Universidade Federal
do Rio de Janeiro
Escola Politécnica

DESLAB PARA DESENVOLVEDORES

Daniel Ramos Garcia

Projeto de Graduação apresentado ao Curso de Engenharia de Controle e Automação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientadores: João Carlos dos Santos Basilio
Marcos Vinícius Silva Alves

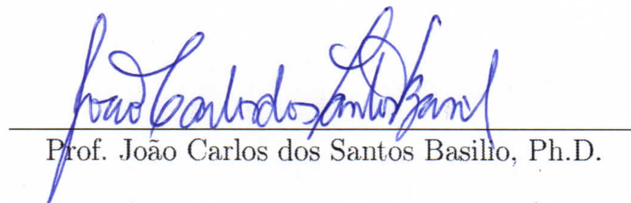
Rio de Janeiro
Setembro de 2018

DESLAB PARA DESENVOLVEDORES

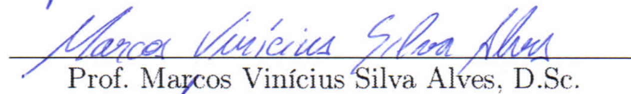
Daniel Ramos Garcia

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE AUTOMAÇÃO.

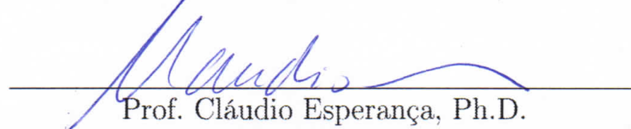
Examinado por:



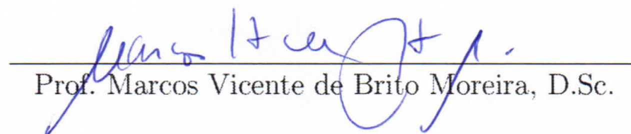
Prof. João Carlos dos Santos Basilio, Ph.D.



Prof. Marcos Vinícius Silva Alves, D.Sc.



Prof. Cláudio Esperança, Ph.D.



Prof. Marcos Vicente de Brito Moreira, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2018

Ramos Garcia, Daniel

DESlab para Desenvolvedores/Daniel Ramos Garcia. – Rio de Janeiro: UFRJ/ Escola Politécnica, 2018.

X, 93 p.: il.; 29, 7cm.

Orientadores: João Carlos dos Santos Basilio
Marcos Vinícius Silva Alves

Projeto de Graduação – UFRJ/ Escola Politécnica/ Curso de Engenharia de Controle e Automação, 2018.

Referências Bibliográficas: p. 92 – 93.

1. Sistemas a Eventos Discretos. 2. Computação Científica. 3. Autômato. I. dos Santos Basilio, João Carlos *et al.* II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Controle e Automação. III. Título.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/ UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Automação.

DESLAB PARA DESENVOLVEDORES

Daniel Ramos Garcia

Setembro/2018

Orientadores: João Carlos dos Santos Basilio
Marcos Vinícius Silva Alves

Curso: Engenharia de Controle e Automação

O programa de computação científica DESLab foi desenvolvido para auxiliar na criação de algoritmos aplicados ao estudo de sistemas a eventos discretos (SED). A linguagem *Python* foi escolhida para o desenvolvimento do DESLab por ter uma sintaxe de fácil compreensão, o que a torna mais acessível para desenvolvedores iniciantes. Desde a criação do DESLab, tanto o *Python* quanto outros softwares utilizados, sofreram atualizações, o que torna necessário o desenvolvimento de uma versão atualizada do DESLab. Um dos objetivos deste trabalho é produzir uma nova versão do DESLab, atualizada para o Python 3.6, revisada, com funções adicionais e de mais fácil instalação. Também se pretende criar um material bibliográfico que sirva de referência para futuros desenvolvedores, visando facilitar o entendimento da estrutura do código do DESLab e seus módulos.

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Engineer.

DESLAB FOR DEVELOPERS

Daniel Ramos Garcia

September/2018

Advisors: João Carlos dos Santos Basilio
Marcos Vinícius Silva Alves

Course: Automation and Control Engineering

The scientific computing program DESLab was developed to help in the creation of algorithms applied in the research of Discrete Event Systems (DES). Language Python was choosen for the development of the DESLab because it has an easy syntax, which makes it more accessible to beginer's developers. Since the creation of the DESLab, Python, as well as other used softwares, were updated, witch makes necessary an updated version of DESLab. One of the objectives of this work is to produce a new version of DESLab, updated to Python 3.6, revised, with additional functions and easy to install. It was also intended to create a bibliographic material to become a reference to future developers, seeking to easy the understanding of the DESLab structure and its modules.

Sumário

Lista de Figuras **viii**

Lista de Tabelas **x**

1 Introdução **1**

1.1 Objetivos do Trabalho de Conclusão de Curso 2

1.2 Organização do Trabalho de Conclusão de Curso 3

2 Instalação e Estrutura do DESLab **4**

2.1 Tutorial de Instalação 5

2.2 Pasta *src* 6

2.2.1 Módulo *automatadefs* 6

2.2.2 Módulo *algorithms* 21

2.2.3 Módulo *comparison* 41

2.2.4 Módulo *def_const* 44

2.2.5 Módulo *exceptions* 44

2.2.6 Módulo *graphs* 45

2.2.7 Módulo *structure* 48

2.2.8 Módulo *utilities* 61

2.3 Pasta *graphics* 62

2.3.1 Módulo *drawing* 63

2.3.2 Pasta *graphics.output* 75

2.3.3 Pasta *graphics.working* 75

2.4 Pasta *readwrite* 79

2.4.1 Módulo *inputoutput* 79

3 *Toolbox* **81**

3.1 Módulo *diagnosis* 81

3.1.1 Função *diagnoser* 81

3.1.2 Função *simplify* 82

3.1.3 Função *Gscc* 83

3.1.4 Função *Gv* 86

3.1.5 Função *is_ diagnosable* 88
3.2 Módulo *supervisory* 89
3.2.1 Funções supCont e is_controllable 89

4 Conclusão e Trabalhos Futuros **91**

Referências Bibliográficas **92**

Lista de Figuras

2.1 Autômato G, do exemplo 15. 22
2.2 Autômatos do exemplo 16. 23
2.3 Autômato G, do exemplo 17. 24
2.4 Comparação entre as funções *coac* e *trim*. 24
2.5 Autômatos gerados no exemplo 18. 27
2.6 Autômato *invproj(G)*, no qual G é o autômato do exemplo 19. 28
2.7 Exemplo da função *pclosure*. 29
2.8 Exemplo da função *langquotient*. 31
2.9 *complement(G1)*. 31
2.10 $L_m\{G1\}\backslash L_m\{G2\}$. 32
2.11 *complete(G1)*. 33
2.12 Exemplo da função *union*. 35
2.13 Autômatos gerados pela função *concatenation*. 36
2.14 Autômatos gerados pela função *paralleldet*. 39
2.15 Autômatos gerados pela função *productdet*. 40
2.16 Autômato G3, gerado pelo comando *draw(G3,'figure')*. 46
2.17 Autômatos do exemplo 35. 50
2.18 Autômatos do exemplo 36. 51
2.19 Autômato G definido no exemplo 46. 61
2.20 Estilos da função *draw*.. 65
2.21 Diagrama de processos da função *draw*. 66
2.22 Estrutura do nome do arquivo PDF. 67
2.23 Diagrama da função *tex2pdf*. 67
2.24 Diagrama de operações da função *automaton2page*. 69
2.25 Modificação dos parâmetros gráficos de G. 71
2.26 Diagrama da função *automaton2tikfig*. 72
2.27 Operações da função *auto2dot*. 73
2.28 Autômato G. 73
2.29 Estrutura dos nomes. 74
2.30 Processos do *script dot2tex_deslab*. 78

3.1 Autômato rotulador A_l. 81
3.2 Autômatos do exemplo 57. 83
3.3 Autômato obtido executando-se *simplify(Gd)*, para o autômato Gd apresentado na figura 3.2b. 84
3.4 Autômatos do exemplo 58. 85
3.5 Autômatos do exemplo 59. 87
3.6 Autômatos do exemplo 61. 90

Lista de Tabelas

2.1 Subpacotes e módulos do DESLab. 4
2.2 Funções e classes do módulo *automatadefs*. 7
2.3 Atributos da classe *fsa*. 8
2.4 Métodos importados do módulo *structure*. 14
2.5 Métodos de verificação. 14
2.6 Atributos que podem ser modificados pelo método *setpar*. 15
2.7 Variáveis em *def_const*. 44
2.8 Classes de erro. 45
2.9 Variáveis Predefinidas. 63
2.10 Complemento da *string figure_texcode* para cada estilo. 69
2.11 Parâmetros da Classe *graphic*. 69
2.12 Parâmetros fixados para cada estilo. 70
2.13 Dicionário do analisador. 77
2.14 Parâmetros da função *save*. 80

3.1 Estados de G_{scc}. 88

Capítulo 1

Introdução

A computação científica tem como objetivo a utilização de ferramentas computacionais para modelar e solucionar problemas científicos em diversas áreas de pesquisa [1]. Criado por Guido van Rossum em 1990, o *Python* é uma linguagem de programação interpretada (*script*) de alto nível, muito utilizada em pesquisas científicas.

Além do Python ser uma linguagem orientada a objeto e escalável, o *Python* foi desenvolvido de forma que a sua sintaxe se tornasse amigável e de fácil compreensão. Isso faz com que o Python seja uma ótima opção de linguagem para iniciantes. Por se tratar de uma linguagem de código aberto, gerenciada pela fundação sem fins lucrativos *Python Software Foundation* [2], e ser compatível com diversos sistemas operacionais, o *Python* possui uma grande comunidade de usuários, o que facilita o acesso à tutoriais, documentações e diversas bibliotecas.

O DESLab [3] é uma ferramenta de computação científica criada, na linguagem *Python*, para ajudar no desenvolvimento de algoritmos para sistemas a eventos discretos (SED) modelados por autômatos.

A linguagem *Python* possui diversos pacotes desenvolvidos para a computação científica, que se tornaram valiosas ferramentas no desenvolvimento das funções e classes do DESLab. Dentre os módulos disponíveis na biblioteca *Python* utilizados no DESLab, o principal módulo é o NetworkX [4]. O NetworkX é um módulo desenvolvido para a criação e manipulação de redes complexas. No DESLab, ele é utilizado na construção de um grafo que armazena informações como os nomes dos estados e dos eventos, e as transições de um autômato.

Além dos módulos do *Python*, o *software* Graphviz [5] e uma distribuição do LaTeX são utilizados pelo DESLab. O Graphviz é um *software* de código aberto desenvolvido para gerar imagens de estruturas e redes de grafos. A interação entre o DESLab e o Graphviz ocorre pelo envio das informações de um autômato construído com o DESLab para que o Graphviz possa gerar uma figura do diagrama de transição de estados desse autômato. Quando os diagramas de transição de estados dos

autômatos são gerados pelo Graphviz, o código da imagem é convertido para LaTeX e compilado usando uma distribuição do LaTeX, para gerar um arquivo PDF com os digramas de transição de estados dos autômatos. A distribuição do LaTeX usada durante a execução desse trabalho foi o TexLive [6].

Com a constante evolução da linguagem *Python*, novas atualizações foram surgindo, tanto para os pacotes da linguagem quanto para o próprio *Python*. Por recomendações dos desenvolvedores da linguagem *Python*, todos os programas desenvolvidos nas versões *2.x* do Python devem ser migrados para as versões *3.x*, pois as versões *2.x* serão descontinuadas em 2020 [7]. Como o DESLab foi inicialmente desenvolvido na versão 2.7 do *Python*, uma migração para uma versão mais recente é extremamente necessária para que o DESLab não se torne obsoleto. Além da necessidade dessa atualização, problemas e erros em algumas funções e operações foram reportados por usuários do DESLab. o que sugere a necessidade de uma ampla revisão das funções implementadas no DESLab. Um dos problemas reportados por diversos usuários consistia na dificuldade de instalação do DESLab em um computador. Muitas vezes, o pacote não era instalado corretamente, gerando conflitos na interação deste com os módulos do Python e/ou com os softwares Graphviz e TexLive.

1.1 Objetivos do Trabalho de Conclusão de Curso

Um dos objetivos desse trabalho de conclusão de curso é apresentar uma nova versão do DESLab atualizada para a versão 3.6 do Python, uma vez que, alguns módulos do Python utilizados pelo DESLab, como, por exemplo, o NetworkX, sofreram atualizações significativas desde a versão 2.7. Pretende-se, também, corrigir os problemas identificados durante o processo de atualização e aqueles relatados por usuários da versão antiga do DESLab.

Em relação ao processo de instalação da nova versão do DESLab, pretende-se torná-lo mais simples. Para isso, será disponibilizado um instalador para essa nova versão, acompanhado de um guia de instalação, no qual se detalha cada etapa desse processo. Além disso, o pacote de instalação irá conter todos os componentes (softwares e módulos do Python) necessários para o correto funcionamento do DESLab, visando, com isso, evitar problemas de compatibilidade entre o DESLab e as versões instaladas desses componentes.

Com o objetivo de ampliar a aplicabilidade do DESLab, dois novos módulos serão acrescentados à nova versão do DESLab. Um desses módulos possui funções voltadas à solução de problemas de diagnose de falhas de sistemas a eventos discretos. O segundo módulo é composto por funções que podem ser aplicadas no contexto do problema de controle supervisório de sistemas a eventos discretos.

Esse trabalho também tem como objetivo criar uma base bibliográfica para orientar futuros desenvolvedores do DESLab sobre a estrutura das suas pastas, módulos, classes e funções, como, também, seus funcionamentos e a forma como interagem entre si e com softwares de terceiros.

1.2 Organização do Trabalho de Conclusão de Curso

Este trabalho de conclusão de curso está organizado da seguinte forma. No capítulo 2, é apresentada toda a estrutura da do DESLab, seus pacotes, módulos, classes e funções. Nas seções do capítulo 2, a descrição e cada função é feita de modo que fiquem claras as interações entre módulos e a finalidade de cada um deles. Nessas seções, são apresentados exemplos para auxiliar nas descrições das funções descritas. No capítulo 3, são apresentados os novos módulos, de diagnose de falhas e de controle supervisório, criados para o DESLab. Por fim, no capítulo 4, são apresentadas as conclusões e possíveis trabalhos futuros.

Capítulo 2

Instalação e Estrutura do DESLab

O pacote DESLab possui um arquivo de inicialização (_init_.py), um arquivo de documentação (version.py) e pastas (subpacotes) contendo seus arquivos de inicialização e alguns módulos, conforme apresentado na tabela 2.1. A divisão dos módulos em pastas facilita a localização de uma determinada função que se queira acessar o código fonte, uma vez que os nomes das pastas já transmitem uma informação inicial da aplicação das funções em seus módulos.

Tabela 2.1: Subpacotes e módulos do DESLab.

Caminho da Pasta	**Módulos**
deslab	_init_ version
deslab.graphics	_init_ drawing
deslab.graphics.output	-
deslab.graphics.working	_init_ dotparsing dot2tex_deslab
deslab.readwrite	_init_ inputoutput
deslab.src	_init_ algorithms automatadefs comparison def_const exceptions graphs structure utilities
deslab.toolbox	_init_ diagnosis supervisory

Neste capítulo será, inicialmente, apresentado um tutorial de instalação da nova versão do DESLab e, em seguida, cada uma das pastas do DESLab será descrita, com

foco no conteúdo de seus módulos. O objetivo será detalhar o funcionamento de cada uma das classes e funções envolvidas nas operações do pacote para que, futuramente, um usuário que deseje implementar novas ferramentas possa compreender melhor o código das funções do DESLab.

2.1 Tutorial de Instalação

Com o objetivo de simplificar o processo de instalação do DESLab, o seu pacote de distribuição contém todos os módulos e softwares necessários para a sua instalação sem a necessidade de conexão do computador com a internet. A versão atual do DESLab foi configurada e testada nos sistemas operacionais Windows 7 e Windows 10. Nessa seção vamos apresentar os passos necessários para a instalação do DESLab.

Caso você não possua o *Python* 3.6 ou possua outras versões do *Python* instaladas no seu computador, certifique-se dos seguintes pontos antes de instalar o DESLab:

- Remover caminhos de versões antigas do *Python* das variáveis de ambiente do *Windows*.
- Instale o *Python* 3.6. Um instalador do *Python* 3.6 foi incluído na pasta *"programas"* do pacote do DESLab. Durante a instalação, existe a opção de incluir o *Python* nas variáveis de ambiente automaticamente, isso evitará a necessidade de executar o passo a seguir.
- Inclua o caminho do *Python* 3.6 nas variáveis de ambiente do *Windows*.
- Instale o *TexLive*. Um instalador do *TexLive* foi incluído na pasta "*programas*" do pacote do DESLab.
- Inclua o caminho do *TexLive*, normalmente em C:\...\Texlive\2017\bin\win32, nas variáveis de ambiente do *Windows*.

Para instalar o DESLab e seus módulos, basta abrir (duplo clique) o arquivo *"Install.bat"* na pasta *"DESLab"*. Os seguintes módulos serão instalados em conjunto com o DESLab:

- Graphviz 2.28
- FaDo 1.3.5.1
- Future 0.16.0
- NetworkX 2.1
- Pyparsing 2.2.0

- Pydot 1.2.4

Após a instalação, certifique-se de que os caminhos para as seguintes pastas foram incluídos nas variáveis de ambiente do Windows:

- ...Python\Python36-32
- ...Graphviz\bin
- ...Texlive\2017\bin\win32

Incluindo/removendo caminhos nas variáveis de ambiente do Windows

1. Acesse *"Painel de Controle\Sistema e Segurança\Sistema"* ou clique com o botão direito do mouse em *"Computador"*, no menu *"Iniciar"*, e acesse em *"Propriedades"*.
2. Acesse as *"Configurações avançadas do sistema"*. Na janela que abrirá, clique em *"Variáveis de Ambiente"* na aba *"Avançado"*.
3. No segmento *"variáveis do sistema"*, encontre a variável *"PATH"*, clique nela e, em seguida, clique em *"Editar..."*.

No segmento *"Valor da variável"* os diversos caminhos contidos são separados por ";" (ponto e vírgula). Para remover um caminho, basta apagá-lo, e para incluir um caminho, basta colar o endereço, lembrando de usar o separador ";". Para incluir um caminho na variável de ambiente do *Windows*, deve-se fornecer o caminho para a pasta que contém o arquivo, e não o caminho completo do arquivo.

2.2 Pasta *src*

Na pasta *src* estão os módulos estruturais do DESLab. Todos os módulos responsáveis pela criação, manipulação e operações entre autômatos são encontrados na pasta *src*. Nesta seção, serão apresentados os módulos da pasta *scr*, mostrados na tabela 2.1, e o funcionamento das classes e funções contidas neles.

2.2.1 Módulo *automatadefs*

O módulo *automatadefs* é a parte fundamental do DESLab, pois ele contém alguns dos principais componentes para a base da criação de um autômato. Todas as funções e classes definidas no módulo *automatadefs*, ilustradas na tabela 2.2, são responsáveis pela construção de um autômato definido no DESLab.

Tabela 2.2: Funções e classes do módulo *automatadefs*.

Funções e classes	**Descrição**
fsa	Classe *finite-state automaton*, que define um autômato
create_graph	Função que cria um grafo do autômato a partir de um objeto do módulo NetworkX
create_FSA_transdicts	Função que cria os dicionários das transições do autômato
verify_fsa_definition	Função que valida o estado inicial do autômato
create_table	Função que constrói um dicionário para os nomes dos estados em LaTeX

Classe *fsa*

Todo autômato criado no DESLab é um objeto da classe *fsa*, nela estão definidos todos os valores necessários para a construção do autômato. Para criar um objeto dessa classe deve-se definir todos os elementos básicos que compõem um autômato [8], quais sejam:

- Conjunto de estados;
- Conjunto de eventos;
- Transições;
- Estados iniciais;
- Estados marcados;
- Eventos observáveis (opcional);
- Eventos controláveis(opcional);

Além dos elementos básicos do autômato, a classe *fsa* também possui alguns atributos adicionais de configuração. Todos os atributos de um objeto da classe *fsa* são apresentados na tabela 2.3.

No construtor da classe *fsa* todos os parâmetros possuem valores predefinidos, de tal forma que, caso nenhum argumento seja passado durante a criação de um objeto da classe *fsa*, um autômato vazio será criado.

Os argumentos que são passados para o construtor da classe *fsa* para criar um autômato são: *X, Sigma, transition, X0, Xm, table, Sigobs, Sigcon, name* e *graphic*. Inicialmente, o construtor da classe verifica os argumentos passados para *X, X0* e *Sigma*, e, caso algum deles seja vazio, o autômato será configurado como vazio. Nessa configuração, os atributos *X, X0, Xm* e *Sigma* serão mudados para *EMPTYSET*, o atributo *empty* será mudado para *True* e o atributo *name* será mudado para a *string* *'Empty Automaton'*. Caso os argumentos passados para *X, X0* e *Sigma* não sejam vazios, os atributos receberão os seus respectivos valores passados como argumentos.

Tabela 2.3: Atributos da classe *fsa*.

Atributo	Descrição
X	Conjunto de estados
Sigma	Conjunto de eventos
Xm	Conjunto de estados marcados
X0	Conjunto com o estado inicial
name	Nome do autômato
empty	Se o autômato for vazio, o valor do atributo será *True*, caso contrário, será *False*
Sigobs	Conjunto dos eventos observáveis
Sigcon	Conjunto dos eventos controláveis
Graph	Grafo do autômato, um objeto da classe *MultiDiGraph* do NetworkX
gammaDict	Dicionário de eventos ativos
deltaDict	Dicionário do alcance dos estados
infoDict	Dicionário de informações do autômato
symDict	Dicionário dos nomes dos estados e eventos do autômato em LaTeX
graphic	Objeto da classe *graphic*

Alguns valores precisam de uma verificação adicional antes de serem atribuídos aos seus respectivos atributos, conforme descrito a seguir:

- O atributo *X0* recebe o valor retornado pela função *verify_fsa definition*;
- Caso os valores para *Sigobs* e *Sigcon* não sejam passados, eles serão iguais ao atributo *Sigma*;
- Caso as listas passadas para *Sigobs* e *Sigcon* contenham eventos que não estejam em *Sigma*, esses eventos serão descartados;
- O atributo *Graph* receberá o valor retornado pela função *create_graph*;
- Os atributos *gammaDict, deltaDict* e *infoDict* receberão os valores retornados pela função *create_fsa_transdicts*;
- O atributo *symDict* receberá o valor retornado pela função *create_table*.

A classe *fsa* possui diversos métodos que serão apresentados nessa seção. O exemplo 1 ilustra a criação de objetos da classe *fsa*.

Exemplo 1
Existem diversas formas de se criar um objeto da classe fsa, uma vez que seus parâmetros possuem valores predefinidos. Alguns exemplos podem ser vistos no código abaixo.

```
>>> syms('a b c f u')
>>> X = [1,2,3,4,5,6]
>>> Sigma = [a,b,c,f,u]
>>> X0 = [1]
>>> Xm = [4,5,6]
```

```
>>> T =[(1,c,2), (2,a,3),(3,b,2),(2,f,4),(4,a,5),(5,b,4),(5,a
    ,5),(5,u,6),(6,a,6)]
>>> table = [(a,r'\alpha'),(b,r'\beta')]
>>> Sigobs=[a,b,c]
>>> Sigcon = [a,b]
>>> graph = graphic()
>>> G = fsa(X,Sigma,T,X0,Xm,table,Sigobs,Sigcon,'$G$',graph)
>>>
>>> G2=fsa()
>>> G2.name
'Empty Automaton'
>>>G3 = fsa(X=[1,2,3],Sigma=[a,b],T=[(1,a,2),(1,b,3)],X0=[1],
    Xm=[2],name = '$G3$')
```

Métodos Especiais

Na linguagem Python, os métodos especiais são definidos com dois símbolos "__" (subtraço duplo) no início e no final do seu nome. Os métodos especiais são funções predefinidas na linguagem, que podem ser inseridas nas classes criadas pelo usuário e configuradas para operar de uma forma especifica, quando chamadas com objetos dessa classe. Um exemplo dessas funções são os operadores lógicos e matemáticos. Muitos métodos especiais da classe *fsa* utilizam funções do módulo *algorithms*, os quais são descritos na subseção 2.2.2.

Método Especial *iter*

O método *iter* foi configurado no DESLab para criar um iterador dos estados do autômato. Com a definição desse método é possível percorrer os estados de um autômato de formas mais simples, utilizando o objeto da classe *fsa* ou passando-o como argumento para a função *iter*, como mostrado no exemplo 2.

Exemplo 2
Seja G o autômato definido no DESLab pelo código abaixo:

```
syms('x1 x2 x3 x4 a b')
table = [(x1,'x_1'),(x2,'x_2'),(x3,'x_3'),(x4,'x_4'),(a,r'\
    alpha'),(b,r'\beta')]
X = [x1,x2,x3,x4]
Sigma = [a,b]
X0 = [x1]
Xm = [x2,x3]
T =[(x1,a,x2),(x1,b,x4),(x2,b,x3),(x3,a,x2),(x4,b,x4),(x4,a,x4
    )]
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')
```

Podemos iterar os estados do autômato das seguintes formas:

```
>>> G.X
```

```
frozenset({'x1', 'x4', 'x3', 'x2'})
>>> states = iter(G)
>>> next(states)
'x1'
>>> next(states)
'x4'
>>> next(states)
'x3'
>>> next(states)
'x2'
>>> for i in G:
        print(i)


x1
x4
x3
x2
>>> states = iter(G)
>>> for i in states:
        print(i)


x1
x4
x3
x2
```

Método Especial *len*

Ao definir o método *len* na classe *fsa*, torna-se possível passar um objeto da classe como argumento para essa função, que normalmente é utilizada para contar os elementos de listas, *strings*, tuplas, etc. Para um objeto da classe *fsa*, a função *len* retornará o número de estados do autômato.

Método Especial *str*

A função *str* é, normalmente, utilizada para transformar uma variável em *string*. Definida como método da classe *fsa*, ela permite que, ao se passar um autômato como argumento para a função *str*, seja retornada uma *string* contendo informações sobre esse autômato. A definição do método *str* na classe também influencia o uso da função *print* que, quando chamada, invoca a função *str*. Com isso, ao definirmos o método *str* na classe *fsa* também será possível passar o autômato como argumento para a função *print*, conforme ilustrado no exemplo 3.

Exemplo 3

Seja G o autômato definido no exemplo 2. Podemos usar a função str das seguintes

formas:

```
>>> str(G)
'finite state machine: $G$ \n          number of states: 4 \n
            number of events: 2 \n          number of
   transitions 6 \n          '
>>> tex = str(G)
>>> print(tex)
finite state machine: $G$
number of states: 4
number of events: 2
number of transitions 6

>>> print(G)
finite state machine: $G$
number of states: 4
number of events: 2
number of transitions 6
```

Método Especial delta

O método especial delta não faz parte das funções predefinidas do Python, ele é um método especial próprio da classe *fsa*. O método delta é utilizado por muitas funções do DESLab, ele recebe um estado x e um evento σ como argumentos e retorna o estado alcançado pela transição do estado x rotulada por σ. Caso o estado x e/ou o evento σ não existam ou o evento σ não seja habilitado no estado x, uma mensagem de erro é retornada. O exemplo 4 ilustra com podemos utilizar esse método.

Exemplo 4
Seja G o autômato definido no exemplo 2. Podemos usar o método delta da seguinte forma:

```
>>> G.__delta__(x1,b)
frozenset({'x4'})
>>> G.__delta__(x3,b)
frozenset()
```

Método Especial *and*

O método especial *and* configura a função que é chamada quando se utiliza o operador "&". Dessa forma, esse operador é sobrecarregado para funcionar de uma forma específica quando utilizado em objetos da classe *fsa*. No DESLab, o operador "&" chama a função *product*, do módulo *algorithms*, e retorna o produto de dois autômatos.

Métodos Especiais *add* e *or*

O método especial *add* configura o operador "+". No DESLab esse operador é sobrecarregado para calcular um autômato que marca a união das linguagens marcadas por dois ou mais autômatos. Para essa operação, a função *union* do módulo *algorithms* é utilizada.

Da mesma forma que o método *add*, o método especial *or* está associado a função *union*. O operador sobrecarregado para esse método é o "|".

Método Especial *floordiv*

O método especial *floordiv* sobrecarrega o operador "//" para executar a composição paralela de dois autômatos. A função executada nessa operação é função parallel importada do módulo *algorithms*.

Método Especial *mul*

O método especial *mul* está associado ao operador "*" que, no DESLab, foi sobrecarregado para efetuar a concatenação das linguagens marcadas de dois autômatos. A função utilizada para esse processo foi a função *concatenation*, importada do módulo *algorithms*, que será apresentado na subseção a seguir junto com todas as instruções de utilização da função e do operador.

Método Especial *truediv*

O método especial *truediv* configura o operador "/" para calcular o quociente das linguagens marcadas de dois autômatos. A função utilizada para efetuar a operação é a função *langquotient* do módulo *algorithms*.

Método Especial *invert*

Na classe *fsa*, o método especial *invert* modifica a função do operador "∼" para calcular o complemento da linguagem marcada de um autômato utilizando a função *complement* do módulo *algorithms*.

Método Especial *sub*

O método especial *sub* sobrecarrega o operador -". Na classe *fsa*, ele utiliza a função *langdiff* do módulo *algorithms*, para calcular a diferença entre as linguagens marcadas por dois autômatos.

Método Especial *le*

No DESLab, o método especial *le*, que sobrecarrega o operador "<=", é configurado para comparar as linguagens marcadas por dois autômatos e determinar se a linguagem marcada do primeiro está contida na linguagem marcada do segundo. Esse método importa a função *issublanguage*, do módulo *comparison*, descrita na subseção 2.2.3.

Método Especial *ge*

De forma análoga ao método especial *le*, o método especial *ge* sobrecarrega o operador ">=" para verificar se a linguagem marcada pelo segundo autômato é uma sub-linguagem da linguagem marcada pelo primeiro.

Método Especial *eq*

O método especial *eq* utiliza a função are_automataequal para verificar se dois autômatos são iguais por meio do operador "==". A função are_automataequal pertence ao módulo *comparison*, descrito na subseção 2.2.3.

Método Especial *ne*

O método especial *ne* utiliza a mesma função que o método especial *eq* para verificar se dois autômatos são diferentes, ele sobrecarrega o operador "! =" para realizar essa comparação.

Métodos Importados

Assim como os métodos especiais utilizam importam funções definidas em outros módulos, as funções do módulo *structure* listadas na tabela 2.4 foram importadas para a classe *fsa*. As descrições dessas funções e as instruções de como utilizá-las podem ser encontradas na subseção 2.2.7.

Método *info*

O método *info* foi criado para retornar informações sobre um autômato. Essas informações estão em um dicionário armazenado no atributo *infoDict*, o qual contém as seguintes chaves:

- isDFA: informa se o autômato é finito e determinístico;
- *hasEpsilon*: informa se o autômato possui o evento ϵ;
- *nonDetTrans*: informa se o autômato possui transições não determinísticas.

Tabela 2.4: Métodos importados do módulo *structure*.

Método	Comentário
deletetransition	Remove transições do autômato
addtransition	Adiciona transições ao autômato
addevent	Adiciona eventos ao autômato
deletevent	Remove eventos do autômato
addstate	Adiciona estados ao autômato
deletestate	Remove estados do autômato
renametransition	Modifica o evento associado à uma transição
addselfloop	Adiciona um auto-laço à um estado
transitions	Retorna uma lista das transições de um autômato
transitions_iter	Retornar um iterador das transições de um autômato
renamevents	Renomeia eventos de um autômato
renamestates	Renomeia estados de um autômato

O método *info* retorna uma tupla com essas informações, na mesma ordem em que foram apresentadas acima. Os valores contidos na tupla serão apenas *True* ou *False*.

Métodos de Verificação

Os métodos de verificação são métodos criados para retornar as informações do atributo infoDict. Diferente do método *info*, eles retornam essas informações separadamente. Os métodos de verificação são apresentados na tabela 2.5.

Tabela 2.5: Métodos de verificação.

Método	Chave do atributo *infoDict* retornada
is_dfa	isDFA
has_epsilon	hasEpsilon
has_nondetrans	nonDetTrans

Método *tmx*

O método *tmx* cria uma matriz com as transições de estados de um autômato, tornando simples a visualização do alcance de cada estado. O método possui o parâmetro *table* que, por padrão, tem valor nulo. Quando nenhum argumento é passado, o método retorna uma lista contendo todos os dados da matriz e transição de estados do autômato. Caso a string *'table'* seja passada como argumento, a matriz de transição de estados será retornada em forma de mensagem. O exemplo 5 mostra como o método também pode ser usado.

Exemplo 5

Seja G o autômato definido no exemplo 2. A matriz de transição de estados de G pode ser gerada das seguintes formas:

```
>>> G.tmx()
[['', 'b', 'a'], ['x4', 'x4', 'x4'], ['x2', 'x3', 'N/D'], ['x1
    ', 'x4', 'x2'], ['x3', 'N/D', 'x2']]
```

```
>>> G.tmx('table')

Transition Matrix:

     b    a
x4   x4   x4
x2   x3   N/D
x1   x4   x2
x3   N/D  x2
```

Método *setpar*

O método *setpar* foi criado com a finalidade de facilitar a modificação de atributos do autômato, uma vez que alterações nessas variáveis podem exigir modificações em outros atributos do objeto da classe *fsa*. A alteração direta dos estados, eventos e de outros atributos de um autômato pode resultar em erros, por isso o método *setpar* deve ser usado para esse fim.

Exemplo 6
Seja o autômato G, definido no exemplo 2, a modificação de um de seus atributos, pelo método setpar, é feita com o código abaixo.

```
>>> syms('c')
['c']
>>> sig = [a,b,c]
>>> G.Sigma
frozenset({'b', 'a'})
>>> G=G.setpar(Sigma = sig)
>>> G.Sigma
frozenset({'c', 'b', 'a'})
```

É importante observar que o método *setpar* só pode ser utilizado para modificar alguns atributos, os quais são apresentados na tabela 2.6. Para outras modificações como, renomear estados e eventos ou adicionar transições, deve-se utilizar os métodos importados do módulo *structure*.

Tabela 2.6: Atributos que podem ser modificados pelo método *setpar*.

Atributos	Comentários
Xm	Conjunto de Estados marcados
Sigma	Conjunto de Eventos
X0	Conjunto com os estados iniciais
name	Nome do autômato
Sigcon	Conjunto de eventos controláveis
Sigobs	Conjunto de eventos observáveis
table	Dicionário para os nomes dos estados e eventos em LaTeX

Método *setgraphic*

O método *setgraphic* modifica o atributo *graphic* do autômato. Como esse atributo é um objeto da classe *graphic*, para modificá-lo, o método possui os mesmos parâmetros que o construtor da classe *graphic*, bem como os seus valores predefinidos. Dessa forma, o usuário só precisará passar os argumentos para os parâmetros que ele desejar modificar. Exemplos da utilização do método *setgraphic* podem ser vistos na descrição da classe *graphic*, na seção 2.3.1.

Método *unobsreach*

O método *unobsreach* retorna o alcance não observável de um estado do autômato. Para isso ele recebe como argumentos um conjunto de estados e uma lista com os eventos observáveis. O conjunto de estados pode ser passado como uma *string* (caso seja um único estado), uma lista ou um *set*, e a lista de eventos observáveis também pode ser passada como um *set*. Caso a lista de eventos observáveis não seja passada, o método utilizará o atributo *Sigobs* do objeto *fsa*.

O conjunto de eventos ativos dos estados passados é obtido utilizando a função *Gamma*. Esse conjunto é comparado com a lista de eventos observáveis para que, utilizando o método especial *delta*, apenas os estados alcançados, a partir dos estados passados como argumento, por meio de transições rotuladas por eventos não-observáveis sejam encontrados. Ao fim do processo, um *set* com os estados encontrados é retornado. O exemplo abaixo mostra como utilizar o método *unobsreach*.

Exemplo 7
Seja Go autômato definido no exemplo 2. Como G não possui eventos não observáveis, podemos passar o conjunto como argumento para o método unobsreach ou utilizar o método setpar para definir os eventos observáveis.

```
>>> G.unobsreach(x2,[a])
frozenset({'x2', 'x3'})
>>> G.unobsreach(x1,[a])
frozenset({'x4', 'x1'})
>>> G = G.setpar(Sigobs=[a])
>>> G.unobsreach(x2)
frozenset({'x2', 'x3'})
>>> G.unobsreach(x1)
frozenset({'x4', 'x1'})
```

Método delta

O método *delta* recebe um estado *state* e um evento *event* como argumentos e retorna o(s) estado(s) alcançado(s) a partir de *state* pela transição rotulada por

event. Inicialmente verifica-se se o autômato é determinístico usando-se o método *is_dfa*. Em seguida, verifica-se a validade do evento passado checando-se se ele pertence ao conjunto de *Sigma*, e utiliza-se o método *Gamma* para conferir se ele é ativo para o estado passado. Finalmente, acessando-se o dicionário do atributo *deltaDict*, o estado alcançado é retornado. Caso o autômato não seja determinístico, o valor retornado pode ser um conjunto de estados. O exemplo abaixo mostra como o método deve ser utilizado.

Exemplo 8
Considere autômato G definido no exemplo 2, e o autômato não determinístico G2, criado a partir de G, conforme descrito no código abaixo.

```
>>> G2=G.addtransition((x2,b,x4))
>>> G.delta(x2,b)
'x3'
>>> G2.delta(x2,b)
frozenset({'x4', 'x3'})
```

Note que, nos dois autômatos, podemos utilizar o método delta da mesma forma.

Método *deltaobs*

O método *deltaobs*, assim como o método *delta* retorna o alcance de um estado para um determinado evento. A diferença desse método é que, além de receber o estado e o evento, ele também deve recebe uma lista de eventos observáveis, assim, o valor retornado será a lista com o alcance observável desse estado.

Método *Gamma*

O método *Gamma* recebe um estado como argumento e, a partir dele, determina quais são os eventos ativos para ele. Para obter essa resposta, uma consulta é feita ao atributo *gammaDict*, que é um dicionário com os eventos ativos de cada estado.

Exemplo 9
Seja G o autômato definido no exemplo 2. Podemos utilizar o método Gamma da seguinte forma:

```
>>> G.Gamma(x2)
frozenset({'b'})
>>> G.Gamma(x4)
frozenset({'b', 'a'})
```

Método *copy*

O método *copy* cria uma cópia do autômato utilizando a função *deepcopy* do módulo *copy* do *Python*. Esse método foi criado, principalmente, para ser utilizado pelos

diversos métodos da classe *fsa*, para que as modificações que eles realizem não afetem o autômato original. O exemplo abaixo ilustra como utilizar o método.

Exemplo 10

Seja G um autômato. Para criar uma cópia de G, utilizamos o método copy da seguinte forma.

```
>>> G2 = G.copy()
>>> G
<deslab.src.automatadefs.fsa object at 0x0404F0F0>
>>> G2
<deslab.src.automatadefs.fsa object at 0x03D20E10>
```

Função *create_graph*

A função *create_graph* utiliza a classe *MultiDiGraph* do módulo NetworkX para criar um grafo que conterá as informações sobre os estados, eventos e transições do autômato. Para que esse grafo seja criado, a função *create_graph* deve receber três listas, com as transições do autômato, os seus estados e os seus eventos, como pode ser visto no exemplo 11. O objeto criado pela função *create_graph* será armazenado no atributo *Graph* de um objeto da classe *fsa* para, posteriormente, ser utilizado na criação do diagrama de transição de estados desse objeto (autômato), cuja construção será apresentada na seção 2.2.

Exemplo 11

Para criar um grafo com a função create_graph, devemos proceder como descrito no código abaixo.

```
>>> from deslab import *
>>> syms('x y z a b')
['x', 'y', 'z', 'a', 'b']
>>> X=[x,y,z]
>>> Sigma = [a,b]
>>> T=[(x,a,y),(x,b,z),(y,a,y),(y,b,z)]
>>> grafo = create_graph(T,X,Sigma)
>>> grafo
<networkx.classes.multidigraph.MultiDiGraph object at 0
   x03CAFEF0>
```

Função *create_FSA_transdicts*

A função *create_FSA_transdicts* foi desenvolvida para ser a construtora de 3 atributos da classe *fsa*:

- *gammaDict*: dicionário contendo os eventos ativos de cada estado;

- *deltaDict*: dicionário com o alcance de cada estado;

- *infoDict*: dicionário de informações sobre o autômato.

Ao chamar a função *create_FSA_transdicts*, devem ser passados como argumentos o atributo *Graph* de um objeto da classe *fsa* e suas listas de estados, eventos e estados iniciais. A construção dos dicionários retornados pela função *create_FSA_-transdicts* segue os seguintes passos:

1. O dicionário *infoDict* é criado com os valores padronizados para um autômato determinístico, ou seja, *infoDict = {'nonDetTrans': False, 'isDFA': True, 'hasEpsilon': False}*;

2. Verifica-se quantos elementos a lista de estados iniciais possui, caso ela possua mais de um elemento, as chaves *isDFA* e *nonDetTrans*, do dicionário *infoDict*, são modificadas para *False* e *True*, respectivamente;

3. Cria-se os dicionários *gammaDict* e *deltaDict*;

Os valores dos dicionários *gammaDict* e *deltaDict* são construídos pela função *determineGamma* para cada estado do grafo passado como argumento. A função *determineGamma* é definida internamente na função *create_FSA_transdicts*. Ela deve receber como argumentos um estado e o dicionário *infoDict*. A partir do grafo passado para a função *create_FSA_transdicts* a função *determineGamma* verifica as transições do estado que recebeu e procede da seguinte forma;

- Caso alguma dessas transições não seja determinística, ela modifica as chaves *isDFA* e *nonDetTrans* do dicionário para *False* e *True*, respectivamente, e, adicionalmente, se houver uma transição rotulada por ε, a chave *hasEpsilon* é alterada para *True*.

- Cria um dicionário cuja chave será o evento da transição e os elementos serão os estados alcançados. Esse dicionário é inserido no dicionário *gammaDict*.

- Cria uma lista com os eventos ativos que é inserida no dicionário *deltaDict*.

O exemplo a seguir apresenta os dicionários *infoDict, gammaDict* e *deltaDict* construídos pela função *create_FSA_transdicts* para um dado autômato.

Exemplo 12
Seja o autômato G, definido pelo seguinte código no DESLab.

```
syms('x y z a b')
X = [x,y,z]
table=[]
Sigma = [a,b]
X0 = [x]
Xm = []
T =[(x,a,y),(x,b,z)]
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')
```

Chamando a função create_FSA_transdicts, teremos os seguintes valores retornados:

```
>>> gammaDict,deltaDict,infoDict = create_FSA_transdicts(G.
    Graph,G.X,G.Sigma,G.X0)
>>> gammaDict
{'x': frozenset({'a', 'b'})}
>>> deltaDict
{'x': {'a': frozenset({'y'}), 'b': frozenset({'z'})}}
>>> infoDict
{'nonDetTrans': False, 'isDFA': True, 'hasEpsilon': False}
```

Função *verify_fsa_definition*

A função *verify_fsa_definition* é utilizada pelo construtor da classe *fsa* antes de armazenar o valor do estado inicial no atributo *X0*. Ela recebe como argumentos a lista de estados de um autômato, a lista de eventos, a lista de estados marcados e a lista dos estados iniciais. A função *verify_fsa_definition* verifica se tanto os estados marcados quanto os estados iniciais estão contidos na lista de estados e se os estados iniciais passados são válidos. Se algum dos testes falhar, uma mensagem de erro será retornada, caso contrário, o conjunto contendo o estado inicial é retornado.

Função *create_table*

A função *create_table* é utilizada pelo construtor da classe *fsa* antes de armazenar o valor do atributo *symDict*, que é um dicionário para a representação dos nomes dos estados e eventos em LaTeX. Para construir o dicionário, a função *create_table* deve receber as listas de estados e de eventos de um autômato e a tabela dos nomes em LaTeX, caso ela tenha sido passada para a classe *fsa* durante a construção do autômato. O exemplo a seguir mostra como é a estrutura do dicionário retornado.

Exemplo 13

Seja G o autômato definido no DESLab pelo código abaixo.

```
syms('x1 x2 x3 x4 a b')
table = [(x1,'x_1'),(x2,'x_2'),(x3,'x_3'),(x4,'x_4'),(a,r'\
    alpha'),(b,r'\beta')]
X = [x1,x2,x3, x4]
```

```
Sigma = [a,b]
X0 = [x1]
Xm = [x2,x3]
T =[(x1,a,x2),(x1,b,x4),(x2,b,x3),(x3,a,x2),(x4,b,x4),(x4,a,x4
    )]
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')
```

Para G, a função create_table retornará o seguinte dicionário:

```
>>> create_table(G.X,G.Sigma,table)
{'x1': 'x_1', 'x2': 'x_2', 'x3': 'x_3', 'x4': 'x_4', 'a': '\\
    alpha', 'b': '\\beta'}
>>> G.symDict
{'x1': 'x_1', 'x2': 'x_2', 'x3': 'x_3', 'x4': 'x_4', 'a': '\\
    alpha', 'b': '\\beta'}
```

2.2.2 Módulo *algorithms*

No módulo *algorithms* estão as funções que realizam operações com autômatos. As funções contidas nesse módulo permitem, por exemplo: operações entre linguagens, combinações e composições de dois ou mais autômatos. Dessa forma, podemos construir modelos de autômatos a partir de múltiplos autômatos individuais. Nas subseções a seguir serão apresentadas todas as classes e funções desse módulo.

Classe *compCount*

A classe *compCount* é utilizada por muitas funções do módulo *algorithms* quando se deseja criar um mero elemento, como, por exemplo, um novo estado de um autômato, para garantir que não haverá conflito entre os nomes dos novos elementos criados. Essa classe foi construída para armazenar apenas um contador e, toda vez que um objeto da classe é chamado, esse contador é incrementado para todos os objetos da classe, como pode ser visto no exemplo 14.

Exemplo 14

Inicialmente o atributo counter da classe compCount tem valor igual a -1. Dessa forma, na primeira vez que um objeto da classe foi criado, ele se tornará igual a 0, como ilustrado a seguir.

```
>>> from deslab import *
>>> c=compCount()
>>> c.counter
0
>>> c1=compCount()
>>> c1.counter
1
>>> c2=compCount()
>>> c2.counter
```

```
2
>>> c.counter
2
>>> c1.counter
2
```

Função *dfs*

A função *dfs* recebe dois argumentos, em que o primeiro argumento pode ser um autômato, ou seu atributo *Graph*, e o segundo argumento pode ser um único estado ou uma lista de estados do autômato. Uma busca em profundidade é realizada, usando os estados da lista como pontos de partida. A cada estado visitado, é gerado um iterador para as transições desse estado utilizando o grafo do autômato. Ao final, um conjunto contendo todos os estados alcançados é retornado.

Exemplo 15
Seja G o autômato cujo diagrama de transição de estados está representado na figura 2.1. O código abaixo ilustra as formas como a função dfs pode ser usada com G.

```
>>> dfs(G,[0])
frozenset({0, 1, 2, 3})
>>> dfs(G,[2])
frozenset({2})
>>> dfs(G.Graph,[1])
frozenset({1, 2})
>>> dfs(G.Graph,[1,3])
frozenset({1, 2, 3})
```

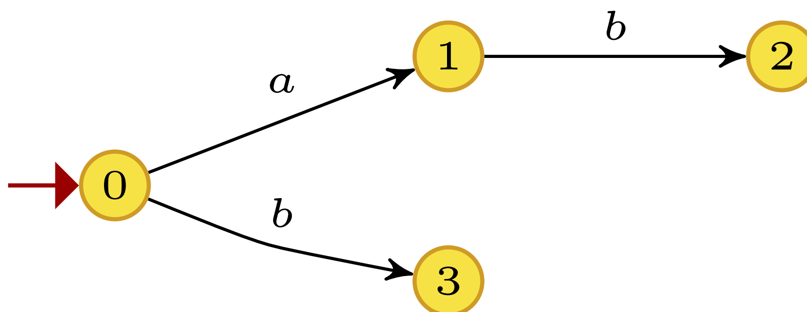


Figura 2.1: Autômato G, do exemplo 15.

Função *ac*

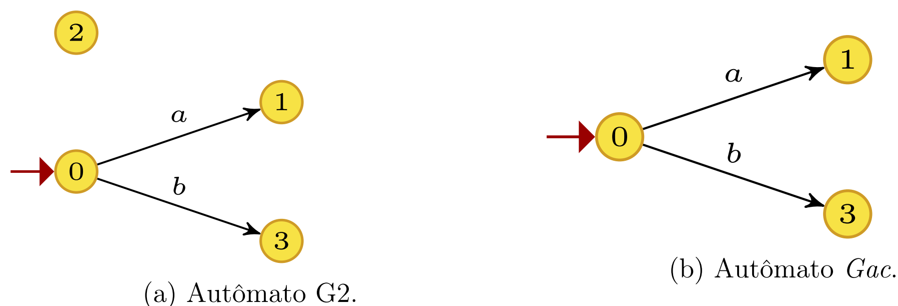
A função *ac* recebe um autômato como argumento e retorna um autômato que corresponde a parte acessível do autômato original. Para encontrar a parte acessível do

autômato, utiliza-se a função *dfs*. A lista de estados marcados do novo autômato é construída comparando-se quais estados marcados do autômato original pertencem a lista de estados acessíveis. No atributo *Graph* do autômato, os *nodes* cujos estados não sejam acessíveis são removidos e, por fim, os dicionários das transições, *gammaDict, deltaDict* e *infoDict*, são criados por meio da função *create_FSA_transdicts*. O exemplo abaixo ilustra o funcionamento da função *ac*.

Exemplo 16

Seja G o autômato, representado na figura 2.1. Removendo-se a sua transição do estado 1 para o estado 2, rotulada pelo evento b, obtém-se o autômato G2, apresentado na figura 2.2a. Note que o estado 2 de G2 não é acessível. Então, aplicando-se a função ac ao autômato, obtém-se o autômato Gac ilustrado na figura 2.2b. Esse procedimento pode ser executado aplicando-se os comandos apresentados no código abaixo.

```
>>> G2 = G.deletetransition((1,b,2))
>>> draw(G2,'figurecolor')
generating latex code of automaton
>>> Gac=ac(G)
>>> draw(Gac,'figurecolor')
generating latex code of automaton
>>> Gac=ac(G2)
```



(a) Autômato G2.

(b) Autômato *Gac*.

Figura 2.2: Autômatos do exemplo 16.

Função *coac*

A função *coac* recebe um autômato como argumento e retorna um autômato que corresponde a parte co-acessível do autômato original. A parte co-acessível de um autômato e formada pelos estados a partir dos quais é possível alcançar um estado marcado.

Para encontrar a parte co-acessível do autômato, utiliza-se o método *reverse* da classe *MultiDiGraph* do NetworkX. Como o atributo *Graph* do autômato é um grafo orientado, objeto da classe *MultiDiGraph*, o método *reverse* inverte o sentido das

transições. Dessa forma, usando-se a função *dfs*, podemos encontrar a lista dos estados da parte co-acessível do autômato partindo dos estados marcados. Uma vez que se gerou o conjunto de estados co-acessíveis, remove-se os demais estados do autômato e os dicionários são criados a partir da função *create_FSA_transdicts*.

Exemplo 17

Seja G o autômato apresentado na figura 2.1. Se marcarmos o estado 2 e calcularmos a sua parte acessível, como descrito no código abaixo, obteremos o autômato apresentado na figura 2.3.

```
>>> G = G.setpar(Xm=[2])
>>> G2 = coac(G)
>>> draw(G2,'figurecolor')
generating latex code of automaton
```

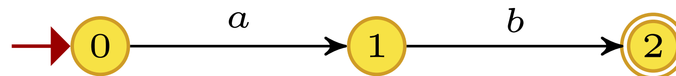
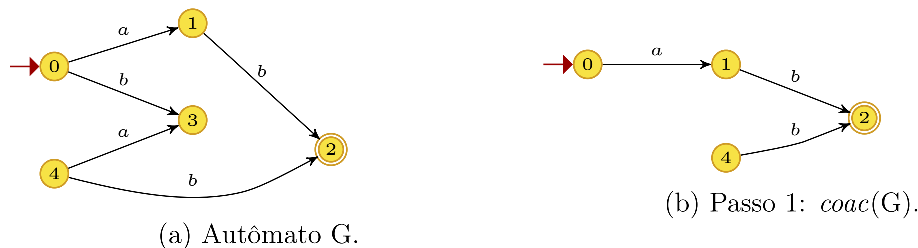


Figura 2.3: Autômato G, do exemplo 17.

Função *trim*

A função *trim*, por definição, representa a operação que retorna a parte acessível e co-acessível de um autômato. Portanto, o seu cálculo é feito utilizando a função *coac*, para calcular a parte co-acessível do autômato passado, e, em seguida, a partir do autômato resultante, calcula-se sua parte acessível utilizando a função *ac*. A figura 2.4 ilustra como a operação *trim* calcula seu resultado, quando aplicada ao autômato G, representado na figura 2.4a



(a) Autômato G.

(b) Passo 1: *coac*(G).

(c) Passo 2: *trim*(G) = *ac(coac(G))*.

Figura 2.4: Comparação entre as funções *coac* e *trim*.

Função *observer* e Função *proj*

A função *observer* recebe um autômato e uma lista de eventos como argumentos. Essa lista de eventos é passada para definir o conjunto de eventos observáveis, *Sigma_obs*, considerado no cálculo do autômato observador do autômato passado como argumento. Caso essa lista de eventos não seja passada, o atributo *Sigobs*[1] do autômato será utilizado.

O autômato observador é um autômato determinístico cujas linguagens gerada e marcada são, respectivamente, as projeções das linguagens gerada e marcada pelo autômato original em relação ao conjunto de eventos observáveis [8].

Duas funções internas são definidas para auxiliar o cálculo da função *observer*:

- *Gamma_obs*: essa função determina o conjunto de eventos ativos em um conjunto de estados, usando o método *Gamma* da classe *fsa*, e retorna a interseção desse conjunto com os eventos observáveis;

- *latexname*: essa função é utilizada para criar os nomes do LaTeX para os estados do observador.

Numa primeira verificação, caso *Sigma_obs* seja igual ao atributo *Sigma* do autômato e o autômato seja determinístico, o autômato original é retornado pela função *observer*. Por sua vez, se o autômato possuir transições rotuladas por ϵ e $Sigma_obs \neq Sigma - \{\epsilon\}$, então, uma mensagem de erro é retornada para autômatos que satisfaçam essa condição, deve-se utilizar a função *epsilonobserver*. Durante o processamento da função, os seguintes passos são executados para gerar o observador:

- Cria-se o estado inicial $X0_obs$ usando o método *unobsreach* do autômato;

- Cria-se a lista de estados X_obs e inclui-se o estado inicial $X0_obs$ em X_obs;

- A lista S é criada com o estado inicial, ela conterá estados do observador que foram definidos, mas ainda não tiveram seus alcances calculados;

- O dicionário *table_obs* é criado com uma chave igual ao estado inicial. Ele irá armazenar os nomes LaTeX para cada estado do observador;

- A lista de estados marcados Xm_obs é criada e, caso o estado inicial, seja marcado, ele é adicionado a Xm_obs.

Inicia-se, então, o processo para definir os demais estados e transições do autômato observador:

[1] Note que, durante a definição do autômato, caso nenhum valor seja passado para *Sigobs*, o mesmo será igual a *Sigma*.

1. Remove-se um estado de S;

2. Usando a função interna *Gamma_ obs*, obtém-se os eventos observáveis ativos para o estado obtido de S no passo anterior;

3. Para cada evento observável ativo, o próximo estado alcançado é determinado usando o método *deltaobs* da classe *fsa* e a nova transição é adicionada à lista *transition*;

4. Se o estado alcançado no passo anterior não pertencer à X_obs:

   - Ele é adicionado à lista S;
   - Se ele possuir algum estado marcado, ele é adicionado à lista Xm_obs;
   - O dicionário *table_ obs* é atualizado com o nome LaTeX desse novo estado.

O processo termina quando S se torna vazia. O autômato observador é criado, usando o construtor da classe *fsa*, e, no seu atributo *graphic*, o estilo é alterado para *observer*.

A função *proj*, usada para calcular um autômato que modela a projeção de uma dada linguagem, recebe um autômato e uma lista de eventos como argumentos. Ela chama a função *observer* e retorna o autômato resultante com o estilo do seu atributo *graphic* modificado para *normal*.

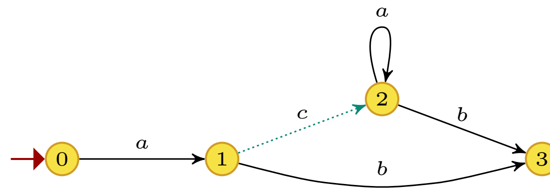
Exemplo 18 *Seja o autômato G, definido pelo código abaixo no DESLab.*

```
syms('a b c')
X     = [0,1,2,3]
Sig   = [a,b, c]
Trans = [(0,a,1),(1,c,2),(1,b,3),(2,a,2),(2,b,3)]
X0    = [0]
Xm    = []
G=fsa(X,Sig,Trans,X0,Xm,name='$G$',Sigobs=[a,b])
```

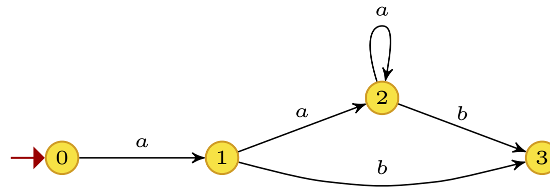
Seu observador e a projeção de G podem ser obtidos como no código abaixo.

```
>>> O = observer(G,[a,b])
>>> P = proj(G,[a,b])
>>> draw(G,'figurecolor')
generating latex code of automaton
>>> draw(O,'figurecolor')
generating latex code of automaton
>>> draw(P,'figurecolor')
generating latex code of automaton
```

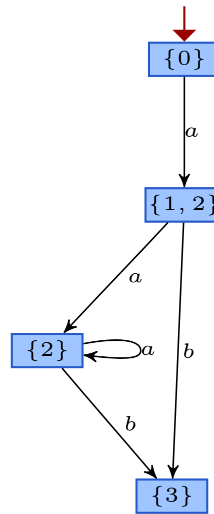
Os diagramas de transição de estados dos autômatos gerados no exemplo podem ser vistos na figura 2.5



(a) Autômato G.



(b) Projeção de G em relação ao conjunto de eventos observáveis $\{a, b\}$.



(c) Autômato do observador O em relação ao conjunto de eventos observáveis $\{a, b\}$.

Figura 2.5: Autômatos gerados no exemplo 18.

Função *epsilonobserver*

A função *epsilonobserver* foi criada para calcular um autômato determinístico equivalente a um dado autômato não determinístico, podendo esse último ter, ou não, transições rotuladas por ϵ. Essa função simplesmente calcula o observador do autômato passado como argumento, usando a função *observer*, assumindo $Sigma - \{\epsilon\}$ como o conjunto de eventos observáveis, em que $Sigma$ é o atributo homônimo do autômato passado como argumento.

Função *invproj*

A função *invproj* recebe dois argumentos: um autômato e uma lista de eventos, a qual deve conter todos os eventos do autômato. Essa função retorna um autô-

mato cujas linguagens gerada e marcada são as projeções inversas das respectivas linguagens do autômato original no conjunto de eventos passado como argumento. Para gerar esse autômato, cria-se um auto-laço em cada estado, de uma cópia do autômato original, com os novos eventos passados que não pertencem ao conjunto de eventos do autômato original. O conjunto de eventos do autômato retornado pela função invproj é formado pelos eventos contidos na lista passada como argumento. Por sua vez, o seu conjunto de eventos observáveis é definido igual ao atributo Sigma do autômato original. O exemplo abaixo mostra como obter a projeção inversa de um autômato.

Exemplo 19

Considere o autômato G definido no código abaixo.

```
syms('a b')
X      = [0,1,2,3]
Sig    = [a,b]
Trans  = [(0,a,1),(1,b,2),(0,b,3)]
X0     = [0]
Xm     = [2]
G=fsa(X,Sig,Trans,X0,Xm)
```

A sua projeção inversa, gerada pelo código abaixo.

```
>>> Ginvp = invproj(G,[c])
>>> Ginvp2 = invproj(G,[a,b,c])
>>> Ginvp == Ginvp2
True
>>> draw(Ginvp, 'figurecolor')
generating latex code of automaton
```

O diagrama de transição de estados de Ginvp é apresentado na figura 2.6

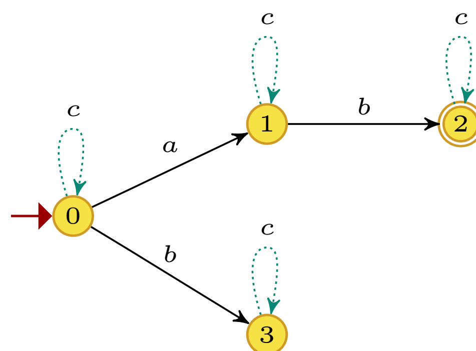
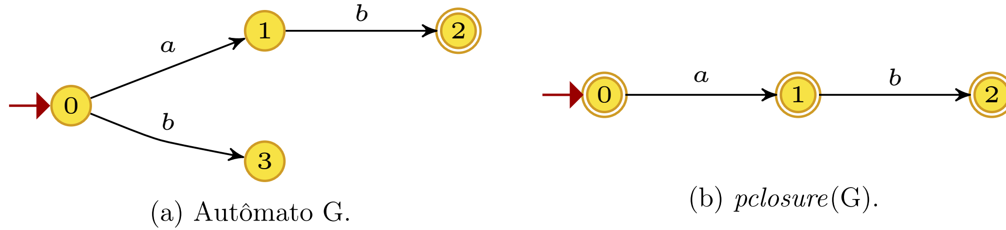


Figura 2.6: Autômato *invproj(G)*, no qual G é o autômato do exemplo 19.

Função *pclosure*

A função *pclosure* recebe um autômato como argumento, e retorna um autômato que marca o prefixo da linguagem marcada pelo autômato recebido. Pra gerar esse

autômato, utiliza-se a função *trim* e, no autômato resultante, todos os estados são marcados, como pode ser visto na figura 2.7.



(a) Autômato G.

(b) *pclosure*(G).

Figura 2.7: Exemplo da função *pclosure*.

Função *langquotient*

A função *langquotient* recebe dois autômatos como argumentos, *H1* e *H2*, e retorna um autômato cuja linguagem marcada é o quociente das linguagens marcadas pelos dois autômatos, ou seja, $L_m\{H1\}/L_m\{H2\}$. O quociente de duas linguagens L_1 e L_2 é definido por:

$$L_1/L_2 := \{s \in E^* : (\exists t \in L_2)[st \in L_1]\}$$

Antes de calcular o quociente, os autômatos passados são verificados. Caso algum deles seja um autômato vazio, um autômato vazio será retornado. Para o cálculo do quocient [9] os seguintes passos são seguidos:

1. Cria-se uma lista de estados marcados *Xm*, inicialmente vazia;

2. Para cada estado x de *H1*, cria-se uma cópia *H1x* de *H1* em que o estado inicial é modificado para o estado x;

3. Para cada autômato *H1x*, o seu produto com *H2* é calculado usando o operador & e, caso o conjunto de estados marcados do autômato resultante não seja vazio, o estado x é adicionado à lista *Xm*;

4. Cria-se o autômato M, que é uma cópia de *H1*, e redefine-se o seu conjunto de estados marcados usando a lista *Xm*, calculada no passo anterior;

5. Por fim, o autômato resultante da operação trim(M) é retornado.

Quando o autômato *H2* tem apenas um estado e a sua linguagem marcada não é vazia, pode-se concluir que a linguagem marcada por *H2* corresponde ao fecho de Kleene do conjunto formado pelos eventos ativos do seu único estado, denotado aqui por *SetOfKleenClosure*. Nesse caso, a função *langquotient* calcula o quociente das linguagens da seguinte forma:

1. Cria-se uma cópia *H1x* de *H1*;

2. Pra cada estado de *H1x*, remove-se as transições rotuladas por eventos não pertencentes a *SetOfKleenClosure*;

3. Todos os estados de *H1x* são configurados como estados iniciais. Isso é feito para evitar que a função *coac*, usada no passo a seguir, retorne um autômato vazio, no caso em que o estado inicial é removido durante o seu cálculo;

4. Calcula-se co-acessível de *H1x* e os seus estados são armazenados na lista *Xm*;

5. Cria-se o autômato *M*, que é uma cópia de *H1*, e redefine-se o seu conjunto de estados marcados usando a lista *Xm*, calculada no passo anterior;

6. Por fim, o autômato resultante da operação trim(M) é retornado.

O operador / foi sobrecarregado, na classe *fsa*, para executar a função *langquotient*. No exemplo 20 vemos as diferentes formas de calcular o quociente de duas linguagens.

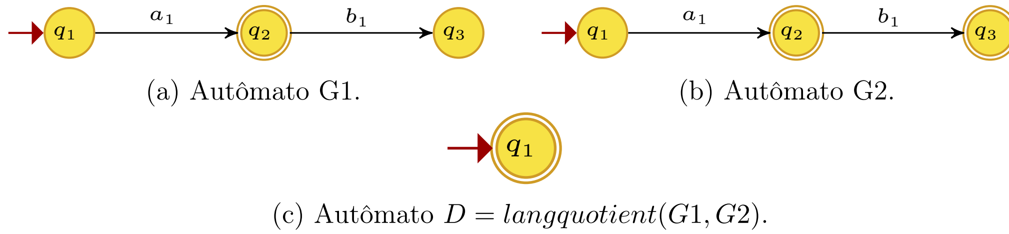
Exemplo 20

Sejam G1 e G2 os autômatos definidos no código abaixo.

```
syms('q1 q2 q3 a1 b1 e f')
tab=[(q1,'q_1'),(q2,'q_2'),(q3,'q_3'),(a1,'a_1'),(b1,'b_1')]
X = [q1,q2,q3]
Sigma = [a1,b1,e]
X0 = [q1]
Xm = [q2]
Xm2 = [q2,q3]
Sigma2= [a1,b1, f]
T =[(q1,a1,q2),(q2,b1,q3)]
G1 = fsa(X,Sigma,T,X0,Xm,tab,name='$G_1$')
G2 = fsa(X,Sigma,T,X0,Xm2,stab,name='$G_2$')
```

Calculando o quociente das linguagens de G1 e G2, como no código abaixo, teremos o autômato apresentado na figura 2.8.

```
>>> Q = langquotient(G1,G2) #Lm{1}/Lm{2}#
>>> Q==G1/G2
True
>>> draw(G1,'figurecolor')
generating latex code of automaton
>>> draw(G2,'figurecolor')
generating latex code of automaton
>>> draw(Q,'figurecolor')
generating latex code of automaton
```



(a) Autômato G1.

(b) Autômato G2.

(c) Autômato $D = langquotient(G1, G2)$.

Figura 2.8: Exemplo da função *langquotient*.

Função *complement*

A função *complement* calcula o complemento da linguagem marcada por um autômato, que é passado como argumento.

Inicialmente, uma cópia do autômato, *auto*, é criada para ser manipulada e retornada ao final. Um estado *_XD_i* é criado, no qual *i* é um número inteiro gerado pelo contador da classe *compCount*. O nome em LaTeX desse estado é definido como *X_D*.

O estado *XD_i* é adicionado ao grafo do autômato, no atributo Graph, e à sua lista de estados. Então, cria-se em *XD_i*, um auto-laço com todos os eventos definidos no atributo *Sigma* do autômato. Em seguida, percorre-se os demais estados do autômato para verificar, usando o dicionário *gammaDict*, quais eventos não estão ativos em cada estado e criar transições para o estado *XD_i* rotuladas por esses eventos. Atualiza-se os dicionários, *gammaDict, deltaDict* e *infoDict*, usando a função *create_FSA_transdicts*, e o dicionário *symDict*, que armazena os nomes em LaTeX. Por fim, inverte-se a marcação dos estados do autômato obtido nos passos anteriores e retorna-se a parte *trim* desse novo autômato. Na classe *fsa*, o operador ~ foi sobrecarregado para executar a função *complement*.

A figura 2.9 mostra o complemento gerado para o autômato *G1* da figura 2.8.

Exemplo 21 *Seja o autômato G1 apresentado na figura 2.8a. O autômato obtido ao se calcular o complemento de G1 é apresentado na figura 2.9.*

```
>>> C = complement(G1)
>>> C == ~G1
True
```

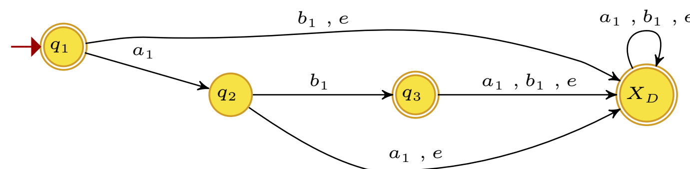


Figura 2.9: *complement(G1)*.

Função *langdiff*

A função *langdiff* recebe dois autômatos *G1* e *G2* como argumento e calcula um autômato D tal que $L_m(D) = L_M(G1) \backslash L_m(G2)$.

No cálculo da função *langdiff*, verifica-se inicialmente, se algum dos autômatos é vazio. Se *G1* for vazio, então um autômato vazio será retornado. Se *G2* for vazio, então o autômato *G1* será retornado. Quando nenhum dos autômatos é vazio, calcula-se o produto de *G1* por *complement(G2)*. O resultado é passado como argumento para a função *trim* e o autômato obtido é retornado. Na classe *fsa* o operador $-$ foi sobrecarregado para executar a função *langdiff*.

Exemplo 22

Considere os autômatos G1 e G2 apresentados na figura 2.8. O autômato obtido ao se executar a função langdiff como no código abaixo é apresentado na figura 2.10.

```
>>> D=langdiff(G1,G2)
>>> D==G1-G2
True
>>>
```

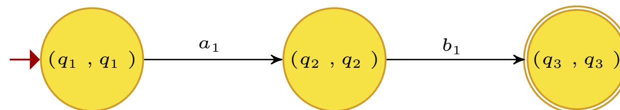


Figura 2.10: $L_m\{G1\} \backslash L_m\{G2\}$.

Função *complete*

A função *complete* cria um autômato completo a partir de um autômato passado como argumento, isto é, um autômato cuja linguagem gerada é Σ^* e a linguagem marcada é igual àquela marcada pelo autômato original.

As operações realizadas são semelhantes às da função *complement*, um novo X_D é criado e transições de cada estado do autômato são completadas, criando-se novas transições, rotuladas pelos eventos não ativos, para o estado X_D. Também se cria um auto-laço em X_D rotulado por todos os eventos. Diferente da função *complement*, nesse caso, os estados marcados permanecem os mesmos.

Exemplo 23

Seja o autômato G1 apresentado na figura 2.8a. O autômato obtido com complete(G1) é apresentado na figura 2.11.

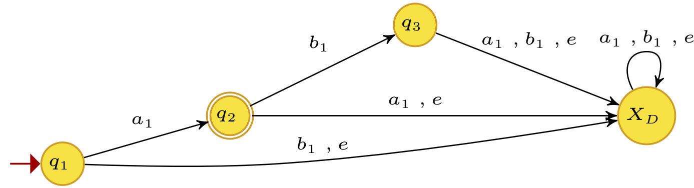


Figura 2.11: *complete(G1).*

Função *sigmakleeneclos*

A função *sigmakleeneclos* cria um autômato de um estado. Esse estado é marcado e possui um auto-laço rotulado pelos eventos contidos na lista de eventos passada como argumento. Caso o usuário deseje, *strings* com o nome e o nome em LaTeX do estado podem ser passadas como argumento. O autômato criado irá gerar e marcar o fecho de Kleene do conjunto de eventos passado como argumento.

Exemplo 24
Seja $\Sigma = \{a, b, c\}$. *Podemos gerar um autômato cujas linguagens gerada e marcada sejam* Σ^* *usando o código abaixo.*

```
>>> G = sigmakleeneclos([a,b,c])
>>> G.tmx('table')

Transition Matrix:

     a    c    b
s0   s0   s0   s0

G = sigmakleeneclos([a,b,c],'x','s12')
>>> G.tmx('table')

Transition Matrix:

     a    c    b
x    x    x    x

>>> G.Graph.nodes(data=True)
NodeDataView({'x': {'label': 's0'}})
>>> G.symDict
{'x': 's12'}
```

Função *union*

A função *union* calcula um autômato que marca a união das linguagens marcadas por dois autômatos *G1* e *G2* passados como argumento.

O primeiro passo é a verificação dos autômatos passados. Usando a função *isitempty*, verifica-se a linguagem marcada dos autômatos passados. Caso alguma delas não seja vazia, o outro autômato é retornado. Se as duas forem vazias, um

autômato vazio é retornado. Em seguida, faz-se o produto dos complementos dos autômatos, usando o operador $\sim$. O resultado é passado para a função *trim* e o complemento do autômato resultante será o autômato cuja linguagem marcada é a união das linguagens marcadas de *G1* e *G2*. O exemplo abaixo mostra como utilizar a função *union*, pois na classe *fsa* o operador + foi sobrecarregado para essa função.

Exemplo 25

Sejam G1 e G2 os autômatos definidos pelo código abaixo.

```
syms('v w x y z  a b')
X = [v,w,x,y,z]
Table=[]
Sigma = [a,b]
X0 = [v]
Xm = [z]
T =[(v,a,w),(w,a,x),(w,b,y),(y,b,z)]
G1 = fsa(X,Sigma,T,X0,Xm,Table,name='$G1$')

X      = [0,1,2,3]
Sig    = [a,b]
Trans  = [(0,a,1),(1,b,2),(0,b,3)]
X0     = [0]
Xm     = [2]
G2=fsa(X,Sig,Trans,X0,Xm,name='$G2$')
```

A união dos autômatos pode ser feita das seguintes formas.

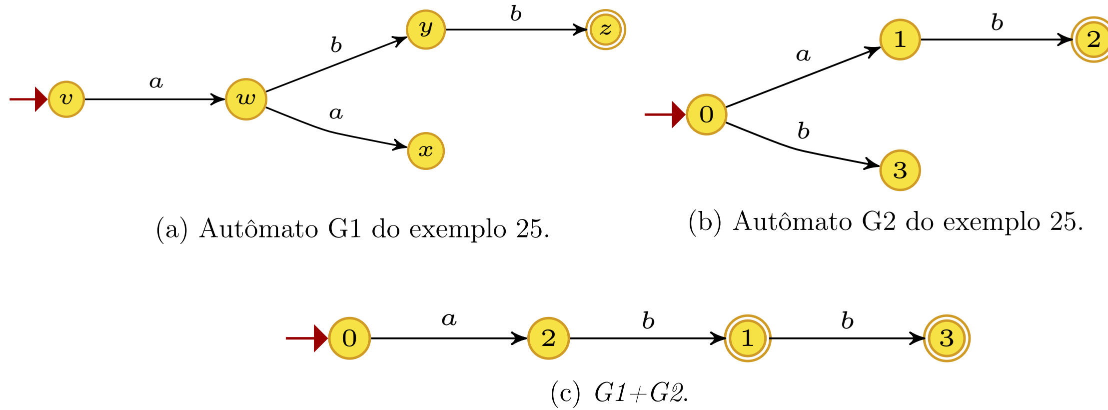
```
>>> G = G1+G2
>>> Gi = union (G1,G2)
>>> Gi==G
True
>>> draw(G,'figruecolor')
generating latex code of automaton
>>> draw(G,'figurecolor')
generating latex code of automaton
```

Note que, como pode ser visto na figura 2.12, antes de retornar o autômato, seus estados são renomeados para números.

Função *concatenation*

A função *concatenation* recebe dois autômatos, *self* e *other*, como argumentos e retorna um autômato cuja linguagem marcada é igual a concatenação das linguagens marcadas pelos autômatos *self* e *other*, ou seja, $L_m(self)L_m(other)$. Um terceiro argumento pode ser passado para definir o tipo do autômato retornado, *True* para determinístico (valor padrão) e *False* para não determinístico.

Para calcular a concatenação, a função *concatenation* cria os argumentos necessários para criar o autômato resultante usando o construtor da classe *fsa*. Para



(a) Autômato G1 do exemplo 25.

(b) Autômato G2 do exemplo 25.

(c) *G1+G2*.

Figura 2.12: Exemplo da função *union*.

isso, cria-se listas que agrupam os conjuntos de estados, os conjuntos de eventos, as transições e as listas de nomes em LaTeX dos autômatos *self* e *other*. O conjunto de estados iniciais é definido igual ao do autômato *self*, e o conjunto de estados marcados é definido igual ao do autômato *other*. É importante ressaltar que, com o objetivo de evitar conflitos causados por estados com nomes iguais entre os autômatos, usa-se a função interna *rename* para adicionar um índice que associa cada estado ao seu autômato original. Antes de construir o autômato, adiciona-se transições rotuladas por ϵ que levam de todos os estados marcados de *self* para todos os estados inicias de *other*. O autômato resultante é, então, construído usando *fsa* e seus estados são renomeados para números. Caso o terceiro argumento seja *False*, esse autômato é retornado. Caso ele seja *True*, calcula-se o autômato determinístico equivalente usando a função *epsilonobserver* e seus estados são renomeados para números.

Vale ressaltar que os conjuntos de eventos *Sigobs* e *Sigcon* do autômato retornado pela função *concatenation* são definidos como a união dos respectivos conjuntos de *self* e *other*.

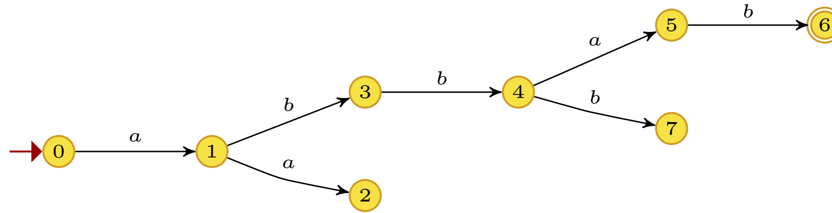
O exemplo abaixo mostra como utilizar a função *concatenation*, uma vez que na classe *fsa* o operador * foi sobrecarregado para executar a função *concatenations*, entretanto, usando do operador, o autômato retornado será sempre determinístico.

Exemplo 26

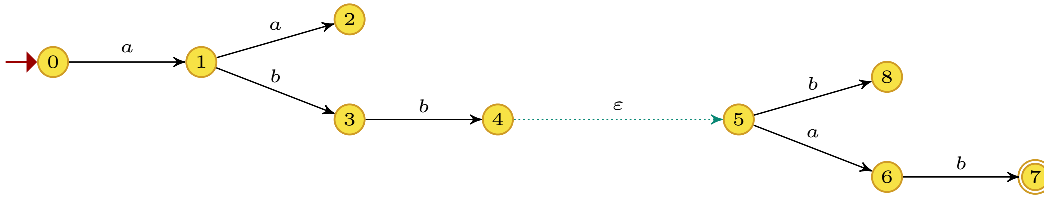
Sejam G1 e G2 os autômatos definidos no exemplo 25 os autômatos, determinístico e não determinístico, gerados com a função concatenation, no código abaixo, são apresentados na figura 2.13.

```
>>> Cdet = G1*G2
>>> Cndet = concatenation(G1,G2,False)
>>> draw(Cdet,'figurecolor')
generating latex code of automaton
>>> draw(Cndet,'figurecolor')
```

```
generating latex code of automaton
```



(a) Autômato *Cdet*.



(b) Autômato *Cndet*.

Figura 2.13: Autômatos gerados pela função *concatenation*.

Função *paralleldet*

A função *paralleldet* calcula a composição paralela de dois autômatos determinísticos. Para isso ela recebe dois autômatos como argumentos, *self* e *other*, além de um argumento que especifica se os nomes dos estados do autômato resultante devem ser simplificados ou não, recebendo o valor *True* ou *False*. Dentro da função paralleldet, 4 funções internas são definidas:

- *pars*: utilizada quando se deseja simplificar os nomes dos estados do autômato. A função recebe um estado, e o simplifica tornando-o uma única tupla.

- *Gamma_p*: recebe uma tupla formada por um estado de *self* e um estado de *other* e retorna os eventos particulares ativos em cada um desses estados e os eventos comuns ativos em ambos.

- *delta_p*: recebe uma tupla formada por um estado de *self* e um estado de *other* e um evento ativo nesse par de estados, e retorna o estado alcançado pela transição a partir desse par de estados e rotulada por esse evento.

- *latexname*: constrói o nome LaTeX do par de estados recebido, para ser incluído na tabela de nomes do novo autômato.

A construção do autômato resultante da composição paralela, entre *self* e *other*, obedece aos seguintes passos:

- A partir das listas de eventos dos autômatos, *Sigma, Sigobs* e *Sigcon* , são criadas as listas do autômato final. Vale ressaltar que os conjuntos de eventos *Sigobs* e *Sigcon* são definidos como a união dos respectivos conjuntos de *self* e *other*.
- O estado inicial é criado a partir do par de estados iniciais, e a lista de estados *X_p* é criada e, inicialmente, possui apenas o estado inicial;
- A lista S é criada com o estado inicial. Essa lista será usada para obter a informação dos estados a serem percorridos;
- A lista de transições *transition* é criada, inicialmente vazia;
- A tabela *table_p*, que armazena os nomes LaTeX é criada e o estado inicial é adicionado;
- Verifica-se os estados iniciais e, caso sejam marcados, o estado inicial formado por eles é adicionado à lista de estados marcados *Xm_p*.

Os passos a seguir são executados até que a lista S se torne vazia:

1. Remove-se um estado de S e verifica-se seus eventos ativos pela função interna *Gamma_p*;
2. Para cada evento ativo obtém-se, usando a função interna *delta_p*, o próximo estado alcançado;
3. Se um estado alcançado q não estiver na lista *X_p*, então:
   - O estado q é adicionado às listas *X_p* e S;
   - Se ambos os estados, de *self* e *other*, que formam q forem marcados, o estado q é adicionado à lista de estados marcados *Xm_p*;
   - A tabela *table_p* e a lista de transições *transition* são atualizadas para o estado q;

Por padrão, a função *paralleldet* retorna um autômato cujos nomes dos estados foram simplificados usando a função interna *pars*, devido ao fato de que, composições paralelas entre muitos autômatos podem gerar estados com nomes muito complexos. Essa simplificação pode ser desabilitada por meio do parâmetro *simplify*, fazendo-o receber o valor *False*. O exemplo a seguir ilustra as diferenças obtidas de acordo com o valor passado para o parâmetro *simplify*.

Exemplo 27

Considere os autômatos G1 e G2, definidos no exemplo 25, e o autômato G3 definido no código abaixo.

```
syms('x1 x2 x3 x4 a b')
table = [(x1,'x_1'),(x2,'x_2'),(x3,'x_3'),(x4,'x_4')]
X = [x1,x2,x3,x4]
Sigma = [a,b]
X0 = [x1]
Xm = [x2,x3]
T =[(x1,a,x2),(x1,b,x4),(x2,b,x3),(x3,a,x2),(x4,b,x4),(x4,a,x4
   )]
G3 = fsa(X,Sigma,T,X0,Xm,table,name='$G3$')
```

No código abaixo, calcula-se o autômato $Gp = G1\|G2$ *e, em seguida, os autômatos* $Gp3$ *e* $Gps3$*, ambos iguais a* $Gp\|G3$*, definindo o parâmetro simplify como False e True, respectivamente. Os digramas de transição de estados gerados podem ser vistos na figura 2.14.*

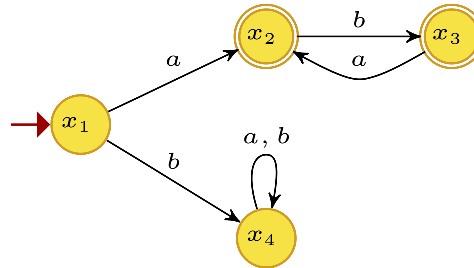
```
>>> Gp=paralleldet(G1,G2,False)
>>> draw(Gp,'figurecolor')
generating latex code of automaton
>>> Gp3 = paralleldet(Gp,G3,False)
>>> Gps3 = paralleldet(Gp,G3,True)
>>> draw(Gp3,'figurecolor')
generating latex code of automaton
>>> draw(Gps3,'figurecolor')
generating latex code of automaton
```

Função *parallelnondet*

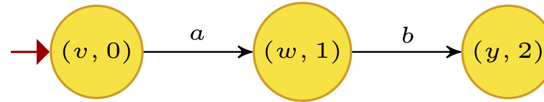
A função *parallelnondet* realiza a composição paralela entre dois autômatos, quando algum deles é não determinístico. Todas as operações realizadas, inclusive as funções internas, são semelhantes às apresentadas na função *paralleldet*. No entanto, alguns ajustes são feitos, dentre os mais relevantes: *(i)* para calcular o estado alcançado por uma transição, usa-se o método especial *delta*, ao invés do método *delta* usado no caso determinístico, e *(ii)* o conjunto de estados iniciais é definido como o produto cartesiano dos conjuntos de estados iniciais dos autômatos de entrada.

Função *parallel*

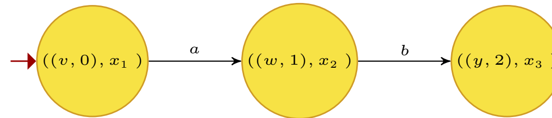
A função *parallel* é a função que deve ser chamada quando se deseja realizar a composição paralela entre dois autômatos. Ela recebe dois autômatos e um argumento referente a simplificação dos estados, que deve ser *True* ou *False*. Essa função verifica se algum dos autômatos passados é não determinístico ou vazio, para decidir o que retornar:



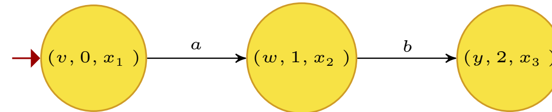
(a) Autômato *G3*.



(b) Autômato *Gp*.



(c) Autômato *Gp3*.



(d) Autômato *Gps3*.

Figura 2.14: Autômatos gerados pela função *paralleldet*.

- Caso algum autômato seja vazio, um autômato vazio é retornado
- Caso algum autômato seja não determinístico, a função *parallelnondet* é chamada para calcular o autômato resultante.
- Caso contrário, a função *paralleldet* é chamada.

Na classe *fsa*, o operador // foi sobrecarregado para a função *parallel*, com o parâmetro *simplify* predefinido como *True*.

Função *productdet*

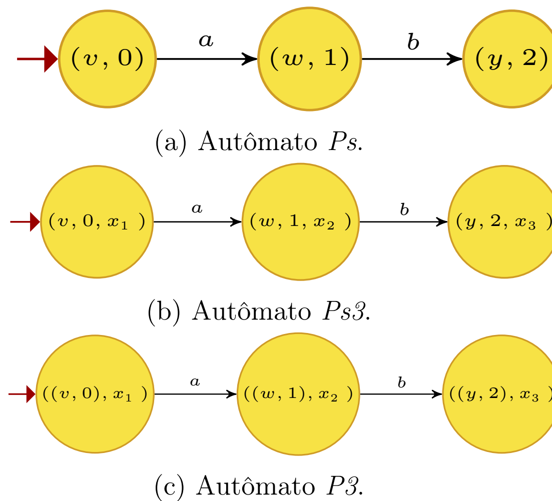
A função *productdet* calcula o produto entre dois autômatos, *self* e *other*, recebidos como argumento, e recebe um terceiro argumento, referente à simplificação dos nomes dos estados do autômato gerado, que deve ser *True* ou *False*. Todo o processo é semelhante ao descrito para a função *paralleldet*, exceto que, a função interna *delta_p* não é utilizada e, no lugar da função interna *Gamma_p*, a função *Gamma_-prod* é definida. Na função interna *Gamma_prod* os eventos ativos retornados são a interseção do conjunto de eventos ativos do par de estados passado. Assim como no caso do paralelo de dois autômatos, a simplificação de estados só produzirá

um autômato diferente quando mais de dois autômatos forem utilizados, como é mostrado no exemplo abaixo.

Exemplo 28

Considere os autômatos G1 e G2, definidos no exemplo 25, e o autômato G3, definido no exemplo 27. No código abaixo, calcula-se o autômato $Ps = G1 \times G2$ *e, em seguida, os autômatos* $Ps3$ *e* $P3$*, ambos iguais a* $Ps \times G3$*, definindo o parâmetro simplify como True e False, respectivamente. Os digramas de transição de estados gerados podem ser vistos na figura 2.15.*

```
>>> Ps = productdet(G1,G2,True)
>>> Ps3 = product(Ps,G3,True)
>>> P3 = product(Ps,G3,False)
>>> draw(Ps,'figurecolor')
generating latex code of automaton
>>> draw(Ps3,'figurecolor')
generating latex code of automaton
>>> draw(P3,'figurecolor')
generating latex code of automaton
```



(a) Autômato *Ps*.

(b) Autômato *Ps3*.

(c) Autômato *P3*.

Figura 2.15: Autômatos gerados pela função *productdet*.

Função *productnondet*

A função *productnondet* realiza as mesmas operações que a função *productdet* e utiliza as mesmas funções internas, com o diferencial de aceitar autômatos não determinísticos como argumentos. Durante o cálculo do produto, em contraste com a função *productdet*, a função *productnondet* usa o método especial *delta*, e o conjunto de estados iniciais é definido como o produto cartesiano dos conjuntos de estados iniciais dos autômatos de entrada.

Função *product*

A função *product* deve ser a função chamada quando se desejar calcular o produto entre dois autômatos. Ela recebe dois autômatos como argumento, além de receber o argumento referente à simplificação dos estados. Quando chamada, a função *product* verifica se algum dos autômatos é não determinístico e, em caso afirmativo, ela executa a função *productnondet*. Caso contrário, ela executa a função *productdet*. Na classe *fsa*, o operador & foi sobrecarredo para a função *product*, com o parâmetro *simplify* predefinido como *True*.

2.2.3 Módulo *comparison*

Como o próprio nome sugere, no módulo *comparison* estão todas as funções que realizam comparações entre autômatos, ou verificam se um autômato possui uma dada propriedade. Nesta seção destacaremos como essas funções funcionam, as variáveis que elas modificam e os valores que retornam.

Função *isitempty*

A função *isitempty* determina se a linguagem marcada pelo autômato é um conjunto vazio. A função retorna *True* quando o autômato não possui estados marcados ou quando os seus estados marcados não são acessíveis. Para isso, ela verifica, inicialmente, se o autômato é vazio, por meio do atributo *empty* e, em seguida, verifica se a parte acessível do autômato possui estados marcados utilizando a função *ac*. O exemplo 29 ilustra como a função pode ser chamada.

Exemplo 29

Seja o autômato G, definido no exemplo 52. Passando G como argumento para a função isitempty:

```
>>> isitempty(G)
True
```

Como G não possui estados marcados esse resultado era esperado. Agora, construa G2, a partir de G, marcando seu estado x_2, teremos o seguinte resultado:

```
>>> G2=G.setpar(Xm=[x2])
>>> isitempty(G2)
False
```

Função *issublanguage*

A função *issublanguage* recebe dois autômatos como argumento e compara se a linguagem marcada do primeiro autômato está contida na linguagem marcada do

segundo, ou seja, a função *issublanguage* determina se a linguagem do primeiro autômato é uma sub-linguagem do segundo.

Uma vez passados os autômatos, caso os conjuntos de eventos deles não sejam iguais, para que a comparação possa ser feita, os eventos de cada um dos autômatos são adicionados ao conjunto de eventos do outro, nesse caso, uma mensagem de alerta aparecerá para notificar essa ação ao usuário.

Para realizar a comparação, computa-se o produto do primeiro autômato com o complemento do segundo e, utilizando a função *isitempty*, verifica-se se a linguagem marcada desse produto é vazia, conforme descrito abaixo.

$$\begin{aligned}
L_m({G_2}^{comp}) &= \Sigma^* \backslash L_m(G_2) \\
L_m(G_1 \times {G_2}^{comp}) &= L_m(G_1) \cap L_m({G_2}^{comp}) \\
L_m(G_1 \times {G_2}^{comp}) = \emptyset &\Leftrightarrow L_m(G_1) \subseteq L_m(G_2)
\end{aligned}$$

O exemplo 30 mostra as diferentes formas em que podemos chamar a função *issublanguage*, uma vez que na classe *fsa* alguns operadores são sobrecarregados.

Exemplo 30

Considere os autômatos definidos pelo código abaixo:

```
syms('x1 x2 a')
table = [(x1,'x_1'),(x2,'x_2'),(a,r'\alpha')]
X = [x1,x2]
Sigma = [a]
X0 = [x1]
Xm = [x2]
T =[(x1,a,x2)]
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')


syms('x1 x2 x3 a b')
table = [(x1,'x_1'),(x2,'x_2'),(a,r'\alpha'),(x3,'x_3'),(b,r'\
    beta')]
X = [x1,x2,x3]
Sigma = [a,b]
X0 = [x1]
Xm = [x2,x3]
T =[(x1,a,x2),(x2,b,x3)]
G2 = fsa(X,Sigma,T,X0,Xm,table,name='$G2$')
```

Comparando os dois autômatos, como as listas de eventos deles são diferentes, receberemos a mensagem de aviso [2] *presente no código abaixo.*

```
>>> issublanguage(G,G2)
```

[2] Pela lógica, deveríamos receber uma mensagem de alerta para todas as chamadas do exemplo, contudo, por padrão do *Python*, mensagens repetidas de alerta de uma mesma fonte são suprimidas.

```
UserWarning: input automata have different alphabets. They
   have been standarsized to compare
True
>>> issublanguage(G2,G)
False
```

A função issublanguage também pode ser chamada utilizando os operadores <= e >=, conforme descrito no código a seguir.

```
>>> G<=G2

UserWarning: input automata have different alphabets. They
   have been standarsized to compare
True
>>> G>=G2
False
```

Note que o operador <= (resp. >=) verifica se $L_m(G_1) \subseteq L_m(G_2)$ *(resp.* $L_m(G_2) \subseteq L_m(G_1)$*)*

Função *are_automataequal*

A função *are_automataequal* verifica a igualdade entre dois autômatos. Essa função compara os estados, eventos, estados iniciais e o alcance dos autômatos e retorna *True* caso todos esses elementos sejam iguais.

O exemplo 31 mostra como utilizar a função e quais operadores foram sobrecarregados na classe *fsa*.

Exemplo 31

Considere os autômatos G e G2, definidos no exemplo 30.

```
>>> are_automataequal(G,G)
True
>>> are_automataequal(G,G2)
False
```

Utilizando os operadores de igualdade, ==, e de diferença, !=:

```
>>> G==G
True
>>> G!=G
False
>>> G==G2
False
>>> G!=G2
True
```

Função *are_langequiv*

A função *are_langequiv* compara se os dois autômatos passados como argumento marcam a mesma linguagem. Da mesma forma que a função *issublanguage*, ela mo-

difica a lista de eventos dos autômatos, quando necessário, para realizar as operações de comparação.

Para realizar a comparação, inicialmente é computado o produto do primeiro autômato pelo complemento do segundo e, em seguida, o produto do complemento do primeiro autômato pelo segundo. Esses dois autômatos resultantes são unidos e verifica-se a existência de estados marcados em sua parte acessível. Caso o conjunto de estados marcados da parte acessível do autômato resultante seja vazio, as linguagens marcadas pelos autômatos comparados são iguais.

Função *isitcomplete*

Essa função *isitcomplete* é usada para verificar se um autômato é completo, ou seja, se $L(G) = \Sigma^*$. Para isso, verifica-se o atributo *gammaDict* do autômato. Esse atributo é um dicionário que contém, para cada estado, a lista de eventos ativos. O autômato será completo se, para cada um dos estados, todos os eventos de Σ forem ativos.

2.2.4 Módulo *def_const*

O módulo *def_const* é utilizado para a definição de algumas variáveis muito utilizadas pelas funções do DESLab, as quais são apresentadas na tabela 2.7.

Tabela 2.7: Variáveis em *def_const*.

Variável	Valor
EMPTY	'empty'
epsilon	'epsilon'
EMPTYSTRINGSET	set([epsilon])
EMPTYSET	frozenset([])
RENAMEVAR	'x'
ALL_EVENTS	'ALL_EVENTS'
UNDEFINED	'UNDEFINED'
set	frozenset

2.2.5 Módulo *exceptions*

No módulo *exceptions* estão definidas todas as classes de erros comuns, de software e de usuários do DESLab. Essas classes são utilizadas pelo *Python* para retornar o tipo de erro que ocorreu durante a execução e, a partir disso, o usuário pode identificar e corrigir o problema no código com mais facilidade. A tabela 2.8 relaciona as classes de erro definidas com suas classes base.

Tabela 2.8: Classes de erro.

Classe	Base
desError	Exception
inputError	Exception
DFAerror	desError
notDFAerror	DFAerror
stateError	desError
deslabError	desError
stateMembershipError	desError
eventMembershipError	desError
markedSetError	desError
initialStateError	desError
epsilonDFAError	DFAerror
inputStringError	desError
invalidAutomaton	desError
invalidArgument	inputError
invalidLabel	inputError
invalidTransition	inputError

2.2.6 Módulo *graphs*

No módulo *graphs* estão as funções que percorrem os estados do autômato em busca de condições específicas.

Função *strconncomps*

A função *strconncomps* importa a função *strongly_ connected_ components* do módulo NetworkX para retornar um objeto iterador do conjunto de componentes fortemente conexos do autômato. Um componente fortemente conexo de um autômato $G = (X, \Sigma, f, \Gamma, X_0, X_m)$ é um conjunto máximo de vértices $U \subseteq X$ tal que, para cada par de estados $u_1, u_2 \in U$, u_2 é alcançável a partir de u_1, e vice-versa [10].

Essa função pode receber como argumento tanto um autômato quanto o atributo *Graph* desse autômato, pois ele é um objeto da classe *NetworkX*. O exemplo 32 apresenta os resultados do uso da função e como observá-los.

Exemplo 32

Considere o autômato G3, cujo diagrama de transição de estados pode ser visto na figura 2.16 definido pelo seguinte código no DESLab:

```
syms('x1 x2 x3 x4 a b')
table = [(x1,'x_1'),(x2,'x_2'),(x3,'x_3'),(x4,'x_4'),(a,r'\
    alpha'),(b,r'\beta')]
X = [x1,x2,x3, x4]
Sigma = [a,b]
X0 = [x1]
Xm = [x2,x3]
T =[(x1,a,x2),(x1,b,x4),(x2,b,x3),(x3,a,x2),(x4,b,x4),(x4,a,x4
    )]
G3 = fsa(X,Sigma,T,X0,Xm,table,name='$G3$')
```

Chamando a função strconncomps, por meio de um loop for, podemos observar todos os valores que ela retorna.

```
>>> stcomp = strconncomps(G3)

>>> for comp in stcomp:
        print(comp)


{'x4'}
{'x3', 'x2'}
{'x1'}
```

Outras formas de se obter os valores da função são descritas no código a seguir.

```
>>> stcomp = strconncomps(G3)

>>> next(stcomp)
{'x4'}
>>> next(stcomp)
{'x3', 'x2'}
>>> next(stcomp)
{'x1'}
>>> list(strconncomps(G3))

[{'x4'}, {'x3', 'x2'}, {'x1'}]
```

Podemos perceber que, da forma como os estados são retornados, não conseguiremos diferenciar um componente fortemente conexo formado por somente um estado com auto-laço de um componente fortemente conexo trivial, isto é, um componente formado por um só estado sem auto-laço. Quando isso for necessário, podemos utilizar função *selfloopnodes*.

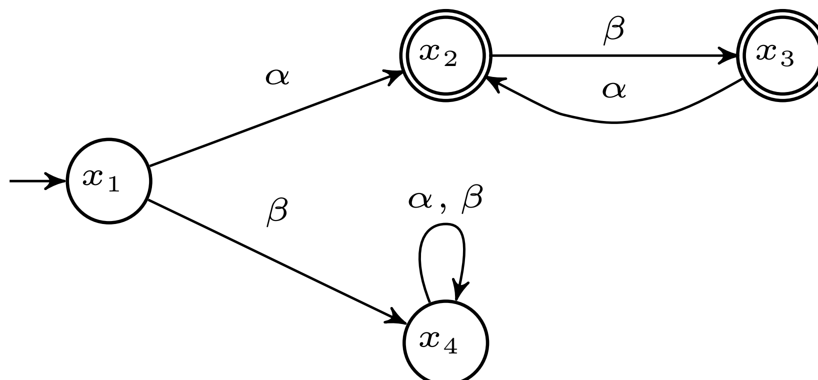


Figura 2.16: Autômato G3, gerado pelo comando *draw(G3,'figure')*.

Função *selfloopnodes*

Assim como a função *strconncomps*, a função *selfloopnodes* pode receber como argumento, tanto um autômato quanto o seu atributo Graph. Essa função retorna um *frozenset* contendo os estados que possuem auto-laços.

O exemplo 33 apresenta as formas de se observar os valores que a função retorna e um código que utiliza a função *strconncomps*, em conjunto com a função *selfloopnodes*, para identificar os componentes fortemente conexos de um autômato. Conforme pode ser visto no exemplo 33, um fator importante que o usuário deve levar em consideração ao usar a função em conjunto com a função *strconncomps*, é que as duas funções retornam variáveis de tipos diferentes.

Exemplo 33

Considere o autômato G3 definido no exemplo 32, cujo diagrama de transição de estados é apresentado na figura 2.16

```
>>> set(selfloopnodes(G3))
frozenset({'x4'})
>>> list(selfloopnodes(G3))
['x4']
>>> next(selfloopnodes(G3))
'x4'
>>>
>>> loops = set(selfloopnodes(G3))
>>> stcomp = list(strconncomps(G3))

#Componentes fortemente conexos nao triviais#
>>> for comp in stcomp:
        if len(comp)>1 or comp==loops:
            print(comp)


{'x4'}
{'x3', 'x2'}
```

Função *condensation*

A função *condensation* recebe o atributo Graph ou o próprio autômato como argumento e retorna a condensação do autômato, no formato de um objeto da classe *MultiDiGraph* do *NetworkX*.

A condensação de um autômato é um mapa onde cada um dos componentes fortemente conexos são contraídos em um único estado. Os estados do mapa de condensação são renomeados para números inteiros e, a relação entre os estados condensados e os do autômato original são armazenadas na chave *mapping* do dicionário *graph* do mapa condensado. O exemplo 34 ilustra como utilizar a função *condensation* e como identificar a relação entre os estados novos e os originais.

Exemplo 34

Considere o autômato G3 definido no exemplo 32 (figura 2.16)

```
>>> C = condensation(G3)


>>> C.nodes()
NodeView((0, 1, 2))
>>> C.edges()
OutEdgeView([(2, 0), (2, 1)])
>>> C.graph['mapping']
{'x3': 0, 'x2': 0, 'x4': 1, 'x1': 2}
>>> C.nodes(data=True)
NodeDataView({0: {'members': {'x3', 'x2'}}, 1: {'members': {'
    x4'}}, 2: {'members': {'x1'}}})
```

Podemos perceber que, como os estados x_2 e x_3 são fortemente conexos, eles foram contraídos no estado "0".

2.2.7 Módulo *structure*

No módulo *structure* estão definidas as funções que realizam modificações na estrutura dos autômatos. Alterar diretamente nas variáveis de um objeto (autômato) da classe *fsa* poderia acarretar em diversos erros no programa. Por exemplo, se o usuário tenta alterar o nome de um estado passando uma nova lista para $G.X$ (estados do autômato), ele precisaria modificar, também, a lista de transições, as listas de alcances e de eventos ativos desse estado, a variável com o nome desse estado em LaTeXe outras variáveis que dependem desses valores. O mesmo ocorreria ao se renomear, deletar ou adicionar eventos e transições. As funções do módulo *structure* foram definidas para que esse processo seja simples para o usuário, sendo todas as variáveis dependentes do valor modificado atualizadas automaticamente.

Para evitar que o autômato original seja alterado, todas as operações realizadas pelas funções do módulo *structure* retornam uma cópia do autômato original com a modificação desejada.

Função *addtransition*

A função *addtransition* é usada para adicionar uma nova transição em um autômato. Essa função recebe dois argumentos, o autômato e a transição. Devido a forma como os valores da transição são utilizados, essa variável pode ser tanto uma *lista* quanto uma *tupla*, variável definida como apresentado no abaixo:

$$(estado_atual, evento, estado_alcancado)$$

Durante a execução da função *addtransition*, inicialmente, verifica-se se o evento

ou algum dos estados passados não faz parte dos conjuntos de eventos e de estados do autômato, respectivamente. Caso algum desses elementos não tenha sido definido no autômato, ele é então adicionado. Por padrão, quando um novo evento é adicionado a um autômato G no DESLab, esse evento também é incluído nos conjuntos G.Sigobs e G.Sigcon.

Caso a transição já exista, ou o evento passado já faça parte da lista de eventos do *"estado_atual"*, então a função retorna o autômato original ou cria um autômato não-determinístico no qual o mesmo evento leva a mais de um estado, respectivamente. Feitas as verificações anteriores, o programa adiciona a transição ao atributo *Graph* da cópia do autômato por meio da função *add_edge* do *NetworkX*. Em seguida atualiza os dicionários *deltaDict* e *gammaDict* e, por fim, retorna o autômato modificado.

O exemplo 35 apresenta como podemos utilizar a função quando os estados e eventos são novos ou quando eles já estão definidos.

Exemplo 35

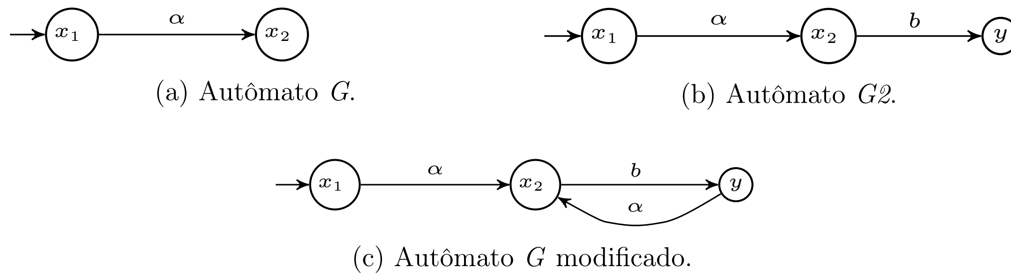
Considere o autômato G, definido no código abaixo:

```
syms('x1 x2 a')
table = [(x1,'x_1'),(x2,'x_2'),(a,r'\alpha')]
X = [x1,x2]
Sigma = [a]
X0 = [x1]
Xm = []
T =[(x1,a,x2)]
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')
```

O código abaixo, adiciona uma transição em G que alcança um novo estado, denotado por y e é rotulada por um novo evento b:

```
>>> G.X
frozenset({'x2', 'x1'})
>>> G.Sigma
frozenset({'a'})
>>> syms('y b')
['y', 'b']
>>> G2=addtransition(G,(x2,b,y))
>>> G2.X
frozenset({'x2', 'x1', 'y'})
>>> G2.Sigma
frozenset({'a', 'b'})
>>> G3=addtransition(G2,(y,a,x2))
>>> G3.X
frozenset({'x2', 'x1', 'y'})
>>> G3.Sigma
frozenset({'a', 'b'})
```

Na figura 2.17 estão os diagramas de estados dos autômatos gerados no exemplo.



(a) Autômato G.

(b) Autômato $G2$.

(c) Autômato G modificado.

Figura 2.17: Autômatos do exemplo 35.

Como a função addtransition foi importada como método da classe fsa, também podemos usá-la da seguinte forma:

```
>>> G.X
frozenset({'x2', 'x1'})
>>> G.Sigma
frozenset({'a'})
>>> syms('y b')
['y', 'b']
>>> G2=G.addtransition((x2,b,y))
>>> G2.X
frozenset({'x2', 'x1', 'y'})
>>> G2.Sigma
frozenset({'a', 'b'})
>>> G3=G2.addtransition((y,a,x2))
>>> G3.X
frozenset({'x2', 'x1', 'y'})
>>> G3.Sigma
frozenset({'a', 'b'})
```

Função *deletetransition*

A função *deletetransition* facilita a remoção de transições de um autômato sem que o usuário tenha a necessidade de alterar outras variáveis. Para remover uma transição, o usuário precisa passar como argumentos o autômato e uma variável, a qual pode ser uma tupla, uma lista ou um *set*, que respeita o modelo abaixo:

$$(estado_atual, evento, estado_alcancado)$$

Durante a execução da função *deletetransition*, primeiramente, cria-se uma cópia do autômato passado. Em seguida, é verificado se o argumento passado respeita o formato descrito no parágrafo anterior. Checadas as informações dos estados e evento, a informação da transição (*edge*) é removida do atributo *Graph* do autômato e altera-se os dicionários *deltaDict* e *gammaDict*. Concluídas as modificações, o autômato modificado é retornado.

No exemplo 36 estão demonstrações de como utilizar a função. Um ponto impor-

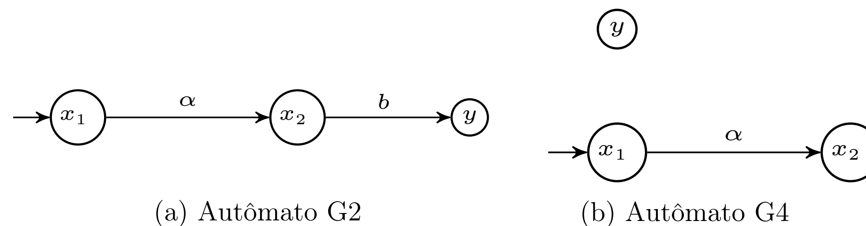
tante que pode ser notado neste exemplo, é que, apesar do estado y não possuir mais transições e o evento b não ser utilizado em nenhuma outra transição, ambos não são removidos do autômato. Isso ocorre pois, nesta função, considera-se a possibilidade de o usuário ainda desejar utilizá-los para outras transformações.

Exemplo 36

Considere o autômato G2, construído no exemplo 35. O código abaixo ilustra como remover uma transição de G2.

```
>>> G2.X
frozenset({'y', 'x1', 'x2'})
>>> G2.Sigma
frozenset({'a', 'b'})
>>> G4=deletetransition(G2,(x2,b,y))
>>> G4.X
frozenset({'y', 'x1', 'x2'})
>>> G4.Sigma
frozenset({'a', 'b'})
```

Os diagramas das transições de estados dos autômatos gerados com o código acima são apresentados na figura 2.18.



(a) Autômato G2 (b) Autômato G4

Figura 2.18: Autômatos do exemplo 36.

Uma vez que a função deletetransition também é um método da classe fsa, ela também pode ser utilizada da seguinte forma:

```
>>> G2.X
frozenset({'y', 'x1', 'x2'})
>>> G2.Sigma
frozenset({'a', 'b'})
>>> G4=G2.deletetransition((x2,b,y))
>>> G4.X
frozenset({'y', 'x1', 'x2'})
>>> G4.Sigma
frozenset({'a', 'b'})
```

Função *renamevents*

A função *renamevents* recebe dois argumentos: um autômato e uma lista. Essa lista deve conter os nomes dos eventos que serão alterados e os seus novos nomes. Os

eventos devem ser arranjados em tuplas, dessa forma é possível renomear mais de um evento sem a necessidade de chamar a função várias vezes. Também é possível passar um dicionário como segundo argumento da função *renamevents*. Nesse caso, as chaves serão os nomes atuais dos eventos e os elementos serão os novos nomes deles.

Na função *renamevents*, uma cópia do autômato é armazenada na variável *auto*, essa cópia será manipulada durante a execução da função e retornada ao final desse processo. Para cada evento passado, seu nome é removido do dicionário *symDict* e atualizado nos conjuntos de eventos ativos e nas transições armazenadas nos dicionários *gammaDict* e *deltaDict*, respectivamente. Por fim, os conjuntos de eventos *Sigma, Sigobs* e *Sigcon* são atualizados.

Vale ressaltar que, como o evento renomeado tem sua chave removida do dicionário *symDict*, o nome utilizado para a criação do diagrama de transição de estados do autômato será obrigatoriamente o novo nome do evento. Caso o usuário deseje utilizar algum símbolo do LaTeX para representar o evento, o mesmo deve ser adicionado no *symDict*. O exemplo 37 mostra como utilizar a função *renamevents* e os tipos permitidos para os argumentos.

Exemplo 37

Considere o autômato G definido no exemplo 35. Podemos renomear seus eventos da seguinte forma apresentada a seguir.

```
>>> syms('y')
>>> G.Sigma
frozenset({'a'})
>>> G2=renamevents(G,[(a,y)])
>>> G2.Sigma
frozenset({'y'})
>>>
>>> G2=renamevents(G,{a:y})
>>> G2.Sigma
frozenset({'y'})
>>>
>>> G2=renamevents(G,[('a','y')])
>>> G2.Sigma
frozenset({'y'})
>>>
>>> G2=renamevents(G,{'a':'y'})
>>> G2.Sigma
frozenset({'y'})
```

A função renamevents também é um método da classe fsa, por isso pode ser usada como descrito no código abaixo:

```
>>> syms('y')
>>> G.Sigma
frozenset({'a'})
```

```
>>> G2=G.renamevents([(a,y)])
>>> G2.Sigma
frozenset({'y'})
```

Função *renamestates*

A função *renamestates* opera da mesma forma que a função *renamevents*, recebendo um autômato e um argumento que pode ser tanto uma lista quanto um dicionário com os nomes dos estados que se deseja renomear. Após copiar o autômato, os estados antigos são removidos do dicionário *symDict* e os conjuntos de estados do autômato *X*, *Xm* e *X0* são atualizadas. Por fim, usando a função *create_FSA_-transdicts* os dicionários *gammaDict*, *deltaDict* e *infoDict* são atualizados.

Existem dois casos especiais que ocorrem quando o segundo argumento for uma *string* e não uma lista. Caso a *string* seja '*number*', será usada a função *lexgraph_-numbermap* para renomear os estados para números. Se a *string* for '*lex*', usando a função *lexgraph_alphamap* os estados serão renomeados para uma *string* dos eventos necessários para alcançá-los.

Deve-se notar que, da mesma forma que ao renomear um evento, caso o usuário queira utilizar um símbolo do LaTeX para representar o novo estado no diagrama de transição de estados do autômato, deve-se inserir a chave para esse estado no dicionário *symDict*. O exemplo 38 mostra como podemos renomear os estados de um autômato utilizando a função *renamestates*.

Exemplo 38

Considere o autômato G, definido no exemplo 35. Um estado de G pode ser renomeado da seguinte forma:

```
>>> G.tmx('table')

Transition Matrix:

     a
x1   x2
x2   N/D

>>> syms('x3')
['x3']
>>> G2=G.renamestates({x1:x3})
>>> G2.('table')

Transition Matrix:

     a
x3   x2
x2   N/D
```

```
>>> G3=G.renamestates('number')
>>> G3.tmx('table')

Transition Matrix:

     a
0    1
1    N/D

>>> G4=G.renamestates('lex')
>>> G4.tmx('table')

Transition Matrix:

               a
a              N/D
epsilon        a
```

Função *addevent*

A função *addevent*, como o nome sugere, adiciona eventos à lista de eventos de um autômato. Assim como as outras funções do módulo *structure*, as alterações são feitas em uma cópia do autômato que é retornada ao final do processo. A função *addevent* deve receber como argumentos um autômato e uma lista de eventos. A função *addevent* também aceita que a lista de evento seja passada como uma tupla, um *set* e, caso seja apenas um evento, uma *string*. Por padrão, os novos eventos adicionados também são inseridos nas listas de eventos controláveis e observáveis, *Sigcon* e *Sigobs*, respectivamente. Os eventos devem inicialmente, declarados com a função *syms*, como pode ser visto no exemplo 39.

Exemplo 39
Considere o autômato G definido no exemplo 35. Podemos adicionar novos eventos em G das seguintes formas:

```
>>> syms('b c d')

['b', 'c', 'd']
>>> G1=addevent(G,[b,c])

>>> G1.Sigma

frozenset({'b', 'c', 'a'})
>>> G2=addevent(G,'c')

>>> G2.Sigma

frozenset({'c', 'a'})
```

Como a função addevent é importada como método da classe fsa, também podemos utilizá-la da seguinte forma:

```
>>> syms('b c d')

['b', 'c', 'd']
>>> G1=G.addevent([b,c])

>>> G1.Sigma

frozenset({'b', 'c', 'a'})
>>> G2=G.addevent('c')

>>> G2.Sigma

frozenset({'c', 'a'})
```

Função *deletevent*

A função *deletevent* recebe um autômato e um evento como argumentos e remove o evento das listas de eventos e transições do autômato. Ela foi importada pela classe *fsa* como um de seus métodos.

Uma cópia do autômato é manipulada para a remoção do evento. Inicialmente, remove-se o evento da lista de eventos, *Sigma*. Em seguida, utiliza-se o grafo no atributo *Graph* do autômato para percorrer todas as transições (*edges*). Caso uma transição seja causada pelo evento a ser removido, essa transição é apagada e os dicionários dos atributos *gammaDict* e *deltaDict* são modificados de acordo com as informações das transições restantes. O exemplo abaixo mostra como deletar um evento, tanto pelo método do autômato, quanto usada a própria função *addevent*.

Exemplo 40

Considere o autômato G3 definido no exemplo 32. Podemos remover eventos de G3 como mostrado no código abaixo.

```
>>> G3.Sigma

frozenset({'a', 'b'})
>>> G = G3.deletevent(a)

>>> G.Sigma

frozenset({'b'})
>>> G3.Sigma

frozenset({'a', 'b'})
>>> G2 = deletevent(G3,a)

>>> G2.Sigma
```

```
frozenset({'b'})
```

Função *addstate*

A função *addstate* recebe um autômato e um estado como argumentos. Ela adiciona o estado ao autômato e, caso o novo estado seja marcado, o valor *True* também deve ser passado como argumento para o parâmetro *marked*.

No grafo do atributo Graph do autômato, os estados são representados como *nodes*. Cada *node* possui um *label*, que é utilizado pelos módulos responsáveis por criar o diagrama de transição de estados do autômato. A estrutura do atributo *Graph* de um autômato, como também seus *nodes* e *labels* serão descritas detalhadamente na seção 2.3.

O primeiro passo executado pela função *addstate* é inserir o novo *node* no grafo do autômato. Para evitar que do novo *node* sobreponha outro já existente, os *labels* de todos os *nodes* do grafo são comparados ao novo *label*. No passo seguinte, o novo estado é adicionado ao conjunto de estados X e, caso o parâmetro *marked* tenha valor *True*, o novo estado também é adicionado ao conjunto de estados marcados. O exemplo a seguir mostra as diferentes formas de se utilizar a função *addstate*, uma vez que essa função foi importada como um método da classe *fsa*.

Exemplo 41

Considere o autômato G definido no exemplo 35. Podemos adicionar novos estados como descrito no código abaixo.

```
>>> G2 = addstate(G,'y')

>>> G2.X

frozenset({'x1', 'x2', 'y'})
>>> G3 = G2.addstate('k',True)

>>> G3.X

frozenset({'x1', 'x2', 'k', 'y'})
>>> G3.Xm

frozenset({'k'})
```

Função *deletestate*

A função *deletestate* recebe um autômato e um estado como argumentos. Ela cria uma cópia desse autômato, dele remove o estado passado como argumento e, ao fim, retorna o autômato modificado.

Inicialmente a função *deletestate* verifica se o estado passado pertence ao conjunto de estados do autômato, armazenado no atributo *X*. Em seguida, ela remove o estado passado do conjunto *X*, e dos conjuntos *X, X0* e *Xm*, se o estado passado for um estado inicial ou um estados marcado, respectivamente. Por fim, os dicionários *gammaDict e deltaDict* são reconstruídos, a partir das informações das transições do autômato contidas no grafo do atributo *Graph*, descartando-se as informações associadas ao estado removido. O exemplo abaixo mostra como remover o estado de um autômato.

Exemplo 42

Considere o autômato G3 definido no exemplo 32. O código abaixo remove o estado de G3 usando o método deletestate da classe fsa, o qual é definido a partir da função deletestate.

```
>>> G3.tmx('table')

Transition Matrix:

     b    a
x4   x4   x4
x2   x3   N/D
x3   N/D  x2
x1   x4   x2

>>> G3 = G3.deletestate(x4)
>>> G3.tmx('table')

Transition Matrix:

     b    a
x2   x3   N/D
x3   N/D  x2
x1   N/D  x2
```

Função *renametransition*

A função *renametransition* substitui o evento de uma transição. Para isso, ela deve receber um autômato e uma lista contendo a transição a ser substituída. Essa lista deve conter os estados da transição e uma tupla com os dois eventos, o antigo e o novo. Como a função *renametransition* faz uso da função *addtransition*, caso o novo evento não faça parte do conjunto de eventos do autômato, ele será adicionado aos conjuntos de eventos, de eventos observáveis e de eventos controláveis do autômato.

Exemplo 43

Considere o autômato G3 definido no exemplo 32. O código abaixo apresenta duas

maneiras para substituir a transição do estado x2 para o estado x3 rotulada por b, por uma transição rotulada pelo evento c.

```
>>> G3.tmx('table')

Transition Matrix:

     a    b
x2   N/D  x3
x4   x4   x4
x3   x2   N/D
x1   x2   x4

>>> G = G3.renametransition([x2,(b,c),x3])
>>> G.tmx('table')

Transition Matrix:

     a    c    b
x2   N/D  x3   N/D
x3   x2   N/D  N/D
x1   x2   N/D  x4
x4   x4   N/D  x4

>>> G2 = renametransition(G3,[x2,(b,c),x3])
>>> G2.tmx('table')

Transition Matrix:

     a    c    b
x2   N/D  x3   N/D
x3   x2   N/D  N/D
x1   x2   N/D  x4
x4   x4   N/D  x4
```

Função *addselfloop*

A função *addselfloop* cria um auto-laço em um estado de um autômato. Para isso, ela deve receber o autômato, o estado e o evento que serão usados na criação da nova transição. O auto-laço é criado usando-se a função *addtransition* e, caso o evento passado como argumento da função não pertença ao conjunto de eventos do autômato, ele será adicionado aos conjuntos de eventos, de eventos observáveis e de eventos controláveis do autômato. Vale também ressaltar que a função *addselfloop* foi importada pela classe *fsa* como um de seus métodos.

Função *transitions*

A função *transitions* recebe um autômato como argumento e, acessando seu atributo *Graph*, cria uma lista com todas as transições desse autômato. Essa função foi importada pela classe *fsa* como um de seus métodos. O exemplo abaixo mostra como utilizá-la.

Exemplo 44
Considere o autômato G3 definido no exemplo 32. A função transitions pode ser usada para obter uma lista com todas as transições de G3, conforme mostrado no código abaixo.

```
>>> G3.transitions()

[('x3', 'a', 'x2'), ('x1', 'a', 'x2'), ('x1', 'b', 'x4'), ('x4
    ', 'b', 'x4'), ('x4', 'a', 'x4'), ('x2', 'b', 'x3')]
>>> transitions(G3)

[('x3', 'a', 'x2'), ('x1', 'a', 'x2'), ('x1', 'b', 'x4'), ('x4
    ', 'b', 'x4'), ('x4', 'a', 'x4'), ('x2', 'b', 'x3')]
>>>
```

Função *transitions_iter*

Semelhante a função *transitions*, a função *transitions_iter* também retorna as transições de um autômato. Entretanto, ela retorna um iterador para as transições do autômato. Assim como a função *transitions*, a função *transitions_iter* também foi importada como um método da classe *fsa*.

Exemplo 45
Considere o autômato G3 definido no exemplo 32. A função transitions_iter pode ser utilizada conforme ilustrado no código a seguir.

```
>>> Trans = G3.transitions_iter()

>>> for t in Trans:
        print(t)


['x3', 'a', 'x2']
['x1', 'a', 'x2']
['x1', 'b', 'x4']
['x4', 'b', 'x4']
['x4', 'a', 'x4']
['x2', 'b', 'x3']
>>> Trans = transitions_iter(G3)

>>> for t in Trans:
```

```
        print(t)


['x3', 'a', 'x2']
['x1', 'a', 'x2']
['x1', 'b', 'x4']
['x4', 'b', 'x4']
['x4', 'a', 'x4']
['x2', 'b', 'x3']
```

Função *lexgraph_dfs*

A função *lexgraph_dfs* recebe um autômato como argumento, o qual deve ser um autômato determinístico. Essa função percorre esse autômato, a partir do estado inicial, realizando uma busca em profundidade. Ao término da busca, a função *lexgraph_dfs* retorna uma lista com os estados percorridos durante a busca, organizados de acordo com a ordem em que eles foram visitados. Esse procedimento é feito da seguinte forma: a partir do estado inicial armazenado no conjunto do atributo *X0* do autômato, obtém-se a lista de eventos ativos para o estado inicial pelo método *Gamma*. Para cada um dos eventos ativos obtidos, o estado alcançado é determinado por meio do método *delta*. Todo o processo é repetido para cada um dos estados visitados e seus nomes são armazenados na lista que é retornada ao final.

Exemplo 46

Seja o autômato definido no DESLab pelo código abaixo e cujo diagrama de transição de estados pode ser visto na figura 2.19.

```
syms('v w x y z  a b')
X = [v,w,x,y,z]
Table=[]
Sigma = [a,b]
X0 = [v]
Xm = []
T =[(v,a,w),(w,a,x),(w,b,y),(y,b,z)]
G = fsa(X,Sigma,T,X0,Xm,Table,name='$G$')
```

O código abaixo aplica a função lexgraph_dfs em G.

```
>>> lexgraph_dfs(G)
['v', 'w', 'x', 'y', 'z']
```

Função *lexgraph_alphamap*

Assim como a função *lexgraph_dfs*, a função *lexgraph_alphamap* também executa uma busca em profundidade a partir do estado inicial de um autômato determinís-

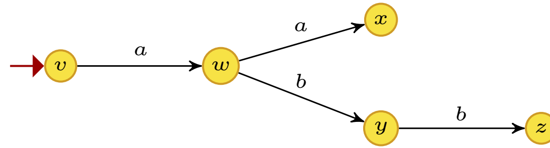


Figura 2.19: Autômato G definido no exemplo 46.

tico. No entanto, essa busca retorna um dicionário que associa cada estado alcançável do autômato com a menor sequência de eventos que leva do estado inicial até esse estado. O exemplo abaixo ilustra a aplicação da função *lexgraph_alphamap*.

Exemplo 47

Considere o autômato G definido no exemplo 46 e apresentado na figura 2.19.

```
>>> lexgraph_alphamap(G)
{'v': 'epsilon', 'w': 'a', 'x': 'aa', 'y': 'ab', 'z': 'abb'}
```

Função *lexgraph_numbermap*

A função *lexgraph_numbermap* opera da mesma forma que a função *lexgraph_dfs*. No entanto, essa função retorna um dicionário, cujas chaves são os estados do autômato e os valores são números entre 1 e N (em que N é o número de estados do autômato) que correspondem a ordem na qual os estados foram encontrados durante a busca em profundidade. Vale ressaltar que não são criadas chaves com os estados que não são acessíveis a partir do estado inicial. O exemplo abaixo ilustra o uso dessa função.

Exemplo 48

Considere o autômato G representado na figura 2.19.

```
>>> lexgraph_numbermap(G)
{'v': 0, 'w': 1, 'x': 2, 'y': 3, 'z': 4}
```

Função *size*

A função *size* recebe um autômato como argumento e, a partir do seu atributo *Graph*, retorna o número de *nodes*, ou estados, presentes no autômato.

2.2.8 Módulo *utilities*

O módulo *utilities* foi construído para conter as funções que não necessariamente efetuam operações com autômatos mas atuam como suporte durante a execução do DESLab.

Função *syms*

A função *syms* é uma das funções mais utilizadas no módulo, pois ela define as variáveis que são criadas para os estados e eventos de todos os autômatos.

Ao chamar a função *syms*, o usuário deve passar uma *string* contendo todas as variáveis desejadas separadas por um espaço simples, então, para cada um dos elementos contidos nela, será criada uma variável de mesmo nome e com a *string* desse elemento atribuída a ela. Caso os nomes dos estados e/ou eventos sejam números inteiros, não é necessário utilizar a função *syms*.

O exemplo 49 ilustra como a função deve ser utilizada na definição de um autômato.

Exemplo 49

Conforme mostrado no código abaixo, a função syms é utilizada para declarar as variáveis que irão representar os estados e eventos.

```
syms('x y a b g')
X = [x,y,3]
Sigma = [a,b,g]
X0 = [x]
Xm = [x,3]
T =[(y,a,x),(x,a,x),(x,g,3),(y,b,y),(3,g,y),(3,a,y),(3,b,3)]
G1 = fsa(X,Sigma,T,X0,Xm,table=[],name='$G_1$')
```

Função *which*

A função *which* é utilizada para identificar arquivos executáveis. É uma função de suporte utilizada durante a instalação do DESLab para identificar a localização dos softwares externos utilizados. Durante a sua execução, ela deve receber uma *string* com o nome do arquivo contendo, ou não, o caminho até ele. Esse argumento é desmembrado a fim de determinar a extensão do arquivo, que é retornado, caso esse seja um arquivo executável.

2.3 Pasta *graphics*

A pasta *graphics* contém os módulos necessários para produzir uma imagem vetorizada, que servirá de instrução para a produção de um arquivo PDF com diagramas de transição de estados de autômatos. Definidos os autômatos, a função *draw* é usada para produzir esse arquivo PDF. De forma resumida, durante a execução da função *draw*, a construção do arquivo PDF obedece aos seguintes passos:

1. A estrutura do autômato é formatada em um arquivo DOT com o auxílio do *NetworkX*;

2. O arquivo DOT é convertido para o formato XDOT usando o *Graphviz*;
3. Um arquivo TEX é criado a partir do arquivo XDOT;
4. O *TexLive* é utilizado para gerar o arquivo PDF.

Nas subseções a seguir, serão descritas todas as classes e funções contidas no módulo *drawing* que, conforme descrito na tabela 2.1, pertence a pasta *graphics*. Como essas funções trabalham em conjunto para gerar o arquivo PDF, serão apresentados diagramas que ilustram as relações entre elas, com o objetivo de facilitar a compreensão da dependência que essas funções têm entre si, bem como das variáveis que elas operam. Exemplos serão utilizados para ilustrar a manipulação dessas funções.

2.3.1 Módulo *drawing*

O módulo *drawing* contém todas as funções necessárias para o processamento dos autômatos e construção dos arquivos necessários para a produção do arquivo PDF contendo os diagramas de transição de estados dos autômatos.

A tabela 2.9 lista todas as variáveis predefinidas no módulo. Elas serão utilizadas pelas diversas funções, descritas nas subseções abaixo, na determinação dos diretórios onde os arquivos serão armazenados, dos nomes dos arquivos DOT e TEX criados e na definição da expressão regular que será utilizada pela função *determine_size*.

Tabela 2.9: Variáveis Predefinidas.

Variável	**Valor**
VIEWERS	{‘evince’:‘evince’, ‘acrobat reader’:‘acroread’}
VIEWER	VIEWERS[‘evince’]
DOTINTERFACE	‘DotInterfaceFile.dot’
TEXPAGEOUT	‘TexOutput.tex’
WORKING	‘working’
OUTPUT	‘output’
patternDim	re.compile(r‘\\node \(\w\d+\) at \((?P<coordX>\d+\.?\d*)pt, (?P<coordY>\d+\.?\d*)pt\)’)
dir_path	{WORKING: ‘’, OUTPUT: ‘’, TEXFILES: ‘’}
BEAMER_TEMPLATE	*String* com o cabeçalho para o arquivo TEX
FIGURE_TEMPLATE	*String* com o cabeçalho para o arquivo TEX
EMPTY_AUTOMATON	Código TEX para o autômato vazio
PEAMBLE_DIC	{‘beamer’: BEAMER_TEMPLATE, ‘figure’: FIGURE_TEMPLATE, ‘figurecolor’: FIGURE_TEMPLATE}
STATE_LAYOUT	Dicionário de estilos da classe *graphic*

Função *draw*

A função *draw* e a principal função do DESLab para a geração dos diagramas de transição de estados dos autômatos. Ela manipula todas as funções da pasta *deslab.graphics* para gerar os arquivos PDF e portanto deve ser a função chamada quando se desejar criar o arquivo PDF de um autômato.

Os argumentos de entrada podem ser um ou mais autômatos, e o último argumento pode ser o estilo de formatação para o arquivo TEX. As opções válidas para o argumento de estilo são as *strings 'beamer'*, *'figure'* e *'figurecolor'*. Caso o argumento de estilo não seja passado, ou seja inválido, o *beamer* será utilizado.

A figura 2.21 apresenta o diagrama dos processos que ocorrem com a chamada da função *draw*. Durante a execução dela, primeiro a função *setupdir* é chamada para definir os diretórios que serão utilizados. Em seguida, de acordo com o argumento de estilo passado, o cabeçalho do arquivo TEX é definido usando o dicionário PREAMBLE_DIC, como apresentado na tabela 2.9. Para cada autômato passado como argumento, ocorre a verificação do limite máximo de 100 estados. Caso ele seja aprovado, o resultado do processamento feito pela função *automaton2page* para esse autômato é concatenado à *string preamble_tex* que armazena o cabeçalho do arquivo TEX e os códigos dos autômatos já processados.

Com a conclusão de todas as etapas anteriores, a *string preamble_tex* é passada para a função *write_texfile*, que cria o arquivo TEX. Em seguida, a função *tex2pdf*, que compila o arquivo TEX, e a função *openviewer*, que abre o arquivo PDF, são chamadas.

O exemplo 50 ilustra as possíveis formas de utilizar a função *draw*.

Exemplo 50

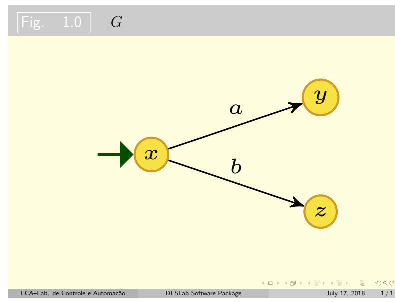
Seja o autômato G, definido pelo seguinte código do DESLab:

```
syms('x y z a b')
X = [x,y,z]
table=[]
Sigma = [a,b]
X0 = [x]
Xm = []
T =[(x,a,y),(x,b,z)]
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')
```

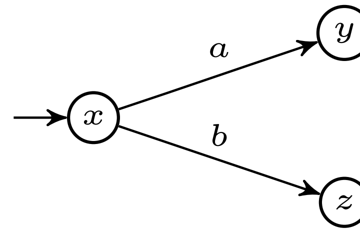
Podemos chamar a função draw das seguintes formas:

```
draw(G)
draw(G,'beamer')
draw(G,'figure')
draw(G,'figurecolor')
```

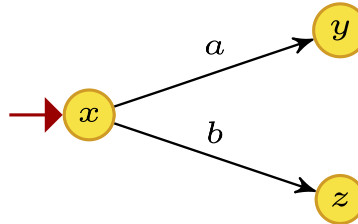
A figura 2.20 apresenta as figuras dos diagramas de transição de estados do autômato G produzidas para cada estilo.



(a) Estilo ‘beamer’



(b) Estilo ‘figure’



(c) Estilo ‘figurecolor’

Figura 2.20: Estilos da função *draw*..

A função draw também pode ser aplicada para mais de um autômato:

```
draw(G,G2...,Gn)
draw(G,G2...,Gn,'beamer')
draw(G,G2...,Gn,'figure')
draw(G,G2...,Gn,'figurecolor')
```

Função *setupdir*

A função *setupdir* não recebe nenhum argumento. Ao ser chamada, ela define os caminhos das pastas necessárias para a construção dos diagramas de transição de estados e a pasta onde o arquivo PDF final será salvo. Primeiro o caminho do módulo *drawing* é salvo na variável local *path_drawing*. Em seguida, a variável *path_drawing* é usada para identificar o caminho da pasta na qual está o módulo drawing. Por fim, as chaves *working*, *output* e *texfiles* do dicionário *dir_path*, visto na tabela 2.9, tem seus valores modificados para os caminhos das pastas de mesmo nome das chaves.

Função *write_texfile*

De acordo com o digrama de processos da função *draw* (figura 2.21), após a execução da função *setupdir*, o conteúdo do arquivo TEX, com o qual se gera o arquivo PDF com os digramas de transição de estados, é construído por meio da *função automaton2page* e, também, usando-se o dicionário *PREAMBLE_DIC*. Como a

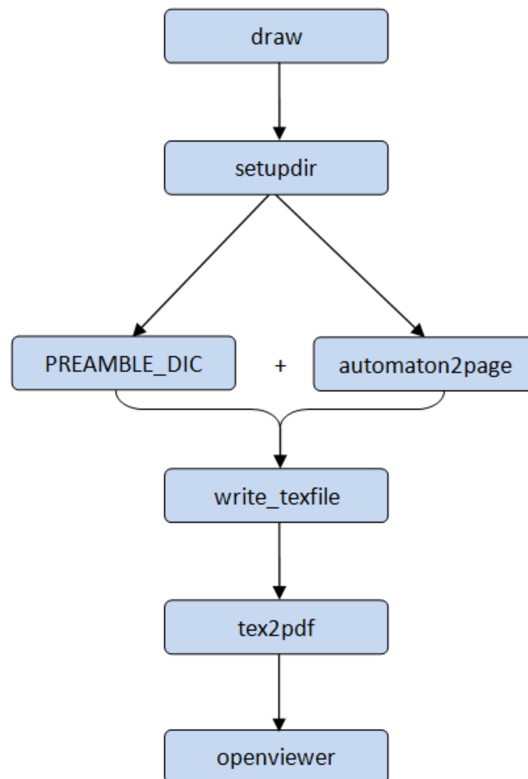


Figura 2.21: Diagrama de processos da função *draw*.

função automaton2page é consideravelmente complexa, a sua análise será postergada, sendo sua descrição detalhada mais adiante, na subseção intitulada "função *automaton2page*".

Uma vez que o conteúdo do arquivo Tex foi construído, então, de acordo com a figura 2.21, a função *write_texfile* é executada. Essa função recebe duas *strings* como argumentos, que são armazenadas nos parâmetros *'TexString'* e *'TexfileOut'*, e cria um arquivo TEX usando essas *strings* para definir o conteúdo e o nome desse arquivo, respectivamente. Quando a função *write_texfile* é chamada pela função *draw*, as *strings preamble_tex* e *TEXPAGEOUT*, contendo respectivamente o código do arquivo TEX e seu nome, são passadas como argumentos.

Vale ressaltar que uma pequena alteração é feita no código TEX, recebido como argumento e armazenado no parâmetro *TexString*, ou seja, para encerrar o código TEX, o segmento "\end{*document*}" é concatenado ao final da *string TexString*. O arquivo TEX é então criado no caminho especificado em "*dir_path[WORKING]*", com o nome armazenado no parâmetro TexfileOut, e o código é inserido no arquivo.

Função *tex2pdf*

Utilizada para produzir o arquivo PDF, a função *tex2pdf* cria uma variável global, *window_counter*, que armazena quantas vezes a função foi chamada durante a execução do DESLab. Isso evita que a criação de um novo PDF sobreponha um arquivo já criado, uma vez que essa numeração é utilizada para nomear o arquivo. O nome do arquivo PDF é construído seguindo o modelo mostrado na figura 2.22.

Figure - xy

x - Número identificador da chamada da função *draw*

y - Número de páginas (autômatos) no arquivo.

Figura 2.22: Estrutura do nome do arquivo PDF.

A função *tex2pdf* recebe o nome do arquivo TEX como argumento e, a partir disso, utilizando o módulo *Subprocess* do *Python*, o seguinte código é executado, via *prompt* de comando, para gerar o arquivo PDF por meio do LaTeX:

```
>pdflatex -interaction=batchmode -no-shell-escape -output-
   directory dir_path[OUTPUT] -jobname pdf_outputname
   tex_filename
```

No código acima, "*dir_path[OUTPUT]*" representa o caminho onde o PDF será criado, "*pdf_outputname*" é o nome que será dado ao arquivo, "*tex_filename*" é o caminho com do arquivo TEX, previamente criado, "*-interaction=batchmode*" especifica o modo de interação do LaTeX e "*-no-shell-escape*" oculta a janela de comando.

Conforme apresentado na figura 2.23, a função *tex2pdf* retorna a *string pdf_outputname+'.pdf'*, que corresponde ao nome do arquivo PDF criado.

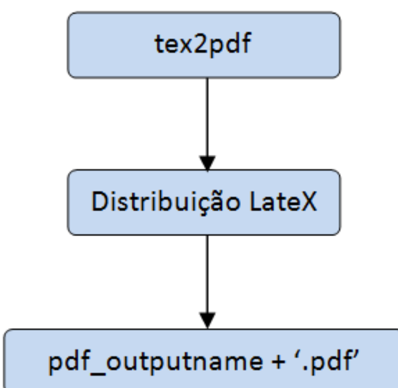


Figura 2.23: Diagrama da função *tex2pdf*.

Função *openviewer*

A função *openviewer* passa o nome do arquivo PDF, criado pela função *tex2pdf*, para o sistema operacional do computador. Então, o software padrão para arquivos PDF do computador do usuário é chamado para ler o documento. Por padrão, o PDF sempre será buscado na pasta *graphics.output*.

Função *automaton2page*

A função *automaton2page* é responsável pela construção do código TEX de cada um dos diagramas de transição de estados que irão compor o arquivo PDF. Para construir as imagens vetorizadas, o pacote TIKZ é utilizado. Por meio desse pacote é possível configurar cada elemento da figura, espaçamento, cor, dimensão, formas dentre outros. A função *automaton2page*, recebe como argumentos um autômato e uma *string*, a qual determina o estilo de formatação usado no arquivo TEX, ou seja, *beamer*, *figure* ou *figurecolor*.

Conforme apresentado na figura 2.24, a construção do código TEX de cada autômato envolve as funções *determine_size* e *automaton2tikfig* e a classe *graphic*, as quais serão analisadas detalhadamente mais adiante. Em linhas gerais, esse procedimento é constituído dos seguintes passos:

1. Os atributos *LineColor, FillColor, state* e *initpos*, do objeto da classe *graphic* associado ao autômato, são armazenados;

2. Por meio da função *automaton2tikfig*, obtém-se a *string* com o código de construção da figura TIKZ do diagrama de transição de estados do autômato. Esse valor é armazenado na variável *tikz_code*;

3. O código inicial de configuração da figura TIKZ, armazenado na variável *init_-tex*, é editado de acordo com o estilo especificado (isto é, *'beamer'*, *'figure'* ou *'figurecolor'*) e os atributos obtidos no passo 1. Caso o estilo seja o *beamer*, o valor retornado pela função *determine_size* também é utilizado na configuração;

4. A variável *figure_texcode* é retornada. Ela é uma *string* composta da concatenação das *strings init_tex, tikz_code* e uma *string* final, definida para cada estilo da forma apresentada na tabela 2.10.

Classe *graphic*

A classe *graphic* armazena os parâmetros de configuração gráfica de um autômato. Eles serão utilizados pelo *Graphviz* na construção do arquivo DOT. Todos os pa-

Tabela 2.10: Complemento da *string figure_ texcode* para cada estilo.

Estilo	***String***
beamer	$'\backslash\backslash end\{tikzpicture\}\}\backslash n\backslash\backslash end\{center\}\backslash n\backslash\backslash end\{frame\}'$
figurecolor	$'\backslash\backslash end\{tikzpicture\}\backslash n\backslash\backslash newpage\backslash n'$
figure	$'\backslash\backslash end\{tikzpicture\}\backslash n\backslash\backslash newpage\backslash n'$

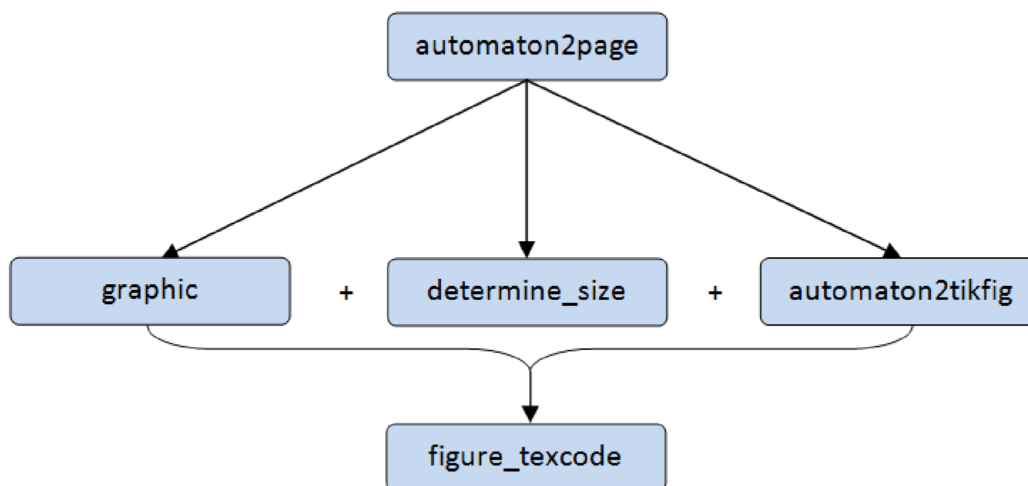


Figura 2.24: Diagrama de operações da função *automaton2page*.

râmetros dessa classe já possuem valores predefinidos, mas eles podem ser modificados. A relação dos parâmetros de um objeto da classe *graphic* pode ser vista na tabela 2.11. O parâmetro *style* possui uma lista de valores específicos que podem ser atribuídos a ele. A modificação do parâmetro *style* pode acarretar em mudanças fixas nos atributos *direction*, *FillColor*, *LineColor*, *state* e *initpos*, conforme descrito na tabela 2.12.

Tabela 2.11: Parâmetros da Classe *graphic*.

Parâmetro	Significado	Predefinição
program	Extensão do arquivo da imagem vetorizada	'dot'
ranksep	Espaçamento das filas usadas no Graphviz para separar os estados para definir o quadro da imagem	0.25
nodesep	distância entre imagens em uma mesma fila	0.25
direction	direção: LR -Da esquerda para a direita; UD - De cima para baixo	'LR'
FillColor	Cor interna do estado	('plantfill',76)
LineColor	Cor da borda do estado	('plantline', 85)
style	Altera os atributos *direction*, *FillColor*, *LineColor*, *state* e *initpos*	'normal'

O formato do argumento que deve ser passado para os parâmetros *FillColor* e *Li-*

Tabela 2.12: Parâmetros fixados para cada estilo.

style	*state*	*initpos*	*direction*	*FillColor*	*LineColor*
‘normal’	‘inner sep= 0.25pt, minimum size=0pt, circle,’	‘’	Variável	Variável	Variável
‘rectangle’	‘minimum height=0pt, inner sep=0.3pt, inner xsep=0.1pt, rectangle’	‘’	‘LR’	(‘plantfill’,76)	(‘plantline’,85)
‘crectangle’	‘minimum height=0mm, inner sep=2mm, chamfered rectangle’	‘’	‘LR’	(‘superfill’,76)	(‘superline’,85)
‘verifier’	‘minimum height=0pt, inner sep=0.3pt, inner xsep=0.1pt, rectangle’	‘above’	‘LR’	(‘yellowfill’,76)	(‘yellowline’,85)
‘diagnoser’	‘minimum height=0pt, inner xsep=0.1pt, inner ysep=0.3pt, rectangle’	‘above’	‘UD’	(‘skyfill’,76)	(‘skyline’,85)
‘observer’	‘minimum height=0pt, inner sep=0.3pt, inner xsep=0.1pt, rectangle’	‘above’	‘UD’	(‘skyfill’,76)	(‘skyline’,85)
‘vertical’	‘inner sep=0.2pt, minimum size=0pt, circle’	‘above’	‘UD’	(‘superfill’,76)	(‘superline’,85)

neColor deve ser uma tupla contendo uma *string*, com o nome da cor, e um número, que representa a intensidade da cor. As cores utilizadas no DESLab (*‘plantfill’*, *‘plantline’*, *‘skyfill’* e *‘superline’*) possuem nomes predefinidos no preâmbulo do código TEX (acessível no dicionário *PREAMBLE_DIC*) mas, caso o usuário deseje, qualquer nome da lista de cores do pacote TIKZ também pode ser utilizado. No exemplo 51, mostra-se como um objeto da classe *graphic* pode ser definido e como a alteração dos valores dos parâmetros pode ser feita.

Exemplo 51

Considere o seguinte código:

```
>>> var1 = graphic();
>>> var2 = graphic(style = 'normal', program = 'dot', ranksep
   = 0.25, nodesep= 0.25, direction = 'LR', FillColor=('
   plantfill',76), LineColor= ('plantline',85))
```

Observe que, como todos os valores passados para a classe em var2 são os valores padrões, então var1 = var2.

Uma vez que todos os parâmetros possuem valores predefinidos, poderíamos passar apenas os valores que desejamos modificar, como feito a seguir:

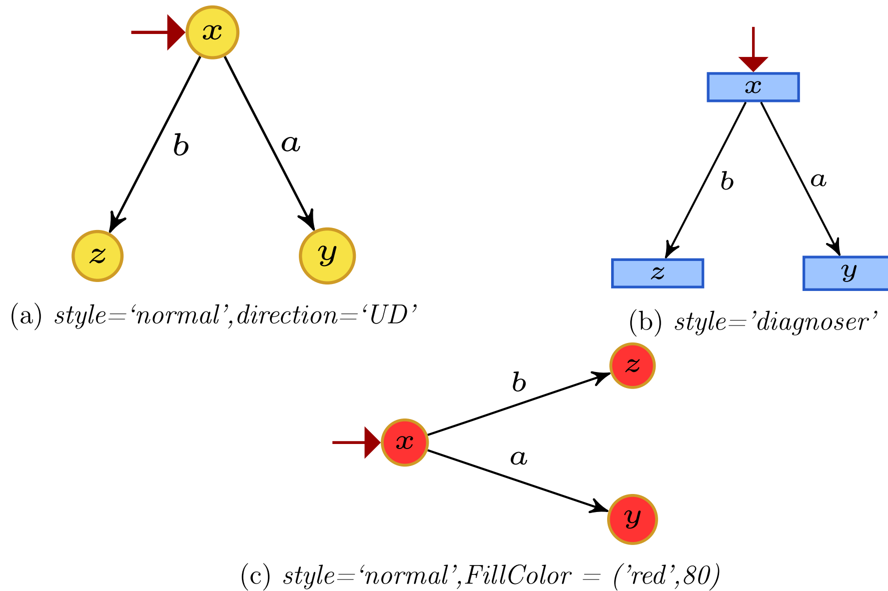
```
>>> var = graphic(style = 'normal', direction = 'UD')
```

Usando o autômato definido no exemplo 50. Para aplicar essas modificações no autômato, utilizamos o método setgraphic.

```
>>> G.setgraphic(style='normal',direction='UD')
```

```
>>> draw(G,‘figurecolor’)
>>> G.setgraphic(style=’diagnoser’)
>>> draw(G,‘figurecolor’)
>>> G.setgraphic(style=‘normal’,FillColor = (‘red’,80))
>>> draw(G,‘figurecolor’)
```

O resultado da execução do código acima é mostrado na figura 2.25.



(a) *style=‘normal’,direction=‘UD’*

(b) *style=’diagnoser’*

(c) *style=‘normal’,FillColor = (’red’,80)*

Figura 2.25: Modificação dos parâmetros gráficos de G.

Função *determine_size*

Utilizada apenas para o modo *beamer*, a função *deterimine_size* recebe uma *string* contendo o código de construção da imagem e o número de estados do autômato. No código TEX, armazenado no parâmetro *texfile*, uma busca pelas coordenadas de posicionamento dos estados e transições é feita. Para identificar esses valores, a referência de busca, armazenada na variável *patternDim* (Tabela 2.9)) é utilizada. A partir das coordenadas identificadas, dimensiona-se o tamanho, em milímetros, do quadro da imagem, e esse valor é retornado.

Função *automaton2tikfig*

A função *automaton2tikfig* é responsável por criar o segmento do código TEX que será usado para gerar a figura de um autômato com o pacote TIKZ do LaTeX.

O processo da função *automaton2tikfig* é apresentado na figura 2.26. Quando chamada, essa função deve receber um autômato como argumento. Durante sua execução, ela chama a função *auto2dot*, que retorna uma string com a estrutura do autômato na linguagem DOT. Em seguida, a string obtida com a função *auto2dot*

é modificada, usando os atributos *Graph* e *graphic* do autômato. Essa nova string, armazenada na variável *auto_dotfile*, é usada para criar o conteúdo do arquivo *DotInterfaceFile.dot*, localizado na pasta working, cujo caminho está armazenado em *dir_path[WORKING]*.

Após a edição do arquivo *DotInterfaceFile.dot*, o código abaixo é utilizado para processar esse arquivo usando o *Graphviz*, via *prompt* de comando.

```
> dot -Txdot DotInterfaceFile.dot |  python dot2tex\_deslab.py
    -ftikz --codeonly --texmode math
```

No código acima, *-Txdot* é o comando que configura o formato da saída do *Graphviz* para um arquivo XDOT, o operador "|"é utilizado para que, ao término do processamento do *Graphviz*, o arquivo XDOT resultante seja passado automaticamente para o *script dot2tex_deslab*, o qual enfim, gera o código TEX. O comando *-ftikz* especifica o formato da figura, *–codeonly* define como será retornado o resultado e *math* informa para o analisador como o código deve ser lido.

Por fim, o segmento do código TEX referente à figura do autômato, armazenado na *string fig_Texcode* é retornado pela função *automaton2tikfig*.

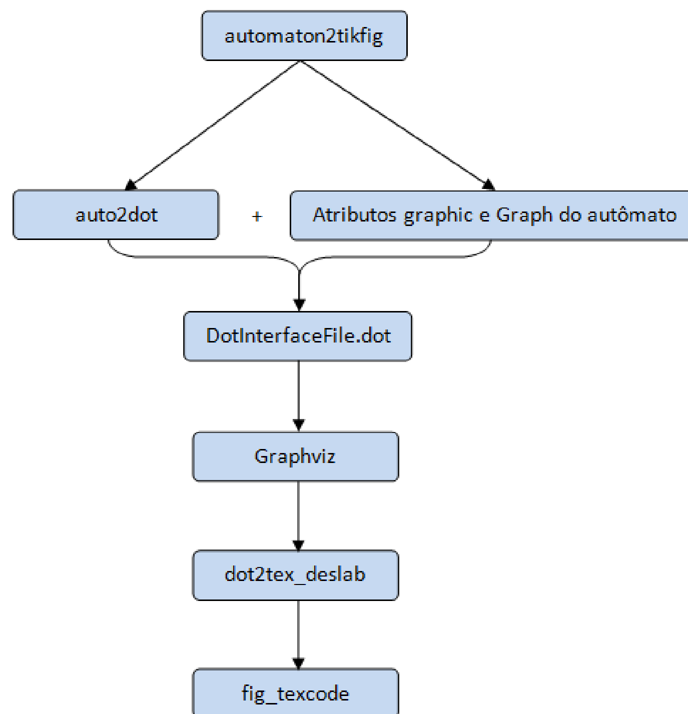


Figura 2.26: Diagrama da função *automaton2tikfig*.

Função *auto2dot*

A função *auto2dot* constrói a estrutura do autômato na linguagem DOT que ao final do processo é retornada como uma *string*.

Conforme mostrado na figura 2.27, primeiro, o autômato recebido como argumento é passado para a função *create_ digraph*, que retorna um objeto da classe *Multidigraph*, do *NetworkX*, que contém informações sobre os estados, eventos e transições do autômato. Então, esse objeto é passado para a função *write_ dot*, do *NetworkX*, que cria o arquivo *DotInterfaceFile.dot* na pasta *working*, cujo caminho está armazenado em *dir_ path[WORKING]*. O código do arquivo criado é lido, processado para remover a *string* *'strict'*, e, em seguida, armazenado numa *string* que é retornada ao final da função *auto2dot*.

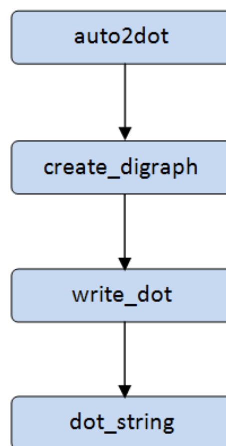


Figura 2.27: Operações da função *auto2dot*.

Exemplo 52

Seja o autômato G definido no DESLab da forma a seguir e cujo diagrama de transição de estados é apresentado na figura 2.28:

```
syms('x1 x2 a')
table = [(x1,'x_1'),(x2,'x_2'),(a,r'\alpha')]
X = [x1,x2]
Sigma = [a]
X0 = [x1]
Xm = []
T =[(x1,a,x2)]
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')
```

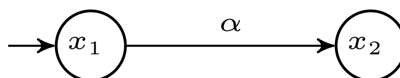


Figura 2.28: Autômato G.

Se a função auto2dot for chamada com o autômato G como argumento, obtém-se a seguinte string como resultado:

```
>>> auto2dot(G)
"digraph {\ns0 [label=x_1, style="state,initial"];\ns1 [label=
    x_2, style="state"];\ns0 -> s1 [key="\\alpha", label="\\
    alpha", style="obs_edge arrow"];\n}\n"
>>> code = auto2dot(G)
>>> print(code)
digraph {
    s0 [label=x_1, style="state,initial"];
    s1 [label=x_2, style="state"];
    s0 -> s1  [key="\alpha", label="\alpha", style="obs_edge
        arrow"];
}
```

Função *create_digraph*

A função *create_digraph* recebe um autômato como argumento e cria um objeto da classe *MultiDiGraph*, do *NetworkX*, contendo todas as informações dos estados, transições e eventos desse autômato.

Todas as informações sobre a construção do autômato são retiradas do seu atributo *Graph* e inseridas no novo objeto *Multidigraph*, o qual, posteriormente, será utilizado na criação do arquivo DOT lido pelo *Graphviz*.

No módulo *NetworkX* os estados do autômato são tratados como *nodes* e as transições são *edges*, cada um desses elementos armazena informações da estrutura do autômato.

Cada *node* recebe um nome distinto para representá-lo. A figura 2.29 ilustra a construção desse nome. Como a numeração dos *nodes* é aleatória, o *node* "$s0$"não irá obrigatoriamente representar o estado inicial do autômato.

Estado do autômato

node do objeto *Multidigraph*

'Estado_1' ⟶ SX

X - Número identificador do *node*, aleatório, mas único para cada estado

Figura 2.29: Estrutura dos nomes.

Cada *node* contém o seu nome e um dicionário. Esse dicionário possui as chaves *label* e *style*, que armazenam, respectivamente, o nome do estado correspondente ao *node* e as informações da sua configuração: *state*, *initial* e/ou *accepting*, que informam que o *node* é um estado, um estado inicial e um estado marcado, respectivamente.

Cada *edge* é construído seguindo o mesmo padrão dos *nodes*, mas, como representam transições, os dois primeiros elementos são os nomes dos dois *nodes* interligados,

label armazena o nome dos eventos e *style* representa o tipo de transição: *unobs_ edge arrow* para eventos não observáveis e *obs_ edge arrow* para eventos observáveis.

Exemplo 53
Considere o autômato G definido no exemplo 52. Chamando a função create_ digraph com G como argumento, podemos observar os valores de nodes e edges:

```
>>> g=create_digraph(G)
>>> g.nodes()
NodeView((‘s0’, ‘s1’))
>>> g.edges()
OutMultiEdgeDataView([(‘s0’, ‘s1’)])
>>> for node in g.nodes(data=True):
        print(node)


(‘s0’, {‘label’: ‘x_1’, ‘"state,initial"’})
(‘s1’, { ‘label’:  ‘x_2’, ‘style’: ‘"state"’})
>>> for edge in g.edges(data=True):
        print(edge)


(‘s0’, ‘s1’, {‘label’: ‘"\\alpha"’, ‘style’: ‘obs_edge arrow’
   })
```

2.3.2 Pasta *graphics.output*

A pasta *graphics.output* é utilizada como local de armazenamento para todos os arquivos que são gerados pelo LaTeX durante a produção do arquivo PDF. Na pasta *graphics.output* são armazenados o arquivo de log, o arquivo PDF e os arquivos AUX, NAV, OUT,SNM e TOC gerados pelo LaTeX.

2.3.3 Pasta *graphics.working*

A pasta *graphics.working* contém o módulo *dotparsing* e o *script dot2tex_ deslab*, que são responsáveis pelo processamento e conversão do código XDOT retornado pelo *software Graphviz*, no código TEX utilizado para gerar as figuras e os arquivos PDF. Além desses módulos, a pasta *graphics.working* também armazena os arquivos DotInterfaceFile.dot e TexOutput.tex, manipuladas durante a execução da função *draw*, mais detalhes sobre esses arquivos podem ser encontrados nas seções que tratam das funções *auto2dot* e *write_ texfile*, respectivamente.

Script dot2tex_ deslab

O *script dot2tex_ deslab* é uma modificação do *script dot2tex*, originalmente desenvolvido por Kjell Magne Fauske [11], feita para ser utilizada no DESLab. O código original possui diversas funcionalidades além das utilizadas pelo DESLab. Nesse trabalho, serão comentadas apenas as partes do código que são utilizadas pelo DESLab.

Como descrito anteriormente, o *script dot2tex_ deslab* é utilizado na função *automaton2tikfig* para criar o código TEX a partir de um arquivo XDOT gerado pelo *Graphviz*. Esse *script* possui uma função *main* que serve como uma interface para as suas demais funções. Essa função recebe os argumentos de configuração, por linha de comando, e o arquivo a ser processado no formato XDOT.

De acordo com a figura 2.30, inicialmente, a função *create_ options_ parser* é utilizada para tratar todos os argumentos passados. Nela, um analisador é declarado usando o módulo *optparser* do *Python*, e todas as opções de parâmetros, bem como os seus valores padrões, são adicionados à sua configuração. Após o tratamento dos dados no analisador, obtém-se o dicionário, contendo todos os argumentos e seus valores, apresentados na tabela 2.13.

No DESLab, todas as figuras geradas são no formato TIKZ, então, o dicionário com os argumentos é passado para a classe *Dot2TikzConv*. Essa classe, na qual estão definidas as funções responsáveis pela construção do código DOT em TIKZ, trata o autômato da mesma forma que o módulo *NetworkX*, ou seja, chamando os estados de *nodes* e as transições de *edges*, e respeitando a nomenclatura criada para o código DOT, vista na função *create_ digraph*. O código DOT a ser convertido é passado como argumento para a função *convert*, que retorna o resultado no formato desejado. A função *convert* não pertence diretamente à classe *Dot2TikzConv*, pertencendo à sua classe base, *DotConvBase*.

Durante a execução de *convert*, a função *parse_ dot_ data* e chamada para processar o código recebido. Ela utiliza o analisador *DotDataParser* definido no módulo *dotparsing*.

Módulo *dotparsing*

O módulo *dotparsing*, desenvolvido por Kjell Magne Fauske, é uma modificação do módulo *dot_ parser*, do pacote *pydot* [12], criado originalmente por Michael Krause e Ero Carrera. Ele foi desenvolvido para trabalhar em conjunto com o *script dot2tex*. Nesse trabalho só serão abordados os segmentos do *dotparsing* usados pelo DESLab.

O *dotparsing* é empregado na análise do código XDOT que será convertido para LaTeX. Para isso, a sua classe *DotDataParser*, que atua como um analisador, é utilizada no processamento do código. O exemplo 54 a seguir ilustra como é a

Tabela 2.13: Dicionário do analisador.

Parâmetro	**Valor**
alignstr	None
autosize	False
cache	False
codeonly	True
crop	False
debug	False
docpreamble	None
duplicate	False
edgeoptions	None
encoding	utf8
figonly	False
figpostamble	None
figpreamble	None
format	tikz
graphstyle	None
gvcols	False
margin	0pt
nodeoptions	None
nominsize	False
outputfile	
pgf118	False
printversion	False
prog	dot
runtests	False
straightedges	False
styleonly	False
switchdraworder	False
templatefile	None
texmode	math
texpreproc	False
tikzedgelabels	False
usepdflatex	False
valignmode	center

estrutura do código XDOT antes e depois do processamento feito com o *dotparsing*.

Exemplo 54

Considere o autômato G definido no exemplo 52. O código XDOT para esse autômato, gerado com o Graphviz durante a execução da função automaton2tikfig, é apresentado a seguir:

```
digraph {
graph [rankdir=LR, nodesep="0.25", ranksep="0.25"];
node [label="\N"];
graph [bb="0,0,156,36",
_draw_="c 9 -#ffffffff C 9 -#ffffffff P 4 0 -1 0 36 157 36 157
    -1 ",
    xdotversion="1.2"];
```

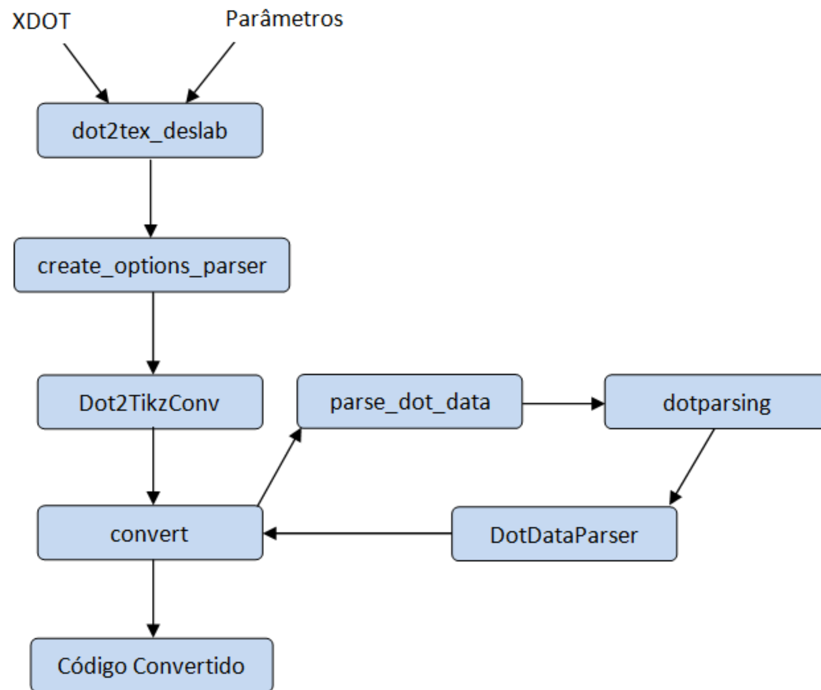


Figura 2.30: Processos do *script dot2tex_ deslab*.

```
s1 [label=x_1, style="state,initial", pos="27,18", width="0.75
   ", height="0.5", _draw_="S 5 -state S 7 -initial c 9
   -#000000ff e 27 18 27 18 ", _ldraw_="F 14.000000 11 -Times-
   Roman c 9 -#000000ff T 27 12 0 22 3 -x_1 "];
s0 [label=x_2, style=state, pos="129,18", width="0.75", height
   ="0.5", _draw_="S 5 -state c 9 -#000000ff e 129 18 27 18 ",
    _ldraw_="F 14.000000 11 -Times-Roman c 9 -#000000ff T 129
   12 0 22 3 -x_2 "];
s1 -> s0 [key="\alpha", label="\alpha", style="obs_edge arrow"
   , pos="e,101.77,18 54.013,18 65.496,18 79.192,18 91.661,18"
   , lp="78,25.5", _draw_="S 14 -obs_edge arrow c 9 -#000000ff
    B 4 54 18 65 18 79 18 92 18 ", _hdraw_="S 5 -solid c 9
   -#000000ff C 9 -#000000ff P 3 92 22 102 18 92 15 ", _ldraw_
   ="F 14.000000 11 -Times-Roman c 9 -#000000ff T 78 19 0 30 5
    -alpha "];
}
```

Após o processamento do código XDOT, a seguinte string contendo o código formatado para LaTeXé retornada:

```
%%
\node (s1) at (2.70pt,1.80pt) [draw,ellipse,state,initial] {
   $x_1$};
\node (s0) at (12.90pt,1.80pt) [draw,ellipse,state] {$x_2$};
\draw [,obs_edge arrow] (s1) ..controls (6.55pt,1.80pt) and
   (7.92pt,1.80pt)  .. (s0);
\definecolor{strokecol}{rgb}{0.0,0.0,0.0};
```

```
\pgfsetstrokecolor{strokecol}
\draw (7.80pt,2.55pt) node {$\alpha$};
%
```

2.4 Pasta *readwrite*

A pasta *readwrite* foi construída para armazenar os módulos responsáveis por salvar em arquivos as informações dos autômatos. Na atual versão, ele contém apenas o módulo *inputoutput*.

2.4.1 Módulo *inputoutput*

O módulo *inputoutput* faz uso do módulo *_pickle* do *Python* 3.6 para carregar e retornar informações de arquivos. Ele possui duas funções, uma responsável por salvar informações e outra responsável pelo carregamento de informações salvas, as quais são descritas nas subseções a seguir.

Função *save*

A função *save* pode receber até quatro argumentos distintos, descritos na tabela 2.14, mas apenas o primeiro é obrigatório pois os outros já possuem valores predefinidos.

Caso o argumento *tmx* seja *False*, será criado um arquivo com a extensão do DESLab (*.des*) utilizando o nome armazenado em *filename* e nele serão salvas todas as informações do autômato passado.

Quando *tmx* possuir valor verdadeiro (*True*), a função criará uma tabela relacionando os estados e as transições do autômato em um arquivo de texto.

Os arquivos criados sempre serão salvos no caminho especificado em *path* ou, caso esse valor não seja passado, na pasta onde o DESLab está sendo executado.

Exemplo 55 *Uma vez criado o autômato "G", a função para salvar seus dados pode ser chamada das seguintes formas:*

```
>>> save(G)
>>> save(G,"name","folder")
>>> save(G,tmx = True)
>>> save(G,"name","folder",True)
```

Função *load*

A função *load* recebe como argumentos o nome do arquivo e o caminho dele. Através desses valores, o arquivo especificado é localizado e seu conteúdo é carregado e retornado no formato de um objeto da classe *fsa*. Essa função funciona apenas com arquivos do DESLab (*.des*).

Tabela 2.14: Parâmetros da função *save*.

Variável	Significado	Valor Predefinido
G	Autômato	
filename	Nome para o arquivo	Nome do autômato
path	Caminho para o arquivo criado	Pasta atual
tmx	Matrix de transições	Falso

Exemplo 56

A função load pode ser chamada da seguinte forma:

```
from deslab import *
G = load("arquivo.des","caminho")
G = load("arquivo.des")
```

Capítulo 3

Toolbox

A pasta *toolbox* é um novo pacote do DESLab, no qual estão contidos módulos que armazenam funções voltadas a aplicações específicas. São propostos dois módulos para essa pasta: *diagnosis* e *supervisory*. O módulo *diagnosis* possui funções voltadas à solução de problemas de diagnose de falhas de sistemas a eventos discretos, e o módulo *supervisory* contém funções aplicáveis ao problema de controle supervisório de sistema a eventos discretos. As funções presentes nesses módulos são descritas nas seções a seguir.

3.1 Módulo *diagnosis*

3.1.1 Função *diagnoser*

A função *diagnoser* constrói o autômato diagnosticador G_d [13] de uma planta modelada por um autômato G. Para isso, ela deve receber três argumentos: O autômato G, uma *string, failevent*, que representará o evento não observável de falha σ_f, e uma *string*, *ret*, que serve de instrução para qual autômato será retornado (*'GD'* ou *'GL'*).

Inicialmente, cria-se o autômato rotulador A_l apresentado na figura 3.1. Em

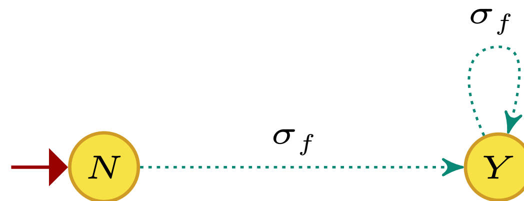


Figura 3.1: Autômato rotulador A_l.

seguida é calculado o autômato G_l pela composição paralela entre G e Al, utilizando o operador $//$. Se *ret*=*'GL'*, o estilo do atributo *graphic* será mudado para *'observer'* e o autômato G_l será retornado pela função *diagnoser*. Vale ressaltar que, por

padrão, a *string ret* sempre terá valor igual a *'GD'*, quando o usuário desejar que a função *diagnoser* retorne o autômato G_l, o valor *'GL'* deverá ser passado para *ret*.

Quando *ret='GD'*, a função *diagnoser* retornará o autômato G_d, que é o observador de G_l em relação ao conjunto de eventos observáveis armazenado no atributo *Sigobs* de G. Esse observador é calculado utilizando a função *observer*. O exemplo abaixo mostra como utilizar a função *diagnoser*.

Exemplo 57

Seja G o autômato definido pelo código abaixo.

```
syms('1 2 3 4 5 6 a b c f u')
X = [1,2,3,4,5,6]
Sigma = [a,b,c,f,u]
X0 = [1]
Xm = []
T =[(1,c,2), (2,a,3),(3,b,2),(2,f,4),(4,a,5),(5,b,4),(5,a,5)
   ,(5,u,6),(6,a,6)]
G = fsa(X,Sigma,T,X0,Xm,name='$G$',Sigobs=[a,b,c])
```

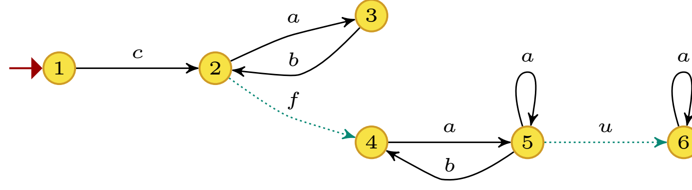
Calculamos os autômatos Gd e Gl da seguinte forma:

```
>>> draw(G,'figurecolor')
generating latex code of automaton
>>> Gd = diagnoser(G,f)
>>> draw(Gd,'figurecolor')
generating latex code of automaton
>>> Gl = diagnoser(G,f,'GL')
>>> draw(Gl,'figurecolor')
generating latex code of automaton
```

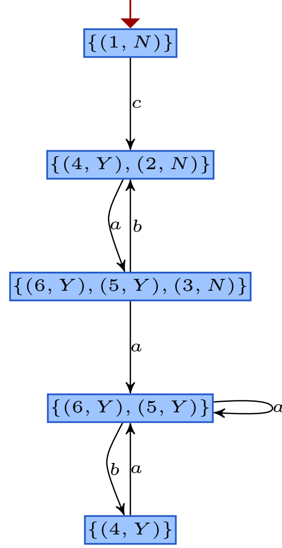
Os diagramas de transição de estados gerados são ilustrados na figura 3.2

3.1.2 Função *simplify*

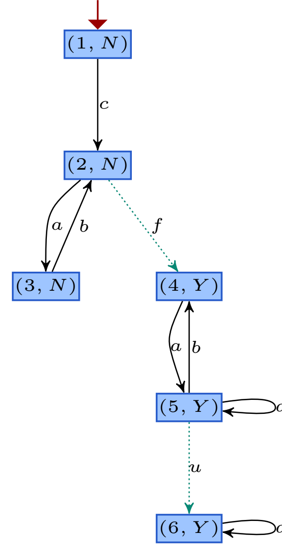
A função *simplify* foi criada para simplificar os estados a fim de facilitar a visualização e tratamento das variáveis. Como as operações do módulo *diagnosis* podem realizar muitas modificações nos autômatos, isso pode gerar nomes muito grandes ou complexos para os estados, como um *set* de tuplas. A função percorre os estados dos autômatos transformando seus nomes em *strings* e eliminando caracteres separadores como vírgulas e parênteses. Os novos nomes são armazenados na lista *mapping* que é passada para o método *renamestates* do autômato que é retornado ao final. A figura 3.3 ilustra o resultado da simplificação dos nomes dos estados de um autômato.



(a) Autômato G.



(b) Autômato *Gd*.



(c) Autômato *Gl*.

Figura 3.2: Autômatos do exemplo 57.

3.1.3 Função *Gscc*

A função *Gscc* calcula o diagnosticador G_{scc} [14] executando a composição paralela entre os autômatos *Gd* e *Gl*. Para realizar essa operação a função deve receber o autômato da planta, uma *string* que representa o evento de falha, e uma lista com os eventos observáveis do autômato que, caso não seja passada, será obtida por meio do atributo *Sigobs* do autômato da planta.

A função *Gscc* também pode ser aplicada em problemas de diagnose descentralizada, na qual, ao invés de um só diagnosticador, tem-se um conjunto de N diagnosticadores locais. Nesse caso, no lugar de uma lista de eventos observáveis, deve-se passar uma lista contendo as listas de eventos observáveis de cada diagnosticador local. O autômato retornado pela função *Gscc* será igual a $G_{scc}^N = G_{d_1} \| G_{d_2} \| \ldots \| G_{d_N} \| G_l$, em que G_{d_i} é o i-ésimo diagnosticador local.

Para gerar o autômato resultante, inicialmente, obtém-se o autômato G_l simplificado por meio das funções *diagnoser* e *simplify*. Para cada lista de eventos observáveis passada, calcula-se o autômato G_{d_i}. No caso da diagnose centralizada, define-se G_d como $G_d = G_{d_1}$, enquanto que, para a diagnose descentralizada, $G_d = G_{d_1} \| G_{d_2} \| \ldots \| G_{d_N}$. Então, calcula-se o autômato G_{scc} pela composição para-

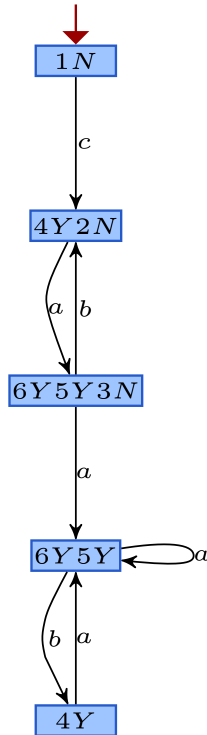


Figura 3.3: Autômato obtido executando-se *simplify(Gd)*, para o autômato Gd apresentado na figura 3.2b.

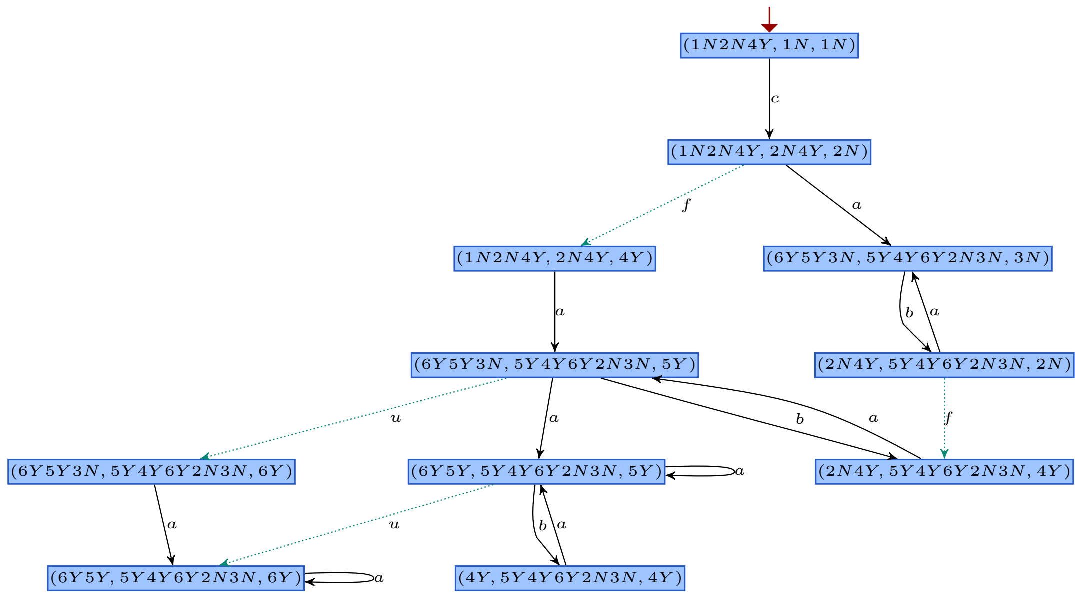
lela entre G_d e G_l. Em seguida, certifica-se que o evento de falha não faz parte do atributo *Sigobs* de G_{scc}, modifica-se o tipo do atributo *graphic* para *'observer'* e, por fim, G_{scc} é retornado. O exemplo abaixo ilustra como utilizar a função *Gscc*.

Exemplo 58

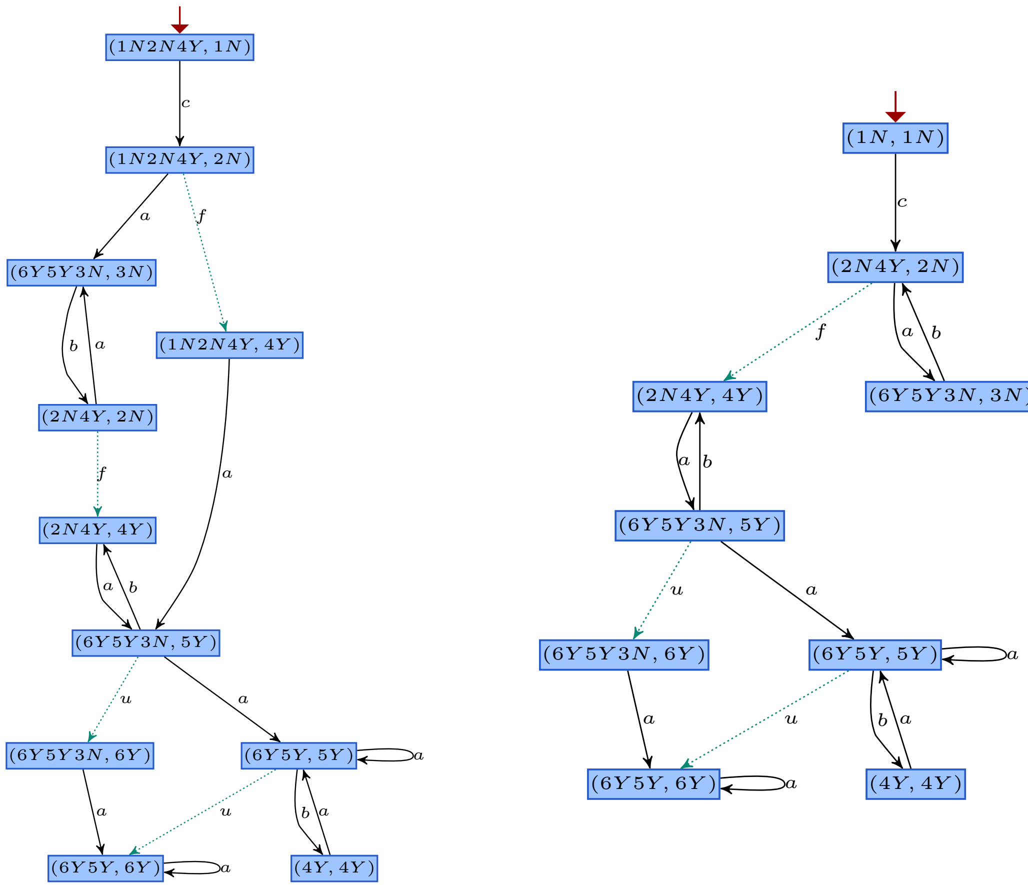
Seja G o autômato definido no exemplo 57. Calculamos G_{scc} para os casos com uma ou mais listas de eventos como mostrado no código abaixo.

```
>>> Gscc1 = Gscc(G,'f',[a,b,c])
>>> Gscc2 = Gscc(G,'f',[a,b])
>>> Gscc3 = Gscc(G,'f',[[a,b],[a,c]])
>>> draw(Gscc1,Gscc2,Gscc3,'figurecolor')
generating latex code of automaton
generating latex code of automaton
generating latex code of automaton
```

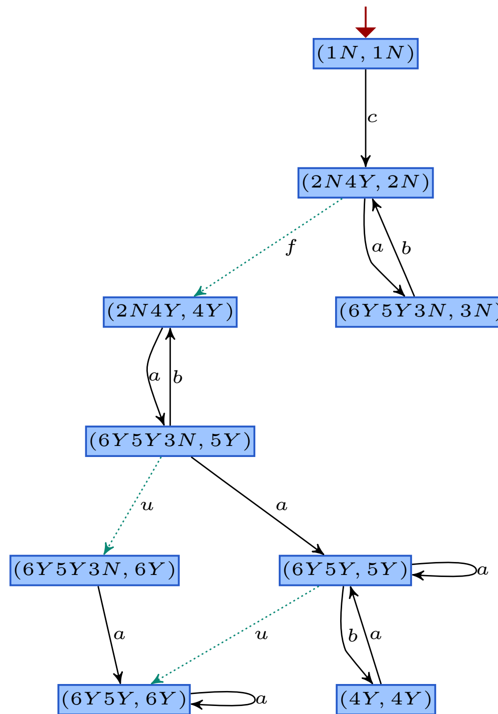
Deve-se observar, por meio da figura 3.4, que o uso da função simplify só foi feito para agrupar os nomes dos estados em G_d e G_l. Com isso, as informações sobre os estados na composição paralela entre G_{d_i} e G_l são preservadas.



(a) Autômato *Gscc3*.



(b) Autômato *Gscc2*.



(c) Autômato *Gscc1*.

Figura 3.4: Autômatos do exemplo 58.

3.1.4 Função *Gv*

A função *Gv* constrói o autômato verificador [15] ao receber como argumentos o autômato da planta, a *string* do evento de falha e o conjunto de eventos observáveis que, assim como na função *Gscc*, pode ser uma lista contendo as listas de eventos de cada diagnosticador local.

Para auxiliar nas operações, a função interna *Ri* foi definida. Ela recebe duas listas de eventos, *sigma* e *sigmaOi*, e um caractere, *'i'*, que será usado para renomear os eventos, que pelo método usado será um número inteiro. Cada evento $e \in (sigma - sigmaOi)$ terá o seu nome modificado para *eRi* e seu nome LaTeX será $e_\{R_i\}$. A lista dos nomes dos eventos e a lista dos nomes LaTeX são retornadas pela função *Ri*.

Primeiro, verifica-se o conjunto de eventos observáveis passado, caso seja vazio, o atributo *Sigobs* do autômato é usado. Em seguida os passos apresentados em [15] são executados, como descrito a seguir:

1. Cria-se a lista *SIGMAn* = *G.Sigma* - $\{\sigma_f\}$ (em que σ_f é o evento de falha);

2. Cria-se um autômato *An* com um único estado N, e com um auto-laço em N para cada evento em *SIGMAn*;

3. o autômato $Gn = G\&An$ é criado, usando o operador da função *product* e seu conjunto de eventos é modificado para *SIGMAn*;

4. O autômato *Gl* é obtido por meio da função *diagnoser(G,*σ_f*,'GL')*. Em seguida, seus estados que contém o rótulo Y são marcados;

5. Obtém-se Gf = *simplify*(*coac*(Gl));

6. Para cada conjunto de eventos observáveis passado para a função *Gv*, utiliza-se a função interna *Ri* para renomear os eventos que não sejam observáveis, gerando os autômatos *Gni*, $i = 1, 2, ..., N$;

7. O autômato $G_v = Gn1//...Gni//Gf$ é calculado usando o operador $//$;

8. Por fim, o estilo do atributo *graphic* de *Gv* é modificado para *'observer'* e ele é retornado pela função *Gv*.

O exemplo abaixo ilustra como gerar o autômato verificador *Gv* para um dado autômato.

Exemplo 59

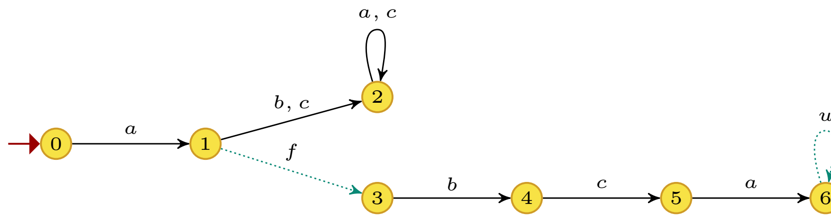
Seja G o autômato definido pelo código abaixo.

```
syms('0 1 2 3 4 5 6 a b c f u')
X = [0,1,2,3,4,5,6]
Sigma = [a,b,c,f,u]
X0 = [0]
Xm = []
T = [(0,a,1), (1,c,2), (1,b,2), (2,a,2), (2,c,2), (1,f,3), (3,
    b,4), (4,c,5), (5,a,6), (6,u,6)]
G = fsa(X,Sigma,T,X0,Xm,name='$G$',Sigobs=[a,b,c])
```

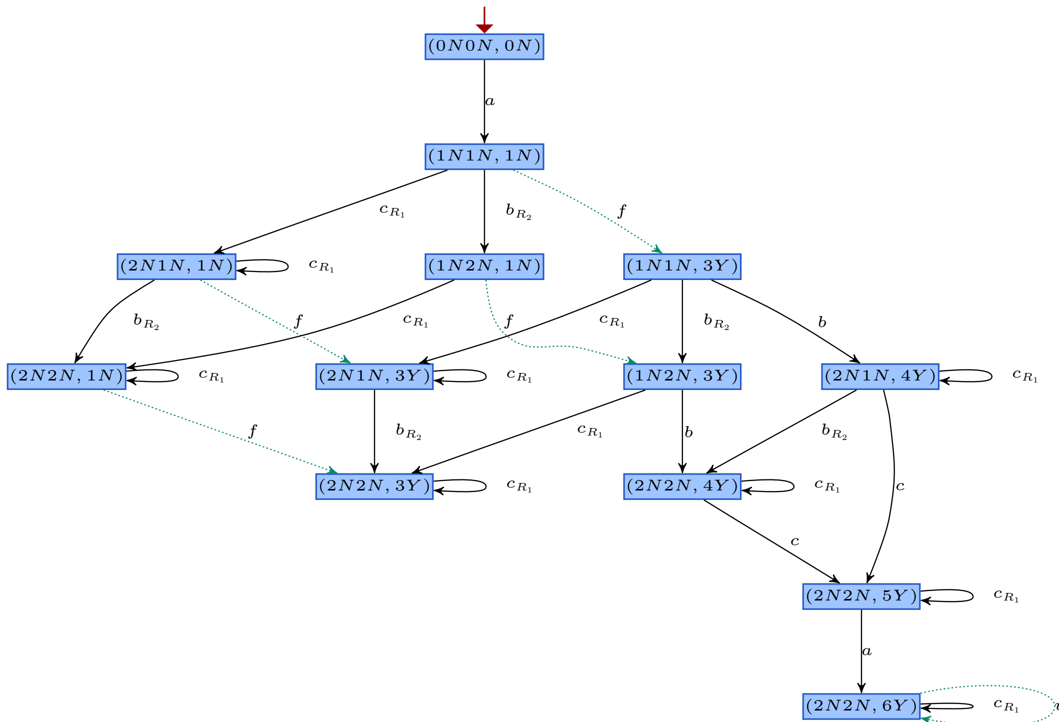
Podemos gerar o autômato verificador da seguinte forma:

```
>>> G_v = Gv(G,'f',[[a,b],[a,c]])
>>> draw(G,G_v,'figurecolor')
generating latex code of automaton
```

Os diagramas de transição de estados dos autômatos gerados nesse exemplo são apresentados na figura 3.5.



(a) Autômato G.



(b) Autômato G_v.

Figura 3.5: Autômatos do exemplo 59.

3.1.5 Função *is_ diagnosable*

A função *is_ diagnosable* analisa a (co)diagnosticabilidade de uma planta a partir do método escolhido, ou seja, usando o diagnosticador *Gscc*, ou o verificador *Gv*. A função deve receber como argumentos, o autômato que modela a planta, *G*, uma *string* do evento de falha, *failevent*, a lista de eventos observáveis, que também pode ser uma lista contendo listas para o caso descentralizado, e uma *string* que especifica o método a ser utilizado (*'Gscc'* ou *'Gv'*).

Se o método escolhido for *'Gscc'*, a função *Gscc* é utilizada para gerá-lo. Para auxiliar na verificação, a função interna *N_ Y* é definida. Ela recebe uma lista de estados e, para cada estado, verifica em qual caso dentre aqueles apresentados na tabela 3.1 ele se enquadra. A função *N_ Y* cria uma lista de zeros para cada estado passado e, caso algum dos estados seja do tipo *(YN,Y)* o valor será modificado para 1.

Tabela 3.1: Estados de G_{scc}.

G_{scc}	G_d	G_l
(Y,Y)	Certo	Falha ocorreu: certo
(N,N)	Normal	Falha não ocorreu: normal
(YN,Y)	Incerto	Falha ocorreu: certo
(YN,N)	Incerto	Falha não ocorreu: normal

Os componentes fortemente conexos não triviais do autômato *Gscc* são identificados por meio da função *strconncomps* e da função *node_ with_ selfloops* do NetworkX, respectivamente. A lista de estados obtida é passada para a função interna *N_ Y* e, caso nenhum estado *(YN,Y)* seja identificado a função *is_ diagnosable* retornará *False*, caso contrário, será retornado *True*.

Se o método escolhido for *'Gv'*, a mesma verificação dos componentes fortemente conexos e componentes com auto-laço, feita para o método *'Gscc'*, será executada no autômato verificador *Gv*, obtido com a função *Gv*.

Será retornado *False* se existir um componente fortemente conexo não trivial, cuja última coordenada de seus estados é rotulada com *Y* e existe uma transição entre dois dos seus estados rotulada por um evento da planta. Caso contrário, será retornado o valor *True*. O exemplo 60 mostra como utilizar a função *is_ diagnosable*.

Exemplo 60

Seja G1 o autômato definido no exemplo 57 e seja G2 o autômato definido no exemplo 59. Usando a função is_ diagnosable, checamos a diagnosticabilidade deles usando os autômatos Gscc1,da figura 3.4, e G_ v, da figura 3.5, respectivamente.

```
>>> is_diagnosable(G1,'f',[a,b,c],'Gscc')

False
```

```
>>> is_diagnosable(G2,'f',[[a,b],[a,c]],'Gv')


False
```

3.2 Módulo *supervisory*

3.2.1 Funções supCont e is_controllable

A função *supCont* recebe dois autômatos, H e G, e, a partir deles, calcula e retorna um autômato cuja linguagem marcada será a sub-linguagem controlável suprema de $L_m(H)$ em relação a $L(G)$ e o conjunto de eventos não controláveis de G. Vale ressaltar que o autômato H deve ser não-bloqueante. Para realizar as operações, primeiro, obtém-se o autômato Gm, que é uma cópia de G com todos os estados marcados. Em seguida, calcula-se o autômato $Hi = H \times Gm$, e os conjuntos *Sigcon* e *Sigobs* de H são igualados aos de G, usando a função *setpar*, que também é usada para renomear *Hi*. Para cada estado *(x,xg)* de *Hi*, é checado, usando o método *Gamma* dos autômatos, se a interseção entre o conjunto de eventos ativos no estado *xg* de *Gm* e o conjunto de eventos não controláveis está contida no conjunto de eventos ativos do estado *(x,xg)*. Cada estado de Hi que não satisfizer essa condição será removido de *Hi* e, em seguida, calcula-se *Hi* = *trim(Hi)*. Essa condição é checada novamente, para o novo Hi, e se repete essa operação até que todos os estados de Hi satisfação a condição anterior.

A função *is_controllable* também recebe dois autômatos, H e G, e verifica se $L_m(H)$ é controlável em relação a $L(G)$ e o conjunto de eventos não controláveis de G. Vale ressaltar que, semelhante ao caso da função *supCont*, o autômato H deve ser não-bloqueante. O código dessa função é bastante semelhante ao da função anterior. Calcula-se o autômato Hi, da mesma forma que na função *supCont*, e verifica-se se, para todo estado *(x,xg)* de *Hi*, a interseção entre o conjunto de eventos ativos no estado *xg* de *Gm* e o conjunto de eventos não controláveis está contida no conjunto de eventos ativos do estado *(x,xg)*. Caso essa condição seja satisfeita, retorna-se *True*, caso contrário, retorna-se *False*.

Exemplo 61

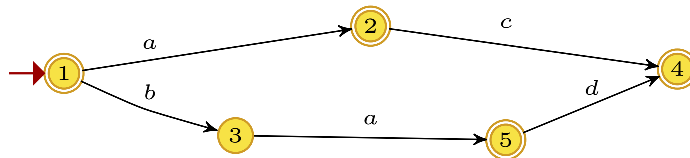
Considere os autômatos G e H definidos no DESLab pelo código abaixo.

```
syms('a b c d')
X     = [1,2,3,4,5]
Sig   = [a,b,c,d]
Trans = [(1,a,2),(1,b,3),(2,c,4),(3,a,5),(5,d,4)]
X0    = [1]
Xm    = [1,2,4,5]
G = fsa(X,Sig,Trans,X0,Xm,name='$G$',Sigcon=[a,b,c])
```

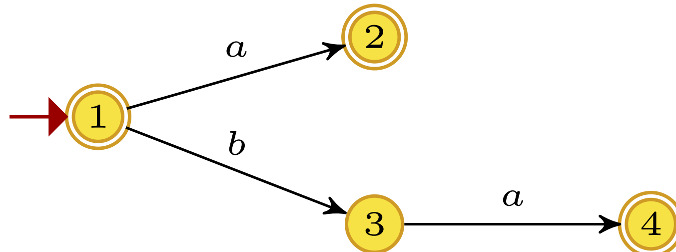
```
X2      = [1,2,3,4]
Sig2    = [a,b]
Trans2  = [(1,a,2),(1,b,3),(3,a,4)]
X02     = [1]
Xm2     = [1,2,4]
H   =fsa(X2,Sig2,Trans2,X02,Xm2,name='$H$')
```

Checa-se os autômatos com a função is_controllable, e, em seguida, utiliza-se a função supCont para gerar o autômato apresentado na figura 3.6.

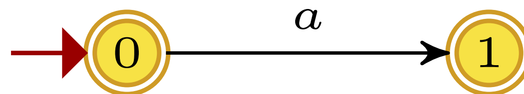
```
>>> is_cont(H,G)
False
>>> SC = supCont(H,G)
>>> draw(SC,'figurecolor')
generating latex code of automaton
```



(a) Autômato G com $Sigcon = [a, b, c]$.



(b) Autômato H.



(c) Autômato *supCont(H,G)*.

Figura 3.6: Autômatos do exemplo 61.

Capítulo 4

Conclusão e Trabalhos Futuros

Nesse trabalho, foi desenvolvida uma nova versão do DESLab atualizada para a versão 3.6 do Python. Esse processo de atualização envolveu várias adequações das funções do DESLab às novas versões dos módulos da biblioteca Python usados pelo DESLab. O módulo NetworkX foi um dos que sofreu mudanças significativas desde o desenvolvimento da versão anterior do DESLab, e, com isso, demandou mais adequações nos códigos das funções.

Além das alterações ocasionadas pelas atualizações das partes fornecidas por terceiros, foram corrigidas diversas redundâncias e problemas nos códigos das funções, identificados durante o processo de atualização ou relatados por usuários da versão antiga do DESLab.

Foi desenvolvido um instalador para essa nova versão, acompanhado de um guia de instalação, no qual se detalha cada etapa desse processo. Além disso, o pacote de instalação, a ser disponibilizado, contém todos os componentes (softwares e módulos do Python) necessários para o correto funcionamento do DESLab, visando, com isso, evitar problemas de compatibilidade entre o DESLab e as versões instaladas desses componentes.

Dois novos módulos foram acrescentados à nova versão do DESLab. Um módulo voltado para problemas de diagnose de SEDs e outro destinado a problemas de controle supervisório de SEDs.

Por fim, elaborou-se um material bibliográfico, apresentado no capítulo 2, que pode se tornar uma ferramenta na orientação de novos desenvolvedores do DESLab.

O próximo passo no desenvolvimento do DESLab seria a evolução do módulo de controle, criando novas funções para problemas de controle supervisório sob observação parcial.

Referências Bibliográficas

[1] PRABHU, P., KIM, H., OH, T., et al. "A survey of the practice of computational science". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Seattle, WA, USA, Nov 2011.

[2] FOUNDATION, P. S. "Python". . Disponível em: <`https://www.python.org/`>. Acessado em 25 de agosto de 2018.

[3] CLAVIJO, L. B., BASILIO, J. C., CARVALHO, L. K. "DESLAB: A scientific computing program for analysis and synthesis of discrete-event systems", *IFAC Proceedings Volumes*, v. 45, n. 29, pp. 349–355, 2012.

[4] HAGBERG, A., SCHULT, D., SWART, P. "NetworkX". Disponível em: <`https://networkx.github.io/`>. Acessado em 25 de agosto de 2018.

[5] ELLSON, J., GANSNER, E., HU, Y., et al. "Graphviz distribution". Disponível em: <`https://www.graphviz.org/`>. Acessado em 25 de agosto de 2018.

[6] RAHTZ, S., KAKUTO, A., BERRY, K., et al. "TexLive distribution". Disponível em: <`https://www.tug.org/texlive/`>. Acessado em 25 de agosto de 2018.

[7] FOUNDATION, P. S. "Python 2.7". . Disponível em: <`https://legacy.python.org/dev/peps/pep-0373/`>. Acessado em 25 de agosto de 2018.

[8] CASSANDRAS, C. G., LAFORTUNE, S. *Introduction to Discrete Event Systems*. 2nd ed. New York, Springer, 2008.

[9] LINZ, P. *An introduction to formal languages and automata*. 3rd ed. Nova Yorque, NY, USA, Jones and Bartlett Publishers, 2001.

[10] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., et al. *Introduction to algorithms*. MIT press, 2009.

[11] FAUSKE, K. M. "dot2tex - A Graphviz to LaTeX converter". Disponível em: <http://dot2tex.readthedocs.io/en/latest/index.html>. Acessado em 28 de junho de 2018.

[12] CARRERA, E. "pydot - Python interface to Graphviz's Dot language". Disponível em: <https://github.com/eventbrite/pydot>. Acessado em 28 de junho de 2018.

[13] SAMPATH, M., SENGUPTA, R., LAFORTUNE, S., et al. "Diagnosability of discrete-event systems", *IEEE Transactions on Automatic Control*, v. 40, n. 9, pp. 1555–1575, Sept 1995.

[14] VIANA, G. S., BASILIO, J. C., MOREIRA, M. V. "Computation of the maximum time for failure diagnosis of discrete-event systems". In: *2015 American Control Conference (ACC)*, pp. 396–401, July 2015.

[15] MOREIRA, M. V., JESUS, T. C., BASILIO, J. C. "Polynomial Time Verification of Decentralized Diagnosability of Discrete Event Systems", *IEEE Transactions on Automatic Control*, v. 56, n. 7, pp. 1679–1684, 2011.