ALGORITMOS EM TEMPO POLINOMIAL PARA VERIFICAÇÃO DA OBSERVABILIDADE E NORMALIDADE DE LINGUAGENS REGULARES

Bernardo Pestana Bouzan

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: João Carlos dos Santos Basilio

Rio de Janeiro
Agosto de 2013

ALGORITMOS EM TEMPO POLINOMIAL PARA VERIFICAÇÃO DA OBSERVABILIDADE E NORMALIDADE DE LINGUAGENS REGULARES

Bernardo Pestana Bouzan

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

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

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

Prof. Antonio Eduardo Carrilho da Cunha, Dr.Eng.

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 2013

Bouzan, Bernardo Pestana

Algoritmos em tempo polinomial para verificação da observabilidade e normalidade de linguagens regulares/Bernardo Pestana Bouzan. – Rio de Janeiro: UFRJ/COPPE, 2013.

XVIII, 157 p.: il.; 29, 7cm.

Orientador: João Carlos dos Santos Basilio

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2013.

Referências Bibliográficas: p. 139 – 141.

1. Controle supervisório. 2. Sistemas a eventos discretos. 3. Observabilidade. 4. Normalidade. 5. Autômatos. I. Basilio, João Carlos dos Santos. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

À minha esposa Elly, aos meus pais Reinaldo e Lidia, e ao meu padrasto Carlos.

Agradecimentos

Dentre os que contribuíram diretamente para a criação deste trabalho, agradeço:

- Ao professor e orientador João Carlos dos Santos Basilio, pela dedicação e pela paciência com as limitações impostas pela minha atividade profissional;

- Ao doutorando Leonardo Bermeo Clavijo, pelo direcionamento e suporte oferecido ao longo deste trabalho (inclusive nos fins de semana);

- Aos professores Marcos Vicente de Brito Moreira e Lilian Kawakami Carvalho, pela atenciosa contribuição no desenvolvimento dos algoritmos.

Com respeito àqueles que não tiveram participação direta neste trabalho, mas que foram responsáveis indiretamente pela sua existência, agradeço:

- À Elly, pelo companheirismo em todos os aspectos de minha vida, sem o apoio da qual nenhuma página deste trabalho seria possível;

- Aos meus pais e minha familia, pela compreensão de minha ausência e pelo apoio dado ao longo desses anos;

- À Chemtech e à FMC Technologies, por investirem em meu desenvolvimento;

- Aos companheiros de disciplina do mestrado, Leonardo e Lucas, pela amizade e companheirismo tão importantes na conciliação da vida profissional com a acadêmica.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ALGORITMOS EM TEMPO POLINOMIAL PARA VERIFICAÇÃO DA OBSERVABILIDADE E NORMALIDADE DE LINGUAGENS REGULARES

Bernardo Pestana Bouzan

Agosto/2013

Orientador: João Carlos dos Santos Basilio

Programa: Engenharia Elétrica

São recorrentes na literatura de controle supervisório desenvolvimentos de algoritmos mais eficientes para a análise das diversas propriedades de sistemas a eventos discretos. Isso ocorre porque modelos de SEDs podem facilmente resultar em representações com número elevado de estados e transições, o que pode gerar problemas computacionais. Com este objetivo, apresenta-se, nesta dissertação, novos algoritmos capazes de verificar as propriedades de observabilidade, normalidade e coobservabilidade de linguagens regulares. Esses algoritmos possuem complexidade computacional igual ou menor do que os algoritmos mais eficientes existentes na literatura.

Outra contribuição deste trabalho é a extensão das funcionalidades do programa DESLAB, com o desenvolvimento de uma biblioteca dedicada à teoria de controle supervisório, tendo sido desenvolvidas as principais funções utilizadas na síntese e análise de supervisores.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

POLYNOMIAL TIME ALGORITHMS FOR THE VERIFICATION OF OBSERVABILITY AND NORMALITY OF REGULAR LANGUAGES

Bernardo Pestana Bouzan

August/2013

Advisor: João Carlos dos Santos Basilio

Department: Electrical Engineering

Several works in the supervisory control literature address the synthesis of new algorithms that aim at reducing computational complexity in the analysis of discrete event systems properties. The reason for this is that system models can easily achieve a huge number of states and transitions, which can lead to computational issues. With this objective in mind, this work presents new algorithms capable of verifying the observability, normality and coobservability properties of regular languages. These algorithms have equal or lower computational complexity than the most efficient ones existing in the literature.

Another contribution of this work is the extension of DESLAB program functionalities, by developing a library dedicated to supervisory control theory. This library implements the most commonly used functions for the synthesis and analysis of supervisors.

Sumário

Lista de Figuras xi

Lista de Tabelas xiv

Lista de Símbolos xv

Lista de Abreviaturas xviii

1 Introdução 1

2 Controle supervisório de Sistemas a Eventos Discretos 4

2.1 Sistemas a Eventos Discretos 4

2.1.1 Conceitos Básicos 4

2.1.2 Linguagem de um SED 9

2.1.3 Autômatos 13

2.2 Controle Supervisório 27

2.2.1 SED controlado considerando observação total 28

2.2.2 Controle de SEDs com observação parcial 33

2.2.3 Controle Supervisório Descentralizado 38

2.3 Complexidade Computacional 42

3 Algoritmos em tempo polinomial para verificação da observabilidade, normalidade e coobservabilidade 44

3.1 Revisão Bibliográfica 44

3.1.1 Observabilidade 45

3.1.2 Normalidade 51

3.2 Verificação de Observabilidade 52

3.2.1 Algoritmo para Verificação 53

3.2.2 Complexidade Computacional 59

3.3 Verificação da Normalidade 60

3.3.1 Algoritmo para Verificação 60

3.3.2 Complexidade Computacional 62

3.4 Exemplos ilustrativos 63
3.4.1 Exemplo 1 63
3.4.2 Exemplo 2 65
3.4.3 Exemplo 3 67
3.4.4 Exemplo 4 70
3.5 Verificação de Coobservabilidade 73
3.5.1 Revisão Bibliográfica 73
3.5.2 Algoritmo Proposto 79
3.5.3 Exemplos ilustrativos 89
3.6 Considerações Finais 95

4 Uma biblioteca DESLAB para Controle Supervisório 98
4.1 DESLAB 98
4.2 Uma toolbox para Controle Supervisório 99
4.2.1 IsControllable 100
4.2.2 SupControllable 102
4.2.3 SupControllablePfClosed 103
4.2.4 SupControllableGeneral 105
4.2.5 ConDat 106
4.2.6 IsObservable (ObsVerifier) 109
4.2.7 IsObservable_Observer (ObsVerifier_Observer) 110
4.2.8 IsObservable_Tsitsiklis (ObsVerifier_Tsitsiklis) 111
4.2.9 IsObservable_Wang (ObsVerifier_Wang) 113
4.2.10 IsObservable_BB (ObsVerifier_BB) 115
4.2.11 IsCoobservable (CoobsVerifier) 116
4.2.12 IsCoobservable_Wang (CoobsVerifier_Wang) 117
4.2.13 IsCoobservable_BB (CoobsVerifier_BB) 121
4.2.14 IsNormal (NormalityVerifier) 123
4.2.15 IsNormal_Observer (NormalityVerifier_Observer) 124
4.2.16 IsNormal_BB (NormalityVerifier_BB) 125
4.2.17 SupNormal 126
4.2.18 SupNormalPfClosed 127
4.2.19 SupNormalGeneral 128
4.2.20 SupControllableNormal 130
4.3 Exemplo de aplicação da biblioteca 132

5 Conclusões e Trabalhos Futuros 137

Referências Bibliográficas 139

A Código fonte da biblioteca DESLAB para Controle Supervisório 142
A.1 IsControllable 142
A.2 SupControllable 142
A.3 SupControllablePfClosed 143
A.4 SupControllableGeneral 143
A.5 ConDat 143
A.6 IsObservable 144
A.7 ObsVerifier 144
A.8 IsObservable_Observer 144
A.9 ObsVerifier_Observer 145
A.10 IsObservable_Tsitsiklis 145
A.11 ObsVerifier_Tsitsiklis 145
A.12 IsObservable_Wang 147
A.13 ObsVerifier_Wang 147
A.14 ObsToDiag 147
A.15 IsObservable_BB 148
A.16 ObsVerifier_BB 149
A.17 IsCoobservable 149
A.18 CoobsVerifier 150
A.19 IsCoobservable_Wang 150
A.20 CoobsVerifier_Wang 150
A.21 CoobsToDiag 150
A.22 IsCoobservable_BB 152
A.23 CoobsVerifier_BB 153
A.24 IsNormal 154
A.25 NormalityVerifier 154
A.26 IsNormal_Observer 155
A.27 NormalityVerifier_Observer 155
A.28 IsNormal_BB 155
A.29 NormalityVerifier_BB 155
A.30 SupNormal 156
A.31 SupNormalPfClosed 156
A.32 SupNormalGeneral 156
A.33 SupControllableNormal 157

Lista de Figuras

2.1 Sistema e seu modelo ([9]). 5
2.2 Trajetória de estado em um espaço de estados contínuo ([9]). 6
2.3 Trajetória de estado em um espaço de estados discreto ([9]). 7
2.4 Possíveis dimensões do tempo para um espaço de estados contínuo. . 7
2.5 Exemplo de Sistema a Eventos Discretos. 8
2.6 Diagrama de transição de estados. 14
2.7 Autômato G para ilustração das operações unárias. 19
2.8 Autômatos resultantes de operações unárias sobre o autômato da figura 2.7. 20
2.9 Autômatos para exemplo de operações de composição. 22
2.10 Autômatos resultantes de operações de composição sobre figura 2.9. . 23
2.11 Autômatos para exemplo de criação de subautômato. 23
2.12 Execução do algoritmo 2.3 para criar $G'_1 \sqsubseteq G'_2$ a partir de G_1 e G_2. . 24
2.13 Exemplo de construção do observador. 26
2.14 Laço de controle supervisório realimentado $S|G$. 28
2.15 Exemplo de malha fechada formada por modelo (G) e especificação (H). 31
2.16 Autômatos do exemplo 2.6. 32
2.17 Laço de controle supervisório realimentado com observação parcial. . 33
2.18 Exemplo simplificado de malha fechada formada por modelo (G) e especificação (H). 36
2.19 Laço de controle supervisório descentralizado e realimentado com dois agentes supervisores. 38
2.20 Exemplo de malha fechada formada por modelo (G) e especificação (H). 41

3.1 Autômatos de entrada do exemplo 3.1. 46
3.2 Autômato H_{obs} do exemplo 3.1. 46
3.3 Autômato $ObsTest$ do exemplo 3.2. 48
3.4 Autômatos do exemplo 3.3. 50
3.5 Autômatos do exemplo 3.4. 52

3.6 Diagrama ilustrativo da definição equivalente da observabilidade. . . . 53
3.7 Autômatos do exemplo 3.5. 58
3.8 Autômatos do Exemplo 1. 64
3.9 Autômatos do Exemplo 2. 66
3.10 Autômatos G, H_R e H_C^D do Exemplo 3. 68
3.11 Autômato V do Exemplo 3. 69
3.12 Autômatos H_m, H_R e H_C^D do Exemplo 4. 71
3.13 Autômato V do Exemplo 4. 72
3.14 Autômatos de entrada para o exemplo 3.7. 76
3.15 Autômato $\tilde{H}$ do exemplo 3.7. 77
3.16 Autômato V_{diag} do exemplo 3.7. 78
3.17 Autômatos $H_{R,1}$, $H_{R,2}$, H_C e H_C^D do exemplo 3.8. 86
3.18 Autômato V do exemplo 3.8. 87
3.19 Autômatos G, H_m e $H_{R,1}$do Exemplo 6. 91
3.20 Autômatos $H_{R,2}$, H_C e H_C^D do Exemplo 6. 92
3.21 Autômato V do Exemplo 6. 93
3.22 Autômatos G e H do Exemplo 7. 95
3.23 Autômatos $H_{R,1}$, $H_{R,2}$ e $H_{R,3}$ do Exemplo 7. 96
3.24 Autômatos H_C e H_C^D do Exemplo 7. 97

4.1 Autômato G gerado pela execução do código 4.1. 100
4.2 Malha fechada formada pelo modelo do sistema (G) e pelo modelo da especificação (H) para o exemplo 4.1. 102
4.3 Autômato H_{supc} gerado pela execução do código 4.3. 104
4.4 Autômato H_{supc} gerado pela execução do código 4.4. 107
4.5 Malha fechada formada pelo modelo do sistema (G) e pelo modelo da especificação (H) para o exemplo 4.5. 111
4.6 Autômato H_{obs} gerado pela execução do código 4.6. 111
4.7 Autômato $ObsTest$ gerado pela execução do código 4.7. 113
4.8 Autômato V_{diag} gerado pela execução do código 4.8. 114
4.9 Autômato V gerado pela execução do código 4.9. 116
4.10 Malha fechada formada pelo modelo do sistema (G) e pelo modelo da especificação (H) para o exemplo 4.9. 119
4.11 Autômato V_{diag} gerado pela execução do código 4.10. 120
4.12 Autômato V gerado pela execução do código 4.11. 122
4.13 Autômato V_{norm} gerado pela execução do código 4.12. 125
4.14 Autômato H_{supn} gerado pela execução do código 4.14. 129
4.15 Autômato H_{supn} gerado pela execução do código 4.15. 130
4.16 Diagrama do controle de uma ferrovia. 133

4.17 Modelos independentes de cada veículo viajando entre as estações A e B. 133
4.18 Modelo síncrono dos veículos viajando entre as estações A e B. 134
4.19 Especificação de segurança para o controle de tráfego de trens. 135
4.20 Especificação controlável e normal para o controle de tráfego de trens. 135

Lista de Tabelas

3.1 Complexidade Computacional do Algoritmo 3.5. 60
3.2 Complexidade Computacional do Algoritmo 3.8. 89

4.1 Sintaxe para acesso às propriedades matemáticas de um autômato. . . 99

Lista de Símbolos

$(\cdot)^T$	Operação de transposição de um vetor ou matriz, p. 5
$(\cdot)^{\uparrow CN}$	Sub-linguagem controlável e normal suprema, p. 38
$(\cdot)^{\uparrow C}$	Sub-linguagem controlável suprema, p. 32
$(\cdot)^{\uparrow N}$	Sub-linguagem normal suprema, p. 37
$Ac(\cdot)$	Parte acessível, p. 17
$CoAc(\cdot)$	Parte coacessível, p. 18
E	Conjunto de eventos, p. 9
E'	$E_R \cup E$, p. 56
E_R	Conjunto de eventos renomeados, p. 54
E_c	Conjunto de eventos controláveis, p. 29
E_o	Conjunto de eventos observáveis, p. 24
$E_{R,i}$	Conjunto de eventos renomeados pelo i-ésimo agente, p. 80
$E_{c,i}$	Conjunto de eventos controláveis pelo i-ésimo agente, p. 40
$E_{o,i}$	Conjunto de eventos observáveis pelo i-ésimo agente, p. 39
E_{uc}	Conjunto de eventos não-controláveis, p. 29
E_{uo}	Conjunto de eventos não-observáveis, p. 24
G^C	Operação complemento sobre o autômato G, p. 19
$G_1 \sqsubseteq G_2$	G_1 é um sub-autômato de G_2, p. 22
$G_1 \times G_2$	Operação produto sobre os autômatos G_1 e G_2, p. 21
$G_1 \| G_2$	Operação de composição paralela sobre os autômatos G_1 e G_2, p. 21

K	Linguagem especificada para um SED, p. 30
M	Linguagem do modelo de um SED, p. 30
P	$E^* \to E_o^*$, p. 33
P'	$E'^* \to E^*$, p. 56
P'_R	$E'^* \to E_R^*$, p. 56
$P'_{R,i}$	$E'^* \to E_{R,i}^*$, p. 82
$P(\cdot)$	Operação de projeção, p. 18
$P^{-1}(\cdot)$	Operação de projeção inversa, p. 18
P_i	$E^* \to E_{o,i}^*$, p. 39
P'_o	$E'^* \to E_o^*$, p. 56
P_{R_o}	$E_R^* \to E_o^*$, p. 56
P_{R,i_o}	$E_{R,i}^* \to E_{o,i}^*$, p. 82
R	Função de renomeação de eventos, p. 54
R_i	Função de renomeação de eventos para o i-ésimo agente, p. 80
S/G	Supervisor S controlando o autômato G, p. 28
$Trim(\cdot)$	Operação trim, p. 18
X	Conjunto finito de estados, p. 14
X_m	Conjunto de estados marcados, p. 14
Γ	Função dos eventos ativos, p. 14
$\Gamma(x)$	Conjunto de eventos ativos no estado x, p. 14
$\emptyset$	Conjunto vazio, p. 11
$\mathbf{u}$	Vetor coluna, p. 5
$\mathcal{L}(\cdot)$	Linguagem gerada por um autômato, p. 15
$\mathcal{L}_m(\cdot)$	Linguagem marcada por um autômato, p. 15
$\mathcal{O}(\cdot)$	Ordem de crescimento do pior caso para um algoritmo, p. 42

$\mathcal{R}$	Classe de linguagens regulares, p. 16
ε	Sequência vazia, p. 9
x_0	Estado inicial, p. 14
$\lvert \cdot \rvert$	Comprimento de uma sequência, ou cardinalidade de um conjunto, p. 9

Lista de Abreviaturas

COPPE	Instituto Alberto Luiz Coimbra de Pós-graduação e Pesquisa de Engenharia, p. 98
DESLAB	*Discrete Event Systems Laboratory*, p. 2
LCA	Laboratório de Controle e Automação, p. 98
PDF	*Portable Document Format*, p. 99
SED	Sistema a eventos discretos, p. 1
UFRJ	Universidade Federal do Rio de Janeiro, p. 98

Capítulo 1

Introdução

Uma simples reflexão sobre a evolução dos processos produtivos permite observar uma característica comum a eles: o constante aumento de escala. Impulsionados por diversos motivos, como ganhos econômicos e aumento da população, o aumento da escala dos processos produtivos implica necessariamente um aumento da complexidade desses, e consequentemente, também, do seu controle. Manter um processo operando de maneira segura e ótima torna-se impossível sem o advento de uma nova camada de controle. Denominada controle supervisório, essa camada é responsável por supervisionar a interligação entre diversos sub-processos, individualmente controlados, de forma a mantê-los íntegros e operando da maneira mais benéfica para a planta como um todo.

Todavia, apesar de atualmente todo processo produtivo possuir um sistema de supervisão capaz de fornecer aos operadores as informações e meios para que o controle supervisório seja executado, ainda não se encontra disseminado o conceito de controle supervisório automático, no qual, baseado em um modelo da planta, um supervisor automático seja sintetizado para fechar a malha de controle supervisório, retirando este papel do operador. A formalização matemática do conceito de controle supervisório feita em [1] visa, efetivamente, fornecer uma base para que esse objetivo seja alcançado. Ressalta-se que, além de processos produtivos, a teoria de controle supervisório pode ser aplicada a diversos outros domínios, como por exemplo, no campo da robótica ([2], [3] e [4]) e em controle de *workflows* ([5] e [6]).

A base da teoria de controle supervisório é o conceito de sistemas a eventos discretos (SEDs), os quais possuem conjuntos de estados discretos e têm a evolução realizada pela ocorrência de eventos assíncronos ([7], [8] e [9]). Dentre os formalismos utilizados para a modelagem de SEDs, os autômatos de estados finitos serão utilizados neste trabalho para modelar os sistemas, sendo também utilizados como base para os algoritmos estudados.

Uma preocupação recorrente no desenvolvimento da teoria de controle supervisório tem sido a ordem de crescimento dos algoritmos desenvolvidos, uma vez que

algoritmos pouco eficientes aplicados a sistemas com muitos estados podem criar problemas computacionais, como falta de memória física, por exemplo. Com esta preocupação em mente, a principal contribuição deste trabalho será apresentada: o desenvolvimento de algoritmos mais eficientes para verificar as propriedades de observabilidade e normalidade de linguagens regulares ([10]), as quais são condições necessárias para que o controle supervisório seja viável sob existência de eventos não-observáveis[1] pelo supervisor.

Este estudo será composto primeiramente pela revisão dos principais algoritmos existentes na literatura. Métodos imediatos, baseados na síntese de um autômato observador são apresentados tanto para a verificação da observabilidade ([9]) como da normalidade. Todavia, esses métodos podem apresentar crescimento exponencial em relação ao tamanho do sistema, o que leva à necessidade de se obter métodos mais eficientes. Assim, para a observabilidade, serão apresentados o primeiro método de complexidade polinomial, proposto por TSITSIKLIS [11], e também um método mais eficiente, proposto por WANG *et al.* [12]. Com respeito à propriedade da normalidade, não foram identificados na literatura algoritmos de complexidade polinomial.

Na sequência desse estudo, inspirado pelos conceitos utilizados em MOREIRA *et al.* [13], um novo verificador de complexidade polinomial será proposto para a verificação simultânea das propriedades de observabilidade e normalidade. Para tanto, dois algoritmos serão definidos. O primeiro algoritmo visa verificar a observabilidade e possui a mesma complexidade de pior caso (pela notação $\mathcal{O}$) que o proposto por WANG *et al.* [12]. O segundo algoritmo visa verificar a normalidade, sendo, de acordo com o conhecimento do autor, o primeiro algoritmo polinomial proposto.

Ainda sobre o problema da síntese de supervisores sob existência de eventos não-observáveis, uma investigação será feita para a arquitetura descentralizada de controle supervisório proposta em [10]. Mais uma vez o principal algoritmo existente é o proposto por WANG *et al.* [12], sendo analisado neste trabalho. Em seguida, será apresentada uma generalização do verificador proposto para a observabilidade, resultando em um novo algoritmo proposto para verificação da coobservabilidade, também com mesma complexidade de pior caso (pela notação $\mathcal{O}$) que o proposto por ([12]).

Por fim, com o objetivo de facilitar o estudo e aplicação da teoria de controle supervisório, será também apresentada neste trabalho uma biblioteca para o programa DESLAB ([14]), que implementa as funções mais utilizadas na síntese e análise de supervisores. Essa biblioteca será apresentada através do detalhamento de cada algoritmo, acompanhado de um exemplo de utilização.

A organização deste trabalho é feita da seguinte forma. No capítulo 2 serão

[1] Um evento é dito ser observável se ele é detectado pelo observador.

apresentados os fundamentos teóricos necessários para a compreensão dos resultados apresentados nos capítulos seguintes: sistemas a eventos discretos, controle supervisório e complexidade computacional de algoritmos. No capítulo 3 os problemas da verificação da observabilidade, normalidade, e coobservabilidade serão investigados a partir da análise dos principais algoritmos existentes na literatura, e da proposta de algoritmos mais eficientes. No capítulo 4 o programa DESLAB será apresentado, e em seguida, cada função da biblioteca para controle supervisório desenvolvida neste trabalho será detalhadamente descrita. Finalmente, no capítulo 5 um resumo dos principais resultados deste trabalho será feito, e serão apresentadas algumas propostas de trabalhos futuros.

Capítulo 2

Controle supervisório de Sistemas a Eventos Discretos

Baseado em CASSANDRAS e LAFORTUNE [9] e WONHAM [15], este capítulo apresenta uma revisão de todos os conceitos que formam a base para o desenvolvimento apresentado nos próximos capítulos. A seção 2.1 apresenta os conceitos básicos de sistema a eventos discretos e também um formalismo para o estudo desse tipo de sistema, o autômato. A seção 2.2 estende o estudo para o caso em que se deseja controlar um SED, por meio do conceito de realimentação. Por fim, a seção 2.3 apresenta o conceito de complexidade computacional de algoritmos.

2.1 Sistemas a Eventos Discretos

2.1.1 Conceitos Básicos

Sistemas podem ser definidos de várias formas, de acordo com o contexto da análise, sendo muitas vezes intuitivo. Porém, para o âmbito das ciências exatas, de um modo geral, é possível defini-los da seguinte maneira [16]:

> *"Um sistema é uma combinação de componentes que agem juntos para desempenhar uma função que um componente isolado não é capaz de desempenhar."*

Contudo, uma definição puramente qualitativa como esta não engloba os aspectos quantitativos que a engenharia exige. De uma maneira geral, a engenharia está interessada em estudar uma dada porção do universo, a qual é, então, definida como sistema em estudo. Sobre este é, então, possível definir um conjunto de variáveis mensuráveis que interagem com o sistema em um dado período de tempo $[t_0, t_f]$. Essas variáveis podem ser divididas em dois tipos: aquelas que nos possibilitam alterá-las com o passar do tempo (variáveis de entrada) e aquelas que são modificadas

com a variação de algumas das variáveis anteriores (variáveis de saída). Desse modo, é possível estender a definição anterior para permitir uma representação matemática capaz de descrever o comportamento de um sistema, conhecido como o modelo de um sistema. A figura 2.1 sintetiza o processo de modelagem de um sistema.

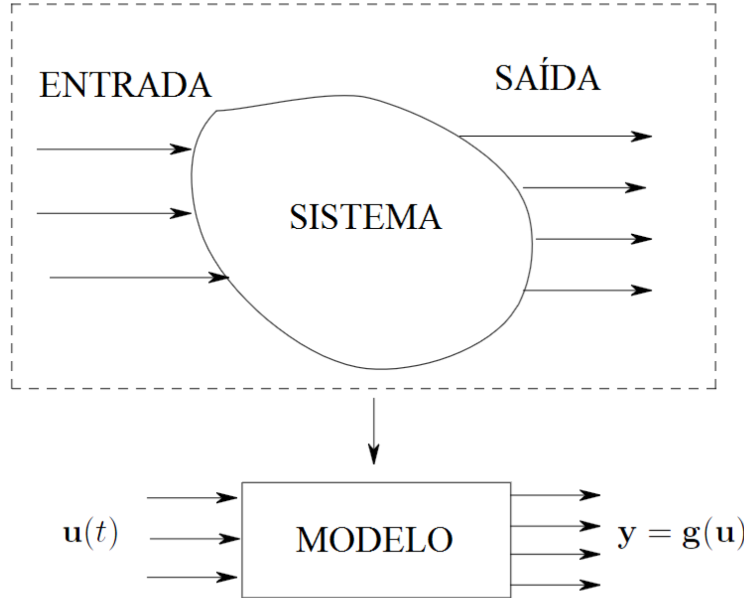


Figura 2.1: Sistema e seu modelo ([9]).

Matematicamente, o conjunto de variáveis de entrada é representando pelo vetor $\mathbf{u}(t) = [u_1(t), ..., u_p(t)]^T$, em que $[\cdot]^T$ denota a operação de transposição de um vetor, enquanto que as variáveis de saída são representadas pelo vetor $\mathbf{y}(t) = [y_1(t), ..., y_m(t)]^T$. O processo de modelagem pode ser resumido, então, em duas etapas. A primeira consiste na definição dos conjuntos de variáveis $\mathbf{u}(t)$ e $\mathbf{y}(t)$. A segunda etapa consiste na obtenção da relação matemática dinâmica que relaciona o efeito que cada variável $u_i(t)$, $i = 1, ..., p$, gera em cada saída $y_j(t)$, $j = 1, ..., m$. Essa relação pode ser sintetizada da seguinte forma:

$$\mathbf{y} = \mathbf{g}(\mathbf{u}) = [g_1(u_1(t), ..., u_p(t)), ..., g_m(u_1(t), ..., u_p(t))]^T, \tag{2.1}$$

em que $\mathbf{g}[\cdot]$ denota o vetor coluna formado pelas funções $g_1(\cdot), ..., g_m(\cdot)$.

Apesar de um sistema estar totalmente descrito pelo seu modelo da equação (2.1), é possível modificar essa representação para colocar em evidência seus modos dinâmicos principais, formalizados pela definição de estado. Representados pelas variáveis $x_k(t)$, $k = 1, ..., n$, o vetor de estados de um sistema dinâmico pode ser definido como:

$$\mathbf{x}(t) = [x_1(t), ..., x_n(t)]^T. \tag{2.2}$$

Definição 2.1. *(Estado) O estado de um sistema dinâmico ([9]),* $\mathbf{x}(t_0)$*, é a informação em* t_0 *que junto com o conhecimento da entrada* $\mathbf{u}(t)$*, para* $t \geq t_0$*, é suficiente*

para determinar unicamente a saída $\mathbf{y}(t), \forall t \geq t_0$.

Além da própria definição de estado, outro importante conceito é o de espaço de estados.

Definição 2.2. *(Espaço de Estados) O espaço de estados de um sistema, usualmente denotado por* X*, é o conjunto de todos os possíveis valores que o estado* $\mathbf{x}(t)$ *pode assumir.*

Com isso é possível definir um modelo de sistema com base em seus estados, resultando na equação de estados de um sistema dinâmico:

$$\dot{\mathbf{x}}(t) = f[\mathbf{x}(t), \mathbf{u}(t), t] \ \ , \ \ \mathbf{x}(t_0) = \mathbf{x}_0 \ ; \tag{2.3}$$

$$\mathbf{y}(t) = g[\mathbf{x}(t), \mathbf{u}(t), t], \tag{2.4}$$

em que a primeira equação contém as equações de estados atreladas às condições iniciais, e a segunda contém as equações de saída.

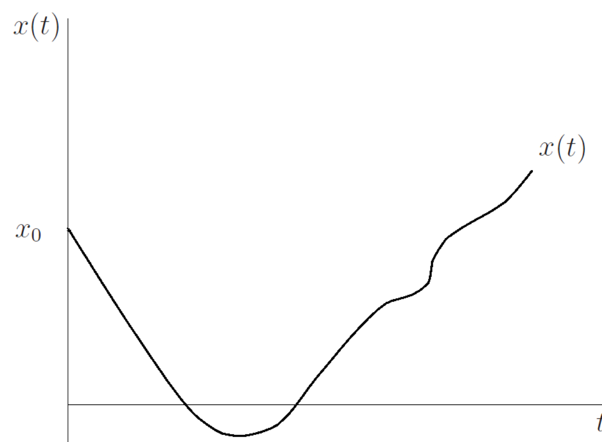


Figura 2.2: Trajetória de estado em um espaço de estados contínuo ([9]).

É importante observar que, apesar de extremamente comum, o uso de equações diferenciais na modelagem de um sistema em espaço de estados limita a aplicação desse método aos sistemas cujos espaços de estados são contínuos, ou seja, o estado do sistema pode assumir qualquer valor, normalmente real, dentro de um intervalo. Nesse caso, a trajetória do estado ao longo do tempo pode ser representada por uma função contínua por partes, ou por uma única função contínua, como representado na figura 2.2. Porém, existe um grande universo de problemas nos quais os estados assumem valores discretos, como por exemplo uma válvula possui o espaço de estados {ABERTO, FECHADO}, enquanto que o nível em um tanque pode assumir um dos seguintes valores {MUITO ALTO, ALTO, NORMAL, BAIXO, MUITO BAIXO}. Sistemas caracterizados por possuir espaços de estados enumeráveis ou discretos são chamados de *sistemas de estados discretos*. É possível notar nesse tipo de sistema

que o estado muda de valor em instantes específicos do tempo. Assim, a trajetória do estado do sistema ao longo do tempo é normalmente representada utilizando uma função constante por partes, como exemplificado da figura 2.3.

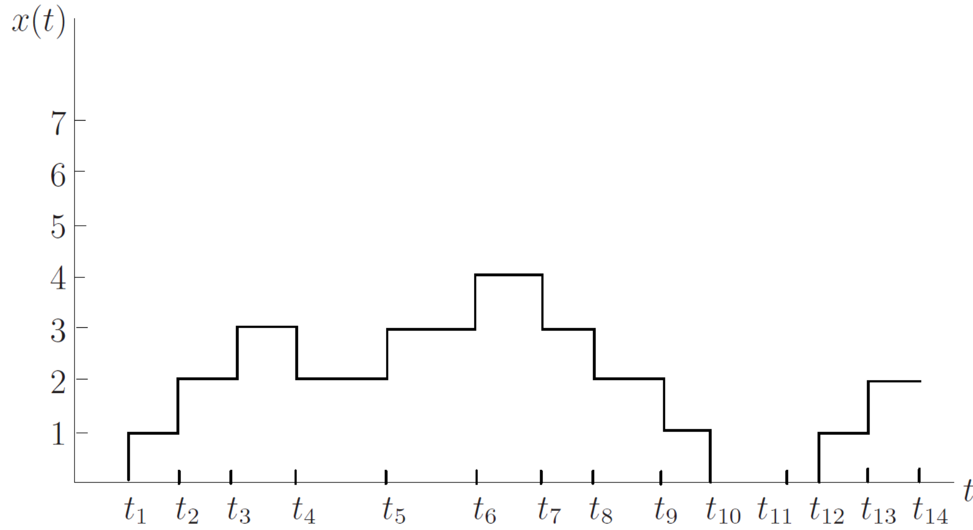
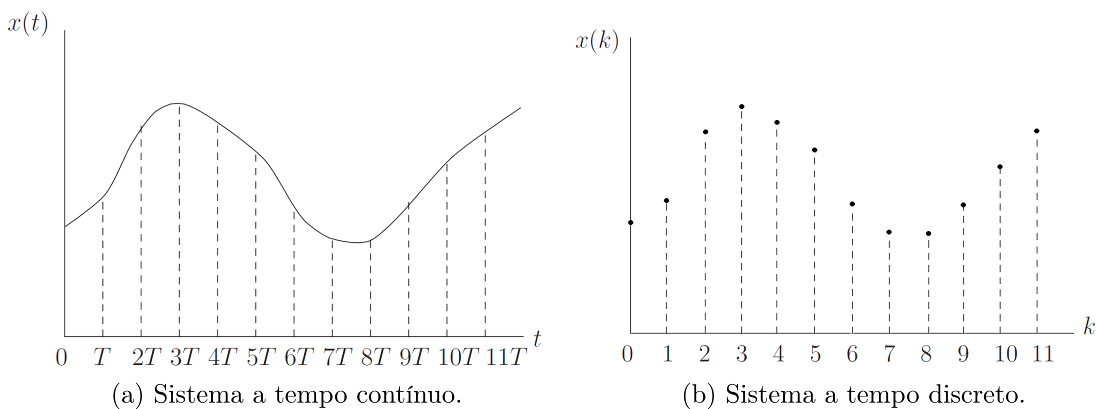


Figura 2.3: Trajetória de estado em um espaço de estados discreto ([9]).

Até este ponto, o tempo foi considerado como uma variável contínua. Porém, existe uma outra classe de sistemas nos quais é preciso determinar os valores das variáveis apenas em instantes discretos de tempo. Sistemas desse tipo são chamados de *sistemas a tempo discreto*, uma vez que a dimensão do tempo é uma variável discreta. A grande motivação para o estudo desse tipo de sistema está no aspecto estritamente discreto dos computadores. Ressalta-se, porém, que a discretização do tempo é independente da discretização do espaço de estados de um sistema. Um exemplo desse fato está ilustrado na figura 2.4. Note que a notação $x(k)$ é usada na figura 2.4b para se referir ao estado no k-ésimo instante de amostragem.



(a) Sistema a tempo contínuo.

(b) Sistema a tempo discreto.

Figura 2.4: Possíveis dimensões do tempo para um espaço de estados contínuo.

No estudo de sistemas com espaço de estados contínuo, o entendimento de que o estado varia de acordo com o tempo (seja ele contínuo ou discreto) é intuitivo. Essa

característica fica ainda mais evidente no caso de sistemas com espaço de estados discreto, uma vez que é possível identificar claramente o instante quando o estado mudou de valor. Porém, se atrelarmos essa variação de valor do estado a um evento, é possível definir um novo tipo de sistema, chamado de sistema dirigido por eventos. Assim, quando em um sistema, a transição de estados estiver associada ao tempo, diz-se que este sistema é dirigido pelo tempo. Por outro lado, quando a transição de estados estiver associada à ocorrência de um evento, esse sistema é referido como dirigido por eventos. Exemplos de eventos podem ser a ativação de um alarme ou um comando inicializado por um operador.

Com o desenvolvimento feito até este ponto é então possível apresentar a caracterização dos SEDs, os quais formam a base para a teoria apresentada neste trabalho.

Definição 2.3. *(Sistemas a Eventos Discretos) Sistemas a eventos discretos ([9]) são sistemas dinâmicos de estados discretos e dirigido por eventos, isto é, cuja transição de estados se dá por meio da ocorrência, em geral assíncrona, de eventos.*

A figura 2.5 exemplifica como ocorre a evolução de um SED. É possível ver que no instante inicial, o sistema se encontra no estado s_2. Após a ocorrência do evento e_1 o estado muda para s_5. Em seguida, após a ocorrência da sequência de eventos $e_2e_3e_4e_5e_6e_7$ o sistema encontra-se no estado s_6. Note que a definição do estado atual do sistema não depende do tempo, mas sim da sequência de eventos ocorrida até aquele ponto.

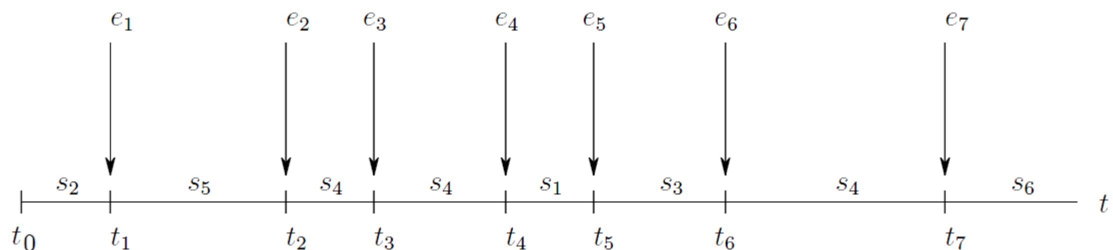


Figura 2.5: Exemplo de Sistema a Eventos Discretos.

Um ponto importante do conceito de SEDs é que ele permite modelar qualquer sistema físico, bastando para isso que um grau suficiente de abstração seja considerado. Um exemplo disso são os sistemas supervisórios industriais, que ao se localizarem numa camada acima do nível de controle de processo (onde os sistemas estão intimamente atrelados ao tempo) funcionam com base em eventos criados por estes.

2.1.2 Linguagem de um SED

A análise de sistemas a eventos discretos consiste em dois preceitos principais: qual é a sequência de estados visitados e quais eventos estão associados a essa sequência. Qualquer sistema a eventos discretos possui um conjunto finito de eventos $E = \{e_1, ..., e_n\}$ associado a ele, também conhecido como o *alfabeto* do SED. Supondo que um dado comportamento de um SED pode ser descrito por uma *sequência*[1] de eventos, do tipo $e_1e_2...e_n$, então, o conjunto de todos os possíveis comportamentos de um sistema pode ser definido por um conjunto de sequências, denominado *linguagem*. Sobre um mesmo conjunto de eventos E é possível definir diferentes linguagens. Por exemplo, seja um conjunto de eventos definidos por $E = \{a, b, g\}$. Então é possível definir uma linguagem L_1 contendo todas as possíveis sequências de comprimento três que comecem com o evento a, ou então uma linguagem L_2 que contém todas as sequências de comprimento finito que comecem com o evento a. Essas linguagens podem também ser representadas por:

$$L_1 = \{aaa, abb, abg, agg, agb, aab, aag, aba, aga\};$$
$$L_2 = \{a, aa, ab, ag, aaa, aab, aag, aba, abb, abg, aga, agb, agg, aaaa, ...\}.$$

Vamos apresentar agora algumas definições básicas com respeito à teoria das linguagens.

Definição 2.4.

(i) *O comprimento de uma sequência s é o número de eventos contidos nessa sequência, contando as múltiplas ocorrências de um mesmo evento. Se s é uma sequência, o seu comprimento é denotado por $|s|$.*

(ii) *A sequência que não contém eventos é chamada de sequência vazia e é denotada por ε. Por definição $|\varepsilon| = 0$.*

(iii) *(Concatenação) É a operação chave utilizada para criar uma sequência a partir de duas ou mais sequências. Por exemplo, a sequência abg é a concatenação da sequência ab com o evento (ou sequência de comprimento um) g.*

(iv) *ε é o elemento identidade da operação de concatenação, ou seja, $s\varepsilon = \varepsilon s = s$, $\forall s$.*

(v) *(Fecho de Kleene) O Fecho de Kleene de um conjunto de eventos E, denotado por E^*, é o conjunto formado por todas as sequências de comprimento finito geradas a partir dos elementos de E, incluindo a sequência vazia ε. Assim*

[1] Também conhecida como *palavra*, *traço* ou *cadeia*.

sendo, a maior linguagem que pode ser definida sobre um conjunto de eventos E é E^*, e qualquer outra linguagem é um subconjunto de E^*.

(vi) *Dada a sequência $s = tuv$, com $t, u, v \in E^*$, então:*

- t *é um prefixo de* s;
- u *é uma subsequência de* s;
- v *é um sufixo de* s;
- s *é um prefixo, subsequência e sufixo de* s;
- ε *é também prefixo, subsequência e sufixo de* s.

(vii) *A notação s/t (lê-se "s após t") é usada para denotar o sufixo de s após o prefixo t. Se t não for um prefixo de s, então a operação não é definida.*

Operações com Linguagens

Uma vez que uma linguagem é um conjunto cujos elementos são palavras, toda a teoria de conjuntos se aplica a ela. Desse modo, operações como união, interseção, diferença[2] e complemento com respeito a E^* também se aplicam a linguagens. Porém, outras operações muito úteis no estudo de SEDs podem ser definidas sobre linguagens.

Definição 2.5. *(Concatenação) Sejam $L_a, L_b \subseteq E^*$. Então, uma palavra s pertence à concatenação das linguagens L_a e L_b, se ela puder ser escrita como a concatenação de uma palavra pertencente a L_a com outra pertencente a L_b, ou seja:*

$$L_a L_b := \{s \in E^* : (\exists s_a \in L_a)(\exists s_b \in L_b)[s = s_a s_b]\}.$$

Definição 2.6. *(Fecho do Prefixo[3]) Seja $L \subseteq E^*$. Então o fecho do prefixo de L, denotado por $\overline{L}$, é a linguagem formada por todos os prefixos de todas as sequências pertencentes a L, isto é:*

$$\overline{L} := \{s \in E^* : (\exists t \in E^*)[st \in L]\}.$$

Se $L = \overline{L}$ então L é dita ser prefixo-fechada.

Definição 2.7. *(Fecho do Kleene) Seja $L \subseteq E^*$. Então o fecho de Kleene de L, denotado por L^*, é o conjunto formado por todas as possíveis concatenações entre as sequências pertencentes a L. Logo essa operação pode ser expressa como:*

$$L^* := \{\varepsilon\} \cup L \cup LL \cup LLL \cup \ldots.$$

---

[2] Denota-se a operação de diferença entre as linguagens L_1 e L_2 como $L_1 \backslash L_2$.

[3] Também chamado de *prefixo-fechamento*.

Note que o fecho de Kleene é uma operação idempotente, ou seja, $(L^*)^* = L^*$.

Definição 2.8. *(Pós-linguagem) Seja $L \subseteq E^*$. Então a pós-linguagem de L após s, denotada por L/s, é o conjunto formado por todos os sufixos de todas as sequências de L que possuem s como prefixo:*

$$L/s := \{t \in E^* : st \in L\}.$$

Por definição, $L/s = \emptyset$ se $s \notin L$.

Definição 2.9. *(Quociente de Linguagem) Sejam $L_1, L_2 \subseteq E^*$. Define-se o quociente entre L_1 e L_2 (L_1/L_2) como o conjunto formado por todas os as sequências de L_1 cujos sufixos pertencem a L_2:*

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

De modo a exemplificar as operações definidas acima, considere um conjunto $E = \{a, b, g\}$, e três linguagens $L_1 = \{\varepsilon, a, abb\}$, $L_2 = \{g\}$ e $L_3 = \{a\}$. Então:

$$\begin{aligned}
L_1L_2 &= \{g, ag, abbg\};\\
\overline{L_1} &= \{\varepsilon, a, ab, abb\};\\
\overline{L_2} &= \{\varepsilon, g\};\\
L_1^* &= \{\varepsilon, a, abb, aa, aabb, abba, abbabb, ...\};\\
L_2^* &= \{\varepsilon, g, gg, ggg, ...\};\\
L_1/L_2 &= \emptyset;\\
L_1/L_3 &= \{\varepsilon\}.
\end{aligned}$$

Por fim, é possível fazer as seguintes observações a respeito das operações aqui definidas:

(i) Se $L = \emptyset$ então $\overline{L} = \emptyset$. E, se $L \neq \emptyset$, então, necessariamente, $\varepsilon \in \overline{L}$;

(ii) $\emptyset^* = \{\varepsilon\}$, $\{\varepsilon\}^* = \{\varepsilon\}$;

(iii) $L\emptyset = \emptyset L = L$.

Projeção de Linguagens

As operações de projeção, e de projeção inversa, de linguagens, constituem uma importante ferramenta para o estudo de sistemas a eventos discretos. Sua definição se inicia com a projeção de sequências a seguir.

Definição 2.10. *(Projeção) Seja E_s um conjunto menor de eventos, e E_l um conjunto maior de eventos, tais que $E_s \subset E_l$. A projeção $P : E_l^* \to E_s^*$ é definida recursivamente da seguinte forma:*

$$P(\varepsilon) := \varepsilon;$$

$$P(e) := \begin{cases} e & se\ e \in E_s; \\ \varepsilon & se\ e \in E_l \backslash E_s; \end{cases}$$

$$P(se) := P(s)P(e) \ \ para \ \ s \in E_l^*, e \in E_l.$$

Como exemplo, sejam $E_l = \{a, b, c\}$, $E_s = \{a, b\}$ e $s = abbca \in E_l^*$. Então:

$$P(s) = P(abbca) = abba.$$

O mapeamento inverso, definido pela operação de projeção inversa, pode também ser definida para uma dada sequência da seguinte maneira.

Definição 2.11. *(Projeção Inversa) Seja E_s um conjunto menor de eventos, e E_l um conjunto maior de eventos tais que $E_s \subset E_l$. A projeção inversa*[4] $P^{-1} : E_s^* \to 2^{E_l^*}$ *é definida da seguinte forma:*

$$P^{-1}(s) := \{t \in E_l^* : P(t) = s\}.$$

Como exemplo, sejam os conjuntos $E_l = \{a, b, c\}$, $E_s = \{a, b\}$ e a sequência $s = ab \in E_s^*$. Então:

$$P^{-1}(s) = \{c\}^* a \{c\}^* b \{c\}^*.$$

É possível estender a operação projeção para linguagens, fazendo com que a operação seja aplicada sobre cada elemento do conjunto. Formalmente, isso pode ser representado pelas definições a seguir.

Definição 2.12. *(Projeção de uma Linguagem) Seja a operação de projeção dada na definição 2.10, e seja a linguagem $L \subseteq E_l^*$. Então:*

$$P(L) := \{t \in E_s^* : (\exists s \in L)[P(s) = t]\}.$$

Definição 2.13. *(Projeção Inversa de uma Linguagem) Seja a operação de projeção inversa dada na definição 2.11, e seja a linguagem $L \subseteq E_s^*$. Então:*

$$P^{-1}(L) := \{s \in E_l^* : (\exists t \in L)[P(s) = t]\}.$$

[4] 2^A é o conjunto potência de A, isto é, o conjunto formado por todos os subconjuntos de A.

As operações de projeção sobre linguagens possuem propriedades importantes que serão utilizadas ao longo deste trabalho. Considere que L, A e B são linguagens, então:

(i) $P[P^{-1}(L)] = L$;

(ii) $P^{-1}[P(L)] \supseteq L$;

(iii) Se $A \subseteq B$, então $P(A) \subseteq P(B)$ e $P^{-1}(A) \subseteq P^{-1}(B)$;

(iv) $P(A \cup B) = P(A) \cup P(B)$;

(v) $P(A \cap B) \subseteq P(A) \cap P(B)$;

(vi) $P^{-1}(A \cup B) = P^{-1}(A) \cup P^{-1}(B)$;

(vii) $P^{-1}(A \cap B) = P^{-1}(A) \cap P^{-1}(B)$;

(viii) $P(AB) = P(A)P(B)$;

(ix) $P^{-1}(AB) = P^{-1}(A)P^{-1}(B)$.

2.1.3 Autômatos

Na seção anterior apresentamos os diversos conceitos de linguagem e as suas operações. No contexto de SEDs, linguagens são utilizadas para descrever o comportamento de um sistema, especificando todas as sequências de eventos admissíveis que um dado SED pode executar. Também foi visto que linguagens podem ser descritas por um número infinito de sequências, o que mostra a dificuldade de se utilizar apenas a teoria de conjuntos para o estudo de linguagens, e consequentemente de SEDs.

Com o intuito de resolver essas questões, será apresentado nesta seção um formalismo para o estudo de SEDs, os *autômatos*. Com uso desta teoria, a representação, manipulação, operação e resolução de problemas para uma determinada classe de linguagens se torna mais simples, possibilitando o trabalho com sistemas de grande complexidade.

Um autômato, também chamado de máquina de estados, é um dispositivo capaz de representar uma linguagem de acordo com regras bem definidas. Uma de suas principais vantagens é possibilitar uma visão gráfica de um sistema por meio de um diagrama de transição de estados. Nesses diagramas, os estados são representados por circunferências e são conectados entre si por arcos identificados (rotulados) com símbolos, que representam os eventos que determinam as transições entre os estados ligados pelo arco. Os estados marcados são identificados por duas circunferências concêntricas e estão, em geral, relacionados ao cumprimento de uma tarefa a ser

realizada pelo sistema modelado pelo autômato. O estado inicial é indicado por uma seta apontada a ele, não oriunda de qualquer outro estado. Um exemplo de um diagrama de transição de estados pode ser visto na figura 2.6.

Neste ponto é possível realizar a apresentação formal do conceito de autômato.

Definição 2.14. *(Autômato[5]) Um autômato G é uma sêxtupla*

$$G = (X, E, f, \Gamma, x_0, X_m), \tag{2.5}$$

em que

- X *é o conjunto finito de estados;*
- E *é o conjunto finito de eventos associados a* G*;*
- $f : X \times E \rightarrow X$ *é a função de transição de estados, geralmente parcial sobre seu domínio.* $f(x, e) = y$ *significa que existe uma transição do estado* x *para o estado* y *rotulada pelo evento* e*. O fato de que cada evento* e *só pode estar ativo uma única vez em cada estado* x *caracteriza o autômato como determinístico.*
- $\Gamma : X \rightarrow 2^E$ *é a função dos eventos ativos (viáveis);* $\Gamma(x)$ *denota o conjunto de eventos ativos no estado* x*;*
- x_0 *é o estado inicial;*
- $X_m \subseteq X$ *é o conjunto de estados marcados.*

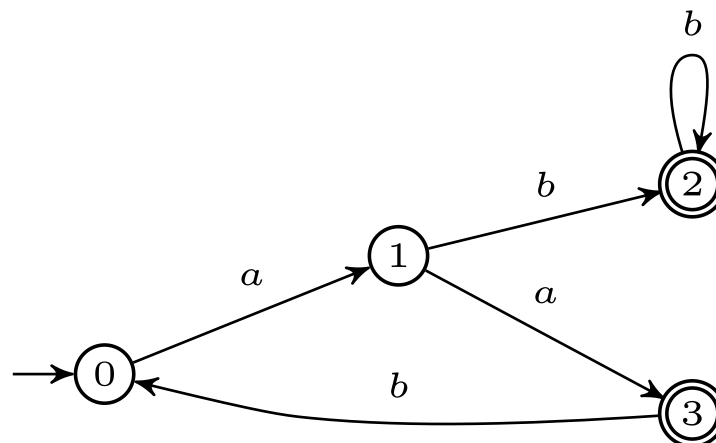


Figura 2.6: Diagrama de transição de estados.

Um autômato G funciona sempre de maneira sequencial, iniciando a partir de x_0. O estado do sistema permanece o mesmo até que ocorra um evento $e \in \Gamma(x_0) \subseteq E$.

[5]Neste trabalho, o termo *autômato* se refere a autômatos determinísticos ($f : X \times E \rightarrow X$) e de estados finitos (conjunto X finito).

Quando da ocorrência do evento e, acontece uma transição de estado $f(x_0, e) = x \in X$ (podendo ser $x = x_0$), que leva o sistema para o estado x. Esse processo acontece continuamente enquanto o sistema estiver em operação.

Seja, então, o diagrama de transição de estados da figura 2.6. Ele representa um autômato determinístico $G = (X, E, f, \Gamma, x_0, X_m)$ em que $X = \{0, 1, 2, 3\}$, $E = \{a, b\}$, $x_0 = 0$ e $X_m = \{2, 3\}$. A evolução dinâmica do autômato se dá da seguinte forma. Quando ligado, o sistema se encontra no estado $x_0 = 0$. Nessa situação, somente o evento a pode ocorrer e, portanto, $\Gamma(0) = \{a\}$. A ocorrência do evento a muda o estado do autômato de 0 para 1; formalmente tem-se que $f(0, a) = 1$. No estado $x = 1$, há duas possibilidades de ocorrência de eventos: a ou b, o que é caracterizado pela função dos eventos ativos, isto é, $\Gamma(1) = \{a, b\}$. Se o evento b ocorrer, o estado do sistema mudará para $x = 2$ e se a ocorrer, ter-se-á a transição para o estado $x = 3$. Note que existe uma transição definida por um autolaço no estado $x = 2$, significando que o autômato permanecerá no estado $x = 2$, mesmo com a ocorrência do evento b.

Linguagens de Autômatos

Uma vez que a motivação inicial para a introdução de autômatos foi sua capacidade de representar linguagens de um SED, faz-se necessário estabelecer uma ligação entre um autômato e a linguagem modelada por ele.

São dois os tipos de linguagens que podem ser associadas ao comportamento de um autômato: a linguagem gerada e a linguagem marcada. A linguagem gerada (denotada por $\mathcal{L}$) representa todas as sequências que podem ocorrer no diagrama de transição de estados, começando pelo estado inicial. Já a linguagem marcada (denotada por $\mathcal{L}_m$) é um subconjunto da linguagem gerada e consiste em todas as sequências que terminam em um estado marcado no diagrama de transição de estados.

Porém, antes de se definir $\mathcal{L}$ e $\mathcal{L}_m$ formalmente, deve-se inicialmente estender o domínio de f de $X \times E$ para $X \times E^*$ da seguinte forma recursiva:

$$
\begin{aligned}
f(x, \varepsilon) &:= \varepsilon; \\
f(x, se) &:= f[f(x, s), e] \text{ , para } s \in E^* \text{ e } e \in E.
\end{aligned}
$$

Essa extensão da função de transição de estados é uma generalização de f, para os casos em que o segundo argumento não tem comprimento igual a um. Ao usar essa generalização não é necessária uma nova notação para f, pois ambas as definições são consistentes para o caso de um único evento.

Feito isso, é possível definir formalmente as linguagens gerada e marcada por um

autômato $G = (X, E, f, \Gamma, x_0, X_m)$ como:

$$\mathcal{L}(G) := \{s \in E^* : (\exists x \in X)[f(x_0, s) = x]\} \text{ e,}$$
$$\mathcal{L}_m(G) := \{s \in L : f(x_0, s) \in X_m\}.$$

Retornando ao autômato representado na figura 2.6, é possível afirmar que $\mathcal{L}_m(G) = \{a\}\{aba\}^*[\{b\}\{b\}^* \cup \{a\}\{baa\}^*]$ e $\mathcal{L}(G) = \overline{\mathcal{L}_m(G)}$.

Ainda sobre as linguagens de um autômato, é possível realizar as seguintes observações:

(i) Note que $f(x_0, \varepsilon) = x_0$ e, portanto, $\varepsilon \in \mathcal{L}(G)$;

(ii) $\mathcal{L}(G)$ é prefixo-fechada, ou seja, $\mathcal{L}(G) = \overline{\mathcal{L}(G)}$;

(iii) A linguagem marcada $\mathcal{L}_m(G)$ não é necessariamente prefixo-fechada;

(iv) O autômato vazio, denotado por $G = \emptyset$, é aquele que gera e marca o conjunto vazio.

Por fim, é importante ressaltar que existem linguagens que não podem ser marcadas por um autômato com um número finito de estados. Por este motivo, faz-se necessária a definição da classe de linguagens que podem ser modeladas por máquinas de estados finitos.

Definição 2.15. *(Linguagem Regular) Uma linguagem é regular quando pode ser marcada por um autômato de estados finitos. A classe de linguagens regulares é denotada por $\mathcal{R}$.*

Todas as linguagens consideradas neste trabalho, a partir deste ponto, serão linguagens regulares.

Bloqueio

A partir da definição de linguagens gerada e marcada por um autômato G, é possível obter a seguinte relação de inclusão:

$$\mathcal{L}_m(G) \subseteq \overline{\mathcal{L}_m(G)} \subseteq \mathcal{L}(G).$$

A primeira relação de inclusão se deve ao fato de que X_m pode ser um subconjunto próprio de X. A segunda relação de inclusão é consequência da definição de $\mathcal{L}_m(G)$ e do fato de que $\mathcal{L}(G)$ é prefixo-fechada.

Examinando em detalhes a segunda relação, é possível notar que um autômato G, no decorrer de sua evolução de estados, pode alcançar um estado x tal que $\Gamma(x) = \emptyset$ mas $x \notin X_m$. Este é um tipo de bloqueio, conhecido como *deadlock*, uma vez que

nenhum evento poderá ser executado a partir de então. Diz-se que o sistema entra em bloqueio porque não chegou ao objetivo final de sua execução (simbolizado por um estado marcado).

Existe um outro cenário de bloqueio que ocorre quando um conjunto de estados não-marcados de G formam um componente fortemente conexo mas sem transições capazes de levar para um estado marcado fora desses componentes. Esse tipo de bloqueio é conhecido como *livelock*, já que, embora o sistema permaneça "vivo" (eventos permanecem ativos), um estado marcado nunca será alcançado.

É possível formalizar o conceito de bloqueio da seguinte forma.

Definição 2.16. *(Bloqueio) Um autômato G possui um bloqueio (*deadlock *ou* livelock*) se $\overline{\mathcal{L}_m(G)} \subset \mathcal{L}(G)$. Se o autômato G não possui bloqueio, então $\overline{\mathcal{L}_m(G)} = \mathcal{L}(G)$.*

Operações com Autômatos

Para tornar possível a utilização de autômatos como um formalismo, é necessário definir um conjunto de operações sobre eles. Assim será apresentado a seguir, um conjunto de operações, tanto unárias como de composição, que permitem modificar e compor autômatos.

Operações Unárias Vamos considerar, inicialmente, as operações que alteram o diagrama de transição de estados de um único autômato. Em todas elas o conjunto de eventos E permanece inalterado.

- **Parte Acessível** Pela definição de $\mathcal{L}(G)$ e $\mathcal{L}_m(G)$, é possível notar que a remoção de G de todos os estados que não são acessíveis ou alcançáveis[6] a partir de x_0 por alguma sequência em $\mathcal{L}(G)$, não afeta a linguagem gerada ou marcada de G. Ao remover um estado, todas as transições associadas a ele devem também ser removidas. Essa operação é denotada por $Ac(G)$, e pode ser formalmente definida como:

$$Ac(G) := (X_{ac}, E, f_{ac}, x_0, X_{ac,m})$$

em que

$$\begin{aligned} X_{ac} &= \{x \in X : (\exists s \in E^*)[f(x_0, s) = x]\}; \\ X_{ac,m} &= X_m \cap X_{ac}; \\ f_{ac} &= f|_{X_{ac} \times E \to X_{ac}}. \end{aligned}$$

[6]Um estado x_f é dito ser acessível (ou alcançável) a partir de um estado x_i se existe uma sequência s tal que $f(x_i, s)$ seja definida e $f(x_i, s) = x_f$.

A notação $f|_{X_{ac}\times E\to X_{ac}}$ indica que f_{ac} é igual à função f, só que restrita ao subconjunto de X formado apenas pelos estados acessíveis de G.

- **Parte Coacessível** Um estado x de G é dito ser coacessível com respeito a X_m, ou simplesmente coacessível, se existe um caminho do diagrama de transição de estados de G partindo do estado x e atingindo um estado marcado. A operação de remoção de G de todos os seus estados não coacessíveis é denominada $CoAc(G)$. Tomar a parte coacessível de G significa construir

$$CoAc(G) := (X_{coac}, E, f_{coac}, x_{0,coac}, X_m)$$

sendo

$$X_{coac} = \{x \in X : (\exists s \in E^*)[f(x,s) \in X_m]\};$$
$$x_{0,coac} = \begin{cases} x_0, & \text{se } x_0 \in X_{coac}; \\ \text{não definido}, & \text{caso contrário}; \end{cases}$$
$$f_{coac} = f|_{X_{coac}\times E\to X_{coac}}.$$

Note que essa operação pode reduzir $\mathcal{L}(G)$, dado que pode apagar estados que são acessíveis a partir de x_0. Porém, a operação $CoAc$ não afeta $\mathcal{L}_m(G)$, a menos que o estado inicial seja apagado por ela. Se $G = CoAc(G)$, então diz-se que G é coacessível, e neste caso, $\mathcal{L}(G) = \overline{\mathcal{L}_m(G)}$.

- **Trim** Um autômato que é tanto acessível como coacessível é dito ser *trim*. A operação trim é definida como:

$$Trim(G) := CoAc[Ac(G)] = Ac[CoAc(G)].$$

- **Projeção** Seja G um autômato com conjunto de eventos E. Defina $E_s \subset E$ e a operação de projeção $P : E^* \to E_s^*$. As operações $P[\mathcal{L}(G)]$ e $P[\mathcal{L}_m(G)]$ podem ser implementadas em G substituindo-se os rótulos das transições rotuladas por eventos pertencentes a $E\backslash E_s$ por ε. É importante ressaltar, porém, que o resultado desta operação leva a um autômato não-determinístico.

- **Projeção Inversa** Seja G um autômato com conjunto de eventos $E_s \subset E_l$. Defina a operação de projeção $P : E_l^* \to E_s^*$. As operações $P^{-1}[\mathcal{L}(G)]$ e $P^{-1}[\mathcal{L}_m(G)]$ podem ser implementadas em G pela adição de autolaços contendo todos os eventos em $E_l\backslash E_s$ em todos os estados de G.

- **Complemento** Considere um autômato G, determinístico e trim, que marca a linguagem $L_m \subseteq E^*$ e gera a linguagem $\overline{L_m}$. Um autômato G^C que marque a linguagem $E^* \backslash L_m$ é construído por meio da operação $Comp$, definida em duas partes, quais sejam:

1. Obter um autômato G_{total} cuja linguagem gerada é igual a E^*. Para tanto, cria-se um novo estado x_d, e define-se a função de transição de G_{total}, f_{total}, da seguinte maneira:

$$f_{total}(x, e) = \begin{cases} f(x, e), & \text{se } e \in \Gamma(x); \\ x_d, & \text{caso contrário;} \end{cases}$$
$$f_{total}(x_d, e) = x_d \quad , \forall e \in E.$$

   o estado x_d é denominado estado "dump" (despejo), e é não-marcado.

2. Marcar todos os estados de G_{total} não-marcados e desmarcar todos os marcados fazendo:

$$G^C = Comp(G) := (X \cup \{x_d\}, E, f_{total}, x_0, (X \cup \{x_d\}) \backslash X_m).$$

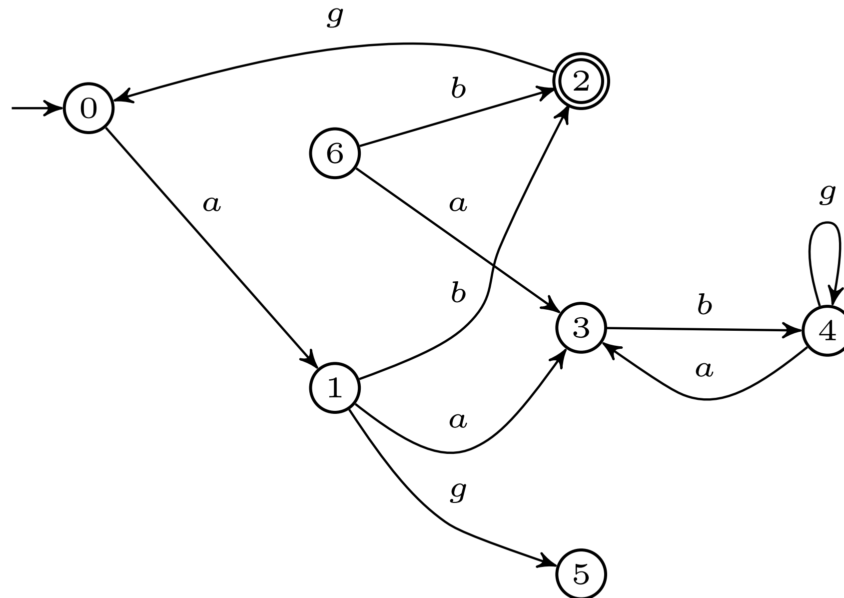
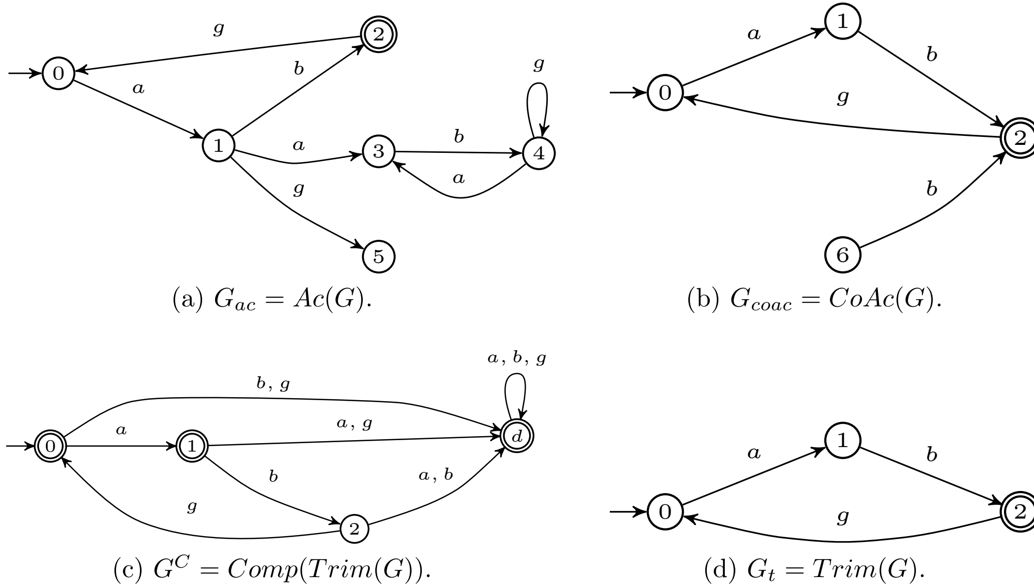


Figura 2.7: Autômato G para ilustração das operações unárias.

Exemplo 2.1. *Para exemplificar as operações unárias com autômatos, considere o autômato G cujo diagrama de transição de estados está representado na figura 2.7. Para se obter $G_{ac} = Ac(G)$ é suficiente remover o estado 6, e as duas transições atreladas a este, resultando no autômato da figura 2.8a. Para obter $G_{coac} = CoAc(G)$ é*

necessário identificar os estados que não são coacessível em relação ao único estado marcado 2: 3, 4 e 5. Esses estados, ao serem removidos juntamente com suas transições resultam no autômato da figura 2.8b. A operação $Trim(G)$ leva ao autômato G_t da figura 2.8d. Por fim, para obter $G^C = Comp[Trim(G)]$, cria-se, inicialmente, o estado x_d e a função de transição é completada para $E = \{a, b, g\}$, obtendo-se o autômato G_{total}. Após a troca das marcações dos estados do autômato G_{total} obtido no passo anterior, obtém-se o autômato G^C representado na figura 2.8c.



(a) $G_{ac} = Ac(G)$.

(b) $G_{coac} = CoAc(G)$.

(c) $G^C = Comp(Trim(G))$.

(d) $G_t = Trim(G)$.

Figura 2.8: Autômatos resultantes de operações unárias sobre o autômato da figura 2.7.

Operações de Composição Nessas operações são utilizados dois autômatos com o objetivo de gerar um terceiro autômato capaz de capturar características dos dois autômatos de entrada. Operações desse tipo facilitam substancialmente a modelagem de sistemas ao permitir que componentes sejam modelados individualmente, e, com sua posterior composição, possibilitar a síntese do comportamento do sistema completo.

As duas operações de composição definidas são: *produto*[7], denotada por $\times$, e *composição paralela*[8], denotada por $\|$. Pode-se pensar em $G_1 \times G_2$ e $G_1 \| G_2$, com conjunto de eventos E_1 e E_2, como duas formas de se conectar dois componentes que operam de maneira concorrente. A diferença entre essas operações está na maneira como os eventos privados de cada autômato são tratados.

Assim, para as operações a seguir, considere os autômatos acessíveis $G_1 = (X_1, E_1, f_1, \Gamma_1, x_{0,1}, X_{m,1})$ e $G_2 = (X_2, E_2, f_2, \Gamma_2, x_{0,2}, X_{m,2})$.

[7]Também chamada de *composição completamente síncrona*.
[8]Também chamada de *composição síncrona*.

- **Produto** O produto de G_1 e G_2 é o autômato

$$G_1 \times G_2 := Ac(X_1 \times X_2, E_1 \cup E_2, f_{1\times 2}, \Gamma_{1\times 2}, (x_{0,1}, x_{0,2}), X_{m,1} \times X_{m,2}),$$

sendo

$$f_{1\times 2}((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), f_2(x_2, e)), & \text{se } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2); \\ \text{não definido}, & \text{caso contrário}; \end{cases}$$
$$\Gamma_{1\times 2} = \Gamma_1(x_1) \cap \Gamma_2(x_2).$$

Na operação produto, as transições dos dois autômatos de entrada devem estar sempre sincronizadas em um evento comum. Em outras palavras, $G_1 \times G_2$ representa a interconexão completamente síncrona de G_1 e G_2, na qual um evento só ocorre se estiver ativo nos dois autômatos. Dessa forma, se um evento pertence ao conjunto de eventos de apenas um dos autômatos, as transições geradas por esse evento no autômato de entrada não estarão presentes no autômato de saída. Deve ser ressaltado que, contudo, esse evento, mesmo não pertencendo a ambos os autômatos, pertencerá ao conjunto dos eventos do autômato obtido na operação de composição.

Assim, no que diz respeito às linguagens resultantes tem-se que:

$$\mathcal{L}(G_1 \times G_2) = \mathcal{L}(G_1) \cap \mathcal{L}(G_2);$$
$$\mathcal{L}_m(G_1 \times G_2) = \mathcal{L}_m(G_1) \cap \mathcal{L}_m(G_2).$$

- **Composição Paralela** A composição paralela de G_1 e G_2 é o autômato

$$G_1 \| G_2 := Ac(X_1 \times X_2, E_1 \cup E_2, f_{1\|2}, \Gamma_{1\|2}, (x_{0,1}, x_{0,2}), X_{m,1} \times X_{m,2}),$$

em que

$$f_{1\|2}((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), f_2(x_2, e)), & \text{se } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2); \\ (f_1(x_1, e), x_2), & \text{se } e \in \Gamma_1(x_1) \backslash E_2; \\ (x_1, f_2(x_2, e)), & \text{se } e \in \Gamma_2(x_2) \backslash E_1; \\ \text{não definido}, & \text{caso contrário}; \end{cases}$$
$$\Gamma_{1\|2}(x_1, x_2) = [\Gamma_1(x_1) \cap \Gamma_2(x_2)] \cup [\Gamma_1(x_1) \backslash E_2] \cup [\Gamma_2(x_2) \backslash E_1].$$

Na operação de composição paralela, um evento comum (pertencente a $E_1 \cap E_2$) só pode ser executado se sua execução ocorrer simultaneamente nos dois autômatos de entrada. Por outro lado, os eventos privados de cada autômato de entrada (eventos em $(E_1 \backslash E_2)$ e $(E_2 \backslash E_1)$), não estão sujeito a nenhuma restrição, podendo ocorrer

livremente no autômato resultante, assim como ocorria no respectivo autômato de entrada.

Para analisar a linguagem resultante dessa operação, faz-se necessário definir uma operação de projeção sobre linguagem, para cada autômato de entrada, da seguinte forma:

$$P_i : (E_1 \cup E_2)^* \rightarrow E_i^* \text{ para } i = 1, 2.$$

Com isso, pode-se mostrar que as linguagens resultantes da composição paralela são dadas por:

$$\mathcal{L}(G_1 \| G_2) = P_1^{-1}[\mathcal{L}(G_1)] \cap P_2^{-1}[\mathcal{L}(G_2)];$$
$$\mathcal{L}_m(G_1 \| G_2) = P_1^{-1}[\mathcal{L}_m(G_1)] \cap P_2^{-1}[\mathcal{L}_m(G_2)].$$

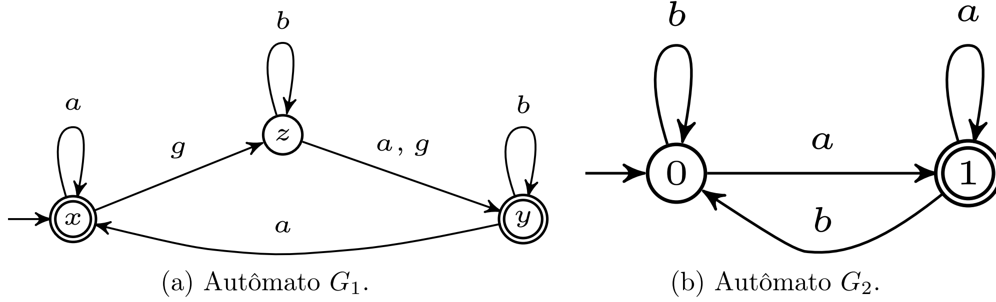


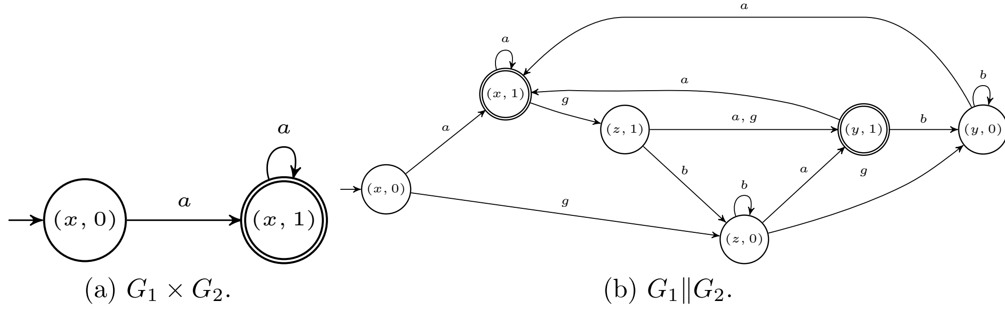
Figura 2.9: Autômatos para exemplo de operações de composição.

Exemplo 2.2. *Para exemplificar as operações de composição, considere o autômato G_1 definido de acordo com a figura 2.9a com $E_1 = \{a, b, g\}$, e o autômato G_2 definido de acordo com a figura 2.9b com $E_2 = \{a, b\}$. Os resultados das operações de produto e composição paralela podem ser vistos na figura 2.10. Note que o comportamento completamente síncrono de G_1 e G_2 é limitado ao evento a, conforme mostrado pela figura 2.10a, contudo a operação de composição paralela, além de incluir o comportamento síncrono, permite a ocorrência assíncrona dos autômatos, resultando na figura 2.10b.*

Conceito de Subautômato

Dados dois autômatos G_1 e G_2, dizemos que G_1 é um subautômato de G_2 se o diagrama de transição de estados de G_1 for um sub-grafo do diagrama de G_2. Formalmente, diz-se que G_1 é um sub-autômato de G_2, denotado por $G_1 \sqsubseteq G_2$, se:

$$f_1(x_{0,1}, s) = f_2(x_{0,2}, s), \ \forall s \in \mathcal{L}(G_1).$$



(a) $G_1 \times G_2$. (b) $G_1 \| G_2$.

Figura 2.10: Autômatos resultantes de operações de composição sobre figura 2.9.

Note que esta condição implica que $X_1 \subseteq X_2$, $x_{0,1} = x_{0,2}$, e $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$. Quando algum dos autômatos G_1 e G_2 possuir estados marcados, um requisito adicional para definir $G_1 \sqsubseteq G_2$ é que $X_{m,1} = X_{m,2} \cap X_1$; em outras palavras, a marcação de G_1 deve ser consistente com a marcação de G_2.

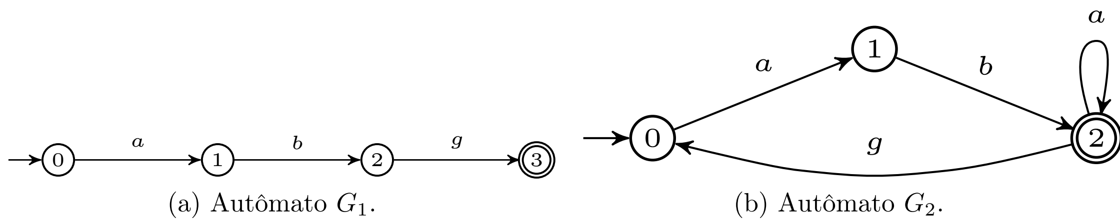
Motivado pelo benefício da correspondência direta de estados que o conceito de subautômato gera, considere, agora, o problema de modificar dois autômatos G_1 e G_2, tais que $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$, para obter novos autômatos G_1' e G_2' tais que $G_1' \sqsubseteq G_2'$. O algoritmo apresentado a seguir é um método genérico para resolver este problema.

Algoritmo 2.1 ([9]). *Sejam autômatos G_1 e G_2, tais que $\mathcal{L}(G_i) = L_i$, $i = 1, 2$, e $L_1 \subseteq L_2$.*

Passo 1: *Construa $G_1' = G_1 \times G_2$.*

Passo 2: *Construa o autômato G_1^{sl} adicionando auto-laços em $x_1 \in X_1$ para cada evento em $E_2 \backslash \Gamma_1(x_1)$.*

Passo 3: *Construa $G_2' = G_1^{sl} \times G_2$.*

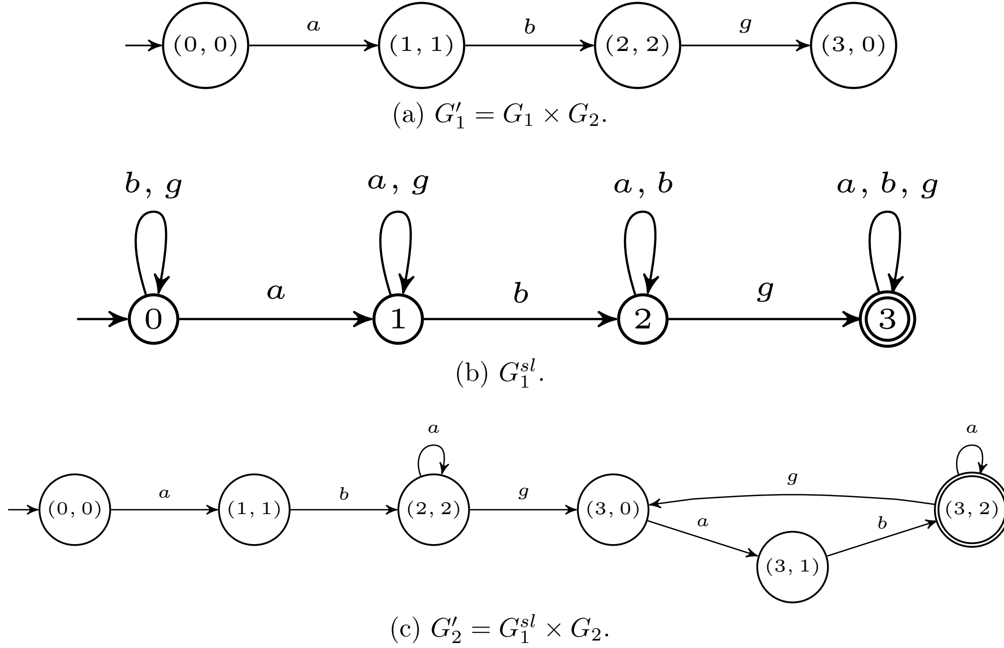


(a) Autômato G_1. (b) Autômato G_2.

Figura 2.11: Autômatos para exemplo de criação de subautômato.

Exemplo 2.3. *Para exemplificar a aplicação do algoritmo 2.1, sejam os autômatos G_1 e G_2 ilustrados nas figuras 2.11a e 2.11b, respectivamente. A execução do algoritmo constrói, primeiramente, o autômato G_1' (figura 2.12a). A adição de auto-laços dos eventos pertencentes a E_2, mas não ativos em G_1, resulta no autômato*

G_1^{sl} (figura 2.12b). Por fim, a operação produto de G_1^{sl} com G_2 permite obter G_2', conforme mostra a figura 2.12c. É fácil verificar que $G_1' \sqsubseteq G_2'$ e que $\mathcal{L}(G_1) = \mathcal{L}(G_1')$ e $\mathcal{L}(G_2) = \mathcal{L}(G_2')$.



(a) $G_1' = G_1 \times G_2$.

(b) G_1^{sl}.

(c) $G_2' = G_1^{sl} \times G_2$.

Figura 2.12: Execução do algoritmo 2.3 para criar $G_1' \sqsubseteq G_2'$ a partir de G_1 e G_2.

Observador

Suponha, agora, que o conjunto de eventos E de um autômato seja particionado como $E = E_o \dot{\cup} E_{uo}$, isto é, $E = E_o \cup E_{uo}$ e $E_o \cap E_{uo} = \emptyset$ e $E_{uo} \neq \emptyset$, sendo E_o um conjunto de eventos observáveis e E_{uo} um conjunto de eventos não-observáveis. Um evento é observável quando sua ocorrência puder ser registrada por meio de sensores ou quando estiver associado a comandos. Os eventos não-observáveis, por sua vez, designam aqueles eventos do sistema cuja ocorrência não pode ser observada por sensores (incluindo possíveis eventos de falhas) ou, embora haja sensores para registrá-los, esses eventos não podem ser observados em função da natureza distribuída do sistema. Quando $E = E_o \dot{\cup} E_{uo}$ tem-se o chamado autômato determinístico com eventos não-observáveis.

O comportamento dinâmico de um autômato determinístico com eventos não-observáveis pode ser descrito por um autômato determinístico, denominado observador, cujo conjunto de eventos é formado pelos eventos observáveis. Os estados do observador são todos os estados em que um autômato determinístico com eventos não-observáveis pode estar após a observação de uma sequência de eventos observá-

veis. O observador para G, denotado por $Obs(G)$, é definido da seguinte forma:

$$Obs(G) = (X_{obs}, E_o, f_{obs}, \Gamma_{obs}, x_{0,obs}, X_{m,obs}),$$

sendo $X_{obs} \in 2^X$ e $X_{m,obs} = \{B \in X_{obs} : B \cap X_m \neq \emptyset\}$. Para a definição de Γ_{obs}, $x_{0,obs}$ e f_{obs}, é necessário introduzir o conceito de alcance não-observável de um estado $x \in X$ (denotado por $UR(x)$):

$$UR(x) = \{y \in X : (\exists t \in E_{uo}^*)[f(x,t) = y]\}. \tag{2.6}$$

De forma análoga, o alcance não-observável de um conjunto $B \in 2^X$ é definido como

$$UR(B) = \bigcup_{x \in B} UR(x). \tag{2.7}$$

Usando (2.6) e (2.7), pode-se definir Γ_{obs}, $x_{0,obs}$ e f_{obs} no algoritmo a seguir.

Algoritmo 2.2 (Construção de observadores [17]).

Passo 1: *Defina* $x_{0,obs} = UR(x_0)$ *e faça* $X_{obs} = \{x_{0,obs}\}$ *e* $\tilde{X}_{obs} = X_{obs}$.

Passo 2: $\hat{X}_{obs} = \tilde{X}_{obs}$ *e* $\tilde{X}_{obs} = \emptyset$.

Passo 3: *Para cada* $B \in \hat{X}_{obs}$,

- $\Gamma_{obs}(B) = (\bigcup_{x \in B} \Gamma(x)) \cap E_o$;
- *Para cada* $e \in \Gamma_{obs}(B)$,
  $f_{obs}(B, e) = UR(\{x \in X : (\exists y \in B)[x = f(y, e)]\})$;
- $\tilde{X}_{obs} \leftarrow \tilde{X}_{obs} \cup f_{obs}(B, e)$.

Passo 4: $X_{obs} \leftarrow X_{obs} \cup \tilde{X}_{obs}$.

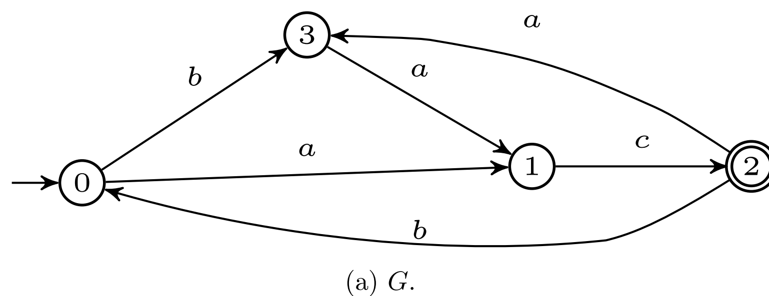
Passo 5: *Repita os passos 2 a 4 até que toda a parte acessível de* $Obs(G)$ *tenha sido construída.*

Passo 6: $X_{m,obs} = \{B \in X_{obs} : B \cap X_m \neq \emptyset\}$.

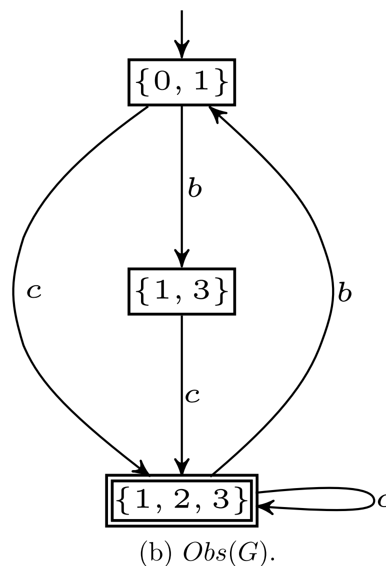
A ideia do algoritmo 2.2 é calcular o alcance não-observável para cada estado de G alcançado por um evento observável. Assim, no passo 1 calcula-se o alcance não-observável do estado inicial x_0 formando o estado inicial do observador. No passo 3 calculam-se os conjuntos dos eventos ativos dos estados do observador obtidos no passo anterior ou na iteração anterior (o primeiro se refere ao alcance observável do estado inicial e o último aos estados de G alcançados por meio de eventos observáveis juntamente com os respectivos alcances não-observáveis). Além disso são calculados

os próximos estados do observador, que correspondem aos alcances não-observáveis dos estados de G alcançados a partir do estado atual do observador por meio de eventos observáveis. Essa sequência é repetida até que todos os estados acessíveis do observador tenham sido encontrados.

A partir da construção do observador, pode-se concluir que a linguagem gerada por $Obs(G)$ é a projeção da linguagem de G sobre o conjunto de eventos observáveis, isto é, $\mathcal{L}(Obs(G)) = P_o[\mathcal{L}(G)]$. É possível notar, também, que o algoritmo 2.2 possui complexidade exponencial no número de estados, devendo ser aplicado com precaução em sistemas com elevado número de transições e estados.



(a) G.



(b) $Obs(G)$.

Figura 2.13: Exemplo de construção do observador.

Exemplo 2.4. *Para ilustrar a construção de observadores, considere o autômato G da figura 2.13a e suponha que o evento a seja não-observável, isto é, tem-se que $E = \{a, b, c\}$, $E_o = \{b, c\}$ e $E_{uo} = \{a\}$. Assim, quando esse autômato inicia sua operação, não é possível precisar se ele ainda está no estado inicial $x_0 = 0$ ou se mudou para o estado $x = 1$, uma vez que a ocorrência do evento a não pode ser registrada; daí o estado inicial do observador mostrado na figura 2.13b ser $\{0, 1\}$. Caso o próximo evento observável a ocorrer seja o evento b, pode-se,*

então, afirmar que o autômato estará, inicialmente no estado $x = 3$, porém, como o evento a é não-observável, o autômato pode mudar para o estado $x = 1$, sem que essa mudança seja percebida; por conseguinte, a transição rotulada pelo evento b no observador leva do estado inicial para o estado $\{1, 3\}$. Há ainda transições rotuladas pelo evento c que levam ao estado $\{1, 2, 3\}$ do observador, partindo tanto do estado inicial quanto do estado $\{1, 3\}$, uma vez que o evento c é o único evento observável pertencente aos conjuntos dos eventos ativos dos estados de G que compõem esses dois estados, e os estados $x = 3$ e $x = 1$ pertencem ao alcance não-observável do estado $x = 2$. Finalmente, quando a ocorrência do evento c for registrada, o observador irá permanecer no estado $\{1, 2, 3\}$. Contudo, se o evento b ocorrer, o observador retornará ao seu estado inicial.

2.2 Controle Supervisório

O conhecimento em detalhes de como um sistema funciona leva, naturalmente, ao ponto de desejarmos modificar o seu comportamento, seja por questões de segurança ou mesmo com o objetivo de produzir um resultado final esperado por nós. Este conceito é traduzido na engenharia pela teoria de controle.

Na teoria de controle desenvolvida para sistemas contínuos, o conceito mais utilizado para fazer um sistema se comportar de acordo com uma dada referência (ou *set-point*) é o de realimentação. A comparação da referência com o que está efetivamente ocorrendo no sistema (por meio da medição de alguma variável de saída) resulta na quantificação de um erro. Dessa maneira, se a magnitude desse erro for constantemente utilizada para modificar proporcionalmente alguma variável de entrada no sistema, é possível, então, levar a saída do sistema para o nível desejado de referência.

Como resultado dos estudos de W. M. Wonham e P. J. Ramadge [1], foi criada a fundação para a aplicação do conceito de realimentação a SEDs, denominada *Teoria de Controle Supervisório*. Nessa teoria, considera-se que um SED é modelado por um autômato G, sendo E o conjunto de eventos desse autômato. O autômato G representa o comportamento não controlado do sistema, possuindo comportamentos (sequências) que se deseja modificar. Entende-se por modificar o comportamento de um autômato G, à ação de restringir sua linguagem para algum sub-conjunto dela, chamado de especificação (autômato H)[9]. Para efetivamente realizar essa modificação, será definido um supervisor (controlador do sistema), denotado por S. Quando em operação, S "observa" a ocorrência dos eventos em G e "diz" a G quais dos eventos ativos são permitidos, pela capacidade de S de desabilitar a ocorrência de certos

[9]Não é objetivo deste trabalho a síntese de especificações. Metodologias para este fim podem ser encontradas em [9] e [15].

eventos em G. Isso na prática limita as linguagens do autômato G às linguagens do autômato H. No entanto, existem eventos em G cuja ocorrência não pode ser observada por S, seja por falha ou mesmo pela inexistência de um sensor para esse fim. Esses eventos são chamados de não-observáveis. Podem, também, existir eventos cuja ocorrência não pode ser desabilitada, como para o caso de sensores, que tem apenas a função de leitura de variáveis, e não de atuação. Esses eventos são chamados de não-controláveis.

2.2.1 SED controlado considerando observação total

Considere um SED, cujo comportamento é modelado pelo par de linguagens L e L_m, em que $L = \overline{L}$ é o conjunto de todas as sequências de eventos que o SED pode gerar e L_m é um subconjunto de L usado para representar o término de uma tarefa ou uma ação específica que se queira destacar, sendo ambas definidas sobre um mesmo conjunto de eventos E. Pode-se dizer, então, que um autômato $G = (X, E, f, \Gamma, x_0, X_m)$ modela o referido SED quando $\mathcal{L}(G) = L$ e $\mathcal{L}_m(G) = L_m$.

De maneira semelhante, considere um autômato de especificação H ($\mathcal{L}(H) = L_a \subseteq \mathcal{L}(G)$ e $\mathcal{L}_m(H) = L_{am} \subseteq \mathcal{L}_m(G)$) que contenha todas as sequências desejadas de G, mas que exclua as sequências cuja ocorrência se deseja impedir.

Assim, a formalização do problema de controle supervisório para um autômato G e uma especificação H é: obtenha um supervisor S para interagir com G através de realimentação (figura 2.14), resultando em um autômato $S|G$ (lê-se S controlando G) com as seguintes características:

$$\begin{aligned} \mathcal{L}(S|G) &= L_a \subset L; \\ \mathcal{L}_m(S|G) &= L_{am} = \mathcal{L}(S|G) \cap \mathcal{L}_m(G) = L_a \cap L_m. \end{aligned}$$

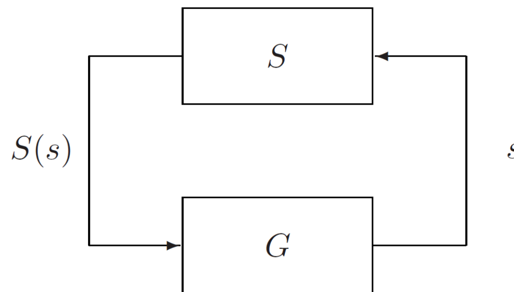


Figura 2.14: Laço de controle supervisório realimentado $S|G$.

Como pode ser observado na figura 2.14, o paradigma de controle estabelecido pela realimentação faz com que a função de transição de G seja controlada por S, no sentido de que os eventos de G possam ser dinamicamente habilitados ou desabilitados por S. Assim, o supervisor S pode ser interpretado como uma função

cujo domínio é a linguagem gerada por G, e cuja imagem é o conjunto potência de E, isto é:

$$S : \mathcal{L}(G) \to 2^E.$$

Com isso, a função dos eventos ativos da malha fechada $(S|G)$, denotada por Γ_N, pode ser escrita como

$$\Gamma_N[f(x_0, s)] = S(s) \cap \Gamma[f(x_0, s)],$$

o que significa dizer que G não pode executar um evento no estado $f(x_0, s)$ se esse evento não pertencer a $S(s)$.

Com o entendimento desses conceitos, é possível, então, formalizar as linguagens de $S|G$.

Definição 2.17. *(Linguagens gerada e marcada por $S|G$) A linguagem gerada por um autômato $S|G$ pode ser recursivamente definida como*

(i) $\varepsilon \in \mathcal{L}(S|G)$;

(ii) $(s \in \mathcal{L}(S|G))$ *e* $(s\sigma \in \mathcal{L}(G))$ *e* $(\sigma \in S(s)) \Leftrightarrow s\sigma \in \mathcal{L}(S|G)$.

A linguagem marcada por esse mesmo autômato será definida por

$$\mathcal{L}_m(S|G) := \mathcal{L}(S|G) \cap \mathcal{L}_m(G).$$

Claramente, $\mathcal{L}(S|G) \subseteq \mathcal{L}(G)$ e é prefixo-fechada por definição. Já $\mathcal{L}_m(S|G)$ é composta exatamente pelas sequências marcadas de G que sobrevivem à ação de controle de S. As seguintes relações de inclusão podem, então, ser verificadas[10]:

$$\emptyset \subseteq \mathcal{L}_m(S|G) \subseteq \overline{\mathcal{L}_m(S|G)} \subseteq \mathcal{L}(S|G) \subseteq \mathcal{L}(G).$$

A solução do problema de controle supervisório quando todos os eventos são controláveis e observáveis sempre existe e é relativamente simples de ser obtida. Porém, em geral, não é possível afirmar que todos os eventos são controláveis. Em outras palavras, o conjunto E do autômato G é particionado da seguinte forma:

$$E = E_c \dot{\cup} E_{uc},$$

em que $E_c \subseteq E$ é o subconjunto de eventos controláveis, $E_{uc} \subseteq E$ é o subconjunto de eventos não-controláveis e $\dot{\cup}$ denota partição, isto é, $E_c \cap E_{uc} = \emptyset$ e $E_c \cup E_{uc} =$

[10]Não é objetivo deste trabalho o estudo detalhado de $\mathcal{L}_m(S|G)$ e consequentemente de bloqueio em $S|G$. Seu estudo e aplicação na teoria de Controle Supervisório podem ser encontrados na literatura ([9] e [15]).

E. Uma vez que, por definição, os eventos de E_{uc} não podem ser desabilitados, eles necessariamente pertencem a $S(s)$. Com isso é possível definir que um dado supervisor S será *admissível* quando

$$E_{uc} \cap \Gamma[f(x_0, s)] \subseteq S(s),$$

o que significa dizer que $S(s)$ não pode desabilitar eventos ativos não-controláveis. Todos os supervisores S considerados neste trabalho são admissíveis.

Considere, agora, o problema da síntese de um supervisor S admissível. Nesse caso, o objetivo é encontrar uma função de supervisão S que limite o comportamento de G ao especificado em H, com a restrição $E = E_c \dot{\cup} E_{uc}$. Porém, não existe nenhuma garantia de existência de S, uma vez que podem existir eventos em E_{uc} que precisem ser desabilitados em algum momento da execução para que se obtenha a especificação H. Para tanto, é preciso primeiramente garantir a existência de S para então efetuar a sua síntese. O conceito que define essa existência é o de *controlabilidade.*

Definição 2.18. *(Controlabilidade) Sejam K e $M = \overline{M}$ linguagens definidas sobre um conjunto de eventos E. Seja E_{uc} um sub-conjunto de E. Diz-se que K é controlável em relação a M e E_{uc} se*

$$\overline{K} E_{uc} \cap M \subseteq \overline{K}.$$

É possível, então, aplicar o conceito de controlabilidade no projeto de supervisores admissíveis utilizando o seguinte teorema.

Teorema 2.1. *(Teorema da controlabilidade) Seja $G = (X, E, f, \Gamma, x_0, X_m)$, em que $E_{uc} \subseteq E$ é o conjunto de eventos não-controláveis, e $M = \overline{M} = \mathcal{L}(G)$. Seja a especificação de linguagem $K \subseteq M$, sendo $K \neq \emptyset$. Então, existe um supervisor S, tal que $\mathcal{L}(S|G) = \overline{K}$ se e somente se*

$$\overline{K} E_{uc} \cap M \subseteq \overline{K}.$$

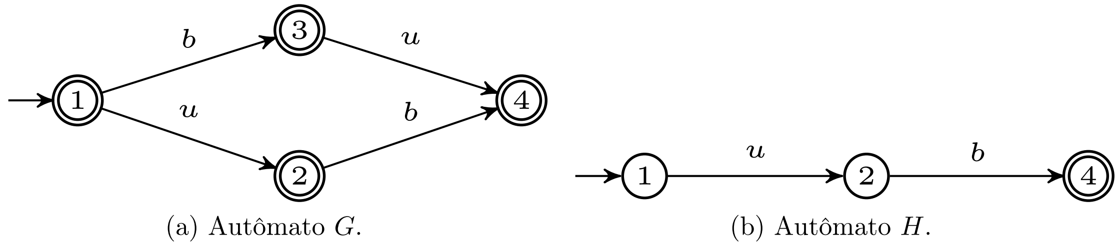
Em palavras, o teorema da controlabilidade pode ser enunciado da seguinte forma:

> *"Se eu não posso impedir que alguma sequência não desejável ocorra, então ela tem que fazer parte da especificação."*

Por definição, a controlabilidade é uma propriedade do fecho do prefixo de uma linguagem, ou seja, K é controlável se, e somente se, $\overline{K}$ for controlável. Note que a controlabilidade é uma propriedade independente da observação dos eventos de E.

Dentre as propriedades que a controlabilidade apresenta, é de especial importância ressaltar que ela é preservada na operação de união, ou seja, se K_1 e K_2 são controláveis, então $K_1 \cup K_2$ também será controlável.

Exemplo 2.5. *Para exemplificar o conceito de controlabilidade, considere os autômatos* G *e* H *definidos na figura 2.15, em que* $E = \{u, b\}, M = \mathcal{L}(G) = \overline{\{ub, bu\}}$ *e* $K = \mathcal{L}_m(H) = \{ub\}$*. Seja* $E_{uc} = \{b\}$*. Logo, no estado inicial* 1*, o supervisor precisaria desabilitar* b *para que apenas* u *pudesse ocorrer, e assim satisfazer a especificação. Porém, como* b *é não-controlável, isto não é possível. Logo, diz-se que* K *(e também* $\overline{K}$*) não é controlável em relação a* M *e* E_{uc}*. Por outro lado, se* $E_{uc} = \{u\}$*, então o sistema torna-se controlável.*



(a) Autômato G. (b) Autômato H.

Figura 2.15: Exemplo de malha fechada formada por modelo (G) e especificação (H).

Como consequência da controlabilidade de uma linguagem K, em relação a M e E_{uc}, é possível verificar que, quando K é controlável em relação a M e E_{uc}, então o supervisor S existe e é definido por:

$$S(s) = [E_{uc} \cap \Gamma(f(x_0, s))] \cup \{\sigma \in E_c : s\sigma \in \overline{K}\}.$$

Linguagem Controlável Suprema

O conceito de controlabilidade expõe a principal dificuldade associada à síntese de supervisores, que é a incapacidade de se satisfazer a especificação quando a linguagem especificada não for controlável em relação à linguagem gerada pela planta e aos eventos não-controláveis. Supondo que a planta a ser controlada não pode ser modificada, para tornar possível a síntese de um supervisor é necessário alterar, então, a especificação, retirando as sequências que levam à violação da controlabilidade. Com isso em mente, é possível definir a classe formada por todas as sub-linguagens controláveis, como:

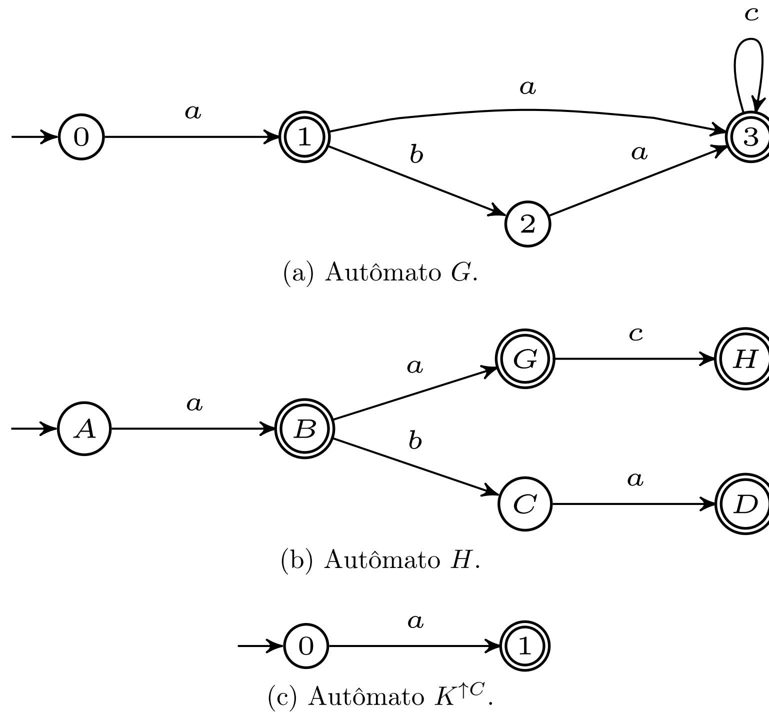
$$\mathcal{C}_{in}(K) := \{L \subseteq K \,:\, \overline{L}E_{uc} \cap M \subseteq \overline{L}\}.$$

A partir da classe de linguagens $\mathcal{C}_{in}(K)$ e do fato de que o objetivo é retirar o menor número possível de sequências da especificação, pode-se mostrar que a

linguagem controlável suprema é obtida, por meio do supremo do conjunto $\mathcal{C}_{in}(K)$, da seguinte forma:

$$K^{\uparrow C} := \bigcup_{L \in \mathcal{C}_{in}(K)} L.$$

A notação $\uparrow$ é usada para indicar que esta linguagem está dentro da especificação K, o que pode ser verificado pelo fato de ser resultado da união de linguagens que também são sub-conjuntos de K. Por sua vez, a notação C é utilizada para indicar o resultado de que $K^{\uparrow C}$ é sempre controlável com respeito a M e E_{uc}. Esse resultado tem origem na preservação da controlabilidade sobre a operação de união, e sua consequente extensão para um número infinito de uniões.



(a) Autômato G.

(b) Autômato H.

(c) Autômato $K^{\uparrow C}$.

Figura 2.16: Autômatos do exemplo 2.6.

Exemplo 2.6. *Como exemplo de obtenção da linguagem controlável suprema, considere os autômatos G e H definidos na figura 2.16, nos quais $E = \{a, b, c\}$ e $E_{uc} = \{c\}$ e são tais que $M = \mathcal{L}(G)$ e $K = \mathcal{L}_m(H)$. Analisando o sistema, é possível observar que, embora a especificação contenha a sequência aac, a planta permite a ocorrência posterior de infinitos eventos c. Como c não pode ser desabilitado por ser não-controlável, então a propriedade de controlabilidade não é satisfeita. Visando modificar a especificação para atender a esta propriedade, é necessário remover de K todas as sequências que permitam a ocorrência continuada de c, ou seja, as sequências aac, aa e aba. Com isso chega-se ao autômato da figura 2.16c que marca a*

linguagem controlável suprema $K^{\uparrow C}$.

2.2.2 Controle de SEDs com observação parcial

Considere agora o caso em que o autômato G é um SED com observação parcial. Conforme visto na definição 2.14, isto significa dizer que:

$$E = E_o \dot{\cup} E_{uo},$$

em que $E_o \subseteq E$ é o subconjunto de eventos observáveis e $E_{uo} \subseteq E$ é o subconjunto de eventos não-observáveis[11].Com isso, o supervisor S não será mais capaz de observar todos os eventos que ocorrerem em G, o que leva a uma modificação na estrutura de realimentação: a introdução da operação de projeção $P : E^* \to E_o^*$ entre G e S, que pode ser vista na figura 2.17. Note que para refletir essa modificação, o supervisor passa a ser denotado por S_P, uma vez que não mais age sobre s e sim sobre $P(s)$.

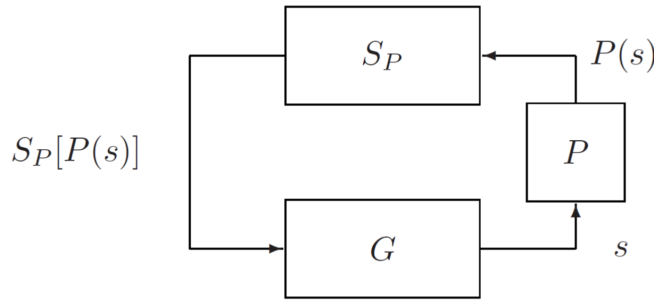


Figura 2.17: Laço de controle supervisório realimentado com observação parcial.

Devido à existência da projeção P, o supervisor não é capaz de distinguir entre duas sequências s_1 e s_2 que tenham a mesma projeção, isto é, $P(s_1) = P(s_2)$. Para essas sequências $s_1, s_2 \in \mathcal{L}(G)$, o supervisor irá necessariamente tomar a mesma ação de controle $S_P[P(s_1)]$. Para capturar esse comportamento, é preciso definir uma nova função de supervisão, qual seja:

$$S_P : P[\mathcal{L}(G)] \to 2^E,$$

em que S_P é chamado de *Supervisor-P*. Isso significa que a ação de controle só será modificada após a ocorrência de eventos observáveis, isto é, quando $P(s)$ se modifica.

Assim como feito para o caso de controlabilidade parcial, é preciso garantir que um Supervisor-P seja admissível, isto é, que não desabilite nenhum evento ativo não controlável. Seja, então, a ação de controle $S_P(t)$, para $t \in P[\mathcal{L}(G)]$, aplicada imediatamente após a última execução de um evento controlável em G. Faça $t = t'\sigma$,

[11]Ressalta-se que não existe nenhuma relação de inclusão entre os conjuntos E_c e E_o, ou seja, ambos são definidos independentemente a partir de E.

em que $\sigma \in E_o$. Então $S_P(t)$ é a ação de controle aplicada a todas as sequências pertencentes a $\mathcal{L}(G) \cap P^{-1}(t')\{\sigma\}$, assim como para todas as suas continuações não-observáveis. Defina

$$L_t = P^{-1}(t')\{\sigma\}(S_P(t) \cap E_{uo})^* \cap \mathcal{L}(G).$$

Note que L_t contém todas as sequências em $\mathcal{L}(G)$ que estão efetivamente sujeitas à ação de controle $S_P(t)$. Como um supervisor é admissível se ele não desabilitar eventos ativos não-controláveis, é possível concluir que S_P será admissível se $\forall t = t'\sigma \in P[\mathcal{L}(G)]$,

$$E_{uc} \cap [\bigcup_{s \in L_t} \Gamma(f(x_0, s))] \subseteq S_P(t),$$

na qual o termo entre colchetes representa todas as continuações viáveis de todas as sequências às quais $S_P(t)$ é a ação de controle correspondente.

Uma vez que estamos interessados em utilizar essa estrutura para modificar a linguagem de um sistema, é preciso definir quais serão as linguagens obtidas para o caso de observação parcial.

Definição 2.19. *(Linguagens gerada e marcada por $S_P|G$) A linguagem gerada por um autômato $S_P|G$ pode ser recursivamente definida como*

(i) $\varepsilon \in \mathcal{L}(S_P|G)$;

(ii) $(s \in \mathcal{L}(S_P|G))$ *e* $(s\sigma \in \mathcal{L}(G))$ *e* $(\sigma \in S_P[P(s)]) \Leftrightarrow s\sigma \in \mathcal{L}(S_P|G)$.

A linguagem marcada por $S_P|G$ será definida por

$$\mathcal{L}_m(S_P|G) := \mathcal{L}(S_P|G) \cap \mathcal{L}_m(G).$$

Ressalta-se que $\mathcal{L}(S_P|G)$ e $\mathcal{L}_m(S_P|G)$ estão definidas sobre E, e não sobre E_o, dado que representam o comportamento da malha fechada como um todo.

Observabilidade

Considere agora o problema da síntese de um supervisor-P admissível, no qual o objetivo é encontrar uma função de supervisão S que limite o comportamento de G ao especificado em H. Porém, além da restrição de controlabilidade $E = E_c \dot{\cup} E_{uc}$, considere também a restrição dada por $E = E_o \dot{\cup} E_{uo}$. Assim, podem existir sequências diferentes, mas com mesmas projeções, que levam à necessidade de inibir a ocorrência de um evento controlável, ao mesmo tempo que a especificação exija que o mesmo ocorra. Dado que isso inviabiliza a síntese de um supervisor-P, é preciso primeiramente garantir que essa situação não ocorra. O conceito utilizado para esse fim é o da *observabilidade*.

Definição 2.20. *(Observabilidade) Seja $G = (X, E, f, \Gamma, x_0, X_m)$, em que $E_c \subseteq E$ é o conjunto de eventos controláveis, $E_o \subseteq E$ é o conjunto de eventos observáveis e suponha que $M = \overline{M} = \mathcal{L}(G)$. Seja a especificação de linguagem $K \subseteq M$, em que $K \neq \emptyset$. Então, K é observável em relação a M, E_o, e E_c, se $\forall s \in \overline{K}$ e $\forall \sigma \in E_c$,*

$$(s\sigma \notin \overline{K}) \text{ e } (s\sigma \in M) \Rightarrow P^{-1}[P(s)]\sigma \cap \overline{K} = \emptyset.$$

Em palavras, a observabilidade pode ser definida da seguinte forma:

> *"Se não for possível diferenciar duas sequências, então essas sequências devem requerer a mesma ação de controle."*

Por definição, assim como a controlabilidade, a observabilidade é uma propriedade do fecho do prefixo de uma linguagem, ou seja, K é observável se, e somente se, $\overline{K}$ for observável. Note que se K for não-observável, então K contém s e s' tais que $P(s) = P(s')$, e $s\sigma \notin \overline{K}$, enquanto $s'\sigma \in \overline{K}$. Desse modo, não existe um supervisor-P capaz de obter a linguagem K, dado que um supervisor-P não é capaz de diferenciar a sequência s de s' que exigem ações de controle distintas para o evento σ.

É importante ressaltar que a observabilidade não é preservada sob união, o que formalmente pode ser dito como: se K_1 e K_2 são observáveis, então $K_1 \cup K_2$ não será necessariamente observável.

Exemplo 2.7. *Para ilustrar o conceito de observabilidade, considere os autômatos G e H definidos na figura 2.15, em que $E = \{u, b\}, M = \mathcal{L}(G) = \overline{\{ub, bu\}}$ e $K = \mathcal{L}_m(H) = \{ub\}$. Seja $E_{uo} = \{u\}$ e $E_{uc} = \{u\}$. Então, no estado inicial 1, um supervisor-P teria que inibir b para não alcançar o estado 3. Ao mesmo tempo, ele teria que habilitar b para permitir chegar no estado 4, uma vez que a ocorrência de u não é observável. Logo pode-se concluir que K não é observável com relação a M, E_o e E_c. Por outro lado, se $E_{uc} = \{b\}$ então o sistema torna-se observável.*

Uma observação importante acerca da definição 2.20, é que como pode ser visto em [18], na definição inicial da literatura não se utilizou a condição de $\sigma \in E_c$. Porém, em [10], [11] e [9], exige-se que $\sigma \in E_c$ para que a propriedade de observabilidade não se sobreponha à propriedade de controlabilidade. Essa sobreposição ocorre porque, caso $s \in \overline{K}$, $s\sigma \in M$ e $\sigma \in E_{uc}$, então a controlabilidade exige que $s\sigma \in \overline{K}$, o que implica em observabilidade. Logo, se K é controlável, então a análise de observabilidade para $\sigma \in E_{uc}$ não precisa ser feita, pois é garantida.

Com base neste fato, é possível então enunciar um teorema que relaciona as duas propriedade fundamentais de controle supervisório:

Teorema 2.2. *(Controlabilidade e Observabilidade) Seja $G = (X, E, f, \Gamma, x_0, X_m)$, em que $E_{uc} \subseteq E$ é o conjunto de eventos não-controláveis, e $E_o \subseteq E$ é o conjunto*

de eventos observáveis. Seja $P : E^* \rightarrow E_o^*$, $K \subseteq \mathcal{L}(G)$, $K \neq \emptyset$. Então, existe S_P tal que

$$\mathcal{L}(S_P|G) = \overline{K}$$

se e somente se K é controlável em relação a $\mathcal{L}(G)$ e E_{uc}, e observável em relação a $\mathcal{L}(G)$, E_o e E_c.

Normalidade

Tendo em vista que a observabilidade não é preservada em relação à união, um conceito similar, porém mais restritivo, foi introduzido: a *normalidade*.

Definição 2.21. *(Normalidade) Seja $M = \overline{M} \subseteq E^*$, $K \subseteq M$, e a projeção natural $P : E^* \rightarrow E_o^*$. Diz-se que a linguagem K é normal em relação a M e P se*

$$\overline{K} = P^{-1}[P(\overline{K})] \cap M.$$

Em palavras, a normalidade pode ser definida como:

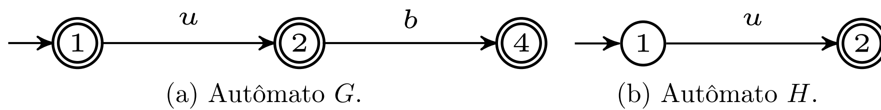
> *"Não podem existir duas sequências com mesma projeção, onde uma faz parte da especificação e a outra não."*

Ou, ainda:

> *"O fecho do prefixo de K pode ser recuperado pela interseção, com M, da projeção inversa de sua projeção."*

Assim como para a controlabilidade e observabilidade, a normalidade também é uma propriedade do fecho de prefixo de uma linguagem. Logo, K é normal se, e somente se, $\overline{K}$ for normal.

Exemplo 2.8. *Para ilustrar o conceito de normalidade, considere os autômatos G e H definidos na figura 2.18, em que $E = \{u, b\}, M = \mathcal{L}(G) = \overline{\{ub\}}$ e $K = \mathcal{L}_m(H) = \{u\}$, e seja $E_o = \{u\}$. Então, para qualquer $E_c \subseteq E$ o sistema será observável, já que se b for controlável, bastará desabilitá-lo, e caso b não seja controlável, a definição não leva em consideração. Porém, esse mesmo sistema não é normal, já que a sequência $ub \notin \overline{K}$.*



(a) Autômato G. (b) Autômato H.

Figura 2.18: Exemplo simplificado de malha fechada formada por modelo (G) e especificação (H).

Apesar de ser uma condição mais forte do que a observabilidade, a normalidade possui algumas vantagens que motivam seu estudo e utilização. A primeira e mais direta é que, sendo uma versão mais restrita da observabilidade, é possível demonstrar que a normalidade implica observabilidade. Um modo intuitivo de pensar nessa relação é considerar que uma linguagem será observável se, para todas as possíveis confusões de sequências (sequências diferentes com mesma projeção observável) onde uma continuação faça uma delas sair da especificação e a outra permanecer, nenhuma requer ações de controle distintas. Já a normalidade é mais restritiva, e não permite que nenhuma confusão desse tipo ocorra. Formalmente, isso pode ser escrito como:

Corolário 2.1. *(Normalidade e Observabilidade) Se $K \subseteq M$ for normal em relação a M e $P : E^* \to E_o^*$, então K será observável em relação a M, E_o e E_c, $\forall E_c \subseteq E$.*

Indo além nas motivações do estudo da normalidade estão o fato de que suas características se preservam sobre a operação de união, e por fim o fato de que, sob certas condições, a normalidade reúne informações sobre controlabilidade e observabilidade simultaneamente, conforme o corolário a seguir.

Corolário 2.2. *(Normalidade, Observabilidade e Controlabilidade) Suponha que $E_c \subseteq E_o$. Se K for controlável (em relação a M e E_{uc}) e observável (em relação a M, E_o e E_c), então K será normal (em relação a M e $P : E^* \to E_o^*$).*

Linguagem Normal Suprema

De maneira semelhante ao que foi feito na definição de $K^{\uparrow C}$, quando uma linguagem não for normal, torna-se importante, para permitir a síntese de um supervisor, a definição de uma sublinguagem, denotada por $K^{\uparrow N}$. Essa linguagem é chamada de suprema ($\uparrow$) pois provoca a menor modificação na especificação K. Ao mesmo tempo, ela é normal como consequência da preservação da propriedade da normalidade sob união.

Exemplo 2.9. *Como exemplo de obtenção da linguagem normal suprema, reconsidere o exemplo 2.6, porém, adicione, agora, a informação $E_{uo} = \{c\}$. Note que a sequência aa é a projeção tanto da sequência especificada aac como de qualquer continuação dessa sequência com o evento c. Do mesmo modo, a sequência aba é a projeção tanto da sequência especificada aba como de qualquer continuação dessa sequência com o evento c. Assim, conclui-se que a propriedade de normalidade não é respeitada. Logo, objetivando modificar a especificação para atender a essa propriedade, é necessário remover de $\mathcal{L}_m(H)$ todas as sequências que permitam a ocorrência continuada de c, ou seja, as sequências aac, aa e aba. Com isso chega-se à linguagem normal suprema exibida na figura 2.16c.*

Linguagem Controlável e Normal Suprema

Considere agora o problema global da síntese de um supervisor que deve respeitar a controlabilidade e a normalidade[12]. Quando a especificação não atende uma dessas propriedades, será necessário obter $K^{\uparrow C}$ e/ou $K^{\uparrow N}$. Todavia, nenhuma das operações sobre a linguagem K garante a preservação da outra. Desse modo, para obter uma linguagem de especificação que seja globalmente controlável e normal, define-se a linguagem controlável e normal suprema, denotada por $K^{\uparrow CN}$.

Exemplo 2.10. *Como visto nos exemplos 2.6 e 2.9, para o sistema formado por G e H da figura 2.16, com $E_{uc} = E_{uo} = \{c\}$, tem-se que $K^{\uparrow C} = K^{\uparrow N} = \{a\}$. Logo, pode-se concluir que essa linguagem também é controlável e normal suprema ($K^{\uparrow CN}$).*

2.2.3 Controle Supervisório Descentralizado

A teoria de controle supervisório vista até aqui considerou a arquitetura na qual uma planta com linguagem M é controlada por um único supervisor S de maneira a restringir o comportamento do sistema à linguagem especificada K (conhecida como arquitetura centralizada). Todavia, motivados pela existência de sistemas distribuídos, onde diversos controladores podem existir ao longo de uma única linha de produção, cada qual com um dado conjunto de sensores e atuadores, é possível definir uma arquitetura de controle supervisório descentralizado ([9]), ilustrada pela figura 2.19 para dois agente supervisores.

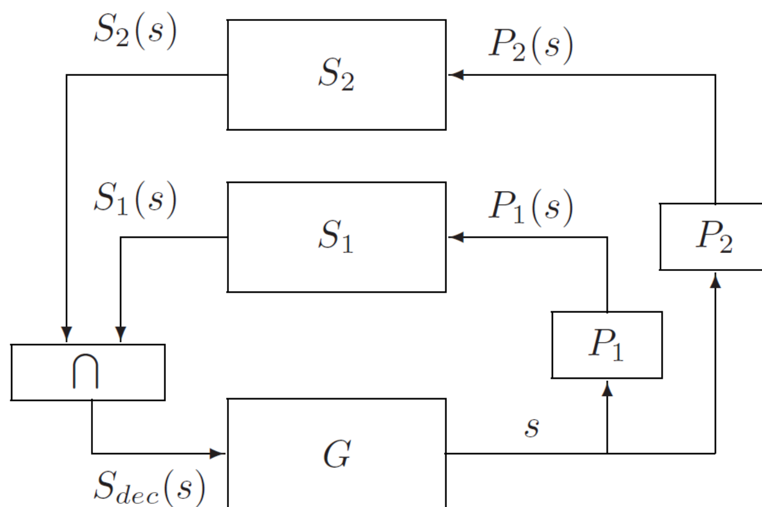


Figura 2.19: Laço de controle supervisório descentralizado e realimentado com dois agentes supervisores.

[12]Considera-se a normalidade no lugar da observabilidade devido à falta de características desejáveis nessa última.

O problema de controle supervisório descentralizado pode ser formalizado da seguinte maneira: suponha que existam n agentes supervisores com observabilidade parcial do conjunto de eventos E, e defina as projeções $P_i : E^* \rightarrow E^*_{o,i}$, para $i \in \{1, 2, ..., n\}$, em que $E_{o,i} \subseteq E$ é o conjunto dos eventos observáveis pelo i-ésimo agente supervisor S_i. Desse modo, a ação de controle do i-ésimo supervisor, para $s \in \mathcal{L}(G)$, é definida por:

$$S_i(s) = S_{P_i}[P_i(s)].$$

As ações de controle $S_i(s)$ precisam, então, ser combinadas para gerar uma ação de controle resultante $S_{dec}(s)$ a ser aplicada à planta G. Note que, a presença de um "módulo centralizador único" na figura 2.19, responsável por realizar a fusão das ações de controle e gerar S_{dec}, não implica sua existência na implementação do sistema. No caso, ele apenas simboliza que a reunião das ações de controle são realizadas por cada atuador, ou seja, por cada evento controlável.

Diferentes estratégias de combinação das ações de controle foram propostas na literatura, porém, neste trabalho considera-se a definição original feita em [10], na qual a interseção de todas as ações de controle $S_i(s)$ é a estratégia responsável pela definição de $S_{dec}(s)$. Além disso, em caso de ambiguidade na definição da habilitação de um evento controlável por um agente, este evento deve ser habilitado pelo agente. Essa arquitetura é conhecida como arquitetura conjuntiva e permissiva, e leva à seguinte função de supervisão resultante:

$$S_{dec}(s) = \bigcap_{i=1}^{n} S_i(s).$$

O comportamento "global" resultante da malha fechada será dado pelas linguagens $\mathcal{L}(S_{dec}|G)$ e $\mathcal{L}_m(S_{dec}|G)$, enquanto que o comportamento "local" visto pelo i-ésimo agente supervisor será dado pelas linguagens $P_i[\mathcal{L}(S_{dec}|G)]$ e $P_i[\mathcal{L}_m(S_{dec}|G)]$.

Coobservabilidade

Considere, agora, o problema da síntese de n agentes supervisores-P, construídos de acordo com uma arquitetura de controle supervisório descentralizado, que juntos sejam capazes de limitar o comportamento da planta G ao especificado por uma linguagem K. Além da condição de controlabilidade da linguagem K em relação à linguagem da planta M e E_{uc}, é preciso garantir que a função de supervisão resultante S_{dec} seja admissível. Uma vez que a combinação das ações de controle S_i é feita pela regra da interseção, é possível notar que, após a ocorrência de $s \in \overline{K}$, um evento só estará presente em S_{dec} se estiver em todo S_i. Logo, é possível concluir que para evitar que um evento σ, tal que $\sigma \in M \backslash \overline{K}$, leve o sistema para fora da especificação, é suficiente que apenas um único agente supervisor (por exemplo,

o i-ésimo) seja capaz de desabilitá-lo. Para tanto, é necessário que duas coisas aconteçam: (i) $\sigma \in E_{c,i}$, e, (ii) nenhuma outra sequência com mesma projeção para o agente i esteja dentro da especificação, ou seja, $P_i^{-1}[P_i(s)]\sigma \cap \overline{K} = \emptyset$. A formalização deste entendimento é definida pelo conceito de *coobservabilidade*.

Definição 2.22. *(Coobservabilidade[13]) Sejam K e $M = \overline{M}$ linguagens regulares definidas sobre o conjunto de eventos E. Sejam $E_{c,i}$ e $E_{o,i}$ subconjuntos de E que contém, respectivamente, os eventos controláveis e observáveis pelo agente $i \in \{1, 2, ..., n\}$. Seja P_i a projeção natural correspondente à $E_{o,i}$ ($P_i : E^* \rightarrow E_{o,i}^*$). Então, K será coobservável em relação a M, $E_{o,i}$ e $E_{c,i}$, com $i \in \{1, 2, ..., n\}$, se $\forall s \in \overline{K}$ e $\forall \sigma \in E_c = \cup_{i=1}^{n} E_{c,i}$,*

$$
\begin{aligned}
&(s\sigma \in M)\ (s\sigma \notin \overline{K}) \Rightarrow \\
&\exists i \in \{1, ..., n\} \text{ tal que } P_i^{-1}[P_i(s)]\sigma \cap \overline{K} = \emptyset \ \wedge \ \sigma \in E_{c,i}.
\end{aligned}
$$

Em palavras, a coobservabilidade pode ser definida da seguinte forma:

> *"Se um evento precisa ser desabilitado, então é suficiente que um agente supervisor capaz de controlar esse evento possa fazê-lo sem ambiguidade."*

Assim como a observabilidade, a coobservabilidade é uma propriedade do fecho do prefixo de uma linguagem, ou seja, K é coobservável se, e somente se, $\overline{K}$ for coobservável. Além disso, é possível notar que se existir apenas um agente supervisor, então a definição de coobservabilidade se reduz à de observabilidade, conforme esperado. Observe, também, que se $E_{c,i} \cap E_{c,j} = \emptyset$, para $i, j \in \{1, 2, ..., n\}$ e $i \neq j$, então, como cada agente é totalmente responsável por desabilitar seus eventos não-controláveis, a coobservabilidade reduzir-se-á à observabilidade de K em relação a M, $E_{c,i}$ e $E_{o,i}$ para cada $i \in \{1, 2, ..., n\}$.

Exemplo 2.11. *Para ilustrar o conceito de coobservabilidade, considere os autômatos G e H representados na figura 2.20, em que $E = \{a, b, g\}, M = \mathcal{L}(G) = \overline{\{g, bg, ag\}}$ e $K = \mathcal{L}_m(H) = \overline{\{g, b, a\}}$. Considere que uma arquitetura de controle supervisório descentralizado com dois agentes seja utilizada, na qual $E_{c,1} = \{g\}$, $E_{c,2} = \{g\}$, $E_{o,1} = \{a\}$ e $E_{o,2} = \{b\}$. Então, no estado inicial 0, ambos os agentes habilitam o evento controlável g, além dos eventos não-controláveis a e b. Logo, tem-se que:*

$$S_1(\varepsilon) = \{a, b, g\},\ S_2(\varepsilon) = \{a, b, g\},\ S_{dec}(\varepsilon) = \{a, b, g\}.$$

Logo, qualquer evento pode ocorrer. Se o evento não-observável g ocorrer, então o sistema continuará dentro da linguagem especificada K e chegará ao estado marcado

[13]Conhecido na literatura atual como *CP-coobservabilidade*.

1 *com as seguintes funções de supervisão:*

$$S_1(g) = \{a, b, g\},\ S_2(g) = \{a, b, g\},\ S_{dec}(g) = \{a, b, g\}.$$

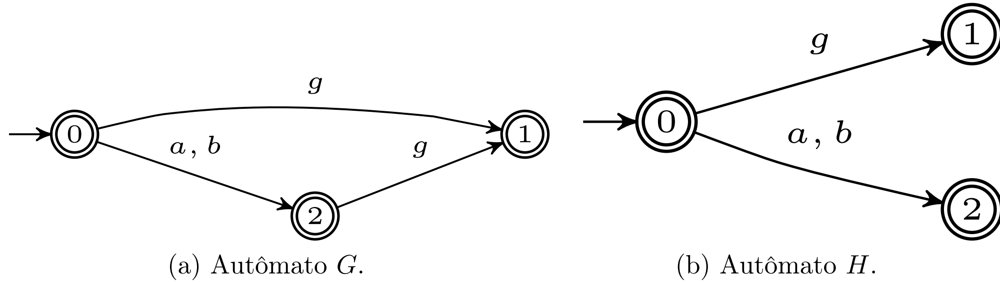
Como $\Gamma(1) = \emptyset$, *o sistema atingiu a primeira de suas especificações. Por outro lado, se no estado* 0 *ocorrer o evento* b, *então:*

$$S_1(b) = \{a, b, g\},\ S_2(b) = \{a, b\},\ S_{dec}(b) = \{a, b\}.$$

Note que, enquanto o agente 1 *confunde as sequências* ε *e* b, *o agente* 2 *não, o que o leva a desabilitar* g, *impedindo a ocorrência da sequência* bg *que viola a especificação. De maneira semelhante, se no estado* 0 *ocorrer o evento* a, *então:*

$$S_1(a) = \{a, b\},\ S_2(a) = \{a, b, g\},\ S_{dec}(a) = \{a, b\}.$$

Nesse caso, enquanto o agente 2 *confunde as sequências* ε *e* a, *o agente* 1 *não, o que o leva a desabilitar* g, *impedindo a ocorrência da sequência* ag *que viola a especificação. Com isto, é possível afirmar que* K *é coobservável em relação a* M, $E_{o,1}$, $E_{o,2}$, $E_{c,1}$ *e* $E_{c,2}$.



(a) Autômato G. (b) Autômato H.

Figura 2.20: Exemplo de malha fechada formada por modelo (G) e especificação (H).

Baseado no teorema 2.2, podemos apresentar um último resultado para a arquitetura de controle supervisório descentralizado, unificando os conceitos de controlabilidade e coobservabilidade:

Teorema 2.3. *(Controlabilidade e Coobservabilidade) Seja* $G = (X, E, f, \Gamma, x_0, X_m)$, *em que* $E_c \subseteq E$ *é o conjunto de eventos controláveis, e* $E_o \subseteq E$ *é o conjunto de eventos observáveis. Para cada agente* $i \in \{1, 2, ..., n\}$ *considere o conjunto dos eventos controláveis* $E_{c,i}$, *e o conjunto dos eventos observáveis* $E_{o,i}$, *tais que* $E_c = \cup_{i=1}^{n} E_{c,i}$ *e* $E_o = \cup_{i=1}^{n} E_{o,i}$. *Seja* P_i *a projeção natural correspondente a* $E_{o,i}$ *(*$P_i : E^* \rightarrow E_{o,i}^*$*). Considere, também, a linguagem* $K \subseteq \mathcal{L}(G)$, *tal que* $K \neq \emptyset$. *Então, existe* S_{dec} *para* G *(de acordo com a arquitetura*

da figura 2.19) tal que

$$\mathcal{L}(S_{dec}|G) = \overline{K},$$

se e somente se K for controlável em relação a $\mathcal{L}(G)$ e E_{uc}, e coobservável com respeito a $\mathcal{L}(G)$, $E_{c,i}$ e $E_{o,i}$ para $i \in \{1, 2, ..., n\}$.

2.3 Complexidade Computacional

Dentro do campo de estudo de SEDs, tão importante quanto as propriedades e operações definidas sobre estes, são os algoritmos que as implementam. Assim, para permitir o avanço da teoria, faz-se necessária a definição de uma métrica de comparação da complexidade resultante de cada algoritmo. Com esse propósito, baseado em [19], esta seção apresenta o conceito mais utilizado para comparação de algoritmos, o do comportamento assintótico de funções.

Ao comparar algoritmos genéricos estamos interessados em saber qual será seu comportamento no pior caso, o que pode ser feito por meio do cálculo do tempo assintótico. Esse cálculo indica como o tempo gasto pelo algoritmo cresce em função de suas entradas. Por assintótico entende-se que são levados em consideração apenas os termos de maior ordem, dado que, para valores grandes da magnitude das entradas, os termos de menor ordem podem ser desprezados. A notação matemática $\mathcal{O}(\cdot)$ é utilizada para representar esse conceito.

Definição 2.23. *(Notação $\mathcal{O}$) Uma função $g(n)$ é $\mathcal{O}(f(n))$ se existem duas constantes positivas c e m tais que*

$$g(n) \leq c.f(n), \text{ para todo } n \geq m.$$

Desse modo, diz-se que se um dado algoritmo cresce na taxa de $g(n)$, em que $g(n)$ é $\mathcal{O}(f(n))$, então, no pior caso, e para valores de n elevados, o algoritmo representado por $g(n)$ cresce tanto quanto $f(n)$.

Indo além na utilização dessa métrica para comparar algoritmos, é possível definir classes de comportamento assintótico para os algoritmos baseando-se na função matemática representada por $f(n)$. Assim, se um algoritmo possui complexidade $\mathcal{O}(f(n))$ e $f(n)$ é um polinômio, diz-se que esse algoritmo tem complexidade polinomial. Por outro lado, se o algoritmo é $\mathcal{O}(f(n))$ e $f(n)$ é uma função exponencial, então o algoritmo tem complexidade exponencial.

De uma maneira geral, algoritmos com crescimento exponencial não são úteis sob o ponto de vista prático, o que leva à constante busca por algoritmos de complexidade polinomial ou inferior.

No âmbito deste trabalho, os algoritmos em estudo são sempre definidos sobre

autômatos determinísticos. Logo, o cálculo da complexidade computacional dos algoritmos levará sempre em consideração a cardinalidade dos conjuntos de estados dos autômatos de entrada ($|X|$) e a cardinalidade das transições no pior caso que será dada por $|X| \times |E|$. Note que para o cálculo da complexidade de um algoritmo pela notação $\mathcal{O}$, é sempre válido considerar que $|X| >> |E|$, já que o número de estados em um sistema é tipicamente muito superior ao número de eventos.

Além disso, é importante notar que $|X|$ e $|E|$ crescem exponencialmente com o número de componentes da planta e da especificação, em função da composição síncrona que normalmente é utilizada para suas obtenções. Assim, muito embora diversos algoritmos possuam, isoladamente, complexidade polinomial, esses algoritmos são implicitamente exponenciais em relação ao número de componentes da planta e da especificação.

Capítulo 3

Algoritmos em tempo polinomial para verificação da observabilidade, normalidade e coobservabilidade

Este capítulo apresenta a principal contribuição deste trabalho: o desenvolvimento de algoritmos mais eficientes para a verificação das propriedades de observabilidade e normalidade. Será, inicialmente, considerado o caso centralizado, e em seguida, será feita a generalização dos resultados para o cenário de controle supervisório descentralizado.

A organização deste capítulo é feita da seguinte maneira: a seção 3.1 abordará uma revisão dos principais métodos existentes na literatura; em seguida nas seções 3.2 e 3.3 serão propostos algoritmos para verificação de observabilidade e normalidade em tempo polinomial, e na seção 3.4 será ilustrada a aplicação dos algoritmos; a seção 3.5 apresenta a generalização do verificador proposto na seção 3.2 para a coobservabilidade; por fim, a síntese dos principais resultados apresentados neste capítulo é feita na seção 3.6.

3.1 Revisão Bibliográfica

Com o objetivo de possibilitar a comparação dos algoritmos a serem propostos com os existentes na literatura, iremos fazer uma breve revisão dos principais algoritmos já existentes.

Neste capítulo vamos supor que K e M são linguagens regulares, isto é, existem autômatos $G = (X_G, E, f_G, \Gamma_G, x_0, X_{m,G})$, e $H = (X_H, E, f_H, \Gamma_H, x_0, X_{m,H})$, tais que $\mathcal{L}(G) = M = \overline{M}$, $\mathcal{L}_m(H) = K$, e $H \sqsubseteq G$. Note que essa última condição pode ser colocada sem perda de generalidade, uma vez que sempre é possível modificar os autômatos G e H para torná-la verdadeira, conforme apresentado pelo algoritmo

2.1. Vamos, a seguir, apresentar os principais algoritmos existentes na literatura para verificar a observabilidade de linguagens.

3.1.1 Observabilidade

Seja $K \subseteq M = \overline{M}$. De acordo com a definição 2.20, K será observável em relação a M, E_o e E_c se $\forall s \in \overline{K}$ e $\forall \sigma \in E_c$,

$$(s\sigma \notin \overline{K}) \ e \ (s\sigma \in M) \Rightarrow P^{-1}[P(s)]\sigma \cap \overline{K} = \emptyset.$$

- **Algoritmo baseado no Observador**

O método mais simples e intuitivo para a verificação da observabilidade de linguagens regulares se baseia na construção de um observador para H. Dessa forma, é possível realizar a busca por estados onde ações de controle conflitantes são geradas, caracterizando assim a violação da condição de observabilidade. O algoritmo a seguir detalha esse procedimento:

Algoritmo 3.1 ([9]). *Sejam G e H autômatos tais que $H \sqsubseteq G$. Seja o conjunto de eventos observáveis representado por $E_o \subseteq E$ e o conjunto de eventos controláveis representado por $E_c \subseteq E$.*

Passo 1: *Construa $H_{obs} := Obs(H, E_o)$.*

Passo 2: *Para cada estado x_{obs} de H_{obs}, crie o conjunto $Enable(x_{obs})$ dos eventos controláveis que devem ser habilitados por cada estado $x \in x_{obs}$. Faça o mesmo para o conjunto $Disable(x_{obs})$ dos eventos que devem ser desabilitados em x_{obs}.*

$$Enable(x_{obs}) := \{e \in E_c \ : \ (\exists x \in x_{obs})[e \in \Gamma_H(x)]\}$$
$$Disable(x_{obs}) := \{e \in E_c \ : \ (\exists x \in x_{obs})[e \in \Gamma_G(x) \backslash \Gamma_H(x)]\}$$

Passo 3: *Se $\exists x_{obs} \mid Enable(x_{obs}) \cap Disable(x_{obs}) \neq \emptyset$ então K é não-observável com relação a M, E_o e E_c.*

Apesar da busca por eventos em cada estado x de cada estado x_{obs} possuir complexidade polinomial, a construção do observador H_{obs} possui complexidade exponencial (como visto na seção 2.1.3), o que torna esse método recomendado apenas para casos específicos com poucas ocorrências de transições não-observáveis.

Exemplo 3.1. *Sejam os autômatos G e H representados nas figuras 3.1a e 3.1b, em que $E = \{u, b\}, M = \mathcal{L}(G) = \overline{\{ub, bu\}}$ e $K = \mathcal{L}_m(H) = \{ub\}$. Seja $E_c = \{b\}$ e $E_o = \{b\}$. Nas figuras 3.1a e 3.1b é possível verificar a presença de transições pontilhadas. Note que essas transições estão associadas aos eventos não-observáveis*

u e u_R. Nesse sentido, de agora em diante, as transições associadas a eventos não-observáveis serão pontilhadas.

É imediato notar que as sequências $s = \varepsilon$ e $s' = u$ possuem a mesma projeção observável ($P(s) = P(s') = \varepsilon$) e $s, s' \in \overline{K}$. Além disso para $\sigma = b \in E_c$, como $s\sigma \in M \backslash \overline{K}$ e $s'\sigma \in \overline{K}$, então conclui-se que K é não-observável em relação a M, E_c e E_o.

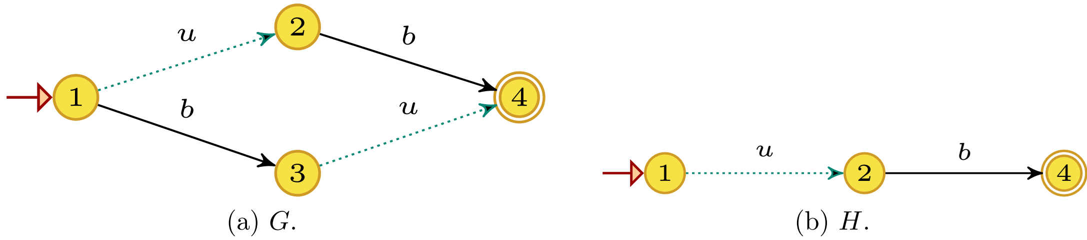


Figura 3.1: Autômatos de entrada do exemplo 3.1.

A figura 3.2 exibe o autômato $H_{obs} = Obs(H, E_o)$ construído de acordo com o algoritmo 3.1. O estado inicial de H_{obs} engloba os estados 1 e 2. Como b deve ser desabilitado no estado 1, então $b \in Disable(\{1,2\})$, e como b deve ser habilitado em 2, $b \in Enable(\{1,2\})$. Logo $b \in Enable(\{1,2\}) \cap Disable(\{1,2\})$, o que, como esperado, permite concluir que K é não-observável em relação a M, E_c e E_o.

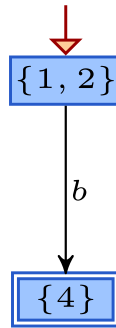


Figura 3.2: Autômato H_{obs} do exemplo 3.1.

- **Algoritmo proposto por TSITSIKLIS [11]**

Um algoritmo eficiente para a verificação da observabilidade de uma linguagem foi proposto ainda no início do desenvolvimento da teoria de controle supervisório por TSITSIKLIS [11]. Esse mesmo algoritmo é apresentado como sendo o mais eficiente em [9].

A base do funcionamento do algoritmo é a procura por pares de sequências (s', s) de mesma projeção em H, enquanto simultaneamente acompanha-se o estado atingido em G pela sequência s, objetivando identificar algum par de sequências

(s', s) que viole a condição de observabilidade. Para tanto, um autômato verificador com uma tripla de estados $(x_1, x_2, x_3) \in (X_H \times X_H \times X_G)$ é construído, no qual triplas de eventos $(\sigma', \sigma, \sigma)$ são usadas para identificar as transições. Iniciando pelo estado $(x_{0,H}, x_{0,H}, x_{0,G})$, a definição de transições passa, primeiramente, pela escolha de eventos $\sigma', \sigma \in E_\varepsilon = E \cup \{\varepsilon\}$ do seguinte modo:

(i) Escolha de algum $\sigma' \in \Gamma_H(x_1)$. Se $\sigma' \in E_o$ então simultaneamente deve-se escolher algum $\sigma \in \Gamma_G(x_3)$. Se $\sigma' \notin E_o$, então $\sigma = \varepsilon$.

(ii) Escolha de algum $\sigma \in \Gamma_G(x_3)$. Se $\sigma \in E_o$ então simultaneamente deve-se escolher algum $\sigma' \in \Gamma_H(x_1)$. Se $\sigma \notin E_o$, então $\sigma' = \varepsilon$.

(iii) Escolha de ambos: $\sigma' \in \Gamma_H(x_1)$ e $\sigma \in \Gamma_G(x_3)$.

Após a escolha dos eventos, a evolução do sistema é feita de acordo com as seguintes regras:

(i) x_1 não se modifica se nenhum σ' for escolhido ($\sigma' = \varepsilon$). Caso algum σ' for escolhido, x_1 se modifica para $f_H(x_1, \sigma')$.

(ii) x_2 e x_3 não se modificam se nenhum σ for escolhido ($\sigma = \varepsilon$). Caso algum σ seja escolhido, x_2 se modifica para $f_H(x_2, \sigma)$ e x_3 se modifica para $f_G(x_3, \sigma)$, a menos que $\sigma \notin \Gamma_H(x_2)$, o que leva à violação da condição de observabilidade.

O algoritmo 3.2 implementa esta ideia por meio da construção de um verificador $ObsTest(H, G)$, no qual o estado *dead* simboliza a violação da condição de observabilidade identificada pela regra (ii).

Algoritmo 3.2 ([9])**.** *Sejam G e H autômatos tais que $H \sqsubseteq G$. Seja o conjunto de eventos observáveis representado por $E_o \subseteq E$ ($P : E^* \to E_o^*$), e o conjunto dos eventos controláveis representado por $E_c \subseteq E$. Defina o conjunto de eventos $E_\varepsilon = E \cup \{\varepsilon\}$.*

Passo 1: *Construa o autômato*

$$\begin{aligned} ObsTest(H, G) := Ac((X_H \times X_H \times X_G) \cup \{dead\}, E_\varepsilon \times E_\varepsilon \times E_\varepsilon, \\ f_{test}, (x_{0,H}, x_{0,H}, x_{0,G}), \{dead\}), \end{aligned}$$

com a função de transição

$$\begin{aligned} f_{test} \ : \ [(X_H \times X_H \times X_G) \cup \{dead\}] \times [E_\varepsilon \times E_\varepsilon \times E_\varepsilon] \to \\ [(X_H \times X_H \times X_G) \cup \{dead\}] \end{aligned}$$

definida da seguinte maneira:

· *Se* $e \in E_o$*:*

$$f_{test}((x_1, x_2, x_3), (e, e, e)) = (f_H(x_1, e), f_H(x_2, e), f_G(x_3, e));$$
$$f_{test}((x_1, x_2, x_3), (e, \varepsilon, e)) = dead.$$

· *Se* $e \in E_{uo}$*:*

$$f_{test}((x_1, x_2, x_3), (e, \varepsilon, \varepsilon)) = (f_H(x_1, e), x_2, x_3);$$
$$f_{test}((x_1, x_2, x_3), (\varepsilon, e, e)) = (x_1, f_H(x_2, e), f_G(x_3, e));$$
$$f_{test}((x_1, x_2, x_3), (e, \varepsilon, e)) = dead.$$

Passo 2: *K é observável em relação a M, E_o e E_c, se e somente se*

$$\mathcal{L}_m(ObsTest(H, G)) = \emptyset.$$

A complexidade desse algoritmo é polinomial na cardinalidade do espaço de estados de G e H, mais especificamente, no pior caso tem complexidade $\mathcal{O}((|X_H \times X_H \times X_G| + 1) \times |E_\varepsilon \times E_\varepsilon \times E_\varepsilon|) = \mathcal{O}((|X|^3 + 1)(|E| + 1)^3) = \mathcal{O}(|X|^3|E|^3)$.

Exemplo 3.2. *Considere novamente o sistema não-observável analisado no exemplo 3.1. A figura 3.3 ilustra o verificador obtido com a execução do algoritmo 3.2, no qual a existência do estado marcado dead permite concluir que K é não-observável em relação a M, E_c e E_o.*

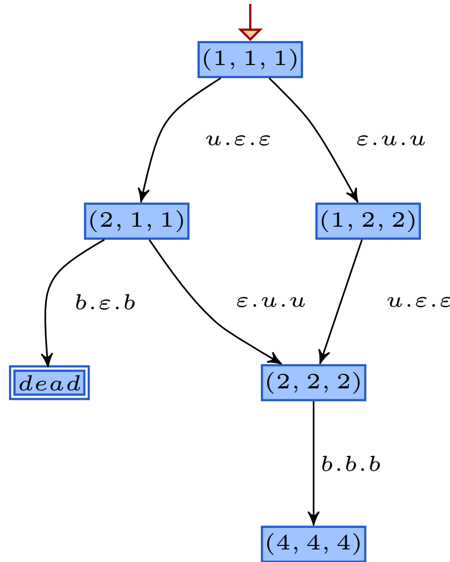


Figura 3.3: Autômato *ObsTest* do exemplo 3.2.

- **Algoritmo proposto por WANG *et al.* [12]**

Mais recentemente, uma nova e ainda mais eficiente proposta para a verificação de observabilidade foi feita por WANG *et al.* [12]. Partindo da semelhança entre os problemas de diagnose de falhas de sistemas a eventos discretos e observabilidade, demonstrou-se, em [12], que o problema de observabilidade pode ser transformado em um problema de diagnose de falhas, o que permite a utilização da teoria já desenvolvida para a diagnose de falhas (ver [17], [13] e [12]).

Em WANG *et al.* [12] são propostas duas transformações semelhantes e com o mesmo objetivo: transformar o problema de verificar a observabilidade em um problema de verificar a diagnosticabilidade. A primeira dessas transformações é chamada COOBS-TO-DIAG-I, e resulta na criação de um autômato para cada $e \in E_c$, com complexidade $\mathcal{O}(|X||E|)$. Isso gera uma complexidade total de $\mathcal{O}(|X||E|^2)$ para o algoritmo. A segunda transformação, denominada COOBS-TO-DIAG-II, possui a vantagem de gerar apenas um autômato de saída, no qual a verificação de diagnosticabilidade implica diretamente observabilidade. Essa transformação possui complexidade total $\mathcal{O}(|X||E|^2)$ e é implementada conforme o algoritmo a seguir.

Algoritmo 3.3 (COOBS-TO-DIAG-II[1])**.** *Sejam G e H autômatos tais que $H \sqsubseteq G$. Seja o conjunto de eventos observáveis denotado por $E_o \subseteq E$, e o conjunto dos eventos controláveis denotado por $E_c \subseteq E$. Suponha que $z \notin E$ e, para todo $e \in E_c$ defina $d_e \notin X_H$ e $v_e, f_e, u_e, r_e \notin E$. Considere $E^t = \{z\} \cup \{v_e, f_e, u_e, r_e : e \in E_c\}$. Seja $X_{\tilde{H}} = X_H \cup \{d_e : e \in E_c\}$ e $E_{\tilde{H}} = E \cup E^t$. O autômato $\tilde{H} = (X_{\tilde{H}}, E_{\tilde{H}}, f_{\tilde{H}}, x_0)$ pode ser construído da seguinte forma:*

Passo 1: *Faça $\tilde{H} := H$. Para todo $e \in E_c$, adicione estados não marcados d_e ao espaço de estados de $\tilde{H}$.*

Passo 2: *Para cada estado d_e, adicione o auto-laço definido por $f_{\tilde{H}}(d_e, v_e) = d_e$. Faça $v_e \in E_o$.*

Passo 3: *Para todo $e \in E_c$ e $x \in X_H$, se $e \in \Gamma_G(x) \backslash \Gamma_H(x)$, adicione as transições $f_{\tilde{H}}(x, f_e) = d_e$ e $f_{\tilde{H}}(x, r_e) = d_e$, tais que $f_e \in E_{uo}$ e $r_e \in E_o$. Se $e \in \Gamma_H(x)$, adicione a transição $f_{\tilde{H}}(x, u_e) = d_e$, em que $u_e \in E_{uo}$.*

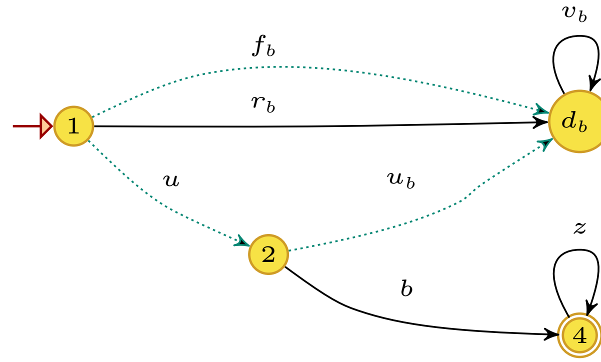
Passo 4: *Adicione um auto-laço com evento observável z em cada estado $x \in X_H \subseteq X$ que seja um* deadlock *em G.*

Definindo o conjunto E_f contendo todos os eventos $f_e \in E_{\tilde{H}}$, a análise da diagnosticabilidade de $\tilde{H}$ em relação a E_f pode, então, ser realizada utilizando o

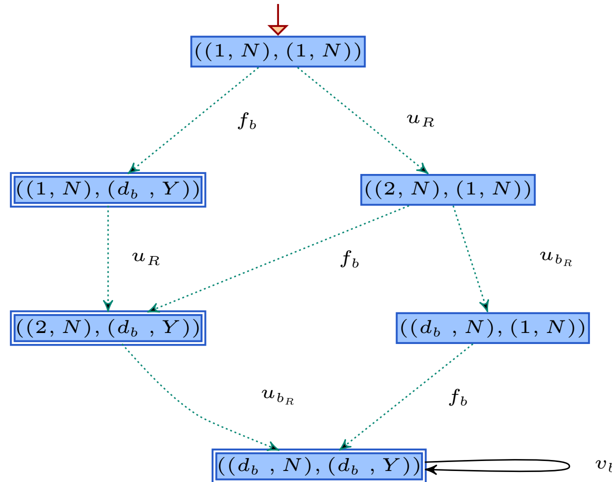
[1]O algoritmo apresentado neste é o caso particular do algoritmo proposto por WANG *et al.* [12], aplicado ao caso centralizado e com observação estática.

algoritmo de complexidade polinomial proposto em MOREIRA *et al.* [13]. O verificador resultante (V_{diag}) possui complexidade $\mathcal{O}(|X|^2|E|)$. Desse modo, pode-se concluir que a verificação da observabilidade feita a partir da execução do algoritmo 3.3, seguida pela execução do algoritmo de verificação de diagnosticabilidade de [13], resulta em um algoritmo global de complexidade $\mathcal{O}(|X||E|^2 + |X_{\tilde{H}}|^2 \times |E_{\tilde{H}}|) = \mathcal{O}(|X||E|^2 + (|X| + |E|)^2 \times (5|E| + 1)) = \mathcal{O}(|X|^2|E|)$ (uma vez que tipicamente $|X| >> |E|$).

Exemplo 3.3. *Considere novamente o sistema não-observável analisado no exemplo 3.1. A figura 3.4a mostra o autômato $\tilde{H}$ construído de acordo com o algoritmo 3.3, e a figura 3.4b mostra o autômato V_{diag}, que é o verificador correspondente a $\tilde{H}$ utilizando [13]. Conforme [13], a existência de um ciclo no estado $((d_b, N), (d_b, Y))$, com $v_b \in E$, leva à não diagnosticabilidade, o que consequentemente implica em não observabilidade de K em relação a M, E_c e E_o.*



(a) $\tilde{H}$.



(b) V_{diag}.

Figura 3.4: Autômatos do exemplo 3.3.

3.1.2 Normalidade

Diferentemente da observabilidade, a verificação da normalidade de um sistema não foi considerada em profundidade na literatura. Apenas KUMAR *et al.* [20] e LIN e MORTAZAVIAN [21] colocam esta verificação como algo possível de ser feito a partir do modelo da planta e de sua especificação.

- **Algoritmo baseado no Observador para verificar a Normalidade**

Uma maneira imediata de verificar a normalidade de uma linguagem regular pode ser facilmente deduzida a partir de sua definição (definição 2.21). Sejam $M = \overline{M} \subseteq E^*$ e o conjunto de eventos observáveis denotado por $E_o \subseteq E$. Diz-se que a linguagem K é normal em relação a M e E_o se

$$\overline{K} = P^{-1}[P(\overline{K})] \cap M.$$

Note que a relação de inclusão $\overline{K} \subseteq P^{-1}[P(\overline{K})] \cap M$ sempre é válida. Logo, estamos interessados em verificar apenas se $P^{-1}[P(\overline{K})] \cap M \subseteq \overline{K}$. O algoritmo a seguir executa a análise dessa relação de inclusão, e consequentemente, permite concluir sobre a normalidade de uma linguagem regular em relação a M e E_o.

Algoritmo 3.4. *Sejam G e H autômatos tais que $H \sqsubseteq G$. Seja o conjunto de eventos observáveis representado por $E_o \subseteq E$.*

Passo 1: *Construa os autômatos G_m e H_m, com todos os estados marcados, da seguinte forma:*

$$G_m := (X_G, E, f_G, \Gamma_G, x_0, X_G);$$
$$H_m := (X_H, E, f_H, \Gamma_H, x_0, X_H).$$

Passo 2: *Construa $H_{obs} := Obs(H_m, E_o)$.*

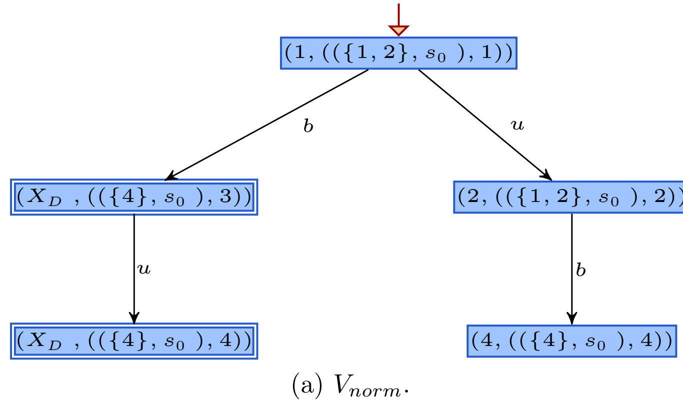
Passo 3: *Construa H_{obs}^{sl} adicionando auto-laços com todos os eventos $\sigma \in E_{uo}$ em todos os estados de H_{obs}.*

Passo 4: *Construa $H_{int} := H_{obs}^{sl} \times G_m$.*

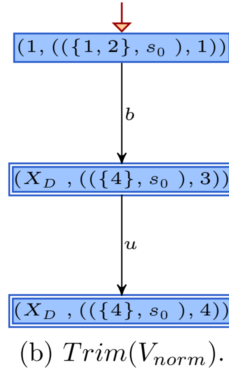
Passo 5: *Construa $V_{norm} := H_m^C \times H_{int}$, sendo H_m^C o complementar de H_m.*

Passo 6: *Se $Trim(V_{norm}) = \emptyset$ então K é normal em relação a M e E_o.*

Por incluir a construção de um observador para o autômato H_m, que marca $\overline{K}$, a complexidade desse algoritmo é exponencial, o que torna esse método recomendado apenas para casos específicos com poucas ocorrências de transições não-observáveis.



(a) V_{norm}.



(b) $Trim(V_{norm})$.

Figura 3.5: Autômatos do exemplo 3.4.

Exemplo 3.4. *Considere novamente o sistema analisado no exemplo 3.1. A existência das sequências $s = b$ e $s' = ub$, tais que $P(s) = P(s') = b$, $s \in M \backslash \overline{K}$ e $s' \in \overline{K}$ implicam que K não é normal em relação a M e E_o.*

A figura 3.5a exibe o autômato V_{norm}, o verificador resultante da execução do algoritmo 3.4, enquanto que a figura 3.5b ilustra o autômato $Trim(V_{norm})$. Como $Trim(V_{norm})$ não é o autômato vazio, então, conforme esperado, conclui-se que K não é normal em relação a M e E_o.

3.2 Verificação de Observabilidade

Inspirado pelos conceitos utilizados em MOREIRA *et al.* [13], onde a ferramenta de renomeação de eventos é utilizada para capturar as confusões que a observabilidade parcial causa para o diagnosticador (i.e. sequência diferentes mas com mesma projeção observável), vamos, a seguir, propor um novo algoritmo para a verificação da observabilidade de uma linguagem regular que possui complexidade computacional igual ou menor que algoritmos existentes na literatura. O motivo que leva à menor complexidade do algoritmo aqui proposto é o fato de a procura pela violação ocorrer somente dentre as possíveis confusões causadas pela observação parcial.

3.2.1 Algoritmo para Verificação

Seja $K \subseteq M = \overline{M}$ e considere o problema de se verificar se K é observável em relação a M, E_o e E_c. De acordo com a definição de observabilidade tem-se que K será observável em relação a M, E_o e E_c se $\forall \sigma \in E_c$ e $\forall s \in \overline{K}$ tal que $s\sigma \in M$ e $s\sigma \notin \overline{K}$ implicar que $P^{-1}[P(s)]\sigma \cap \overline{K} = \emptyset$, ou, equivalentemente, $\forall \sigma \in E_c$ e $\forall s \in \overline{K}$ tal que $s\sigma \in M$ e $s\sigma \notin \overline{K}$, $\nexists s' \in \overline{K}$ tal que $s'\sigma \in \overline{K}$ e $P(s) = P(s')$. Essa definição equivalente sugere a construção do diagrama da figura 3.6.

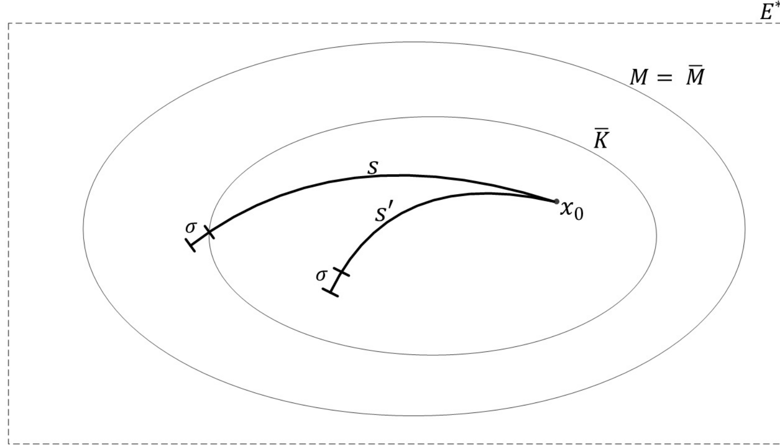


Figura 3.6: Diagrama ilustrativo da definição equivalente da observabilidade.

De acordo com o diagrama da figura 3.6, a busca pelas sequências $s, s' \in \overline{K}$ capazes de violar a condição de observabilidade, quando seguidas de um evento $\sigma \in E_c$, pode ser feita por meio da construção de dois autômatos, em que um contenha o comportamento "normal" do sistema, contendo todas as possíveis sequências s', e outro que contenha o comportamento de "falha" (sair da especificação $\overline{K}$), contendo todas as possíveis sequências s. Se o autômato que representa o comportamento "normal" for todo marcado, e tiver todos os eventos não-observáveis renomeados, e se o autômato que representa o comportamento de "falha" tiver marcado apenas os estados que estejam fora da especificação, então a operação de composição paralela entre estes autômatos irá capturar todas as possíveis sequências de mesma projeção observável ($P(s) = P(s')$). Além disso, as transições que saem de um estado não-marcado para um estado marcado simbolizam todos os eventos σ ocorrendo após alguma sequência s. Adicionalmente, se alguma dessas transições ocorre com evento observável, então este evento σ também ocorreu após a sequência s', indicando violação da observabilidade. Por fim, se alguma dessas transições ocorre com um evento não-observável, então se no estado de origem dela existe uma transição com evento σ renomeado, conclui-se que σ também ocorreu após a sequência s', indicando violação da observabilidade. O algoritmo a seguir implementa essa estratégia para verificar a propriedade de observabilidade de um sistema.

Algoritmo 3.5. *Sejam os autômatos* $G = (X_G, E, f_G, \Gamma_G, x_0, X_{m,G})$ *e* $H = (X_H, E, f_H, \Gamma_H, x_0, X_{m,H})$ *tais que* $\mathcal{L}(G) = M = \overline{M}$, $\mathcal{L}_m(H) = K$ *e* $H \sqsubseteq G$. *Sejam os conjuntos de eventos controláveis e observáveis representados respectivamente por* $E_c \subseteq E$ *e* $E_o \subseteq E$.

Passo 1: *Construa os autômatos* G_m *e* H_m, *com todos os estados marcados, da seguinte forma:*

$$G_m := (X_G, E, f_G, \Gamma_G, x_0, X_G);$$
$$H_m := (X_H, E, f_H, \Gamma_H, x_0, X_H).$$

Passo 2: *Defina a função de renomeação* $R : E \to E_R$ *como*[2]*:*

$$R(\sigma) = \begin{cases} \sigma & \text{se } \sigma \in E_o \\ \sigma_R & \text{se } \sigma \notin E_o. \end{cases}$$

Construa o autômato $H_R := (X_H, E_R, f_R, \Gamma_R, x_0, X_H)$ *com* $E_R = R(E)$, $f_R(x, R(\sigma)) = f_H(x, \sigma)$ *e* $\Gamma_R(x) = R[\Gamma_H(x)]$, $\forall x \in X_H$.

Passo 3: *Construa o autômato* H_C^D *por meio dos seguintes passos:*

3.1: $H_C := G_m \times H_m^C$, *sendo* H_m^C *o complementar de* H_m*;*

3.2: *Defina a função de renomeação de estado* $\mathcal{D} : X \to X_D$ *como*[3]*:*

$$\mathcal{D}(x) = \begin{cases} x & \text{se } x \notin X_m \\ D & \text{se } x \in X_m; \end{cases}$$

Defina o autômato $H_C^D := (X_{H_C}^D, E, f_{H_C}^D, \Gamma_{H_C}^D, x_{0,H_C}, X_{m,H_C}^D)$ *fazendo:*

$$X_{H_C}^D = \mathcal{D}(X_{H_C});$$
$$\Gamma_{H_C}^D(x) = \Gamma_{H_C}[\mathcal{D}(x)];$$
$$f_{H_C}^D(x, \sigma) = \mathcal{D}[f_{H_C}(x, \sigma)];$$
$$X_{m,H_C}^D = \{D\}.$$

Passo 4: *Construa o autômato* V_{RC} *da seguinte forma:*

[2] Note que a função R apenas renomeia os eventos que não pertencem a E_o, permitindo a captura do comportamento assíncrono junto a outro autômato não renomeado. A notação $R(E)$ será usada para representar a renomeação de todos os eventos de um conjunto E.

[3] Note que a função $\mathcal{D}$ é utilizada para reunir todos os estados marcados. Seu emprego é válido quando o comportamento após atingir uma marcação não for mais relevante para a análise. A vantagem de sua utilização é diminuir a complexidade das operações de composição que utilizam o autômato como argumento. $\mathcal{D}(X)$ denota a aplicação da função $\mathcal{D}$ sobre cada estado do conjunto X. $\mathcal{D}(f(x, \sigma))$ denota a aplicação da função $\mathcal{D}$ sobre o estado retornado por $f(x, \sigma)$.

4.1: *$V_{RC} := H_R \| H_C^D = (X_{V_{RC}}, E_{V_{RC}}, f_{V_{RC}}, \Gamma_{V_{RC}}, x_{0,V_{RC}}, X_{m,V_{RC}})$, não sendo necessário obter as transições que partem dos estados marcados de V_{RC}, ficando a cargo da implementação essa escolha.*

4.2: *Defina o autômato $V := (X_V, E_{V_{RC}}, f_V, \Gamma_V, x_{0,V_{RC}}, X_{m,V})$ fazendo:*

$$\begin{aligned} X_V &= \mathcal{D}(X_{V_{RC}}); \\ \Gamma_V(x) &= \Gamma_{V_{RC}}[\mathcal{D}(x)]; \\ f_V(x,\sigma) &= \mathcal{D}[f_{V_{RC}}(x,\sigma)]; \\ X_{m,V} &= \{D\}. \end{aligned}$$

Passo 5: *Para todo evento $\sigma \in E_c$ tal que $(f_V(x,\sigma) = D) \wedge (x \neq D)$ verifique:*

(a) $\sigma \in E_o$;

(b) $\sigma \notin E_o$ *e* $R(\sigma) \in \Gamma_V(x)$.

Se (a) ou (b) então K não é observável em relação a M, E_c e E_o. Caso contrário, K é observável em relação a M, E_c e E_o. □

O teorema a seguir demonstra a corretude do algoritmo 3.5 na verificação da observabilidade.

Teorema 3.1. *Sejam K e $M = \overline{M}$ $(K \subseteq M)$ linguagens regulares tais que $\mathcal{L}_m(H) = K$, $\mathcal{L}(G) = M$ e seja $V = (X_V, E_V, f_V, \Gamma_V, x_{0,V}, X_{m,V})$ um autômato verificador construído de acordo com o algoritmo 3.5. Então K será não-observável em relação a M, E_o e E_c, se e somente se existir $\sigma \in E_c$ que satisfaça $f_V(x,\sigma) = D$ e $x \neq D$ tal que*

$$(\sigma \in E_o) \quad \vee \quad [(\sigma \notin E_o) \quad \wedge \quad (R(\sigma) \in \Gamma_V(x))].$$

Demonstração.

($\Rightarrow$) Suponha que K seja não-observável em relação a M, E_o e E_c. Seja a operação de projeção P definida por

$$P : E^* \to E_o^*.$$

Então, pela definição de observabilidade, existem sequências $s_1 \neq s_2$, $s_1, s_2 \in \overline{K}$ tais que: *(i)* $P(s_1) = P(s_2)$, *(ii)* $s_1\sigma \in \overline{K}$, *(iii)* $s_2\sigma \in M \backslash \overline{K}$, para algum $\sigma \in E_c$.

Pela construção de H_R, como $s_1\sigma \in \overline{K}$ então $s_1\sigma \in \mathcal{L}_m(H_m)$. Seja $s_{1_R} = R(s_1)$. Então $s_{1_R} \in \mathcal{L}_m(H_R) = \mathcal{L}(H_R)$.

Considere, agora, a operação de projeção P_{R_o} definida por:

$$P_{R_o} : E_R^* \to E_o^*,$$

sendo $E_R = R(E)$. Não é difícil verificar que $P(s_1) = P_{R_o}(s_{1_R})$, uma vez que a função de renomeação não altera a observabilidade de um evento.

Pela construção de H_C^D, $\mathcal{L}(H_C) \subseteq \mathcal{L}(H_C^D)$ e $\mathcal{L}_m(H_C) \subseteq \mathcal{L}_m(H_C^D)$. Assim, como $s_2 \in \overline{K}$ e $s_2\sigma \in M \backslash \overline{K}$, então $s_2 \in \mathcal{L}(H_C^D)$ porém $s_2 \notin \mathcal{L}_m(H_C^D)$ e $s_2\sigma \in \mathcal{L}_m(H_C^D)$.

Seja $E' = E_R \cup E$ e defina operações de projeção P', P'_R e P'_o da seguinte forma:

$$P' : E'^* \rightarrow E^*;$$
$$P'_R : E'^* \rightarrow E_R^*;$$
$$P'_o : E'^* \rightarrow E_o^*.$$

Considere, agora, o autômato V_{RC}. É fácil verificar que $\mathcal{L}(V_{RC}) \subseteq \mathcal{L}(V)$ e $\mathcal{L}_m(V_{RC}) \subseteq \mathcal{L}_m(V)$. Como $V_{RC} = H_R \| H_C^D$ então:

$$\mathcal{L}(V_{RC}) = P_R'^{-1}[\mathcal{L}(H_R)] \cap P'^{-1}[\mathcal{L}(H_C^D)] \subseteq \mathcal{L}(V); \tag{3.1}$$
$$\mathcal{L}_m(V_{RC}) = P_R'^{-1}[\mathcal{L}_m(H_R)] \cap P'^{-1}[\mathcal{L}_m(H_C^D)] \subseteq \mathcal{L}_m(V). \tag{3.2}$$

Como visto acima, $s_{1_R} \in \mathcal{L}(H_R)$ e $s_2 \in \mathcal{L}(H_C^D)$. Uma vez que s_{1_R} difere de s_2 apenas por eventos não-observáveis, é possível afirmar que $\exists s' \in P_R'^{-1}(s_{1_R}) \cap P'^{-1}(s_2)$, e portanto $s' \in \mathcal{L}(V)$. Porém, $s' \notin \mathcal{L}_m(V)$, uma vez que $s_2 \notin \mathcal{L}_m(H_C^D)$.

Existem duas opções para σ: $(i)\sigma \in E_o$; $(ii)\sigma \notin E_o$.

Considere o caso (i). Note que, como

$$\left\{\begin{matrix} s_{1_R} \in \mathcal{L}_m(H_R), s_{1_R}\sigma \in \mathcal{L}_m(H_R), \\ s_2 \notin \mathcal{L}_m(H_C^D), s_2\sigma \in \mathcal{L}_m(H_C^D) \end{matrix}\right\} \Rightarrow \left\{\begin{matrix} \exists x_R \in X_{H_R}, x_R \in X_{m,H_R} : \sigma \in \Gamma_{H_R}(x_R) \text{ e} \\ \exists x_C^D \in X_{H_C^D}, x_C^D \notin X_{m,H_C^D} : \sigma \in \Gamma_{H_C^D}(x_C^D) \end{matrix}\right\}$$

Portanto $f_V(x_{0,V}, s') = (x_R, x_C^D) = x$, e dessa forma, $f_V(x,\sigma) = D$ para $x \neq D$.

Considere o caso (ii). Nesse caso

$$\left\{\begin{matrix} s_{1_R} \in \mathcal{L}_m(H_R), s_{1_R}\sigma_R \in \mathcal{L}_m(H_R), \\ s_2 \notin \mathcal{L}_m(H_C^D), s_2\sigma \in \mathcal{L}_m(H_C^D) \end{matrix}\right\} \Rightarrow \left\{\begin{matrix} \exists x_R \in X_{H_R}, x_R \in X_{m,H_R} : \sigma_R \in \Gamma_{H_R}(x_R) \text{ e} \\ \exists x_C^D \in X_{H_C^D}, x_C^D \notin X_{m,H_C^D} : \sigma \in \Gamma_{H_C^D}(x_C^D) \end{matrix}\right\}$$

Portanto $f_V(x_{0,V}, s') = (x_R, x_C^D) = x$ e, dessa forma, $f_V(x,\sigma) = D$ para $x \neq D$. Além disso, como $\sigma_R \in \Gamma_{H_R}(x_R) \backslash E$, então $\sigma_R \in \Gamma_V(x)$.

$(\Leftarrow)$ Suponha que existe um autômato V construído de acordo com o algoritmo 3.5, no qual existe $\sigma \in E_c$ tal que $f_V(x,\sigma) = D$ e $x \neq D$. Em outras palavras, $\exists s' \in \mathcal{L}(V), s' \notin \mathcal{L}_m(V)$ tal que $s'\sigma \in \mathcal{L}_m(V)$. Pela construção de V, pode-se afirmar que $s' \in \mathcal{L}(V_{RC})$, $s' \notin \mathcal{L}_m(V_{RC})$ e $s'\sigma \in \mathcal{L}_m(V_{RC})$.

Considere agora o autômato V_{RC}. De acordo com as equações (3.1) e (3.2),

tem-se que

$$\mathcal{L}(V_{RC}) = P_R'^{-1}[\mathcal{L}(H_R)] \cap P'^{-1}[\mathcal{L}(H_C^D)],$$
$$\mathcal{L}_m(V_{RC}) = P_R'^{-1}[\mathcal{L}_m(H_R)] \cap P'^{-1}[\mathcal{L}_m(H_C^D)].$$

Portanto, $\exists s_{1_R} \in \mathcal{L}(H_R)$ e $s_2 \in \mathcal{L}(H_C^D)$ tais que $s' \in P_R'^{-1}(s_{1_R}) \cap P'^{-1}(s_2)$. Além disso, note que $P_o'(s') = P_{R_o}(s_{1_R}) = P(s_2) = s_P$, uma vez que s' foi gerado a partir da adição de sequências não-observáveis sobre duas sequências com mesma projeção observável. Assim, a projeção observável de s' será a mesma das projeções observáveis de s_{1_R} e s_2.

Pela existência de $s_P = P_o'(s') = P_{R_o}(s_{1_R}) = P(s_2)$, é possível concluir que existem duas sequências $s_1 = R^{-1}(s_{1_R})$ e s_2 pertencentes a $\overline{K}$.

Suponha que $\sigma \in E_o$. Como $s' \notin \mathcal{L}_m(V_{RC})$ e $s'\sigma \in \mathcal{L}_m(V_{RC})$, então existem sequências $s_{1_R}\sigma \in \mathcal{L}_m(H_R)$ e $s_2\sigma \in \mathcal{L}_m(H_C^D)$. Logo, pela construção de H_R e H_C^D, $R^{-1}(s_{1_R})\sigma = s_1\sigma \in \overline{K}$ e $s_2\sigma \in M\backslash\overline{K}$, o que implica que $\overline{K}$ é não-observável em relação a M, E_o e E_c.

Suponha agora que $(\sigma \notin E_o) \wedge (R(\sigma) \in \Gamma_V(x))$. Como $s' \notin \mathcal{L}_m(V_{RC})$ e $s'\sigma \in \mathcal{L}_m(V_{RC})$, então a ativação de σ em $f_V(x_{0,V}, s')$ está atrelada à ocorrência assíncrona de σ em H_C^D. Logo, pela construção de H_C^D, $s_2\sigma \in M\backslash\overline{K}$. Além disso, se $\sigma_R = R(\sigma) \in \Gamma_V(x)$ para $x = f_V(x_{0,V}, s')$, então esta ativação estará atrelada à ocorrência assíncrona de σ_R em H_R após s_{1_R}. Logo, pela construção de H_R, $R^{-1}(s_{1_R})\sigma = s_1\sigma \in \overline{K}$. Dessa forma, podemos concluir que $\overline{K}$ é não-observável em relação a M, E_o e E_c. □

Exemplo 3.5. *Considere novamente o sistema não-observável analisado no exemplo 3.1. A execução do algoritmo 3.5 é feita da seguinte forma. O primeiro passo consiste em criar G_m e H_m com todos os estados marcados, conforme figuras 3.7a e 3.7b. O segundo passo consiste na renomeação dos eventos não-observáveis da especificação, resultando no diagrama de transição mostrado na figura 3.7c, onde se pode ver que o evento u foi substituído por u_R. O passo seguinte trata, então, da construção do autômato complementar H_C (figura 3.7d), isto é, que marca $M\backslash\overline{K}$. Isto fica evidenciado pelo fato de conter exatamente a sequência bu que se deseja evitar. Note que, a ocorrência de b já é suficiente para sair da especificação, o que fica evidenciado pela chegada a um estado marcado. A operação seguinte resulta na reunião dos estados marcados em um único estado D (figura 3.7e). O último passo cria o verificador, levando à composição paralela mostrada na figura 3.7f, e, por fim, reunindo os estados marcados em D, resulta no autômato V (figura 3.7g). Ao analisá-lo, é possível notar que chega-se ao estado marcado D por meio de um evento controlável e observável b, o que viola a condição de observabilidade.*

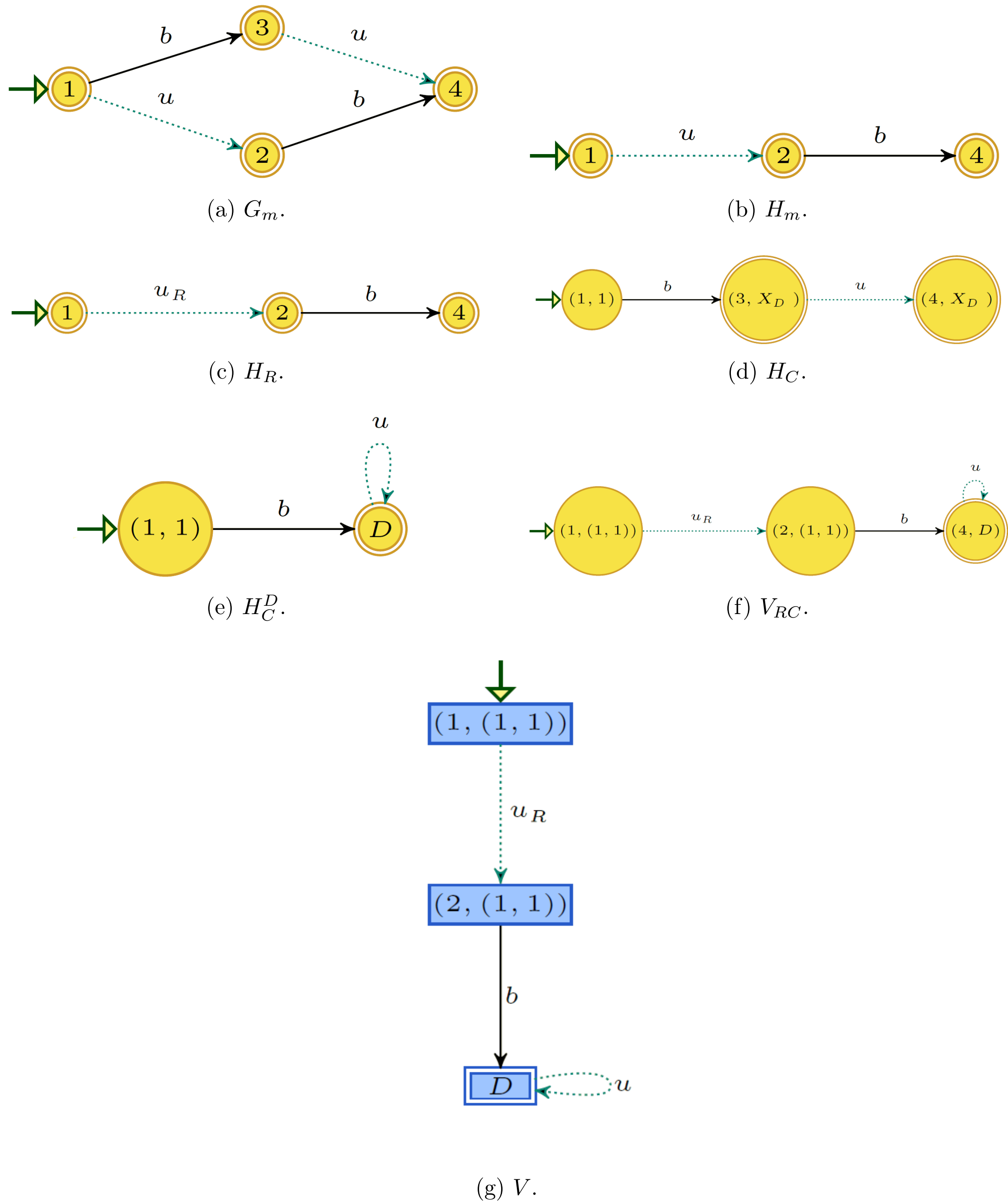


Figura 3.7: Autômatos do exemplo 3.5.

3.2.2 Complexidade Computacional

A análise da complexidade do algoritmo 3.5 é determinada pela análise dos passos que levam à obtenção do autômato V, uma vez que a verificação das condições de violação, realizada no passo 5, é linear em relação aos eventos não-observáveis, e se baseia apenas nas transições que levam ao estado D.

A tabela 3.1 contém o número máximo de estados e transições dos autômatos que necessitam ser gerados para a obtenção de V. Como entrada para esse cálculo são considerados, para G e H, os conjuntos de estados no pior caso, ambos dados por X, e os conjuntos de eventos, ambos dados por E. É imediato chegar aos autômatos G_m e H_m, que se diferenciam de G e H por conterem todos os estados marcados, possuindo assim a mesma cardinalidade de estados e transições de G e H, respectivamente. O espaço de estados do autômato resultante da renomeação dos eventos não-observáveis, H_R, também terá a mesma cardinalidade de H. Por sua vez, o autômato complementar de H_m, H_m^C, terá um estado a mais que H_m. Já o autômato H_C, que marca a linguagem $M \backslash \overline{K}$ possuirá, no máximo, o dobro de estados de G, uma vez que ele é resultado do produto de dois autômatos tais que: (i) a linguagem gerada pelo segundo está contida na linguagem do primeiro; (ii) o conjunto de estados do segundo é um subconjunto do conjunto de estados do primeiro, sendo que os estados pertencentes ao conjunto diferença estão representados no segundo por um único estado X_D. Na sequência, a construção de H_C^D é apenas uma unificação dos estados marcados, o que remove todas as duplicidades de estados contendo X_D, voltando ao pior caso de cardinalidade $|X| + 1$ para o espaço de estados. A operação de composição paralela, que gera o autômato V_{RC}, leva o espaço de estados ao tamanho $|X|^2 + |X|$ e o de transições à $(|X|^2 + |X|) \times (|E| + |E_{uo}|)$, uma vez que o conjunto de eventos de V_{RC} é formado por E mais todas as versões renomeadas dos eventos não-observáveis. Por fim, a unificação dos estados marcados em D gera um verificador V com espaço de estados de tamanho $|X|^2 + 1$ e número de transições igual a $(|X|^2 + 1) \times (|E| + |E_{uo}|)$. Assim sendo, o algoritmo 3.5 possui complexidade $\mathcal{O}(|X|^2)$ no número de estados, e $\mathcal{O}(|X|^2|E|)$ no número de transições. Assim, como os mais eficientes algoritmos existentes mencionados em 3.1, a complexidade do algoritmo aqui proposto também é polinomial.

Indo além, numa comparação direta de desempenho entre o algoritmo 3.5 e o mais eficiente existente proposto por WANG *et al.* [12], conclui-se, usando a notação $\mathcal{O}$, que o algoritmo proposto neste trabalho possui a mesma complexidade no pior caso. Porém, ao comparar em detalhes a complexidade desses algoritmos, é possível afirmar que, como ressaltado no método da seção 3.1.1, a tarefa de verificar a diagnosticabilidade de um autômato é executada por um algoritmo com o mesmo funcionamento do aqui proposto, resultando inclusive em mesma complexidade pela

Tabela 3.1: Complexidade Computacional do Algoritmo 3.5.

Autômato	Número de Estados	Número de Transições
G	$\lvert X\rvert$	$\lvert X\rvert \times \lvert E\rvert$
H	$\lvert X\rvert$	$\lvert X\rvert \times \lvert E\rvert$
G_m	$\lvert X\rvert$	$\lvert X\rvert \times \lvert E\rvert$
H_m	$\lvert X\rvert$	$\lvert X\rvert \times \lvert E\rvert$
H_R	$\lvert X\rvert$	$\lvert X\rvert \times \lvert E\rvert$
H_m^C	$\lvert X\rvert + 1$	$(\lvert X\rvert + 1) \times \lvert E\rvert$
H_C	$2\lvert X\rvert$	$(2\lvert X\rvert) \times \lvert E\rvert$
H_C^D	$\lvert X\rvert + 1$	$(\lvert X\rvert + 1) \times \lvert E\rvert$
V_{RC}	$\lvert X\rvert^2 + \lvert X\rvert$	$(\lvert X\rvert^2 + \lvert X\rvert) \times (\lvert E\rvert + \lvert E_{uo}\rvert)$
V	$\lvert X\rvert^2 + 1$	$(\lvert X\rvert^2 + 1) \times (\lvert E\rvert + \lvert E_{uo}\rvert)$
Complexidade		$\mathcal{O}(\lvert X\rvert^2\lvert E\rvert)$

notação $\mathcal{O}$. Com isto em mente, pode-se entender a necessidade de executar o algoritmo de conversão a priori como um trabalho excedente, e que leva como entrada ao verificador de diagnose um autômato aumentado em até $|E|$ estados e $4|E|+1$ transições, o que não ocorre no algoritmo aqui proposto.

Como será visto a seguir, o verificador resultante do algoritmo 3.5 permite também verificar a normalidade de uma linguagem regular, sem que isto implique em um aumento de complexidade.

3.3 Verificação da Normalidade

Motivado pelo resultado obtido na seção 3.2, e dado o fato de que a normalidade é uma restrição da observabilidade, vamos propor nesta seção uma extensão do algoritmo 3.5. Novamente, a ideia principal por trás do algoritmo proposto é a renomeação dos eventos não-observáveis.

3.3.1 Algoritmo para Verificação

Seja $K \subseteq M = \overline{M}$. É sabido que a linguagem K é normal em relação a M e E_o se $\overline{K} = P^{-1}[P(\overline{K})] \cap M$, ou, equivalentemente, se não existem sequências $s' \in \overline{K}$ e $s \in M \backslash \overline{K}$ tais que $P(s) = P(s')$. É fácil observar que esta condição é imediatamente identificada pelo verificador proposto para a observabilidade (seção 3.2.1), uma vez que de acordo com seu princípio de funcionamento, a existência de sequências de mesma projeção observável ($P(s) = P(s')$), tais que $s' \in \overline{K}$ e $s \in M \backslash \overline{K}$ implica na existência de estados marcados neste verificador. Este fato sugere o algoritmo a seguir.

Algoritmo 3.6. *Sejam os autômatos* $G = (X_G, E, f_G, \Gamma_G, x_0, X_{m,G})$ *e*

$H = (X_H, E, f_H, \Gamma_H, x_0, X_{m,H})$ tais que $\mathcal{L}(G) = M = \overline{M}$, $\mathcal{L}_m(H) = K$ e $H \sqsubseteq G$. Seja o conjunto de eventos observáveis representado por $E_o \subseteq E$.

Passo 1: *Construa o autômato verificador V de acordo com o algoritmo 3.5.*

Passo 2: *Se $\mathcal{L}_m(V) = \emptyset$, então K é normal em relação a M e E_o. Caso contrário, K não é normal em relação a M e E_o.*

□

O teorema a seguir demonstra a corretude do algoritmo 3.6 na verificação da normalidade.

Teorema 3.2. *Sejam K e $M = \overline{M}$ ($K \subseteq M$) linguagens regulares tais que $\mathcal{L}_m(H) = K$, $\mathcal{L}(G) = M$, e seja $V = (X_V, E_V, f_V, \Gamma_V, x_{0,V}, X_{m,V})$ um autômato verificador construído de acordo com o algoritmo 3.6. Então K será normal com respeito a M e E_o se e somente se*

$$\mathcal{L}_m(V) = \emptyset.$$

Demonstração.

($\Rightarrow$) Suponha que K não seja normal com respeito a M e E_o. Então, pela definição de normalidade, existem sequências $s_1 \in \overline{K}$ e $s_2 \in M \backslash \overline{K}$ tais que $P(s_1) = P(s_2)$ e $s_1 \neq s_2$.

Pela construção de H_R, como $s_1 \in \overline{K}$ então $s_1 \in \mathcal{L}_m(H_m)$. Seja $s_{1_R} = R(s_1)$. Então $s_{1_R} \in \mathcal{L}_m(H_R) = \mathcal{L}(H_R)$.

Considere a operação de projeção P_{R_o} definida por

$$P_{R_o} : E_R^* \rightarrow E_o^*.$$

Assim $P(s_1) = P_{R_o}(s_{1_R})$, uma vez que a função de renomeação não altera a observabilidade de um evento.

Pela construção de H_C^D, $\mathcal{L}(H_C) \subseteq \mathcal{L}(H_C^D)$ e $\mathcal{L}_m(H_C) \subseteq \mathcal{L}_m(H_C^D)$. Assim, como $s_2 \in M \backslash \overline{K}$, então $s_2 \in \mathcal{L}_m(H_C^D)$.

Seja $E' = E_R \cup E$ e defina as operações de projeção P', P'_R e P'_o da seguinte forma:

$$P' : E'^* \rightarrow E^*;$$
$$P'_R : E'^* \rightarrow E_R^*;$$
$$P'_o : E'^* \rightarrow E_o^*.$$

Considere, agora, o autômato V_{RC}. É fácil verificar que $\mathcal{L}(V_{RC}) \subseteq \mathcal{L}(V)$ e

$\mathcal{L}_m(V_{RC}) \subseteq \mathcal{L}_m(V)$. Como $V_{RC} = H_R \| H_C^D$ então:

$$\mathcal{L}(V_{RC}) = {P'_R}^{-1}[\mathcal{L}(H_R)] \cap P'^{-1}[\mathcal{L}(H_C^D)] \subseteq \mathcal{L}(V), \tag{3.3}$$

$$\mathcal{L}_m(V_{RC}) = {P'_R}^{-1}[\mathcal{L}_m(H_R)] \cap P'^{-1}[\mathcal{L}_m(H_C^D)] \subseteq \mathcal{L}_m(V). \tag{3.4}$$

Como visto acima, $s_{1_R} \in \mathcal{L}_m(H_R)$ e $s_2 \in \mathcal{L}_m(H_C^D)$. Uma vez que s_{1_R} difere de s_2 apenas por eventos não-observáveis é possível afirmar que $\exists s' \in {P'_R}^{-1}(s_{1_R}) \cap P'^{-1}(s_2)$, e portanto $s' \in \mathcal{L}_m(V)$. Isto implica que $\mathcal{L}_m(V) \neq \emptyset$.

($\Leftarrow$) Suponha, agora, que $\mathcal{L}_m(V) \neq \emptyset$. Logo $\exists s' \in \mathcal{L}_m(V)$ e, pela construção de V, pode-se afirmar que $s' \in \mathcal{L}_m(V_{RC})$.

Considere, agora, o autômato V_{RC}. De acordo com as equações (3.3) e (3.4), tem-se que

$$\mathcal{L}(V_{RC}) = {P'_R}^{-1}[\mathcal{L}(H_R)] \cap P'^{-1}[\mathcal{L}(H_C^D)],$$
$$\mathcal{L}_m(V_{RC}) = {P'_R}^{-1}[\mathcal{L}_m(H_R)] \cap P'^{-1}[\mathcal{L}_m(H_C^D)].$$

Portanto, $\exists s_{1_R} \in \mathcal{L}_m(H_R)$ e $s_2 \in \mathcal{L}_m(H_C^D)$ tais que $s' \in {P'_R}^{-1}(s_{1_R}) \cap P'^{-1}(s_2)$. Além disso, nota-se que $P'_o(s') = P_{R_o}(s_{1_R}) = P'_o(s_2) = s_P$, uma vez que s' foi gerado a partir da adição de sequências não-observáveis sobre duas sequências com mesma projeção observável. Assim, a projeção observável de s' será a mesma das projeções observáveis de s_{1_R} e s_2.

Pela existência de $s_P = P'_o(s') = P_{R_o}(s_{1_R}) = P'_o(s_2)$ é possível concluir que existem sequências $s_1 = R^{-1}(s_{1_R}) \in \overline{K}$ e $s_2 \in M \setminus \overline{K}$. Além disso, pode-se concluir que $s_1 \neq s_2$, uma vez que, por construção, $\mathcal{L}_m(H_R) = R(\mathcal{L}_m(H_m)) = R(\overline{K})$, e $M \setminus \overline{K} = \mathcal{L}_m(H_C) \subseteq \mathcal{L}_m(H_C^D)$, o que conclui a prova do teorema. □

Exemplo 3.6. *Considere novamente o sistema não-observável analisado no exemplo 3.1. Como visto em detalhes no exemplo 3.4, a normalidade também não é verificada para este sistema. A execução do algoritmo 3.6 leva à construção do verificador V ilustrado na figura 3.7g. Nesta é possível verificar que $\mathcal{L}_m(V) \neq \emptyset$, o que novamente permite concluir que K não é normal em relação a M e E_o.*

3.3.2 Complexidade Computacional

Dado que o mesmo verificador é utilizado tanto para a verificação da normalidade quanto para a verificação da observabilidade, e que a condição $\mathcal{L}_m(V) = \emptyset$ pode ser analisada de maneira imediata, os resultados para a complexidade do verificador de observabilidade também se aplicam ao verificador de normalidade, tendo, portanto, complexidade polinomial $\mathcal{O}(|X|^2)$ no número de estados, e $\mathcal{O}(|X|^2|E|)$ no número

de transições. Note que a complexidade polinomial obtida representa um grande avanço quando comparada à complexidade exponencial do algoritmo 3.4, que foi construído de maneira imediata a partir da definição de normalidade.

3.4 Exemplos ilustrativos

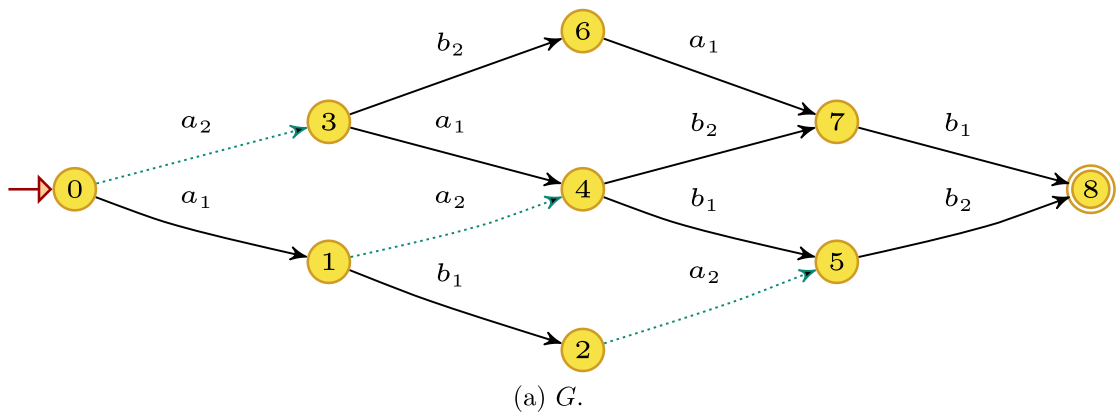
Visando ilustrar a utilização e a eficácia do verificador aqui proposto, vamos apresentar nesta seção alguns exemplos.

3.4.1 Exemplo 1

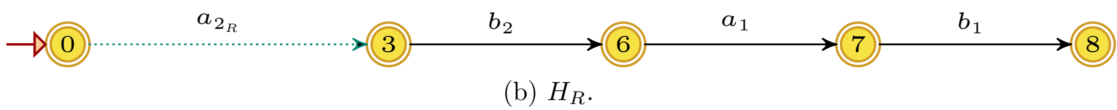
Considere o autômato G da figura 3.8a, e seja o sistema definido pela linguagem $M = \overline{M} = \mathcal{L}(G)$, no qual $E = \{a_1, b_1, a_2, b_2\}$, $E_c = \{a_1, b_1, a_2, b_2\}$, $E_o = \{a_1, b_1, b_2\}$ e, como consequência, $E_{uo} = \{a_2\}$. Seja a linguagem especificada $K = \{a_2 b_2 a_1 b_1\} = \mathcal{L}_m(H)$. A construção do verificador V, de acordo com o algoritmo 3.5, se inicia pela construção dos autômatos G_m e H_m, nos quais $\mathcal{L}_m(G_m) = \overline{M}$ e $\mathcal{L}_m(H_m) = \overline{K}$. Prosseguindo com a execução do algoritmo, é construído o autômato H_R, ilustrado na figura 3.8b, no qual a ocorrência do evento não-observável a_2 é renomeada para a_{2_R}. A continuação da execução do algoritmo leva, então, à obtenção do autômato H_C, que marca $M \backslash \overline{K}$, e em seguida, com a reunião dos estados marcado em um único estado D, chega-se ao autômato H_C^D representado na figura 3.8c. Note que a ocorrência do evento a_1 nos estados 0 e 3 leva o sistema para fora da especificação. O último passo do algoritmo 3.5 realiza a operação de composição paralela $H_R \| H_C^D$, levando à construção do verificador V da figura 3.8d.

A busca, em V, por eventos $\sigma \in E_c$ que violem a condição de observabilidade dada pelo algoritmo 3.5 falha devido à inexistência de um estado D, pois $\mathcal{L}_m(V) = \emptyset$. Com isto é possível concluir que K é observável em relação a M, E_c e E_o.

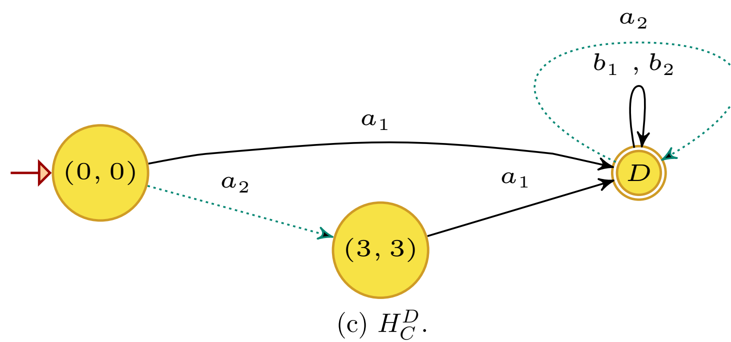
Para a verificação da normalidade (algoritmo 3.6) faz-se uso do mesmo autômato verificador V. Uma vez que $\mathcal{L}_m(V) = \emptyset$, a condição dada pelo algoritmo é satisfeita, e conclui-se que K é normal em relação a M e E_o.



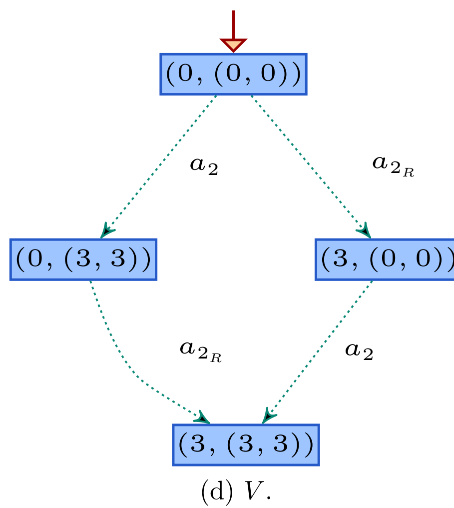
(a) G.



(b) H_R.



(c) H_C^D.



(d) V.

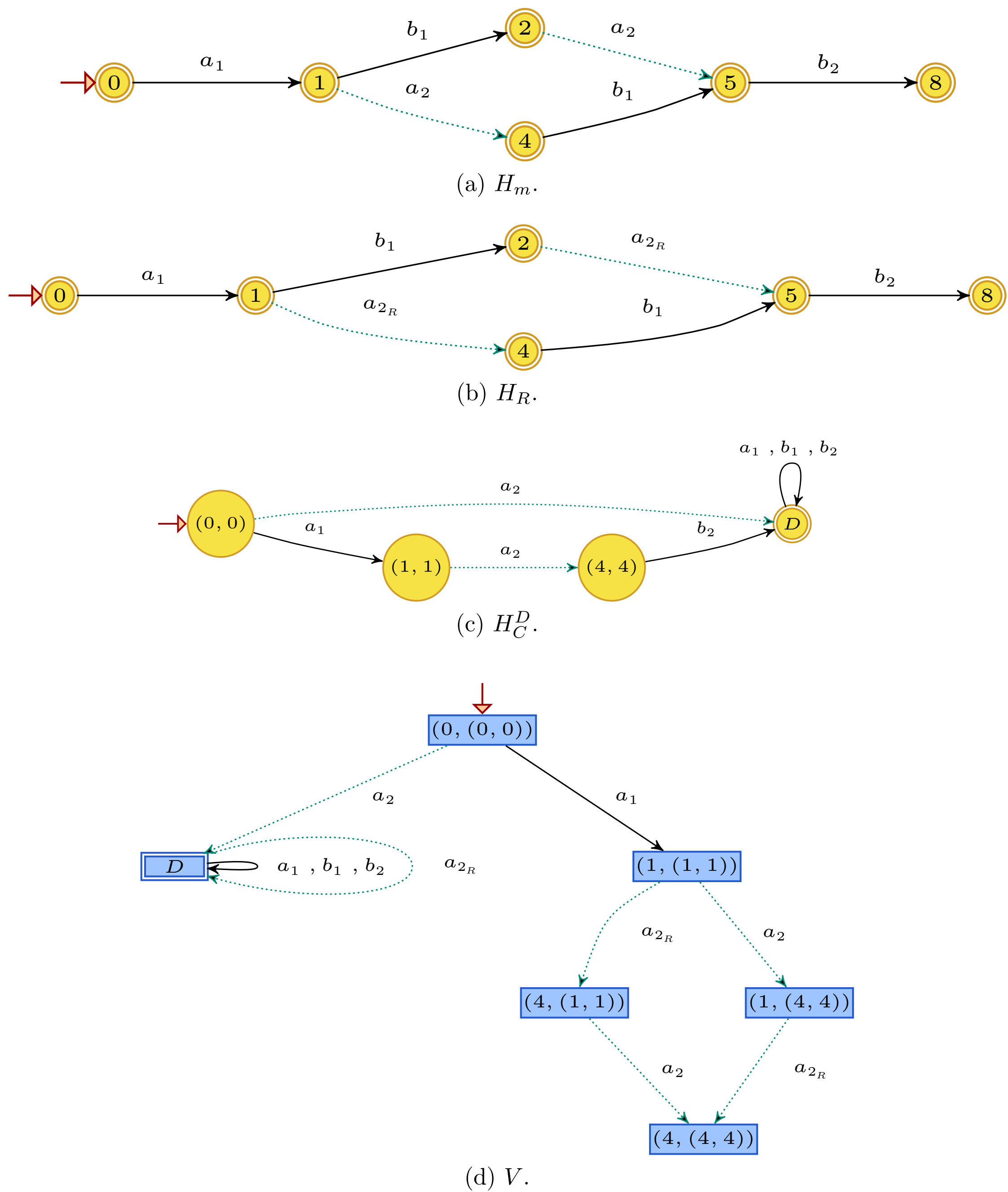
Figura 3.8: Autômatos do Exemplo 1.

3.4.2 Exemplo 2

Considere novamente o sistema definido pela linguagem $M = \overline{M} = \mathcal{L}(G)$, no qual G é o autômato da figura 3.8a. Seja $E = \{a_1, b_1, a_2, b_2\}$, $E_c = \{a_1, b_1, a_2, b_2\}$, $E_o = \{a_1, b_1, b_2\}$ e, como consequência, $E_{uo} = \{a_2\}$. Seja a linguagem especificada $K = \{a_1b_1a_2b_2, a_1a_2b_1b_2\} = \mathcal{L}_m(H)$. A construção do verificador V, de acordo com o algoritmo 3.5, se inicia pela construção dos autômatos G_m e H_m, nos quais $\mathcal{L}_m(G_m) = \overline{M}$ e $\mathcal{L}_m(H_m) = \overline{K}$. Prosseguindo com a execução do algoritmo, é construído o autômato H_R, ilustrado na figura 3.9b, no qual as ocorrências do evento não-observável a_2 são renomeadas para a_{2_R}. A continuação da execução do algoritmo leva, então, à obtenção do autômato H_C, que marca $M \backslash \overline{K}$, e em seguida, com a reunião dos estados marcado em um único estado D, chega-se ao autômato H_C^D representado na figura 3.9c. Devido à modificação da especificação em relação ao Exemplo 1 nota-se que, agora, as ocorrências de a_2 no estado 0 e de b_2 no estado 4 são as transições que levam o sistema para fora da especificação. O último passo do algoritmo 3.5 realiza a operação de composição paralela $H_R \| H_C^D$, levando à construção do verificador V da figura 3.9d.

A busca, em V, por eventos $\sigma \in E_c$ que violem a condição de observabilidade dada pelo algoritmo 3.5 identifica apenas o evento a_2 como candidato. Como $a_2 \in E_c$ mas $a_2 \notin E_o$, para concluir sobre a observabilidade de K em relação a M, E_c e E_o faz-se necessário verificar se $R(a_2) \in \Gamma_V((0,(0,0)))$. Como $a_{2_R} \notin \{a_1, a_2\}$, não existe a necessidade concorrente de habilitar a_2 para permanecer na especificação e desabilitar a_2 para não sair da mesma. Logo, conclui-se que K é observável em relação a M, E_c e E_o.

Para a verificação da normalidade (algoritmo 3.6) faz-se uso do mesmo autômato verificador V. Uma vez que $\mathcal{L}_m(V) \neq \emptyset$, a condição dada pelo algoritmo não é satisfeita, o que permite concluir que K não é normal em relação a M e E_o.



(a) H_m.

(b) H_R.

(c) H_C^D.

(d) V.

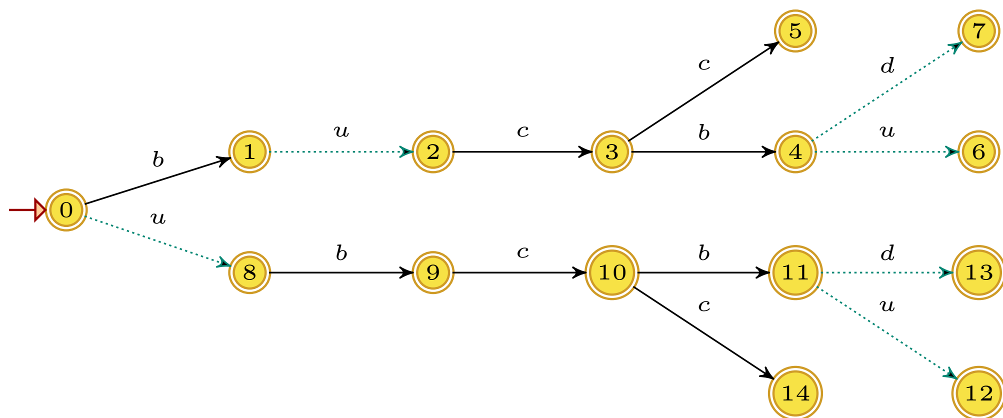
Figura 3.9: Autômatos do Exemplo 2.

3.4.3 Exemplo 3

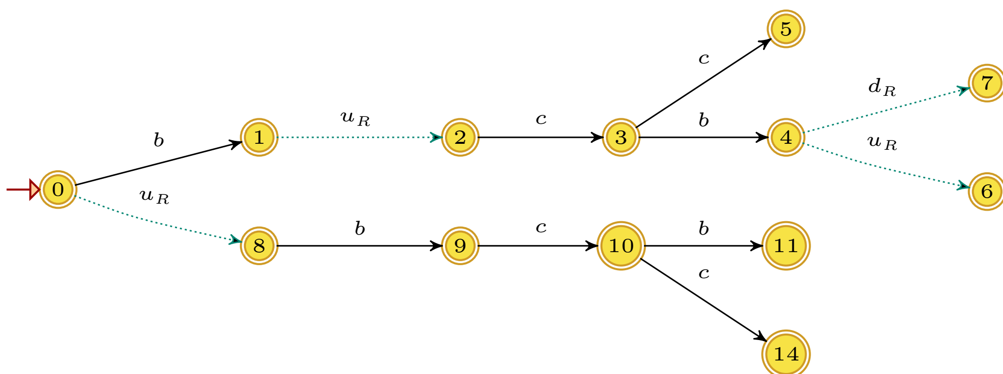
Considere o autômato G da figura 3.10a, e suponha que $M = \overline{M} = \mathcal{L}(G)$, no qual $E = \{u, b, c, d\}$, $E_c = \{b, c, d\}$ e $E_o = \{b, c\}$. Seja $K = \overline{K}$ a linguagem especificada marcada pelo autômato G a menos dos estados 12 e 13 (e suas respectivas transições). A construção do verificador V, de acordo com o algoritmo 3.5, se inicia pela construção dos autômatos $G_m = G$ e $H_m = H$. Prosseguindo com a execução do algoritmo, construímos o autômato H_R, ilustrado na figura 3.10b, no qual as ocorrências do eventos não-observáveis u e d são renomeadas para u_R e d_R, respectivamente. A continuação da execução do algoritmo leva, então, à obtenção do autômato H_C, que marca $M \backslash \overline{K}$, e em seguida, com a reunião dos estados marcado em um único estado D, chega-se ao autômato H_C^D representado na figura 3.10c. Note que as ocorrências dos eventos u e d no estado 11 levam o sistema para fora da especificação. O último passo do algoritmo 3.5 realiza a operação de composição paralela $H_R \| H_C^D$, levando à construção do verificador V da figura 3.11.

A busca, em V, por eventos que violem a condição de observabilidade dada pelo algoritmo 3.5 identifica os eventos u e d como candidatos. Como $u \notin E_c$, este não pode violar a condição de observabilidade, ficando a análise restrita ao evento d. Para que a condição de observabilidade seja violada é necessário que $R(d) \in \Gamma_V(x)$, para algum estado $x \in X_V$ tal que $x \neq D$ e $f_V(x, d) = D$. Essa condição é identificada no estado $(4, (11, 11))$, o que permite concluir que K é não-observável em relação a M, E_c e E_o. Note que a execução das sequências $s_1 = bucb$ ou $s_2 = ubcb$ deixa o supervisor em duvida quanto à decisão a tomar, pois ambas têm a mesma projeção e a primeira requer que o evento d seja habilitado, enquanto a segunda desabilita d.

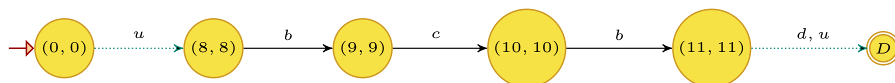
Para a verificação da normalidade (algoritmo 3.6) faz-se uso do mesmo autômato verificador V. Note que, como $\mathcal{L}_m(V) \neq \emptyset$, a condição dada pelo algoritmo não é satisfeita, o que permite concluir que K não é normal em relação a M e E_o.



(a) G.



(b) H_R.



(c) H_C^D.

Figura 3.10: Autômatos G, H_R e H_C^D do Exemplo 3.

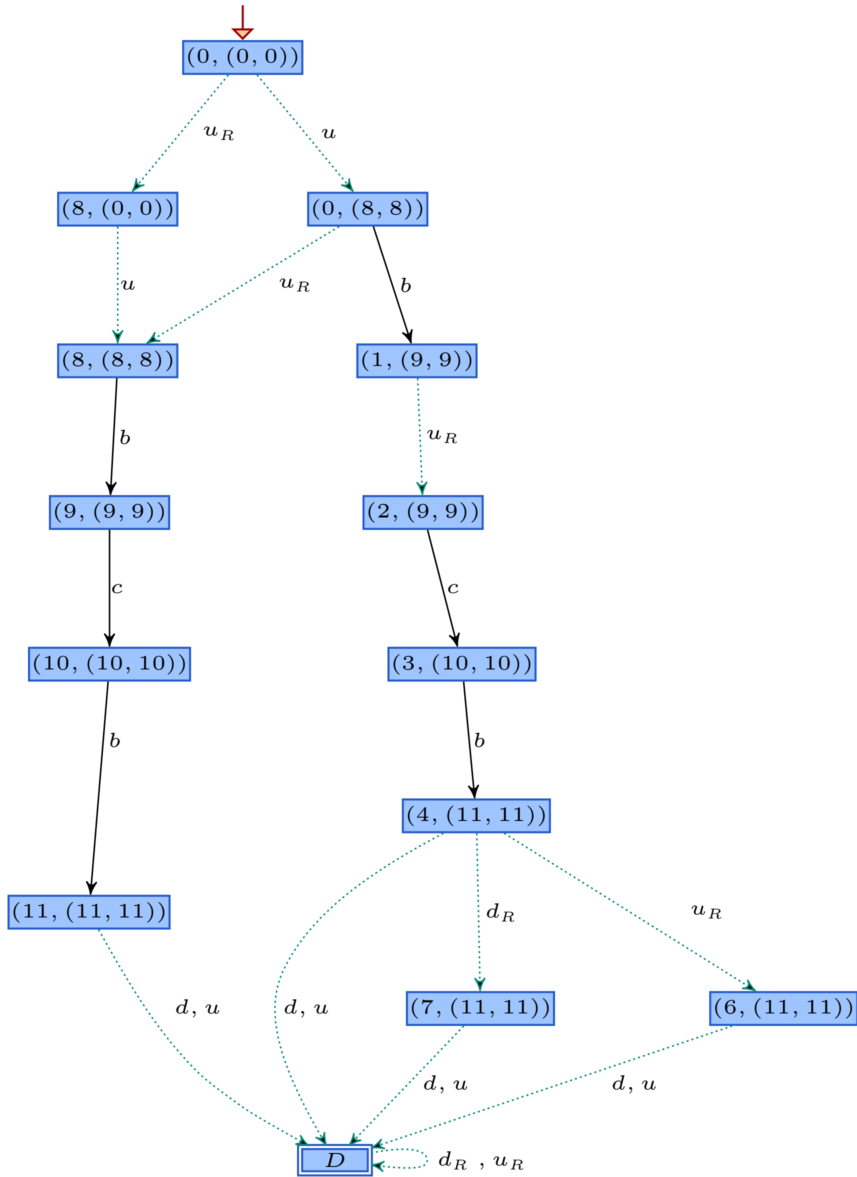


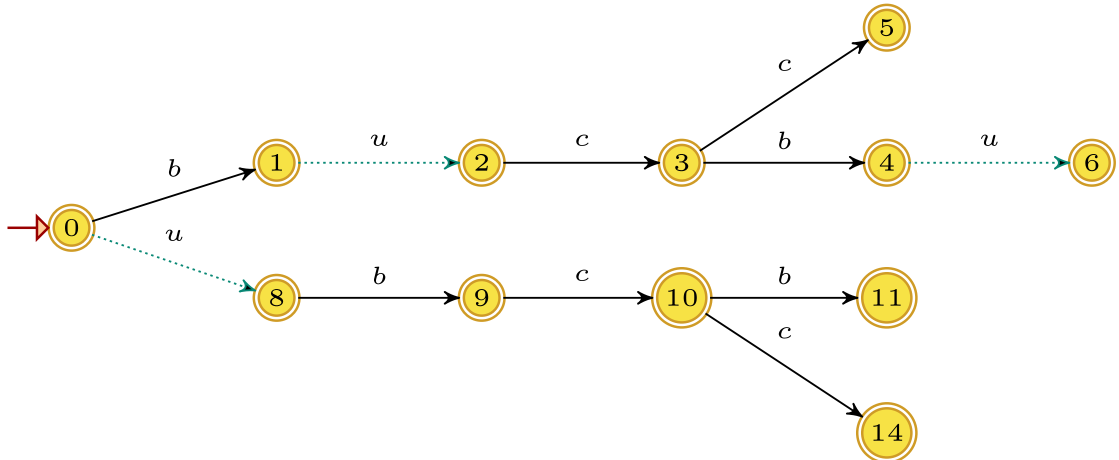
Figura 3.11: Autômato V do Exemplo 3.

3.4.4 Exemplo 4

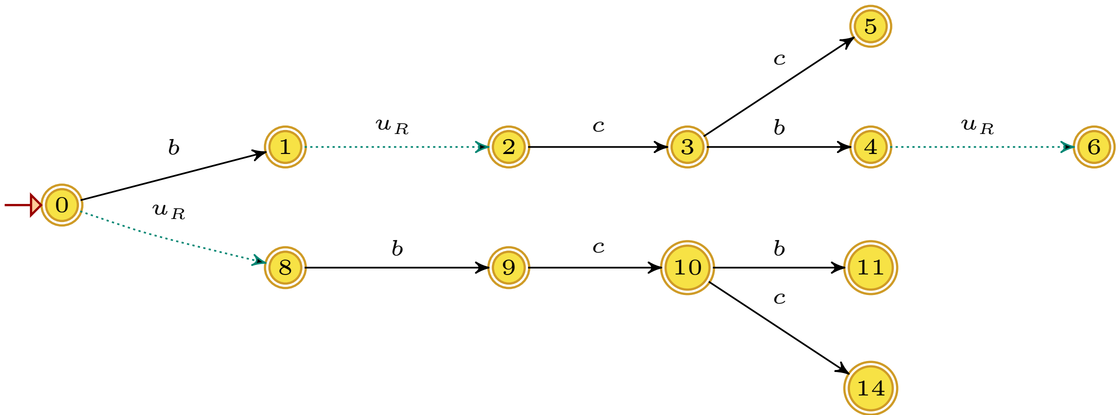
Considere novamente o sistema definido pela linguagem $M = \overline{M} = \mathcal{L}(G)$, no qual G é o autômato da figura 3.10a. Seja $E = \{u, b, c, d\}$, $E_c = \{b, c, d\}$ e $E_o = \{b, c\}$. Seja a linguagem especificada $K = \overline{K} = \mathcal{L}_m(H)$, ilustrada na figura 3.12a. A construção do verificador V, de acordo com o algoritmo 3.5, se inicia pela construção dos autômatos $G_m = G$ e $H_m = H$. Prosseguindo com a execução do algoritmo, constrói-se o autômato H_R, ilustrado na figura 3.12b, no qual as ocorrências do evento não-observável u são renomeadas para u_R. A continuação da execução do algoritmo leva, então, à obtenção do autômato H_C, que marca $M \backslash \overline{K}$, e em seguida, com a reunião dos estados marcado em um único estado D, chega-se ao autômato H_C^D representado na figura 3.12c. Note que, agora, as ocorrências de u e d no estado 11 e de d no estado 4 são as transições que levam o sistema para fora da especificação. O último passo do algoritmo 3.5 realiza a operação de composição paralela $H_R \| H_C^D$, levando à construção do verificador V da figura 3.13.

A busca, em V, por eventos que violem a condição de observabilidade dada pelo algoritmo 3.5 identifica os eventos u e d como candidatos. Como $u \notin E_c$, este não pode violar a condição de observabilidade, ficando a análise restrita ao evento $d \in E_c$. Para que a condição de observabilidade seja violada é necessário que $R(d) \in \Gamma_V(x)$, para algum estado $x \in X_V$ tal que $x \neq D$ e $f_V(x, d) = D$. Esta condição não é identificada em V, o que permite concluir que K é observável em relação a M, E_c e E_o.

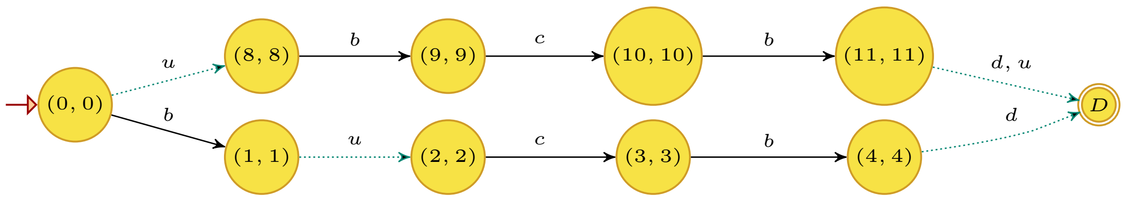
Para a verificação da normalidade (algoritmo 3.6) faz-se uso do mesmo autômato verificador V. Uma vez que $\mathcal{L}_m(V) \neq \emptyset$, a condição dada pelo algoritmo não é satisfeita, o que permite concluir que K não é normal em relação a M e E_o.



(a) H.



(b) H_R.



(c) H_C^D.

Figura 3.12: Autômatos H_m, H_R e H_C^D do Exemplo 4.

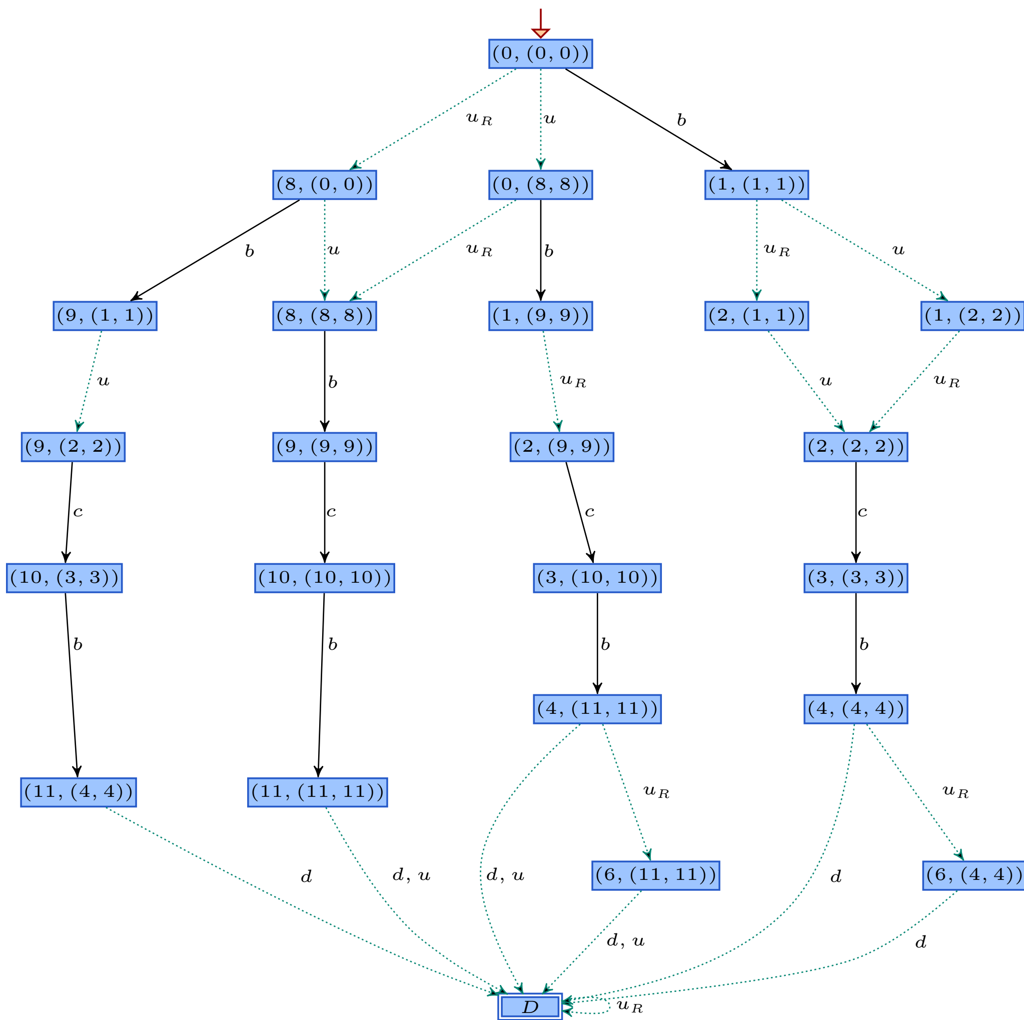


Figura 3.13: Autômato V do Exemplo 4.

3.5 Verificação de Coobservabilidade

Motivados pela obtenção de uma nova abordagem para a verificação das propriedades de observabilidade e normalidade, e do conhecimento do sucesso na generalização para o caso descentralizado do algoritmo proposto por MOREIRA *et al.* [13], no qual os algoritmos aqui apresentados foram inspirados, uma investigação para a utilização da mesma abordagem na verificação da coobservabilidade foi realizado. Nesta seção, os resultados deste estudo são apresentados, com a descrição do principal algoritmo existente e a proposta de um novo verificador. Por fim, para permitir um completo entendimento, exemplos ilustrativos são apresentados.

3.5.1 Revisão Bibliográfica

Com o objetivo de possibilitar a comparação do algoritmo a ser proposto com os existentes na literatura, iremos fazer uma breve revisão dos principais algoritmos existentes.

Nesta seção vamos novamente supor que K e M são linguagens regulares, isto é, existem autômatos $G = (X_G, E, f_G, \Gamma_G, x_0, X_{m,G})$, e $H = (X_H, E, f_H, \Gamma_H, x_0, X_{m,H})$, tais que $\mathcal{L}(G) = M = \overline{M}$, $\mathcal{L}_m(H) = K$, e $H \sqsubseteq G$. Note, novamente, que essa última condição pode ser colocada sem perda de generalidade, uma vez que sempre é possível modificar os autômatos que marcam G e H para torná-la verdadeira, conforme apresentado pelo algoritmo 2.1. Será também considerada a existência de n agentes supervisores possuindo conjuntos de eventos controláveis e observáveis representados respectivamente por $E_{c,i} \subseteq E$ e $E_{o,i} \subseteq E$, $i \in \{1, 2, ..., n\}$.

Considere agora a propriedade da coobservabilidade, e seja $K \subseteq M = \overline{M}$. Como apresentado na definição 2.22, K será coobservável em relação a M, $E_{o,i}$ e $E_{c,i}$, com $i \in \{1, 2, ..., n\}$, se $\forall s \in \overline{K}$ e $\forall \sigma \in E_c = \cup_{i=1}^{n} E_{c,i}$,

$$
\begin{aligned}
&(s\sigma \in M)\ (s\sigma \notin \overline{K}) \Rightarrow \\
&\exists i \in \{1, ..., n\} \text{ tal que } P_i^{-1}[P_i(s)]\sigma \cap \overline{K} = \emptyset \wedge \sigma \in E_{c,i}.
\end{aligned}
$$

Vamos, a seguir, apresentar os principais algoritmos existentes na literatura para verificar a coobservabilidade de linguagens.

- **Algoritmos baseados em TSITSIKLIS [11]**

Como visto na seção 3.1.1, o algoritmo seminal para a verificação da observabilidade foi elaborado por TSITSIKLIS [11]. Baseados no mesmo princípio de funcionamento desse algoritmo, duas generalizações para o caso de controle supervisório descentralizado foram propostas. A primeira, proposta por RUDIE e WILLEMS [22], estende

o algoritmo para a análise de até dois agentes supervisores. A segunda, proposta por HUANG *et al.* [23], utiliza a mesma ideia, porém realiza a análise da coobservabilidade com base em estados, e não em eventos, conforme a teoria apresentada neste trabalho. De qualquer forma, a proposta de HUANG *et al.* [23] também se limita ao caso descentralizado de dois agentes supervisores.

Assim, por não apresentarem uma generalização para problemas com um número indeterminado n de agentes, esses algoritmos não serão analisados em detalhes neste trabalho.

- **Algoritmo proposto por WANG *et al.* [12]**

Embora a análise do trabalho de WANG *et al.* [12], apresentada na seção 3.1.1, tenha se restringido ao caso centralizado, as duas propostas de transformações feitas em WANG *et al.* [12] já contemplam o caso descentralizado. Ambos possuem a mesma finalidade, transformar um problema de verificar a coobservabilidade em um problema de verificar a codiagnosticabilidade. A primeira transformação, chamada COOBS-TO-DIAG-I, resulta na criação de um autômato para cada $e \in E_c = \cup_{i=1}^{n} E_{c,i}$. Cada transformação possui complexidade $\mathcal{O}(|X||E|)$, o que gera a complexidade total de $\mathcal{O}(|X||E|^2)$. Como a segunda transformação, denominada COOBS-TO-DIAG-II, possui a vantagem de gerar apenas um autômato de saída, no qual a verificação de codiagnosticabilidade implica diretamente verificação da coobservabilidade, esta será novamente analisada aqui. Possuindo complexidade $\mathcal{O}(|X||E|^2)$, sua implementação é descrita a seguir.

Algoritmo 3.7 (COOBS-TO-DIAG-II[4])**.** *Sejam G e H autômatos tais que $H \sqsubseteq G$. Para agentes $i \in \{1, 2, ..., n\}$, sejam os conjuntos de eventos observáveis denotados por $E_{o,i}$, e os conjuntos dos eventos controláveis denotados por $E_{c,i}$. Suponha que $z \notin E$ e, para todo $e \in E_c = \cup_{i=1}^{n} E_{c,i}$, $d_e \notin X_H$ e $v_e, f_e, u_e, r_e \notin E$. Considere $E^t = \{z\} \cup \{v_e, f_e, u_e, r_e : e \in E_c\}$. Seja $X_{\tilde{H}} = X_H \cup \{d_e : e \in E_c\}$ e $E_{\tilde{H}(e)} = E \cup E^t$. O autômato $\tilde{H} = (X_{\tilde{H}}, E_{\tilde{H}}, f_{\tilde{H}}, x_0)$ pode ser construído da seguinte forma:*

Passo 1: *Faça $\tilde{H} := H$. Para todo $e \in E_c$, adicione estados não marcados d_e ao espaço de estados de $\tilde{H}$.*

Passo 2: *Para cada estado d_e, adicione o auto-laço definido por $f_{\tilde{H}}(d_e, v_e) = d_e$. Faça $v_e \in E_{o,i}$, para $i \in \{1, 2, ..., n\}$.*

Passo 3: *Para todo $e \in E_c$ e $x \in X_H$, se $e \in \Gamma_G(x) \backslash \Gamma_H(x)$, adicione as transições $f_{\tilde{H}}(x, f_e) = d_e$ e $f_{\tilde{H}}(x, r_e) = d_e$, tais que $f_e \in E_{uo,i}$ para $i \in \{1, 2, ..., n\}$,*

[4] O algoritmo apresentado nesta seção é um caso particular do algoritmo proposto por WANG *et al.* [12], aplicado ao caso com observação estática.

e $r_e \in E_{o,i}$ para i tal que $e \in E_{c,i}$. Se $e \in \Gamma_H(x)$, adicione a transição $f_{\tilde{H}}(x, u_e) = d_e$, em que $u_e \in E_{uo,i}$ para $i \in \{1, 2, ..., n\}$.

Passo 4: *Adicione um auto-laço com evento z em cada estado $x \in X_H \subseteq X$ que seja um* deadlock *em G, em que $z \in E_{o,i}$, para $i \in \{1, 2, ..., n\}$.*

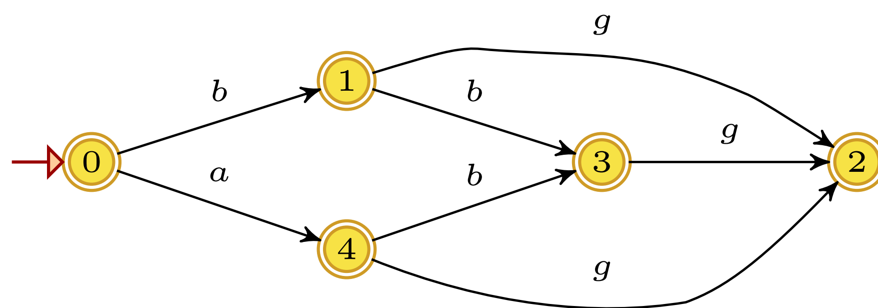
Definindo o conjunto E_f contendo todos os eventos $f_e \in E_{\tilde{H}}$, a análise da codiagnosticabilidade de $\tilde{H}$ em relação a E_f pode, então, ser realizada utilizando o algoritmo de complexidade polinomial proposto em MOREIRA *et al.* [13]. O verificador resultante (V_{diag}) possui complexidade $\mathcal{O}(n|X|^{n+1}|E|)$. Desse modo, pode-se concluir que a verificação da coobservabilidade feita a partir da execução do algoritmo 3.7, seguida da execução do algoritmo de verificação de codiagnosticabilidade de [13], resulta em um algoritmo global de complexidade $\mathcal{O}(|X||E|^2 + n|X_{\tilde{H}}|^{n+1} \times |E_{\tilde{H}}|) = \mathcal{O}(|X||E|^2 + n(|X| + |E|)^{n+1} \times (5|E| + 1)) = \mathcal{O}(n|X|^{n+1}|E|)$ (uma vez que tipicamente $|X| >> |E|$).

Exemplo 3.7. *Seja o SED que se deseja controlar definido pela linguagem regular $M = \overline{\{bg, bbg, ag, abg\}}$ e conjunto de eventos $E = \{a, b, g\}$, o qual se encontra modelado pelo autômato G ilustrado na figura 3.14a ($\mathcal{L}(G) = \mathcal{L}_m(G) = M$). Seja a linguagem regular, e prefixo-fechada, $K = \overline{\{bg, bb, ab\}}$, a linguagem especificada para o sistema, modelada pelo autômato H ilustrado na figura 3.14b.*

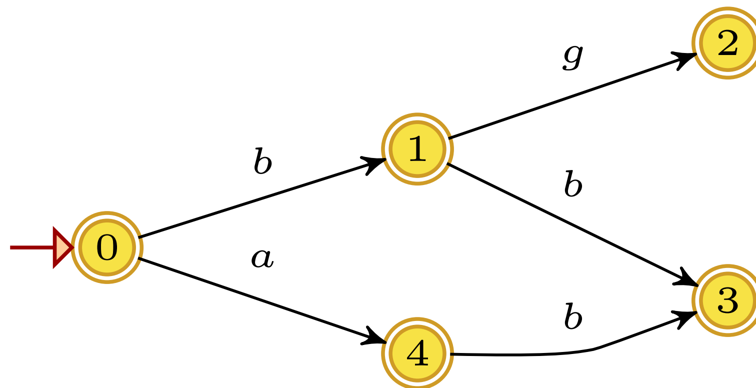
Considere, então, que, para manter o sistema dentro da especificação desejada, uma arquitetura de controle supervisório descentralizada é projetada, sendo constituída por dois agentes. Os conjuntos de eventos controláveis e observáveis por cada agente são definidos por: $E_{c,1} = \{a, g\}$, $E_{c,2} = \{b, g\}$, $E_{o,1} = \{a\}$ e $E_{o,2} = \{b\}$.

É possível notar que este sistema é coobservável, pois a ocorrência do evento a no estado inicial, observada pelo agente 1, permite que o agente 1 desabilite o evento g tanto no estado 3 como no estado 4. Já a ocorrência do evento b no estado inicial, observada pelo agente 2, permite que o agente 2 desabilite o evento g no estado 3, mantendo o sistema na especificação.

Considere, então, a utilização do método proposto por WANG et al. *[12]. Com o uso da transformação do algoritmo 3.7 chega-se ao autômato $\tilde{H}$, ilustrado na figura 3.15. Na sequência, o emprego do algoritmo [13] permite obter o verificador V_{diag} ilustrado na figura 3.16. A busca de ciclos que violem a condição de codiagnosticabilidade falha. Assim, como esperado, é possível concluir que a linguagem K é coobservável em relação a M, $E_{c,1}$, $E_{c,2}$, $E_{o,1}$ e $E_{o,2}$.*



(a) G.



(b) H.

Figura 3.14: Autômatos de entrada para o exemplo 3.7.

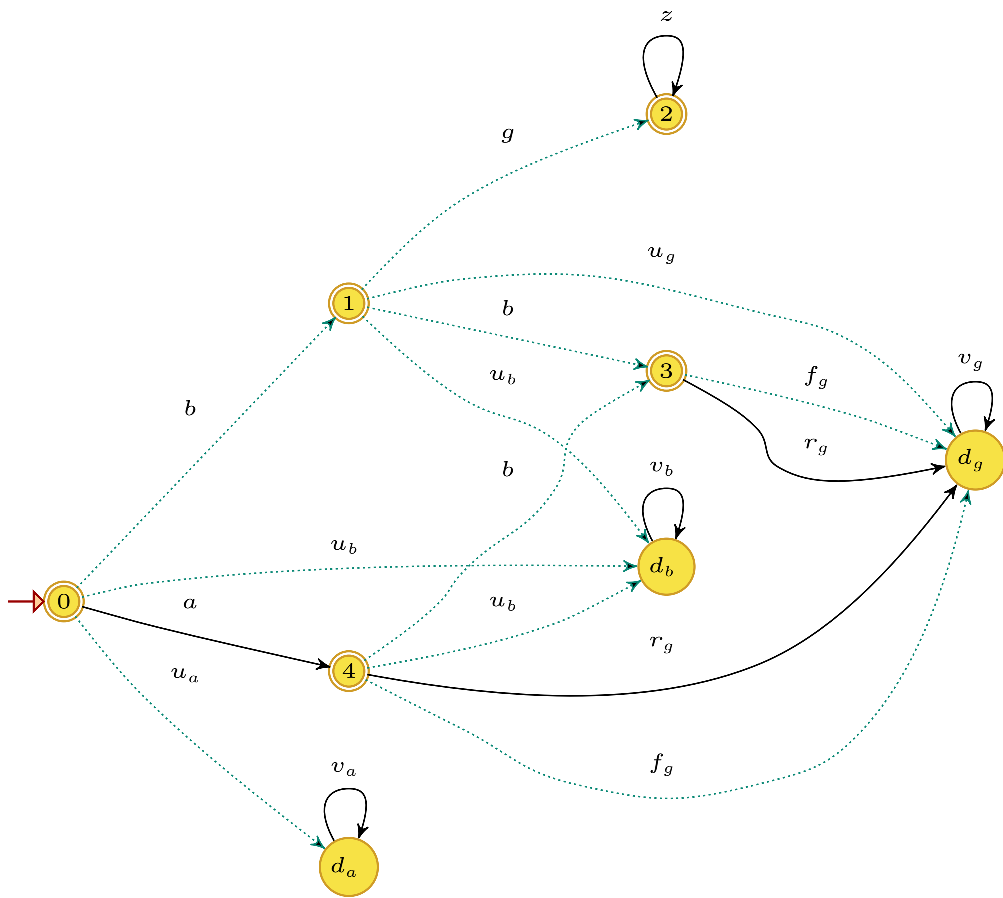


Figura 3.15: Autômato $\tilde{H}$ do exemplo 3.7.

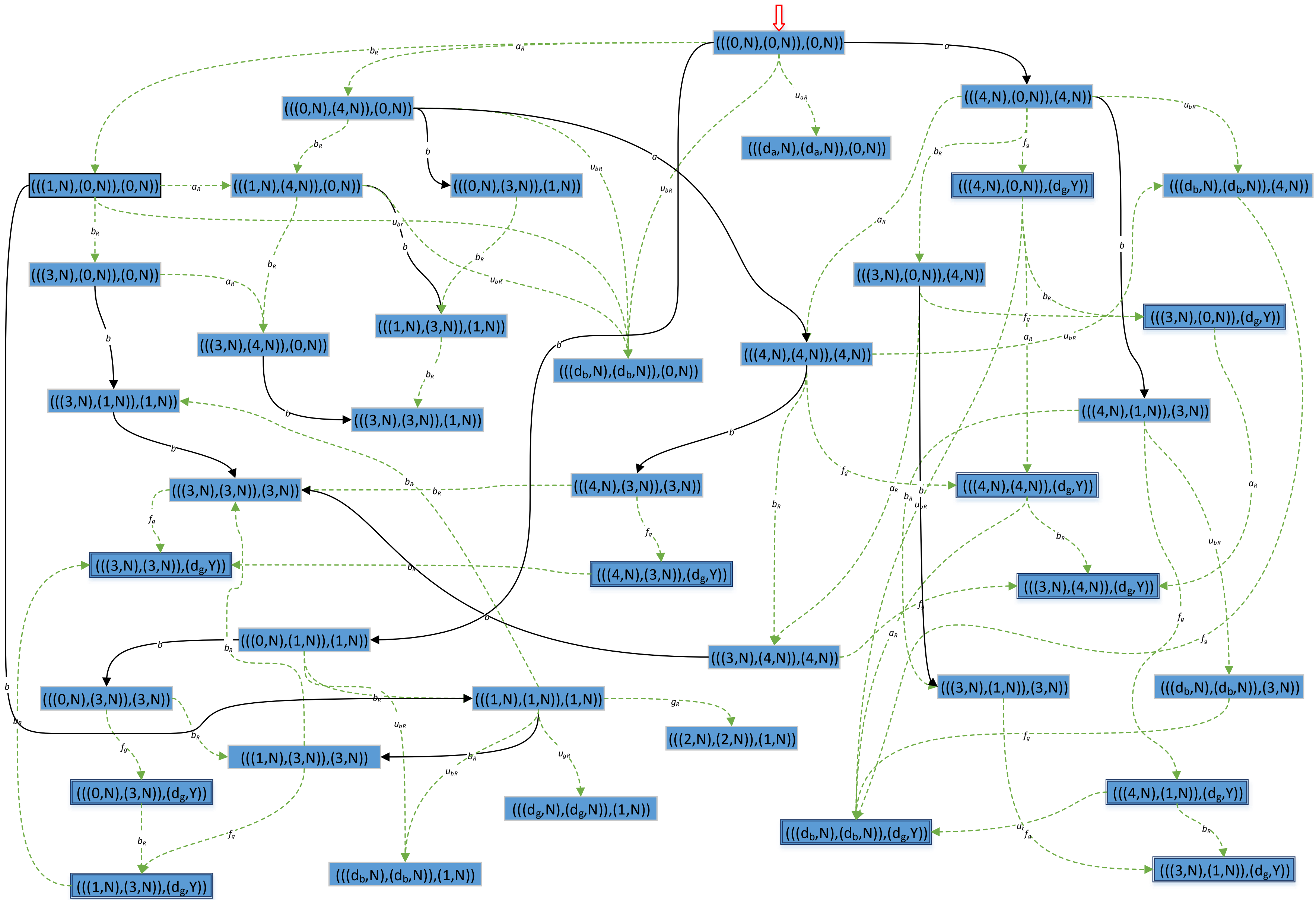


Figura 3.16: Autômato V_{diag} do exemplo 3.7.

3.5.2 Algoritmo Proposto

Nesta seção será apresentada e demonstrada a generalização do algoritmo 3.5 para o caso de controle supervisório descentralizado.

- **Algoritmo para Verificação**

Seja $K \subseteq M = \overline{M}$ e considere o problema de se verificar se K é coobservável em relação a M, $E_{o,i}$ e $E_{c,i}$, com $i \in \{1, 2, ..., n\}$. De acordo com a definição de coobservabilidade tem-se que K será coobservável em relação a M, $E_{o,i}$ e $E_{c,i}$, com $i \in \{1, 2, ..., n\}$, se

$$\forall \sigma \in E_c = \cup_{i=1}^{n} E_{c,i} \text{ e } \forall s \in \overline{K} \text{ tal que } s\sigma \in M \text{ e } s\sigma \notin \overline{K} \Rightarrow$$
$$\exists i \in \{1, 2, ..., n\} \text{ tal que } P_i^{-1}[P_i(s)]\sigma \cap \overline{K} = \emptyset \ \wedge \ \sigma \in E_{c,i}.$$

Ou equivalentemente, se

$$\forall \sigma \in E_c = \cup_{i=1}^{n} E_{c,i} \text{ e } \forall s \in \overline{K} \text{ tal que } s\sigma \in M \text{ e } s\sigma \notin \overline{K} \Rightarrow$$
$$\exists i \in \{1, 2, ..., n\} \text{ no qual } \nexists s' \in \overline{K} \text{ tal que } s'\sigma \in \overline{K} \text{ e } P_i(s) = P_i(s') \ \wedge \ \sigma \in E_{c,i}.$$

Uma vez que esta definição é uma generalização da observabilidade, é possível estender o princípio de funcionamento do algoritmo visto na seção 3.2.1 para a verificação da coobservabilidade. Se forem construídos n autômatos que representem o comportamento “normal”, tais que o i-ésimo autômato tenha os eventos não-observáveis pelo i-ésimo agente renomeados, então a composição paralela dos n autômatos, seguida da composição paralela com o autômato que representa o comportamento de “falha”, resulta em um autômato verificador com as mesmas características do verificador da observabilidade. Todavia, a generalização da coobservabilidade implica dois requisitos adicionais para a verificação da falha de coobservabilidade:

(i) Para cada transição σ ligando um estado não-marcado a um estado marcado, basta que um agente tenha a observabilidade “garantida”. Assim, para cada evento σ basta que um agente não viole as condições de observabilidade.

(ii) As condições de observabilidade para um dado agente i precisam ser estendidas para permitir identificar, no verificador, que o evento σ está ativo no estado correspondente ao i-ésimo agente. Assim, para o caso em que $\sigma \in E_{o,i}$ é necessário, adicionalmente, analisar se σ está ativo, na especificação, no estado correspondente ao agente i. Já para o caso em que $\sigma \notin E_{o,i}$, o requisito de existência do evento σ renomeado para $\sigma_{R,i}$ no estado de origem da transição, já garante que ele esteja ativo na especificação para o estado relativo a i. Por fim,

passa a ser necessário, também, incluir a condição do evento σ ser controlável para o i-ésimo agente.

O algoritmo a seguir implementa este funcionamento generalizado para verificar a propriedade de coobservabilidade de um sistema descentralizado.

Algoritmo 3.8. *Sejam os autômatos* $G = (X_G, E, f_G, \Gamma_G, x_0, X_{m,G})$ *e* $H = (X_H, E, f_H, \Gamma_H, x_0, X_{m,H})$ *tais que* $\mathcal{L}(G) = M = \overline{M}$*,* $\mathcal{L}_m(H) = K$ *e* $H \sqsubseteq G$*. Sejam os conjuntos de eventos controláveis e observáveis representados respectivamente por* $E_{c,i} \subseteq E$ *e* $E_{o,i} \subseteq E$*,* $i \in \{1, 2, ..., n\}$*.*

Passo 1: *Construa os autômatos* G_m *e* H_m*, com todos os estados marcados, da seguinte forma:*

$$G_m := (X_G, E, f_G, \Gamma_G, x_0, X_G);$$
$$H_m := (X_H, E, f_H, \Gamma_H, x_0, X_H).$$

Passo 2: *Defina a função de renomeação* $R_i : E \to E_{R,i}$ *como*[5]*:*

$$R_i(\sigma) = \begin{cases} \sigma & \textit{se } \sigma \in E_{o,i} \\ \sigma_{R,i} & \textit{se } \sigma \notin E_{o,i}. \end{cases}$$

Para $i = 1, ..., n$*, construa o autômato* $H_{R,i} := (X_H, E_{R,i}, f_{R,i}, \Gamma_{R,i}, x_0, X_H)$ *com* $E_{R,i} = R_i(E)$*,* $f_{R,i}(x, R_i(\sigma)) = f_H(x, \sigma)$ *e* $\Gamma_{R,i}(x) = R_i[\Gamma_H(x)]$*,* $\forall x \in X_H$*.*

Passo 3: *Construa o autômato* H_C^D *por meio dos seguintes passos:*

3.1: $H_C := G_m \times H_m^C$*, sendo* H_m^C *o complementar de* H_m*;*

3.2: *Defina a função de renomeação de estado* $\mathcal{D} : X \to X_D$ *como*[6]*:*

$$\mathcal{D}(x) = \begin{cases} x & \textit{se } x \notin X_m \\ D & \textit{se } x \in X_m; \end{cases}$$

[5]Note que a função R_i apenas renomeia os eventos que não pertencem a $E_{o,i}$, permitindo a captura do comportamento assíncrono junto a outro autômato não renomeado. A notação $R_i(E)$ será usada para representar a renomeação de todos os eventos de um conjunto E.

[6]Note que a função $\mathcal{D}$ é utilizada para reunir todos os estados marcados. Seu emprego é válido quando o comportamento após atingir uma marcação não for mais relevante para a análise. A vantagem de sua utilização é diminuir a complexidade das operações de composição que utilizam o autômato como argumento. $\mathcal{D}(X)$ denota a aplicação da função $\mathcal{D}$ sobre cada estado do conjunto X. $\mathcal{D}(f(x, \sigma))$ denota a aplicação da função $\mathcal{D}$ sobre o estado retornado por $f(x, \sigma)$.

Defina o autômato $H_C^D := (X_{H_C}^D, E, f_{H_C}^D, \Gamma_{H_C}^D, x_{0,H_C}, X_{m,H_C}^D)$ *fazendo:*

$$\begin{aligned} X_{H_C}^D &= \mathcal{D}(X_{H_C}); \\ \Gamma_{H_C}^D(x) &= \Gamma_{H_C}[\mathcal{D}(x)]; \\ f_{H_C}^D(x,\sigma) &= \mathcal{D}[f_{H_C}(x,\sigma)]; \\ X_{m,H_C}^D &= \{D\}. \end{aligned}$$

Passo 4: *Construa o autômato* V_{RC} *da seguinte forma:*

4.1: $V_{RC} := (\|_{i=1}^n H_{R,i}) \| H_C^D = (X_{V_{RC}}, E_{V_{RC}}, f_{V_{RC}}, \Gamma_{V_{RC}}, x_{0,V_{RC}}, X_{m,V_{RC}})$, *não sendo necessário obter as transições que partem dos estados marcados de* V_{RC}*, ficando a cargo da implementação essa escolha.*

4.2: *Defina o autômato* $V := (X_V, E_{V_{RC}}, f_V, \Gamma_V, x_{0,V_{RC}}, X_{m,V})$ *fazendo:*

$$\begin{aligned} X_V &= \mathcal{D}(X_{V_{RC}}); \\ \Gamma_V(x) &= \Gamma_{V_{RC}}[\mathcal{D}(x)]; \\ f_V(x,\sigma) &= \mathcal{D}[f_{V_{RC}}(x,\sigma)]; \\ X_{m,V} &= \{D\}. \end{aligned}$$

Passo 5: *Para cada evento* $\sigma \in E_c = \cup_{i=1}^n E_{c,i}$ *tal que* $(f_V(x,\sigma) = D) \wedge (x \neq D)$*, verifique se para todo* $i \in \{1, 2, ..., n\}$ *pelo menos uma das condições abaixo é verdadeira:*

(i) $\sigma \notin E_{c,i}$;

(ii) $\sigma \in E_{c,i} \quad \wedge \quad \sigma \in E_{o,i} \quad \wedge \quad \sigma \in \Gamma_H(x_i)$;

(iii) $\sigma \in E_{c,i} \quad \wedge \quad \sigma \notin E_{o,i} \quad \wedge \quad R_i(\sigma) \in \Gamma_V(x)$;

em que $x_i \in X_H$ *é a i-ésima componente do estado* $x \in X_V$*.*

Passo 6: *Se existir* $\sigma \in E_c$ *tal que pelo menos uma das condições acima for satisfeita* $\forall i \in \{1, 2, ..., n\}$*, então* K *será não coobservável em relação a* M*,* $E_{o,i}$ *e* $E_{c,i}$*,* $i \in \{1, 2, ..., n\}$*. Caso contrário,* K *será coobservável em relação a* M*,* $E_{o,i}$ *e* $E_{c,i}$*,* $i \in \{1, 2, ..., n\}$*.* □

O teorema a seguir demonstra a corretude do algoritmo 3.8 na verificação da coobservabilidade.

Teorema 3.3. *Sejam* K *e* $M = \overline{M}$ $(K \subseteq M)$ *linguagens regulares tais que* $\mathcal{L}_m(H) = K$*,* $\mathcal{L}(G) = M$*, e seja* $V = (X_V, E_V, f_V, \Gamma_V, x_{0,V}, X_{m,V})$ *um autômato verificador construído de acordo com o algoritmo 3.8. Então* K *não será coobservável em relação*

a M, $E_{o,i}$ e $E_{c,i}$, $i \in \{1,2,...,n\}$, se e somente se existir $\sigma \in E_c = \cup_{i=1}^{n} E_{c,i}$ que satisfaça $f_V(x,\sigma) = D$ e $x \neq D$, tal que para todo $i \in \{1,2,...,n\}$:

$$(\sigma \notin E_{c,i}) \;\vee\; [(\sigma \in E_{c,i}) \;\wedge\; (\sigma \in E_{o,i}) \;\wedge\; (\sigma \in \Gamma_H(x_i))] \;\vee$$
$$[(\sigma \in E_{c,i}) \;\wedge\; (\sigma \notin E_{o,i}) \;\wedge\; (R_i(\sigma) \in \Gamma_V(x))],$$

sendo $x_i \in X_H$ a i-ésima componente do estado $x \in X_V$.

Demonstração.

($\Rightarrow$) Suponha que K não seja coobservável em relação a M, $E_{o,i}$ e $E_{c,i}$, $i \in \{1,2,...,n\}$, e considere a operação de projeção P_i definida por

$$P_i : E^* \rightarrow E_{o,i}^*.$$

Então, pela definição de coobservabilidade, existem sequências $s_1 \neq s_2$, $s_1, s_2 \in \overline{K}$ tais que para algum $\sigma \in E_c = \cup_{i=1}^{n} E_{c,i}$:

$$(s_1\sigma \in \overline{K})\ (s_2\sigma \in M \backslash \overline{K}) \Rightarrow$$
$$\forall i \in \{1,2,...,n\}, (\sigma \notin E_{c,i}) \;\vee\; (P_i(s_1) = P_i(s_2)).$$

Pela construção de $H_{R,i}$, como $s_1\sigma \in \overline{K}$ então $s_1\sigma \in \mathcal{L}_m(H_m)$. Seja $s_{1_{R,i}} = R_i(s_1)$. Então $s_{1_{R,i}} \in \mathcal{L}_m(H_{R,i}) = \mathcal{L}(H_{R,i})$.

Considere, agora, a operação de projeção P_{R,i_o} definida por

$$P_{R,i_o} : E_{R,i}^* \rightarrow E_{o,i}^*.$$

Assim $P_i(s_1) = P_{R,i_o}(s_{1_{R,i}})$, uma vez que a função de renomeação não altera a observabilidade de um evento.

Pela construção de H_C^D, $\mathcal{L}(H_C) \subseteq \mathcal{L}(H_C^D)$ e $\mathcal{L}_m(H_C) \subseteq \mathcal{L}_m(H_C^D)$. Assim, como $s_2 \in \overline{K}$ e $s_2\sigma \in M \backslash \overline{K}$, então $s_2 \in \mathcal{L}(H_C^D)$ porém $s_2 \notin \mathcal{L}_m(H_C^D)$ e $s_2\sigma \in \mathcal{L}_m(H_C^D)$.

Seja $E_R = \cup_{i=1}^{n} E_{R,i}$ e $E' = E_R \cup E$. Defina as operações de projeção P' e $P'_{R,i}$ da seguinte forma:

$$P' : E'^* \rightarrow E^*;$$
$$P'_{R,i} : E'^* \rightarrow E_{R,i}^*.$$

Considere, agora, o autômato V_{RC}. É fácil verificar que $\mathcal{L}(V_{RC}) \subseteq \mathcal{L}(V)$ e

$\mathcal{L}_m(V_{RC}) \subseteq \mathcal{L}_m(V)$. Como $V_{RC} = (\|_{i=1}^n H_{R,i})\|H_C^D$ então:

$$\mathcal{L}(V_{RC}) = (\bigcap_{i=1}^{n} P_{R,i}^{'-1}[\mathcal{L}(H_{R,i})]) \cap P^{'-1}[\mathcal{L}(H_C^D)] \subseteq \mathcal{L}(V); \tag{3.5}$$

$$\mathcal{L}_m(V_{RC}) = (\bigcap_{i=1}^{n} P_{R,i}^{'-1}[\mathcal{L}_m(H_{R,i})]) \cap P^{'-1}[\mathcal{L}_m(H_C^D)] \subseteq \mathcal{L}_m(V). \tag{3.6}$$

Como visto acima, $s_{1_{R,i}} \in \mathcal{L}(H_{R,i})$ e $s_2 \in \mathcal{L}(H_C^D)$. Uma vez que $s_{1_{R,i}}$ difere de s_2 e de $s_{1_{R,j}}$ $(i \neq j)$ apenas por eventos renomeados, que pertencem a E', é possível afirmar que $\exists s' \in (\bigcap_{i=1}^n P_{R,i}^{'-1}(s_{1_{R,i}})) \cap P^{'-1}(s_2)$ tal que $s' \in \mathcal{L}(V)$. Porém, $s' \notin \mathcal{L}_m(V)$, uma vez que $s_2 \notin \mathcal{L}_m(H_C^D)$.

Para cada $i \in \{1, 2, ..., n\}$ existem duas opções para σ: $\sigma \in E_{o,i}$ ou $\sigma \notin E_{o,i}$.

Suponha, inicialmente, que $\sigma \in E_{o,i}$. Como $s_1\sigma \in \overline{K}$ e $s_{1_{R,i}} \in \mathcal{L}_m(H_{R,i})$, então $s_{1_{R,i}}\sigma \in \mathcal{L}_m(H_{R,i})$, e, portanto, $\exists x_{R,i} \in X_{H_{R,i}}, x_{R,i} \in X_{m,H_{R,i}} : \sigma \in \Gamma_{H_{R,i}}(x_{R,i})$. Além disso, como $s_2 \notin \mathcal{L}_m(H_C^D)$ mas $s_2\sigma \in M\backslash\overline{K}$, então $s_2\sigma \in \mathcal{L}_m(H_C^D)$, e, portanto, $\exists x_C^D \in X_{H_C^D}, x_C^D \notin X_{m,H_C^D} : \sigma \in \Gamma_{H_C^D}(x_C^D)$.

Dessa forma, $\exists x = (x_{V,1}, x_{V,2}, ..., x_{V,n}, x_{V,n+1}) = f_V(x_{0,V}, s')$, tal que

(i) $x_{V,i} = x_{R,i}$ e $x_{V,n+1} = x_C^D$;

(ii) $f_V(x, \sigma) = D$;

(iii) $x \neq D$.

Considere, agora, que $\sigma \notin E_{o,i}$. Como $s_1\sigma \in \overline{K}$ e $s_{1_{R,i}} \in \mathcal{L}_m(H_{R,i})$, então $s_{1_{R,i}}\sigma_{R,i} \in \mathcal{L}_m(H_{R,i})$, e, portanto, $\exists x_{R,i} \in X_{H_{R,i}}, x_{R,i} \in X_{m,H_{R,i}} : \sigma_{R,i} \in \Gamma_{H_R}(x_{R,i})$. Além disso, como $s_2 \notin \mathcal{L}_m(H_C^D)$ mas $s_2\sigma \in M\backslash\overline{K}$, então $s_2\sigma \in \mathcal{L}_m(H_C^D)$, e, portanto, $\exists x_C^D \in X_{H_C^D}, x_C^D \notin X_{m,H_C^D} : \sigma \in \Gamma_{H_C^D}(x_C^D)$. Por fim, como $\sigma_{R,i} \in \Gamma_{H_{R,i}}(x_{R,i})\backslash E$, então $\sigma_{R,i} \in \Gamma_V(x)$.

Dessa forma, $\exists x = (x_{V,1}, x_{V,2}, ..., x_{V,n}, x_{V,n+1}) = f_V(x_{0,V}, s')$, tal que

(i) $x_{V,i} = x_{R,i}$ e $x_{V,n+1} = x_C^D$;

(ii) $R_i(\sigma) \in \Gamma_V(x)$;

(iii) $f_V(x, \sigma) = D$;

(iv) $x \neq D$.

Note que no desenvolvimento acima não foi levado em conta se $\sigma \in E_c$. Assim, se $\sigma \notin E_c$ então σ não poderá ser desabilitado, levando, então, à não-observabilidade pelo agente i. Por outro lado, quando $\sigma \in E_c$, a confusão somente ocorrerá nas situações descritas acima, o que prova a suficiência da condição.

($\Leftarrow$) Suponha que para um autômato V construído de acordo com o algoritmo 3.8, existe $\sigma \in \cup_{i=1}^{n} E_{c,i}$ e $x \in X_V$, $x \neq D$, que satisfaz $f_V(x, \sigma) = D$ e seja tal que para todo $i \in \{1, 2, ..., n\}$:

$$(\sigma \notin E_{c,i}) \ \vee \ [(\sigma \in E_{c,i}) \ \wedge \ (\sigma \in E_{o,i}) \ \wedge \ (\sigma \in \Gamma_H(x_i))] \ \vee$$
$$[(\sigma \in E_{c,i}) \ \wedge \ (\sigma \notin E_{o,i}) \ \wedge \ (R_i(\sigma) \in \Gamma_V(x))],$$

sendo $x_i \in X_H$ é a i-ésima componente do estado $x \in X_V$.

Logo, $\exists s' \in \mathcal{L}(V), s' \notin \mathcal{L}_m(V)$ tal que $s'\sigma \in \mathcal{L}_m(V)$. Pela construção de V, pode-se afirmar que $s' \in \mathcal{L}(V_{RC})$, $s' \notin \mathcal{L}_m(V_{RC})$ e $s'\sigma \in \mathcal{L}_m(V_{RC})$.

Considere, agora, o autômato V_{RC}. De acordo com as equações (3.5) e (3.6), tem-se que

$$\mathcal{L}(V_{RC}) = (\bigcap_{i=1}^{n} P_{R,i}^{'-1}[\mathcal{L}(H_{R,i})]) \cap P^{'-1}[\mathcal{L}(H_C^D)],$$
$$\mathcal{L}_m(V_{RC}) = (\bigcap_{i=1}^{n} P_{R,i}^{'-1}[\mathcal{L}_m(H_{R,i})]) \cap P^{'-1}[\mathcal{L}_m(H_C^D)].$$

Portanto, $\forall i \in \{1, 2, ..., n\}$, $\exists s_{R,i} \in \mathcal{L}(H_{R,i})$ ($s_{R,i} \in \mathcal{L}_m(H_{R,i})$ uma vez que todos os estados de $H_{R,i}$ são marcados), e $s_2 \in \mathcal{L}(H_C^D)$ ($s_2 \notin \mathcal{L}_m(H_C^D)$ uma vez que $s' \notin \mathcal{L}_m(V_{RC})$) tais que

$$s' \in (\bigcap_{i=1}^{n} P_{R,i}^{'-1}(s_{R,i})) \cap P^{'-1}(s_2). \tag{3.7}$$

Assim sendo, existe $s'_{R,i} \in P_{R,i}^{'-1}(s_{R,i})$ e $s'_2 \in P^{'-1}(s_2)$ tal que $s' = s'_{R,i} = s'_2$. Note que:

(i) $E_{R,i} \cap E_{o,i} = E_{o,i}$, ou seja, se $e \in E_{o,i}$, então $e \in E_{R,i}$. Logo, como $s_{R,i}$ está definida sobre $E_{R,i}^*$, então $P_{R,i}^{'-1}(s_{R,i})$ não acrescenta a $s_{R,i}$ nenhum evento $e \in E_{o,i}$.

(ii) $E \cap E_{o,i} = E_{o,i}$, ou seja, se $e \in E_{o,i}$, então $e \in E$. Logo, como s_2 está definida sobre E^*, então $P^{'-1}(s_2)$ não acrescenta a s_2 nenhum evento $e \in E_{o,i}$.

Portanto:

$$P_i[P_{R,i}^{'-1}(s_{R,i})] = P_i(s_{R,i}),$$
$$P_i[P^{'-1}(s_2)] = P_i(s_2),$$

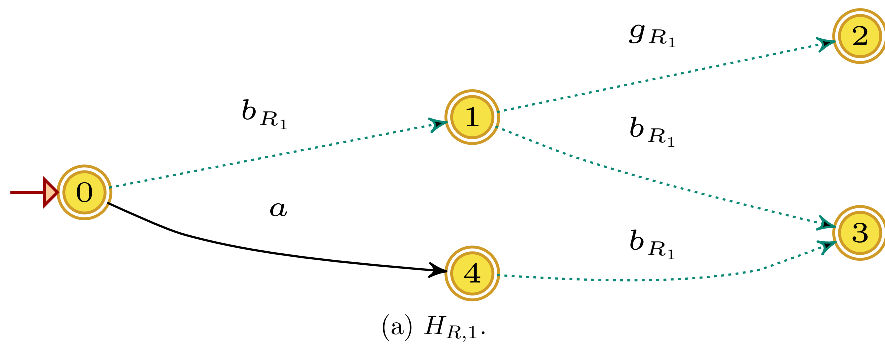
e, dessa forma:

$$P_i(s') = P_i(s_{R,i}) = P_i(s_2). \tag{3.8}$$

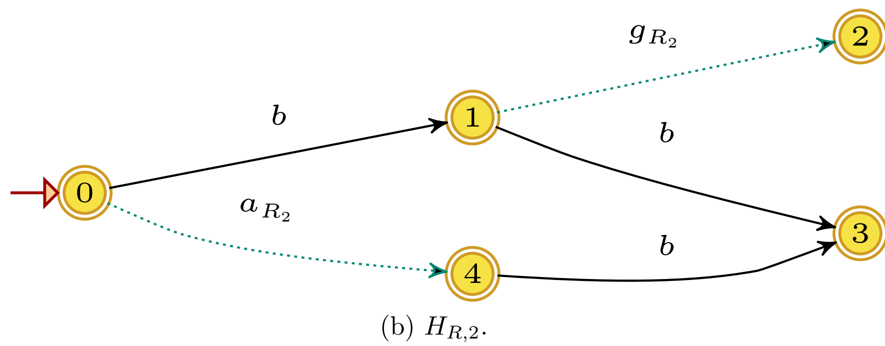
Consequentemente, pode-se afirmar que $\exists s_i = R_i^{-1}(s_{R,i}) \in \mathcal{L}(H_m)$, $i = 1, 2, ..., n$, e portanto $\{s_1\sigma, s_2\sigma, ..., s_n\sigma\} \subseteq \overline{K}$. Ao mesmo tempo, $\exists s_2 \in \overline{K}$ tal que $s_2\sigma \in M \backslash \overline{K}$. Desse modo, quando $\sigma \in E_{c,i}$ nenhum agente será capaz de decidir corretamente sobre a desativação de $\sigma \in E_c$. Além disso, se $\sigma \notin E_{c,i}$ então σ não poderá ser desabilitado pelo agente i. Portanto, K não é coobservável em relação a M, $E_{o,i}$ e $E_{c,i}$, $i \in \{1, 2, ..., n\}$. □

Exemplo 3.8. *Considere novamente o sistema coobservável analisado no exemplo 3.7. A execução do algoritmo 3.8 é realizada do seguinte modo. Como os autômatos G e H possuem todos os estados marcado, o primeiro passo não modifica os autômatos de entrada para construir G_m e H_m. O passo seguinte realiza a construção dos autômatos $H_{R,1}$ e $H_{R,2}$, a partir da correspondente renomeação dos eventos não-observáveis por cada agente. Com isso, os eventos b e g são renomeados, respectivamente, para b_{R_1} e g_{R_1} no autômato $H_{R,1}$ (figura 3.17a). Do mesmo modo, os eventos a e g são renomeados respectivamente para a_{R_2} e g_{R_2} no autômato $H_{R,2}$ (figura 3.17b). O terceiro passo trata, então, da construção do autômato H_C^D que marca a linguagem não especificada $M \backslash \overline{K}$. Para tanto, inicialmente o autômato complementar H_C é construído, ilustrado na figura 3.17c. Nesse autômato é possível observar que as ocorrências de g nos estados* 3 *e* 4 *levam o sistema a sair da linguagem especificada. A operação de construção de H_C^D efetua a reunião dos estados marcados em um único estado D (figura 3.17d). O último passo do algoritmo cria o verificador, o que é realizado inicialmente pela composição paralela $H_{R,1} \| H_{R,2} \| H_C^D$ e finalmente pela reunião dos estados marcados em um único estado D, resultando no autômato V, ilustrado na figura 3.18.*

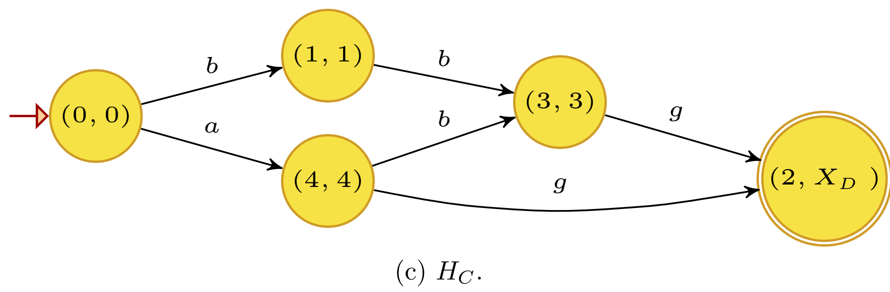
A busca por $\sigma \in E_c = \cup_{i=1}^{n} E_{c,i}$ que leve o verificador de um estado não marcado para o estado marcado D atendendo as condições dadas pelo passo 5 *do algoritmo 3.8 falha. Logo, é possível concluir que a linguagem K é coobservável com relação a M, $E_{c,1}$, $E_{c,2}$, $E_{o,1}$ e $E_{o,2}$.*



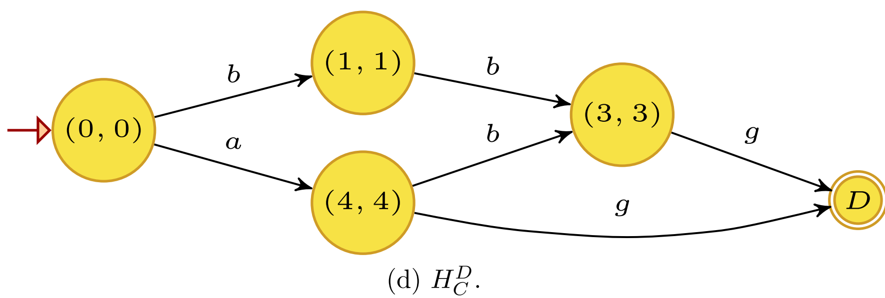
(a) $H_{R,1}$.



(b) $H_{R,2}$.



(c) H_C.



(d) H_C^D.

Figura 3.17: Autômatos $H_{R,1}$, $H_{R,2}$, H_C e H_C^D do exemplo 3.8.

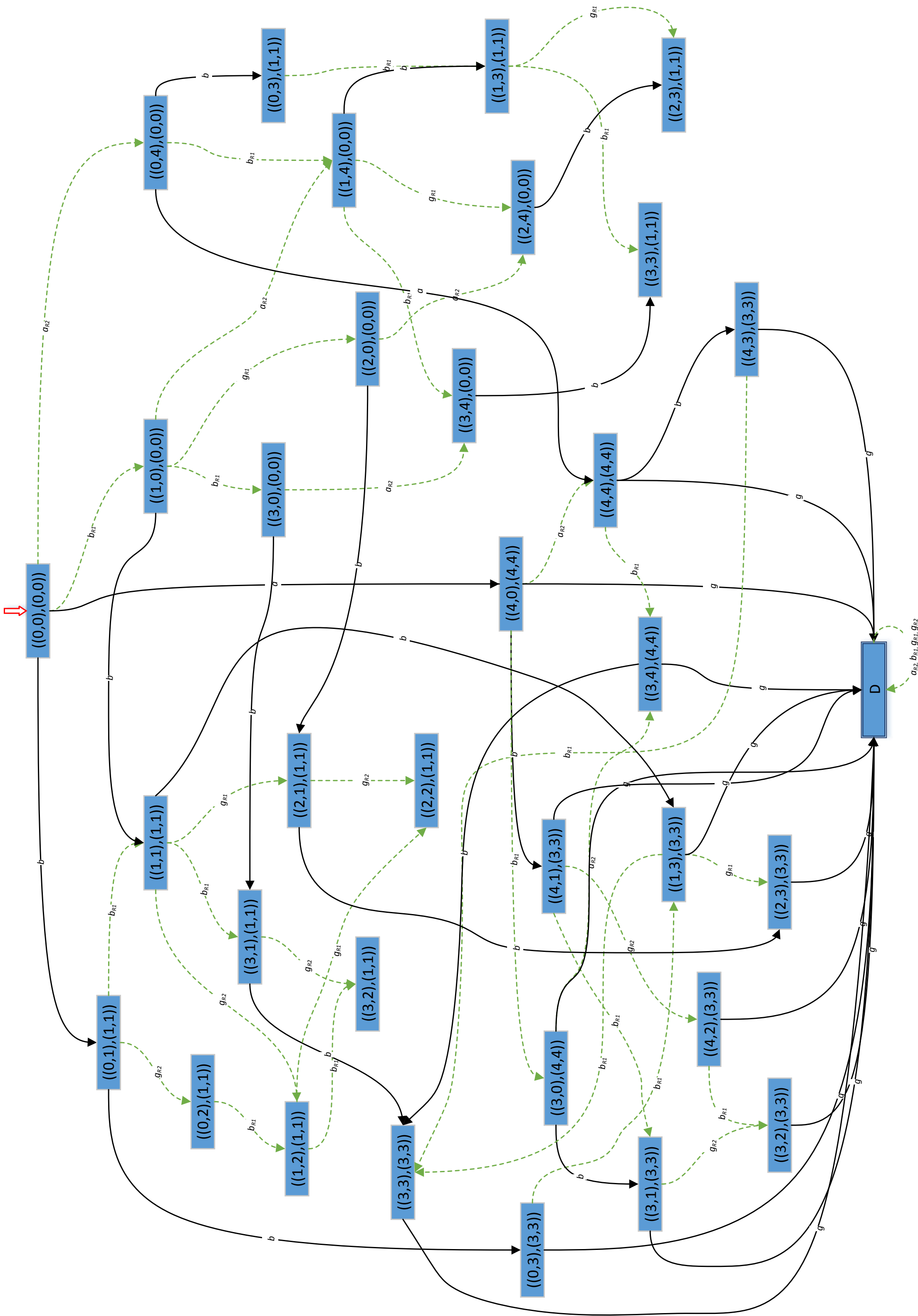


Figura 3.18: Autômato V do exemplo 3.8.

- **Complexidade Computacional**

A análise da complexidade do algoritmo 3.8 é determinada pela análise dos passos que levam à obtenção do autômato V, uma vez que a verificação das condições é linear e se baseia apenas nas transições que levam ao estado D.

A tabela 3.1 contém o número máximo de estados e transições dos autômatos que necessitam se gerados para a obtenção de V. Como entrada para esse cálculo são considerados, para G e H, os conjuntos de estados no pior caso, ambos dados por X, e os conjuntos de eventos, ambos dados por E. É imediato chegar aos autômatos G_m e H_m, que se diferenciam de G e H por conterem todos os estados marcados, possuindo assim a mesma cardinalidade de estados e transições de G e H, respectivamente. O espaço de estados de cada autômato resultante da renomeação dos eventos não-observáveis, $H_{R,i}$, terá a mesma cardinalidade de H. Por sua vez, o autômato complementar de H_m, H_m^C, terá um estado a mais que H_m. Já o autômato H_C, que marca a linguagem $M \backslash \overline{K}$ possuirá, no máximo, o dobro de estados de G, uma vez que ele é resultado do produto de dois autômatos tais que: (i) a linguagem gerada pelo segundo está contida na linguagem do primeiro; (ii) o conjunto de estados do segundo é um subconjunto do conjunto de estados do primeiro, sendo que os estados pertencentes ao conjunto diferença estão representados no segundo por um único estado X_D. Na sequência, a construção de H_C^D é apenas uma unificação dos estados marcados, o que remove todas as duplicidades de estados, voltando ao pior caso de cardinalidade $|X| + 1$ para o espaço de estados. As operações de composição paralela, que geram o autômato V_{RC}, levam o espaço de estados ao tamanho $|X|^{n+1} + |X|^n$ e o de transições à $(|X|^{n+1} + |X|^n) \times (|E| + n|E_{uo}|)$, uma vez que o conjunto de eventos de V_{RC} é formado por E mais todas as versões renomeadas dos eventos não-observáveis. Por fim, a unificação dos estados marcados em D gera um verificador V com espaço de estados de tamanho $|X|^{n+1} + 1$ e número de transições igual a $(|X|^{n+1}+1)\times(|E|+n|E_{uo}|)$, e, portanto, o algoritmo 3.8 possui complexidade $\mathcal{O}(|X|^{n+1})$ no número de estados, e $\mathcal{O}(n|X|^{n+1}|E|)$ no número de transições. Assim, como os algoritmos existentes mencionados em 3.5.1, a complexidade do algoritmo aqui proposto também é polinomial.

Uma comparação direta de desempenho entre o algoritmo 3.8 e o mais eficiente existente proposto por WANG *et al.* [12], mostra que ambos possuem a mesma complexidade no pior caso. Todavia, cabe ressaltar que os mesmos comentários feitos na seção 3.2.2 se aplicam: (i) a necessidade de executar o algoritmo de conversão a priori pode ser interpretada como um trabalho excedente; (ii) para os exemplos testados, o algoritmo aqui proposto resulta em verificadores de igual ou menor cardinalidade.

Tabela 3.2: Complexidade Computacional do Algoritmo 3.8.

Autômato	Número de Estados	Número de Transições
G	$\|X\|$	$\|X\| \times \|E\|$
H	$\|X\|$	$\|X\| \times \|E\|$
G_m	$\|X\|$	$\|X\| \times \|E\|$
H_m	$\|X\|$	$\|X\| \times \|E\|$
$H_{R,i}$	$\|X\|$	$\|X\| \times \|E\|$
H_m^C	$\|X\| + 1$	$(\|X\| + 1) \times \|E\|$
H_C	$2\|X\|$	$(2\|X\|) \times \|E\|$
H_C^D	$\|X\| + 1$	$(\|X\| + 1) \times \|E\|$
V_{RC}	$\|X\|^{n+1} + \|X\|^n$	$(\|X\|^{n+1} + \|X\|^n) \times (\|E\| + n\|E_{uo}\|)$
V	$\|X\|^{n+1} + 1$	$(\|X\|^{n+1} + 1) \times (\|E\| + n\|E_{uo}\|)$
Complexidade		$\mathcal{O}(n\|X\|^{n+1}\|E\|)$

3.5.3 Exemplos ilustrativos

Visando ilustrar a utilização e a eficácia do verificador de coobservabilidade aqui proposto, vamos apresentar nesta seção alguns exemplos.

- **Exemplo 6**

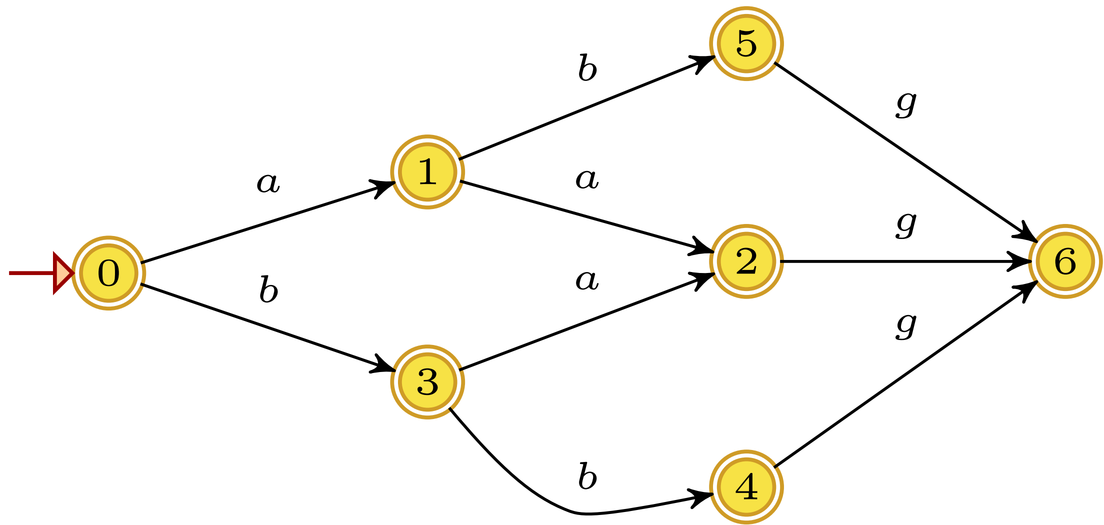
Seja o SED que se deseja controlar definido pela linguagem regular $M = \overline{\{aag, abg, bag, bbg\}}$ e conjunto de eventos $E = \{a, b, g\}$, o qual se encontra modelado pelo autômato G ilustrado na figura 3.19a ($\mathcal{L}(G) = \mathcal{L}_m(G) = M$). Seja a linguagem regular, e fechada em prefixo, $K = \overline{\{aa, bb, ba, abg\}}$ a linguagem especificada para o sistema, modelada pelo autômato H ilustrado na figura 3.19b.

Considere, então, que para manter o sistema dentro da especificação desejada uma arquitetura de controle supervisório descentralizada é projetada, sendo constituída por dois agentes. Os conjuntos de eventos controláveis e observáveis por cada agente são definidos por: $E_{c,1} = \{a, g\}$, $E_{c,2} = \{b, g\}$, $E_{o,1} = \{a\}$ e $E_{o,2} = \{b\}$.

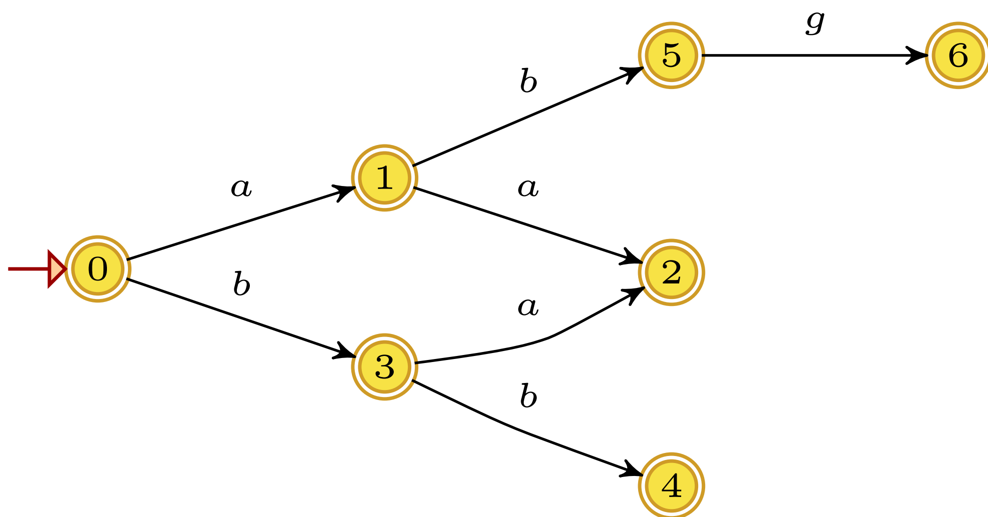
Utilizando o algoritmo 3.8, o primeiro passo é a construção dos autômatos G_m e H_m. Como ambas as linguagens são prefixo-fechadas, o autômato G_m é igual a G (figura 3.19a), e o autômato H_m é igual a H (figura 3.19b). O passo seguinte realiza a construção dos autômatos $H_{R,1}$ e $H_{R,2}$, por meio da respectiva renomeação dos eventos não-observáveis por cada agente. Com isso, os eventos b e g são renomeados, respectivamente, para b_{R_1} e g_{R_1} no autômato $H_{R,1}$ (figura 3.19c). Do mesmo modo, os eventos a e g são renomeados respectivamente para a_{R_2} e g_{R_2} no autômato $H_{R,2}$ (figura 3.20a). O terceiro passo trata, então, da construção do autômato H_C^D que marca a linguagem não especificada $M \backslash \overline{K}$. Para tanto, inicialmente o autômato complementar H_C é construído, ilustrado na figura 3.20b. Nesse autômato é possível observar que as ocorrências de g nos estados 2 e 4 levam o sistema a sair da linguagem especificada. A operação de construção de H_C^D efetua a reunião dos estados marcados

em um único estado D (figura 3.20c). O último passo do algoritmo cria o verificador, o que é realizado inicialmente pela composição paralela $H_{R,1} \| H_{R,2} \| H_C^D$ e finalmente pela reunião dos estados marcados em um único estado D, resultando no autômato V, ilustrado na figura 3.21.

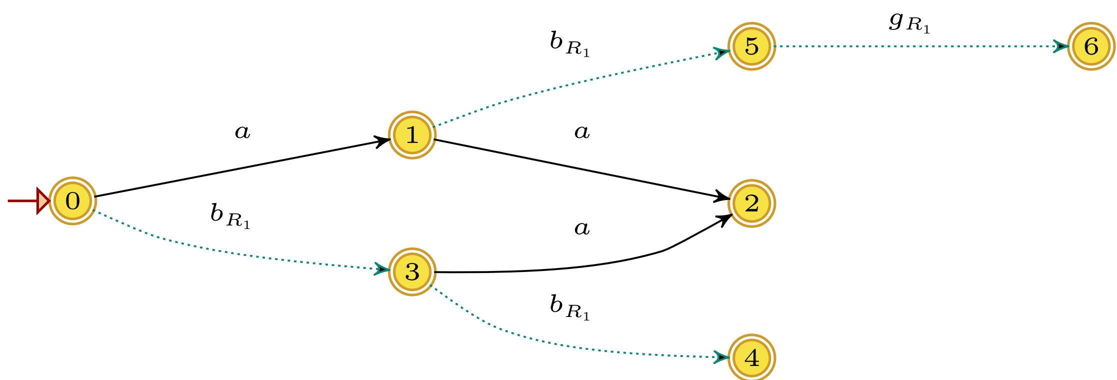
É possível identificar que a ocorrência do evento g no estado $((5,5),(2,2))$ leva à falha de coobservabilidade, uma vez que para os dois agentes, o evento g é controlável, encontra-se ativo no estado 5 do autômato H, é não-observável, e os eventos g_{R_1} e g_{R_2} estão ativos no estado $((5,5),(2,2))$ de V. Logo, conclui-se que K não é coobservável com relação a M, $E_{c,1}$, $E_{c,2}$, $E_{o,1}$ e $E_{o,2}$. Note que esta falha de coobservabilidade é gerada pela confusão de ambos os agentes em relação as sequências $s_1 = ab$ e $s_2 = ba$. Para o agente 1, $P_1(ab) = P_1(ba) = a$, porém a sequência ab requer que o evento controlável g seja habilitado, enquanto a sequência ba requer que o evento controlável g seja desabilitado. Pela característica permissiva da arquitetura em estudo, g será habilitado pelo agente 1. De maneira semelhante, para o agente 2, $P_2(ab) = P_2(ba) = b$, porém a sequência ab requer que o evento controlável g seja habilitado, enquanto a sequência ba requer que o evento controlável g seja desabilitado. Novamente, pela característica permissiva da arquitetura em estudo, g será habilitado pelo agente 2. Assim, como nenhum agente é capaz de desabilitar g após a ocorrência de ba, K não é coobservável com relação a M, $E_{c,1}$, $E_{c,2}$, $E_{o,1}$ e $E_{o,2}$.



(a) $G = G_m$.

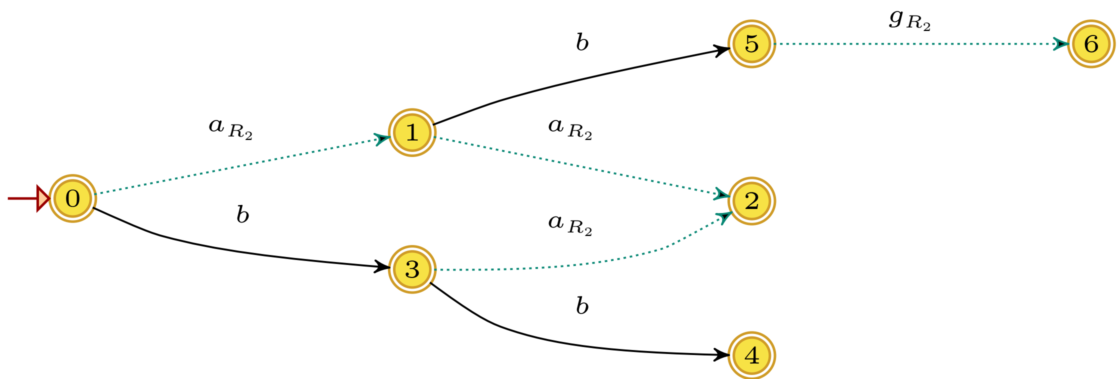


(b) $H = H_m$.

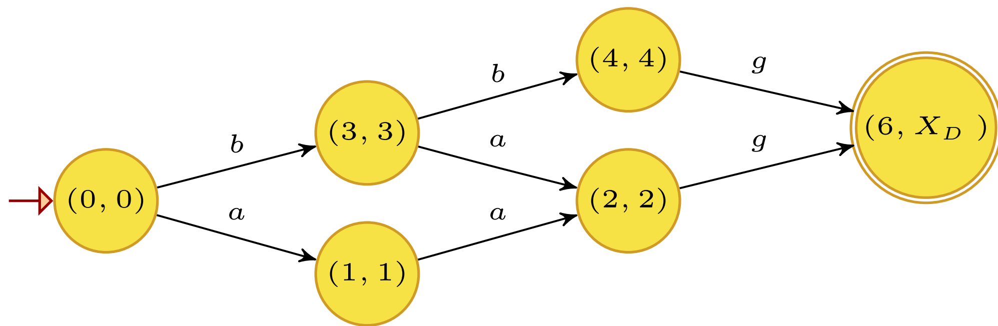


(c) $H_{R,1}$.

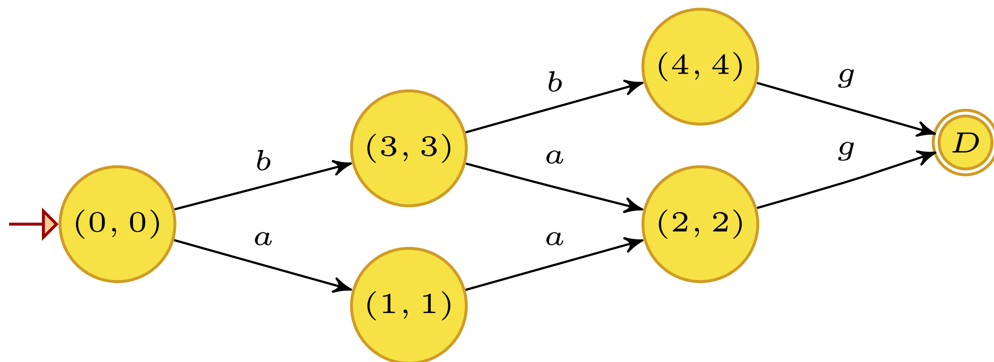
Figura 3.19: Autômatos G, H_m e $H_{R,1}$do Exemplo 6.



(a) $H_{R,2}$.



(b) H_C.



(c) H_C^D.

Figura 3.20: Autômatos $H_{R,2}$, H_C e H_C^D do Exemplo 6.

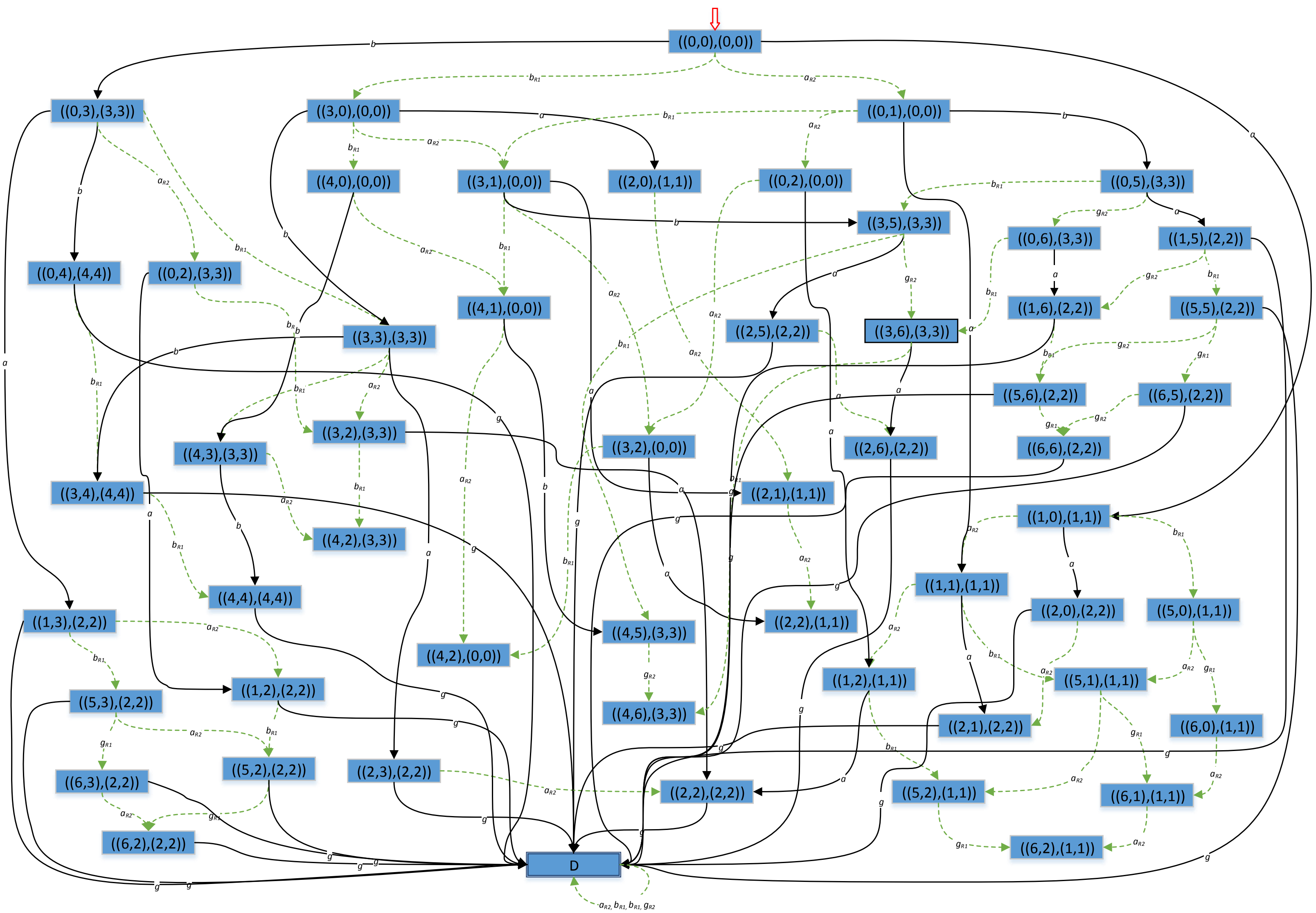


Figura 3.21: Autômato V do Exemplo 6.

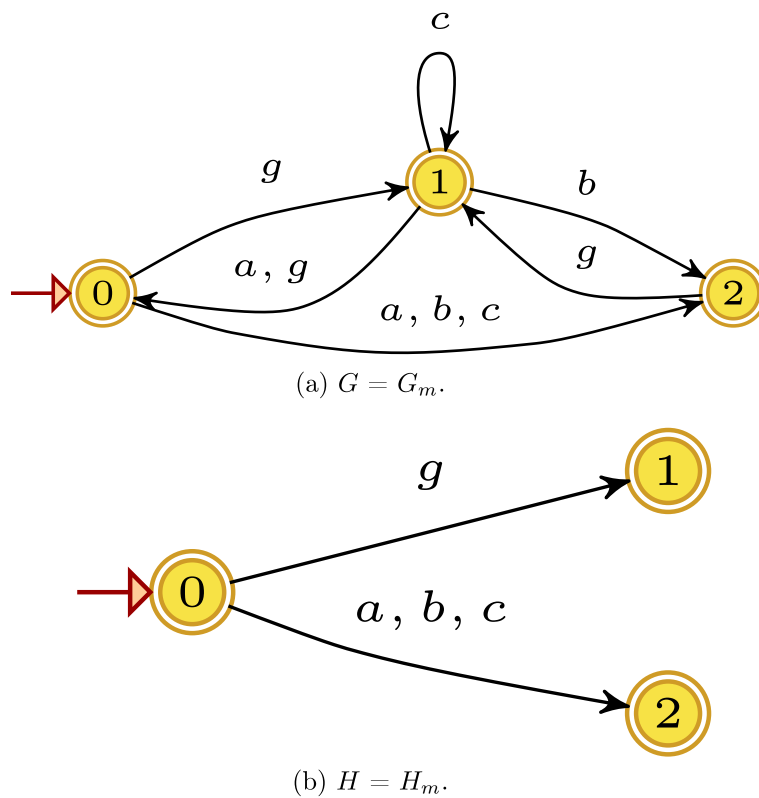
- **Exemplo 7**

Considere o autômato G da figura 3.22a, e suponha que $M = \overline{M} = \mathcal{L}(G)$, no qual $E = \{a, b, c, g\}$. Seja $K = \overline{K}$ a linguagem especificada marcada pelo autômato H ilustrado na figura 3.22b. Considere, então, que para manter o sistema dentro da especificação desejada uma arquitetura de controle supervisório descentralizada é projetada, sendo constituída por três agentes. Os conjuntos de eventos controláveis e observáveis por cada agente são definidos por: $E_{c,1} = \{g\}$, $E_{c,2} = \{g\}$, $E_{c,3} = \{g\}$, $E_{o,1} = \{a\}$, $E_{o,2} = \{b\}$ e $E_{o,3} = \{c\}$.

Utilizando o algoritmo 3.8, o primeiro passo consiste na construção dos autômatos $G_m = G$ e $H_m = H$, já que ambos já possuem todos os estados marcados. O passo seguinte realiza a construção dos autômatos $H_{R,1}$, $H_{R,2}$ e $H_{R,3}$ por meio da respectiva renomeação dos eventos não-observáveis por cada agente. Com isso, os eventos b, c e g são renomeados, respectivamente, para b_{R_1}, c_{R_1} e g_{R_1} no autômato $H_{R,1}$ (figura 3.23a). Do mesmo modo, os eventos a, c e g são renomeados respectivamente para a_{R_2}, c_{R_2} e g_{R_2} no autômato $H_{R,2}$ (figura 3.23b) e, os eventos a, b e g são renomeados respectivamente para a_{R_3}, b_{R_3} e g_{R_3} no autômato $H_{R,3}$ (figura 3.23c). O terceiro passo trata da construção do autômato H_C^D que marca a linguagem não especificada $M \backslash \overline{K}$. Para tanto, inicialmente, o autômato complementar H_C, ilustrado na figura 3.24a, é construído. Nesse autômato é possível observar que as ocorrências de g nos estados 1 e 2, assim como as ocorrências de a, b e c no estado 1, levam o sistema a sair da linguagem especificada. A operação de construção de H_C^D efetua a reunião dos estados marcados em um único estado D (figura 3.20c), colocando em evidência as transições mencionadas acima. O último passo do algoritmo cria o verificador, o que é realizado inicialmente pela composição paralela $H_{R,1} \| H_{R,2} \| H_{R,3} \| H_C^D$ e finalmente pela reunião dos estados marcados em um único estado D, resultando no autômato V. Pela limitação de espaço, esses dois autômatos não se encontram ilustrados neste trabalho.

É possível identificar, em V, que a ocorrência do evento g no estado $(((0,0),0),(1,1))$ leva à falha de coobservabilidade, uma vez que para os três agentes, o evento g é controlável, encontra-se ativo no estado 0 do autômato H, é não-observável, e os eventos g_{R_1}, g_{R_2} e g_{R_3} estão ativos no estado $(((0,0),0),(1,1))$ de V. Logo, conclui-se que K não é coobservável com relação a M, $E_{c,1}$, $E_{c,2}$, $E_{o,1}$ e $E_{o,2}$. Note que a não coobservabilidade é gerada pela confusão dos três agentes em relação as sequências $s_1 = \varepsilon$ e $s_2 = g$. Para os três agentes $P_i(\varepsilon) = P_i(g) = \varepsilon$, porém, a sequência ε requer que o evento controlável g seja habilitado, enquanto a sequência g requer que o evento controlável g seja desabilitado. Pela característica permissiva da arquitetura em estudo, g será habilitado pelos três agentes após a ocorrência de $s_2 = g$. Assim, como nenhum agente é capaz de desabilitar g após a

ocorrência de g, K não é coobservável com relação a M, $E_{c,1}$, $E_{c,2}$, $E_{c,3}$, $E_{o,1}$, $E_{o,2}$ e $E_{o,3}$.



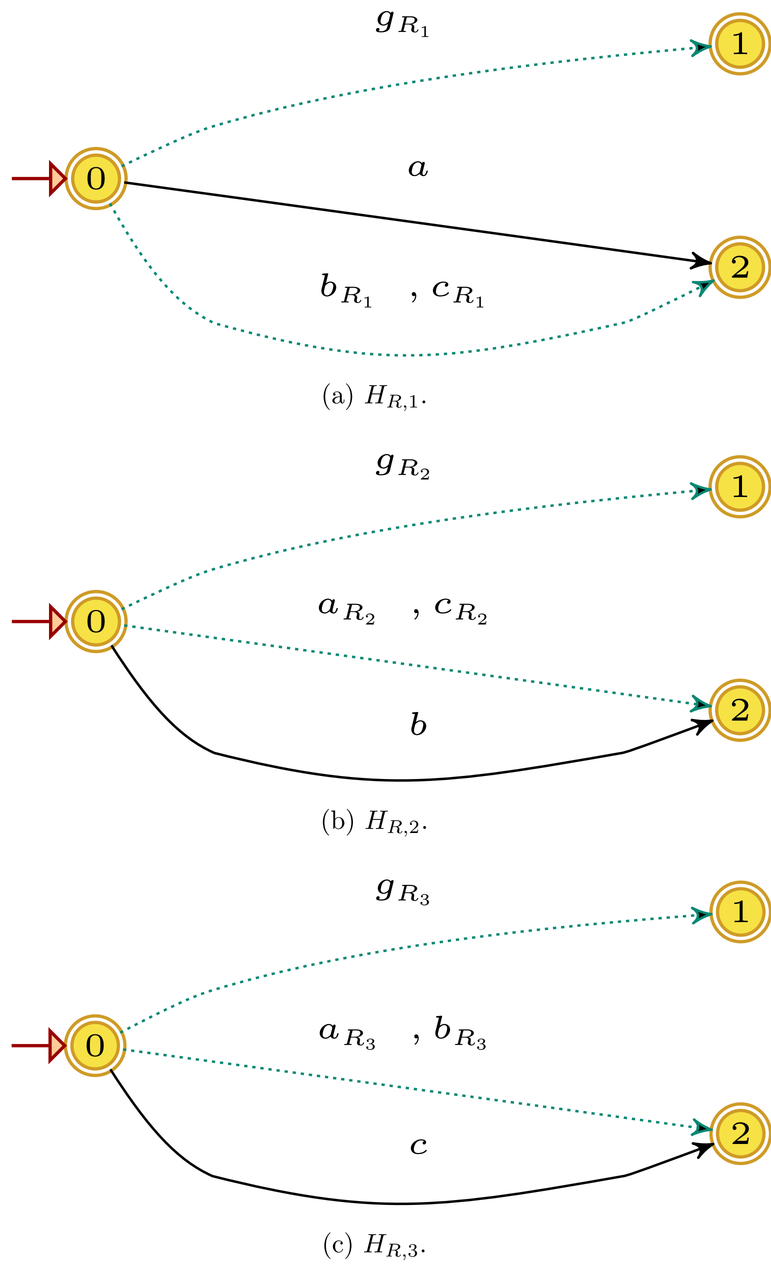
(a) $G = G_m$.

(b) $H = H_m$.

Figura 3.22: Autômatos G e H do Exemplo 7.

3.6 Considerações Finais

Este capítulo apresentou as principais contribuições deste trabalho, quais sejam, foram propostos novos algoritmos mais eficientes do que os algoritmos existentes na literatura para a verificação das propriedades de observabilidade e coobservabilidade. Um mesmo autômato verificador usado para a verificação da observabilidade permite, também, verificar a propriedade de normalidade. Não é de conhecimento do autor que outro algoritmo de complexidade polinomial para a verificação da normalidade tenha sido proposto na literatura.

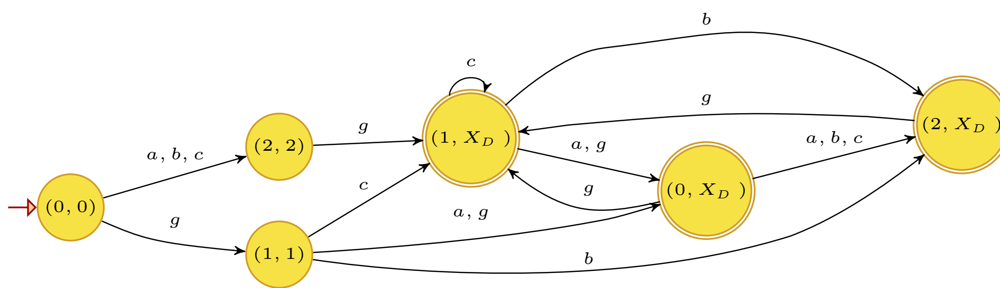


(a) $H_{R,1}$.

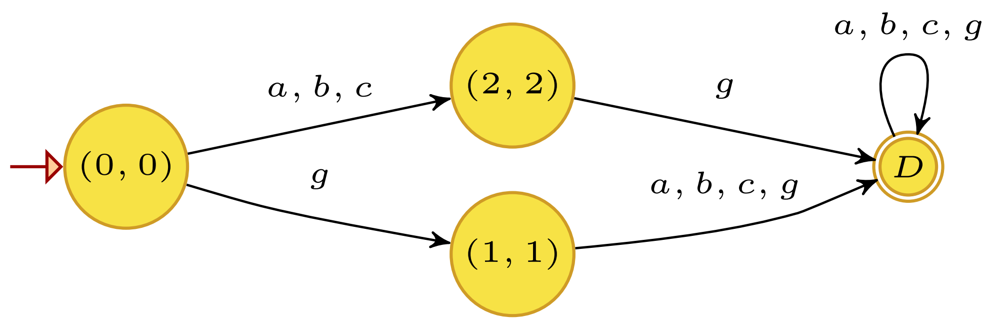
(b) $H_{R,2}$.

(c) $H_{R,3}$.

Figura 3.23: Autômatos $H_{R,1}$, $H_{R,2}$ e $H_{R,3}$ do Exemplo 7.



(a) H_C.



(b) H_C^D.

Figura 3.24: Autômatos H_C e H_C^D do Exemplo 7.

Capítulo 4

Uma biblioteca DESLAB para Controle Supervisório

Apesar de ser muito útil para a análise e projeto de SEDs, os autômatos são complexos demais para serem manipulados manualmente quando o tamanho do sistema aumenta. Para resolver esse problema é necessário utilizar algumas ferramentas computacionais. Dentre as ferramentas existentes, a mais moderna, flexível e de código aberto é o DESLAB ([14]).

Como mais uma contribuição deste trabalho, iremos apresentar neste capítulo uma biblioteca (também referida ao longo do texto como *toolbox*) para o DESLAB que implementa os principais métodos necessários para o projeto de supervisores. O código fonte completo desta biblioteca pode ser visto no apêndice A. Como maneira de ilustrar seu uso, um estudo de caso aplicado também será visto.

4.1 DESLAB

Desenvolvido pelo LCA na COPPE\UFRJ, o DESLAB é um programa de computação científica escrito em PYTHON que permite o desenvolvimento de algoritmos para análise e síntese de SEDs modelados por autômatos, por meio da integração de uma implementação dedicada de autômatos, e uso de bibliotecas PYTHON *open-source* para tratamento de grafos[1] e visualização em LaTeX[2]. O DESLAB proporciona uma sintaxe simples e um nível de abstração bem próximo das notações utilizadas na teoria de SEDs. Isso fica evidenciado na tabela 4.1, que mostra a correlação entre a notação utilizada em SEDs e a sintaxe do DESLAB.

O autômato de estados finitos (**fsa**) é o objeto central da estrutura do DESLAB, sendo, para tanto, disponibilizados métodos para manipulação, operação, análise e visualização. A partir do uso desse conjunto de métodos, e das estruturas e

[1]Biblioteca NETWORKX [24]
[2]Biblioteca GRAPHVIZ [25]

Notação	Sintaxe	Tipo
G	`G`	fsa
X	`G.X`	set
E	`G.Sigma`	set
$f(x,e)$	`G.delta(x,e)`	function
$\Gamma(x)$	`G.Gamma(x)`	function
X_0	`G.X0`	set
X_m	`G.Xm`	set
E_o	`G.Sigobs`	set
E_c	`G.Sigcon`	set

Tabela 4.1: Sintaxe para acesso às propriedades matemáticas de um autômato.

facilidades proporcionados pelo PYTHON, é possível estender o DESLAB com a criação de funções e bibliotecas personalizadas pelo usuário.

Com intuito de ilustrar o uso do DESLAB, considere um autômato G definido da seguinte maneira: $X = \{1,2,3,4\}$, $X_0 = \{1\}$, $X_m = \{1,2,3,4\}$, $E = \{u,b\}$, $E_c = \{b\}$, $E_o = \{b\}$, $f(1,u) = 2$, $f(1,b) = 3$, $f(2,b) = 4$, $f(3,u) = 4$. O código 4.1 contém a sequência de comandos que cria esse autômato por meio do objeto **fsa**, e representa o seu diagrama de transição de estados (por meio da função *draw*), gerando a figura 4.1. Note que que essa figura é gerada diretamente no formato PDF, permitindo utilização direta no LaTeX. É importante ressaltar que por ser implementado em PYTHON, o DESLAB pode ser utilizado tanto pela execução de um arquivo contendo uma sequência de comandos que se deseja executar (script), como diretamente por meio de comandos individuais no terminal do PYTHON.

Código 4.1: Definição e visualização do autômato G da figura 4.1.

```
1 from deslab import *
2 u,b,s1,s2,s3,s4 = syms('u b 1 2 3 4')
3 G_X = [s1,s2,s3,s4]
4 G_E = [u,b]
5 G_T = [(s1,u,s2),(s1,b,s3),(s2,b,s4),(s3,u,s4)]
6 G_X0 = [s1]
7 G_Xm = [s4]
8 G_Econ = [b]
9 G_Eobs = [b]
10 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ,Sigobs = G_Eobs, name='$G$')
11 draw(G,'figurecolor')
```

A versão mais atual do DESLAB, assim como de todas as suas bibliotecas, e todo o código-fonte encontram-se disponíveis em [26].

4.2 Uma toolbox para Controle Supervisório

Nesta seção a contribuição deste trabalho ao programa DESLAB será apresentada, por meio da análise detalhada de cada algoritmo desenvolvido. Para cada algoritmo

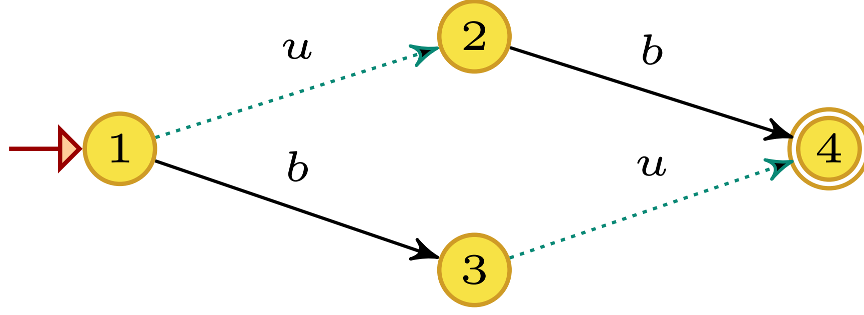


Figura 4.1: Autômato G gerado pela execução do código 4.1.

serão descritos: (i) o problema a ser resolvido, (ii) o algoritmo correspondente, e (iii) um exemplo simplificado de uso.

4.2.1 IsControllable

A controlabilidade de uma linguagem, apresentada na definição 2.18, é um dos principais requisitos no projeto de um sistema de controle supervisório. A função `iscontrollable` (listada no apêndice A.1) permite verificar a controlabilidade de uma linguagem regular K em relação a M e E_{uc}.

Descrição do algoritmo

O algoritmo implementado na função `iscontrollable` é aquele apresentado em [9]. Suas entradas são os autômatos G e H tais que $M = \mathcal{L}(G)$ e $K = \mathcal{L}_m(H)$. Uma busca exaustiva por uma sequência que viole a condição $\overline{K}E_{uc} \cap M \subseteq \overline{K}$ é feita comparando-se o conjunto dos eventos ativos em cada estado de $H \times G$, com o conjunto dos eventos ativos do respectivo estado em G. Se existir um evento não controlável ativo em $x_G \in X_G$, mas não em $x_{HG} \in X_{H\times G}$, em que x_G apareça como segundo componente, então K é não controlável em relação a M e E_{uc}. A formalização desse algoritmo, de complexidade $\mathcal{O}(|X|^2|E|)$, é apresentada a seguir.

Algoritmo 4.1 ([9])**.** *Seja G o autômato que representa o SED modelado, tal que $\mathcal{L}(G) = M$ e seja H o autômato que representa o comportamento desejado, tal que $\mathcal{L}_m(H) = K$. Seja o conjunto de eventos controláveis representado por $E_c \subseteq E$.*

Passo 1: *Construa o autômato $H_0 := H \times G$.*

Passo 2: *Para cada estado (y, x) de H_0, construa o conjunto dos eventos ativos não controláveis de G, fazendo:*

$$Enable_{uc}(x) := \{e \notin E_c \ : \ e \in \Gamma_G(x)\}.$$

Em seguida verifique se:

$$Enable_{uc}(x) \cap \Gamma_{H_0}((y,x)) \neq Enable_{uc}(x).$$

Caso essa verificação seja verdadeira para algum par (y,x), então K é não controlável em relação a M e E_{uc}.

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 4.1 possui a seguinte sintaxe:

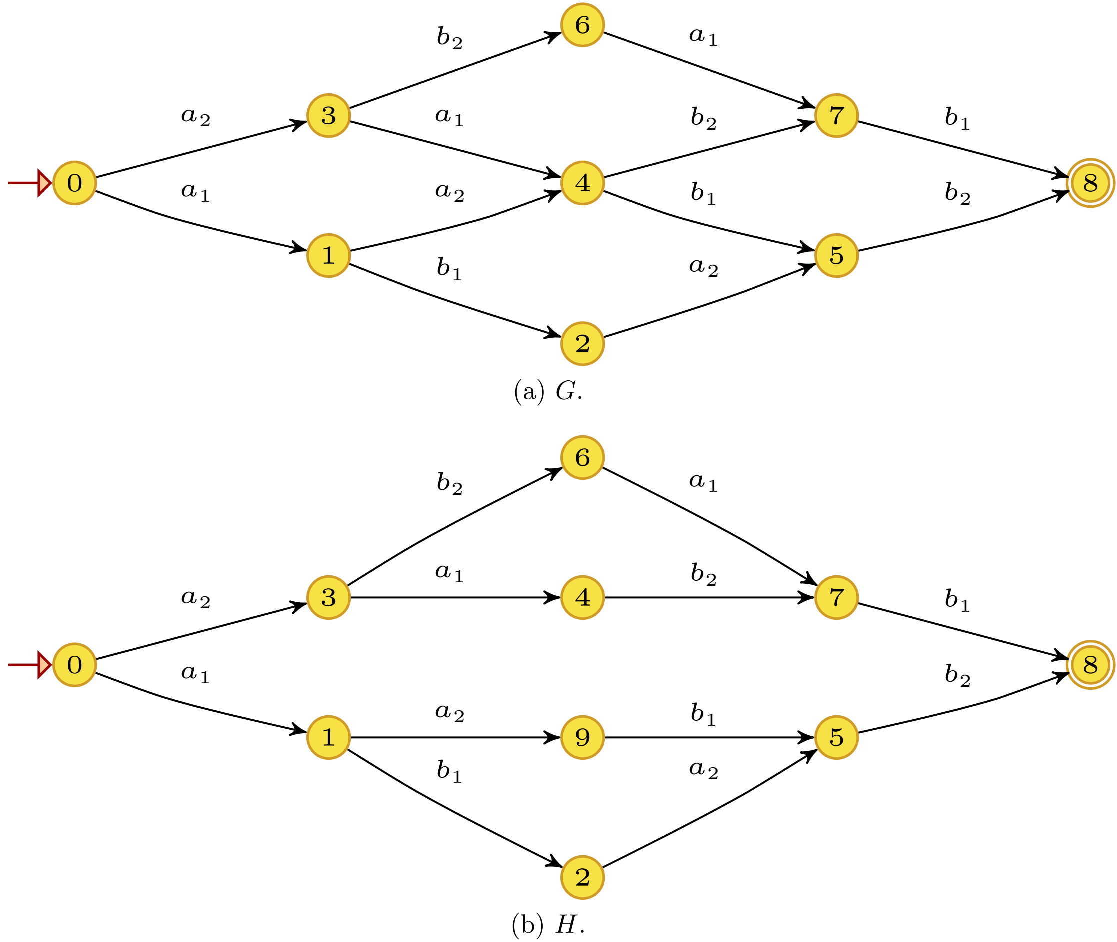
`controllable = iscontrollable(H,G)`,

na qual as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigcon` do autômato H contém o conjunto de eventos controláveis E_c. O retorno da função é uma variável booleana com estado `True` se a controlabilidade for verificada, e `False`, caso contrário.

Exemplo 4.1. *Considere como exemplo os autômatos G e H das figuras 4.2a e 4.2b, respectivamente, para os quais $E = [a_1, b_1, a_2, b_2]$ e $E_c = [a_1, b_1]$. Note que no autômato G (figura 4.2a), se o evento controlável a_1 ocorrer, saindo do estado 0 para o estado 1, a ocorrência posterior da sequência de eventos não controláveis a_2b_2 viola a especificação. Logo, esse sistema não é controlável. Isto pode ser visto através da execução do código 4.2, que gera o seguinte retorno no terminal:* `False`.

Código 4.2: Exemplo de uso da função `iscontrollable`.

```
1  from deslab import *
2  a1,b1,a2,b2,s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 = syms('a_1 b_1 a_2 b_2 0 1
3                                                   2 3 4 5 6 7 8 9')
4  G_X = [s0,s1,s2,s3,s4,s5,s6,s7,s8]
5  G_E = [a1,b1,a2,b2]
6  G_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s4),(s2,a2,s5),(s3,a1,s4),
7        (s3,b2,s6),(s4,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
8  G_X0 = [s0]
9  G_Xm = [s8]
10 G_Econ = [a1,b1]
11 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ, name='$G$')
12 H_X = G_X + [s9]
13 H_E = G_E
14 H_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s9),(s2,a2,s5),(s3,a1,s4),
15       (s3,b2,s6),(s9,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
16 H_X0 = G_X0
17 H_Xm = G_Xm
18 H_Econ = G_Econ
19 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ, name='$H$')
20 print iscontrollable(H,G)
```



(a) G.

(b) H.

Figura 4.2: Malha fechada formada pelo modelo do sistema (G) e pelo modelo da especificação (H) para o exemplo 4.1.

4.2.2 SupControllable

Conforme apresentado na seção 2.2.1, toda vez que uma linguagem K é não controlável em relação a M e E_{uc}, é importante saber qual é o maior subconjunto controlável possível de ser formado a partir de K. Na biblioteca desenvolvida neste trabalho implementam-se dois algoritmos com esse objetivo: (i) a função `supcontrollablepfclosed`, que implementa um algoritmo específico para o caso em que K é uma linguagem prefixo-fechada; (ii) a função `supcontrollablegeneral`, que implementa um algoritmo genérico (sem restrição para K), porém menos eficiente.

Todavia, é possível deixar a escolha do melhor algoritmo para a biblioteca, o que pode ser feito utilizando-se a função `supcontrollable` (listada no apêndice A.2). Nessa função, é feita uma análise prévia para saber se K é prefixo-fechada,

permitindo assim a seleção da função apropriada. A sintaxe da função é:

```
Hsupc = supcontrollable(H,G),
```

na qual as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigcon` do autômato H contém o conjunto de eventos controláveis E_c. Seu retorno é o autômato H_{supc} (objeto **fsa**) que marca a linguagem controlável suprema ($K^{\uparrow C}$). Caso essa linguagem não exista, o autômato vazio é retornado.

4.2.3 SupControllablePfClosed

Uma vez que é comum a existência de especificações K prefixo-fechadas, um algoritmo eficiente para o cálculo de $K^{\uparrow C}$ será implementado pela função `supcontrollablepfclosed` (listada no apêndice A.3).

Descrição do algoritmo

O algoritmo implementado na função `supcontrollablepfclosed` é aquele apresentado em [9]. Suas entradas são os autômatos G e H tais que $M = \mathcal{L}(G)$ e $K = \overline{K} = \mathcal{L}_m(H)$, e também, o conjunto dos eventos controláveis dado por $E_c \subseteq E$. O cálculo da linguagem controlável suprema ($K^{\uparrow C}$) é então realizado pela avaliação da seguinte expressão:

$$K^{\uparrow C} = K \backslash [(M \backslash K)/E_{uc}^*]E^*.$$

A formalização de um algoritmo implementando essa expressão possui complexidade $\mathcal{O}(|X|^2|E|)$, e encontra-se descrita a seguir.

Algoritmo 4.2 ([9]). *Seja G o autômato que representa o SED modelado, tal que $\mathcal{L}(G) = M$ e seja H o autômato que representa o comportamento desejado, tal que $\mathcal{L}_m(H) = \mathcal{L}(H) = K$. Seja o conjunto de eventos controláveis representado por $E_c \subseteq E$.*

Passo 1: *Construa o autômato G_m, com todos os estados marcados, da seguinte forma:*

$$G_m := (X_G, E, f_G, \Gamma_G, x_{0,G}, X_G).$$

Passo 2: *Construa o autômato A que gera e marca E^*, construindo um autômato com apenas um estado inicial marcado e um auto-laço para cada evento $\sigma \in E$.*

Passo 3: *Construa o autômato A_{uc} que gera e marca E_{uc}^*, construindo um autômato com apenas um estado inicial marcado e um auto-laço para cada evento $\sigma \in E_{uc}$.*

Passo 4: *Construa o autômato* $H_C := G_m \times H^C$*, sendo* H^C *o complementar de* H*.*

Passo 5: *Construa o autômato* Q *que marca o quociente entre as linguagens marcadas por* H_C *e* A_{uc}*.*

Passo 6: *Construa o autômato* Q_{concat} *que marca* $\mathcal{L}_m(Q)\mathcal{L}_m(A)$*, a concatenação das linguagens marcadas de* Q *e* A*.*

Passo 7: *Construa o autômato* $H_{supc} := H \times Q^C_{concat}$*, sendo* Q^C_{concat} *o complementar de* Q_{concat}*.*

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 4.2 possui a seguinte sintaxe:

```
Hsupc = supcontrollablepfclosed(H,G),
```

na qual as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K = \overline{K}$ e $\mathcal{L}(G) = M$, e a propriedade `Sigcon` do autômato H contém o conjunto de eventos controláveis E_c. Seu retorno é o autômato H_{supc} (objeto **fsa**) que marca a linguagem controlável suprema ($K^{\uparrow C}$). Caso essa linguagem não exista, a função retorna o autômato vazio.

Exemplo 4.2. *Considere novamente o sistema apresentado no exemplo 4.1, em que as linguagens do modelo (*M*) e da especificação (*K*) são modeladas, respectivamente, pelos autômatos* G *e* H*. Considere, agora, que a linguagem de especificação* K *é prefixo-fechada, ou seja,* $K = \overline{K}$ *e* $\mathcal{L}_m(H) = \mathcal{L}(H)$ *(*H *possui todos os estados marcados). Como concluído no exemplo 4.1, a condição de controlabilidade é violada pela ocorrência do evento* a_1 *no estado inicial. Logo, a retirada, em* $\mathcal{L}_m(H)$*, de todas as sequências que possuem* a_1 *como evento inicial, leva à uma linguagem controlável suprema para o sistema em questão. Isto pode ser comprovado pela execução do código 4.3, que gera como resultado o autômato* H_{supc} *cujo diagrama de transição de estados está representado na figura 4.3.*

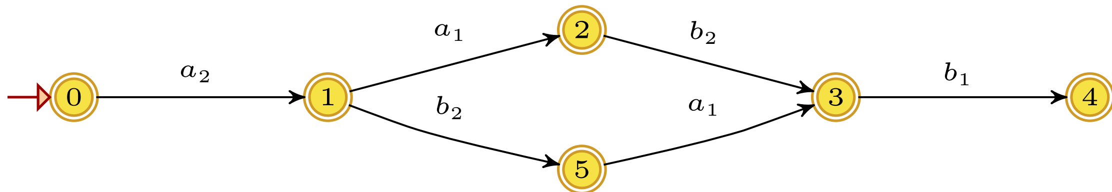


Figura 4.3: Autômato H_{supc} gerado pela execução do código 4.3.

Código 4.3: Exemplo de uso da função `supcontrollablepfclosed`.

```
1 from deslab import *
2 a1,b1,a2,b2,s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 = syms('a_1 b_1 a_2 b_2 0 1
3                                                   2 3 4 5 6 7 8 9')
4 G_X = [s0,s1,s2,s3,s4,s5,s6,s7,s8]
5 G_E = [a1,b1,a2,b2]
6 G_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s4),(s2,a2,s5),(s3,a1,s4),
7       (s3,b2,s6),(s4,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
8 G_X0 = [s0]
9 G_Xm = [s8]
10 G_Econ = [a1,b1]
11 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ, name='$G$')
12 H_X = G_X + [s9]
13 H_E = G_E
14 H_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s9),(s2,a2,s5),(s3,a1,s4),
15       (s3,b2,s6),(s9,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
16 H_X0 = G_X0
17 H_Xm = H_X
18 H_Econ = G_Econ
19 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ, name='$H$')
20 Hsupc = supcontrollablepfclosed(H,G)
21 draw(Hsupc,'figurecolor')
```

4.2.4 SupControllableGeneral

Para os casos em que a linguagem de especificação, representada por K, não for prefixo-fechada, um algoritmo alternativo para o cálculo de $K^{\uparrow C}$ será implementado pela função `supcontrollablegeneral` (listada no apêndice A.4).

Descrição do algoritmo

O algoritmo implementado na função `supcontrollablegeneral` é aquele apresentado em [9]. Suas entradas são os autômatos G e H tais que $M = \mathcal{L}(G)$ e $K = \mathcal{L}_m(H)$, e também, o conjunto dos eventos controláveis dado por $E_c \subseteq E$. O cálculo da linguagem controlável suprema ($K^{\uparrow C}$) é realizado por meio de um algoritmo iterativo, no qual a técnica de refinamento por produto é utilizada, e a identificação, seguida de remoção dos estados que levem à perda de controlabilidade é feita até que a convergência ocorra. Esse conceito é formalizado no algoritmo 4.3.

Algoritmo 4.3 ([9]). *Seja G o autômato que representa o SED modelado, tal que $\mathcal{L}(G) = M$ e seja H o autômato que representa o comportamento desejado, tal que $\mathcal{L}_m(H) = K$. Seja o conjunto de eventos controláveis representado por $E_c \subseteq E$.*

Passo 1: *Construa o autômato G_m, com todos os estados marcados, da seguinte forma:*

$$G_m := (X_G, E, f_G, \Gamma_G, x_{0,G}, X_G).$$

Passo 2: *Construa $H_0 := H \times G_m$.*

Passo 3: *Enquanto H_0 não for um autômato vazio, ou não for modificado pelos passos a seguir, faça:*

3.1: *Para cada estado* (y, x) *de* H_0*, construa o conjunto dos eventos ativos não-controláveis de* G*, fazendo:*

$$Enable_{uc}(x) := \{e \notin E_c \ : \ e \in \Gamma_G(x)\}.$$

Em seguida verifique se:

$$Enable_{uc}(x) \cap \Gamma_{H_0}((y, x)) \neq Enable_{uc}(x)$$

3.2: *Caso essa verificação seja verdadeira, remova de* H_0 *o estado* (y, x) *e todas as transições associadas a este.*

3.3: *Faça* $H_0 := Trim(H_0)$*.*

Passo 4: $H_{supc} := H_0$

A complexidade de pior caso deste algoritmo é $\mathcal{O}(|X|^4|E|)$, o que verifica a melhor eficiência computacional do algoritmo 4.2, e justifica a implementação de ambos.

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 4.3 possui a seguinte sintaxe:

`Hsupc = supcontrollablegeneral(H,G)`,

na qual as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigcon` do autômato H contém o conjunto de eventos controláveis E_c. Seu retorno é o autômato H_{supc} (objeto **fsa**) que marca a linguagem controlável suprema ($K^{\uparrow C}$). Caso essa linguagem não exista, a função retorna o autômato vazio.

Exemplo 4.3. *Considere, mais uma vez, o sistema apresentado no exemplo 4.1, para o qual as linguagens do modelo (*M*) e da especificação (*K*) são modeladas, respectivamente, pelos autômatos* G *e* H*. Como concluído no exemplo 4.1, a condição de controlabilidade é violada pela ocorrência do evento* a_1 *no estado inicial. Logo, a retirada, em* $\mathcal{L}_m(H)$*, de todas as sequências que possuam* a_1 *como evento inicial, leva à uma linguagem controlável suprema para o sistema em questão. Isto pode ser comprovado pela execução do código 4.4, que gera como resultado o autômato* H_{supc} *cujo diagrama de transição de estados está representado na figura 4.4.*

4.2.5 ConDat

Com o objetivo de facilitar o entendimento do comportamento de uma malha fechada de SEDs, a função `condat` (listada no apêndice A.33) implementa um algoritmo que

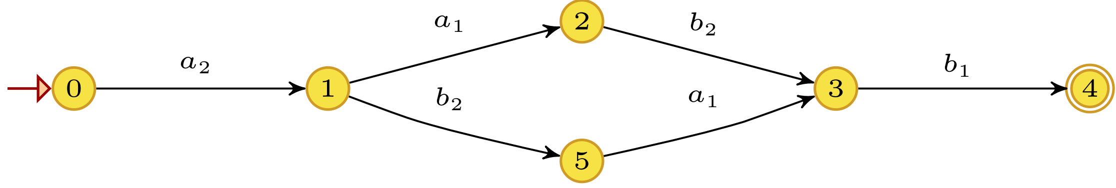


Figura 4.4: Autômato H_{supc} gerado pela execução do código 4.4.

Código 4.4: Exemplo de uso da função `supcontrollablegeneral`.

```
1 from deslab import *
2 a1,b1,a2,b2,s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 = syms('a_1 b_1 a_2 b_2 0 1
3                                                   2 3 4 5 6 7 8 9')
4 G_X = [s0,s1,s2,s3,s4,s5,s6,s7,s8]
5 G_E = [a1,b1,a2,b2]
6 G_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s4),(s2,a2,s5),(s3,a1,s4),
7       (s3,b2,s6),(s4,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
8 G_X0 = [s0]
9 G_Xm = [s8]
10 G_Econ = [a1,b1]
11 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ, name='$G$')
12 H_X = G_X + [s9]
13 H_E = G_E
14 H_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s9),(s2,a2,s5),(s3,a1,s4),
15       (s3,b2,s6),(s9,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
16 H_X0 = G_X0
17 H_Xm = G_Xm
18 H_Econ = G_Econ
19 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ, name='$H$')
20 Hsupc = supcontrollablegeneral(H,G)
21 draw(Hsupc,'figurecolor')
```

sintetiza quais eventos devem ser desabilitados por um supervisor em cada estado de sua especificação controlável em relação ao sistema e ao conjunto de eventos controláveis.

Descrição do algoritmo

O algoritmo implementado na função `condat` foi criado neste trabalho. Suas entradas são os autômatos G e H tais que $K = \mathcal{L}_m(H)$ é controlável em relação a $M = \mathcal{L}(G)$ e ao conjunto dos eventos controláveis E_c. A síntese da tabela contendo quais eventos devem ser desabilitados em cada estado do autômato H é realizada pela técnica de refinamento por produto, no qual, para cada estado do comportamento síncrono de H e G (par (x_H, x_G)) é armazenado o conjunto $\Gamma_G(x_G) \backslash \Gamma_H(x_H)$. Esse conceito é formalizado no algoritmo 4.4.

Algoritmo 4.4. *Seja H o autômato que representa o comportamento desejado K, tal que $\mathcal{L}_m(H) = K$ é controlável em relação a linguagem $\mathcal{L}(G) = M$ do SED modelado por G, e ao conjunto de eventos controláveis $E_c \subseteq E$.*

Passo 1: *Construa o autômato G_m, com todos os estados marcados, da seguinte forma:*

$$G_m := (X_G, E, f_G, \Gamma_G, x_{0,G}, X_G).$$

Passo 2: *Construa* $Prod := H \times G_m$.

Passo 3: *Para cada estado* (x_H, x_G) *de* $Prod$, *armazene na memória que no estado* x_H *os eventos pertencentes a* $\Gamma_G(x_G) \backslash \Gamma_H(x_H)$ *devem ser desabilitados.*

A complexidade de pior caso deste algoritmo é $\mathcal{O}(|X|^2|E|)$.

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 4.4 possui a seguinte sintaxe:

`eventstodisable = condat(H,G)`,

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigcon` do autômato H contêm o conjuntos dos eventos controláveis E_c. Além disso é necessário que a linguagem K seja controlável em relação a M e E_c. Seu retorno é um tabela do tipo **dictionary** que possui como chave os estados de H em que algum evento precisa ser desabilitado, e como valor os eventos que precisam ser desabilitados naquele estado. Adicionalmente, esta tabela é automaticamente impressa no terminal de maneira legível para o usuário.

Exemplo 4.4. *Considere o resultado apresentado no exemplo 4.3, no qual foi gerado o autômato* H_{supc} *que marca a linguagem controlável suprema, ilustrado na figura 4.4. A execução do código 4.21 após a execução prévia do código 4.4 resulta na seguinte resposta no terminal:*

```
Control Data:
0:  a_1
2:  b_1
```

Note que esta tabela está de acordo com a especificação e planta das figuras 4.4 e 4.2a, uma vez que no estado 0 *o evento controlável* a_1 *deve ser desabilitado pelo supervisor. No estado* 1 *ambos os eventos ativos na planta também estão ativos na especificação, não exigindo nenhum ação de controle. O mesmo ocorre no estado* 5, *porém no estado* 2 *o supervisor deve desabilitar o evento* b_1 *para manter o comportamento desejado. Por fim, no estado* 3 *mais uma vez não é necessária nenhuma ação de controle.*

Código 4.5: Exemplo de uso da função `condat`.

```
1 condat(Hsupc,G)
```

4.2.6 IsObservable (ObsVerifier)

O conceito de observabilidade de uma linguagem, apresentado na definição 2.20, é de grande importância para a teoria de controle supervisório com observação parcial. A função `isobservable` (listada no apêndice A.6) permite verificar a observabilidade de uma linguagem regular K em relação a M, E_c e E_o.

Ao longo de todo o capítulo 3, o conceito de observabilidade de uma linguagem foi estudado em detalhes, o que exigiu a análise e implementação dos principais métodos existentes na literatura, além do método proposto nesse trabalho. Assim, a biblioteca em estudo neste capítulo implementa as seguintes funções para a verificação da observabilidade de uma linguagem regular K em relação a M, E_c e E_o:

- `isobservable_observer` Algoritmo baseado no Observador, visto na seção 3.1.1;
- `isobservable_tsitsiklis` Algoritmo proposto por TSITSIKLIS [11], visto na seção 3.1.1;
- `isobservable_wang` Algoritmo proposto por WANG *et al.* [12], visto na seção 3.1.1;
- `isobservable_bb` Algoritmo proposto neste trabalho, visto na seção 3.2.1.

Todavia, para permitir o acesso aos algoritmos acima a partir de uma sintaxe unificada, a função `isobservable` foi implementada, sendo declarada da seguinte forma:

```
observable = isobservable(H,G,method=''),
```

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e as propriedades `Sigcon` e `Sigobs` do autômato H contêm, respectivamente, os conjuntos dos eventos controláveis E_c, e dos eventos observáveis E_o. Além disso, é opcional a utilização da entrada "method" (do tipo *string*), que corresponde a algum dos métodos acima, quais sejam: `observer`, `tsitsiklis`, `wang`, `bb`. Caso o método não seja escolhido, será utilizado o algoritmo `bb`. O retorno da função é uma variável booleana com estado `True` se a observabilidade for verificada, e `False`, caso contrário.

É importante ressaltar que a função utilizada por cada algoritmo para a construção de seu respectivo verificador também pode ser acessada diretamente pelo usuário, da seguinte forma:

```
V = obsverifier(H,G,method=''),
```

em que as entradas são as mesmas definidas para a função `isobservable`, porém seu retorno é o autômato verificador V construído de acordo com o método escolhido, conforme pode ser visto na listagem apresentada no apêndice A.7.

4.2.7 IsObservable_Observer (ObsVerifier_Observer)

Objetivando possibilitar a comparação entre os algoritmos verificadores da propriedade de observabilidade analisados no capítulo 3, foi implementada a função `isobservable_observer` (listada no apêndice A.8), que verifica a observabilidade de uma linguagem regular K em relação a M, E_c e E_o.

Descrição do algoritmo

O algoritmo implementado na função `isobservable_observer` é o algoritmo 3.1, analisado em detalhes na seção 3.1.1.

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 3.1 possui a seguinte sintaxe:

```
observable = isobservable_observer(H,G),
```

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e as propriedades `Sigcon` e `Sigobs` do autômato H contêm, respectivamente, os conjuntos dos eventos controláveis E_c, e dos eventos observáveis E_o. Seu retorno é uma variável booleana com estado `True` se a observabilidade for verificada, e `False`, caso contrário.

O acesso à função responsável pela construção do autômato verificador utilizado pelo algoritmo 3.1 é feito da seguinte forma:

```
V = obsverifier_observer(H,G),
```

em que as entradas são as mesmas definidas para a função `isobservable_observer`, porém seu retorno é o autômato V (objeto **fsa**) utilizado como verificador pelo algoritmo, conforme pode ser visto na listagem apresentada no apêndice A.9.

Exemplo 4.5. *Considere novamente o exemplo ilustrativo 3.1, no qual a linguagem do sistema é* $M = \mathcal{L}(G) = \overline{\{ub, bu\}}$*, e a linguagem da especificação é* $K = \mathcal{L}_m(H) = \{ub\}$*, as quais são modeladas, respectivamente, pelos autômatos* G *(figura 4.5a) e* H *(figura 4.5b). Considerando que* $E_c = \{b\}$ *e* $E_o = \{b\}$*, é fácil verificar que um supervisor não consegue diferenciar a sequência* u *da sequência* ε*, gerando ações de controle conflitantes para o evento* b*. Assim é possível concluir que*

K é não-observável em relação a M, E_c e E_o. O mesmo resultado é obtido com a execução do código 4.6, que gera no terminal o retorno `False`*, seguido do autômato H_{obs} ilustrado na figura 4.6.*

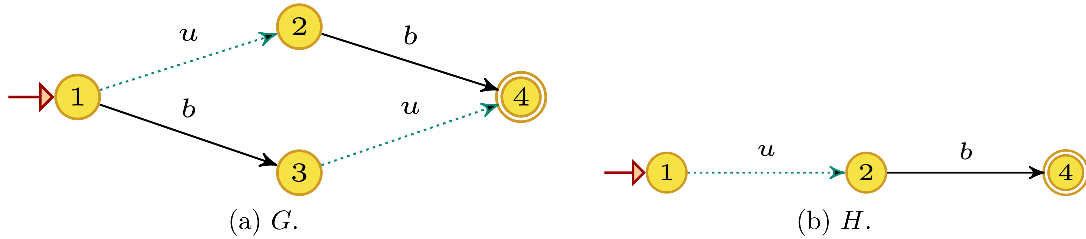


Figura 4.5: Malha fechada formada pelo modelo do sistema (G) e pelo modelo da especificação (H) para o exemplo 4.5.

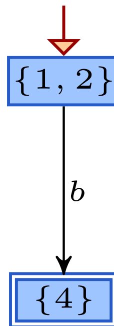


Figura 4.6: Autômato H_{obs} gerado pela execução do código 4.6.

4.2.8 IsObservable_Tsitsiklis (ObsVerifier_Tsitsiklis)

Conhecido durante décadas como o método mais eficiente para a verificação da observabilidade de uma linguagem regular K em relação a M, E_c e E_o, o algoritmo proposto por TSITSIKLIS [11] é aqui implementado por meio da função `isobservable_tsitsiklis` (listada no apêndice A.10).

Descrição do algoritmo

O algoritmo implementado na função `isobservable_tsitsiklis` é o algoritmo 3.2, analisado em detalhes na seção 3.1.1.

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 3.2 possui a seguinte sintaxe:

```
observable = isobservable_tsitsiklis(H,G),
```

Código 4.6: Exemplo de uso da função `isobservable_observer`.

```
1 from deslab import *
2 u,b,s1,s2,s3,s4 = syms('u b 1 2 3 4')
3 G_X = [s1,s2,s3,s4]
4 G_E = [u,b]
5 G_T = [(s1,u,s2),(s1,b,s3),(s2,b,s4),(s3,u,s4)]
6 G_X0 = [s1]
7 G_Xm = [s4]
8 G_Econ = [b]
9 G_Eobs = [b]
10 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ,Sigobs = G_Eobs, name='$G$')
11 H_X = [s1,s2,s4]
12 H_E = [u,b]
13 H_T = [(s1,u,s2),(s2,b,s4)]
14 H_X0 = [s1]
15 H_Xm = [s4]
16 H_Econ = G_Econ
17 H_Eobs = G_Eobs
18 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ,Sigobs = H_Eobs, name='$H$')
19 print isobservable_observer(H,G)
20 draw(obsverifier_observer(H,G),'figurecolor')
```

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e as propriedades `Sigcon` e `Sigobs` do autômato H contêm, respectivamente, os conjuntos dos eventos controláveis E_c, e dos eventos observáveis E_o. Seu retorno é uma variável booleana com estado `True` se a observabilidade for verificada, e `False`, caso contrário.

Para permitir ao usuário o acesso à função responsável pela construção do autômato verificador utilizado pelo algoritmo, a seguinte sintaxe é oferecida:

`V = obsverifier_tsitsiklis(H,G)`,

em que as entradas são as mesmas definidas para a função `isobservable_tsitsiklis`, porém seu retorno é o autômato V (objeto **fsa**) utilizado como verificador pelo algoritmo, conforme pode ser visto na listagem apresentada no apêndice A.11.

Exemplo 4.6. *Considere novamente o exemplo 4.5, no qual as linguagens do sistema (M) e da especificação (K) são modeladas pelos autômatos G (figura 4.5a) e H (figura 4.5b), sendo $E_c = \{b\}$ e $E_o = \{b\}$. Substituindo-se as linhas* 19 *e* 20 *do código 4.6 pelas linhas* 1 *e* 2 *do código 4.7, obtém-se o mesmo que no exemplo 4.5, porém, a segunda linha do código 4.7 retorna o autômato verificador da figura 4.7.*

Código 4.7: Exemplo de uso da função `isobservable_tsitsiklis`.

```
1 print isobservable_tsitsiklis(H,G)
2 draw(obsverifier_tsitsiklis(H,G),'figurecolor')
```

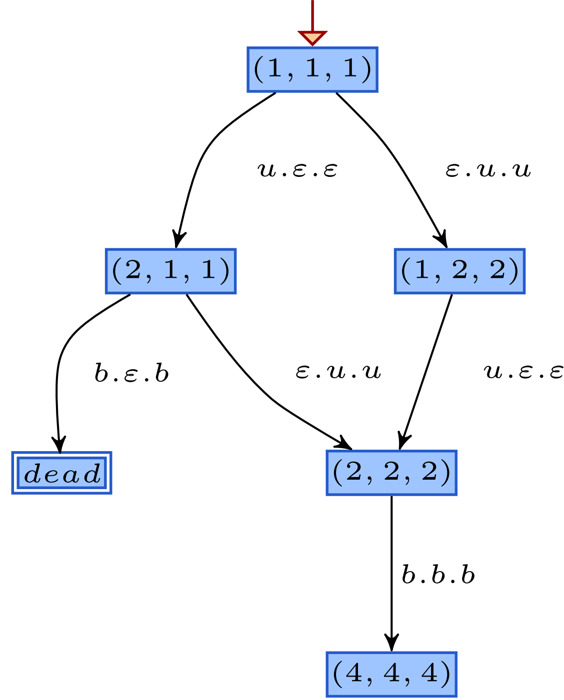


Figura 4.7: Autômato *ObsTest* gerado pela execução do código 4.7.

4.2.9 IsObservable_Wang (ObsVerifier_Wang)

Conhecido como o método mais eficiente que existe na literatura para verificação de observabilidade de uma linguagem regular K em relação a M, E_c e E_o, o algoritmo proposto por WANG *et al.* [12] é implementado nesta biblioteca por meio da função `isobservable_wang` (listada no apêndice A.12).

Descrição do algoritmo

O algoritmo implementado na função `isobservable_wang` é o algoritmo 3.3, analisado em detalhes na seção 3.1.1. O diagnosticador utilizado para a verificação de diagnosticabilidade é o proposto por MOREIRA *et al.* [13].

Definição e Exemplo

A função desenvolvida para implementar este algoritmo possui a seguinte sintaxe:

```
observable = isobservable_wang(H,G),
```

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e as propriedades `Sigcon` e `Sigobs` do autômato H contêm, respectivamente, os conjuntos dos eventos controláveis E_c, e dos eventos observáveis E_o. Seu retorno é uma variável booleana com estado `True` se a observabilidade for verificada, e

`False`, caso contrário. Note que esta função faz uso da função `obstodiag` (listada no apêndice A.14).

Para permitir ao usuário o acesso à função responsável pela construção do autômato verificador utilizado pelo algoritmo, a seguinte sintaxe é oferecida:

```
V = obsverifier_wang(H,G),
```

em que as entradas são as mesmas definidas para a função `isobservable_wang`, porém seu retorno é o autômato V (objeto **fsa**) utilizado como verificador pelo algoritmo, conforme pode ser visto na listagem apresentada no apêndice A.13. Note que esta função também faz uso da função `obstodiag` (listada no apêndice A.14).

Exemplo 4.7. *Considere novamente o exemplo 4.5, e suponha que as linguagens do sistema (M) e da especificação (K) sejam modeladas pelos autômatos G (figura 4.5a) e H (figura 4.5b), e que $E_c = \{b\}$ e $E_o = \{b\}$. Substituindo-se as linhas* 19 *e* 20 *do código 4.6 pelas linhas* 1 *e* 2 *do código 4.8, obtém-se o mesmo que no exemplo 4.5, porém, a segunda linha do código 4.8 retorna o autômato verificador da figura 4.8.*

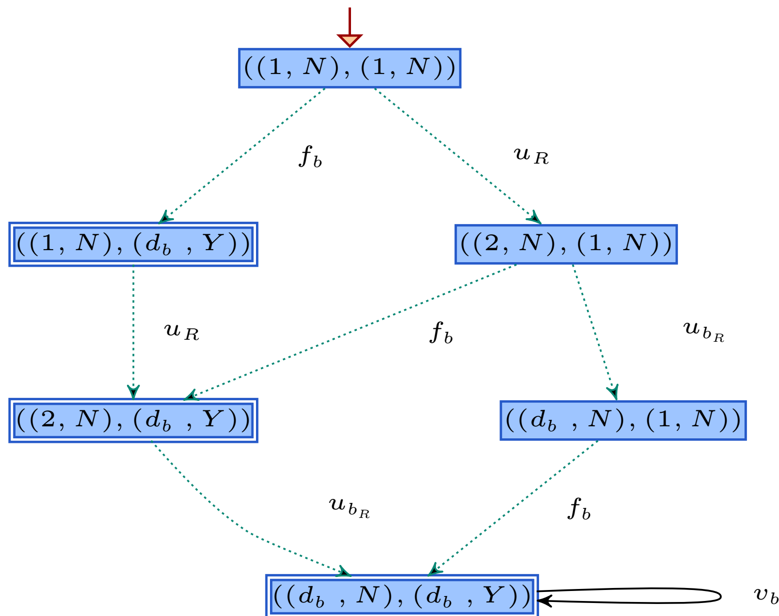


Figura 4.8: Autômato V_{diag} gerado pela execução do código 4.8.

Código 4.8: Exemplo de uso da função `isobservable_wang`.

```
1 print isobservable_wang(H,G)
2 draw(obsverifier_wang(H,G),'figurecolor')
```

4.2.10 IsObservable_BB (ObsVerifier_BB)

A função `isobservable_bb` (listada no apêndice A.15) implementa o algoritmo proposto no capítulo 3 deste trabalho para a verificação de observabilidade de uma linguagem regular K em relação a M, E_c e E_o.

Descrição do algoritmo

O algoritmo implementado na função `isobservable_bb` é o algoritmo 3.5, apresentado na seção 3.2.1.

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 3.5 possui a seguinte sintaxe:

`observable = isobservable_bb(H,G)`,

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e as propriedades `Sigcon` e `Sigobs` do autômato H contêm, respectivamente, os conjuntos dos eventos controláveis E_c, e dos eventos observáveis E_o. Seu retorno é uma variável booleana com estado `True` se a observabilidade for verificada, e `False`, caso contrário.

Para permitir ao usuário o acesso à função responsável pela construção do autômato verificador utilizado pelo algoritmo, a seguinte sintaxe é oferecida:

`V = obsverifier_bb(H,G)`,

em que as entradas são as mesmas definidas para a função `isobservable_bb`, porém seu retorno é o autômato V (objeto **fsa**) utilizado como verificador pelo algoritmo, conforme pode ser visto na listagem apresentada no apêndice A.16.

Exemplo 4.8. *Considere novamente o exemplo 4.5, no qual as linguagens do sistema (M) e da especificação (K) são modeladas pelos autômatos G (figura 4.5a) e H (figura 4.5b), sendo $E_c = \{b\}$ e $E_o = \{b\}$. Substituindo-se as linhas* 19 *e* 20 *do código 4.6 pelas linhas* 1 *e* 2 *do código 4.9, obtém-se o mesmo que no exemplo 4.5, porém, a segunda linha do código 4.9 retorna o autômato verificador da figura 4.9.*

Código 4.9: Exemplo de uso da função `isobservable_bb`.

```
1 print isobservable_bb(H,G)
2 draw(obsverifier_bb(H,G),'figurecolor')
```

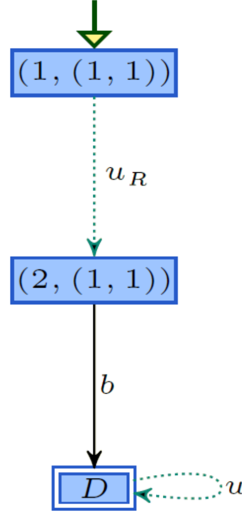


Figura 4.9: Autômato V gerado pela execução do código 4.9.

4.2.11 IsCoobservable (CoobsVerifier)

A extensão do conceito de observabilidade para o caso de controle supervisório descentralizado é feita pelo conceito de coobservabilidade (apresentada na seção 2.2.3). A função `iscoobservable` (listada no apêndice A.17) permite verificar a coobservabilidade de uma linguagem regular K em relação a M, $E_{o,i}$ e $E_{c,i}$, com $i \in \{1, 2, ..., n\}$.

Com o intuito de possibilitar a comparação do algoritmo proposto nesse trabalho, com o mais eficiente existente na literatura, a biblioteca em estudo neste capítulo implementa ambos os métodos para a verificação da coobservabilidade de uma linguagem regular K em relação a M, $E_{o,i}$ e $E_{c,i}$, com $i \in \{1, 2, ..., n\}$, a saber:

- `iscoobservable_wang` Algoritmo proposto por WANG *et al.* [12], visto na seção 3.5.1;

- `iscoobservable_bb` Algoritmo proposto neste trabalho, visto na seção 3.5.2.

Todavia, para permitir o acesso aos algoritmos acima a partir de uma sintaxe unificada, a função `iscoobservable` foi também implementada, sendo declarada da seguinte forma:

```
coobservable = iscoobservable(H,G,Ec,Eo,method=''),
```

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$. As entradas E_c e E_o correspondem a listas de tamanho n, nas quais o i-ésimo elemento contém, respectivamente, o conjunto dos eventos controláveis pelo

agente i ($E_{c,i}$), e o conjunto dos eventos observáveis pelo agente i ($E_{o,i}$). Além disso, é opcional a utilização da entrada "method" (do tipo *string*), que corresponde a algum dos métodos acima, quais sejam: `wang` ou `bb`. Caso o método não seja escolhido, será utilizado o algoritmo `bb`. O retorno da função é uma variável booleana com estado `True` se a coobservabilidade for verificada, e `False`, caso contrário.

Por fim, é importante ressaltar que a função utilizada em cada algoritmo para a construção de seu respectivo verificador, também pode ser acessada diretamente pelo usuário. Tal função possui a seguinte sintaxe:

```
V = coobsverifier(H,G,Ec,Eo,method=''),
```

em que as entradas são as mesmas definidas para a função `iscoobservable`, porém seu retorno é o autômato verificador V construído de acordo com o método escolhido, conforme pode ser visto na listagem apresentada no apêndice A.18.

4.2.12 IsCoobservable_Wang (CoobsVerifier_Wang)

O algoritmo proposto por WANG *et al.* [12] é o método mais eficiente para verificação de coobservabilidade de uma linguagem regular K em relação a M, $E_{o,i}$ e $E_{c,i}$, com $i \in \{1, 2, ..., n\}$, encontrado na literatura. Para o caso descentralizado, esse algoritmo é implementado pela biblioteca através da função `iscoobservable_wang` (listada no apêndice A.19).

Descrição do algoritmo

O algoritmo implementado na função `iscoobservable_wang` é o algoritmo 3.7, analisado em detalhes na seção 3.5.1. O diagnosticador utilizado para a verificação de diagnosticabilidade é o proposto por MOREIRA *et al.* [13].

Definição e Exemplo

A função desenvolvida para implementar este algoritmo possui a seguinte sintaxe:

```
coobservable = iscoobservable_wang(H,G,Ec,Eo),
```

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$. As entradas E_c e E_o correspondem a listas de tamanho n, nas quais o i-ésimo elemento contém, respectivamente, o conjunto dos eventos controláveis pelo agente i ($E_{c,i}$), e o conjunto dos eventos observáveis pelo agente i ($E_{o,i}$). O retorno da função é uma variável booleana com estado `True` se a coobservabilidade for verificada, e `False`, caso contrário. Note que esta função faz uso da função

`coobstodiag` (listada no apêndice A.21).

Para permitir ao usuário o acesso à função responsável pela construção do autômato verificador utilizado pelo algoritmo, a seguinte sintaxe é oferecida:

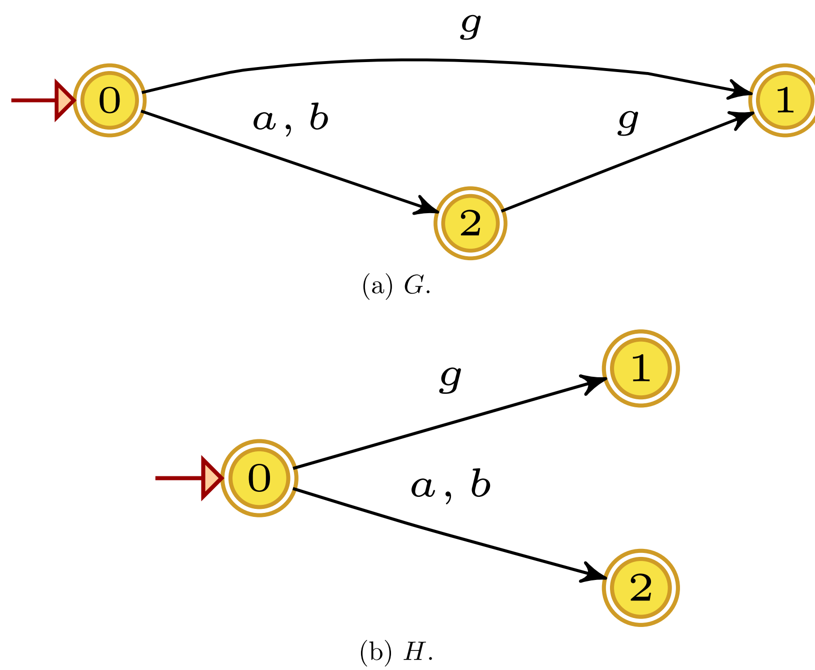
```
V = coobsverifier_wang(H,G,Ec,Eo),
```

em que as entradas são as mesmas definidas para a função `iscoobservable_wang`, porém seu retorno é o autômato V (objeto **fsa**) utilizado como verificador pelo algoritmo, conforme pode ser visto na listagem apresentada no apêndice A.20. Note que esta função também faz uso da função `coobstodiag` (listada no apêndice A.21).

Código 4.10: Exemplo de uso da função `iscoobservable_wang`.

```
1 from deslab import *
2 a,b,g,s0,s1,s2 = syms('a b g 0 1 2')
3 G_X = [s0,s1,s2]
4 G_E = [a,b,g]
5 G_T = [(s0,g,s1),(s0,a,s2),(s0,b,s2),(s2,g,s1)]
6 G_X0 = [s0]
7 G_Xm = G_X
8 G_Econ = [a,b,g]
9 G_Eobs = [a,b,g]
10 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ,Sigobs = G_Eobs, name='$G$')
11 H_X = [s0,s1,s2]
12 H_E = [a,b,g]
13 H_T = [(s0,g,s1),(s0,a,s2),(s0,b,s2)]
14 H_X0 = [s0]
15 H_Xm = H_X
16 H_Econ = [a,b,g]
17 H_Eobs = [a,b,g]
18 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ,Sigobs = H_Eobs, name='$H$')
19 Ec = [[g],[g]]
20 Eo = [[a],[b]]
21 print iscoobservable_wang(H,G,Ec,Eo)
22 draw(coobsverifier_wang(H,G,Ec,Eo),'figurecolor')
```

Exemplo 4.9. *Considere novamente o exemplo 2.11, em que a linguagem do sistema é* $M = \mathcal{L}(G) = \overline{\{g, bg, ag\}}$*, e a linguagem da especificação é* $K = \mathcal{L}_m(H) = \overline{\{g, b, a\}}$*, as quais encontram-se modeladas, respectivamente, pelos autômatos* G *(figura 4.10a) e* H *(figura 4.10b). Considere também que* $E = \{a, b, g\}$*, e os conjuntos de eventos controláveis e observáveis por cada agente* $i \in \{1, 2\}$ *são definidos por:* $E_{c,1} = \{g\}$*,* $E_{c,2} = \{g\}$*,* $E_{o,1} = \{a\}$ *e* $E_{o,2} = \{b\}$*. Note que após a ocorrência de* a*, apesar de* S_2 *não ser capaz de desabilitar* g*,* S_1 *o é. De maneira semelhante, após a ocorrência de* b*, apesar de* S_1 *não ser capaz de desabilitar* g*,* S_2 *o é. Logo, conclui-se que* K *é coobservável em relação a* M*,* $E_{c,1}$*,* $E_{c,2}$*,* $E_{o,1}$ *e* $E_{o,2}$*. O mesmo resultado é obtido com a execução do código 4.10, que gera no terminal o retorno* `True`*, seguido do autômato* V_{diag} *ilustrado na figura 4.11.*



(a) G.

(b) H.

Figura 4.10: Malha fechada formada pelo modelo do sistema (G) e pelo modelo da especificação (H) para o exemplo 4.9.

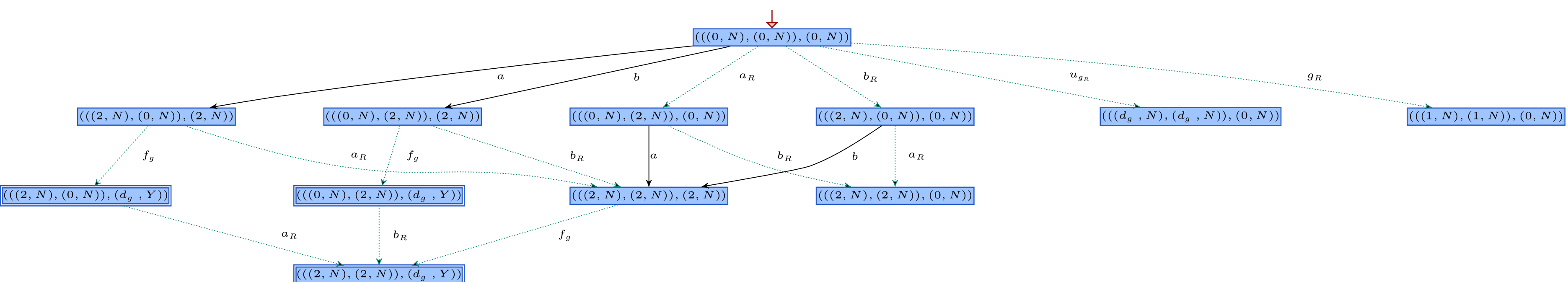


Figura 4.11: Autômato V_{diag} gerado pela execução do código 4.10.

4.2.13 IsCoobservable_BB (CoobsVerifier_BB)

A função `iscoobservable_bb` (listada no apêndice A.22) implementa o algoritmo proposto neste trabalho para a verificação da coobservabilidade de uma linguagem regular K em relação a M, $E_{o,i}$ e $E_{c,i}$, com $i \in \{1, 2, ..., n\}$.

Descrição do algoritmo

O algoritmo implementado na função `iscoobservable_bb` é o algoritmo 3.8, apresentado na seção 3.5.2.

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 3.8 possui a seguinte sintaxe:

```
coobservable = iscoobservable_bb(H,G,Ec,Eo),
```

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$. As entradas E_c e E_o correspondem a listas de tamanho n, nas quais o i-ésimo elemento contém, respectivamente, o conjunto dos eventos controláveis pelo agente i ($E_{c,i}$), e o conjunto dos eventos observáveis pelo agente i ($E_{o,i}$). O retorno da função é uma variável booleana com estado `True` se a coobservabilidade for verificada, e `False`, caso contrário.

Para permitir o acesso à função responsável pela construção do autômato verificador utilizado pelo algoritmo, utiliza-se a função `coobsverifier_bb`, cujo código fonte está listado no apêndice A.23. A seguinte sintaxe é oferecida:

```
V = coobsverifier_bb(H,G,Ec,Eo),
```

em que as entradas são as mesmas definidas para a função `iscoobservable_bb`, porém seu retorno é o autômato V (objeto **fsa**) utilizado como verificador pelo algoritmo.

Exemplo 4.10. *Considere novamente o exemplo 4.9, em que as linguagens do sistema (M) e da especificação (K) são modeladas pelos autômatos G (figura 4.10a) e H (figura 4.10b). Assim como no exemplo anterior, considere que os conjuntos de eventos controláveis e observáveis por cada agente $i \in \{1, 2\}$ são definidos por $E_{c,1} = \{a, g\}$, $E_{c,2} = \{b, g\}$, $E_{o,1} = \{a\}$ e $E_{o,2} = \{b\}$. Substituindo-se as linhas 21 e 22 do código 4.10 pelas linhas 1 e 2 do código 4.11, obtém-se o mesmo que no exemplo 4.9, porém, a segunda linha do código 4.11 retorna o autômato verificador V ilustrado na figura 4.12.*

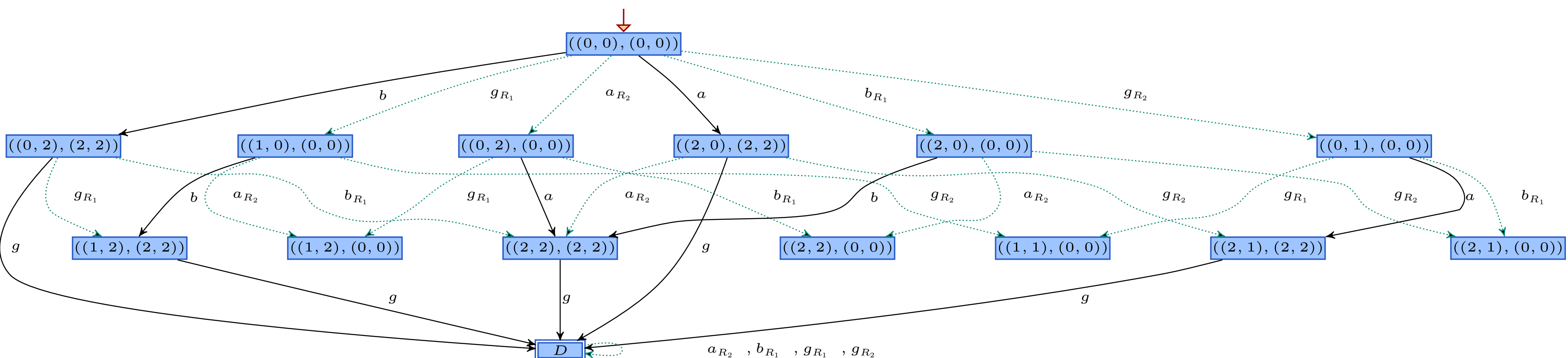


Figura 4.12: Autômato V gerado pela execução do código 4.11.

Código 4.11: Exemplo de uso da função `iscoobservable_bb`.

```
1 print iscoobservable_bb(H,G,Ec,Eo)
2 draw(coobsverifier_bb(H,G,Ec,Eo),'figurecolor')
```

4.2.14 IsNormal (NormalityVerifier)

Apresentado na definição 2.21, o conceito de normalidade possui relevante aplicação para a teoria de controle supervisório com observação parcial de eventos. A função `isnormal` (listada no apêndice A.24) permite verificar se uma linguagem regular K é normal em relação a M e E_o.

Diferentemente da propriedade de observabilidade, cuja verificação foi analisada por diferentes estudos, poucos desenvolvimentos encontram-se disponíveis na literatura para a verificação da normalidade. Todavia, essa biblioteca implementa os dois métodos apresentados no capítulo 3 para a verificação da normalidade de uma linguagem regular K em relação a M e E_o, quais sejam:

- `isnormal_observer` Algoritmo baseado no Observador, visto na seção 3.1.2;
- `isnormal_bb` Algoritmo proposto neste trabalho, visto na seção 3.3.1.

O acesso aos algoritmos acima pode ser feito a partir da função `isnormal` da seguinte forma:

`normal = isnormal(H,G,method='')`,

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigobs` do autômato H contém o conjunto dos eventos observáveis E_o. Além disso, é opcional a utilização da entrada "method" (do tipo *string*), que corresponde a algum dos métodos acima, quais sejam: `observer` ou `bb`. Caso o método não seja escolhido, será utilizado o algoritmo `bb`. O retorno da função é uma variável booleana com estado `True` se a normalidade for verificada, e `False`, caso contrário.

Por fim, é importante ressaltar que a função utilizada em cada algoritmo para a construção de seu respectivo verificador, também pode ser acessada diretamente pelo usuário. Tal função possui a seguinte sintaxe:

`V = normalityverifier(H,G,method='')`,

em que as entradas são as mesmas definidas para a função `isnormal`, porém seu retorno é o autômato verificador V construído de acordo com o método escolhido, conforme pode ser entendido na listagem apresentada no apêndice A.25.

4.2.15 IsNormal_Observer (NormalityVerifier_Observer)

Objetivando possuir um meio de comparação para o algoritmo verificador da normalidade proposto neste trabalho, a função `isnormal_observer` (listada no apêndice A.26) foi também implementada. Essa função verifica a normalidade de uma linguagem regular K em relação a M e E_o a partir de uma implementação direta da definição desta propriedade.

Descrição do algoritmo

O algoritmo implementado na função `isnormal_observer` é o algoritmo 3.4, analisado em detalhes na seção 3.1.2.

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 3.4 possui a seguinte sintaxe:

```
normal = isnormal_observer(H,G),
```

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigobs` do autômato H contém o conjunto dos eventos observáveis E_o. Seu retorno da função é uma variável booleana com estado `True` se a normalidade for verificada, e `False`, caso contrário.

Para permitir ao usuário o acesso à função responsável pela construção do autômato verificador utilizado pelo algoritmo, a seguinte sintaxe é oferecida:

```
V = normalityverifier_observer(H,G),
```

em que as entradas são as mesmas definidas para a função `isnormal_observer`, porém seu retorno é o autômato V (objeto **fsa**) utilizado como verificador pelo algoritmo. A função `normalityverifier_observer` encontra-se listada no apêndice A.27.

Exemplo 4.11. *Considere novamente o exemplo 4.5, onde as linguagens do sistema (M) e da especificação (K) são modeladas pelos autômatos G (figura 4.5a) e H (figura 4.5b). Considerando que $E_c = \{b\}$ e $E_o = \{b\}$, é fácil verificar que a sequência $bu \in P^{-1}[P(ub)] \cap M$, porém $bu \notin \overline{K}$, o que permite concluir que K é não-normal em relação a M e E_o. O mesmo resultado é obtido com a execução do código 4.12, que gera no terminal o retorno `False`, seguido do autômato V_{norm} ilustrado na figura 4.13.*

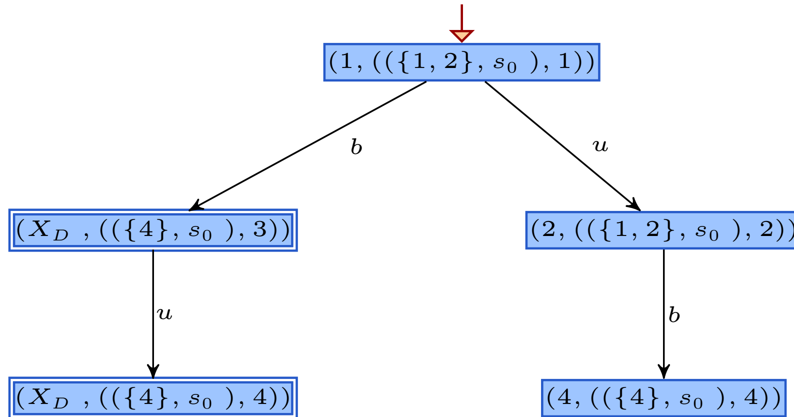


Figura 4.13: Autômato V_{norm} gerado pela execução do código 4.12.

Código 4.12: Exemplo de uso da função `isnormal_observer`.

```
1 from deslab import *
2 u,b,s1,s2,s3,s4 = syms('u b 1 2 3 4')
3 G_X = [s1,s2,s3,s4]
4 G_E = [u,b]
5 G_T = [(s1,u,s2),(s1,b,s3),(s2,b,s4),(s3,u,s4)]
6 G_X0 = [s1]
7 G_Xm = G_X
8 G_Econ = [b]
9 G_Eobs = [b]
10 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ,Sigobs = G_Eobs, name='$G$')
11 H_X = [s1,s2,s4]
12 H_E = [u,b]
13 H_T = [(s1,u,s2),(s2,b,s4)]
14 H_X0 = [s1]
15 H_Xm = [s4]
16 H_Econ = G_Econ
17 H_Eobs = G_Eobs
18 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ,Sigobs = H_Eobs, name='$H$')
19 print isnormal_observer(H,G)
20 draw(normalityverifier_observer(H,G),'figurecolor')
```

4.2.16 IsNormal_BB (NormalityVerifier_BB)

A função `isnormal_bb` implementa o algoritmo proposto neste trabalho para a verificação da normalidade de uma linguagem regular K em relação a M e E_o.

Descrição do algoritmo

O algoritmo implementado na função `isnormal_bb` é o algoritmo 3.6, apresentado na seção 3.3.1.

Definição e Exemplo

Conforme a listagem mostrada no apêndice A.28, a função desenvolvida para implementar o algoritmo 3.6 possui a seguinte sintaxe:

`normal = isnormal_bb(H,G)`,

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigobs` do autômato H contém o conjunto dos eventos observáveis E_o. O retorno dessa função é uma variável booleana com estado `True` se a normalidade for verificada, e `False`, caso contrário.

Para permitir ao usuário o acesso à função responsável pela construção do autômato verificador utilizado pelo algoritmo, a seguinte sintaxe é oferecida:

`V = normalityverifier_bb(H,G)`,

em que as entradas são as mesmas definidas para a função `isnormal_bb`, porém seu retorno é o autômato V (objeto **fsa**) utilizado como verificador pelo algoritmo, conforme pode ser entendido na listagem apresentada no apêndice A.29. No momento esta função funciona apenas como uma interface para o método `obsverifier_bb`, porém sua existência é justificada para manter o padrão aplicado a biblioteca, permitindo que, no futuro, melhorias particulares à normalidade sejam realizadas.

Exemplo 4.12. *Para verificar a normalidade da linguagem K em relação a M e E_o para o exemplo 4.11, basta substituir as linhas* 19 *e* 20 *do código 4.12, pelas linhas* 1 *e* 2 *do código 4.13, obtendo, então, o mesmo retorno no terminal, porém seguido do autômato V ilustrado na figura 4.9.*

Código 4.13: Exemplo de uso da função `isnormal_bb`.

```
1 print isnormal_bb(H,G)
2 draw(normalityverifier_bb(H,G),'figurecolor')
```

4.2.17 SupNormal

Conforme apresentado na seção 2.2.2, toda vez que uma linguagem K não é normal em relação a M e E_o, é importante saber qual é o maior subconjunto normal possível de ser formado a partir de K. Dois algoritmos que realizam a operação $\uparrow N$ foram implementados neste trabalho:

- A função `supnormalpfclosed`, que implementa um algoritmo específico para o caso em que K é uma linguagem prefixo-fechada;
- A função `supnormalgeneral`, que implementa um algoritmo genérico (sem restrição para K).

Todavia, é possível deixar a escolha do melhor algoritmo para a biblioteca, o que pode ser feito utilizando-se a função `supnormal`, listada no apêndice A.30. Nessa

função, é feita uma análise prévia para saber se K é prefixo-fechada, permitindo assim a seleção da função apropriada. A sintaxe da função é:

```
Hsupn = supnormal(H,G),
```

na qual as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigobs` do autômato H contém o conjunto de eventos observáveis E_o. Seu retorno é o autômato H_{supn} (objeto **fsa**) que marca a linguagem normal suprema ($K^{\uparrow N}$). Caso essa linguagem não exista, o autômato vazio é retornado.

4.2.18 SupNormalPfClosed

Como a existência de especificações K prefixo-fechadas é bastante comum, um algoritmo eficiente para o cálculo de $K^{\uparrow N}$ será implementado pela função `supnormalpfclosed` (listada no apêndice A.31).

Descrição do algoritmo

O algoritmo implementado na função `supnormalpfclosed` é aquele apresentado em [9]. Suas entradas são os autômatos G e H tais que $M = \mathcal{L}(G)$ e $K = \overline{K} = \mathcal{L}_m(H)$, e também, o conjunto dos eventos observáveis dado por $E_o \subseteq E$. O cálculo da linguagem normal suprema ($K^{\uparrow N}$) é então realizado pela avaliação da seguinte expressão:

$$K^{\uparrow N} = K \backslash (P^{-1}[P(M \backslash K)])E^*. \tag{4.1}$$

A formalização de um algoritmo implementando essa expressão possui complexidade exponencial, e encontra-se descrita a seguir.

Algoritmo 4.5 ([9]). *Seja G o autômato que representa o SED modelado, tal que $\mathcal{L}(G) = M$ e seja H o autômato que representa o comportamento desejado, tal que $\mathcal{L}_m(H) = \mathcal{L}(H) = K$. Seja o conjunto de eventos observáveis representado por $E_o \subseteq E$.*

Passo 1: *Construa o autômato G_m, com todos os estados marcados, da seguinte forma:*

$$G_m := (X_G, E, f_G, \Gamma_G, x_{0,G}, X_G).$$

Passo 2: *Construa o autômato $H_C := G_m \times H^C$, sendo H^C o complementar de H.*

Passo 3: *Construa o autômato $H_{C,obs} := Obs(H_C, E_o)$.*

Passo 4: *Construa $H^{sl}_{C,obs}$ adicionando auto-laços com todos os eventos $\sigma \in E_{uo}$ em todos os estados de $H_{C,obs}$.*

Passo 5: *Construa o autômato A que gera e marca E^*, construindo um autômato com apenas um estado inicial marcado e um auto-laço para cada evento $\sigma \in E$.*

Passo 6: *Construa o autômato H_{concat} que marca $\mathcal{L}_m(H^{sl}_{C,obs})\mathcal{L}_m(A)$, a concatenação das linguagens marcadas de $H^{sl}_{C,obs}$ e A.*

Passo 7: *Construa o autômato $H_{supn} := H \times H^{C}_{concat}$, sendo H^{C}_{concat} o complementar de H_{concat}.*

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 4.5 possui a seguinte sintaxe:

```
Hsupn = supnormalpfclosed(H,G),
```

na qual as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K = \overline{K}$ e $\mathcal{L}(G) = M$, e a propriedade `Sigobs` do autômato H contém o conjunto de eventos observáveis E_o. Seu retorno é o autômato H_{supn} (objeto **fsa**) que marca a linguagem normal suprema ($K^{\uparrow N}$). Caso essa linguagem não exista, a função retorna o autômato vazio.

Exemplo 4.13. *Considere os autômatos G e H do exemplo 4.1, ilustrados respectivamente nas figuras 4.2a e 4.2b. Porém, defina agora que $E_{uo} = \{a_2\}$, e considere a linguagem de especificação K sendo prefixo-fechada, ou seja, $K = \overline{K}$. Assim, o autômato H que marca K deve ter todos os estados marcados. É possível notar que $K = \mathcal{L}_m(H)$ é não normal em relação a $M = \mathcal{L}(G)$ e E_o, pois existem sequências de mesma projeção a_1b_2, tais que uma se encontra dentro da linguagem especificada ($a_2a_1b_2$), e a outra não ($a_1a_2b_2$). Além deste, outro exemplo de violação de normalidade ocorre com duas sequências de projeção a_1b_1, a sequência especificada a_1b_1 e a não-especificada $a_2a_1b_1$. Logo, a linguagem normal suprema será gerada pela remoção de todas as sequências de $\mathcal{L}_m(H)$ que possuam como prefixo sequências de projeção a_1b_2 ou a_1b_1. Esta conclusão pode ser comprovada pela execução do código 4.14, que gera o autômato exibido na figura 4.14.*

4.2.19 SupNormalGeneral

Para os casos em que a linguagem de especificação, representada por K, não for prefixo-fechada, um algoritmo alternativo para o cálculo de $K^{\uparrow N}$ será implementado pela função `supnormalgeneral` (listada no apêndice A.32).

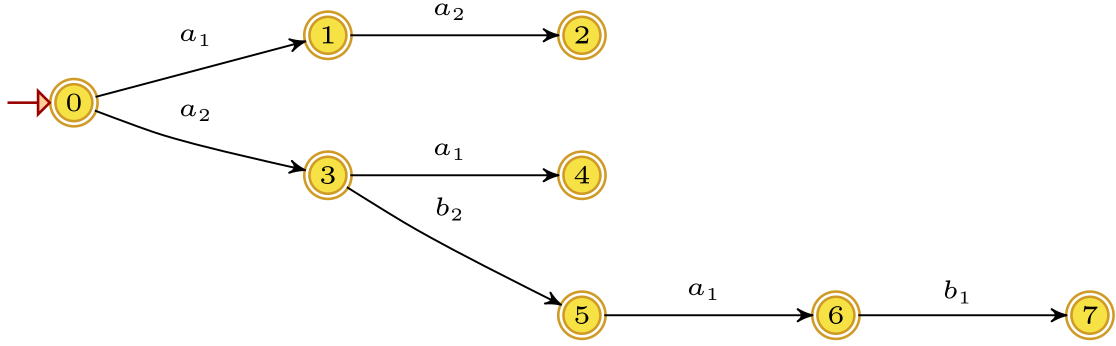


Figura 4.14: Autômato H_{supn} gerado pela execução do código 4.14.

Código 4.14: Exemplo de uso da função `supnormalpfclosed`.

```
1  from deslab import *
2  a1,b1,a2,b2,s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 = syms('a_1 b_1 a_2 b_2 0 1
3                                                  2 3 4 5 6 7 8 9')
4  G_X = [s0,s1,s2,s3,s4,s5,s6,s7,s8]
5  G_E = [a1,b1,a2,b2]
6  G_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s4),(s2,a2,s5),(s3,a1,s4),
7         (s3,b2,s6),(s4,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
8  G_X0 = [s0]
9  G_Xm = [s8]
10 G_Econ = [a1,b1,a2,b2]
11 G_Eobs = [a1,b1,b2]
12 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ,Sigobs = G_Eobs, name='$G$')
13 H_X = G_X + [s9]
14 H_E = G_E
15 H_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s9),(s2,a2,s5),(s3,a1,s4),
16        (s3,b2,s6),(s9,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
17 H_X0 = G_X0
18 H_Xm = H_X
19 H_Econ = G_Econ
20 H_Eobs = G_Eobs
21 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ,Sigobs = H_Eobs, name='$H$')
22 Hsupn = supnormalpfclosed(H,G)
23 draw(Hsupn,'figurecolor')
```

Descrição do algoritmo

O algoritmo implementado na função `supnormalgeneral` é aquele apresentado em [9]. Seu funcionamento é simples, e consiste no uso recursivo da equação (4.1), conforme a seguinte regra de iteração:

$$K_{i+1} = (\overline{K_i})^{\uparrow N} \cap K \ , \quad \text{com} \ \ K_0 = K.$$

A formalização de um algoritmo implementando essa expressão possui complexidade exponencial, e encontra-se descrita a seguir.

Algoritmo 4.6 ([9]). *Seja G o autômato que representa o SED modelado, tal que $\mathcal{L}(G) = M$ e seja H o autômato que representa o comportamento desejado, tal que $\mathcal{L}_m(H) = K$. Seja o conjunto de eventos observáveis denotado por $E_o \subseteq E$.*

Passo 1: *Faça $H_{i+1} := H$ e $H_i := \emptyset$.*

Passo 2: *Enquanto* $(\mathcal{L}(H_{i+1}) \neq \mathcal{L}(H_i)) \wedge H_{i+1} \neq \emptyset$ *faça:*

2.1: $H_i := H_{i+1}$

2.2: *Construa* $H_{i,m}$ *marcando todos os estados de* H_i*, fazendo:*

$$H_{i,m} := (X_{H_i}, E, f_{H_i}, \Gamma_{H_i}, x_{0,H_i}, X_{H_i}).$$

2.3: *Obtenha* $H_{supn} :=$ `supnormalpfclosed`$(H_{i,m}, G)$.

2.4: *Faça* $H_{i+1} := H_{supn} \times H$

Passo 3: $H_{supn} := Trim(H_{i+1})$

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 4.6 possui a seguinte sintaxe:

```
Hsupn = supnormalgeneral(H,G),
```

na qual as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e a propriedade `Sigobs` do autômato H contém o conjunto de eventos observáveis E_o. Seu retorno é o autômato H_{supn} (objeto **fsa**) que marca a linguagem normal suprema ($K^{\uparrow N}$). Caso essa linguagem não exista, a função retorna o autômato vazio.

Exemplo 4.14. *Considere novamente o exemplo 4.13, porém suponha que* $K = \mathcal{L}_m(H)$*, sendo* H *o autômato da figura 4.2b. Mais uma vez, a linguagem normal suprema será gerada pela remoção de todas as sequências de* $\mathcal{L}_m(H)$ *que possuam como prefixo sequências cujas projeções são* a_1b_2 *ou* a_1b_1*. Esta conclusão pode ser comprovada pela execução do código 4.15, que gera o autômato exibido na figura 4.15.*

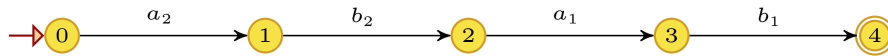


Figura 4.15: Autômato H_{supn} gerado pela execução do código 4.15.

4.2.20 SupControllableNormal

Visando possibilitar a síntese simultânea de uma linguagem controlável e normal, a função `supcontrollablenormal` (listada no apêndice A.33) implementa um algoritmo para a obtenção da linguagem controlável e normal suprema (vista na seção 2.2.2).

Código 4.15: Exemplo de uso da função `supnormalgeneral`.

```
1 from deslab import *
2 a1,b1,a2,b2,s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 = syms('a_1 b_1 a_2 b_2 0 1
3                                                   2 3 4 5 6 7 8 9')
4 G_X = [s0,s1,s2,s3,s4,s5,s6,s7,s8]
5 G_E = [a1,b1,a2,b2]
6 G_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s4),(s2,a2,s5),(s3,a1,s4),
7        (s3,b2,s6),(s4,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
8 G_X0 = [s0]
9 G_Xm = [s8]
10 G_Econ = [a1,b1,a2,b2]
11 G_Eobs = [a1,b1,b2]
12 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ,Sigobs = G_Eobs, name='$G$')
13 H_X = G_X + [s9]
14 H_E = G_E
15 H_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s9),(s2,a2,s5),(s3,a1,s4),
16        (s3,b2,s6),(s9,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
17 H_X0 = G_X0
18 H_Xm = [s8]
19 H_Econ = G_Econ
20 H_Eobs = G_Eobs
21 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ,Sigobs = H_Eobs, name='$H$')
22 Hsupn = supnormalgeneral(H,G)
23 draw(Hsupn,'figurecolor')
```

Descrição do algoritmo

O algoritmo implementado na função `supcontrollablenormal` é aquele apresentado em [9]. Suas entradas são os autômatos G e H tais que $M = \mathcal{L}(G)$ e $K = \mathcal{L}_m(H)$, e também, o conjunto dos eventos controláveis $E_c \subseteq E$, e o conjunto dos eventos observáveis $E_o \subseteq E$. O cálculo da linguagem controlável e normal suprema ($K^{\uparrow CN}$) é realizado pela execução iterativa das operações $\uparrow C$ e $\uparrow N$ até que a convergência ocorra. Iterações são necessárias porque a realização de uma das operações não garante a preservação da propriedade da outra. A formalização de um algoritmo implementando essa iteração possui complexidade exponencial e encontra-se descrita a seguir.

Algoritmo 4.7 ([9]). *Seja G o autômato que modela um SED, tal que $\mathcal{L}(G) = M$ e seja H o autômato que representa o comportamento desejado, tal que $\mathcal{L}_m(H) = K$. Seja o conjunto de eventos controláveis denotado por $E_c \subseteq E$, e o conjunto dos eventos observáveis representado por $E_o \subseteq E$.*

Passo 1: *Faça $H_{supcn} := H$ e $H_{prev} := \emptyset$.*

Passo 2: *Enquanto $\mathcal{L}(H_{supcn}) \neq \mathcal{L}(H_{prev})$ faça:*

2.1: $H_{prev} := H_{supcn}$

2.2: *Obtenha $H_{supc} := \texttt{supcontrollable}(H_{supcn}, G)$.*

2.3: *Obtenha $H_{supcn} := \texttt{supnormal}(H_{supc}, G)$.*

Definição e Exemplo

A função desenvolvida para implementar o algoritmo 4.7 possui a seguinte sintaxe:

`Hsupcn = supcontrollablenormal(H,G),`

em que as entradas H e G correspondem a autômatos tais que $\mathcal{L}_m(H) = K$ e $\mathcal{L}(G) = M$, e as propriedades `Sigcon` e `Sigobs` do autômato H contêm, respectivamente, os conjuntos dos eventos controláveis E_c, e dos eventos observáveis E_o. Seu retorno é o autômato H_{supcn} (objeto **fsa**) que marca a linguagem controlável e normal suprema ($K^{\uparrow CN}$). Caso essa linguagem não exista, a função retorna o autômato vazio.

Exemplo 4.15. *Considere os autômatos G e H vistos nos exemplos 4.3 e 4.14. Conforme o exemplo 4.3, $E_{uc} = \{a_2, b_2\}$. Ao mesmo tempo, de acordo com o exemplo 4.14, $E_{uo} = \{a_2\}$. Ao realizar a operação $\uparrow C$ em H obtém-se o autômato da figura 4.4, onde a linguagem marcada é $\{a_2b_2a_1b_1, a_2a_1b_2b_1\}$. A realização independente de $\uparrow N$ em H leva ao autômato da figura 4.15, que marca a linguagem $\{a_2b_2a_1b_1\}$. Como $\{a_2b_2a_1b_1\}$ pertence a ambas as linguagens, então $K^{\uparrow CN} = \{a_2b_2a_1b_1\}$. Este resultado pode ser comprovado pela execução do código 4.20, que resulta no autômato da figura 4.15.*

Código 4.16: Exemplo de uso da função `supcontrollablenormal`.

```
1 from deslab import *
2 a1,b1,a2,b2,s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 = syms('a_1 b_1 a_2 b_2 0 1
3                                                 2 3 4 5 6 7 8 9')
4 G_X = [s0,s1,s2,s3,s4,s5,s6,s7,s8]
5 G_E = [a1,b1,a2,b2]
6 G_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s4),(s2,a2,s5),(s3,a1,s4),
7       (s3,b2,s6),(s4,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
8 G_X0 = [s0]
9 G_Xm = [s8]
10 G_Econ = [a1,b1]
11 G_Eobs = [a1,b1,b2]
12 G = fsa(G_X,G_E,G_T,G_X0,G_Xm,Sigcon = G_Econ,Sigobs = G_Eobs, name='$G$')
13 H_X = G_X + [s9]
14 H_E = G_E
15 H_T = [(s0,a1,s1),(s0,a2,s3),(s1,b1,s2),(s1,a2,s9),(s2,a2,s5),(s3,a1,s4),
16       (s3,b2,s6),(s9,b1,s5),(s4,b2,s7),(s5,b2,s8),(s6,a1,s7),(s7,b1,s8)]
17 H_X0 = G_X0
18 H_Xm = [s8]
19 H_Econ = G_Econ
20 H_Eobs = G_Eobs
21 H = fsa(H_X,H_E,H_T,H_X0,H_Xm,Sigcon = H_Econ,Sigobs = H_Eobs, name='$H$')
22 Hsupcn = supcontrollablenormal(H,G)
23 draw(Hsupcn,'figurecolor')
```

4.3 Exemplo de aplicação da biblioteca

Como exemplo de aplicação real do DESLAB e da biblioteca para controle supervisório desenvolvida neste trabalho, será analisado o problema prático do controle de

tráfego de dois trens que compartilham uma ferrovia. Este problema foi considerado pela primeira vez em [1], porém também foi analisado em outros trabalhos, como, por exemplo, [15].

Conforme ilustrado pela figura 4.16, duas estações ferroviárias A e B são interligadas por um único trilho, o qual é dividido em quatro seções. Semáforos (ilustrados por *) e detectores (ilustrados por !) são instalados em várias das junções entre duas seções.

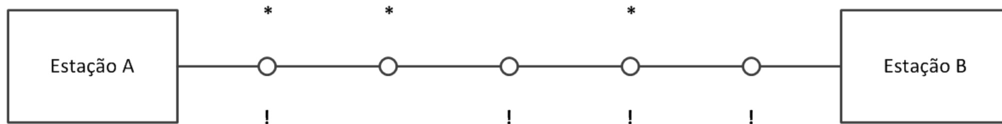
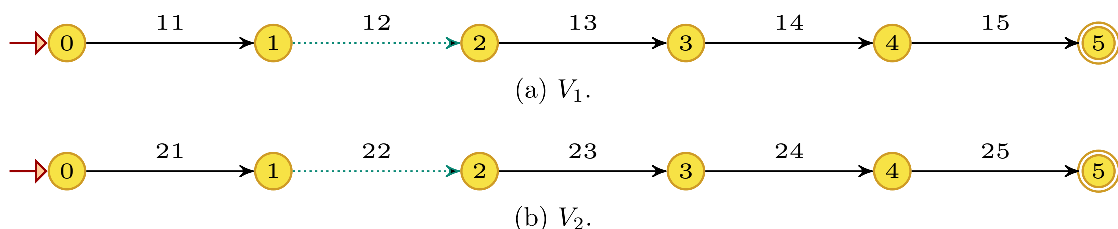


Figura 4.16: Diagrama do controle de uma ferrovia.

Dois veículos (trens) V_1 e V_2 compartilham o uso da ferrovia para se deslocar da estação A até a estação B. O veículo V_j está no estado 0 quando parado na estação A, no estado i quando na seção i, $i = 1, ..., 4$, e no estado 5 quando parado na estação B. Para evitar colisões, o controle dos semáforos deve garantir que nunca os dois veículos estejam na mesma seção da ferrovia simultaneamente.

Com estas informações, é possível prosseguir com a modelagem do comportamento de cada veículo. Defina que o evento ij corresponde ao veículo i entrando na seção j. Devido à ausência de semáforo nas junções entre as seções 2 e 3, e entre as seção 4 e a estação B, entende-se que os eventos 13, 23, 15 e 25 não são controláveis. Além disso, devido à ausência de detector na junção entre as seções 1 e 2, entende-se que os eventos 12 e 22 não são observáveis. Baseado nesse dado, é possível chegar ao código 4.17 que cria o autômato G para modelar o sistema. Para tanto, inicialmente, autômatos V_1 e V_2, com o comportamento independente de cada veículo viajando entre A e B são criados (figuras 4.17a e 4.17b). Na sequência, o comportamento global do sistema é sintetizado pela composição paralela entre os dois veículos, resultando no autômato G (figura 4.18).



(a) V_1.

(b) V_2.

Figura 4.17: Modelos independentes de cada veículo viajando entre as estações A e B.

Código 4.17: Construção do modelo.

```
from deslab import *
s0,s1,s2,s3,s4,s5,e11,e12,e13,e14,e15,e21,e22,e23,e24,e25 = syms('0 1 2 3 4 5 11
                                                        12 13 14 15 21 22 23 24 25')
V1_X = [s0,s1,s2,s3,s4,s5]
V1_E = [e11,e12,e13,e14,e15]
V1_T = [(s0,e11,s1),(s1,e12,s2),(s2,e13,s3),(s3,e14,s4),(s4,e15,s5)]
V1_X0 = [s0]
V1_Xm = [s5]
V1_Econ = [e11,e12,e14]
V1_Eobs = [e11,e13,e14,e15]
V1 = fsa(V1_X,V1_E,V1_T,V1_X0,V1_Xm,Sigcon = V1_Econ, Sigobs = V1_Eobs,
                                                            name='$V1$')
V2_X = [s0,s1,s2,s3,s4,s5]
V2_E = [e21,e22,e23,e24,e25]
V2_T = [(s0,e21,s1),(s1,e22,s2),(s2,e23,s3),(s3,e24,s4),(s4,e25,s5)]
V2_X0 = [s0]
V2_Xm = [s5]
V2_Econ = [e21,e22,e24]
V2_Eobs = [e21,e23,e24,e25]
V2 = fsa(V2_X,V2_E,V2_T,V2_X0,V2_Xm, Sigcon = V2_Econ, Sigobs = V2_Eobs,
                                                            name='$V2$')
G = V1//V2
```

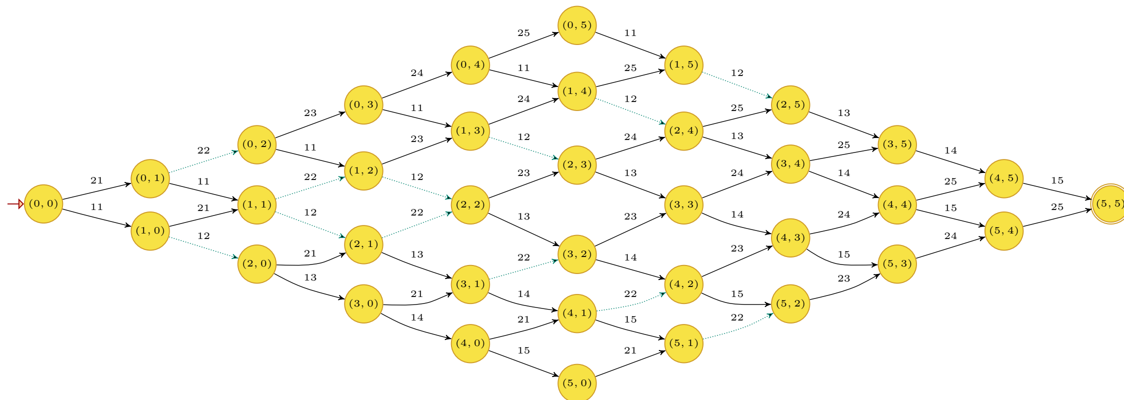


Figura 4.18: Modelo síncrono dos veículos viajando entre as estações A e B.

Para implementar a especificação de controle de tráfego na ferrovia, basta remover do modelo do sistema todos os estados (e suas transições associadas) em que os veículo estejam viajando na mesma seção, ou seja, todos os estado (i, i) para $i = 1, ..., 4$. Essa operação encontra-se implementada pelo código 4.18, no qual um autômato H é criado como cópia do modelo G, e na sequência os estados "proibidos" são removidos. O resultado dessa operação encontra-se ilustrado pelo diagrama de transição da figura 4.19.

Código 4.18: Construção da especificação.

```
H = G.copy()
for xh in H.X:
    if (xh[0] != '0') and (xh[0] != '5') and (xh[0] == xh[1]):
        H = H.deletestate(xh)
```

A análise da controlabilidade e normalidade dessa malha fechada pode, então, ser realizada pela utilização das funções `iscontrollable` (seção 4.2.1) e `isnormal`

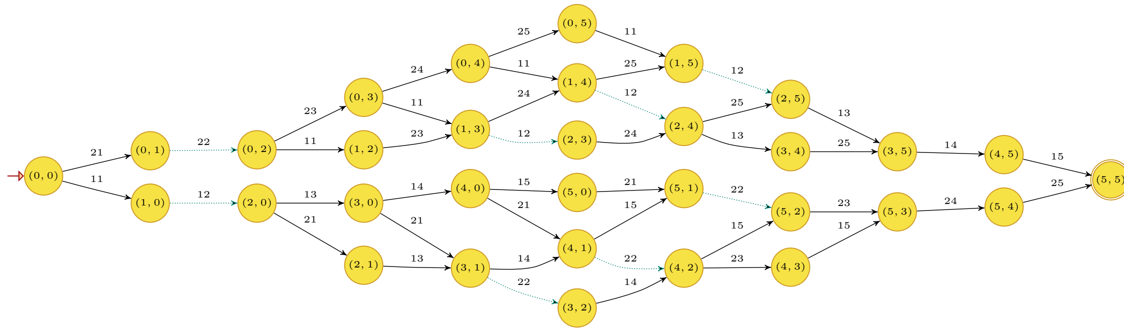


Figura 4.19: Especificação de segurança para o controle de tráfego de trens.

(seção 4.2.14). O código 4.19 chama estas funções, e obtém o seguinte resultado no terminal:

```
False
False
```

Código 4.19: Verificação da controlabilidade e normalidade.

```
1 print iscontrollable(H,G)
2 print isnormal(H,G)
```

A obtenção de uma especificação simultaneamente controlável e normal pode ser obtida por meio da função `supcontrollablenormal` (seção 4.2.20), conforme ilustra o código 4.20. Seu retorno é o autômato H_{supcn} que marca a linguagem controlável e normal suprema, cujo diagrama de transição encontra-se ilustrado na figura 4.20.

Código 4.20: Cálculo da linguagem controlável e normal suprema.

```
1 Hsupcn = supcontrollablenormal(H,G)
```

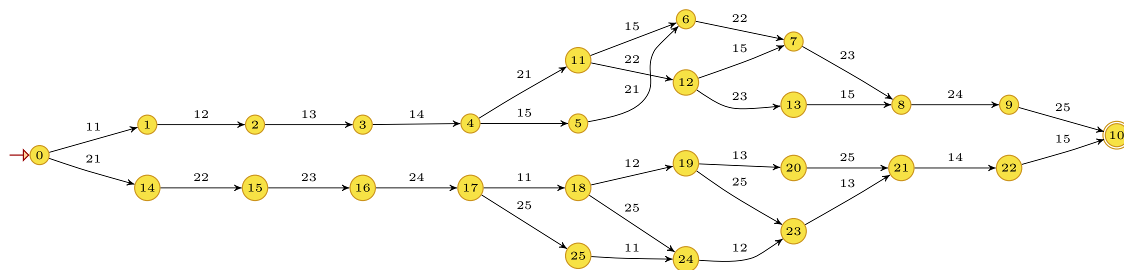


Figura 4.20: Especificação controlável e normal para o controle de tráfego de trens.

Por fim, o resultado pode ser analisado por meio da função `condat` (seção 4.2.5), conforme ilustra o código 4.21. Sua execução gera no terminal o seguinte retorno:

```
Control Data:
1:  21
2:  21
3:  21
```

```
13:  24
14:  11
15:  11
16:  11
20:  14
```

Para fins de análise do resultado, suponha que o veículo V_2 sai da estação antes de V_1, ou seja, no estado inicial 0 da especificação ocorre o evento 21, levando ao estado 14. Nota-se que, nesse caso, a ação de controle do supervisor desabilita o evento 11 nos estados 14, 15 e 16, ou seja, apenas quando veículo 2 entra na seção 4 (evento 24 levando ao estado 17) é que o veículo 1 pode sair da estação (evento 11). Porém, o veículo 1 só pode se deslocar até a seção 3 enquanto o veículo 2 não chegar a estação B. Isso é simbolizado pelo fato da chegada ao estado 21 depender de alguma ocorrência do evento 25.

Código 4.21: Análise das ações de controle.

```
1 condat(Hsupcn,G)
```

Capítulo 5

Conclusões e Trabalhos Futuros

Este trabalho trata principalmente do problema da verificação de duas propriedades do controle supervisório de sistemas a eventos discretos, que são condições necessárias para a existência de supervisores: a observabilidade e a normalidade. O estudo considera sistemas capazes de serem modelados por linguagens regulares, e o formalismo dos autômatos é utilizado tanto para a modelagem como para o tratamento dos sistemas a eventos discretos em análise. Ressalta-se, também, que o problema da observabilidade foi abordado tanto para uma arquitetura centralizada, como para uma arquitetura descentralizada, composta por um número indeterminado de agentes supervisores. Como um subproduto do estudo este trabalho, são também apresentadas em detalhes a implementação de uma biblioteca contendo as principais funções utilizados na teoria de controle supervisório. Esta biblioteca encontra-se implementada no programa DESLAB, e está disponível para utilização.

No que se refere ao problema da verificação das propriedades de observabilidade e normalidade, neste trabalho é proposto um novo verificador em tempo polinomial que possibilita a verificação simultânea de ambas as propriedades. A construção deste verificador possui complexidade computacional, pela notação $\mathcal{O}$, igual ao mais eficiente algoritmo existente na literatura para a verificação da observabilidade. Com respeito à normalidade, não é de conhecimento do autor que outro algoritmo de tempo polinomial tenha sido proposto. O melhor desempenho do verificador proposto neste trabalho se deve à restrição à procura por sequências que possam violar a condição de observabilidade somente.

Além disso, a generalização do algoritmo capaz de verificar a observabilidade permitiu a síntese de um novo algoritmo para a verificação da coobservabilidade. O algoritmo proposto possui complexidade computacional, pela notação $\mathcal{O}$, igual ao mais eficiente existente na literatura.

Em relação à biblioteca para controle supervisório desenvolvida como contribuição secundária deste trabalho, faz-se necessário ressaltar a importância que o programa DESLAB possuiu em todo seu desenvolvimento. As vantagens trazidas

pelo DESLAB na manipulação de autômatos foram fundamentais para o desenvolvimento eficiente dos algoritmos, eliminando qualquer risco de erro e perda de tempo envolvidos no tratamento manual de autômatos.

Por fim, durante o desenvolvimento deste estudo foram detectados alguns pontos que poderiam ser fontes de futuros trabalhos, dentre os quais destacam-se:

(i) Uma análise detalhada da complexidade computacional média dos algoritmos propostos neste trabalho, e dos existente na literatura. Isso pode ser feito por meio de uma análise estatística, usando autômatos gerados aleatoriamente, permitindo concluir sobre a real diferença de complexidade dos algoritmos, que aqui foi analisada somente pelo uso da complexidade de pior caso;

(ii) Realizar a ligação entre as propriedades de coobservabilidade e coobservabilidade a luz do funcionamento dos algoritmos propostos;

(iii) Extensão dos algoritmos aqui propostos para cenários de arquiteturas descentralizadas com outras estratégias de combinação das ações de controle, ou mesmo para observação dinâmica;

(iv) Desenvolvimento incremental da biblioteca para controle supervisório por meio da implementação de algoritmos já desenvolvidos, assim como de novos que venham a ser propostos;

(v) Realização de estudos de casos de aplicação da biblioteca em sistemas físicos existentes.

Referências Bibliográficas

[1] RAMADGE, P., WONHAM, W. M. "The Control of Discrete Event Systems", *Proceedings of the IEEE*, v. 77, n. 1, pp. 81–98, jan. 1989.

[2] KOSECKÁ, J. *A Framework for Modeling and Verifying Visually Giuded Agents: Design, Analysis and Experiments.* Tese de D.Sc., University of Pennsylvania, mar. 1996.

[3] WANG, X., LEE, P., RAY, A., et al. *Quantitative Measure for Discrete Event Supervisory Control.* New York, Springer, 2004.

[4] MOLINA, L. *Desenvolvimento de uma arquitetura de navegação deliberativa para robôs móveis utilizando a teoria de controle supervisório.* Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2010.

[5] WANG, Y., KELLY, T., LAFORTUNE, S. "Discrete control for safe execution of IT automation workflows". In: *EuroSys*, pp. 305–314, 2007.

[6] WANG, Y., MOTAHARI-NEZHAD, H., SINGHAL, S. "A Language-Based Framework For Analyzing Service Representation Models and Service Composition Approaches", *IEEE International Conference on e-Business Engineering*, 2010.

[7] PETERSON, J. L. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[8] DAVID, R., ALLA, H. *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems.* Prentice Hall, 1992.

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

[10] CIESLAK, R., DESCLAUX, C., FAWAZ, A. S., et al. "Supervisory Control of Discrete-Event Processes with Partial Observations", *IEEE Transactions on Automatic Control*, v. 33, n. 3, pp. 249–260, mar. 1988.

[11] TSITSIKLIS, J. N. "On the Control of Discrete-Event Dynamical Systems", *Mathematics of Control, Signals and Systems*, v. 2, pp. 95–107, 1989.

[12] WANG, W., GIRARD, A., LAFORTUNE, S., et al. "On Codiagnosability and Coobservability With Dynamic Observations", *IEEE Transactions on Automatic Control*, v. 56, n. 7, pp. 1551–1566, jul. 2011.

[13] 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, jul. 2011.

[14] CLAVIJO, L. B., BASILIO, J. C., CARVALHO, L. K. "DESLAB: A scientific computing program for analysis and synthesis of discrete-event systems", *Preprints of WODES 2012*, pp. 349–355, 2012.

[15] WONHAM, W. M. *Supervisory Control of Discrete-Event Systems*. jul. 2012. Disponível em: <http://www.control.utoronto.ca/cgi-bin/dldes.cgi>. Acesso em: 16 jul. 2013.

[16] IEEE. *IEEE Standards Dictionary: Glossary of Terms & Definitions*. IEEE, 1997.

[17] BASILIO, J. C., MOREIRA, M. V., CARVALHO, L. K. "Diagnose de falhas em sistemas a eventos discretos modelados por autômatos finitos", *Revista Controle & Automação*, v. 21, n. 5, pp. 510–533, set. 2010.

[18] LIN, F., WONHAM, W. M. "On observability of discrete-event systems", *Information Sciences*, v. 44, n. 3, pp. 173–198, 1988.

[19] ZIVIANI, N. *Projeto de algoritmos : com Implementação em Pascal e C*. 2 ed. São Paulo, Pioneira Thomson Learning, 2004.

[20] KUMAR, R., GARG, V., MARCUS, S. "On Controllability and Normality of Discrete Event", *Systems & Control Letters*, v. 17, n. 3, pp. 157–168, set. 1991.

[21] LIN, F., MORTAZAVIAN, H. "A Normality Theorem for Decentralized Control of Discrete-Event Systems", *IEEE Transactions on Automatic Control*, v. 39, n. 5, pp. 1089–1093, maio 1994.

[22] RUDIE, K., WILLEMS, J. C. "The Computational Complexity of Decentralized Discrete-Event Control Problems", *IEEE Transactions on Automatic Control*, v. 40, n. 7, pp. 1313–1318, jul. 1995.

[23] HUANG, Y., RUDIE, K., LIN., F. "Decentralized Control of Discrete-Event Systems When Supervisors Observe Particular Event Occurrences", *IEEE Transactions on Automatic Control*, v. 53, n. 1, pp. 384–388, fev. 2008.

[24] NETWORKX. "NetworkX - High-productivity software for complex networks". 2013. Disponível em: <`http://networkx.github.com/`>. Acesso em: 16 jul. 2013.

[25] GRAPHVIZ. "Graphviz - Graph Visualization Software". 2013. Disponível em: <`http://www.graphviz.org/`>. Acesso em: 16 jul. 2013.

[26] DESLAB. "DESlab - The scientific computing package for DES". 2012. Disponível em: <`http://www.dee.ufrj.br/lca/`>. Acesso em: 16 jul. 2013.

Apêndice A

Código fonte da biblioteca DESLAB para Controle Supervisório

A.1 IsControllable

```
def iscontrollable(H,G):
    controllable = True
    Sigma_uc = G.Sigma - H.Sigcon
    H0 = product(H,G)
    for (y,x) in H0.X:
        G_events_uc =  G.Gamma(x) & Sigma_uc
        if not H0.Gamma((y,x)) >= G_events_uc:
            controllable = False
    return controllable
```

A.2 SupControllable

```
def supcontrollable(H,G,rename_states=True):
    if ispfclosed(H):
        print 'Using Prefix Closed Algorithm'
        Hsupc = supcontrollablepfclosed(H,G)
    else:
        print 'Using General Algorithm'
        Hsupc =  supcontrollablegeneral(H,G)

    if rename_states and Hsupc <> fsa():
        Hsupc = Hsupc.renamestates('number')

    Hsupc= Hsupc.setpar(name ='$H^{{\uparrow}C}$')

    return Hsupc
```

A.3 SupControllablePfClosed

```
def supcontrollablepfclosed(H,G):
    G = G.setpar(Xm = G.X)

    A = sigmakleeneclos(G.Sigma)
    Sigma_uc = G.Sigma - H.Sigcon
    A_uc = sigmakleeneclos(Sigma_uc)

    #Hsupc = H-(((G-H)/A_uc)*A)
    Hsupc = H-(langquotient(G-H,A_uc,G2kleenclos=True)*A)

    return Hsupc
```

A.4 SupControllableGeneral

```
def supcontrollablegeneral(H,G):
    Sigma_uc = G.Sigma - H.Sigcon
    H0 = H&G.setpar(Xm = G.X)
    H0_changed = True
    while (H0_changed & (H0 <> fsa())):
        H0_changed = False
        for (y,x) in H0.X:
            G_events_uc = G.Gamma(x) & Sigma_uc
            if not(G_events_uc <= H0.Gamma((y,x))):
                H0 = H0.deletestate((y,x))
                H0_changed = True
        H0 = trim(H0)

    return H0
```

A.5 ConDat

```
def condat(H,G):
    if not iscontrollable(H,G):
        raise invalidArgument, 'H must be controllable with respect to G'

    Prod = H & G.setpar(Xm = G.X)

    table = {}
    for (xh,xg) in Prod:
        if xh in table:
            table[xh] |= G.Gamma(xg) - H.Gamma(xh)
        else:
            table[xh] = G.Gamma(xg) - H.Gamma(xh)

    print "Control Data:\r"
    for xh in table:
        disable = ""
        for e in table[xh]:
            disable = disable + str(e)
        if disable <> "":
```

```
20            print str(xh) + ":     " + disable + "\r"
21    return table
```

A.6 IsObservable

```
1 def isobservable(H,G,method=''):
2     if method and (method in 'tsitsiklis'):
3         return isobservable_tsitsiklis(H,G)
4     elif method and (method in 'wang'):
5         return isobservable_wang(H,G)
6     elif method and (method in 'observer'):
7         return isobservable_observer(H,G)
8     else:
9         return isobservable_bb(H,G)
```

A.7 ObsVerifier

```
1 def obsverifier(H,G,method=''):
2     if method and (method in 'tsitsiklis'):
3         return obsverifier_tsitsiklis(H,G)
4     elif method and (method in 'wang'):
5         return obsverifier_wang(H,G)
6     elif method and (method in 'observer'):
7         return obsverifier_observer(H,G)
8     else:
9         return obsverifier_bb(H,G)
```

A.8 IsObservable_Observer

```
1 def isobservable_observer(H,G):
2     Hobs = obsverifier_observer(H,G)
3
4     for xobs in Hobs:
5         Enable = set()
6         Disable = set()
7         for e in H.Sigcon:
8             for xh in xobs:
9                 if e in H.Gamma(xh):
10                    Enable |= {e}
11                elif e in G.Gamma(xh):
12                    Disable |= {e}
13        if Enable & Disable <> EMPTYSET:
14            return False
15    return True
```

A.9 ObsVerifier_Observer

```
def obsverifier_observer(H,G):
    return observer(H).setpar(name ='$H_{obs}$')
```

A.10 IsObservable_Tsitsiklis

```
def isobservable_tsitsiklis(H,G):
    if trim(obsverifier_tsitsiklis(H,G,stop=True)) <> fsa():
        return False
    else:
        return True
```

A.11 ObsVerifier_Tsitsiklis

```
def obsverifier_tsitsiklis(H,G,stop = False):

    def createstateandtrans(X_new,E_new):
        if not(X_new in X):
            X.append(X_new)
            if X_new <> dead:
                X_new_string = '(' + str(X_new[0]) + ',' + str(X_new[1]) +
                                                 ',' + str(X_new[2]) + ')'
                Table.update({X_new: X_new_string})
            if X_new <> dead:
                S.append(X_new)

        if not(E_new in E):
            E.append(E_new)
        E_new_string = str(E_new[0]) + '.' + str(E_new[1]) + '.' + str(E_new[2])
        Table.update({E_new: E_new_string})

        T_new = (state,E_new,X_new)
        if not(T_new in T):
            T.append(T_new)

    Hm = H.copy()
    Gm = G.copy()

    dead = 'dead'
    eps = 'eps'

    X0 = (iter(Hm.X0).next(),iter(Hm.X0).next(),iter(Gm.X0).next())
    X = [X0]
    E = []
    T = []
    Table = {X0: '(' + str(X0[0]) + ',' + str(X0[1]) + ',' + str(X0[2]) + ')'}

    S = [X0]
    while S <> []:
        state = S.pop()

```

```
38          H1_Gamma = Hm.Gamma(state[0])
39          H2_Gamma = Hm.Gamma(state[1])
40          G_Gamma = Gm.Gamma(state[2])
41
42          active_events = H1_Gamma.union(H2_Gamma)
43
44          for event in active_events:
45              if event in Hm.Sigobs:
46                  if (event in H1_Gamma) and (event in H2_Gamma) and
47                                          (event in G_Gamma):
48                      X_new = (Hm.delta(state[0],event),Hm.delta(state[1],event),
49                                             Gm.delta(state[2],event))
50                      E_new = (event,event,event)
51                      createstateandtrans(X_new,E_new)
52                  elif (event in Gm.Sigcon) and (event in H1_Gamma) and
53                          not(event in H2_Gamma) and (event in G_Gamma):
54                      X_new = dead
55                      E_new = (event,eps,event)
56                      createstateandtrans(X_new,E_new)
57                      if stop:
58                          S = []
59                          break
60              else:
61                  if (event in H1_Gamma):
62                      X_new = (Hm.delta(state[0],event),state[1],state[2])
63                      E_new = (event,eps,eps)
64                      createstateandtrans(X_new,E_new)
65                  if (event in H2_Gamma) and (event in G_Gamma):
66                      X_new = (state[0],Hm.delta(state[1],event),Gm.delta(state[2],
67                                                                          event))
68                      E_new = (eps,event,event)
69                      createstateandtrans(X_new,E_new)
70                  if (event in Gm.Sigcon) and (event in H1_Gamma) and
71                                  not(event in H2_Gamma) and (event in G_Gamma):
72                      X_new = dead
73                      E_new = (event,eps,event)
74                      createstateandtrans(X_new,E_new)
75                      if stop:
76                          S = []
77                          break
78      if dead in X:
79          Xm = [dead]
80      else:
81          Xm = []
82
83      V = fsa(X,E,T,X0,Xm,table = Table, name='$ObsTest$')
84      V.setgraphic('observer')
85
86      return V
```

A.12 IsObservable_Wang

```
1 def isobservable_wang(H,G):
2     from deslab.toolboxes import diagnosis
3 
4     Ht, sf = obstodiag(H,G)
5 
6     isdiag = True
7     if not(sf == set()):
8         isdiag, Vdiag = diagnosis.is_codiagnosable(Ht, sf, Ht.Sigobs)
9 
10    return isdiag
```

A.13 ObsVerifier_Wang

```
1 def obsverifier_wang(H,G):
2     from deslab.toolboxes import diagnosis
3 
4     Ht, sf = obstodiag(H,G)
5 
6     Vdiag = fsa()
7     if not(sf == set()):
8         isdiag, Vdiag = diagnosis.is_codiagnosable(Ht, sf, Ht.Sigobs)
9         Vdiag.setgraphic('observer')
10        Vdiag = Vdiag.setpar(name ='$V_{diag}$')
11 
12    return Vdiag
```

A.14 ObsToDiag

```
1 def obstodiag(H,G):
2     sf = set()
3     su = set()
4 
5     Ht = H.copy()
6 
7     for event in Ht.Sigcon:
8             de = 'd_{' + event + '}'
9             ve = 'v_{' + event + '}'
10            Ht = Ht.addstate(de)
11            Ht = Ht.addtransition([de,ve,de])
12 
13    for state in H:
14        for event in H.Sigcon:
15            if event in G.Gamma(state):
16                if event in H.Gamma(state):
17                    ue = 'u_{' + event + '}'
18                    de = 'd_{' + event + '}'
19                    su |= {ue}
20                    Ht = Ht.addtransition([state,ue,de])
21                else:
22                    fe = 'f_{' + event + '}'
```

```
23                          re = 'r_{' + event + '}'
24                          de = 'd_{' + event + '}'
25                          Ht = Ht.addtransition([state,fe,de])
26                          Ht = Ht.addtransition([state,re,de])
27                          sf |= {fe}
28      z = 'z'
29      for state in H:
30          if G.Gamma(state) == EMPTYSET:
31              Ht = Ht.addtransition([state,z,state])
32
33      Ht = Ht.setpar(Sigobs = Ht.Sigobs - sf)
34      Ht = Ht.setpar(Sigobs = Ht.Sigobs - su)
35      Ht = Ht.setpar(Sigcon = H.Sigcon)
36      Ht = Ht.setpar(name ='$\\tilde{H}$')
37
38      return Ht, sf
```

A.15 IsObservable_BB

```
1  def isobservable_bb(H,G):
2
3      def verifyobscondition(V):
4
5          def isrenamedactive(state,event):
6
7              def getrenamed(event):
8                  s = str(event)
9                  if '_' in event:
10                     ind = s.index('_') + 1
11                     eventStr = s[:ind] + '{' + s[ind:] + '_R}'
12                 else:
13                     eventStr = s + '_R'
14                 return eventStr
15
16             if getrenamed(event) in V.Gamma(state):
17                 return True
18             return False
19
20         dump = 'D'
21         if V <> fsa():
22             for state in V.X:
23                 if dump == state:
24                     for edge in V.Graph.in_edges_iter(state,keys=True):
25                         if (edge[0] <> dump) and (edge[2] in V.Sigcon):
26                             if (edge[2] in V.Sigobs):
27                                 return False
28                             elif isrenamedactive(edge[0],edge[2]):
29                                 return False
30         return True
31
32     return verifyobscondition(obsverifier_bb(H,G))
```

A.16 ObsVerifier_BB

```
1 def obsverifier_bb(H,G):
2
3     def renamenonobsevents(auto):
4         for event in (auto.Sigma - auto.Sigobs):
5             s = str(event)
6             if '_' in s:
7                 ind = s.index('_') + 1
8                 eventStr = s[:ind] + '{' + s[ind:] + '_R}'
9             else:
10                eventStr = s + '_R'
11            auto = auto.renamevents({ event : eventStr})
12        return auto
13
14    def createdumpfrommarked(auto):
15        for state in auto.Xm:
16            auto = auto.renamestates({ state : dump})
17        return auto
18
19    dump = 'D'
20
21    Gm = G.setpar(Xm = G.X)
22    Gm = Gm.setpar(name ='$G$')
23
24    Hm = H.setpar(Xm = H.X)
25    Hm = Hm.setpar(name ='$H_m$')
26
27    Hr = Hm.copy()
28    Hr = Hr.setpar(name ='$H_R$')
29    Hr = renamenonobsevents(Hr)
30
31    Hc = trim(Gm-Hm)
32    Hc = Hc.setpar(name ='$H_C$')
33    Hcd = createdumpfrommarked(Hc)
34    Hcd = Hcd.setpar(name ='$H_C^D$')
35
36    Vrc = Hr//Hcd
37    Vrc = Vrc.setpar(name ='$V_{RC}$')
38    V = createdumpfrommarked(Vrc)
39    V = V.setpar(name ='$V$')
40    V.setgraphic('observer')
41
42    return V
```

A.17 IsCoobservable

```
1 def iscoobservable(H,G,Ec,Eo,method=''):
2     if method and (method in 'wang'):
3         return iscoobservable_wang(H,G,Ec,Eo)
4     else:
5         return iscoobservable_bb(H,G,Ec,Eo)
```

A.18 CoobsVerifier

```
def coobsverifier(H,G,Ec,Eo,method=''):
    if method and (method in 'wang'):
        return coobsverifier_wang(H,G,Ec,Eo)
    else:
        return coobsverifier_bb(H,G,Ec,Eo)
```

A.19 IsCoobservable_Wang

```
def iscoobservable_wang(H,G,Ec,Eo):
    from deslab.toolboxes import diagnosis

    Ht, sf, Sigobs_list = coobstodiag(H,G,Ec,Eo)

    isdiag = True
    if not(sf == set()):
        isdiag, Vdiag = diagnosis.is_codiagnosable(Ht, sf, Sigobs_list)

    return isdiag
```

A.20 CoobsVerifier_Wang

```
def coobsverifier_wang(H,G,Ec,Eo):
    from deslab.toolboxes import diagnosis

    Ht, sf, Sigobs_list = coobstodiag(H,G,Ec,Eo)

    Vdiag = fsa()
    if not(sf == set()):
        isdiag, Vdiag = diagnosis.is_codiagnosable(Ht, sf, Sigobs_list)
        Vdiag.setgraphic('observer')
        Vdiag = Vdiag.setpar(name ='$V_{diag}$')

    return Vdiag
```

A.21 CoobsToDiag

```
def coobstodiag(H,G,Ec,Eo):
    if isinstance(Ec, set):
        Ec = list(Ec)
    elif isinstance(Ec, list):
        Ec = Ec
    else:
        raise invalidArgument, 'Ec must be list or set'

    if isinstance(Eo, set):
        Eo = list(Eo)
    elif isinstance(Ec, list):
```

```
12            Eo = Eo
13        else:
14            raise invalidArgument, 'Ec must be list or set'
15
16        if len(Ec) != len(Eo):
17            raise invalidArgument, 'Length of Ec and Eo must match'
18
19        sf = set()
20        su = set()
21
22        SigconTotal = set()
23        Sigobs_list = list()
24        for i in range(0, len(Ec)):
25            SigconTotal |= set(Ec[i])
26            Sigobs_list.append(set(Eo[i]))
27
28        Ht = H.copy()
29
30        for event in SigconTotal:
31                de = 'd_{' + event + '}'
32                ve = 'v_{' + event + '}'
33                Ht = Ht.addstate(de)
34                Ht = Ht.addtransition([de,ve,de])
35                for i in range(0, len(Ec)):
36                    Sigobs_list[i] |= {ve}
37
38        for state in H:
39            for event in SigconTotal:
40                if event in G.Gamma(state):
41                    if event in H.Gamma(state):
42                        ue = 'u_{' + event + '}'
43                        de = 'd_{' + event + '}'
44                        su |= {ue}
45                        Ht = Ht.addtransition([state,ue,de])
46                    else:
47                        fe = 'f_{' + event + '}'
48                        re = 'r_{' + event + '}'
49                        de = 'd_{' + event + '}'
50                        Ht = Ht.addtransition([state,fe,de])
51                        Ht = Ht.addtransition([state,re,de])
52                        sf |= {fe}
53                        for i in range(0, len(Ec)):
54                            if event in Ec[i]:
55                                Sigobs_list[i] |= {re}
56        z = 'z'
57        for state in H:
58            if G.Gamma(state) == EMPTYSET:
59                Ht = Ht.addtransition([state,z,state])
60                i = 0
61                for i in range(0, len(Ec)):
62                    Sigobs_list[i] |= {z}
63
64        Ht = Ht.setpar(Sigobs = Ht.Sigobs - sf)
65        Ht = Ht.setpar(Sigobs = Ht.Sigobs - su)
66        Ht = Ht.setpar(Sigcon = SigconTotal)
67        Ht = Ht.setpar(name ='$\\tilde{H}$')
68
69        return Ht, sf, Sigobs_list
```

A.22 IsCoobservable_BB

```
1 def iscoobservable_bb(H,G,Ec,Eo):
2
3     def verifycoobscondition(V):
4
5         def iscontrollable(event):
6             for Eci in Ec:
7                 if event in Eci:
8                     return True
9             return False
10
11        def isobsbyany(edge):
12
13            def isrenamedactive(state,event,i):
14
15                def getrenamed(event,i):
16                    s = str(event)
17                    if '_' in event:
18                        ind = s.index('_') + 1
19                        eventStr = s[:ind] + '{' + s[ind:] + '_{R_' +
20                                                        str(i) + '}}'
21                    else:
22                        eventStr = s + '_{R_' + str(i) + '}'
23                    return eventStr
24
25                if getrenamed(event,i) in V.Gamma(state):
26                    return True
27                return False
28
29            def coobscondition(edge,H,Eci,Eoi,i):
30
31                def getstatei(state,i):
32                    def find(s, ch):
33                        return [i for i, ltr in enumerate(s) if ltr == ch]
34
35                    indexes = find(state,"'")
36                    if i == 1:
37                        return state[indexes[0]+1:indexes[1]]
38                    else:
39                        return state[indexes[2*i-2]+1:indexes[2*i-1]]
40
41                if (edge[2] in Eci):
42                    if (edge[2] in H.Gamma(getstatei(str(edge[0][0]),i))):
43                        if (edge[2] in Eoi):
44                            return False
45                        elif isrenamedactive(edge[0],edge[2],i):
46                            return False
47                else:
48                    return False
49                return True
50
51            for i in range(0, len(Ec)):
52                if (coobscondition(edge,H,Ec[i],Eo[i],i+1) == True):
53                    return True
54            return False
55
56        dump = 'D'
```

```
57          if V <> fsa():
58              for state in V.X:
59                  if dump == state:
60                      for edge in V.Graph.in_edges_iter(state,keys=True):
61                          if (edge[0] <> dump) and iscontrollable(edge[2]):
62                              if (isobsbyany(edge) == False):
63                                  return False
64          return True
65
66      return verifycoobscondition(coobsverifier_bb(H,G,Ec,Eo))
```

A.23 CoobsVerifier_BB

```
1  def coobsverifier_bb(H,G,Ec,Eo):
2
3      def renamenonobsevents(auto,i):
4          for event in (auto.Sigma - auto.Sigobs):
5              s = str(event)
6              if '_' in s:
7                  ind = s.index('_') + 1
8                  eventStr = s[:ind] + '{' + s[ind:] + '_{R_' + str(i) + '}}'
9              else:
10                 eventStr = s + '_{R_' + str(i) + '}'
11             auto = auto.renamevents({ event : eventStr})
12         return auto
13
14     def createdumpfrommarked(auto):
15         for state in auto.Xm:
16             auto = auto.renamestates({ state : dump})
17         return auto
18
19     if isinstance(Ec, set):
20         Ec = list(Ec)
21     elif isinstance(Ec, list):
22         Ec = Ec
23     else:
24         raise invalidArgument, 'Ec must be list or set'
25
26     if isinstance(Eo, set):
27         Eo = list(Eo)
28     elif isinstance(Ec, list):
29         Eo = Eo
30     else:
31         raise invalidArgument, 'Ec must be list or set'
32
33     if len(Ec) != len(Eo):
34         raise invalidArgument, 'Length of Ec and Eo must match'
35
36     dump = 'D'
37
38     Gm = G.setpar(Xm = G.X)
39     Gm = Gm.setpar(name ='$G$')
40
41     Hm = H.copy()
42     Hm = Hm.setpar(Xm = Hm.X)
43     Hm = Hm.setpar(Sigobs = Hm.Sigma)
```

```
44      Hm = Hm.setpar(name ='$H_m$')
45
46      Hrlist = list()
47      Hrtotal = fsa()
48      for i in range(0, len(Ec)):
49          Hr = Hm.copy()
50          Hr = Hr.setpar(Sigcon = Ec[i])
51          Hr = Hr.setpar(Sigobs = Eo[i])
52          Hr = Hr.setpar(name ='$H_{R,' + str(i) +'}$')
53          Hr = renamenonobsevents(Hr,i+1)
54          Hrlist.append(Hr)
55          if i > 0:
56              Hrtotal = Hrtotal // Hr
57          else:
58              Hrtotal = Hr
59
60      Hrtotal = Hrtotal.setpar(name ='$H_{R,total}$')
61
62      Hc = trim(Gm-Hm)
63      Hc = Hc.setpar(name ='$H_C$')
64      Hcd = createdumpfrommarked(Hc)
65      Hcd = Hcd.setpar(name ='$H_C^D$')
66
67      Vrc = Hrtotal//Hcd
68      Vrc = Vrc.setpar(name ='$V_{RC}$')
69      V = createdumpfrommarked(Vrc)
70      V = V.setpar(name ='$V$')
71      V.setgraphic('observer')
72
73      return V
```

A.24 IsNormal

```
1 def isnormal(H,G,method=''):
2     if method and (method in 'observer'):
3         return isnormal_observer(H,G)
4     else:
5         return isnormal_bb(H,G)
```

A.25 NormalityVerifier

```
1 def normalityverifier(H,G,method=''):
2     if method and (method in 'observer'):
3         return normalityverifier_observer(H,G)
4     else:
5         return normalityverifier_bb(H,G)
```

A.26 IsNormal_Observer

```
def isnormal_observer(H,G):

    def verifynormalitycondition(V):
        if trim(V) == fsa():
            return True
        return False

    return verifynormalitycondition(normalityverifier_observer(H,G))
```

A.27 NormalityVerifier_Observer

```
def normalityverifier_observer(H,G):
    Gm = G.setpar(Xm = G.X)
    Hm = H.setpar(Xm = H.X)

    Hobs = observer(Hm)

    E_uo_clos = sigmakleeneclos(Hm.Sigma - Hm.Sigobs)
    Hobs_sl = Hobs // E_uo_clos

    Hint = Hobs_sl & Gm

    Vnorm = complement(Hm) & Hint
    Vnorm.setgraphic('observer')
    Vnorm = Vnorm.setpar(name ='$V_{norm}$')

    return Vnorm
```

A.28 IsNormal_BB

```
def isnormal_bb(H,G):

    def verifynormalitycondition(V):
        if V.Xm <> EMPTYSET:
            return False
        return True

    return verifynormalitycondition(normalityverifier_bb(H,G))
```

A.29 NormalityVerifier_BB

```
def normalityverifier_bb(H,G):
    return(obsverifier_bb(H,G))
```

A.30 SupNormal

```
def supnormal(H,G,rename_states=True):
    if ispfclosed(H):
        print 'Using Prefix Closed Algorithm'
        Hsupn = supnormalpfclosed(H,G)
    else:
        print 'Using General Algorithm'
        Hsupn =  supnormalgeneral(H,G)

    if rename_states and Hsupn <> fsa():
        Hsupn = Hsupn.renamestates('number')
    Hsupn = Hsupn.setpar(name ='$H^{{\uparrow}N}$')

    return Hsupn
```

A.31 SupNormalPfClosed

```
def supnormalpfclosed(H,G):
    G = G.setpar(Xm = G.X)

    #Hsupn = H-((invproj(proj(G-H,G.Sigobs),G.Sigma))*E_clos)
    Hc = G-H
    Hcobs = observer(Hc, Sigma_o = H.Sigobs)

    A_uo = sigmakleeneclos(H.Sigma - H.Sigobs)
    Hcobs_sl = Hcobs // A_uo

    A = sigmakleeneclos(G.Sigma)
    Hconcat = Hcobs_sl*A

    Hsupn = H-Hconcat

    return Hsupn
```

A.32 SupNormalGeneral

```
def supnormalgeneral(H,G):
    Hinext = H.copy()
    Hi = fsa()

    while not are_langequiv(Hi,Hinext) and Hinext <> fsa():
        Hi = Hinext
        Him = Hi.setpar(Xm = Hi.X)
        Hsupn = supnormalpfclosed(Him,G)
        Hinext = Hsupn & H

    return trim(Hinext)
```

A.33 SupControllableNormal

```
def supcontrollablenormal(H,G,rename_states=True):
    Hsupcn = H.copy()
    Hprev = fsa()

    while not are_langequiv(Hsupcn,Hprev):
        Hprev = Hsupcn
        Hsupc = supcontrollable(Hsupcn,G,False)
        Hsupcn = supnormal(Hsupc,G,False)

    if rename_states and Hsupcn <> fsa():
        Hsupcn = Hsupcn.renamestates('number')
    Hsupcn = Hsupcn.setpar(name ='$H^{{\uparrow}CN}$')

    return Hsupcn
```