória
Determinística min	23,2558	72,4069	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min	23,2558	-	-
Não-Determinística caminho mínimo	73,0546	55,7475	Estouro de memória
Determinística caminho mínimo	29,2358	74,6167	Estouro de memória
Determinística min caminho mínimo	29,2358	71,8278	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminho mínimo	29,2358	-	-
Não-Determinística sem retrabalho	36,5448	57,6628	Estouro de memória
Determinística sem retrabalho	36,5448	65,2632	Estouro de memória
Determinística min sem retrabalho	36,5448	56,9418	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min sem retrabalho	36,5448	-	-
Não-Determinística caminhos unidos	29,2358	43,8413	Estouro de memória
Determinística caminhos unidos	29,2358	47,3436	Estouro de memória
Determinística min caminhos unidos	29,2358	75,1825	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminhos unidos	29,2358	-	-
Não-Determinística caminhos unidos sem retrabalho	36,5448	61,3027	Estouro de memória
Determinística caminhos unidos sem retrabalho	36,5448	75,5009	Estouro de memória
Determinística min caminhos unidos sem retrabalho	36,5448	62,3836	Estouro de memória
Operação Sequências min/max:3-25 estados DFA min caminhos unidos sem retrabalho	36,5448	-	-

Fonte: Próprio autor (2023)

Percebe-se que, utilizando a acurácia dos autômatos, o resultado é baixo para entradas não observadas no *log* de treino. Isso se deve a complexidade dos modelos gerados somados com

a dificuldade de lidar com atividades em paralelo dos sistemas de transições. O próprio *log* de eventos utilizado é o que possui mais atividades entre os *logs* utilizados nestes experimentos com algumas dessas atividades aparecendo poucas vezes, assim caso haja um desalinhamento entre a entrada lida e o modelo, nem que seja por apenas uma transição ou um caractere da entrada, toda a entrada será considerada inválida. O número de sub-máquinas encontradas também é um número alto indicando muitas sequências unitárias, ou seja, pouca interseção entre os *traces*. Utilizando a abordagem *token based replay*, a acurácia das máquinas é bem maior que utilizando a abordagem das entradas dos autômatos, mas como não há um padrão de acurácia para os modelos, mesmo aqueles gerados a partir da determinização de um NFA ou a minização de um DFA, é levado em consideração que pode haver uma inundação de *tokens* gerando resultados de acurácia melhores do que o que realmente são. Não foi possível calcular a acurácia das máquinas usando a abordagem baseada em *alignments*, pois o processo de cálculo da acurácia é demasiado complexo, levando a um estouro de memória.

As Tabelas 52 e 53 nos mostram as tabelas comparativas do nosso menor modelo com os modelos gerados pelos algoritmos *alpha miner*, *heuristic miner* e *inductive miner*, utilizando as abordagens *token based replay* e *alignments*, respectivamente.

Tabela 52 – Tabela de comparação dos modelos acurácia *token based replay* - Processo 3

Modelo	**Places**	**Transições**	**Arcos**	**Componentes**	**Acurácia**
Alpha miner	15	32	46	47	33,7212
Heuristic miner	57	118	270	175	89,7016
Inductive miner	108	151	322	259	99,85
DFA min caminhos unidos sem retrabalho	1035	1213	2426	2248	62,3706

Fonte: Próprio autor (2023)

Tabela 53 – Tabela de comparação dos modelos acurácia *alignments* - Processo 3

Modelo	**Places**	**Transições**	**Arcos**	**Componentes**	**Acurácia**
Alpha miner	15	32	46	47	33,7212
Heuristic miner	57	118	270	175	89,7016
Inductive miner	108	151	322	259	99,85
DFA min caminhos unidos sem retrabalho	1035	1213	2426	2248	76,7562

Fonte: Próprio autor (2023)

Nestes casos, podemos notar que nosso modelo é mais complexo que os outros, pois possui muitos componentes a mais e, em relação à acurácia, nosso modelo tem um pior

desempenho do que os modelos gerados pelos algoritmos *inductive miner* e *heuristic miner*, precisamos considerar a questão que nosso modelo é maior e mesmo assim possui uma acurácia menor, devido ao modo que construímos os autômatos a partir do *log* de eventos, temos uma generalização menor. Perecebe-se que as acurácias baseadas em *token based replay* e *alignments* têm resultados melhores do que a acurácia diretamente do autômato, que pode ser acarretada pelos fatos de ou o paralelismo dos eventos não ser bem aceito em um sistema de transições, enquanto que as redes de Petri lidam melhor com este fator, ou haver uma inundação de *tokens*.

Por fim, fizemos o procedimento reverso ao transformar as redes de Petri geradas pelos algoritmos em autômatos finitos. Para isto, usamos o grafo de alcançabilidade das redes de Petri e os comparamos com o menor modelo gerado pelo nosso algoritmo. A Tabela 54 nos mostra a comparação entre os modelos.

Tabela 54 – Tabela de comparação dos modelos acurácia *autômatos* - Processo 3

Modelo	**Estados**	**Alfabeto**	**Transições**	**Acurácia**
Alpha miner	0	0	0	Estouro de tempo
Heuristic miner	0	0	0	Estouro de tempo
Inductive miner	0	0	0	Estouro de memória
DFA min caminhos unidos sem retrabalho	279	24	431	36,5448

Fonte: Próprio autor (2023)

Podemos verificar que para os modelos gerados pelos algoritmos *alpha miner* e *heuristic miner* não foi possível obter o grafo de alcançabilidade no tempo limite estabelecido de 2 horas, enquanto que, para o modelo do *inductive miner*, conseguimos obter o sistema de transições correspondente, mas ao calcular a acurácia do modelo para dados não vistos, houve um estouro de memória devido a sua complexidade.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, propomos uma abordagem que utiliza propriedades de autômatos para gerar autômatos baseados em *logs* de processos que sejam mais legíveis, como um adendo, fizemos um algoritmo inicial de decomposição de autômatos para modularizar processos. Os experimentos foram realizados em dois *logs* de eventos utilizados em competições, *Finale* e *Prepaid travel cost*, e em três *logs* de eventos reais disponibilizados pela Secretaria da Fazenda do estado do Ceará (SEFAZ-CE), Processo 1, Processo 2 e Processo 3.

Como resultados conseguimos fazer a simplificação de autômatos utilizando conceitos de caminhos felizes e retrabalho e, para cada autômato mínimo, conseguimos modularizar suas sequências de estados unitários, ou seja, que têm no máximo uma entrada e uma saída.

Utilizamos três algoritmos de descoberta, *alpha miner*, *heuristic miner* e *inductive miner*, nos mesmos dados e comparamos os modelos gerados por eles com os nossos menores modelos. A comparação leva em conta a acurácia e o número de componentes dos modelos. Para cálculo de acurácia utilizamos três abordagens, *token based replay* e *alignments* nas redes de Petri, e a acurácia dos autômatos utilizando palavras de entrada.

Os modelos gerados pelo nosso algoritmo foram capazes de serem simplificados em número de estados e de transições, o que melhora sua legibilidade, principalmente para modelos gerados a partir de *logs* de eventos não tão complexos, como os das bases de dados *Finale*. Outro ponto a favor da legibilidade de nossos modelos é que a modelagem por autômatos possui bem menos tipos de componentes do que outros tipos de modelagem, como modelos em BPMN, por exemplo. Na base de dados *Finale*, nosso menor modelo, quando convertido para rede de Petri, teve um número de componentes parecido com o do modelo gerado pelo *inductive miner*, enquanto que para todos os outros *logs* de eventos, nossos modelos possuíram muito mais componentes que os modelos gerados pelos algoritmos de descoberta. Isso nos leva a concluir que mesmo havendo uma grande diminuição do número de componentes dos nossos modelos em relação ao NFA "puro" gerado a partir do *log*, nossos modelos ainda são demasiados grandes, o que pode diminuir sua legibilidade. Por outro lado, quando convertemos os modelos gerados pelos algoritmos de descoberta para autômatos, temos que nossos autômatos são bem menos complexos do que os autômatos gerados a partir das redes de Petri, sendo que os autômatos equivalentes aos modelos gerados pelo *alpha miner* e o *heuristic miner* não puderam ser obtidos em tempo hábil de 2 horas.

Quanto à acurácia, utilizando o método *token based replay*, nosso modelo se saiu

melhor que o *alpha miner* em todos os experimentos, foi pior que os modelos do *heuristic miner* em todos os experientos e foi pior que os modelos do *inductive miner* nos *logs Finale*, Processo 1, Processo 2 e Processo 3, sendo melhor que o modelo do *inductive miner* apenas no *log Prepaid Travel Cost*. Utilizando o método *alignments*, nosso modelo foi melhor que os modelos do *alpha miner* em todos os experimentos exceto no *log* Processo 2, foi melhor que os modelos do *heuristic miner* em todos os experimentos exceto nos modelos dos *logs* Processo 1 e Processo 3, e por fim, nossos modelos foram piores em relação aos modelos gerados pelo *inductive miner* em todos os experimentos exceto no *log Prepaid Travel Cost*, pois, por ser mais complexo o cálculo dos alinhamentos, não foi possível calcular a acurácia baseada em *alignments* do modelo do *inductive miner*, havendo um estouro de memória. Por fim, quando transformamos os modelos gerados pelos algoritmos de descoberta em autômatos, percebemos que em todos os casos, nossos modelos são os únicos que conseguem dar um resultado em questão de acurácia, pois em quase todos os casos não conseguirmos transformar os modelos em autômatos em tempo hábil e, quando conseguimos, o modelo é tão complexo que o cálculo da acurácia acarreta em estouro de memória.

Como trabalhos futuros, propomos:

- Modularização de processos utilizando como base as outras formas de composição de autômatos, como paralelismo, por exemplo;
- Utilização de autômatos probabilísticos;
- Converter os autômatos diretamente para rede de Petri utilizando teoria das regiões para um melhor cálculo de acurácia;
- Realizar testes em máquinas com mais memória;

REFERÊNCIAS

AALST, W. M. van der. Everything you always wanted to know about petri nets, but were afraid to ask. In: SPRINGER. **International Conference on Business Process Management**. [S.l.], 2019. p. 3–9.

AALST, W. M. van der; RUBIN, V.; DONGEN, B. F. van; KINDLER, E.; GÜNTHER, C. W. Process mining: A two-step approach using transition systems and regions. **BPM Center Report BPM-06-30, BPMcenter. org**, v. 6, 2006.

AALST, W. V. D. Process mining: Overview and opportunities. **ACM Transactions on Management Information Systems (TMIS)**, ACM New York, NY, USA, v. 3, n. 2, p. 1–17, 2012.

AALST, W. V. D. Data science in action. In: **Process mining**. [S.l.]: Springer, 2016.

AALST, W. V. D.; ADRIANSYAH, A.; MEDEIROS, A. K. A. D.; ARCIERI, F.; BAIER, T.; BLICKLE, T.; BOSE, J. C.; BRAND, P. V. D.; BRANDTJEN, R.; BUIJS, J. *et al.* Process mining manifesto. In: SPRINGER. **International Conference on Business Process Management**. [S.l.], 2011. p. 169–194.

AALST, W. Van der; WEIJTERS, T.; MARUSTER, L. Workflow mining: Discovering process models from event logs. **IEEE transactions on knowledge and data engineering**, IEEE, v. 16, n. 9, p. 1128–1142, 2004.

BERTI, A.; ZELST, S. J. van; AALST, W. van der. Process mining for python (pm4py): bridging the gap between process-and data science. **arXiv preprint arXiv:1905.06169**, 2019.

BPI Challenges: Ieee task force in process mining. 2020. [S.I]. Online. Disponível em: https://www.tf-pm.org/competitions-awards/bpi-challenge. Acesso em: 27 mar 2023.

BRITO, R. C. de; MARTENDAL, D. M.; OLIVEIRA, H. E. M. de. Máquinas de estados finitos de mealy e moore. 2003. [S.I: s.n].

CEARá, G. do Estado do. **Secretaria da Fazenda**. 2017. Online. Disponível em: https://www.sefaz.ce.gov.br/ Acesso em: 27 mar 2023.

CORTADELLA, J.; KISHINEVSKY, M.; LAVAGNO, L.; YAKOVLEV, A. Synthesizing petri nets from state-based models. In: IEEE. **Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)**. [S.l.], 1995. p. 164–171.

CORTADELLA, J.; KISHINEVSKY, M.; LAVAGNO, L.; YAKOVLEV, A. Deriving petri nets from finite transition systems. **IEEE transactions on computers**, IEEE, v. 47, n. 8, p. 859–882, 1998.

DIEKERT, V.; KUFLEITNER, M.; STEINBERG, B. The krohn-rhodes theorem and local divisors. **Fundamenta Informaticae**, IOS Press, v. 116, n. 1-4, p. 65–77, 2012.

DIJKMAN, R. M.; DUMAS, M.; OUYANG, C. Formal semantics and automated analysis of bpmn process models. **preprint**, Citeseer, v. 7115, 2007.

DONGEN, B. F. V.; AALST, W. M. Van der. Multi-phase process mining: Building instance graphs. In: SPRINGER. **International Conference on Conceptual Modeling**. [S.l.], 2004. p. 362–376.

DONGEN, B. F. V.; MEDEIROS, A. A. D.; WEN, L. Process mining: Overview and outlook of petri net discovery algorithms. **transactions on petri nets and other models of concurrency II**, Springer, p. 225–242, 2009.

GAŽI, P.; ROVAN, B. Assisted problem solving and decompositions of finite automata. In: SPRINGER. **International Conference on Current Trends in Theory and Practice of Computer Science**. [S.l.], 2008. p. 292–303.

GOMES, L. R. M. Construção automática de ontologia em prolog a partir de modelos de processos em bpmn. 2016.

GOOGLE. **Google colab**. 2023. Online. Disponível em: https://colab.research.google.com/ Acesso em: 15 abr 2023.

HOFSTEDE, A. H. M. T.; WESKE, M. Business process management: A survey. In: **Proceedings of the 1st International Conference on Business Process Management, volume 2678 of LNCS**. [S.l.]: Springer-Verlag, 2003. p. 1–12.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Automata theory, languages, and computation**. [S.l.]: Pearson Education, Inc, 2006. v. 24.

KALENKOVA, A. A.; LOMAZOVA, I. A.; AALST, W. M. van der. Process model discovery: A method based on transition system decomposition. In: SPRINGER. **International Conference on Applications and Theory of Petri Nets and Concurrency**. [S.l.], 2014. p. 71–90.

KHERBOUCHE, O. M.; AHMAD, A.; BASSON, H. Using model checking to control the structural errors in bpmn models. In: IEEE. **IEEE 7th International Conference on Research Challenges in Information Science (RCIS)**. [S.l.], 2013. p. 1–12.

KROHN, K.; RHODES, J. Algebraic theory of machines. i. prime decomposition theorem for finite semigroups and machines. **Transactions of the American Mathematical Society**, JSTOR, v. 116, p. 450–464, 1965.

KROHN, K.; RHODES, J. L.; ARBIB, M. A. **Algebraic theory of machines, languages, and semigroups**. [S.l.], 1968.

LIBRARY, T. D. **4TU.ResearchData**: Prepaid travel cost. 2020. Online. Disponível em: https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_Prepaid_Travel_Costs/12696722. Acesso em: 27 mar 2023.

LIBRARY, T. D. **4TU.ResearchData**: Finale. 2022. Online. Disponível em: https://data.4tu.nl/ndownloader/files/23993303. Acesso em: 20 fev 2023.

MEDEIROS, A. K. A. de; WEIJTERS, A. J.; AALST, W. M. van der. Genetic process mining: an experimental evaluation. **Data Mining and Knowledge Discovery**, Springer, v. 14, n. 2, p. 245–304, 2007.

MENEZES, N. N. C. **Introdução à programação com Python–2ª edição: Algoritmos e lógica de programação para iniciantes**. [S.l.]: Novatec Editora, 2016.

MENEZES, P. B. **Linguagens formais e autômatos**. [S.l.]: Sagra-Dcluzzato, 1998.

MONTEIRO, J. C.; OLIVEIRA, A. L. Finite state machine decomposition for low power. In: **Proceedings of the 35th annual Design Automation Conference**. [S.l.: s.n.], 1998. p. 758–763.

MULLIN, A. A. Algebraic structure theory of sequential machines (j. hartmanis and re stearns). **SIAM Review**, SIAM, v. 11, n. 1, 1969.

PAIM, R.; CARDOSO, V.; CAULLIRAUX, H.; CLEMENTE, R. **Gestão de processos: pensar, agir e aprender**. [S.l.]: Bookman Editora, 2009.

RODGER, S. H.; BRESSLER, B.; FINLEY, T.; READING, S. Turning automata theory into a hands-on course. **ACM SIGCSE Bulletin**, ACM New York, NY, USA, v. 38, n. 1, p. 379–383, 2006.

ROVAN, B.; SÁDOVSKỲ, Š. On usefulness of information: Framework and nfa case. In: **Adventures Between Lower Bounds and Higher Altitudes**. [S.l.]: Springer, 2018. p. 85–99.

SIPSER, M. Introduction to the theory of computation. **ACM Sigact News**, ACM New York, NY, USA, v. 27, n. 1, 1996.

TALWAR, S. On calculating the krohn–rhodes decomposition of automata. **Advances in Applied Mathematics**, Elsevier, v. 19, n. 2, p. 251–281, 1997.

VASCONCELOS, D. R.; GUERRA, P. T. Ensinando teoria da computação com jupyter notebook. In: SBC. **Anais do XXXI Workshop sobre Educação em Computação**. [S.l.], 2023. p. 9–19.

VERBEEK, H.; BUIJS, J.; DONGEN, B. V.; AALST, W. M. van der. Prom 6: The process mining toolkit. **Proc. of BPM Demonstration Track**, sn, v. 615, p. 34–39, 2010.

WEIJTERS, A.; RIBEIRO, J. T. S. Flexible heuristics miner (fhm). In: IEEE. **2011 IEEE symposium on computational intelligence and data mining (CIDM)**. [S.l.], 2011.

WHITE, S. A. Introduction to bpmn. **Ibm Cooperation**, v. 2, n. 0, p. 0, 2004.