DESLab Tutorial

Ney Rafael Guindane da Silva Barbosa

Rio de Janeiro
March 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This manual contains information on how to install and understand the functions in DESLab. The GitHub repository is available at `https://github.com/Neyrgsb/DESLab.git` and has the necessary files to use DESLab.

The functions presented in this manual were taken from three undergraduate theses. Table 1.1 indicates the sources of these functions.

Table 1.1: Thesis and their contributions.

Thesis	**Functions**
Lahis Coutinho [2]	Basic instructions [1]; Operations on Languages; Unary and Composition Operations; Additional Instructions of DESLab.
Daniel Garcia [3]	Fault Diagnosis Toolbox; Supervisory Control Toolbox.
Ney Barbosa [4]	Opacity Verifier Toolbox; Opacity Enforcement Toolbox; Time-Interval Automaton Operations Toolbox; Opacity Verifier Toolbox for TIA; Fault Diagnosis Toolbox for TIA.

For any assistance, please contact us via email at lca@poli.ufrj.br.

[1] There are two extra basic instruction functions presented in [4]: *mtable* and *isitemptymarked*.

Chapter 2

Setup

This chapter presents the steps to correctly setup DESLab in a new machine.

2.1 LaTeX

DESLab works with any TeX distribution: MiKTeX, MiKTeX or any other.
It is recommended to use MiKTeX, available for download at https://miktex.org/download. Run the installer and click *Next* until it is installed.

Figure 2.1: MiKTeX installer.

2.1.1 Add to Path

It is necessary to add the MiKTeX, or any other distribution installed, to Windows environment variables, through the following steps.

1. Open the search bar, search for MiKTeX and click at "Open File Location".

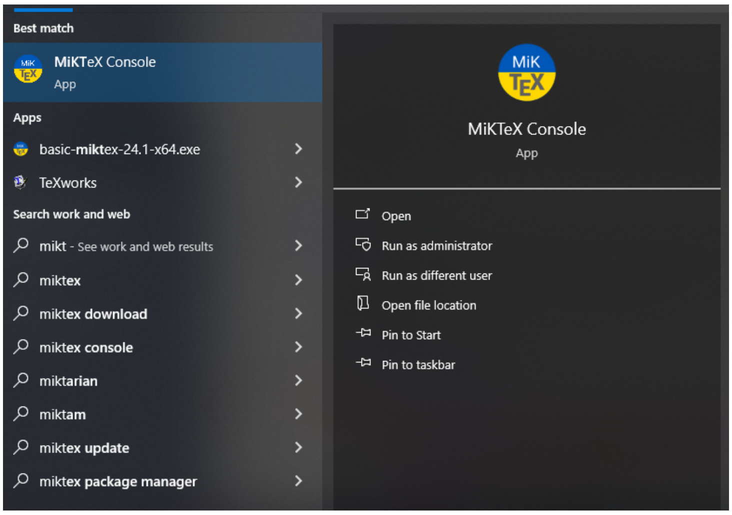


Figure 2.2: MiKTeX in the search bar.

2. Right click at MiKTex Console and click at “Open File Location”.

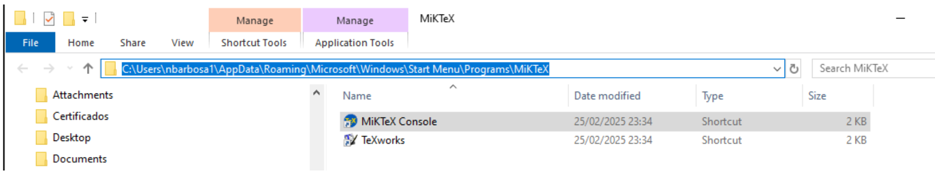


Figure 2.3: MiKTeX Console shortcut location.

3. Copy the path shown at “File Explorer”

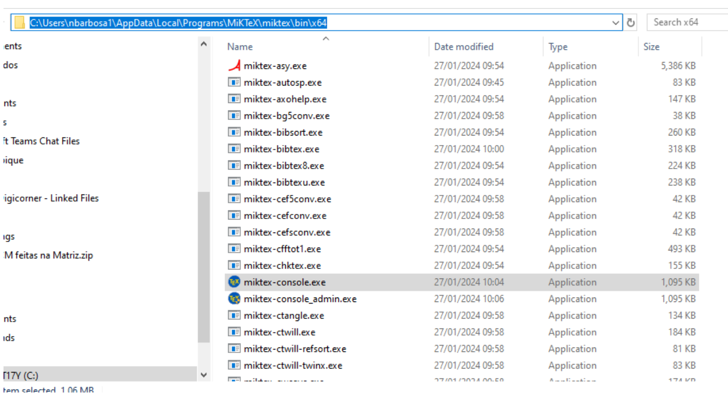


Figure 2.4: MiKTeX location.

4. Open the search bar and search for "system environment variables" and click at "Edit the system environment variables".

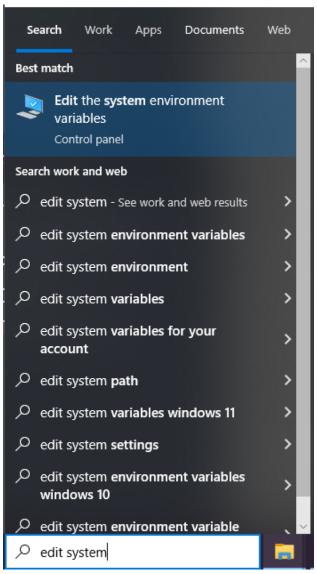


Figure 2.5: Edit the system environment variables in search bar.

5. Click at "Environment Variables..."

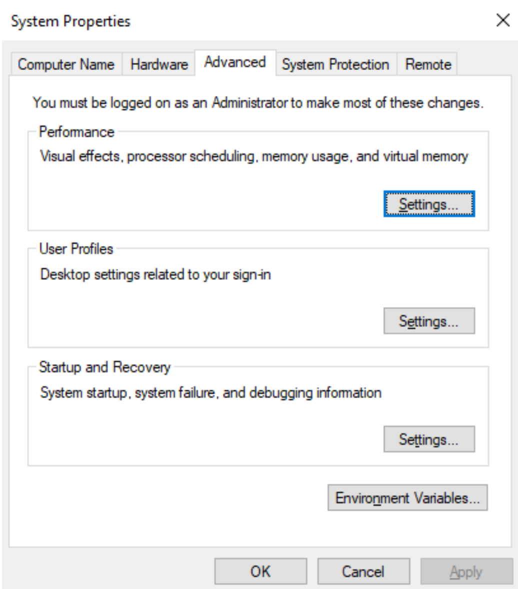


Figure 2.6: System Properties.

6. Click at "Path" and "Edit..."

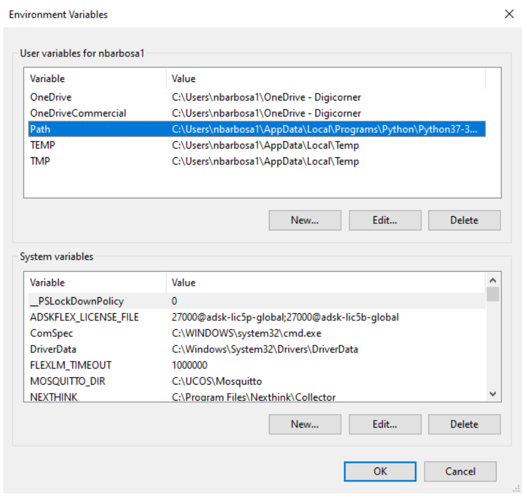


Figure 2.7: Environment Variables.

7. Verify if the MiKTeX path exists and if it is correct. If not, paste the copied path in the previous steps.

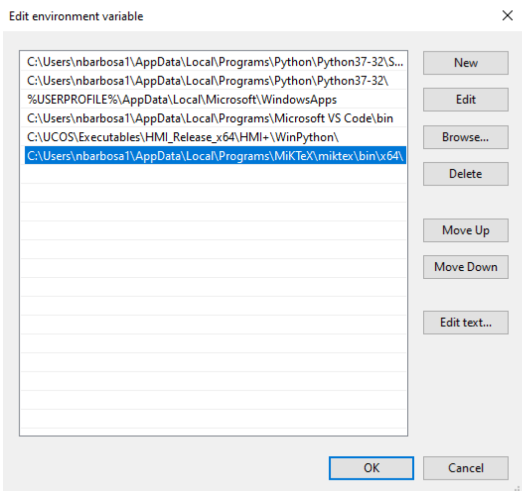


Figure 2.8: Edit Environment Variables.

2.2 Python

Install Python 3.12, available at "programas" folder or online. Make sure to check the option "Add Python 3.12 to PATH" in the Python installation window and click "Install Now".

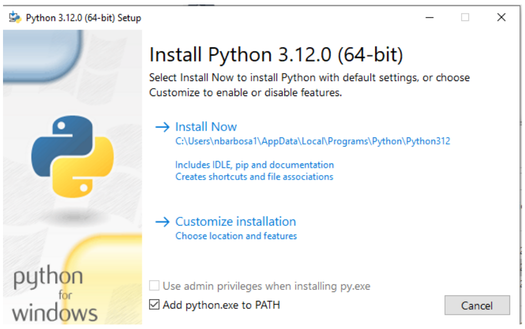


Figure 2.9: Python setup.

After installing Python 3.12, verify if the path is correctly added to Windows Environment Variables by following the steps on previous section.

2.3 DESLab

Install DESLab by running the file “Install.bat” at DESLab folder. It starts with Graphviz 2.28 installation, click “Next” until it is installed. Then, it takes around 10-15 minutes to download all dependencies that are needed. Please note that it could take longer, depending on the computer.

It is recommend to verify if Graphviz was added to Windows Environment Variables, you can check that by following the steps at LaTeX section.

2.4 Testing DESLab

To perform a quick verification, follow the steps:

1. Open “IDLE (Python 3.12 64-bit)”.
2. Click at “File” and “New File”.
3. Paste the following code, save and run the program.

```
from deslab import *
syms('q1 q2 q3 a1 b1 e f')
table = [(a1,'a_1'),(b1,'b_1'),(q1,'q_1'),(q2,'q_2'),(q3,'q_3')]
X = [q1,q2,q3]
Sigma = [a1,b1,e]
X0 = [q1]
Xm = [q3]
T =[(q1,b1,q2),(q2,b1,q3),(q3,e,q3)]
G1 = fsa(X,Sigma,T,X0,Xm,table,name='$G_1$')
draw(G1)
```

4. It prompts the option to choose the program to open the pdf file. It is highly recommended to use Google Chrome as the default program.

5. The verification is successful once the drawing of automaton shown in Figure 2.10 appears on your chosen pdf reader.

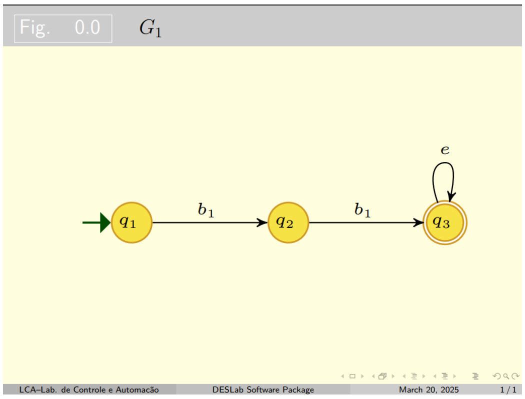


Figure 2.10: Figure generated by the verification code.

Chapter 3

Instructions

This chapter presents all functions implemented at DESLab with an example of their use, respectively.

3.1 Basic Instructions of DESLab

3.1.1 Defining a Finite State Automaton - *fsa*

- Purpose
  This instruction defines an automaton object into DESLab .

- Syntax

$$G = fsa(X, E, T, X0, Xm, options)$$

  - Inputs:
    In order of appearance, the input parameters are:

    X - List of symbols corresponding to the states;

    E - List of symbols corresponding to the events;

    T - List of symbols ordered as tuples corresponding to the transitions such as (X, Σ, X);

    X0[1] - List of symbols corresponding to the initial states;

    Xm - List of symbols corresponding to the marked states;

[1]Deterministic finite state automata have only one initial state. In this case, the correct notation for this parameter is x_0.

– Options:

table - List of association tuples between symbols and latex labels;
Sigobs - List of symbols corresponding to the observable events;
Sigcon - List of symbols corresponding to the controllable events;
name - A string representing the name of the automaton to be defined;

All the inputs may be plugged into the command as lists, as in

G = fsa([list of states], [list of events], [list of transition tuples], [list of initial states], [list of marked states]).

Or they might have their attributed variables, such as

$$\begin{aligned} \text{X} &= \text{[list of states];} \\ \text{E} &= \text{[list of events];} \\ \text{T} &= \text{[list of transition tuples];} \\ \text{X0} &= \text{[list of initial states];} \\ \text{Xm} &= \text{[list of marked states].} \\ G &= fsa(X, E, T, X0, Xm) \end{aligned}$$

However, it demands a little more caution whenever the "options" are used. When all of them (*table, Sigobs, Sigcon, name*) are defined, then the user *must* insut them in the exact order displayed above. When is desired not to use all the options available, for instance whenever it is interested only on either the controllable or observable events, the first list of events will be taken as observable ones. If the goal is to define only controllable events, then the best way to do so is to define Sigcon = [list of controllable events] and plug it straight into the command as follows (in the command it has been chosen to use only the *table* and *Sigcon* options).

G = fsa(X, E, Γ*, X0, Xm, table, Sigcon = [list of controllable events]).*

– Output
The output is the definition of the given automaton into DESLab . Upon *print G* command, the output would be the number of events, states and transitions, along with its classification and name. In order to access the parameters of G such as X or Σ, the

Table 3.1: Syntax for accessing mathematical properties of an automaton object.

Parameter	Symbol	Syntax
List of states	X	G.X
List of events	E	G.E
List of transition tuples	T	G.transitions()
List of initial states	X0	G.X0
List of marked states	Xm	G.Xm

user must print the commands listed in Table 4.1.

- Description
  Let the six-tuple $G = (X, \Sigma, f, \Gamma, x_{0,}, X_m)$ denote a deterministic automaton, where:
  - X is the finite set of states;
  - Σ is the finite set of events associated with G;
  - $f : X \times \Sigma \rightarrow \Sigma$ is the transition function: *f(x,e)=y* meaning that there is an event e that labels the transition from state x to state y;
  - $\Gamma : X \rightarrow 2^{\Sigma}$ is the active event function (or feasible event function): $\Gamma(x)$ is the set of all events e for which *f(x,e)* is defined and it is called the active event set (or feasible event set) of G at x;
  - x_0 is the initial state;
  - $X_m \subseteq X$ is the set of marked states;

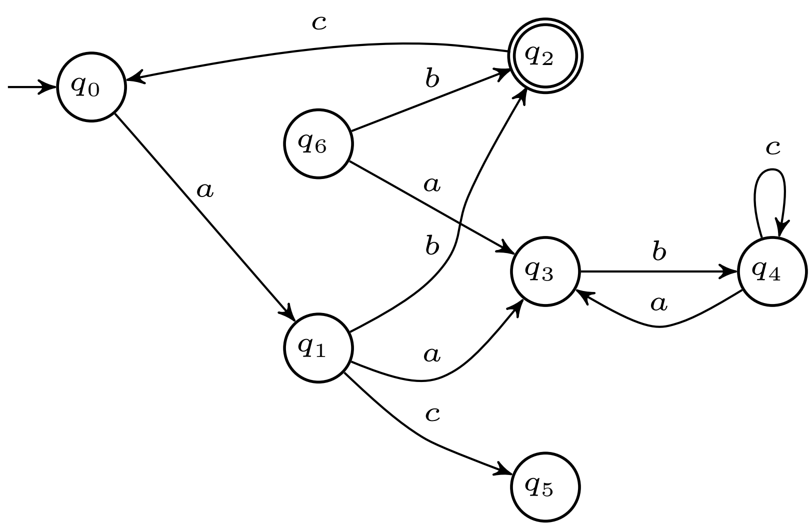
  Note that there is a difference between the formal definition of a deterministic automaton and the syntax used to define it into DESLab. Instead of having the transition function f and the active event (or feasible event) function Γ, the syntax brings a parameter T standing for a description, ordered as tuples, of transitions between states in a form: (x_1, e, x_2), $\{x_1, x_2\} \subset X \wedge e \in \Sigma$.
  Another possibility is to define a finite state automaton from one that has been already defined, that is, once G_1 is known, then an attribution $G_2 = G_1$ is valid.

- Example

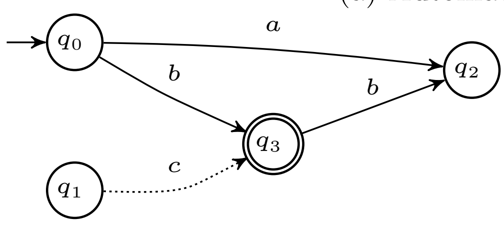
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.1(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$, shown in Figure 3.1(b) where $X_2 = \{q_0, q_1, q_2, q_3\}$, $\Sigma_2 = \{a, b, c\}$, $f_2(q_0, b) = \{q_3\}$, $f_2(q_0, a) = \{q_2\}$, $f_2(q_1, c) = \{q_3\}$, $f_2(q_3, b) = \{q_2\}$, $X_{0,2} = \{q_0\}$, $X_{m,2} = \{q_3\}$. Automata G_1 (shown in Figure 3.1(a)) and G_2 (shown in Figure 3.1(b)) are defined into DESLab by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# automaton definition G2: defining table, Sigobs,
#                          Sigcon, name

# if put in order, it is up to the user whether using
# variables or lists.

X2 = [q0,q1,q2,q3]
Sigma2 = [a,b,c]
X02 = [q0]
Xm2 = [q3]
OBS = [a,b]
T2 =[(q0,b,q3), (q0,a,q2), (q1,c,q3), (q3,b,q2)]
G2=fsa(X2,Sigma2,T2,X02,Xm2,table,OBS,[b,c],name='$G_2$')
```

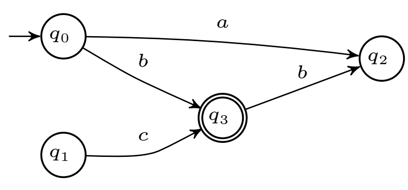
```
# for the same automaton definition, excepting the list of
observable events Sigobs, G3 can be ONLY written as:

G3=fsa(X2,Sigma2,T2,X02,Xm2,table,Sigcon=[b,c],name='$G_3$')

# NOTICE: ANY other way to define only controllable events
#         would lead to a misinterpretation, turning them
#         into OBSERVABLE events instead.


draw (G1, G2, G3, 'figure')
```



(a) Automaton G_1



(b) Automaton G_2



(c) Automaton G_3 .

Figure 3.1: Example of the definition of automata into DESLab.

- See also: Drawing a State Transition Diagram

3.1.2 Drawing a State Transition Diagram

- Purpose
  This operation returns the state transition diagram of a given automaton.

- Syntax

$$draw(G, mode, save\_file = False)$$

  - Inputs

    The input parameters are finite automata of the class fsa, the mode of exhibition, which can be chosen among:

    * Figure - produces a black and white state trasition diagram;

    * Figurecolor - produces a state transition diagram in colors;

    * Beamer - produces a slide page document from the beamer class of latex, provided with further information about DESLab.

    And save_file (optional): if set to True, saves the generated LaTeX file in the same directory where the script is executed.

  - Output
    The output are state transition diagrams according to the mode of exhibition. If save_file = True, the corresponding LaTeX file is also saved.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The graphic representation of G_1 is called the *state transition diagram*, where circles stand for the states and arrows labeled by symbols represent the transitions. The initial state is signalized by a small arrow pointing to it; the marked states are double-circled and non-observable events labeling transitions produce a dotted arrow. Inspecting the state transition diagram of an automaton makes detecting generated and marked languages, as well as blocked paths and other particularities to be pretty straightforward.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$, where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$,

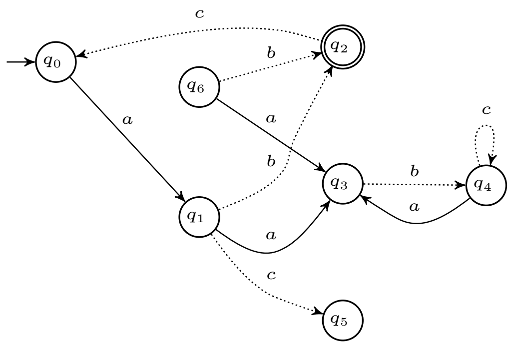
$f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Assume that it is required:

1. the state transition diagram of G_1 in black and white;
2. the state transition diagram G_1 in colors;
3. the state transition diagram of G_1 in beamer format.

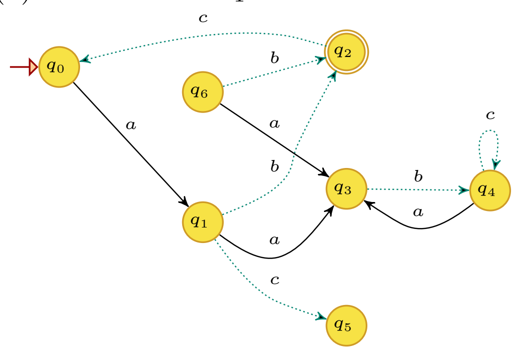
Using DESLab , we can obtain automaton G_1 (shown in Figures 3.2(a), (b) and (c)) displayed respectively in all modes of exhibition required by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# G1 in black and white

draw (G1, 'figure')

# G1 in colors

draw(G1, 'figurecolor')

# G1 in beamer format

draw(G1, 'beamer')

# NOTICE that it is possible to generate state transition
```

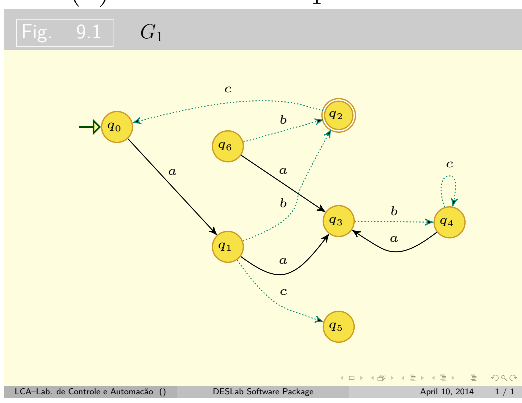
```
#diagrams by implying commands inside draw, such as
#draw(G1.addstate(x),'figurecolor').
```



(a) Automaton G_1 in black and white



(b) Automaton G_1 in colors.



(c) Automaton G_1 in beamer format.

Figure 3.2: Example of state transition diagrams.

- See also: Graph Algorithms

3.1.3 Adding States

- Purpose
  This operation adds a state to the set of sates of a finite state automaton.

- Syntax

$$G_1 = G_1.addstate(x)$$

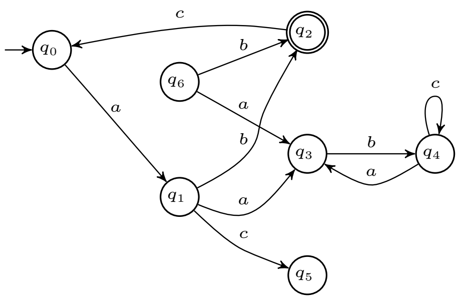
  - Inputs
    The input parameter is the state to be added.
  - Output
    The output is the input automaton with the new state added to its set of states.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Adding a state to G_1 produces an enlargement to its set of states. Note that adding a state has nothing to do with adding a transition associated to it.

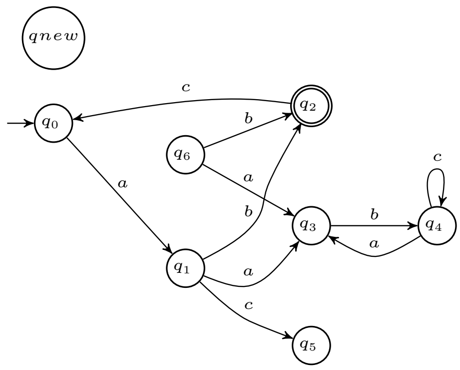
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.3(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to add a new state q_{new} to $X_{0,1}$. Using DESLab we can obtain automaton $G_1 = G_1.addstate$ (shown in Figure 3.3(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
```

```
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# add state q_new

G1=G1.addstate(qnew)

draw (G1, 'figure')
```



(a) Automaton G_1



(b) Automaton G_1 with a state q_{new} added to it.

Figure 3.3: Example of adding a state to G_1.

- See also: Deleting States, Renaming States

3.1.4 Deleting States

- Purpose
  This operation deletes a state from the set of states of a finite state automaton.

- Syntax

$$G_1 = G_1.deletestate(x)$$

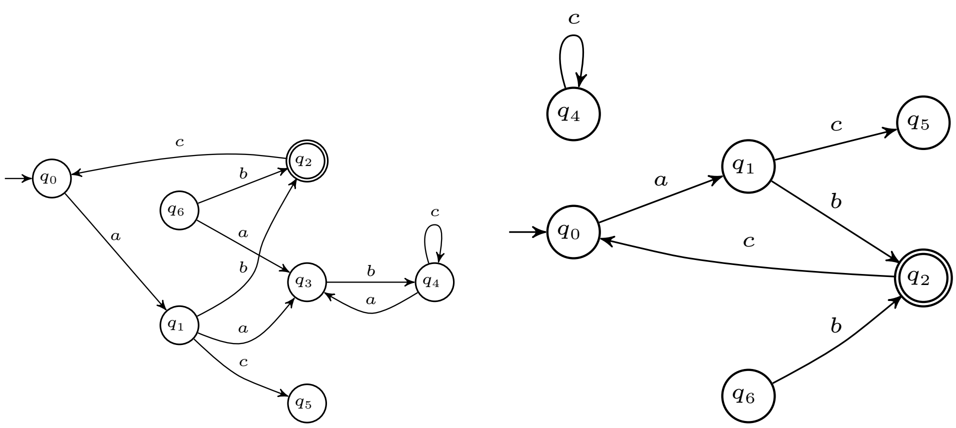
  - Inputs
    The input parameter is the state to be deleted.
  - Output
    The output is the previous automaton without the given state and all the transitions associated to it.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Deleting a state from G_1 not only reduces the set of states X_1 but also reflects on its transition function, since the deleted state takes along all the transitions associated to it.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.4(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to delete the state q_3 from $X_{0,1}$. Using DESLab we can obtain automaton $G_1 = G_1.deletestate$ (shown in Figure 3.4(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
```

```
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# delete state 'q_3'

G1=G1.deletestate(q3)

draw (G1, 'figure')
```



(a) Automaton G_1 (b) Automaton G_1 without the state q_3.

Figure 3.4: Example of deleting a state from G_1.

- See also: Adding States, Deleting Transitions, Number of Transitions, Getting a List of Transitions

3.1.5 Adding Events

- Purpose
  This operation adds events to the set of events of a finite state automaton.

- Syntax

$$G_1 = G_1.addevent(e)$$

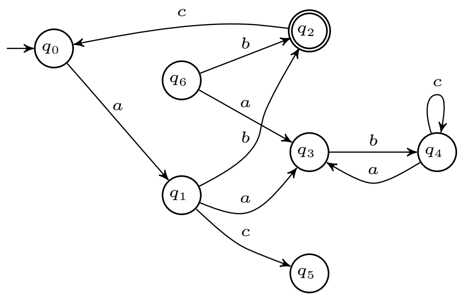
  - Inputs
    The input parameter is the list of events to be added.
  - Output
    The output is the addition of the list of events e to the set of events.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Adding an event to G_1 produces an enlargement to its set of events.

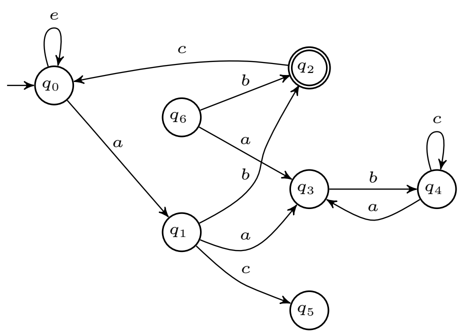
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.5(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to add the event e to Σ_1. Since the addition of an event can only be seen if this event is actually labelling a transition between states, let the new event e occur as in $f_1(q_0, e) = \{q_0\}$. Using DESLab we can obtain automaton $G_1 = G_1.addevent$ (shown in Figure 3.5(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c e')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
```

```
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# add event 'e'

G1=G1.addevent(e)

# add selfloop 'f(q0,e)=q0'

G1=G1.addtransition([q0,e,q0])

draw (G1, 'figure')
```



(a) Automaton G_1



(b) Automaton G_1 with a new event e labelling the new selfloop $q_0 \xrightarrow{e} q_0$.

Figure 3.5: Example of adding an event to G_1.

- See also: Deleting Events, Adding Selfloops, Adding Transitions

3.1.6 Deleting Events

- Purpose
  This operation deletes events from the set of events of a finite state automaton.

- Syntax

$$G_1 = G_1.deletevent(e)$$

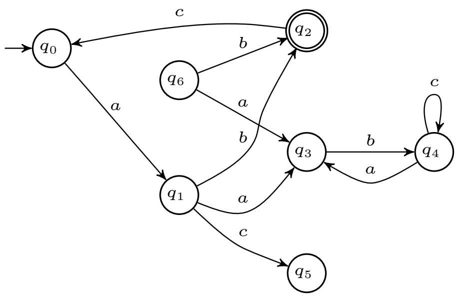
  - Inputs
    The input parameter is the list of events to be deleted.
  - Output
    The output is the input automaton without the events and all the transitions labelled by them.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Deleting an event from G_1 reduces not only the set of events, but also the transition function of the automaton, since the deleted event takes with it all the transitions that it used to label.

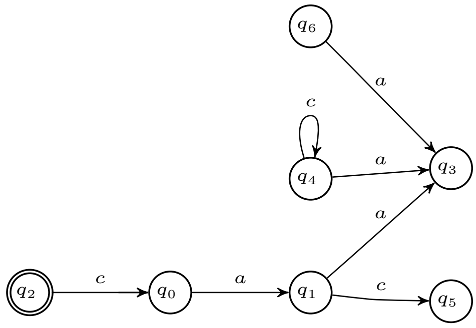
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.6(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to delete the event b from Σ_1. Using DESLab we can obtain automaton $G_1 = G_1.deletevent$ (shown in Figure 3.6(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
```

```
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# delete event 'b'

G1=G1.deletevent(b)

draw (G1, 'figure')
```



(a) Automaton G_1



(b) Automaton G_1 without the event b.

Figure 3.6: Example of deleting an event from G_1.

- See also: Adding Events

3.1.7 Adding Transitions

- Purpose
  This operation adds a transition to a finite state automaton.

- Syntax

$$G_1 = G_1.addtransition([x, a, y])$$

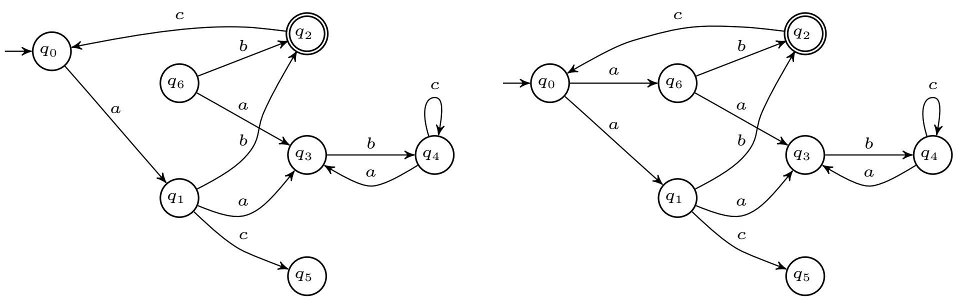
  - Inputs
    The input parameter is the transition to be added to the transition function, given by the states and the event that labels it.
  - Output
    The output is the input automaton with a new transition from state x to state y through the occurrence of the event a.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The addition of the transition from the state x to state y labelled by an event "a" produces an enlargement on the transition function of G_1.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.7(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to add a transition from state q_0 to state q_6, labelled by event "a", is desired. Using DESLab we can obtain automaton $G_1 = G_1.addtransition$ (shown in Figure 3.7(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
```

```
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# add transition f(q0,a,q6)

G1=G1.addtransition([q0,a,q6])

draw (G1, 'figure')
```



(a) Automaton G_1 (b) Automaton G_1 with the new transition $q_0 \xrightarrow{a} q_6$

Figure 3.7: Example of adding a transition to G_1.

- See also: Deleting Transitions, Renaming Transitions

3.1.8 Deleting Transitions

- Purpose
  This operation deletes a transition from a finite state automaton.

- Syntax

$$G_1 = G_1.deletetransition([x, a, y])$$

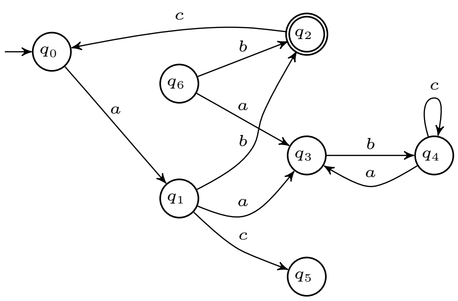
  - Inputs
    The input parameter is the transition to be deleted from the transition function, given by the states and the event that labels it.
  - Output
    The output is the previous automaton without the transition from state x to state y labelled by the event a.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Deleting a transition from G_1 reduces the transition function f_1.

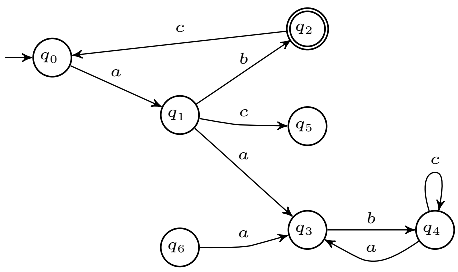
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.8(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to delete the transition from state q_6 to state q_2, labelled by the event b. Using DESLab we can obtain automaton $G_1 = G_1.deletetransition$ (shown in Figure 3.8(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
```

```
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# delete transition f(q6,b,q2)

G1=G1.deletetransition([q6,b,q2])

draw (G1, 'figure')
```



(a) Automaton G_1



(b) Automaton G_1 without the transition $q_6 \xrightarrow{b} q_2$

Figure 3.8: Example of deleting a transition from G_1.

- See also: Adding Transitions, Adding Selfloops, Renaming Transitions

3.1.9 Adding Selfloops

- Purpose
  This operation adds a selfloop to the transition function of the automaton.

- Syntax

$$G_1 = G_1.addselfloop(x, a)$$

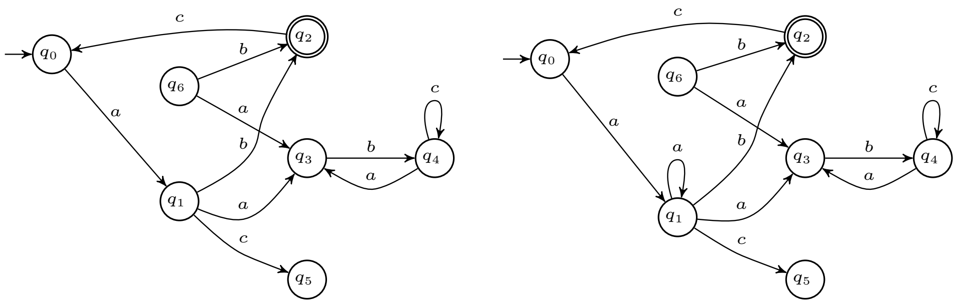
  – Inputs
    The input parameters are the state and the event that characterizes the selfloop to be added.
  – Output
    The output is the addition of a selfloop to the transition function.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Adding a selfloop x $\xrightarrow{a}$ x to an automaton produces an enlargement to its transition function caused by the new transition from the state x to itself, labelled by the event a.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.9(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to add the selfloop $q_1 \xrightarrow{a} q_1$. Using DESLab we can obtain automaton $G_1 = G_1.addselfloop(q1, a)$ (shown in Figure 3.9(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
```

```
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# add selfloop f('q_1',a)='q_1'

G1=G1.addselfloop(q1,a)

draw (G1, 'figure')
```



(a) Automaton G_1

(b) Automaton G_1 with the new transition $q_1 \xrightarrow{a} q_1$.

Figure 3.9: Example of adding a selfloop to G_1.

- See also: Adding Transitions, Adding States, Deleting Transitions, Deleting States

3.1.10 Renaming States

- Purpose
  This operation renames states from a finite state automaton.

- Syntax

$$G_1 = G_1.renamestates(X, mapping)$$

  – Inputs
    The input parameter is a mapping listing, in tuples, the states to be renamed.

  – Output
    The output is the previous automaton with the states renamed according to the input mapping.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. In order to rename any states from it, there is no need to redefine the fsa. Instead, a mapping should be created relating new and old state labels in tuples, as follows:

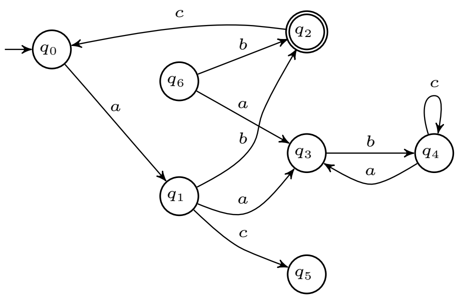
$$\text{mapping} = [(\text{old state label, 'new state label'})]$$

- Remark
  It is mandatory to define the new symbols to be used for labelling before running the command.

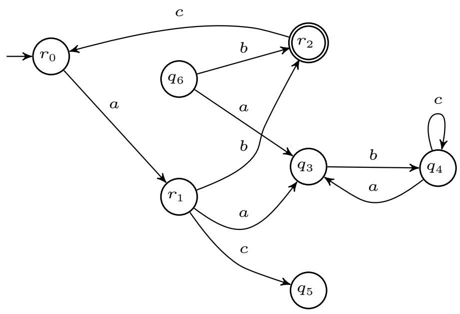
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.10(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. The states $\{q_0, q_1, q_2\}$ are to be renamed as $\{r_0, r_1, r_2\}$. Using DESLab we can obtain automaton $G_1 = G_1.renamestates$ (shown in Figure 3.10(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c r0 r1 r2')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'),
```

```
(q3,'q_3'),(q4,'q_4'), (q5,'q_5'), (q6,'q_6'),
(r0,'r_0'), (r1,'r_1'), (r2,'r_2')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# test 1: rename states without mapping

G2 = G1.renamestates([(q0,'r0'),(q1,'r1'),(q2,'r2')])

# test 2: rename states using mapping:

mapping = [(q0,'r0'),(q1,'r1'),(q2,'r2')]

G3 = G1.renamestates(mapping)

# test 3: rename states using inverse mapping:

mapping = [(q0,'r0'),(q1,'r1'),(q2,'r2')]
inverse = [(q,r) for (r,q) in mapping]

G4 = G3.renamestates(inverse)

# Note that G1 and G4 are the same, as well as
# G2 and G3

draw (G1, G2, G3, G4, 'figure')
```



(a) Automata G_1 and G_4



(b) Automata G_2 and G_3 with renamed states.

Figure 3.10: Example of renaming states of G_1.

- See also: Renaming Events, Adding States, Deleting States

3.1.11 Renaming Events

- Purpose
  This operation renames events from a finite state automaton.

- Syntax

$$G_1 = G_1.renamevents(E, mapping)$$

  - Inputs
    The input parameter is a mapping inserted by the user listing, in tuples, the events to be renamed.
  - Output
    The output is the previous automaton with the events renamed according to the input mapping.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. In order to rename any events pf G_1, there is no need to redefine the fsa. Instead, a mapping should be created relating new and old event labels in tuples, as follows:

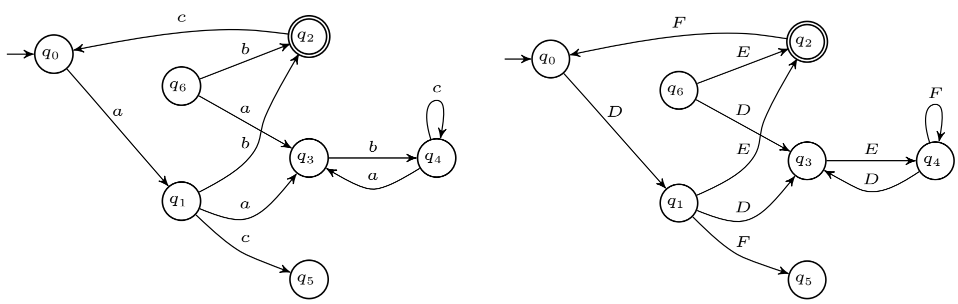
  mapping = [(old event label, 'new event label')]

- Remark
  It is mandatory to define the new symbols to be used for labelling before running the command.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.11(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. The events a, b and c are to be renamed by D, E and F. Using DESLab we can obtain automaton $G_1 = G_1.renamevents(mapping)$ (shown in Figure 3.11(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c D E F')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
```

```
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# rename events

mapping = [(a,'D'),(b,'E'),(c,'F')]
G1 = G1.renamevents(mapping)

draw (G1, 'figure')
```



(a) Automaton G_1 (b) Automaton G_1 with the new events

Figure 3.11: Example of renaming an event from G_1.

- See also: Adding Events, Deleting Events

3.1.12 Renaming Transitions

- Purpose
  This operation renames a transition in terms of the event that labels it.

- Syntax

$$G_1 = G_1.renametransition([x, (a, b), y])$$

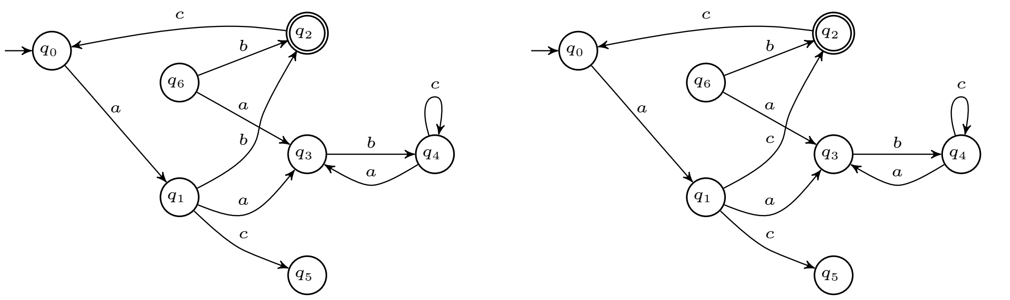
  - Inputs
    The input parameters are the states and events related to the transition to be renamed, where the old event comes first in the pair.
  - Output
    The output is the alteration of the event that labels the specified state transition.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Renaming a transition means altering the event that labels it.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.12(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to rename the transition $f_1(q_1, b) = \{q_2\}$ in such a way that the new event that labels it is the event c. Using DESLab we can obtain automaton $G_1 = G_1.renametransition([q_1, (b, c), q_2]))$ (shown in Figure 3.12(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
```

```
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# rename the transition f(q1,b)=q2, changing b into c.

G1=G1.renametransition([q1,(b,c),q2])
draw (G1, 'figure')
```



(a) Automaton G_1

(b) Automaton G_1 with the transition $q_1 \xrightarrow{b} q_2$ renamed into $q_1 \xrightarrow{c} q_2$.

Figure 3.12: Example of renaming a transition of G_1.

- See also: Number of Transitions, Getting a List of Transitions, Adding Transitions, Deleting Transitions

3.1.13 Redefining X_0, X_m, Σ_{con}, Σ_{obs}

- Purpose
  This operation redefines the set of initial and marked states, as well as the controllable and observable events.

- Syntax

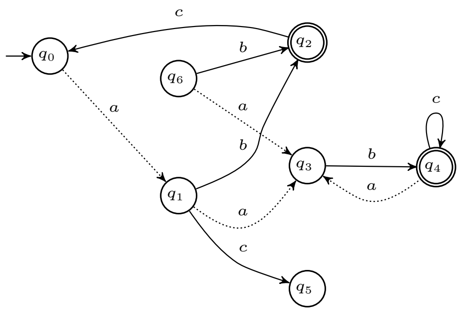
$$G_1 = G_1.setpar(property = value)$$

  - Inputs
    The input parameters are the sets of states or events to be redefined, followed by an equal sign and the contents to be inserted into it. However, the field '*property*' only accepts the terminology:
    * $X0$ and Xm for the initial and marked set of states;
    * $Sigcon$ and $Sigobs$ for the sets of controllable and observable events, respectively.
  - Output
    The output is the redefinition of the predetermined sets of states and events.

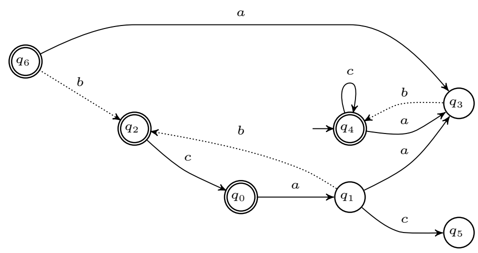
- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. After the machine is created, we may need to alter the sets of initial and marked states, as well as the controllable and observable event sets. These changes would cause a total rearrangement of the automaton, turning it into a new one. The ways of doing so include defining a variable associated to the sets to be redefined and plugging it into the command, or simply using a list straight into it.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1}, Sigcon, Sigobs, name)$ shown in Figure 3.13(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $f_1(q_7, c) = \{q_7\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2, q_4\}$, $Sigobs = \{q_1, q_3\}$, $Sigobs = \{q_5, q_7\}$, $name = G_1$. The sets of initial and marked states, as well as controllable and observable events are to be redefined. Using DESLab we can obtain automaton $G_1 = G_1.setpar()$ (shown in Figure 3.13(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 q7 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6'), (q7, 'q_7')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6, q7]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2,q4]
Sigcon = [a,b]
Sigobs = [b,c]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3),
 (q7,c,q7)].
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table, Sigobs, Sigcon)


# redefining sets of states and events using variables

con = [b,c]
marked = [q0,q2,q4,q6]

# redefining sets of states and events plugging the
# elements straight into the command

G2 = G1.setpar(X0=q4, Xm=marked, Sigcon=con, Sigobs=[a,c])

draw (G1, G2, 'figure')
```



(a) Automaton G_1



(b) Automaton G_2 with the new definitions.

Figure 3.13: Example of redefining X_0, X_m, Σ_{con}, Σ_{obs} from G_1

- See also: Defining a Finite State Automaton, Renaming Transitions, Renaming Events, Renaming States

3.1.14 Number of States

- Purpose
  This operation provides the number of states of an automaton.

- Syntax

$$len(G1)$$

  - Inputs
    The input parameter is an automaton of the class fsa.
  - Output
    The output is the number of states of the automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The number of states provides the length of the set of states X_1.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.14(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to determine the number of states of G_1 that can be obtained by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')
```

```
# return the number of states

print len(G1)
```

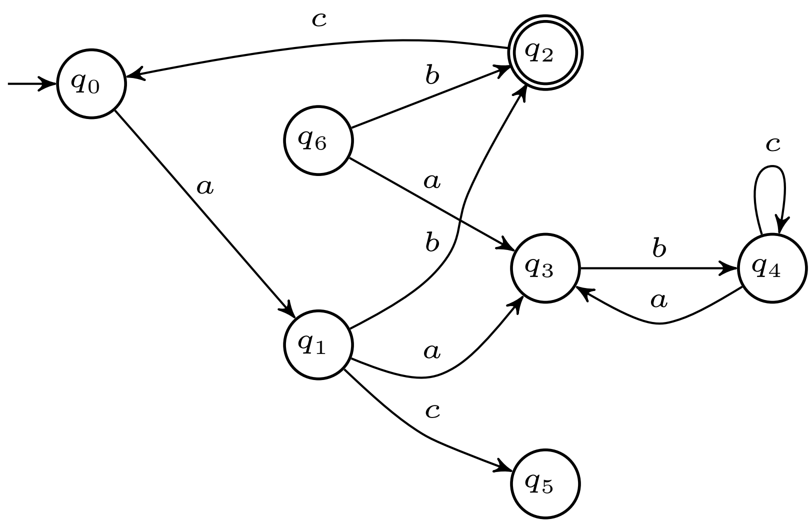


Figure 3.14: Automaton G_1 of the example of the number of states.

- Console outputs
  The response to the example, which should be plugged in the console, is:
  print len(G1)
  >>>7

- See also: Number of Transitions, List of Transitions

3.1.15 Number of Transitions

- Purpose
  This operation provides the number of transitions of an automaton.

- Syntax

$$size(G)$$

  - Inputs
    The input parameter is an automaton of the class fsa.
  - Output
    The output is the number of transitions of the input automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Accessing the number of transitions of G_1 provides the length of its transition function f_1.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.15(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want to determine the size of the transition function. Using DESLab we can obtain the number of transitions of G_1 by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
```

```
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# return the number of transitions

print size(G1)
```

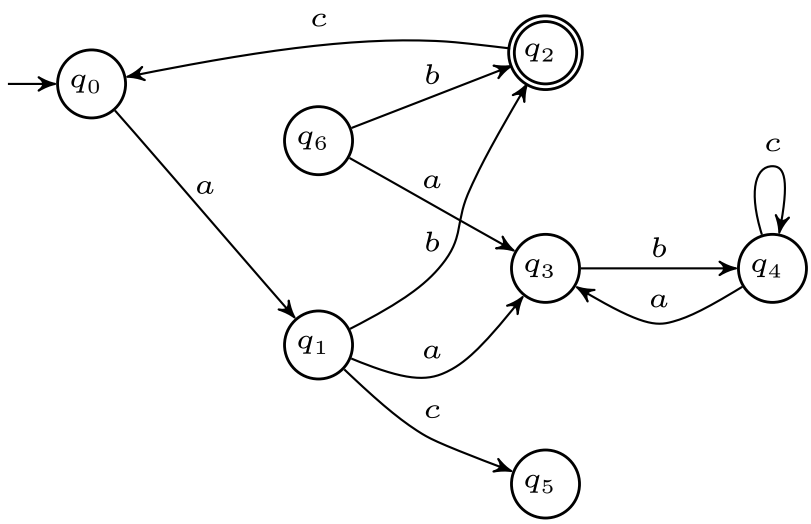


Figure 3.15: Automaton G_1 of the example of the number of transitions.

- Console outputs
  The response to the example, which should be plugged in the console, is:
  print size(G1)
  >>>10

- See also: Number of States, Getting a List of Transitions

3.1.16 Getting a List of Transitions

- Purpose
  This operation returns the list of transitions of a finite state automaton, as previously seen in Table 4.1.

- Syntax

$$transitions(G1)$$
$$G1.transitions()$$

  - Inputs
    The input parameter is an automaton of the class fsa.
  - Output
    The output is the list of transitions given by tuples ordered as (x_1,e,x_2), where $\{x_1, x_2\} \subset X_1$ and $e \in \Sigma_1$.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Its structure is defined by a set of transitions starting from the initial state. The list of transitions is therefore the entire set of transitions, ordered as tuples, in which the first element is the state where the transition comes from; the second element is the event labelling the transition and the third one is the state where the transitions goes to.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.16(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. We want the list of transitions of the given automaton.

  Using DESLab we can obtain the list of transitions by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1
X1 = [q0,q1,q2,q3,q4,q5,q6]
```

```
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# print the list of transitions
transitions(G1)
G1.transitions()
```

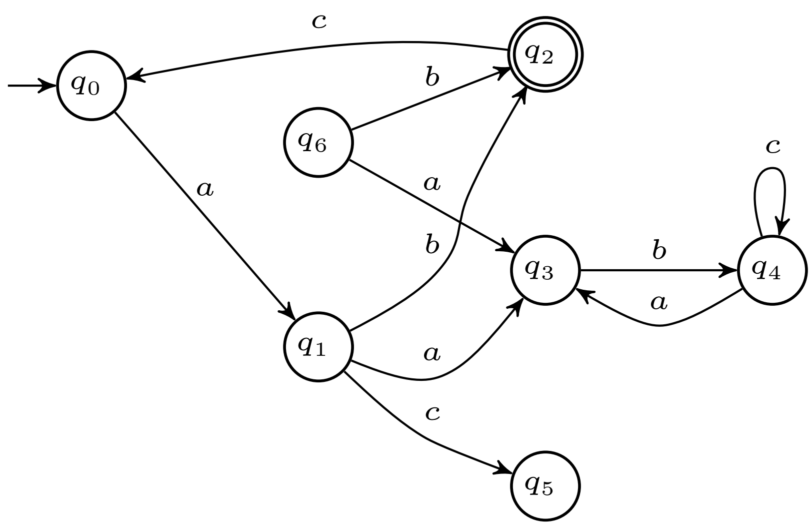


Figure 3.16: Automaton G_1 of the example of listing transitions.

- Console outputs
  The response to the example, which should be plugged in the console, is:

  transitions(G1)
  >>>[('q1', 'a', 'q3'), ('q1', 'b', 'q2'), ('q1', 'c', 'q5'), ('q0', 'a', 'q1'), ('q3', 'b', 'q4'), ('q2', 'c', 'q0'), ('q4', 'a', 'q3'), ('q4', 'c', 'q4'), ('q6', 'a', 'q3'), ('q6', 'b', 'q2')]

- See also: Number of Transitions, Renaming Transitions

3.1.17 Table of events $\times$ states

- Purpose
  Represent the transitions of an automaton in table format.

- Syntax

$$mtable(G, options)$$

  - Input: Automaton of the class fsa, list of states, list of events, creation of a csv, name of the csv. Except for the automaton, all other inputs are optional.
  - Options:
    states - List with the desired order for the states in the first column;
    events - List with the desired order for the events in the first row;
    gen_csv - Boolean variable for the creation of a csv file;
    csv_name - A string with the desired name for the csv file.
  - Output: A table that relates the previous states to the subsequent states is generated. Additionally, if requested, a CSV file containing this data is exported.

- Description
  Given the automaton $G = (X, \Sigma, f, \Gamma, X_0, X_m)$, the function *mtable* creates a table with the structure shown in Table 3.2, where $x_i, y_{ij} \in X$ and $\sigma_j \in \Sigma$, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. If $f(x_k, \sigma_t) = y$ is defined, then the element in the k^{th} row and t^{th} column of the table is $y_{kt} = y$. If the transition is not defined, then $y_{kt} = none$.

Table 3.2: States $\times$ Events.

	σ_1	σ_2	$\cdots$	σ_n
x_1	y_{11}	y_{12}	$\cdots$	y_{1n}
x_2	y_{21}	y_{22}	$\cdots$	y_{2n}
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
x_m	y_{m1}	y_{m2}	$\cdots$	y_{mn}

The order in which the states in the first column and the events in the first row appear is random. A specific order can be defined by passing the state and event parameters as input to the function.

To create a *csv* file from this table, you need to add a variable with the value True to the function input. If it is not given as a parameter in the function input, the default name of the generated file is "new.csv". It is possible to define it directly in the function input as "$csv\_name\_defined$".

- Example
  Consider the automaton $G = (X,\ \Sigma,\ f,\ X_0,\ X_m,\ \Sigma_o)$, shown in Figure 3.17, where $X = \{1, 2, 3, 4\}$, $\Sigma = \{a, b, c, d\}$, $f(0, a) = 1$, $f(1, b) = 2$, $f(1, d) = 3$, $f(2, c) = 2$, $f(2, d) = 4$, $f(3, b) = 4$, $f(4, b) = 4$, $X_0 = 0$, and $X_m = \emptyset$. It is possible, through the following commands, to call the function *mtable*, which generates the output shown in Table 3.3, as well as a csv file named "table".

```
from deslab import *

# automaton definition G
X = [0,1,2,3,4]
Sigma = [a,b,c,d]
X0 = [0]
Xm = [ ]
T =[(0,a,1),(1,b,2),(1,d,3),(2,c,2),(2,d,4), (3,b,4), (4,b,4)]
G = fsa(X,Sigma,T,X0,Xm, name='$G$')

#generate table
table = mtable(G, [0, 2], [a, b, c, d], True, 'table')

draw(G,'figure')
print(table)
```

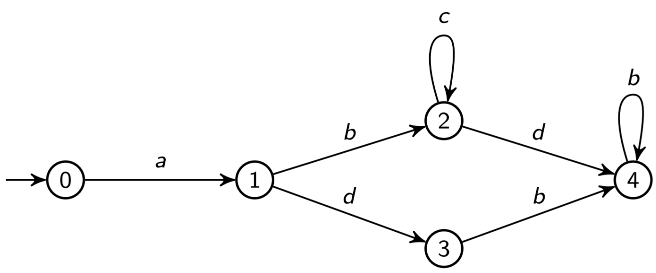


Figure 3.17: Automaton G of the example in subsection 3.1.17.

- Console outputs

Table 3.3: Table generated from the automaton in the Figure 3.17.

	a	b	c	d
0	1	None	None	None
2	None	None	2	4
1	None	2	None	3
3	None	4	None	None
4	None	4	None	None

3.2 Operations on Languages

3.2.1 Concatenation

- Purpose
  This operation returns the concatenation between the languages marked by automata.

- Syntax

$$
\begin{aligned}
G &= G_1 * G_2 \\
G &= concatenation(G_1, G_2) \\
G &= G_1 * G_2 * \cdots * G_n \\
G &= concatenation(G_1, G_2, \cdots, G_n)
\end{aligned}
$$

  - Inputs
    The input parameters are automata of the class fsa.
  - Output
    The output is an automaton that marks the concatenation of the languages marked by the input automata.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0,2}, X_{m_2})$ denote two finite state automata. The concatenation between G_1 and G_2 produces an automaton whose marked language is

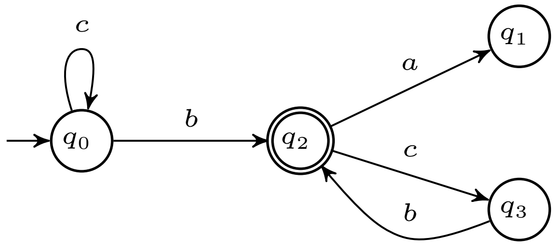
$$
\begin{aligned}
L_m(G) &= L_m(G_1)L_m(G_2) \\
&= \{s \in \Sigma^* : (s = s_1 s_2) \\
&\quad \wedge (s_1 \in L_m(G_1)) \wedge (s_2 \in L_m(G_2))\}
\end{aligned}
$$

  Connecting the marked states of G_1 with the initial state of G_2 by ε-transitions and then unmarking all the states of G_1 results in a non-deterministic automaton that marks exactly $L_m(G_1)L_m(G_2)$.

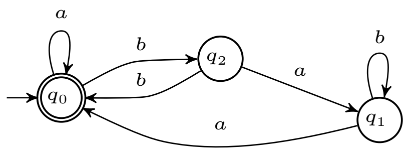
- Example
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figures 3.18(a) and 3.18(b) where $X_1 = \{q_0, q_1, q_2, q_3\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, c) = \{q_0\}$, $f_1(q_0, b) = \{q_2\}$, $f_1(q_2, a) = \{q_1\}$, $f_1(q_2, c) = \{q_3\}$, $f_1(q_3, b) = \{q_2\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $X_2 = \{q_0, q_1, q_2\}$, $\Sigma_2 = \{a, b\}$, $f_2(q_0, a) = \{q_0\}$, $f_2(q_0, b) = \{q_2\}$, $f_2(q_2, a) = \{q_1\}$, $f_2(q_2, b) = \{q_0\}$, $f_2(q_1, b) =$

$\{q_1\}$, $f_2(q_1, a) = \{q_0\}$, $X_{0,2} = \{q_0\}$, $X_{m,2} = \{q_0\}$. Using DESLab we can obtain automaton $G = G_1 * G_2$ (shown in figure 3.18(c)) by writing the following instructions. Note that DESLab will automatically rename the states of G.

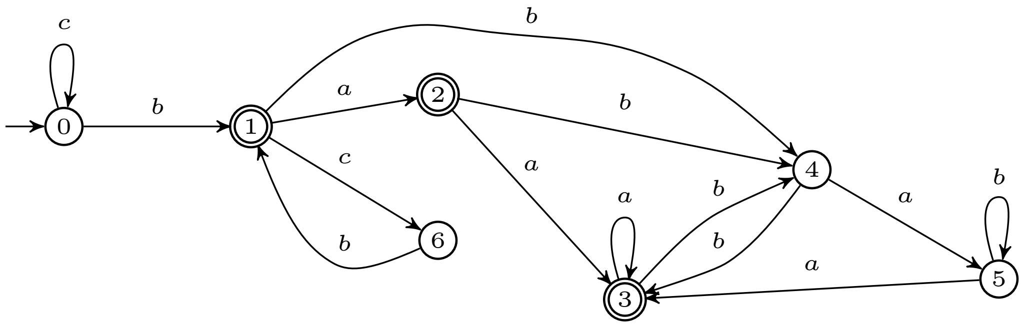
```
from  deslab import *
syms('q0 q1 q2 q3 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3')]

# automaton definition G1

X1 = [q0,q1,q2,q3]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,c,q0), (q0,b,q2), (q2,a,q1), (q2,c,q3), (q3,b,q2)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# automaton definition G2

X2 = [q0,q1,q2]
Sigma2 = [a,b]
X02 = [q0]
Xm2 = [q0]
T2 =[(q0,a,q0), (q0,b,q2), (q2,a,q1), (q2,b,q0), (q1,b,q1), (q1,a,q0)]
G2 = fsa(X2,Sigma2,T2,X02,Xm2,table,name='$G_2$')

# concatenation

G = G1*G2

# another possible notation

G = concatenation(G1,G2)
draw(G1, G2, G,'figure')
```



(a) Automaton G_1



(b) Automaton G_2



(c) Automaton $G = G_1 * G_2$

Figure 3.18: Example of the concatenation operation.

- See also: Union

3.2.2 Union

- Purpose
  This operation returns the union between the languages generated by automata.

- Syntax

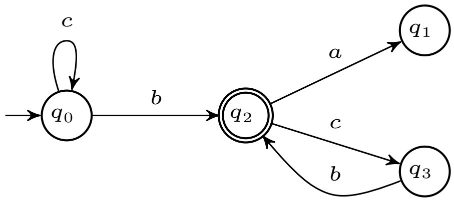
$$
\begin{aligned}
G &= G_1 + G_2 \\
G &= union(G_1, G_2) \\
G &= G_1 + G_2 + \cdots + G_n \\
G &= union(G_1, G_2, \cdots, G_n)
\end{aligned}
$$

  - Inputs
    The input parameters are automata of the class fsa.
  - Output
    The output is an automaton marked by the language resulted from the union of the input automata marked languages.

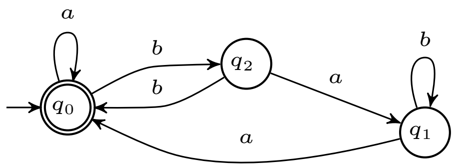
- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0,2}, X_{m_2})$ denote two finite state automata. The union between G_1 and G_2 produces an automaton whose generated language is $L_m(G) = L_m(G_1) \cup L_m(G_2)$.
  Being $L_m(G_1)$ and $L_m(G_2)$ regular languages, $L_m(G)$ can be obtained by creating a new initial state and connecting it to the initial states of G_1 and G_2 through ε-transitions. The result is a nondeterministic automaton marking the union between $L_m(G_1)$ and $L_m(G_2)$.

- Example
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 3.18(a) and (b) where $X_1 = \{q_0, q_1, q_2, q_3\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, c) = \{q_0\}$, $f_1(q_0, b) = \{q_2\}$, $f_1(q_2, a) = \{q_1\}$, $f_1(q_2, c) = \{q_3\}$, $f_1(q_3, b) = \{q_2\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $X_2 = \{q_0, q_1, q_2\}$, $\Sigma_2 = \{a, b\}$, $f_2(q_0, a) = \{q_0\}$, $f_2(q_0, b) = \{q_2\}$, $f_2(q_2, a) = \{q_1\}$, $f_2(q_2, b) = \{q_0\}$, $f_2(q_1, b) = \{q_1\}$, $f_2(q_1, a) = \{q_0\}$, $X_{0,2} = \{q_0\}$, $X_{m,2} = \{q_0\}$. Using DESLab we can obtain automaton $G = G_1 + G_2$ (shown in figure 3.19(c)) by writing the following instructions. Note that DESLab will automatically rename the states of G.

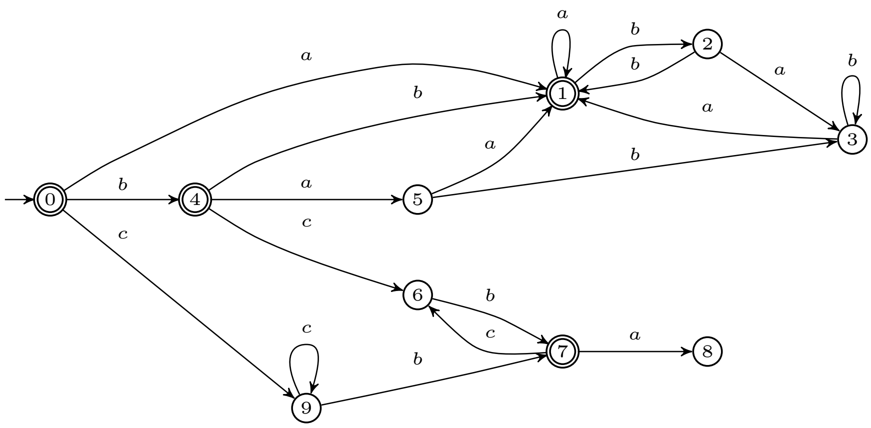
```
from  deslab import *
syms('q0 q1 q2 q3 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3')]

# automaton definition G1

X1 = [q0,q1,q2,q3]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,c,q0), (q0,b,q2), (q2,a,q1), (q2,c,q3), (q3,b,q2)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# automaton definition G2

X2 = [q0,q1,q2]
Sigma2 = [a,b]
X02 = [q0]
Xm2 = [q0]
T2 =[(q0,a,q0), (q0,b,q2), (q2,a,q1), (q2,b,q0), (q1,b,q1), (q1,a,q0)]
G2 = fsa(X2,Sigma2,T2,X02,Xm2,table,name='$G_2$')

# union

G = G1+G2

# another possible notation

G = union(G1,G2)

draw(G2,'figure')
```



(a) Automaton G_1



(b) Automaton G_2



(c) Automaton $G = G_1 + G_2$

Figure 3.19: Example of the union operation.

- See also: Concatenation

3.2.3 Prefix Closure

- Purpose
  This operation returns the prefixes of all the strings in the language marked by an automaton.

- Syntax

$$G = pclosure(G_1)$$

  - Inputs
    The input parameter is an automaton of the class fsa.
  - Output
    The output is an automaton marked by the language containing all the prefixes of all strings in the input automaton marked language.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The language generated by G_1 is L_1. The prefix closure of L_1 is

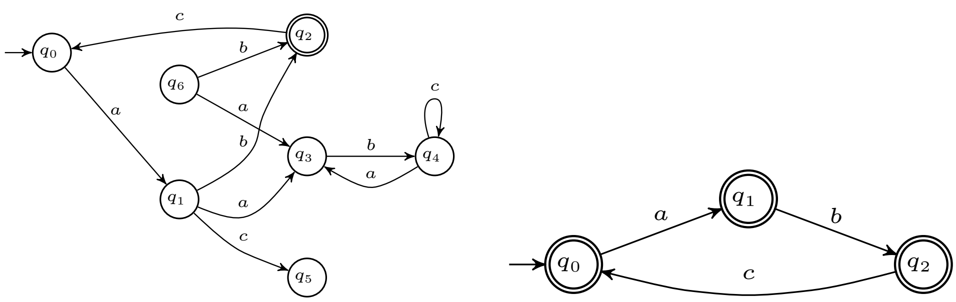
$$\overline{L_1} = \{s \in \Sigma^* : (\exists t \in \Sigma^*)[st \in L_1]\} \tag{3.1}$$

  The automaton marking $\overline{L_1}$ can be obtained by taking the trim of G_1 and then marking all of its states.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.20(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = pclosure(G_1)$ (shown in figure 3.20(b)) by writing the following instructions.

```
    from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
```

```
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# prefix closure

G = pclosure(G1)


draw(G,'figure')
```



(a) Automaton G_1 (b) Automaton $G = pclosure(G_1)$

Figure 3.20: Example of the prefix closure operation.

- See also: Kleene Closure, Kleene Closure Generator

3.2.4 Kleene Closure

- Purpose
  This operation returns the Kleene Closure of a language.

- Syntax

$$G = kleeneclos(G1)$$

  - Inputs
    The input parameter is an automaton of the class fsa.
  - Output
    The output is the Kleene-closure of the language marked by the input automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The language marked by G_1 is $L_{m,1}$. Let $L_{m,1} \subseteq \Sigma^*$, then

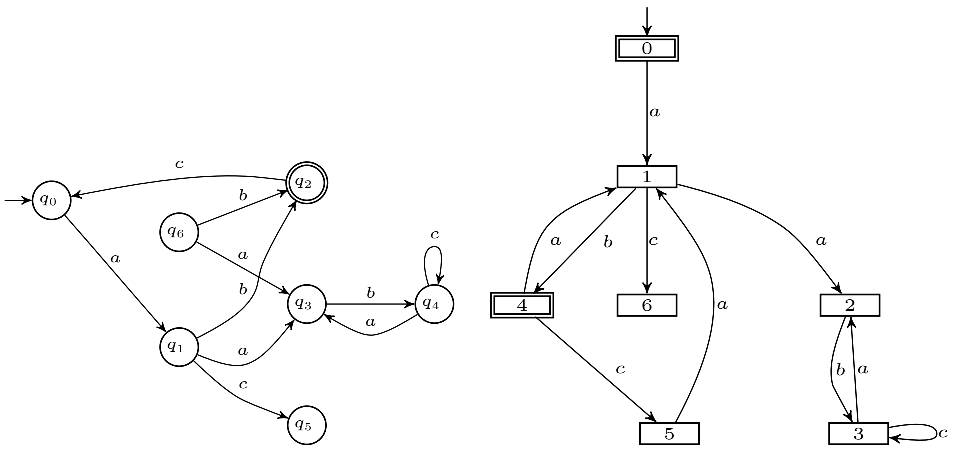
$$L_{m,1}^* := \{\varepsilon\} \cup L_{m,1} \cup L_{m,1}L_{m,1} \cup L_{m,1}L_{m,1}L_{m,1} \cup \cdots \tag{3.2}$$

  Since $L_{m,1}$ is regular, $L_{m,1}^*$ can be obtained through the following instructions. Starting from the input automaton G_1, add a new initial state, mark it, and connect it to the old initial state of G_1. Then, add a ε-transition from every marked state of G_1 to the old initial state. The new finite-state automaton marks $L_{m,1}^*$.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.21(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = kleeneclos(G1)$ (shown in figure 3.21(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1
```

```
X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# kleene closure

G = kleeneclos(G1)

draw(G1, G,'figure')
```



(a) Automaton G_1 (b) Automaton $G = kleenclos(G_1)$

Figure 3.21: Example of the Kleene closure operation.

- See also: Kleene Closure Generator, Prefix Closure

3.2.5 Kleene Closure Generator

- Purpose
  This operation returns a single state automaton that generates and marks a given Σ^*_{input}.

- Syntax

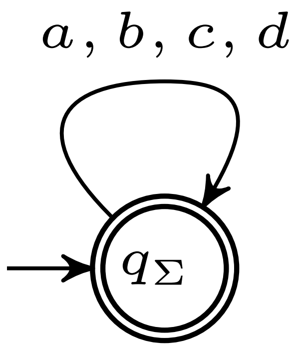
$$G = sigmakleeneclos(\Sigma, x_0 = value, label = value)$$

  - Inputs
    The input parameters are an alphabet Σ_{input}, an inital state x_0 and a label for it. Note that setting the initial state and its label is optional; if only the alphabet is given, then the default initial state is s_0.
  - Output
    The output is a single state automaton generating and marking Σ^*_{input}.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. If the Kleene Closure Generator operation is used to create it, then $X_1 = X_{m,1} = x_{0,1}$, $\Sigma_1 = \Sigma_{input}$ and $f_1(x, e) = x_{0,1}, \forall e \in \Sigma_{input}$.

- Example
  Consider the alphabet Σ_{input}. Using DESLab we can obtain automaton $G = sigmakleeneclos(\Sigma_{input})$ (shown in figure 3.22(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 a b c d q_{\Sigma}')

Sigma1 = [a,b,c,d]

# kleene closure generator

G = sigmakleeneclos(Sigma1, x0=q0, label='q_{\Sigma}')

draw(G,'figure')
```



Automaton $G = sigmakleenclos(\Sigma_{input})$

Figure 3.22: Example of the Kleene closure generator operation.

- See also: Kleene Closure, Prefix Closure

3.2.6 Projection

- Purpose
  This operation returns a nondeterministic automaton generated and marked by the projection of the generated and marked languages of the input automaton, with respect to an observable event set Σ_o.

- Syntax

$$G = proj(G_1, \Sigma_o)$$

  - Inputs
    The input parameter is a finite automaton of the class fsa.
  - Output
    The output is an automaton generated and marked by the projection of the generated and marked languages of the input.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The language generated by G_1 is L. The projection of L which is denoted by P is the mapping

$$\begin{array}{rcl} P : \Sigma^* & \rightarrow & \Sigma_o^*,\ \Sigma_o \subseteq \Sigma \\ s & \mapsto & P(s), \end{array}$$

  satisfying the following properties:

$$\begin{array}{rcl} P(\varepsilon) & = & \varepsilon, \\ P(\sigma) & = & \begin{cases} \sigma, \text{ if } \sigma \in \Sigma_o, \\ \varepsilon, \text{ if } \sigma \in \Sigma \backslash \Sigma_o, \end{cases} \\ P(s\sigma) & = & P(s)P(\sigma), s \in \Sigma^*, \sigma \in \Sigma. \end{array}$$

  The projection operator can be extended to a language L by applying the natural projection to all traces of L. Therefore, if $L \subseteq \Sigma^*$, then

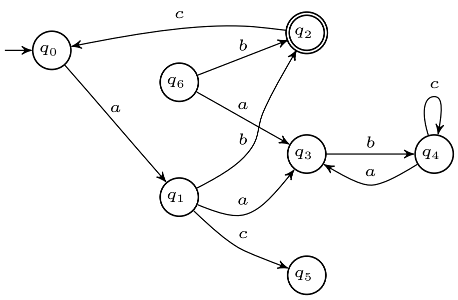
$$P(L) = \{t \in \Sigma_o^* : (\exists s \in L)[P(s) = t]\}$$

  Applying the natural projection concept to the generated and marked languages of G_1, with respect to a Σ_o, will result in two new languages. These new languages generate and mark the automaton resultant of the projection operation.

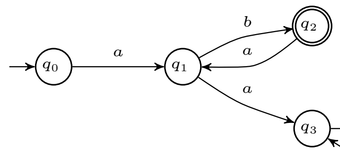
- Example

  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.23(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $\Sigma_o = \{a, b\}$ Using DESLab we can obtain automaton $G = proj(G_1)$ (shown in Figure 3.23(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1
X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
Sigmao = [a,b]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# projection of L(G1)
G = proj(G1,Sigmao)
draw(G1, G,'figure')
```



(a) Automaton G_1



(b) Automaton $G = proj(G_1, Sigma_o)$

Figure 3.23: Example of the projection operation.

- See also: Inverse Projection

3.2.7 Inverse Projection

- Purpose
  This operation returns the inverse projection of an input automaton G_1, with respect to a predefined set of events Σ_s.

- Syntax

$$G = invproj(G_1, \Sigma_s)$$

  - Inputs
    The input parameters are a finite automaton of the class fsa and a set of events Σ_s.
  - Output
    The output is an automaton G that is the result of the inverse projection of G_1.

- Description
  Let $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ denote a finite state automaton generating and marking L_1 and $L_{m,1}$ respectively. Let Σ_s be a predefined smaller set of events, when compared to Σ_1. The inverse map

$$P^{-1} : \Sigma_s^* \to 2^{\Sigma_1^*}$$

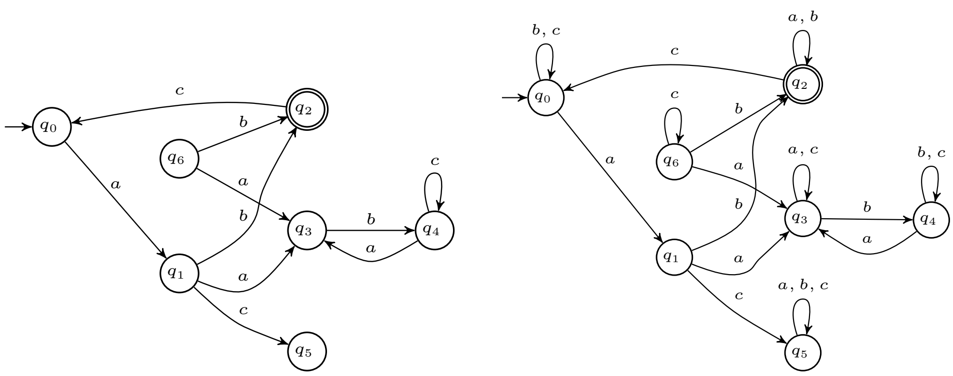
  returns the set of all strings from Σ_1^* that project to the given string. That extended to a language, here the generated (L) and marked (L_m) languages of the output, gives

$$P^{-1}(L) = \{s \in \Sigma_l^* : (\exists t \in L)[P(s) = t]\}, L \subseteq \Sigma_s^*$$

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.24(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$, $\Sigma_s = \{a\}$. Using DESLab we can obtain automaton $G = invproj(G_1, \Sigma_s)$ (shown in figure 3.24(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
```

```
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
Sigmas = [a]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# inverse projection of G1

G2 = invproj(G1,Sigmas)

draw(G1, G2,'figure')
```



(a) Automaton G_1 (b) Automaton $G = invproj(G_1)$

Figure 3.24: Example of the inverse projection operation.

- See also: Projection

3.2.8 Language Difference

- Purpose
  This operation calculates the difference between two languages marked by input automata.

- Syntax

$$\begin{aligned} G &= G_1 - G_2 \\ G &= langdiff(G1, G2) \end{aligned}$$

  - Inputs
    The input parameters are finite automata of the class fsa.
  - Output
    The output is an automaton marked by the difference between the languages marked by the inputs.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0,2}, X_{m_2})$ denote two finite state automata, marked by languages L_{m_1} and L_{m_2}. Denoting language difference as L_D, we have

$$L_D = L_{m_1} \backslash L_{m_2} = L_{m_1} \cap L_{m_2}^c$$

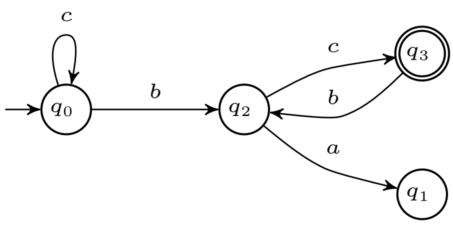
  meaning that the new automaton to be generated and marked will contain all the strings from the marked language of G_1 that are not part of the marked language of G_2, with respect to the same alphabet Σ.

- Example
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 3.18(a) and (b) where $X_1 = \{q_0, q_1, q_2, q_3\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, c) = \{q_0\}$, $f_1(q_0, b) = \{q_2\}$, $f_1(q_2, a) = \{q_1\}$, $f_1(q_2, c) = \{q_3\}$, $f_1(q_3, b) = \{q_2\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_3\}$ and $X_2 = \{q_0, q_1, q_2\}$, $\Sigma_2 = \{a, b, c\}$, $f_2(q_0, a) = \{q_0\}$, $f_2(q_0, b) = \{q_2\}$, $f_2(q_2, a) = \{q_1\}$, $f_2(q_2, c) = \{q_1\}$,$f_2(q_2, c) = \{q_0\}$, $f_2(q_1, b) = \{q_1\}$, $f_2(q_1, a) = \{q_0\}$, $X_{0,2} = \{q_0\}$, $X_{m,2} = \{q_1\}$.

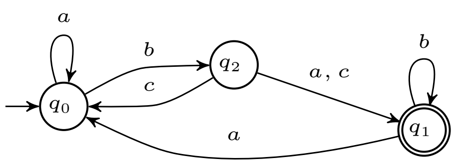
  Using DESLab we can obtain automaton $G_D = G_1 - G_2$ (shown in Figure 3.25(c)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 a b c')
```

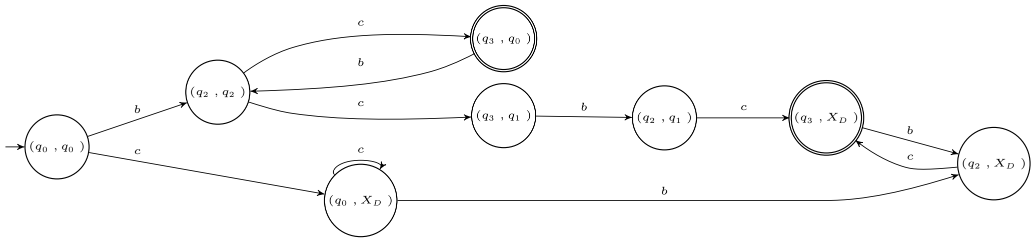
```
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3')]

# automaton definition G1

X1 = [q0,q1,q2,q3]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q3]
T1 =[(q0,c,q0), (q0,b,q2), (q2,a,q1), (q2,c,q3), (q3,b,q2)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# automaton definition G2

X2 = [q0,q1,q2]
Sigma2 = [a,b,c]
X02 = [q0]
Xm2 = [q1]
T2 =[(q0,a,q0), (q0,b,q2), (q2,a,q1), (q2,c,q1), (q2,c,q0),
(q1,b,q1), (q1,a,q0)]
G2 = fsa(X2,Sigma2,T2,X02,Xm2,table,name='$G_2$')

# language difference
GD = G1-G2

draw(G1,G2,GD,'figure')
```



(a) Automaton G_1



(b) Automaton G_2



(c) Automaton $G_D = G_1 - G_2$

Figure 3.25: Example of the language difference operation.

- See also: Concatenation, Language Quotient

3.2.9 Language Quotient

- Purpose
  This operation returns the quotient of languages generated and marked by automata.

- Syntax

$$G = G_1/G_2$$

  - Inputs
    The input parameters are finite automata of the class fsa.
  - Output
    The output is an automaton generated and marked by the quotient of input languages.

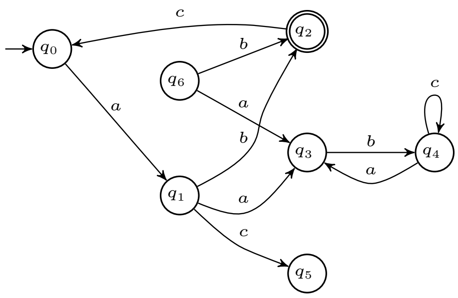
- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0,2}, X_{m_2})$ denote two finite state automata. Let Σ^* be such that $L(G_1), L(G_2) \subseteq \Sigma^*$. The quotient of the languages is defined as follows:

$$L(G_1)/L(G_2) := \{s \in \Sigma^* : (\exists t \in L(G_2))[st \in L(G_1)]\}$$

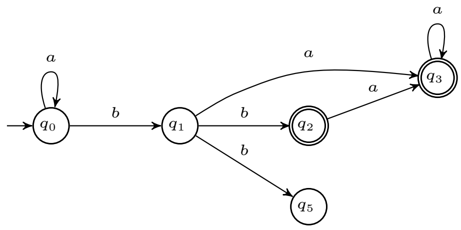
- Example
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 3.26(a) and (b) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $X_2 = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma_2 = \{a, b, c\}$, $f_2(q_0, c) = \{q_1\}$, $f_2(q_2, a) = \{q_2\}$, $f_2(q_1, a) = \{q_3\}$, $f_2(q_1, c) = \{q_2\}$, $f_2(q_2, b) = \{q_0\}$, $f_2(q_3, b) = \{q_4\}$, $f_2(q_4, c) = \{q_4\}$, $f_2(q_4, b) = \{q_3\}$, $f_2(q_0, b) = \{q_2\}$, $f_2(q_1, a) = \{q_3\}$, $X_{0,2} = \{q_2\}$, $X_{m,2} = \{q_4\}$. Using DESLab we can obtain automaton $G_{quo} = G_1/G_2$ (shown in figure 3.26(c)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c Gquo')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6'), (Gquo,'$G_{quo}$')]

# automaton definition G1
```

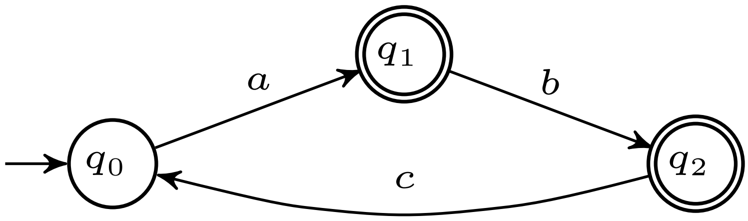
```
X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
SigCon = [a,b,c]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table, Sigcon = SigCon)

# automaton definition G2

X2 = [q0,q1,q2,q3,q5]
Sigma2 = [a,b,c]
X02 = [q1]
Xm2 = [q2,q3]
T2 = [(q0,a,q0),(q0,c,q3),(q1,b,q2),(q2,c,q3),
      (q3,a,q3),(q1,a,q3),(q1,c,q5)]
G2 = fsa(X2,Sigma2,T2,X02,Xm2,table, Sigcon = SigCon)

# language quotient

Gquo = G1/G2

draw(G1,G2,Gquo,'figure')
```



(a) Automaton G_1



(b) Automaton G_2



(c) Automaton $G_{quo} = G_1/G_2$

Figure 3.26: Example of the language quotient operation.

- See also: Language Difference

3.3 Unary and Composition Operations

3.3.1 Accessible Part

- Purpose
  This operation returns the accessible part of an automaton.

- Syntax

$$G = ac(G_1)$$

  - Inputs
    The input parameter is a finite automaton of the class fsa.
  - Output
    The output is the resulting of the accessible part of the input automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The accessible part of G_1 is given by:

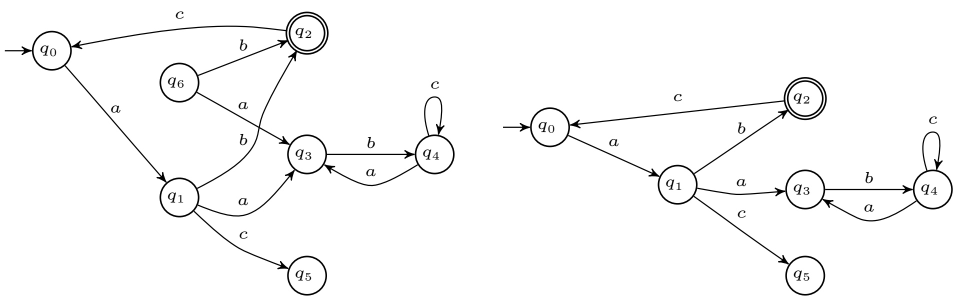
$$\begin{aligned} Ac(G_1) &:= (X_{ac,1}, \Sigma_1, f_{ac,1}, x_{0,1}, X_{ac,m_1}) \text{ where} \\ X_{ac,1} &= \{x \in X_1 : (\exists s \in \Sigma_1^*)[f_1(x_0, s) = x]\} \\ X_{ac,m_1} &= X_{m,1} \cap X_{ac,1} \\ f_{ac,1} &= f_1 \mid X_{ac,1} \times \Sigma_1 \to X_{ac,1} \end{aligned}$$

  Thus, the accessible part operator deletes from G_1 all states that are not accessible or reachable from x_{0_1} by some string in $L(G_1)$.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.27(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = ac(G_1)$ (shown in Figure 3.27(b)) by writing the following instructions:

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]
```

```
# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# accessible part

G = ac(G1)


draw(G1, G,'figure')
```



(a) Automaton G_1 (b) $G = ac(G1)$

Figure 3.27: Example of the accessible part operation.

- See also: Coaccessible Part

3.3.2 Coaccessible Part

- Purpose
  This operation returns the coaccessible part of an automaton.

- Syntax

$$G = coac(G_1)$$

  - Inputs
    The input parameter is a finite automaton of the class fsa.
  - Output
    The output is the resulting of the coaccessible part of the input automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The coaccessible part of G_1 is given by:

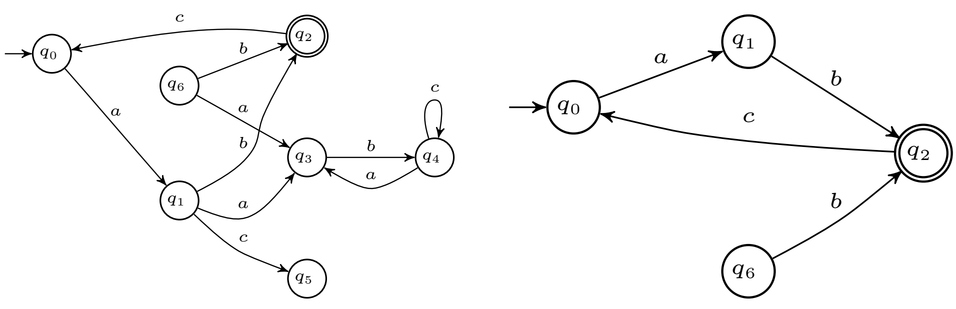
$$\begin{array}{lcl} CoAc(G_1) & := & (X_{coac,1}, \Sigma_1, f_{coac,1}, x_{0,coac_1}, X_{m,1}) \text{ where} \\ X_{coac,1} & = & \{x \in X_1 : (\exists s \in \Sigma_1^*)[f_1(x,s) \in X_{m,1}]\} \\ f_{coac,1} & = & f_1 \mid X_{coac,1} \times \Sigma_1 \rightarrow X_{coac,1} \\ x_{0,coac_1} & = & \begin{cases} x_{0,1} & \text{if } x_{0,1} \in X_{coac,1} \\ undefined & \text{otherwise} \end{cases} \end{array}$$

  Thus, the coaccessible part operator deletes from G_1 all paths that do not get to a marked state.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.28(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = coac(G_1)$ (shown in Figure 3.28(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]
```

```
# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# coaccessible part

G = coac(G1)


draw(G1, G,'figure')
```



(a) Automaton G_1 (b) Automaton $G = coac(G_1)$

Figure 3.28: Example of the coaccessible part operation.

- See also: Accessible Part

3.3.3 Trim

- Purpose
  This operation returns the trim of automaton.

- Syntax

$$G = trim(G_1)$$

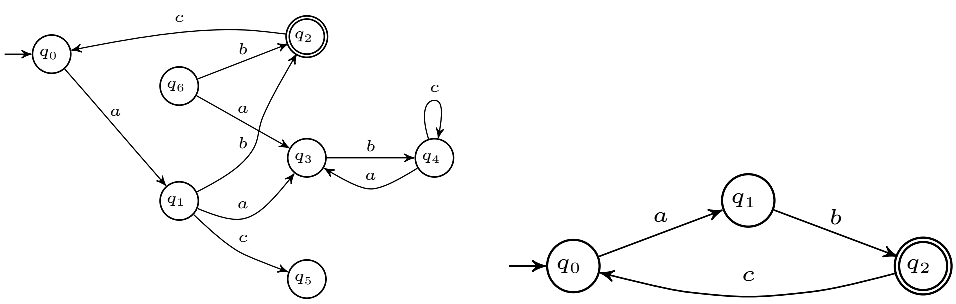
  - Inputs
    The input parameter is a finite automaton of the class fsa.
  - Output
    The output is the resulting of the trim of the input automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. The trim operation of G_1 is an automaton that is accessible and coaccessible at the same time.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.29(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = trim(G_1)$ (shown in Figure 3.29(b)) by writing the following instructions:

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')
```

```
# trim

G = trim(G1)

draw(G,'figure')
```



(a) Automaton G_1 (b) $G = trim(G1)$

Figure 3.29: Example of the trim operation.

- See also: Accessible Part, Coaccessible Part

3.3.4 Complement

- Purpose
  This operation returns the complement of automaton.

- Syntax

$$G^{comp} = complement(G_1)$$

  - Inputs
    The input parameter is a finite automaton of the class fsa.
  - Output
    The output is the resulting of the complement of the input automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton that marks the language $L_{m,1} \subseteq \Sigma_1^*$. The complement of G_1 is an automaton G that marks the language $\Sigma_1^* \backslash L_1$ and can be built in a few steps.

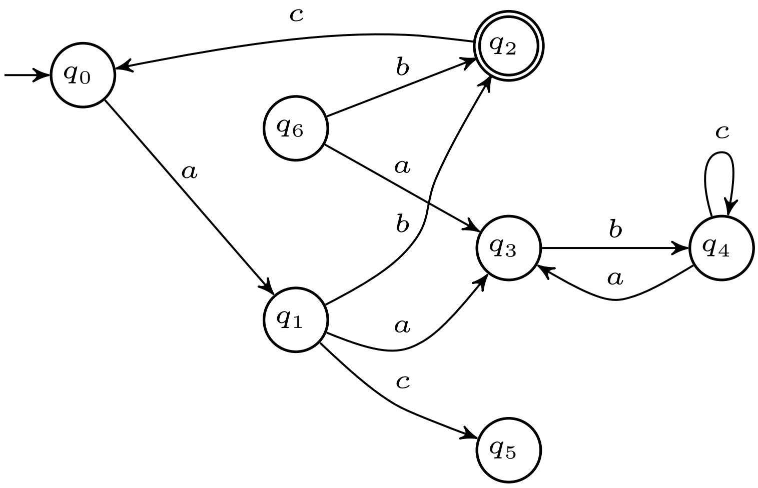
  1. Take the trim part of G_1;
  2. Add a new state x_d to X_1;
  3. Complete f_1. In order to do so, create transitions from every $x \in X_1$ to x_d labbeled by events $e \in \Sigma_1 \backslash \Gamma(x)$;
  4. Change the marking status of the states of G_1 by unmarking the marked states and marking the unmarked ones.

  The result is the automaton G that marks $\Sigma_1^* \backslash L$.

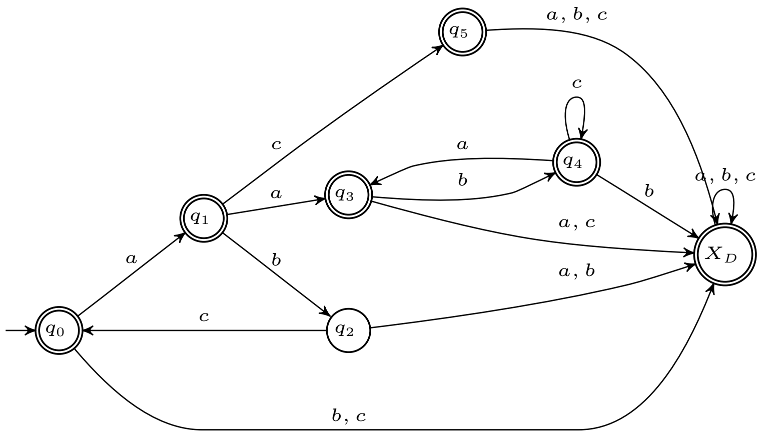
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.30(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = \ G_1$ (shown in Figure 3.30(b)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
```

```
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# complement of G1

G = ~(G1)

# another notation

G = complement(G1)


draw(G1, G,'figure')
```



(a) Automaton G_1



(b) Automaton $G = complement(G_1)$

Figure 3.30: Example of the complement operation.

- See also: Complete Automaton

3.3.5 Complete Automaton

- Purpose
  This operation computes the complete automaton that generates Σ from a given input automaton.

- Syntax

$$G_{complete} = complete(G1)$$

  - Inputs
    The input parameter is a finite state automaton.
  - Output
    The output is the automaton generated by the complete alphabet of the input.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton generated by L_1. Regarding G_1, a complete automaton $G_{complete}$ would be the one whose generated language equals Σ^* and the marked language is L_1. In order to create $G_{complete}$, a simple algorithm should be followed. It consists on completing the transition function f_1 by adding a dump state x_d to X_1. The new automaton $G_{complete} = (X \cup \{x_d\}, \Sigma, f_{complete}, x_0, X_m)$ is then created, where

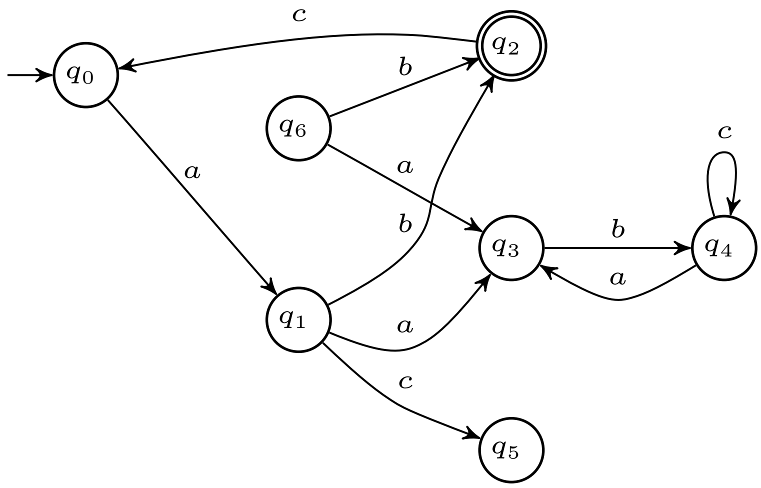
$$f_{complete} = \begin{cases} f_1(x,e) & \text{if } e \in \Gamma_1(x) \\ x_d & \text{if } e \notin \Gamma_1(x) \vee x = x_d, \forall\ e \in \Sigma \end{cases}$$

  $G_{complete}$ generates $L_{complete} = \Sigma^*$ and marks $L_{m,complete} = L_1$ as intended.

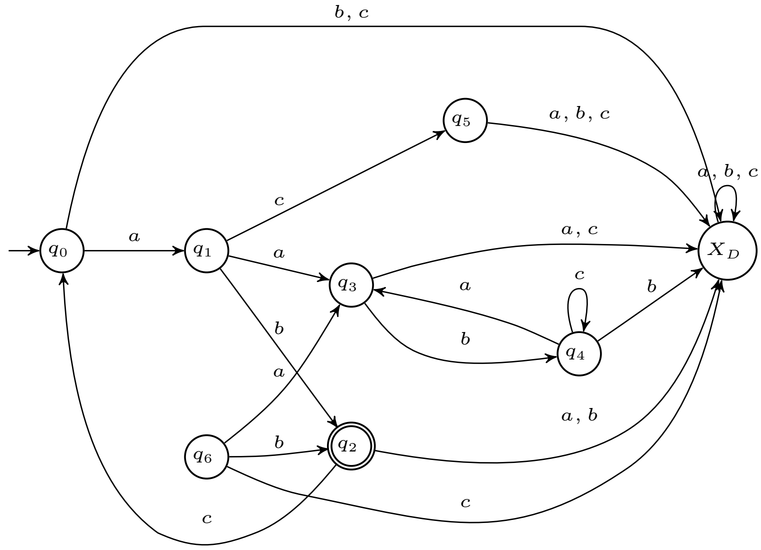
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.31(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = complete(G1)$ (shown in Figure 3.31(b)) by writing the following instructions:

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
```

```
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# complete automaton

G2 = complete(G1)

draw(G1,G2,'figure')
```



(a) Automaton G_1



(b) Automaton $G_{complete} = complete(G_1)$

Figure 3.31: Example of the complete automaton operation.

- See also: Comparison Instructions, Empty Automaton, Complement

3.3.6 Empty Automaton

- Purpose
  This operation returns an empty automaton.

- Syntax

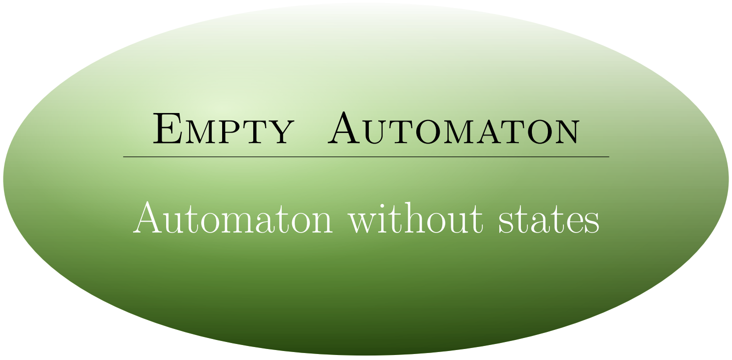
$$G1 = fsa()$$

  - Inputs
    There is no input.
  - Output
    The output is an empty automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton. Considering an empty set of states, follows that the generated and marked languages are also empty. That configuration characterizes the empty automaton.

- Example
  Using DESLab we can obtain automaton $G = empty(G1)$ (shown in Figure 3.32(b)) by writing the following instructions:

```
from  deslab import *

# empty automaton definition G1

G1 = fsa()

draw(G1,'figure')
```



Empty Automaton G_1

Figure 3.32: Example of the empty automaton operation.

- See also: Comparison Instructions, Complete Automaton

3.3.7 Product

- Purpose
  This operation returns the product between automata.

- Syntax

$$
\begin{array}{rcl}
G & = & G_1 \& G_2 \\
G & = & \mathit{product}(G1, G2) \\
G & = & G_1 \& G_2 \& \cdots \& G_n \\
G & = & \mathit{product}(G_1, G_2, \cdots, G_n)
\end{array}
$$

  - Inputs
    The input parameters are finite automata of the class fsa.
  - Output
    The output is an automaton that the result of the product between the input automata.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0,2}, X_{m_2})$ denote two finite state automata. The product between G_1 and G_2 (denoted as $G_1 \times G_2$) is said to completely synchronize both automata in a sense of only allowing common events to label transitions.The active event sets Γ of the states where G_1 and G_2 are must contain the same event so that the transition is feasible. Thus, the product of G_1 and G_2 is defined as $G_1 \times G_2 := Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f, \Gamma_{1\times 2}, (x_{01}, x_{02}), X_{m,1} \times X_{m,2})$, where

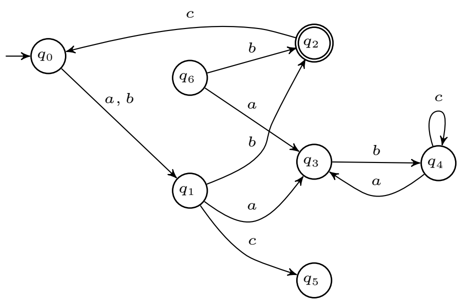
$$
f((x_1, x_2), e) \;:=\; \begin{cases} (f_1(x_1, e), f_2(x_2, e)), & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined}, & \text{otherwise} \end{cases}
$$

  and $\Gamma_{1\times 2}(x_1, x_2) \;=\; \Gamma_1(x_1) \cap \Gamma_2(x_2)$.

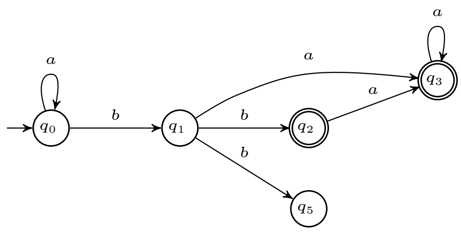
- Example
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 3.33(a) and (b) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_0, b) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $X_2 = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma_2 = \{a, b\}$, $f_2(q_0, a) = \{q_0\}$, $f_2(q_0, b) = \{q_1\}$, $f_2(q_2, a) = \{q_3\}$, $f_2(q_1, b) = \{q_2\}$, $f_2(q_3, a) = \{q_3\}$, $f_2(q_1, a) = \{q_3\}$,

$f_2(q_1, b) = \{q_5\}$, $X_{0,2} = \{q_2\}$, $X_{m,2} = \{q_2, q_3\}$. Using DESLab we can obtain automaton $G = G_1 \times G_2$ (shown in Figure 3.33(c)) by writing the following instructions.

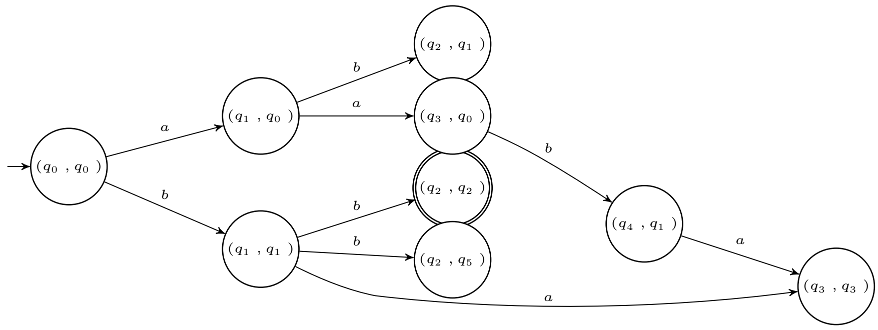
```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c e ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q0,b,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5),
(q2,c,q0),(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# automaton definition G2

X2 = [q0,q1,q2,q3,q5]
Sigma2 = [a,b]
X02 = [q0]
Xm2 = [q2,q3]
T2 = [(q0,a,q0), (q0,b,q1), (q1,b,q2),(q2,a,q3),(q3,a,q3),
(q1,a,q3), (q1,b,q5)]
G2 = fsa(X2,Sigma2,T2,X02,Xm2,table,name='$G_2$')

# product

G = G1&G2

# another possible notation

G = product(G1,G2)

draw(G1, G2, G, 'figure')
```



(a) Automaton G_1



(b) Automaton G_2



(c) Automaton $G = G_1 \times G_2$

Figure 3.33: Example of the product operation.

- See also: Parallel Composition

3.3.8 Parallel Composition

- Purpose
  This operation returns the parallel composition between automata.

- Syntax

$$
\begin{aligned}
G &= G_1//G_2 \\
G &= parallel(G1, G2) \\
G &= G_1//G_2//\cdots//G_n
\end{aligned}
$$

  - Inputs
    The input parameters are finite automata of the class fsa.
  - Output
    The output is an automaton that is the result of the parallel composition between the input automata.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0,2}, X_{m_2})$ denote two finite state automata. The parallel composition between G_1 and G_2 (denoted as $G_1 \| G_2$) produces an automaton with the following behavior: *(i)* a common event of G_1 and G_2 can occur, only when G_1 and G_2 are in states whose active event sets both have this event, *(ii)* private events, *i.e.*, events belonging either to $\Sigma_1 \setminus \Sigma_2$ or to $\Sigma_2 \setminus \Sigma_1$ can occur as long as they belong to the active event set of the current state. Thus, the parallel composition of G_1 and G_2, which is often called synchronous composition, is defined as follows.

$$
G_1 \| G_2 \quad = \quad \mathrm{Ac}(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f_{1\|2}, \Gamma_{1\|2}, (x_{0_1}, x_{0_2}), X_{m_1} \times X_{m_2}),
$$

  where $\times$ denote the cartesian product and Ac denotes the accessible part of $G_1 \| G_2$, which is formed by the states that are reached from the initial state by some trace in $(\Sigma_1 \cup \Sigma_2)^*$. The transition function of $G_1 \| G_2$ is defined as:

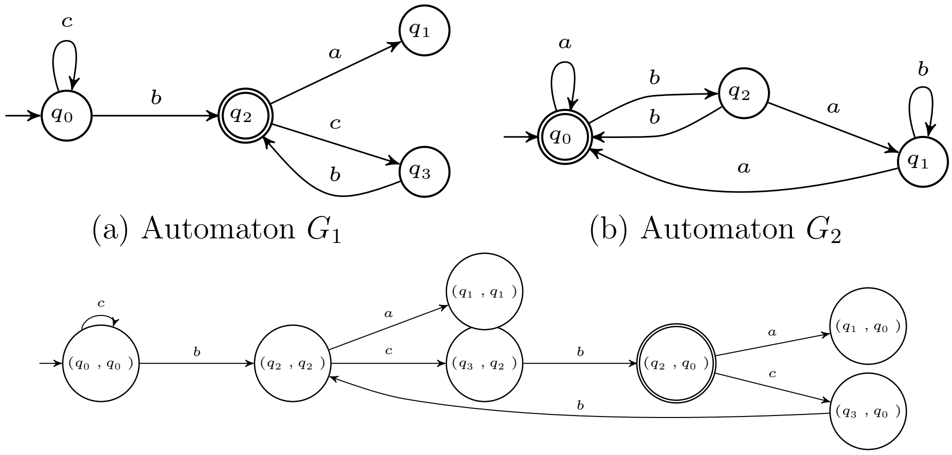
$$
\begin{aligned}
f((x_1, x_2), e) &:= \begin{cases} (f_1(x_1, e), f_2(x_2, e)), & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, e), x_2), & \text{if } e \in \Gamma_1(x_1) \setminus \Sigma_2 \\ (x_1, f_2(x_2, e)), & \text{if } e \in \Gamma_2(x_2) \setminus \Sigma_1 \\ \text{undefined}, & \text{otherwise} \end{cases} \\
\Gamma_{1||2}(x_1, x_2) &= [\Gamma_1(x_1) \cap \Gamma_2(x_2)] \cup [\Gamma_1(x_1) \setminus \Sigma_2] \cup [\Gamma_2(x_2) \setminus \Sigma_1].
\end{aligned}
$$

  Note that for the special case where $\Sigma_1 = \Sigma_2$, the parallel composition works just like the product operation.

- Example
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 3.18(a) and (b) where $X_1 = \{q_0, q_1, q_2, q_3\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, c) = \{q_0\}$, $f_1(q_0, b) = \{q_2\}$, $f_1(q_2, a) = \{q_1\}$, $f_1(q_2, c) = \{q_3\}$, $f_1(q_3, b) = \{q_2\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $X_2 = \{q_0, q_1, q_2\}$, $\Sigma_2 = \{a, b\}$, $f_2(q_0, a) = \{q_0\}$, $f_2(q_0, b) = \{q_2\}$, $f_2(q_2, a) = \{q_1\}$, $f_2(q_2, b) = \{q_0\}$, $f_2(q_1, b) = \{q_1\}$, $f_2(q_1, a) = \{q_0\}$, $X_{0,2} = \{q_0\}$, $X_{m,2} = \{q_0\}$. Using DESLab we can obtain automaton $G = G_1 \parallel G_2$ (shown in figure 3.34(c)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3')]

# automaton definition G1

X1 = [q0,q1,q2,q3]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,c,q0), (q0,b,q2), (q2,a,q1), (q2,c,q3), (q3,b,q2)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# automaton definition G2

X2 = [q0,q1,q2]
Sigma2 = [a,b]
X02 = [q0]
Xm2 = [q0]
T2 =[(q0,a,q0), (q0,b,q2), (q2,a,q1), (q2,b,q0), (q1,b,q1),
    (q1,a,q0)]
G2 = fsa(X2,Sigma2,T2,X02,Xm2,table,name='$G_2$')

# parallel composition

G = G1//G2

# another possible notation

G = parallel(G1,G2)
```

```
draw(G1,G2,G,'figure')
```



(a) Automaton G_1

(b) Automaton G_2

(c) Automaton $G = G_1 \| G_2$

Figure 3.34: Example of the parallel composition operation.

- See also: Product

3.3.9 Observer Automaton

- Purpose
  This operation returns a deterministic automaton which generates and marks the projection of the generated and marked languages of an input automaton, with respect to a determined alphabet of observable events.

- Syntax

$$
\begin{aligned}
G_{obs} &= observer(G1, \Sigma_o) \\
G_{obs} &= observer(G1)
\end{aligned}
$$

  - Inputs
    The input parameters are a finite state automaton and an alphabet of observable events Σ_o. If the alphabet is not provided, then the set of observable events of the input will be taken in account.
  - Output
    The output is the observer automaton with respect to the respective set of observable events.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a "partially-observed" finite state automaton, that is, its set of events is partitioned into the subsets of observable events $\Sigma_{1,o}$ and the unobservable events $\Sigma_{1,uo}$. Events in $\Sigma_{1,uo}$ are treated as ε events, since they cannot be observed. The natural projection P of the the languages of G_1 will be from Σ_1 to $\Sigma_{1,o}$, and the procedure requires the definition of a set of unobservable reach states denoted by $UR(x), x \in X_1$ and defined as

$$UR(x) = \{y \in X : (\exists t \in \Sigma_{1,uo})[(f_1(x,t) = y)]\}$$

  which can be extended to sets of states $B \subseteq X_1$ by

$$UR(B) = U_{x \in B} UR(x).$$

  The new automaton G_{obs} that is deterministic and generates and marks the same languages as G_1 can be built in four simple steps:

  Step 1: Define $x_{0,obs} := UR(x_{0,1})$ and set $X_{obs} = x_{0,obs}$.

  Step 2: For each B $\in X_{obs}$ and $e \in \Sigma_{1,o}$, define

$$f_{obs}(B, e) := UR(x_1 \in X_1 : (\exists x_e \in B)[x_1 \in f_1(x_e, e)])$$

whenever $f_1(x_e, e)$ is defined for some $x_e \in B$. In this case, add the state $f_{obs}(B, e)$ to X_{obs}. If $f_1(x_e, e)$ is not defined for any $x_e \in B$, then $f_{obs}(B, e)$ is not defined.

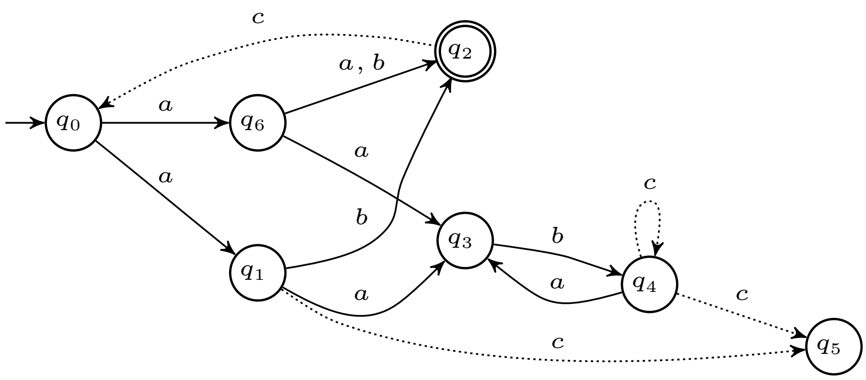
Step 3: Repeat Step 2 until the entire accessible part of G_{obs} has been constructed.

Step 4: $X_{m,obs} := \{B \in X_{obs} : B \cap X_m \neq \emptyset\}$

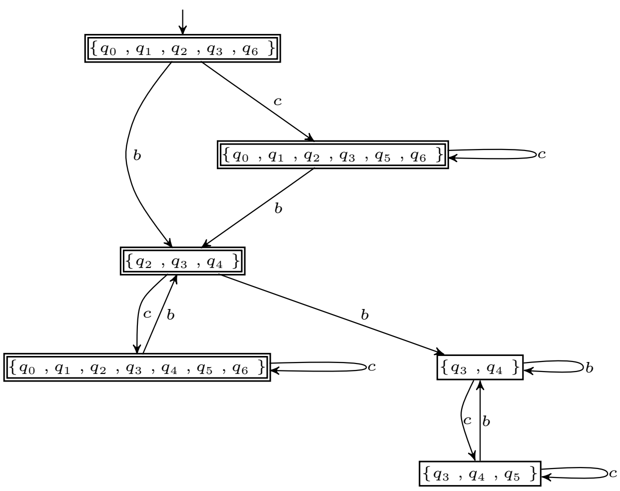
- Example
Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.35(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_0, a1) = \{q_6\}$ $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, c) = \{q_5\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $f_1(q_6, a) = \{q_2\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. It is desired that two automata are generated, being the first one according to a given set of observable events. Using DESLab we can obtain automata $G_{obs,1}$ and $G_{obs,2}$ (shown in Figure 3.35(b) and (c)) by writing the following instructions:

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1
X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
obs = [a,b]
T1 =[(q0,a,q1), (q0,a,q6), (q1,b,q2), (q1,a,q3), (q1,c,q5),
     (q2,c,q0), (q3,b,q4), (q4,c,q4), (q4,c,q5), (q4,a,q3),
     (q6,b,q2), (q6,a,q3), (q6,a,q2)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,Sigobs=obs,name='$G_1$')

# test 1: observer with given Sigma_o
Gobs1 = observer(G1,[b,c])

# test 2: observer with no Sigma_o provided
Gobs2 = observer(G1)
```

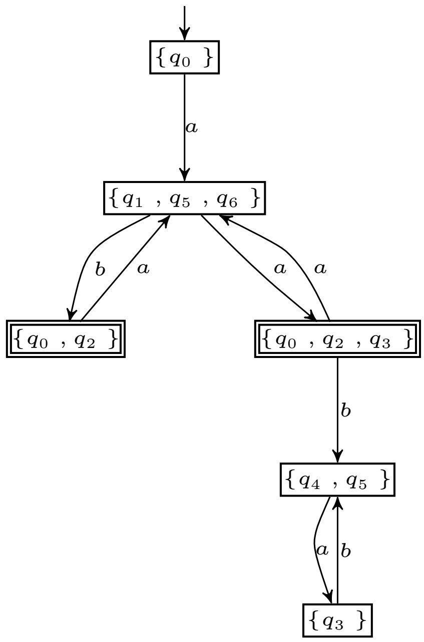
```
draw(G1, Gobs1, Gobs2,'figure')
```



(a) Automaton G_1



(b) Automaton $G_{obs,1}$



(b) Automaton $G_{obs,2}$

Figure 3.35: Example of the observer operation.

- See also: Projection, Inverse Projection, Epsilon Observer

3.3.10 Epsilon Observer

- Purpose
  This function returns a deterministic automaton $G_{\varepsilon-obs}$ which generates and marks the projection of the generated and marked languages of an input G_1, with respect to an alphabet composed by the ε event.

- Syntax

$$G_{\varepsilon-obs} = epsilonobserver(G1)$$

  - Inputs
    The input parameter is a finite state automaton.
  - Output
    The output is the epsilon observer automaton.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0,1}, X_{m_1})$ denote a finite state automaton generated and marked by L_1 and $L_{m,1}$. Consider that $\varepsilon \in \Sigma_1$, and that it labels transitions in f_1. $G_{\varepsilon-obs}$ will then respectively generate and mark the projections of L_1 and $L_{m,1}$ with respect to $\Sigma_\varepsilon = \{\varepsilon\}$. Note that if all the events in G_1 are observable, then $G_{\varepsilon-obs} = G_{obs}$ from the Observer Automaton operation, that can be found in Section 3.3.9. Another special case happens when ε is not labelling any transition from the transition function of an input automaton G. In that case, $G_{\varepsilon-obs} = G$. It is all neatly explained in the example.

  $G_{\varepsilon-obs}$ can be built in four simple steps:

  Step 1: Define $x_{0,\varepsilon-obs} := \varepsilon R(x_{0,1})$ and set $X_{\varepsilon-obs} = x_{0,obs}$.

  Step 2: For each B $\in X_{\varepsilon-obs}$ and $e \in \Sigma_1$, define

$$f_{obs}(B, e) := \varepsilon R(x_1 \in X_1 : (\exists x_e \in B)[x_1 \in f_{nd}(x_e, e)])$$

  whenever $f_{nd}(x_e, e)$ is defined for some $x_e \in B$. In this case, add the state $f_{\varepsilon-obs}(B, e)$ to $X_{\varepsilon-obs}$. If $f_{nd}(x_e, e)$ is not defined for any $x_e \in B$, then $f_{obs}(B, e)$ is not defined.

  Step 3: Repeat Step 2 until the entire accessible part of G_{obs} has been constructed.

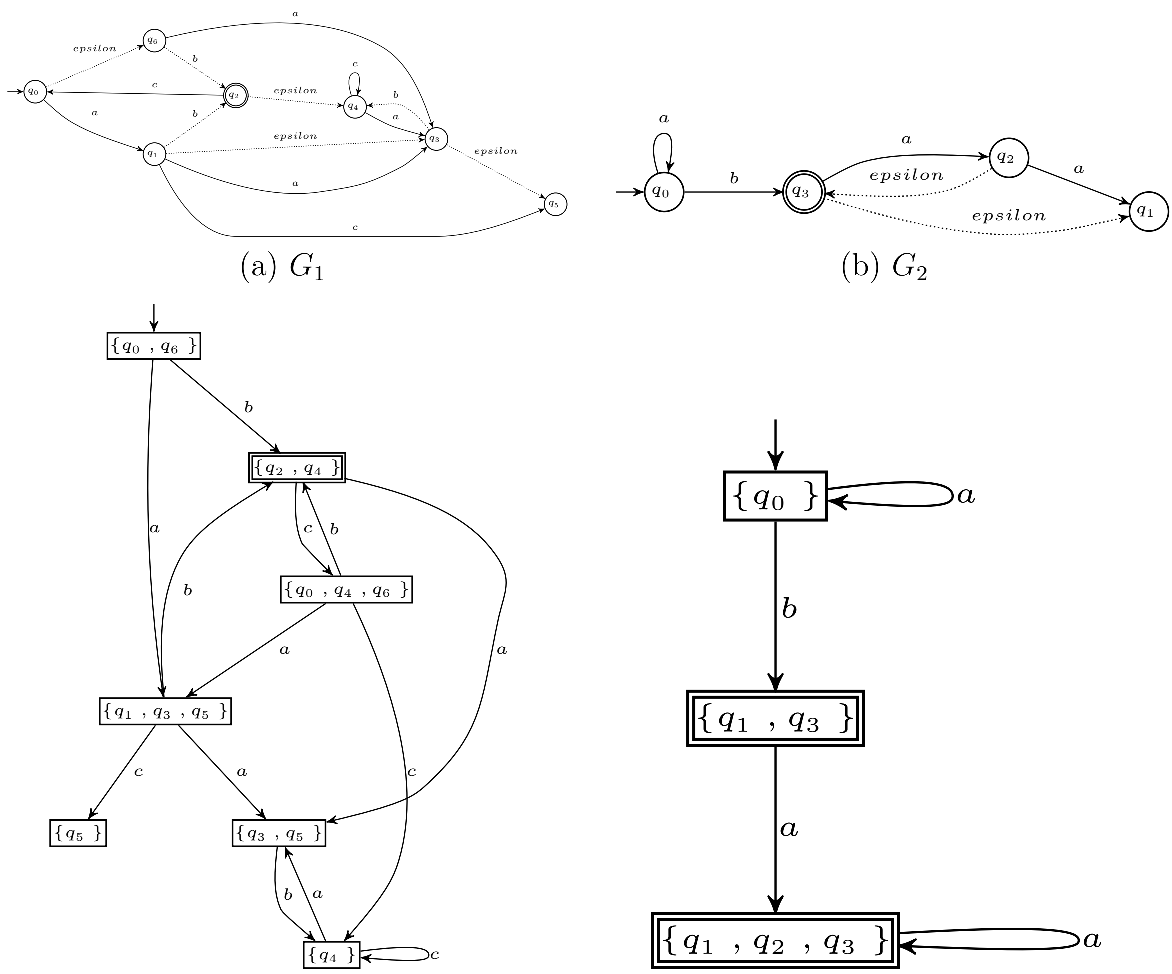
  Step 4: $X_{m,\varepsilon-obs} := \{B \in X_{\varepsilon-obs} : B \cap X_m \neq \emptyset\}$

- Example
Consider the automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.36(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c, \varepsilon\}$, $Sigobs_1 = \{a, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $f_1(q_0, \varepsilon) = \{q_6\}$, $f_1(q_2, \varepsilon) = \{q_4\}$, $f_1(q_1, \varepsilon) = \{q_3\}$, $f_1(q_3, \varepsilon) = \{q_5\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 3.36(b) where $X_2 = \{q_0, q_1, q_2, q_3\}$, $\Sigma_2 = \{a, b, c, \varepsilon\}$, $f_2(q_0, a) = \{q_0\}$, $f_2(q_0, b) = \{q_3\}$, $f_2(q_2, a) = \{q_1\}$, $f_2(q_2, \varepsilon) = \{q_3\}$, $f_2(q_3, \varepsilon) = \{q_1\}$, $f_2(q_3, a) = \{q_2\}$, $X_{0,2} = \{q_0\}$, $X_{m,2} = \{q_3\}$ and $G_3 = (X_3, \Sigma_3, f_3, \Gamma_3, X_{0,3}, X_{m,3})$ shown in Figure 3.37(g) where $X_3 = \{q_0, q_1, q_2, q_3\}$, $\Sigma_3 = \{a, b, c\}$, $Sigobs_3 = \{a\}$, $f_3(q_0, a) = \{q_1\}$, $f_3(q_0, b) = \{q_2\}$, $f_3(q_2, c) = \{q_1\}$, $f_3(q_3, c) = \{q_1\}$, $f_3(q_3, b) = \{q_2\}$, $X_{0,3} = \{q_0\}$, $X_{m,3} = \{q_2\}$.
Using DESLab we can obtain automaton $G_{epsobs1} = epsilonobserver(G1)$ (shown in Figure 3.36(c)), $G_{obs1} = observer(G1)$ (shown in figure 3.37(e)) $G_{epsobs2} = epsilonobserver(G2)$ (shown in Figure 3.36(d)), $G_{obs2} = observer(G2)$ (shown in figure 3.37(f)), $G_{epsobs3} = epsilonobserver(G3)$ (shown in Figure 3.37(h)), and $G_{obs3} = observer(G3)$ (shown in figure 3.37(i)) by writing the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1: not every event is observable

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c,epsilon]
Sigobs1 = [a,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2),(q6,a,q3),
(q0,epsilon,q6),(q2,epsilon,q4), (q1,epsilon,q3),
(q3,epsilon,q5)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,Sigobs1,name='$G_1$')

# automaton definition G2: every event is observable
```

```
X2 = [q0,q1,q2,q3]
Sigma2 = [a,b,c,epsilon]
X02 = [q0]
Xm2 = [q3]
T2 =[(q0,a,q0),(q0,b,q3),(q2,a,q1),(q2,epsilon,q3),
(q3,epsilon,q1),(q3,a,q2)]
G2 = fsa(X2,Sigma2,T2,X02,Xm2,table,name='$G_2$')

# automaton definition G3:not every event is observable
#                          and epsilon is not defined

X3 = [q0,q1,q2,q3]
Sigma3 = [a,b,c]
Sigobs3 = [a]
X03 = [q0]
Xm3 = [q2]
T3 =[(q0,a,q1),(q0,b,q2),(q2,c,q1),(q3,c,q1),(q3,b,q2)]
G3 = fsa(X3,Sigma3,T3,X03,Xm3,table,Sigobs3,name='$G_3$')

# epsilon observer

Gepsobs1 = epsilonobserver(G1)
Gepsobs2 = epsilonobserver(G2)
Gepsobs3 = epsilonobserver(G3)

# observer

Gobs1 = observer(G1)
Gobs2 = observer(G2)
Gobs3 = observer(G3)

draw(G1, Gepsobs1, Gobs1, G2, Gepsobs2, Gobs2,
G3, Gepsobs3, Gobs3, 'figure')
```

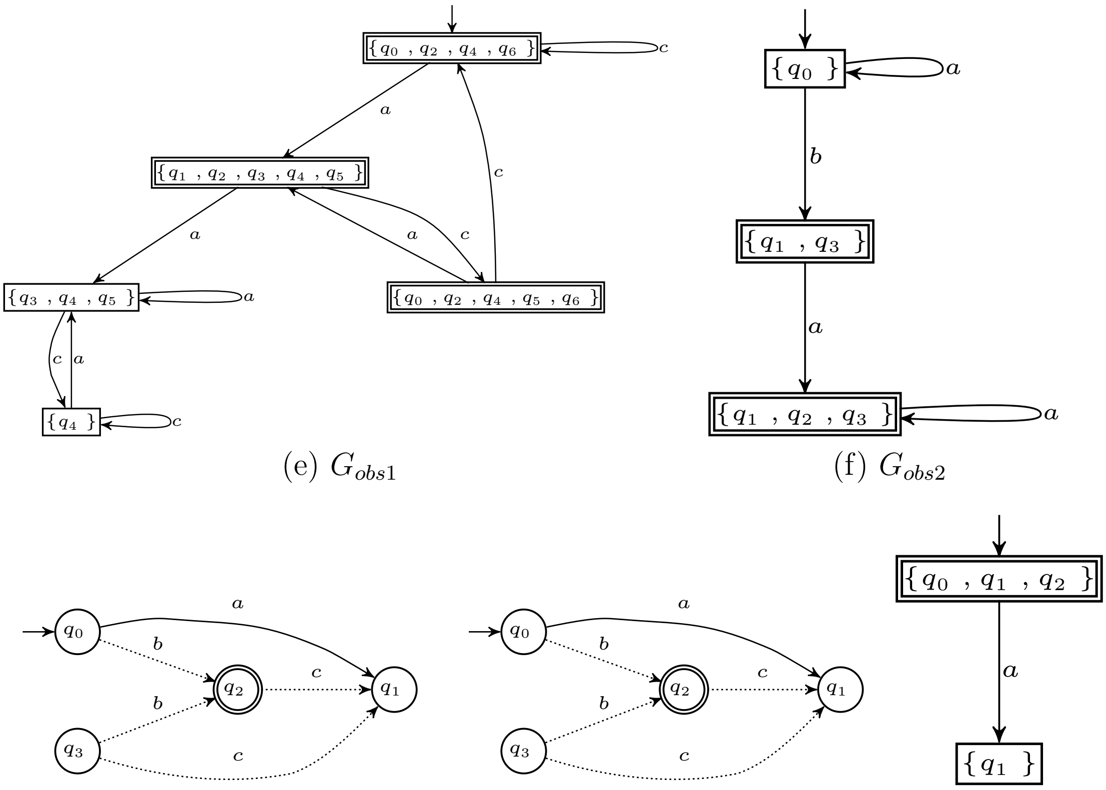
- See also: Projection, Inverse Projection, Observer Automaton



(a) G_1 (b) G_2

(c) $G_{epsobs1}$ (d) $G_{epsobs2}$

Figure 3.36: Example of the epsilon observer operation.



(e) G_{obs1} (f) G_{obs2}

(g) G_3 (h) $G_{epsobs3}$ (i) G_{obs3}

Figure 3.37: Example of the epsilon observer operation.

3.4 Additional Instructions of DESLab

3.4.1 Comparison Instructions

- Purpose
  This set of operations perform comparison tests between automata.

- Syntax
  The syntax of the comparison instructions can be seen in Table 4.2.

Table 3.4: Syntax of the comparison instructions

Instruction	Syntax
Inclusion	$G_1 <= G_2 \vee G_2 >= G_1$
Empty language test	isitempty(G_1)
Automata equivalence	$G_1 == G_2 \vee G_1 <> G_2$
Completeness test	isitcomplete(G_1)

  - Inputs
    The input parameters are finite state automata.
  - Output
    The output are true or false answers on the console regarding the comparison requested.

- Description
  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0_2}, X_{m_2})$ denote two finite state automata. The comparison instructions possible to be performed are simply explained in Table 4.3.

Table 3.5: Description of the comparison instructions

Instruction	Description
Inclusion	answers if $L_m(G_1) \subseteq L_m(G_2)$
Empty language test	evaluates whether or not $L(G_1) = \emptyset$
Automata equivalence	decides if they are equal or different
Completeness test	evaluates whether or not $L(G_1) = \Sigma^*$

- Example

  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 3.38(a) and (b) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $X_2 = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma_2 = \{a, b, c\}$, $f_2(q_3, c) = \{q_1\}$, $f_2(q_2, a) = \{q_2\}$, $f_2(q_1, a) = \{q_3\}$, $f_2(q_1, c) = \{q_2\}$, $f_2(q_2, c) = \{q_0\}$, $f_2(q_3, b) = \{q_4\}$, $f_2(q_4, c) = \{q_4\}$, $f_2(q_4, a) = \{q_3\}$, $f_2(q_0, b) = \{q_2\}$, $f_2(q_1, a) = \{q_3\}$, $X_{0,2} = \{q_2\}$, $X_{m,2} = \{q_4\}$. Using DESLab we can test the comparison instructions by writing the following code.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# automaton definition G2

X2 = [q0,q1,q2,q3,q4]
Sigma2 = [a,b,c]
X02 = [q2]
Xm2 = [q4]
T2 =[(q3,c,q1), (q2,a,q2), (q1,a,q3), (q1,c,q2), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q0,b,q2), (q1,a,q3)]
G2 = fsa(X2,Sigma2,T2,X02,Xm2,table,name='$G_2$')

# inclusion

G1<=G2
```

```
G2>=G1

# empty language test

isitempty(G1)

# automata equivalence

G1==G2
G1<>G2

# completeness test

isitcomplete(G1)
```

- Console outputs
  The response to the example, which should be plugged in the console, is:

  G1<=G2
  >>> False

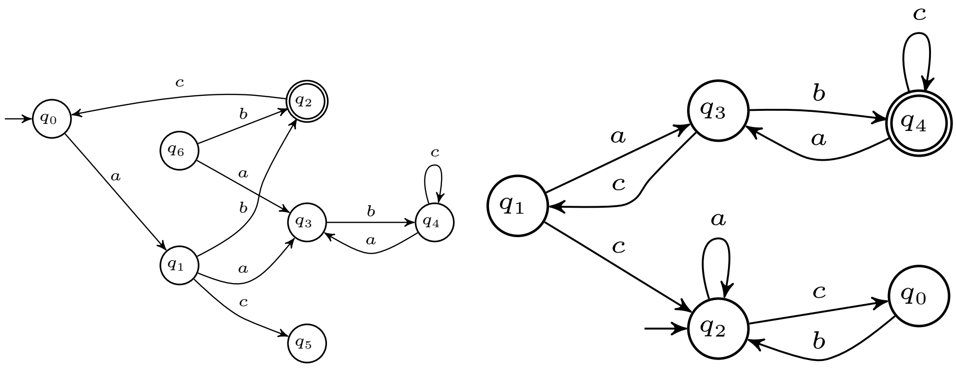
  G2>=G1
  >>>False

  isitempty(G1)
  >>>False

  G1==G2
  >>>False

  G1<>G2 >>>True

  isitcomplete(G1)
  >>>False



(a) Automaton G_1 (b) Automaton G_2

Figure 3.38: Example of the comparison instructions.

- See also: Graph Algorithms

3.4.2 Marked language verifier

- Purpose
  Determine whether the marked language of an automaton is empty or not.

- Syntax

$$isitemptymarked(G)$$

  - Input: Automaton of the class fsa.
  - Output: Boolean output about the marked language.

- Description
  Consider the automaton $G = (X, \Sigma, f, \Gamma, X_0, X_m)$. Initially, it is checked whether the automaton has the *empty* property, that is, if it is empty. If it has this property, the function returns True. If not, a search for marked states is performed starting from the initial state, and states that are not marked are removed. If no marked states remain in the automaton, then True is returned; otherwise, the function returns False.

- Example
  Consider the automaton $G = (X,\ \Sigma,\ f,\ \Gamma,\ X_0,\ X_m)$ shown in Figure 3.39, where $X = \{x_1, x_2, x_3, x_4\}$, $\Sigma = \{a, b, c\}$, $f(x_1, a) = x_2$, $f(x_2, b) = x_3$, $f(x_3, c) = x_4$, $X_0 = x_1$, $X_m = \{x_2\}$. It is possible to call the function $isitemptymarked(G)$, which generates the answer about the marked language through the following commands.

```
from deslab import *

syms('a b c x1 x2 x3 x4')
table = [(x1,'x_1'),(x2,'x_2'),(x3,'x_3'),(x4,'x_4'),
         (a,'a'),(b,'b'),(c,'c')]

# automaton definition G
X1=[x1,x2,x3,x4]
Sigma=[a,b,c]
T=[(x1,a,x2),(x2,b,x3),(x3,c,x4)]
X0,Xm=[x1],[x2]
G=fsa(X1,Sigma,T,X0,Xm,table=table,name='$G_1$')
```

```
#return boolean output about the marked language
print(isitemptymarked(G))
```

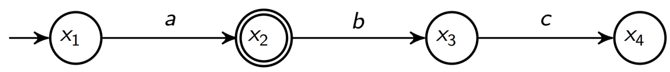


Figure 3.39: Automaton G of the example in subsection 3.4.2.

- Console outputs
  >>> False

3.4.3 Graph Algorithms

- Purpose
  These sort of commands will help dealing with finite state automata transition diagrams.

- Syntax
  The syntax of the available instructions is disposed in Table 3.6.

  Table 3.6: Syntax of the graph algorithm instructions

Command	Syntax
Accessing graph g of the automaton G	$g = G.Graph$
Strongly connected components	$strconncomps(g)$
States with self loops	$selfloopnodes(G)$
Depth First Search	$dfs(g, source)$

  - Inputs
    The input parameters vary according to the command to be used. The lower case g stands for the graph representation of an automaton, not to be mistaken with the transition diagram, which is the graphic representation. It is related to a given finite state automaton, then the upper case G stands for the fsa in use. The field *source* will be filled by the name of the automaton to be searched.
  - Output
    The outputs are lists of states searched by the engines.

- Description
  Let $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ denote a finite state automaton. In order to use the graph algorithms, the user must access a graph at first, then further investigations can be made. The strongly connected components are subsets of states that can be reached from every other state inside the subset by two-way paths. These subsets are complete allowing no other states to be added unless the strong connectivity is violated. States with self loops helps detecting which states present self loops in the graph. The Depth First Search sweeps the entire graph chronologically ordering them in the sense of occurrence.

- Example

  Consider the deterministic automaton $G = (X, \Sigma, f, \Gamma, X_0, X_m)$ shown in Figure 3.40(a) where $X = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma = \{a, b, c\}$, $f(q_0, a1) = \{q_1\}$, $f(q_1, b) = \{q_2\}$, $f(q_1, a) = \{q_3\}$, $f(q_1, c) = \{q_5\}$, $f(q_2, c) = \{q_0\}$, $f(q_3, b) = \{q_4\}$, $f(q_4, c) = \{q_4\}$, $f(q_4, a) = \{q_3\}$, $f(q_6, b) = \{q_2\}$, $f(q_6, a) = \{q_3\}$, $X_0 = \{q_0\}$, $X_m = \{q_2\}$. Using DESLab we can investigate the graph of G (shown in Figure 3.40(b)) by writing the following instructions:

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G

X = [q0,q1,q2,q3,q4,q5,q6]
Sigma = [a,b,c]
X0 = [q0]
Xm = [q2]
T =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')

# All the commands must be run in the console:

# accessing graph of G

g = G.Graph # Attempt for the upper case G on 'Graph'

# finding strongly connected components

strconncomps(g)
strconncomps(G)

# finding states with self loops

selfloopnodes(G)
 selfloopnodes(g)

# running a depth first search
```

```
dfs(g, G)
```

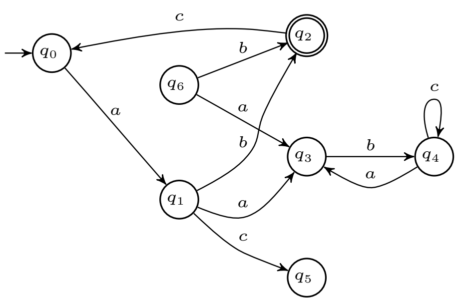
- Console outputs
  The response to the example, which should be plugged in the console, is:

  strconncomps(g)
  >>>[['q1', 'q2', 'q0'], ['q3', 'q4'], ['q5'], ['q6']]

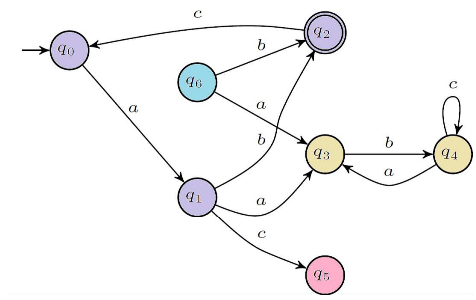
  strconncomps(G)
  >>>[['q1', 'q2', 'q0'], ['q3', 'q4'], ['q5'], ['q6']]

  selfloopnodes(G)
  >>>['q4']

  dfs(g, G)
  >>>frozenset(['q1', 'q0', 'q3', 'q2', 'q5', 'q4', 'q6'])



(a) Automaton G



(b) Automaton G with the strongly connected components sorted by colors.

Figure 3.40: Example of using graph algorithms.

- See also: Drawing a State Transition Diagram, Lexicographical Features

3.4.4 Redefine Graphical Properties

- Purpose
  This operation redefines graphical properties of the state transition diagrams.

- Syntax

$$G.setgraphic(style =' value', ranksep, nodesep, direction =' value')$$

- Inputs
  The inputs are:

  Style - Defines the shape of the states in the state transition diagrams. There are a few options, but basically 'normal' and 'vertical' stand for circles; 'rectangle', 'verifier', 'diagnoser' and 'observer' generate states shaped as rectangles.

  Ranksep - A number defining the separation proportion between the arcs of the state transition diagrams. The default is 0.25.

  Nodesep - A number defining the separation proportion between the nodes of the state transition diagrams. The default is 0.25.

  Direction - Two letters that indicates whether the states are going to be displaced from left to right ('LR') or vertically from the top and growing down ('UD').

- Output
  The output is the new state transition diagram with some aspects redefined.

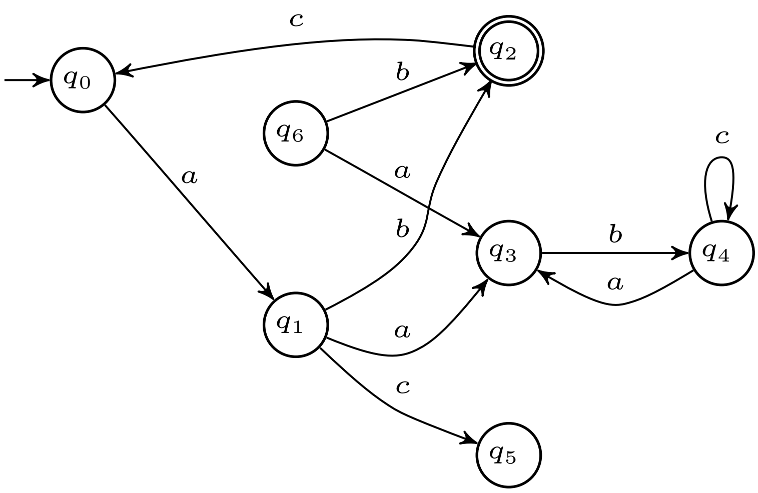
- Description
  Redefining graphical properties can be handy for the user willing to write routines and explore the uses of DESLab. Verifiers and diagnosers, for instance, can be implemented using the commands available. The graphical properties become then relevant since it brings a better presentation of the results. An observer with circle states would be a little inappropriate, if the literature conventions are taken in account. Furthermore, sometimes the drawing is just a little messy, and changing the separation between arcs and nodes enhance the diagram visibility.

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 3.41(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 =$

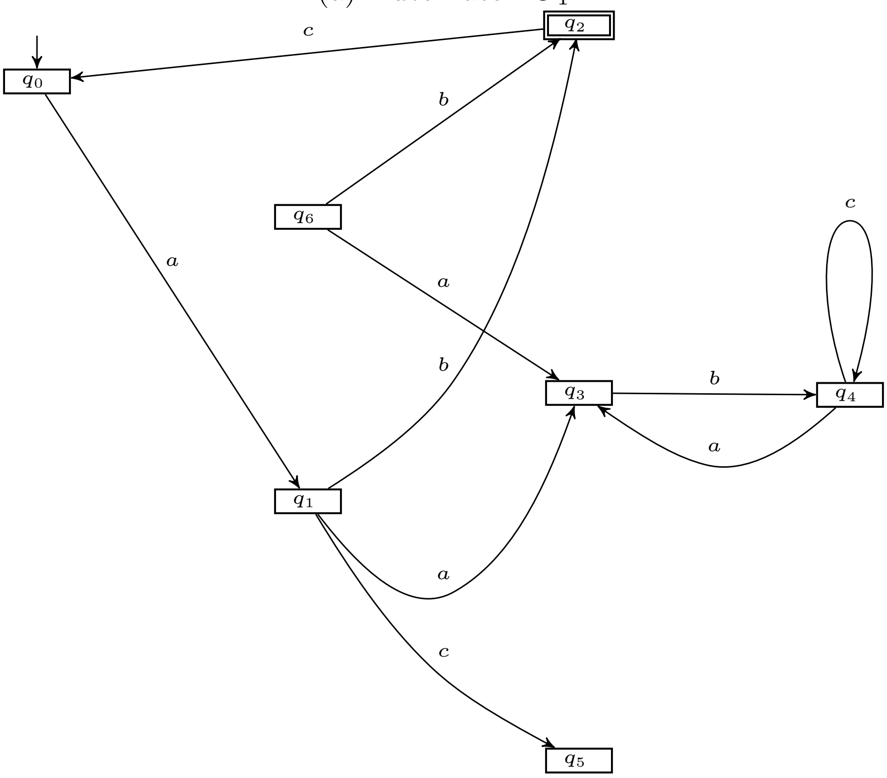
$\{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Is is required that the states are represented by rectangles, with left to right orientation and node and rank separation both equal to 1. The requirements can be met by running the following instructions.

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G1

X1 = [q0,q1,q2,q3,q4,q5,q6]
Sigma1 = [a,b,c]
X01 = [q0]
Xm1 = [q2]
T1 =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G1 = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G_1$')

# Redefining graphical properties

G1.setgraphic(style='verifier',1,1,direction='LR')

draw(G1, 'figure')
```

- See also: Drawing a State Transition Diagram



(a) Automaton G_1



(b) Automaton G_1 with redefined graphical properties.

Figure 3.41: Example of redefining graphical properties.

3.4.5 Lexicographical Features

- Purpose
  These operations search the finite state automaton and provide information concerning the order of appearance of the states, a string mapping, and a number mapping.

- Syntax
  The available instructions are disposed in Table 4.5.

Table 3.7: Syntax of the lexicographical features

Syntax	Brief description
$lexgraph\_dfs(G)$	list depth first searched states
$lexgraph\_alphamap(G)$	list the shortest paths to states
$lexgraph\_numbermap(G)$	list orderly enumerated states

  - Inputs
    The input parameter is the finite state automaton to be searched.
  - Output

    DFS - *Depth First Search* [2]: returns a list of the accessible states of the input.

    String Mapping returns a dictionary [3] associating the accessible states of G with the string composed by the shortest path conducting to each state.

    Number mapping returns a dictionary associating the accessible states of G with a string composed by a number that represents the order of appearance of the state in the lexicographical depth search.

- Description
  Let $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ denote a finite state automaton. There are cases when a closely search through the transitions is needed. For

[2] DFS can be defined as an algorithm that investigates tree or graph structures, taking some arbitrary node as the starting search point. It goes all the way until the end of the structure, and then starts backtracking.

[3] In Computer Science, a dictionary can be defined as a collection of associative (key,value) pairs composing an abstract data type.

these cases, a lexicographical depth search is ideal since it provides the exact order of the states, starting from the initial one, with respect to a hierarchy among the events. Other useful feature is a way of determining the shortest path to a given state, which can be provided by a lexicographical string mapping. Finally, the lexicographical number mapping is handy for its capability of orderly relate states regarding a depth first search.

- Example
  Consider the deterministic automaton $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ shown in Figure 3.42 $X = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma = \{a, b, c\}$, $f(q_0, a1) = \{q_1\}$, $f(q_1, b) = \{q_2\}$, $f(q_1, a) = \{q_3\}$, $f(q_1, c) = \{q_5\}$, $f(q_2, c) = \{q_0\}$, $f(q_3, b) = \{q_4\}$, $f(q_4, c) = \{q_4\}$, $f(q_4, a) = \{q_3\}$, $f(q_6, b) = \{q_2\}$, $f(q_6, a) = \{q_3\}$, $X_0 = \{q_0\}$, $X_m = \{q_2\}$. Using DESLab we can access the lexicographical features by writing the following instructions:

```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# automaton definition G

X = [q0,q1,q2,q3,q4,q5,q6]
Sigma = [a,b,c]
X0 = [q0]
Xm = [q2]
T =[(q0,a,q1), (q1,b,q2), (q1,a,q3), (q1,c,q5), (q2,c,q0),
(q3,b,q4), (q4,c,q4), (q4,a,q3), (q6,b,q2), (q6,a,q3)]
G = fsa(X1,Sigma1,T1,X01,Xm1,table,name='$G$')

# running a lexicographical depth search on G

lexgraph_dfs(G)

# running a lexicographical string mapping of G

lexgraph_alphamap(G)

# running a lexicographical number mapping of G
```

`lexgraph_numbermap(G)`

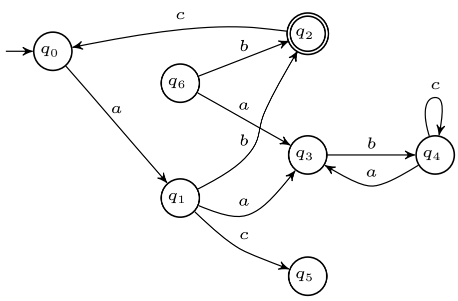


Figure 3.42: Example of running the lexicographical features.

- Console outputs
  The response to the example, which should be plugged in the console, is:

  lexgraph_dfs(G)
  >>>['q0', 'q1', 'q3', 'q4', 'q2', 'q5']

  lexgraph_alphamap(G)
  >>>'q1': 'a', 'q0': 'epsilon', 'q3': 'aa', 'q2': 'ab','q5': 'ac', 'q4': 'aab'

  lexgraph_numbermap(G)
  >>>'q1': 1, 'q0': 0, 'q3': 2, 'q2': 4, 'q5': 5, 'q4': 3

- See also: Graph Algorithms

3.5 Fault Diagnosis toolbox

3.5.1 Diagnoser function

- Purpose
  Generates the diagnoser automaton for fault diagnosis presented in [5].

- Syntax

$$diagnoser(G, \sigma_f, ret)$$

  - Input: Automaton of the class fsa, fault event, and variable *ret* which can be defined as:
    * GD (Default): Returns the diagnoser G_d;
    * GL: Returns the product between G and the labeling automaton A_l.
  - Output: Automaton defined at *ret*.

- Description
  Consider the automaton $G = (X, \Sigma, f, X_0, X_m)$, where the set of events $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$ is the disjoint union of the set of observable events, Σ_o, and the set of unobservable events, Σ_{uo}. The diagnoser automaton is defined as $G_d = Obs(G_l) = Obs(G \times A_l) = (X_d, \Sigma, f_d, x_{0_d})$, where the label automaton $A_l = (X_l, \Sigma_l, f_l, x_{0_l})$, shown in Figure 3.44, is a two-state automaton, such that $X_l = \{N, Y\}$, $x_{0_l} = N$,$\Sigma_l = \{\sigma_f\}$, $f_l(N, \sigma_f) = Y$, $f_l(Y, \sigma_f) = Y$.

- Example
  Consider the automaton $G = (X, \Sigma, f, X_0, X_m)$ shown in Figure 3.43, where $X = \{1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b, c, f, u\}$, $f(1, c) = 2$, $f(2, a) = 3$, $f(3, b) = 2$, $f(2, f) = 4$, $f(4, a) = 5$, $f(5, b) = 4$, $f(5, a) = 5$, $f(5, u) = 6$, $f(6, a) = 6$, $X_0 = 1$, $X_m = \emptyset$ and $\Sigma_o = \{a, b, c\}$. The automaton G_d and G_l, shown in Figure 3.45, are computed through the following commands.

```
from deslab import *

syms ('1 2 3 4 5 6 a b c f u')

# automaton definition G
X = [1, 2, 3, 4, 5, 6]
```

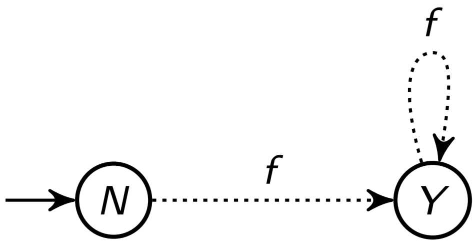


Figure 3.44: Automaton G_l in the subsection 3.5.1.

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

# Generate GD
Gd = diagnoser(G,f)
draw(Gd,'figure')

# Generate GL
Gl = diagnoser(G,f,'GL')
draw(Gl,'figure')
```

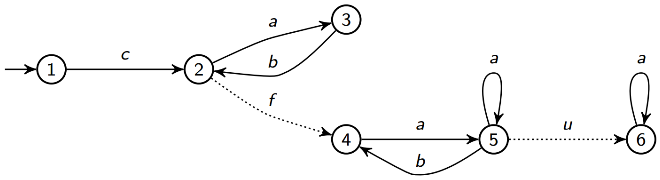


Figure 3.43: Automaton G of the example in subsection 3.5.1.

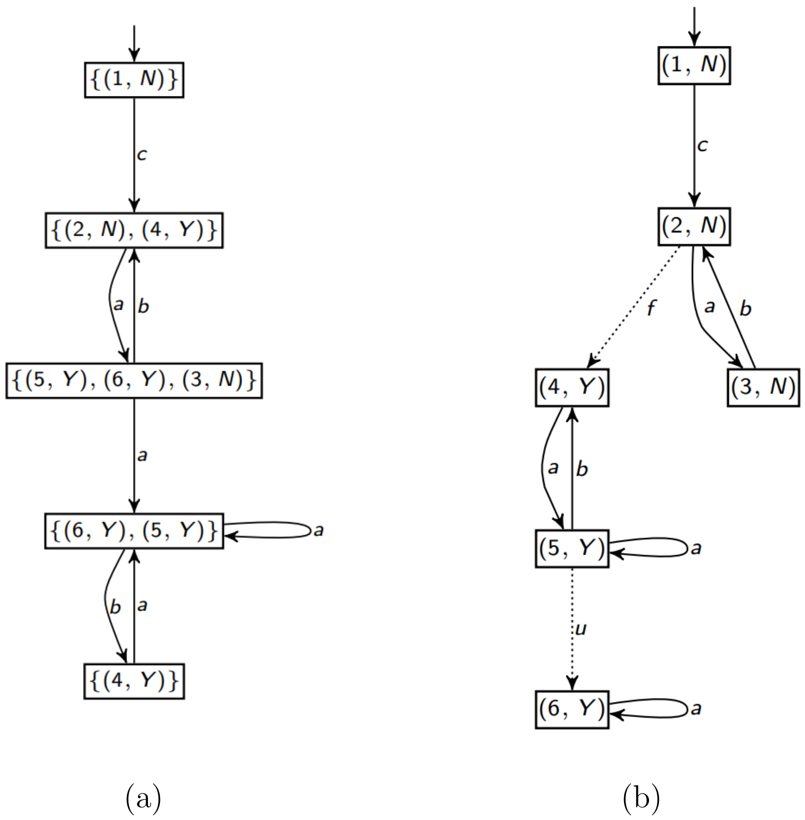


Figure 3.45: Diagnoser automaton G_d (a) and labeled automaton $G_l = G \times A_l$ (b), of the automaton in Figure 3.43.

3.5.2 Simplify function

- Purpose
  Rename the states in order to facilitate the visualization and treatment of variables.

- Syntax

$$simplify(G)$$

  - Input: Automaton of the class fsa.
  - Output: Automaton with renamed states.

- Description
  Consider the automaton $G = (X, \Sigma, f, X_0, X_m)$, where the set of events $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$ is the disjoint union of the set of observable events, Σ_o, and the set of unobservable events, Σ_{uo}. The diagnoser automaton is obtained by $G_d = diagnoser(G, \sigma_f)$ where σ_f is the fault event. The states of G_d are tuples with separators, the function goes through the states of the automaton, transforming its names into strings and eliminating separator characters as commas and parentheses.

- Example
  Consider the automaton $G = (X, \Sigma, f, X_0, X_m)$ shown in Figure 3.46, where $X = \{q1, q2, q3, q4, q5, q6\}$, $\Sigma = \{a, b, c, f, u\}$, $f(q1, c) = q2$, $f(q2, a) = q3$, $f(q3, b) = q2$, $f(q2, f) = q4$, $f(q4, a) = q5$, $f(q5, b) = q4$, $f(q5, a) = q5$, $f(q5, u) = q6$, $f(q6, a) = q6$, $X_0 = q1$, $X_m = \emptyset$ and $\Sigma_o = \{a, b, c\}$. The automaton G_d is renamed by the function *simplify*, both shown in Figure 3.47, by the following commands.

```
from deslab import *

syms ('q1 q2 q3 q4 q5 q6 a b c f u')

# automaton definition G
X = [q1, q2, q3, q4, q5, q6]
Sigma = [a, b, c, f, u]
X0 = [q1]
Xm = [ ]
T =[(q1, c, q2), (q2, a, q3), (q3, b, q2), (q2, f, q4),
    (q4, a, q5), (q5, b, q4), (q5, a, q5), (q5, u, q6),
    (q6, a, q6) ]
```

```
G = fsa (X, Sigma, T, X0, Xm, name ='$G$',Sigobs =[a, b, c])
draw(G,'figure')

# Generate GD
Gd = diagnoser(G,f)
draw(Gd,'figure')

# Symplify GD
Gs = simplify(Gd)
draw(Gs,'figure')
```

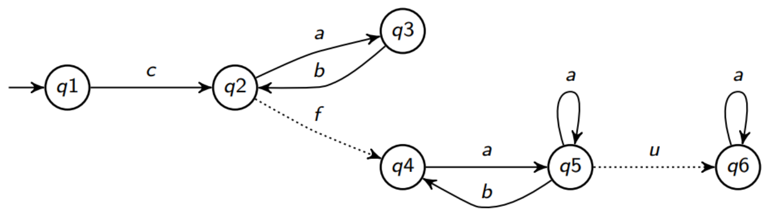
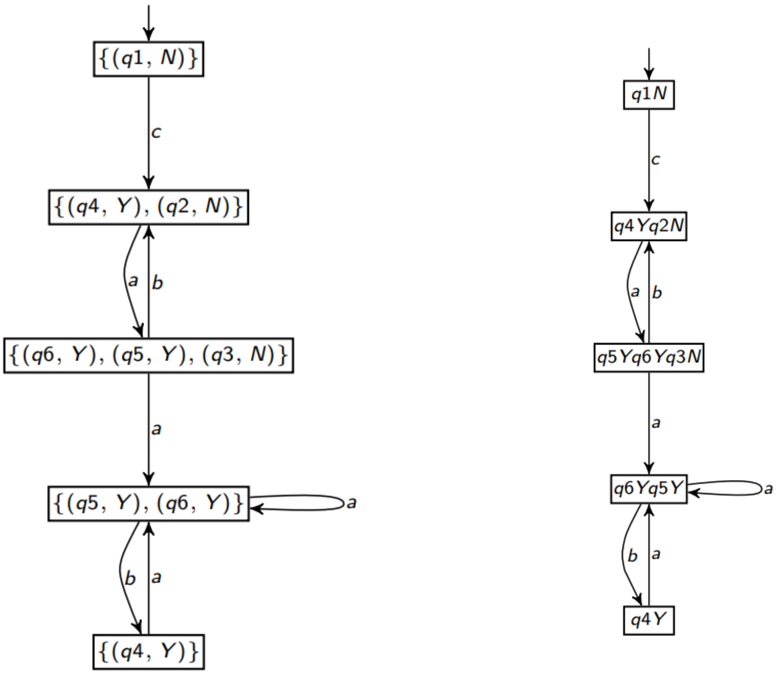


Figure 3.46: Automaton G of the example in subsection 3.5.2.



(a) (b)

Figure 3.47: Diagnoser automaton G_d (a) and simplified automaton (b), of the automaton in Figure 3.46.

3.5.3 Diagnosability test using SCC

- Purpose
  Generates the automaton G_{scc} presented in [6].

- Syntax

$$Gscc(G, \sigma_f, observable\_events)$$

  - Input: Automaton of the class fsa, list of fault events, and a list of observable events or a list of lists of observable events.
  - Output: Automaton G_{scc}.

- Description
  Consider the automaton $G = (X, \Sigma, f, x_0, X_m)$, such that $\Sigma_o \subseteq \Sigma$ is the set of observable events and σ_f is the list of fault events. There are three options for the variable *observable_events*, as follows.

  1. If it is not given as input, the function uses Σ_o from G to create G_d.
  2. If it is a list of events, the function considers it as the set Σ_o of G to create G_d.
  3. If it is a list of lists of events, the function creates N automata G_d, where N is the length of the list.

  After computing automaton G_d, the function executes the parallel composition $G_{scc} = G_d||G_l$, if *observable_events* is a list of events, or $G^N_{scc} = G_{d_1}||G_{d_2}||...||G_{d_N}||G_l$, if *observable_events* is a list of lists of events.

- Example
  Consider the automaton $G = (X, \Sigma, f, x_0, X_m)$ shown in Figure 3.48, where $X = \{1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b, c, f, u\}$, $f(1, c) = 2$, $f(2, a) = 3$, $f(3, b) = 2$, $f(2, f) = 4$, $f(4, a) = 5$, $f(5, b) = 4$, $f(5, a) = 5$, $f(5, u) = 6$, $f(6, a) = 6$, $x_0 = 1$, $X_m = \emptyset$ and $\Sigma_o = \{a, b, c\}$. The set of fault events is $\sigma_f = \{f\}$ and $observable\_events = \emptyset$. The automaton G_{scc}, shown in Figure 3.49, is computed through the following commands.

```
from deslab import *
syms ('1 2 3 4 5 6 a b c f u')

# automaton definition G
```

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

# Generate Gscc
G_scc = Gscc(G, f)
draw(G_scc,'figure')
```

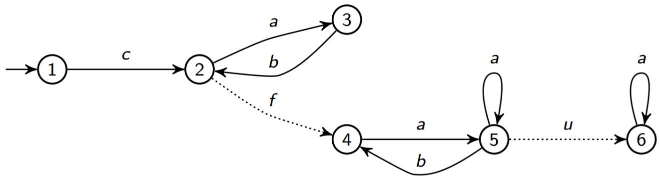


Figure 3.48: Automaton G of the example in subsection 3.5.3.

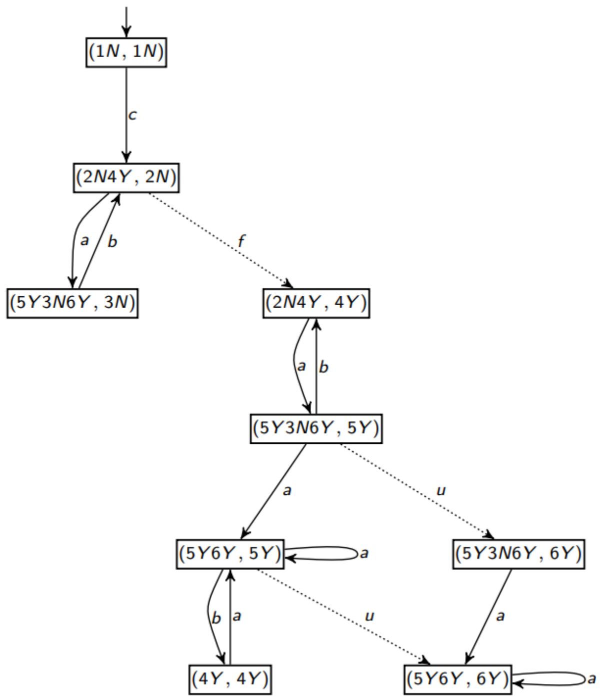


Figure 3.49: Resulting automaton G_{scc} of the automaton in Figure 3.48.

3.5.4 Polynomial Time Verification of Diagnosability

- Purpose
  Generates the verifier automaton G_v presented in [7], regarding fault diagnosis.

- Syntax

$$Gv(G, \sigma_f, observable\_events)$$

  - Input: Automaton of the class fsa, list of fault events, and list of observable events or a list of lists of observable events.
  - Output: Automaton G_v.

- Description
  Consider the automaton $G = (X, \Sigma, f, x_0, X_m)$, such that $\Sigma_o \subseteq \Sigma$ is the set of observable events and σ_f is the list of fault events. There are three options for the variable *observable_events*, as follows.

  1. If it is not given as an input, the function uses Σ_o from G to create G_n.
  2. If it is a list of events, the function considers it as the set Σ_o of G to create G_n.
  3. If it is a list of lists of events, the function creates N automata G_n, where N is the length of the list.

  The construction of the verifier proceeds as follows:

  1. Create $\Sigma_n = \Sigma \backslash \{\sigma_f\}$;
  2. Construct A_n with a single state N and self-loops for every event in Σ_n;
  3. Compute G_n as the product of G and A_n.The event set of G_n is then updated to Σ_n;
  4. Generate G_l by applying te diagnoser function $diagnoser(G, \sigma_f, 'GL')$. States in G_l that contain the label Y are marked.
  5. Generate $G_f = simplify(coac(G_l))$.
  6. For each set of observable events, the internal function R_i is used to rename non-observable events, producing automata G_{ni} for $i = 1, 2, .., N$;

7. G_v is constructed by performing the parallel composition of $G_{n1}, G_{n2}, ..., G_{nN}$ and G_f.

The automaton G_v is returned by the function.

- Example

  Let the automaton $G = (X, \Sigma,\ f,\ X_0,\ X_m,\ \Sigma_o)$ shown in Figure 3.50, where $X = \{0, 1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b, c, f, u\}$, $f(0, a) = 1$, $f(1, c) = 2$, $f(1, b) = 2$, $f(2, a) = 2$, $f(2, c) = 2$, $f(1, f) = 3$, $f(3, b) = 4$, $f(4, c) = 5$, $f(5, a) = 6$, $f(6, u) = 6$, $x_0 = 1$, $X_m = \emptyset$ and $\Sigma_o = \{a, b, c\}$. $\sigma_f = \{f\}$ and *observable_events* $= \emptyset$. The automaton G_v, Figure 3.51, is obtained through the following commands.

```
from deslab import *
syms ('0 1 2 3 4 5 6 a b c f u')

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

#Generate Gv
G_v = Gv(G, f)

draw(G_v,'figure')
```

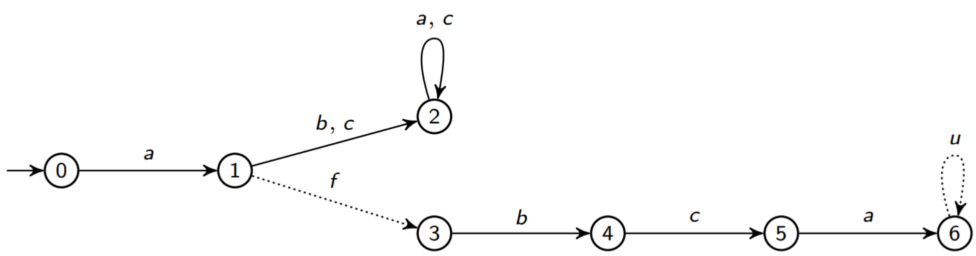


Figure 3.50: Automaton G of the example in subsection 3.5.4.

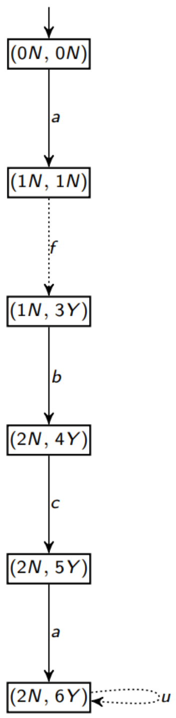


Figure 3.51: Verifier automaton G_v of G shown in Figure 3.50.

3.5.5 Diagnosability verifier

- Purpose
  Returns the (co)diagnosability of G for the given observable events in the chosen method.

- Syntax

$$is\_diagnosable(G, \sigma_f, observable\_events, method)$$

  - Input: Automaton of the class fsa, list of fault events, list of observable events or list of lists of observable events, and a computing method, which can be either *Gscc* or *Gv*.
  - Output: Boolean output about the diagnosability.

- Description
  Consider the automaton $G = (X, \Sigma, f, x_0, X_m)$, such that $\Sigma_o \subseteq \Sigma$ is the set of observable events and σ_f is the list of fault events. There are the following three options for the set *observable_events*.

  1. If it is not given as an input, the function uses Σ_o from G.
  2. If it is a list os events, the function considers it as the set Σ_o of G.
  3. If it is a list of lists of events, the function considers that there are N automata G, namely G_i for $i = 1, \ldots, N$, where N is the number of lists and Σ_{o_i} is the observable events of each G_i.

  The *method* input variable chooses which function is used, either *Gscc*, subsection 3.5.3, or *Gv*, subsection 3.5.4. The internal function *N_Y* is defined to assist in the verification. It takes a list of states and, for each state, checks which case from those presented in Table 3.8 it falls into. The function *N_Y* creates a list of zeros for each state passed, and if any of the states is of type (YN, Y), the value is changed to 1.

Table 3.8: States of G_{scc}

G_{scc}	G_d	G_l
(Y, Y)	Certain	Failure occurred: correct
(N, N)	Normal	Failure did not occur: normal
(YN, Y)	Uncertain	Failure occurred: certain
(YN, N)	Uncertain	Failure did not occur: normal

The nontrivial strongly connected components of the automaton G_{scc} are identified using functions `strconncomps` and `node_with_selfloops`, from NetworkX. The list of computed states is passed to the internal function N_Y. If no state of type (YN, Y) is identified, the function `is_diagnosable` returns `False`; otherwise, it returns `True`.

If the chosen method is Gv, the same verification of strongly connected components and components with self-loops done in G_{scc} method is performed on the verifier automaton G_v, obtained using function Gv.

The function returns `False` if there exists a nontrivial strongly connected component whose states' last component is labeled with Y and there is a transition between two of its states labeled with an event from the plant. Otherwise, it returns `True`.

- Example
  Consider the automaton $G = (X, \Sigma, f, X_0, X_m)$, shown in Figure 3.52, where $X = \{0, 1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b, c, f, u\}$, $f(0, a) = 1$, $f(1, c) = 2$, $f(1, b) = 2$, $f(2, a) = 2$, $f(2, c) = 2$, $f(1, f) = 3$, $f(3, b) = 4$, $f(4, c) = 5$, $f(5, a) = 6$, $f(6, u) = 6$, $x_0 = 1$, $X_m = \emptyset$ and $\Sigma_o = \{a, b, c\}$. Additionally, $\sigma_f = \{f\}$, *observable_events* $= \emptyset$ and *method* is Gv. The answer about the diagnosability is obtained through the following commands.

```
from deslab import *
syms ('0 1 2 3 4 5 6 a b c f u')

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

#Print the diagnosability property
print(is_diagnosable(G, 'f',[[ a, b],[ a, c]], 'Gv'))
```

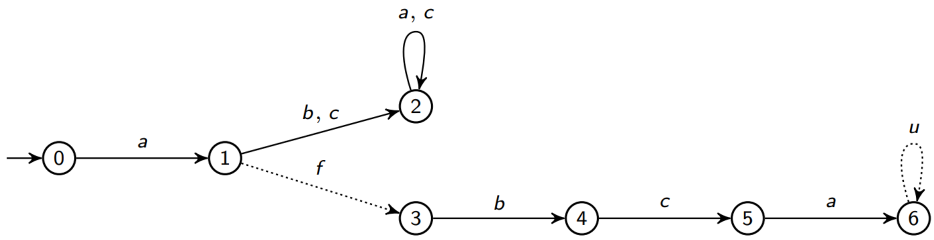


Figure 3.52: Automaton G of the example in subsection 3.5.5.

- Console outputs
  >>> True

3.6 Supervisory toolbox

3.6.1 Supremal Controllable Sublanguage

- Purpose
  Given automata G_1 and G_2, the function computes automaton H_i such that $L_m(H_i)$ is the supremal controllable sublanguage of $L_m(G_1)$ with respect to $L(G_2)$ and $\Sigma_{uc} = G_2.Sigma - G_2.Sigcon$.

- Syntax

$$supCont(G_1, G_2)$$

  - Input: Automata of the class fsa.
  - Output: Automaton Hi.

- Description
  Consider automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0_2}, X_{m_2}, \Sigma_{o_2}, \Sigma_{c_2})$, such that $\Sigma_{o2} \subseteq \Sigma$ is the set of observable events, $\Sigma_{c_2} \subseteq \Sigma_2$ is the set of controllable events and Γ_1 and Γ_2 are the sets of active transitions in G_1 and G_2, respectively. Automaton G_1 must be non-blocking for the function to work correctly.

  The function performs the following instructions to generate H_i:

  - Create G_m, which is a copy of G_2 with all states marked;
  - Compute $H_i = G_1 \times G_2$;
  - Set $\Sigma_{c_{Hi}} = \Sigma_{c_2}$ and $\Sigma_{o_{Hi}} = \Sigma_{o_2}$;
  - For each state (x, x_g) of H_i, check if the intersection between the set of active events in state x_g of G_m and the set of uncontrollable events is contained in the set of active events of state (x, x_g);
  - Remove any state of H_i that does not satisfy the previous condition;
  - Compute $H_i = trim(H_i)$;
  - Repeat the checking and trimming process until all states of H_i satisfy the condition.

- Example
  Let the automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0_2}, X_{m_2})$ in Figures 3.53 and 3.54, respectively, where

$X_1 = \{q_0, q_1, q_2, q_4\}$, $\Sigma_1 = \{a1, b1, d1\}$, $f_1(q_0, a1) = q_1$, $f_1(q_1, b1) = q_2$, $f_1(q_2, d1) = q_4$, $X_{0_1} = \{q_0\}$, and $X_{m_1} = \{q_4\}$, $X_2 = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma_1 = \{a1, b1, c1, d1\}$, $f_2(q_0, a1) = q_1$, $f_2(q_1, b1) = q_2$, $f_2(q_1, d1) = q_3$, $f_2(q_2, c1) = q_2$, $f_2(q_2, d1) = q_4$, $f_2(q_3, b1) = q_4$, $X_{0_2} = \{q_0\}$, and $X_{m_2} = \{q_4\}$. Considering $\Sigma_{c2} = \{a, c, d\}$, the set of controllable events. The automaton that generates H_i, Figure 3.55, is obtained through the following commands.

```
from deslab import *
syms (‘a b c d’)

# automaton definition G1
X1 = [1, 2, 3, 4]
Sig1 = [a, b]
Trans1 = [(1, a, 2),(1, b, 3),(3, a, 4)]
X01 = [1]
Xm1 = [1, 2, 4]
G1 = fsa ( X1, Sig1, Trans1, X01, Xm1, name =‘$G_1$’)

# automaton definition G2
X2 = [1, 2, 3, 4, 5]
Sig2 = [a, b, c, d]
Trans2 = [(1, a, 2), (1, b, 3), (2, c, 4), (3, a, 5), (5, d, 4)]
X02 = [1]
Xm2 = [1, 2, 4, 5]
G2 = fsa (X2, Sig2, Trans2, X02, Xm2, name =‘$G_2$’, Sigcon =[a, b, c])

#Generate Hi
SC = supCont(G1,G2)
draw(G1, ’figure’)
draw(G2, ’figure’)
draw(SC, ’figure’)
```

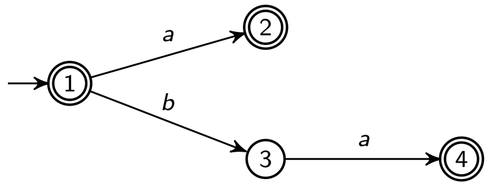


Figure 3.53: Automaton G_1 of the example in subsection 3.6.1.

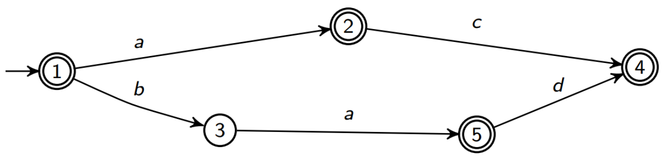


Figure 3.54: Automaton G_2 of the example subsection 3.6.1.

- Console output: Automaton H_i in Figure 3.55.

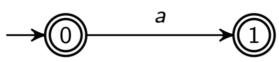


Figure 3.55: Automaton H_i that marks the supremal controllable sublanguage of $L(G_1)$, shown in Figure 3.53, with respect to $L(G_2)$, shown in Figure 3.54, and $\Sigma_{uc} = \{d\}$.

3.6.2 Controllability Verifier

- Purpose
  Given automata G_1 and G_2, the function verifies if the marked language $L_m(G_1)$ is controllable with respect to the language $L(G_2)$ and $\Sigma_{uc} = (G_2.Sigma - G_2.Sigcon)$.

- Syntax

$$is\_cont(G_1, G_2)$$

  - Input: Automata of the class fsa.
  - Output: Boolean output about the controllability.

- Description
  Consider automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0_2}, X_{m_2})$, such that $\Sigma_{o_2} \subseteq \Sigma$ is the set of observable events of G_2, $\Sigma_{c_2} \subseteq \Sigma_2$ is the set of controllable events of G_2, and Γ_1 and Γ_2 are the sets of active transitions in G_1 and G_2, respectively. The automaton G_1 must be non-blocking for the function to work correctly.

  The function performs the following instructions:

  - Create G_m, which is a copy of G_2 with all states marked;
  - Compute $H_i = G_1 \times G_2$;
  - Set $\Sigma_{c_{H_i}} = \Sigma_{c_2}$ and $\Sigma_{o_{H_i}} = \Sigma_{o_2}$;
  - For each state (x, x_g) of H_i, check if the intersection between the set of active events in state x_g of G_m and the set of uncontrollable events is contained in the set of active events of state (x, x_g);
  - If the condition is satisfied it will return True, otherwise, it will return False.

- Example
  Consider automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0_2}, X_{m_2})$ shown in Figures 3.65 and 3.66, respectively, where $X_1 = \{q_0, q_1, q_2, q_4\}$, $\Sigma_1 = \{a1, b1, d1\}$, $f_1(q_0, a1) = q_1$, $f_1(q_1, b1) = q_2$, $f_1(q_2, d1) = q_4$, $X_{0_1} = \{q_0\}$, and $X_{m_1} = \{q_4\}$, $X_2 = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma_1 = \{a1, b1, c1, d1\}$, $f_2(q_0, a1) = q_1$, $f_2(q_1, b1) = q_2$, $f_2(q_1, d1) = q_3$, $f_2(q_2, c1) = q_2$, $f_2(q_2, d1) = q_4$, $f_2(q_3, b1) = q_4$, $X_{0_2} = \{q_0\}$, and $X_{m_2} = \{q_4\}$. Let $\Sigma_{c_2} = \{a, c, d\}$ be the set of controllable events. The answer regarding the controllability is obtained through the following commands.

```
from deslab import *
syms ('a b c d')

# automaton definition G1
X1 = [1, 2, 3, 4]
Sig1 = [a, b]
Trans1 = [(1, a, 2),(1, b, 3),(3, a, 4)]
X01 = [1]
Xm1 = [1, 2, 4]
G1 = fsa ( X1, Sig1, Trans1, X01, Xm1, name ='$G_1$')

# automaton definition G2
X2 = [1, 2, 3, 4, 5]
Sig2 = [a, b, c, d]
Trans2 = [(1, a, 2), (1, b, 3), (2, c, 4), (3, a, 5), (5, d, 4)]
X02 = [1]
Xm2 = [1, 2, 4, 5]
G2 = fsa (X2, Sig2, Trans2, X02, Xm2, name ='$G_2$', Sigcon =[a, b, c])

#Print the controllability property
print(is_cont(G1,G2))

draw(G1, 'figure')
draw(G2, 'figure')
```

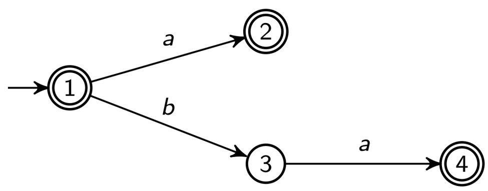


Figure 3.56: Automaton G_1 of the example in subsection 3.6.2.

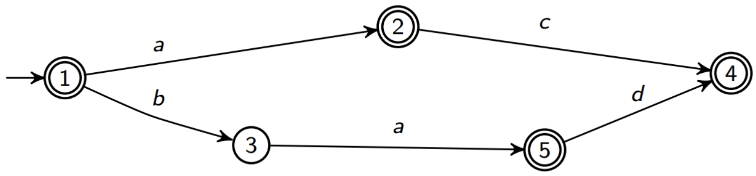


Figure 3.57: Automaton $G2$ of the example in subsection 3.6.2.

- Console outputs
  >>> False

3.7 Opacity Verification toolbox

3.7.1 Current-State Opacity Verifier

- Purpose
  Verify the current-state opacity property.

- Syntax

$$current\_state\_op(G, X_s, X_{ns})$$

  - Input: Automaton of the class fsa, set of secret states and set of non-secret states.
  - Output: Boolean output about the opacity property.

- Description
  Consider automaton $G = (X,\ \Sigma,\ f,\ \Gamma,\ X_0,\ X_m,\ \Sigma_o)$, such that $\Sigma_o \subseteq \Sigma$ is the set of observable events, $X_s \subseteq X$ is the set of secret states and $X_{ns} \subseteq X$ is the set of non-secret states. If X_{ns} is not provided then $X_{ns} = X \backslash X_s$. The function builds the observer of G, then, for every state $x \subseteq 2^X$ of the observer that contains a secret state, it checks if x also contains a non-secret state.

- Example
  Consider the automaton $G = (X,\ \Sigma,\ f,\ \Gamma,\ X_0,\ X_m)$, shown in Figure 3.58, where $X = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b, c, d\}$, $f(q_0, a) = q_1$, $f(q_1, b) = q_2$, $f(q_1, d) = q_3$, $f(q_2, c) = q_2$, $f(q_2, d) = q_4$, $f(q_3, b) = q_4$, $X_0 = q_0$, $X_m = \emptyset$, $\Sigma_o = \{a, b, c\}$, $X_s = \{q_3\}$, and $X_{ns} = \{q_4\}$. The function $current\_state\_op(G, X_s, X_{ns})$ is called through the following commands, and generates the observer of G, shown in Figure 3.59, and the answer about opacity.

```
from deslab import *

syms('q0 q1 q2 q3 q4 a1 b1 c1 d1')
table = [(a1,'a'),(b1,'b'),(c1,'c'),(d1,'d'),(q1,'q_1'),(q2,'q_2'),
(q3,'q_3'), (q0,'q_0'),(q4,'q_4')]

# automaton definition G
X = [q0,q1,q2,q3,q4]
Sigma = [a1,b1,c1,d1]
Sigmao = [a1,c1,d1]
```

```
X0 = [q0]
Xm = [ ]
T =[(q0,a1,q1), (q1,b1,q2), (q1,d1,q3), (q2,c1,q2), (q2,d1,q4),
(q3,b1,q4)]
G = fsa(X,Sigma,T,X0,Xm,table,Sigmao, name='$G$')

#Secret and non-secret states
xs = [q3]
xns = [q4]

# Print the current state opacity property
is_current_state_opaque = current_state_op(G, xs, xns)
print(is_current_state_opaque)
```

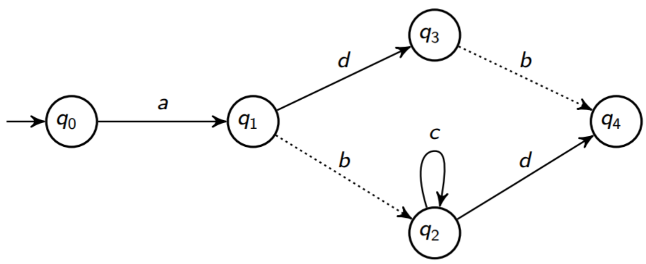


Figure 3.58: Automaton G of the example in the subsection 3.7.1.

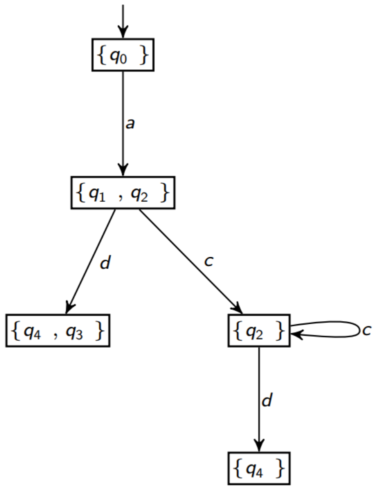


Figure 3.59: Observer of the automaton G in the Figure 3.58, $Obs(G)$.

- Console outputs
  >>> True

3.7.2 Initial State Opacity Verifier

- Purpose
  Verify the initial state opacity property.

- Syntax

$$initial\_state\_opac(G, X_s, X_{ns})$$

  - Input: Automaton of the fsa class, set of secret states and set of non-secret states.
  - Output: Boolean output about the opacity property.

- Description
  Consider automaton $G = (X, \Sigma, f, \Gamma, X_0, X_m)$, such that $\Sigma_o \subseteq \Sigma$ is the set of observable events, $X_s \subseteq X_0$ is the set of secret states, and $X_{ns} \subseteq X_0$ is the set of non-secret states. If X_{ns} is not provided then $X_{ns} = X \backslash X_s$. First, all initial states are placed as marked states, and all states of the automaton are considered initial. Then, all transitions are inverted, generating the reverse automaton G_r. Next, the observer of G_r is constructed. Finally, in all marked states, the existence of a secret state that is not accompanied by a non-secret state is verified. Additionally, a state is only considered non-secret if it belongs to the set of initial states.

- Example
  Consider automaton $G = (X,\ \Sigma,\ f,\ \Gamma,\ X_0,\ X_m)$, shown in Figure 3.60, where $X = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b, c, d\}$, $f(q_0, a) = q_1$, $f(q_1, b) = q_2$, $f(q_1, d) = q_3$, $f(q_2, c) = q_2$, $f(q_2, d) = q_4$, $f(q_3, b) = q_4$, $X_0 = \{q_0, q_3\}$, $X_m = \emptyset$, $\Sigma_o = \{a, b, c\}$, $X_s = \{q_3\}$, and $X_{ns} = \{q_0\}$. The function $initial\_state\_\ opac(G, X_s, X_{ns})$ is called through the following commands, and generates the reverse automaton, G_r (Figure 3.61), the observer of G_r (Figure 3.62), and the answer about opacity.

```
from deslab import *

syms('q0 q1 q2 q3 q4 a1 b1 c1 d1')
table = [(a1,'a'),(b1,'b'),(c1,'c'),(d1,'d'),
         (q1,'q_1'),(q2,'q_2'),(q3,'q_3'), (q0,'q_0'),
         (q4,'q_4')]

# automaton definition G
```

```
X = [q0,q1,q2,q3,q4]
Sigma = [a1,b1,c1,d1]
Sigmao = [a1,b1,c1]
X0 = [q0, q3]
Xm = [ ]
T =[(q0,a1,q1),(q1,b1,q2),(q1,d1,q3),(q2,c1,q2),
    (q2,d1,q4),(q3,b1,q4)]
G = fsa(X,Sigma,T,X0,Xm,table,Sigmao, name='$G$')

#Secret and non-secret states
xs = [q3]
xns = [q0]

# Print the initial state opacity property
is_initial_state_opaque = initial_state_opac(G, xs, xns)
print(is_initial_state_opaque)
```

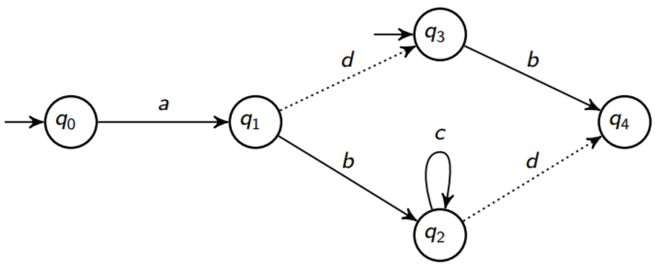


Figure 3.60: Automaton G of the example in subsection 3.7.2.

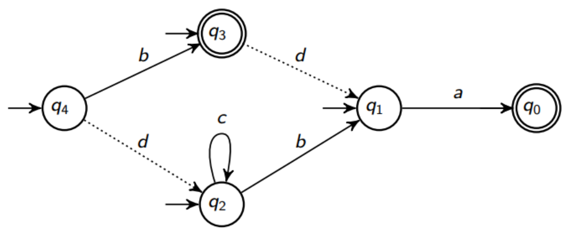


Figure 3.61: Reverse automaton, G_r, of the automaton of Figure 3.60.

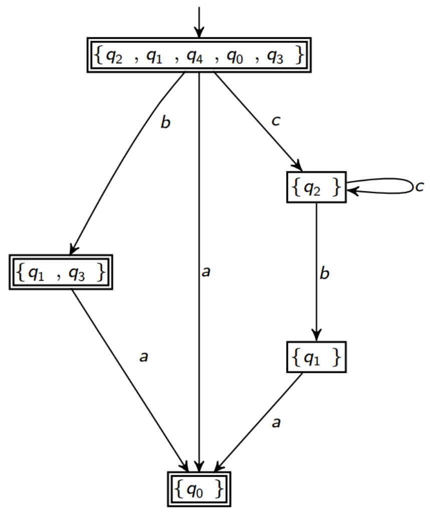


Figure 3.62: Observer of the automaton of Figure 3.61, $Obs(G_r)$.

- Console outputs
  >>> False

3.7.3 Initial-Final State Opacity Verifier

- Purpose
  Verify the existence of initial-final state opacity.

- Syntax

$$initial\_final\_state\_opac(G, X_{sp}, X_{nsp})$$

  - Input: Automaton of the class fsa, set of secret state pairs, and set of non-secret state pairs.
  - Output: Boolean output about the opacity property.

- Description
  Consider automaton $G = (X, \Sigma, f, \Gamma, X_0, X_m)$, such that $\Sigma_o \subseteq \Sigma$ is the set of observable events, and $X_{sp} \subseteq (X_0 \times X)$ and $X_{nsp} \subseteq (X_0 \times X)$ represent the secret and non-secret state pairs, respectively. If X_{nsp} is not provided then $X_{nsp} = (X \times X) \backslash X_{sp}$. First, a state network is created containing the possible initial and final state pairs from an observed transition. Then, it checks whether in any state there only exists a secret pair.

- Example
  Consider automaton $G = (X, \Sigma, f, \Gamma, X_0, X_m)$, shown in Figure 3.63, such that $X = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a1, b1, e1\}$, $f(q_0, a1) = q_0$, $f(q_1, b1) = q_0$, $f(q_1, e1) = q_3$, $f(q_2, a1) = q_1$, $f(q_3, b1) = q_1$, $f(q_0, e1) = q_2$, $X_0 = \{q_0, q_2\}$, and $X_m = \emptyset$. Let $\Sigma_o = \{a, b\}$, $X_{sp} = \{(q_2, q_1)\}$ be the set of observable events and $X_{nsp} = \{(q_2, q_2)\}$ be the set of non-secret state pairs. Function $initial\_final\_ state\_opac(G, X_{sp}, X_{nsp})$ is called through the following commands, internally generating the tree shown in Figure 3.64 and returning the answer about opacity.

```
from deslab import *

syms('q0 q1 q2 q3 q4 a b e')
table = [(q1,'q_1'),(q2,'q_2'), (q3,'q_3'),(q0,'q_0'),(q4,'q_4')]

# automaton definition G
X = [q0,q1,q2,q3]
Sigma = [a,b,e]
SigmaO = [a,b]
X0 = [q0,q2]
```

```
Xm = [ ]
T =[(q0,a,q0),(q0,e,q2),(q1,b,q0),(q2,a,q1),(q1,e,q3),(q3,b,q1)]
G = fsa(X,Sigma,T,X0,Xm,table, Sigma0,name='$G$')

draw(G, 'figure')

#Secret and non-secret pair os states
xps = [('q2','q1')]
xnps = [('q2','q2')]

# Print the inital-final state opacity property
initial_final_state_opac(G, xps, xnps)
print(initial_final_state_opac)
```

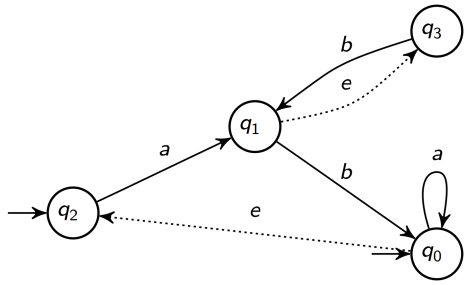


Figure 3.63: Automaton G of the example in subsection 3.7.3.

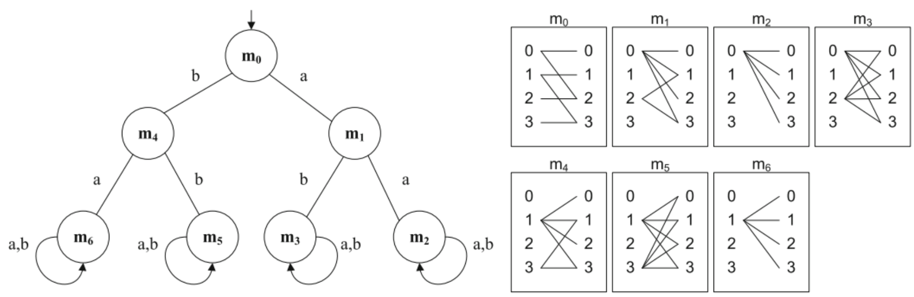


Figure 3.64: Tree created by the function *initial_final_ state_opac*, in subsection 3.7.3 (Source: [1]).

- Console outputs
  >>> False

3.7.4 Language-Based Opacity Verifier

- Purpose
  Verify the existence of language-based opacity.

- Syntax

$$language\_based\_opac(G1, G2, \Sigma_o)$$

  – Input: Automata of the class fsa and list of observable events.
  – Output: Boolean output about the opacity property.

- Description
  Consider two automata $G_s = (X_s, \Sigma, f_s, \Gamma_s, X_{0_s}, X_{m_s})$ and $G_{ns} = (X_{ns}, \Sigma, f_{ns}, \Gamma_s, X_{0_{ns}}, X_{m_{ns}})$, that mark the secret language and the non-secret language, respectively, and whose set of observable events is $\Sigma_o \subseteq \Sigma$.

  First, the function computes the observer of the input automata with respect to Σ_o, which generates $G_{s,o} = Obs(G_s, \Sigma_o)$ and $G_{ns,o} = Obs(G_{ns}, \Sigma_o)$. Then, the product composition between G_s and G_{ns} is computed, namely $G_p = G_s \times G_{ns}$, to verify if there exists an intersection between the secret language and the non-secret language. Finally, the resulting automaton, G_p, is compared with G_s. If G_p is identical to G_s, then, L_s is a subset of L_{ns} and it is opaque with respect to L_{ns} and Σ_o; otherwise, L_s is not opaque.

- Example
  Consider automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0_2}, X_{m_2})$, shown in Figures 3.65 and 3.66, respectively, where $X_1 = \{q_0, q_1, q_2, q_4\}$, $\Sigma_1 = \{a1, b1, d1\}$, $f_1(q_0, a1) = q_1$, $f_1(q_1, b1) = q_2$, $f_1(q_2, d1) = q_4$, $X_{0_1} = \{q_0\}$, and $X_{m_1} = \{q_4\}$, $X_2 = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma_1 = \{a1, b1, c1, d1\}$, $f_2(q_0, a1) = q_1$, $f_2(q_1, b1) = q_2$, $f_2(q_1, d1) = q_3$, $f_2(q_2, c1) = q_2$, $f_2(q_2, d1) = q_4$, $f_2(q_3, b1) = q_4$, $X_{0_2} = \{q_0\}$, and $X_{m_2} = \{q_4\}$. Let $\Sigma_o = \{a, c, d\}$ be the set of observable. Function $language\_based\_opac(G1, G2)$ is called through the following commands, and returns the answer about opacity.

```
from deslab import *

syms('q0 q1 q2 q3 q4 a1 b1 c1 d1 e1 f x y t1 t2')
table = [(a1,'a_1'),(b1,'b_1'),(c1,'c_1'), (d1,'d_1'), (q1,'q_1'),
```

```
        (q2,'q_2'), (q3,'q_3'),(q0,'q_0'),(q4,'q_4')]

# automaton definition G1
X = [q0,q1,q2,q4]
Sigma = [a1,b1,d1]
SigmaO = [a1,d1]
X0 = [q0]
Xm = [q4]
T =[(q0,a1,q1),(q1,b1,q2),(q2,d1,q4)]
G1 = fsa(X,Sigma,T,X0,Xm,table, SigmaO,name='$G_1$')

table1 = [(a1,'a_1'),(b1,'b_1'),(c1,'c_1'),(d1,'d_1'),(q1,'q_1'),
          (q2,'q_2'), (q3,'q_3'), (q0,'q_0'),(q4,'q_4'),(q5,'q_5')]

# automaton definition G2
X1 = [q0,q1,q2,q3,q4]
Sigma1 = [a1,b1,c1,d1]
X01 = [q0]
Xm1 = [q4]
T1 =[(q0,a1,q1),(q1,b1,q2),(q1,d1,q3),(q2,c1,q2),(q2,d1,q4),
     (q3,b1,q4)]
G2 = fsa(X1,Sigma1,T1,X01,Xm1,table1,name='$G_2$')

# Observable events
sigma_o = [a1, c1, d1]

# Print the language opacity property
is_language_based_opaque = language_based_opac(G1, G2, sigma_o)
print(is_language_based_opaque)
```

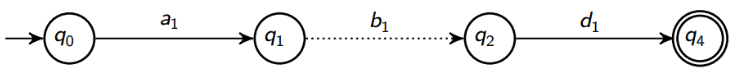


Figure 3.65: Automaton G_1 of the example in subsection 3.7.4.

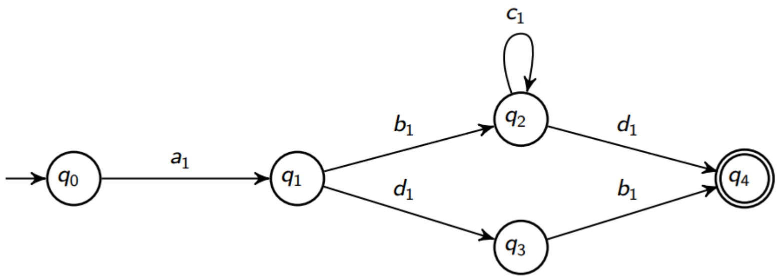


Figure 3.66: Automaton G_2 of the example in subsection 3.7.4.

- Console outputs
  >>> True

3.8 Opacity enforcement toolbox

3.8.1 Shuffling and Deletion Function

- Purpose
  Ensure, through modifications in the observation of events, if possible, the property of current-state opacity for an automaton that does not originally have it, using the method presented in [8, 9].

- Syntax

$$cso\_shuffle\_deletion\_function(G, SD, \Sigma_D, X_s, X_u)$$

  - Input: Automaton of the class fsa, event delay limit, SD, set of deletable events, Σ_D, list of secret states, X_s, and list of useful states, X_u.
  - Output: Automaton with the opacity enforcement strategy or an empty automaton.

- Description
  Consider automaton $G = (X, \Sigma, f, X_0, X_m)$. The algorithm for enforcing opacity consists of shuffling and/or deleting certain events based on specific conditions, forcing the intruder to always believe that it is in a non-secret state, $X_{ns} = X \backslash X_s$. These conditions are:

  1. The algorithm must release or delete all observed events if there is no continuation in the system.
  2. The algorithm must model the entire operation of the automaton, meaning it cannot prevent an event from occurring.
  3. The algorithm must have only one option after observing an event, which must follow the priority order: 1^{st}) release the event, 2^{nd}) wait for another event to occur, and 3^{rd}) delete the event.

  Automaton D contains all possibilities for shuffling the occurred events and observing the released or deleted ones. Its construction is done using the information about the number of steps (time) an event can be delayed until it must be released, given by the variable SD, and the events that can be deleted, Σ_D. The time that an event σ can be delayed is the maximum number of steps (occurred events) that the release of the observation of σ can be delayed.

Finally, the possibility of ensuring the utility property after enforcing opacity is verified. The goal is to ensure that an external observer can estimate specific states of the system, called useful states, X_u, in such a way that it is not revealed when the system is in a secret state.

- Example
  Consider automaton $G = (X, \Sigma, f, x_0)$, shown in Figure 3.67, where X $= \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, $\Sigma = \{a, b, c\}$, $f(0, a) = 1$, $f(0, b) = 6$, $f(1, b) = 2$, $f(1, c) = 4$, $f(2, c) = 3$, $f(4, b) = 5$, $f(6, c) = 7$, $f(7, a) = 8$, $x_0 = \{0\}$. Additionally, let $X_s = \{3\}$, $X_u = \{4\}$, $\Sigma_d = \emptyset$, and SD $= [(2, [a]), (0, [b]), (1, [c])]$.

  Through the following commands, it is possible to obtain the output of function *cso_shuffle_deletion_function*, which can be seen in Figure 3.68. Notice that the intruder always estimates state 8 when the system is in state 3, making the system opaque with respect to the current state. Additionally, state 4 can always be estimated.

```
# automaton definition G
X = [ 0, 1, 2, 3, 4, 5, 6, 7, 8]
E = [ a, b, c]
T = [ (0,a,1), (0,b,6), (1,b,2), (1,c,4), (2,c,3), (4,b,5), (6,c,7),
(7,a,8)]
X0 = [ 0 ]
Xs = [ 3 ]
Xu = [ 4 ]
G = fsa ( X, E, T, X0, name = '$G$')

# Event delay limit
SD = [(2,[a]), (0,[b]), (1,[c])]

# Deletable events
SigmaD = [ ]

# Generate the shuffling and deletion automaton
sdf = cso_shuffle_deletion_function(G, SD, SigmaD, Xs, Xu)
draw(G, sdf, 'figure')
```

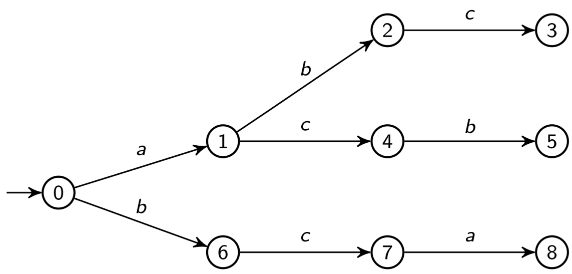


Figure 3.67: Automaton G of the example in subsection 3.8.1.

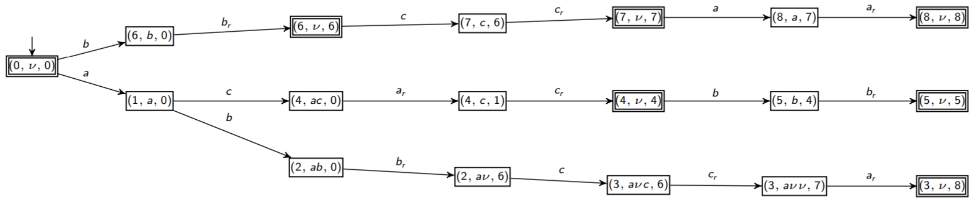


Figure 3.68: Shuffling and deletion automaton with the opacity enforcement strategy of G in Figure 3.67.

3.8.2 Edit Function

- Purpose
  Ensure, if possible, the property of current-state opacity for an automaton that does not originally have it, through modifications in the observation of events, using the method presented in [10].

- Syntax

$$edit\_function(G, X_s, Constraints)$$

  - Input: Automaton of the class fsa, list of secret states, and combinations of forbidden states.
  - Output: Automaton that guarantees opacity or an empty automaton.

- Description

  Consider automaton $G = (X, \Sigma, f, X_0, X_m, \Sigma_o)$, where the set of secret states is $X_s \subseteq X$. First the function generates the observer automaton $\mathscr{E}$ that represents the intruder. Then, two automata are generated from $\mathscr{E}$: the desired estimator $\mathscr{E}^d$, which generates the safe language, and the feasible estimator $\mathscr{E}^f$, which includes all possible edit choices, such as inserting or erasing events.

  The *unfolded verifier* V_u is created using V, which is computed as the parallel composition between $\mathscr{E}^d$ and $\mathscr{E}^f$. V_u represents a two-player game structure where the system and the edit function take turns: the system generates observable events and the edit function modify them. The states of V_u are divided into Y and Z, where Y states represents the system states so it is what the intruder observe and Z states represents the edit function states, where the edit function decide if it will insert or erase events.

  Stating from V initial state, an observable event is chosen, and the system transition to a Z state. From the Z state, the edit function decide the possibles edit choices (insertion or erasure), and for each edit choice, a Y state is reached. The algorithm then returns to one of the new Y states, and the process continues until there are no more Y states and Z states to verify. To prevent infinite loops, loops are not inserted and if a Y state has already appeared, the expansion stops. Using V_u, the Y states that do not satisfy the edit constraints (which are given to the function as forbidden combinations) and Z states where no outgoing transition is defined are removed, resulting in the All Edit Structure under constraints (AES_c).

The last automaton created is the AES_t which uses AES_c to generate all paths from the initial state to a final state. Each Y state is labeled with all previous system events and edit function modifications that occurred up to that state, and each Z state is labeled with the previous event. Lastly, the paths in AES_t are verified and classified into safe and unsafe behaviors. If for each unsafe path there exists a safe path with the same behavior under the effect of an edit function so the AES_t is returned; otherwise, a empty automaton is returned.

- Example

  Consider automaton $G = (X, \Sigma, f, \Gamma, X_0, X_m)$, shown in Figure 3.69, where $X = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $\Sigma = \{a, b, c, d\}$, $f(q_0, d) = q_1$, $f(q_1, a) = q_2$, $f(q_2, b) = q_3$, $f(q_3, c) = q_0$, $f(q_0, b) = q_5$, $f(q_0, a) = q_4$, $f(q_4, b) = q_5$, $f(q_5, c) = q_0$, $X_0 = \{q_0\}$, and $X_m = \emptyset$. The secret state is $X_s = \{q_5\}$, and $Const = \emptyset$. After creating all possibilities, the algorithm will remove the undesired states and verify the mapping, returning the automaton in figure 3.70.

  The editing function works in such a way that it retains information until it is possible to add information so that the intruder does not reach the secret state. Thus, following the example in figure 3.70, after the occurrence of the event 'b', the function will release the observation 'da' before releasing the event 'b', thereby maintaining the system's opacity. In the case of the event sequence 'dabc', the function will not add anything before the events, as the system is always on a safe path.

```
from deslab import *

syms('q0 q1 q2 q3 q4 a b c d e')
table = [(a,'a'),(b,'b'),(c,'c'),(d,'d'),(e,'e'),(1,'1'),
         (4,'4'),(5,'5'),(2,'2'),(3,'3'),(0,'0')]

# automaton definition G
X = [0,1,2,3,4,5]
Sigma = [a,b,c,d]
X0 = [0]
Xm = [ ]
T =[(0,d,1),(0,a,4),(0,b,5),(1,a,2),(2,b,3),(3,c,0),(4,b,5),
    (5,c,0)]
```

```
G = fsa(X,Sigma,T,X0,Xm,table,name='$G$')

#Secret state
x_secret = [5]

#Generate the edit automaton
G_ef = edit_function(G,x_secret)
draw(G, 'figure')
draw(G_ef, 'figure')
```

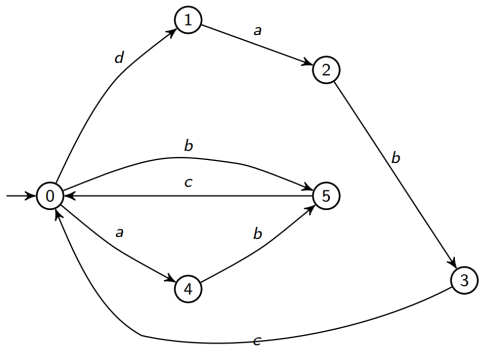


Figure 3.69: Automaton G of subsection 3.8.2.

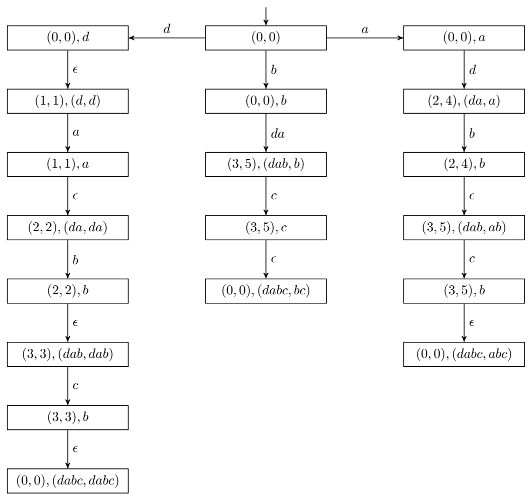


Figure 3.70: Edit automaton.

3.9 Time-Interval Automaton operations toolbox

3.9.1 Time-Interval Automaton

- Purpose
  Create a time-interval automaton (TIA) in DESLab.

- Syntax

$$tia(G, \mu)$$

  - Input: Automaton of the class fsa and a dictionary that associates a time interval with each transition.
  - Output: Tuple (G, μ), which is used by other functions involving TIA as a time interval automaton.

- Description
  A TIA $G_T = (X, \Sigma, f, x_0, X_m, \mu)$ is represented in DESLab by the pair (G, μ). The first element of the pair is the untimed automaton $G = (X, \Sigma, f, x_0, X_m)$, where X is the set of states, Σ is the set of events, $f : X \times \Sigma \rightarrow X$ is the transition function, x_0 is the initial state, and $X_m \subseteq X$ is the set of marked states. The second element of the pair is the dictionary μ that associates each transition in T with a time interval. For the representation of intervals, the *portion* library from *Python* is used, imported in the code as P.

- Example
  Consider TIA $G_T = (X, \Sigma, f, x_0, X_m, \mu)$, shown in Figure 3.71, where $X = \{0, 1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b, u\}$, $f(0, u) = 1$, $f(0, a) = 4$, $f(1, a) = 2$, $f(2, b) = 3$, $f(4, u) = 5$, $f(5, b) = 6$, $f(5, b) = 3$, $x_0 = 0$, $X_m = \emptyset$, and $\mu(0, u, 1) = [0, 2]$, $\mu(0, a, 4) = [1, 3.5]$, $\mu(1, a, 2) = [1, 3]$, $\mu(2, b, 3) = [2, 3]$, $\mu(4, u, 5) = [0.5, 1.5]$, $\mu(5, b, 6) = [2, 4]$, $\mu(5, b, 3) = [1, 3]$. It is possible to create this TIA through the following commands.

```
from deslab import *

syms('q0 q1 q2 q3 q4 a1 b1 c1 d1 e1 f x y t1 t2 u')

# automaton definition G_T
Xt = [0, 1, 2, 3, 4, 5, 6]
```

```
Et =[a,b,u]
sigobst = [a,b,u]
X0t = [0]
Xmt = [ ]
Tt = [(0,u,1),(0,a,4),(1,a,2),(2,b,3),(4,u,5),(5,b,6),(5,b,3)]

mut = {(0,u,1): P.closed(0,2),
      (0,a,4): P.closed(1,3.5),
      (1,a,2): P.closed(1,3),
      (2,b,3): P.closed(2,3),
      (4,u,5): P.closed(0.5,1.5),
      (5,b,6): P.closed(2,4),
      (5,b,3): P.closed(1,3)
      }

G = fsa(Xt,Et,Tt,X0t,Xmt,Sigobs=sigobst,name="$G_T$")
GT = tia(G,mut)
ti_draw(GT)
```

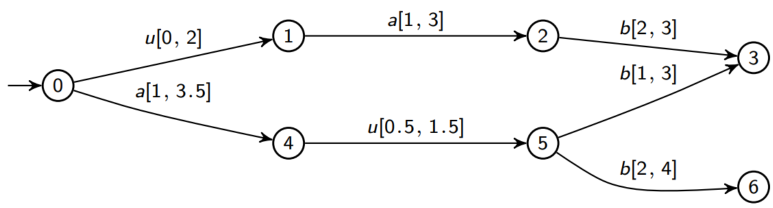


Figure 3.71: TIA G_T of the example in subsection 3.9.1.

- See also: Drawing a TIA State Transition Diagram, in subsection 3.9.2.

3.9.2 Drawing a TIA State Transition Diagram

- Purpose
  Create the state transition diagram of a time-interval automaton.

- Syntax

$$ti\_draw(G_T)$$

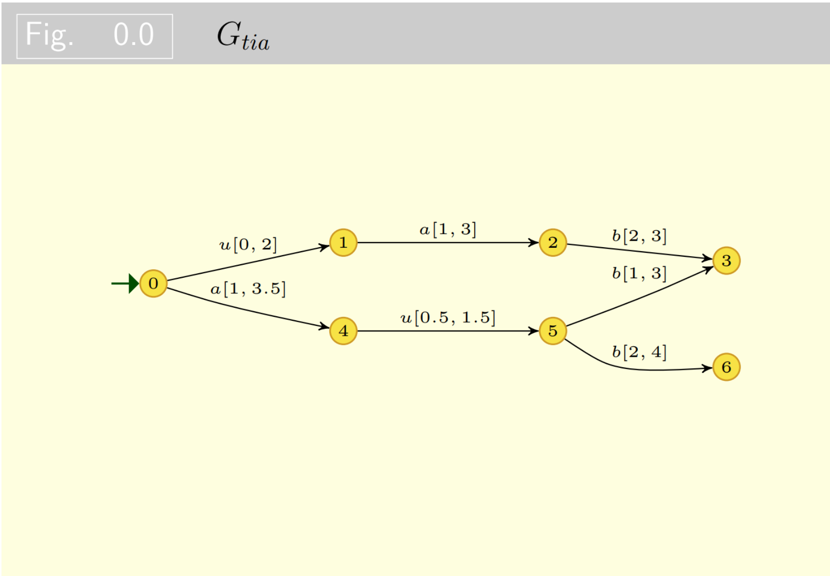
  - Input: Time-interval automaton G_T and display mode if desired (Figure, Figurecolor, or Beamer).
  - Output: The state transition diagram is generated according to the specified display mode. If no mode is defined as input, the diagram is automatically rendered in *Beamer* style.

- Description
  The graphical representation of time-interval automaton $G_T = (X, \Sigma, f, x_0, X_m, \mu)$ is called the state transition diagram, where circles represent states and arrows labeled with symbols represent transitions. The initial state is indicated by a small arrow pointing to it, marked states are represented with a double circle, and unobservable events are represented by dashed arrows. Additionally, transitions consist of the event and the corresponding time interval.

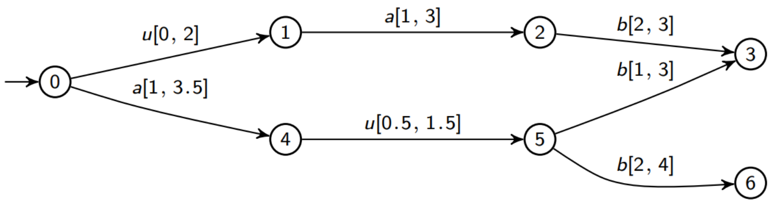
- Example
  Consider TIA $G_T = (X, \Sigma, f, x_0, X_m, \mu)$, shown in Figure 3.72a, where $X = \{0, 1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b, u\}$, $f(0, u) = 1, f(0, a) = 4, f(1, a) = 2, f(2, b) = 3, f(4, u) = 5, f(5, b) = 6, f(5, b) = 3$, $x_0 = 0$, $X_m = \emptyset$ and $\mu(0, u, 1) = [0, 2]$, $\mu(0, a, 4) = [1, 3.5]$, $\mu(1, a, 2) = [1, 3]$, $\mu(2, b, 3) = [2, 3]$, $\mu(4, u, 5) = [0.5, 1.5]$, $\mu(5, b, 6) = [2, 4]$, $\mu(5, b, 3) = [1, 3]$. The display modes can be viewed using the following commands and the results are shown in Figure 3.72.

```
from deslab import *
syms('u')

# automaton definition G_T
Xt = [0, 1, 2, 3, 4, 5, 6]
Et =[a,b,u]
sigobst = [a,b,u]
X0t = [0]
Xmt = [ ]
```

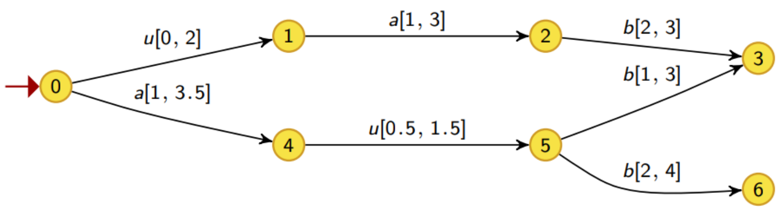
```
Tt = [(0,u,1),(0,a,4),(1,a,2),(2,b,3),(4,u,5),(5,b,6),(5,b,3)]

mut = {(0,u,1): P.closed(0,2),
        (0,a,4): P.closed(1,3.5),
        (1,a,2): P.closed(1,3),
        (2,b,3): P.closed(2,3),
        (4,u,5): P.closed(0.5,1.5),
        (5,b,6): P.closed(2,4),
        (5,b,3): P.closed(1,3)
      }

G = fsa(Xt,Et,Tt,X0t,Xmt,Sigobs=sigobst,name="$G_T$")
GT = tia(G,mut)

# G_T in bearmer format
ti_draw(GT, 'beamer')

# G_T in figure format
ti_draw(GT, 'figure')

# G_T in figurecolor format
ti_draw(GT, 'figurecolor')
```



(a) *Beamer* format.



(b) *Figure* format.



(c) *Figurecolor* format.

Figure 3.72: Automaton G_T of the example in subsection 3.9.2.

3.9.3 Detectable Path of a TIA

- Purpose
  Obtain all the possible detectable paths that start from any state of a time-interval automaton.

- Syntax

$$DP(G_T, state)$$

  - Input: Time-interval automaton and starting state.
  - Output: List with the sequence of transitions of the resulting paths.

- Description
  Consider TIA $G_T = (X, \Sigma, f, x_0, X_m, \mu)$, where $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$, $\Sigma_o \subseteq \Sigma$ is the set of observable events and Σ_{uo} is the set of unobservable events. A detectable path that starts at a state $y \in X$ consists of $n \in \mathbb{N}$ transitions, where the first $n-1$ transitions are unobservable events and the last one is labeled with an observable event.

- Example
  Consider TIA $G_T = (X, \Sigma, f, x_0, X_m, \mu)$, represented in Figure 3.73, where $X = \{0, 1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b, u\}$, $\Sigma_o = \{a, b\}$, $f(0, u) = 1$, $f(0, a) = 4, f(1, a) = 2, f(2, b) = 3, f(4, u) = 5, f(5, b) = 6, f(5, b) = 3.$, $x_0 = 0$, $X_m = \emptyset$ and $\mu(0, u, 1) = [0, 2]$, $\mu(0, a, 4) = [1, 3.5]$, $\mu(1, a, 2) = [1, 3]$, $\mu(2, b, 3) = [2, 3]$, $\mu(4, u, 5) = [0.5, 1.5]$, $\mu(5, b, 6) = [2, 4]$, $\mu(5, b, 3) = [1, 3]$. The set of detectable paths can be computed with the following commands.

```
from deslab import *
syms('q0 q1 q2 q3 q4 a1 b1 c1 d1 e1 f x y t1 t2 u')

# automaton definition G_T
Xt = [0, 1, 2, 3, 4, 5, 6]
Et =[a,b,u]
sigobst = [a,b]
X0t = [0]
Xmt = []
Tt = [(0,u,1),(0,a,4),(1,a,2),(2,b,3),(4,u,5),(5,b,6),(5,b,3)]

mut = {(0,u,1): P.closed(0,2),
```

```
        (0,a,4): P.closed(1,3.5),
        (1,a,2): P.closed(1,3),
        (2,b,3): P.closed(2,3),
        (4,u,5): P.closed(0.5,1.5),
        (5,b,6): P.closed(2,4),
        (5,b,3): P.closed(1,3)
      }

G = fsa(Xt,Et,Tt,X0t,Xmt,Sigobs=sigobst,name="$G_T$")
GT = tia(G,mut)

# Print the detectable path from state 4
print(DP(GT, 4))
```

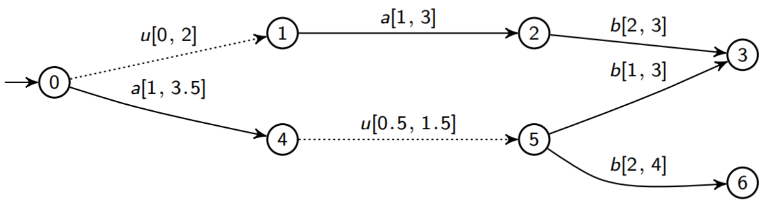


Figure 3.73: Automaton G_T of the example in subsection 3.9.3.

- Console outputs
  $>>> [[(4,'u',5),(5,'b',3)],[(4,'u',5),(5,'b',6)]]$

3.9.4 Projection of a TIA

- Purpose
  Compute the projection TIA of a time-interval automaton.

- Syntax

$$ti\_proj(G_T)$$

  - Input: Time-interval automaton.
  - Output: Resulting projected TIA.

- Description
  Given a time-interval automaton $G_T = (X, \Sigma, f, x_0, X_m, \Sigma_o, \mu)$, the detectable paths from the initial state are computed, then states that have an unobservable transition are removed from the automaton, and the time intervals of these transitions are added to the last observable transition. This process is repeated for the subsequent states of the resulting transitions.

- Example
  Consider the TIA $G_T = (X, \Sigma, f, x_0, X_m, \mu)$, shown in Figure 3.74, where $X = \{0, 1, 2, 3, 4, 5, 6\}$, $\Sigma = \{a, b, u\}$, $\Sigma_o = \{a, b\}$, $f(0, u) = 1$, $f(0, a) = 4, f(1, a) = 2, f(2, b) = 3, f(4, u) = 5, f(5, b) = 6, f(5, b) = 3.$, $x_0 = 0$, $X_m = \emptyset$ and $\mu(0, u, 1) = [0, 2]$, $\mu(0, a, 4) = [1, 3.5]$, $\mu(1, a, 2) = [1, 3]$, $\mu(2, b, 3) = [2, 3]$, $\mu(4, u, 5) = [0.5, 1.5]$, $\mu(5, b, 6) = [2, 4]$, $\mu(5, b, 3) = [1, 3]$. The projection of G_T, shown in Figure 3.75, is computed with the following commands.

```
from deslab import *
syms('u')

# automaton definition G_T
Xt = [0, 1, 2, 3]
Et =[a,b,u]
sigobst = [a,b]
X0t = [0]
Xmt = [ ]
Tt = [(0,u,1),(1,a,2),(2,b,3)]

mut = {(0,u,1): P.closed(0,2),
        (1,a,2): P.closed(1,3),
```

```
        (2,b,3): P.closed(2,3)
      }

G = fsa(Xt,Et,Tt,X0t,Xmt,Sigobs=sigobst,name="$G_T$")
GT = tia(G,mut)
ti_draw(GT, 'figure')

# Generate the projection of G_T
d = ti_proj(GT)
ti_draw(d, 'figure')
```

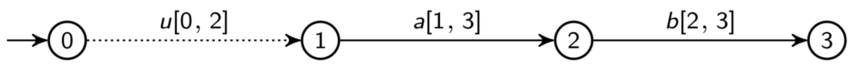


Figure 3.74: TIA G_T of the example in subsection 3.9.4.

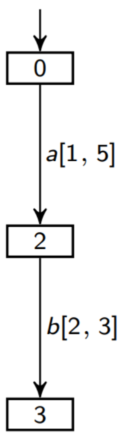


Figure 3.75: Projection TIA of TIA G_T shown in Figure 3.74.

3.9.5 Deterministic Equivalent TIA of a TIA

- Purpose
  Remove nondeterminism from a TIA.

- Syntax

$$ti\_equi\_det(G_T)$$

  - Input: Nondeterministic time-interval automaton.
  - Output: Deterministic equivalent time-interval automaton.

- Description
  A time-interval automaton $G_T = (X, \Sigma, f, X_0, X_m, \mu)$ is considered nondeterministic if at least one of the following conditions is met:

  (*i*) There is more than one initial state, $X_0 \subseteq X$

  (*ii*) There is more than one transition originating from a state, labeled with the same event, and whose time intervals are not disjoint, that is, $(\exists x, y, y' \in X)(\exists \sigma \in \Gamma(x))[(|f(x,\sigma)| > 1) \wedge (\mu(x,\sigma,y) \cap \mu(x,\sigma,y') \neq \emptyset)]$

  (*iii*) G_T has at least one transition labeled by ε.

  Condition *(iii)* must be resolved using the projection function, $ti\_proj(G_T)$, before using function ti_equi_det. This function transforms the set of initial states into a single state composed of X_0. Each state of the resulting automaton is a subset of X. For each new state, x_{det}, of the deterministic equivalent, the transitions of G_T labeled by the same event that originate from each $x \in x_{det}$ are analyzed. The set of time intervals of the considered transitions is then partitioned into a disjoint set, in order to remove this nondeterministic characteristic. Each time interval I of the new disjoint set determines a new transition in the deterministic equivalent, whose destination state is a subset of X composed of the destination states of the transitions that generated I in G_T.

- Example
  Consider the TIA $G_T = (X, \Sigma, f, x_0, X_m, \mu)$, shown in Figure 3.76, where $X = \{0,1,2,3,4,5,6\}$, $\Sigma = \{a,b,c\}$,$f(0,c) = 1, f(0,a) = 4, f(1, a) = 2, f(2,b) = 3, f(4,c) = 5, f(5,b) = 6, f(5,b) = 3.$, $x_0 = 0$, $X_m = \emptyset$ and the time-interval function $\mu(0,c,1) = [0,2]$, $\mu(0,a,4) = [1,3.5]$,

$\mu(1,a,2) = [1,3]$, $\mu(2,b,3) = [2,3]$, $\mu(4,c,5) = [0.5,1.5]$, $\mu(5,b,6) = [2,4]$, $\mu(5,b,3) = [1,3]$. The deterministic equivalent of G_T, shown in Figure 3.77, is computed with the following commands.

```
from deslab import *

# automaton definition G_T
Xt = [0, 1, 2, 3, 4, 5, 6]
Et =[a,b,c]
sigobst = [a,b,c]
X0t = [0,1]
Xmt = [ ]
Tt = [(0,c,1),(0,c,4),(1,a,2),(2,b,3),(4,c,5),(5,b,6),(5,b,3)]

mut = {(0,c,1): P.closed(0,2),
        (0,c,4): P.closed(1,3.5),
        (1,a,2): P.closed(1,3),
        (2,b,3): P.closed(2,3),
        (4,c,5): P.closed(0.5,1.5),
        (5,b,6): P.closed(2,4),
        (5,b,3): P.closed(1,3)
      }

G = fsa(Xt,Et,Tt,X0t,Xmt,Sigobs=sigobst,name="$G_T$")
GT = tia(G,mut)
ti_draw(GT, 'figure')

#Generate the deterministic equivalent of G_T
d = ti_equi_det(GT)
ti_draw(d, 'figure')
```

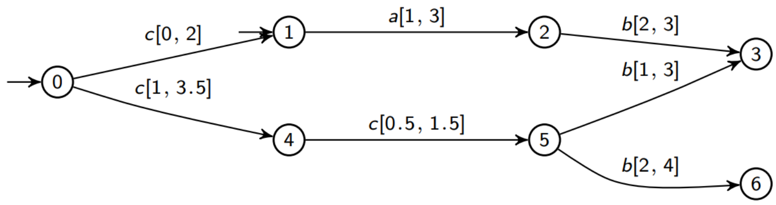


Figure 3.76: Nondeterministic TIA of the example in subsection 3.9.5.

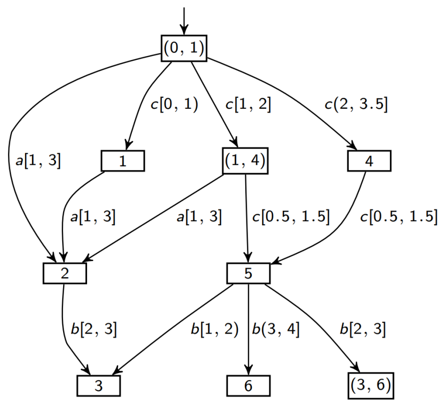


Figure 3.77: Deterministic equivalent TIA of the TIA shown in Figure 3.76.

3.9.6 Product Composition Between two TIAs

- Purpose
  Performs the product composition between two TIAs, where transitions synchronize on common events, and the intersection of their respective time intervals is computed.

- Syntax

$$ti\_product(G1, G2)$$

  - Input: Time-interval automata G_1 and G_2.
  - Output: TIA resulting from the product composition of the input automata.

- Description
  The product of two TIAs $G_{T_1} = (X_1, \Sigma_1, f_1, x_{01}, X_{m1}, \mu_1)$ and $G_{T_2} = (X_2, \Sigma_2, f_2, x_{02}, X_{m2}, \mu_2)$, where Γ_1 and Γ_2 are their active event set functions, respectively, is given by the TIA $G_{T_1} \times G_{T_2} = Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f_{1\times 2}, (x_{01}, x_{02}), X_{m1} \times X_{m2}, \mu_{1\times 2})$, such that for $x_1 \in X_1$ and $x_2 \in X_2$, we define $f_{1\times 2}((x_1, x_2), \sigma) = (f_1(x_1, \sigma), f_2(x_2, \sigma))$ if $\sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2)$ and $\mu_1(x_1, \sigma, y_1) \cap \mu_2(x_2, \sigma, y_2) \neq \emptyset$, or undefined otherwise, and $\mu_{1\times 2}((x_1, x_2), \sigma, (y_1, y_2)) = \mu_1(x_1, \sigma, y_1) \cap \mu_2(x_2, \sigma, y_2)$.

- Example
  Consider the TIA $G_{T_1} = (X_1, \Sigma_1, f_1, x_{01}, X_{m1}, \mu_1)$, shown in Figure 3.78, where $X_1 = \{0, 1, 2\}$, $\Sigma_1 = \{a, b\}$, $f_1(0, a) = 1, f_1(1, b) = 2$, $x_{01} = 0$, $X_{m1} = \emptyset$ and $\mu_1(0, a, 1) = [1, 3]$, $\mu_1(1, b, 2) = [0, 2]$, and TIA $G_{T_2} = (X_2, \Sigma_2, f_2, x_{02}, X_{m2}, \mu_2)$, shown in Figure 3.79, where $X_2 = \{0, 1, 2, 3, 4\}$, $\Sigma_2 = \{a, b, c\}$, $f_2(0, a) = 1, f_2(0, c) = 4, f_2(1, c) = 2, f_2(2, b) = 3$, $x_{02} = 0$, $X_{m2} = \emptyset$ and $\mu_2(0, a, 1) = [2, 4]$, $\mu_2(0, c, 4) = [0, 4]$, $\mu_2(1, c, 2) = [1, 3]$, $\mu_2(2, b, 3) = [1, 4]$. The product composition $G_{T_1} \times G_{T_2}$, depicted in Figure 3.80, is computed with the following commands.

```
from deslab import *

# automaton definition G1
Xt1 = [0, 1, 2]
Et1 =[a,b]
sigobst1 = [a,b]
X0t1 = [0]
Xmt1 = [ ]
```

```
Tt1 = [(0,a,1),(1,b,2)]

mut1 = \{(0,a,1): P.closed(1,3),
        (1,b,2): P.closed(0,2)
    \}

G1 = fsa(Xt1,Et1,Tt1,X0t1,Xmt1,Sigobs=sigobst1,name="$G_{1}$")
G_tia1 = tia(G1,mut1)
ti_draw(G_tia1, 'figure')

# automaton definition G2
Xt2 = [0, 1, 2, 3, 4]
Et2 =[a,b,c]
sigobst2 = [a,b,c]
X0t2 = [0]
Xmt2 = [ ]
Tt2 = [(0,a,1),(1,c,2),(2,b,3),(0,c,4)]

mut2 = {(0,a,1): P.closed(2,4),
        (1,c,2): P.closed(1,3),
        (2,b,3): P.closed(1,4),
        (0,c,4): P.closed(0,4)
    }

G2 = fsa(Xt2,Et2,Tt2,X0t2,Xmt2,Sigobs=sigobst2,name="$G_{2}$")
G_tia2 = tia(G2,mut2)
ti_draw(G_tia2,'figure')

# Product
ti_draw(ti_product(G_tia1,G_tia2),'figure')
```

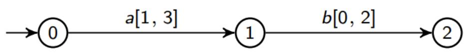


Figure 3.78: TIA G_1 of the example in subsection 3.9.6.

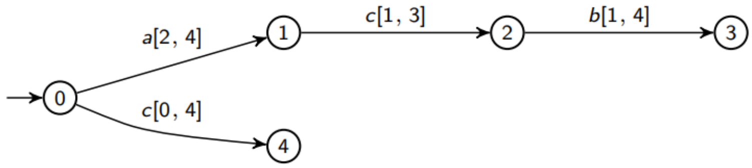


Figure 3.79: TIA G_2 of the example in subsection 3.9.6.

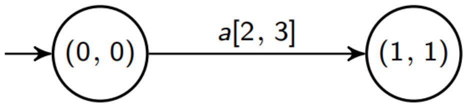


Figure 3.80: Resulting TIA from the product composition $G_1 \times G_2$, where G_1 and G_2 are the TIA in Figures 3.78 and 3.79, respectively.

3.9.7 Complement of a TIA

- Purpose
  Generates the TIA that marks the complement of the marked time-interval language of a given TIA.

- Syntax

  $$ti\_complement(G_T)$$

  - Input: Time-interval automaton G_T.
  - Output: Complement automaton of G_T.

- Description
  Given a time-interval automaton $G_T = (X, \Sigma, f, X_0, X_m, \mu)$, its complement TIA, denoted by $G_T^c = (X \cup \{x_d\}, \Sigma, f^c, x_0, \{X \cup \{x_d\}\} \backslash X_m, \mu^c)$, is generated in three steps:

  (i) Create a copy of G_T and a new state x_d, such that for each $x \in X$ and $\sigma \in \Sigma$, a new transition $(x, (\sigma, I), x_d)$ is defined, where $I = \mathbb{R}^2 \setminus \bigcup_{y \in f(x,\sigma)} \mu(x, \sigma, y)$ if $f(x, \sigma)!$ or $I = [0, +\infty)$ otherwise.

  (ii) For each $\sigma \in \Sigma$, create self-loops $(x_d, (\sigma, [0, +\infty)), x_d)$.

  (iii) Unmark the originally marked states X_m and mark the states in $(X \cup \{x_d\}) \backslash X_m$.

- Example
  Consider the TIA $G_T = (X, \Sigma, f, x_0, X_m, \mu)$, shown in Figure 3.81, where $X = \{0, 1, 2\}$, $\Sigma = \{a\}$, $f(0, a) = 1$, $x_0 = 0$, $X_m = \emptyset$ and $\mu(0, a, 1) = [1, 3]$. The automaton G_T^c, shown in Figure 3.82, is computed with the following command.

```
from deslab import *

# automaton definition G_T
Xt1 = [0, 1]
Et1 =[a]
sigobst1 = [a]
X0t1 = [0]
Xmt1 = [ ]
Tt1 = [(0,a,1)]
```

```
mut1 = {(0,a,1): P.closed(1,3)
       }

G1 = fsa(Xt1,Et1,Tt1,X0t1,Xmt1,Sigobs=sigobst1,name="$G_{1}$")
G_tia1 = tia(G1,mut1)
ti_draw(G_tia1,'figure')

# Complement
ti_draw(ti_complement(G_tia1),'figure')
```

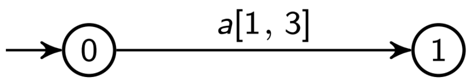


Figure 3.81: TIA G_T of the example in subsection 3.9.7.

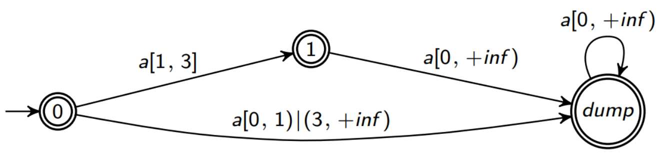


Figure 3.82: Complement of automaton of figure 3.81.

3.10 Opacity verification toolbox for TIA

3.10.1 Timed Language-Based Opacity Verification

- Purpose
  Verify the property of timed language-based opacity (TLBO) in TIA.

- Syntax

$$TLBO(G_{S_T}, G_{NS_T})$$

  - Input: TIA that marks the secret language, G_{S_T}, and TIA that marks the non-secret language, G_{NS_T}.
  - Output: Answer about the type of opacity.

- Description
  Let the automata $G_{S_T} = (X, \Sigma, f, x_0, X_m, \mu)$ and $G_{NS_T} = (X, \Sigma, f, x_0, X_m, \mu)$ represent the secret and non-secret languages, respectively. First, the labeled obfuscated product composition between G_{S_T} and G_{NS_T} is computed to obtain $G_{ofs,l}$, which represents the obfuscated language. Then, the labeled revealed product composition between G_{S_T} and the complement of the non-secret language $G^c_{NS_T}$ is computed to obtain $G_{rev,l}$, which represents the revealed language. Then, an automaton G_v is created, such that its states are of the form {states of $G_{ofs,l}$, states of $G_{rev,l}$} and its transitions are labeled as follows:

  - *co*: completely obfuscated
  - *cr*: completely revealed
  - *por*: partially obfuscated/revealed

  The verification of opacity is done at the end through the automaton G_v, returning one of the 5 types presented in [11]:

  (*i*) TISO : Time Independent Strongly Opaque;
  (*ii*) TDSO : Time Dependent Strongly Opaque;
  (*iii*) TIWO : Time Independent Weakly Opaque;
  (*iv*) TDWO : Time Dependent Weakly Opaque;
  (*v*) Not opaque.

- Example
  Consider the TIA that marks the secret time-interval language, L_S, $G_{S_T} = (X, \Sigma, f, x_0, X_m, \mu)$, shown in Figure 3.83, where $X = \{0, 1, 2, 3, 4\}$, $\Sigma = \{a, b\}$, $f(0, a) = 1$, $f(0, a) = 3$, $f(0, b) = 4$, $f(1, b) = 2$, $x_0 = 0$, $X_m = \{2, 3, 4\}$, and $\mu = \{((0, a, 1), [0, 2]), ((0, a, 3), [0, 1]), ((0, b, 4), [2, 4]), ((1, b, 2), [1, 5])\}$; and the TIA that marks the non-secret time-interval language, L_{NS}, $G_{NS_T} = (X, \Sigma, f, x_0, X_m, \mu)$, Figure 3.84, where $X = \{0, 1, 2, 3, 4\}$, $\Sigma = \{a, b\}$, $f(0, a) = 1$, $f(0, a) = 3$, $f(0, b) = 4$, $f(1, b) = 2$, $x_0 = 0$, $X_m = \{2, 3, 4\}$, and $\mu = \{((0, a, 1), [0, 5]), ((0, a, 3), [0, 2]), ((0, b, 4), [0, 1]), ((1, b, 2), [1, 4])\}$. The verifier automaton, G_v, labeled with information about language opacity, shown in Figure 3.85, is internally generated by the function, and the response about TLBO can be obtained through the following commands.

```
from deslab import *
syms('0 1 2 3 4 a b')


# automaton definition G_ST
Xs = [0,1,2,3,4]
sigobss = [a,b]
Es = [a,b]
X0s = [0]
Xms = [2,3,4]
Ts =[(0,a,1),(1,b,2),(0,a,3),(0,b,4)]

mus = {
     (0,a,1) : P.closed(0,2),
     (1,b,2) : P.closed(1,5),
     (0,a,3) : P.closed(0,1),
     (0,b,4) : P.closed(2,4)
     }

Gs = fsa(Xs,Es,Ts,X0s,Xms,Sigobs = sigobss,name="$G_s$")
Gst = tia(Gs,mus)

# automaton definition G_NST
Xns = [0,1,2,3,4]
sigobsns = [a,b]
Ens = [a,b]
X0ns = [0]
```

```
Xmns = [2,3,4]
Tns =[(0,a,1),(1,b,2),(0,a,3),(0,b,4)]

muns = {
     (0,a,1) : P.closed(0,5),
     (1,b,2) : P.closed(1,4),
     (0,a,3) : P.closed(0,2),
     (0,b,4) : P.closed(0,1)
     }

Gns = fsa(Xns,Ens,Tns,X0ns,Xmns,Sigobs = sigobsns,name='$G_{ns}$')
Gnst = tia(Gns,muns)

# Print the language based opacity property
print(TLBO(Gst,Gnst))
```

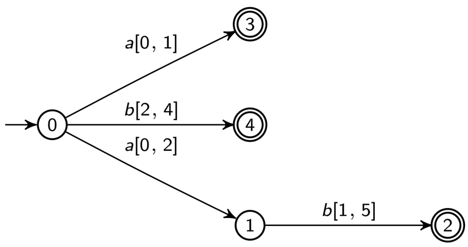


Figure 3.83: TIA G_{S_T} of the example in section 3.10.1.

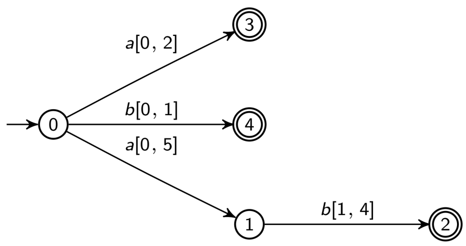


Figure 3.84: TIA G_{NS_T} of the example in section 3.10.1.

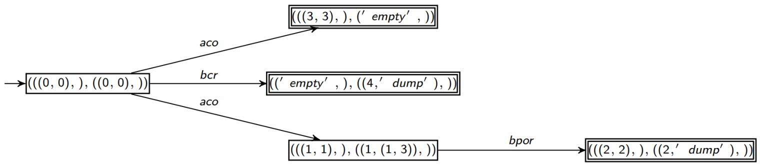


Figure 3.85: Verifier automaton with information about TLBO.

- Console outputs
  >>> TIWO

3.11 Diagnoses toolbox for TIA

3.11.1 Diagnoser TI

- Purpose
  Generates the automaton for fault diagnosis presented in [12].
- Syntax

$$ti\_diag(G_T, fault\_event, ret)$$

  - Input: Time-interval automaton (TIA), fault event, and variable *ret* which can be defined as:
    * GD (Default): Returns the diagnoser G_{d_T};
    * AL: Returns the label automaton with time interval A_{l_T};
    * GL: Returns the product between G_T and A_{l_T}.
  - Output: Automaton defined in *ret*, or GD if *ret* is not given as input.

- Description
  Given a timed automaton $G_T = (X, \Sigma, f, X_0, X_m, \Sigma_o, \mu)$, the diagnoser is defined as $G_{d_T} = Obs(G_{l_T}) = Obs(G_T \times A_{l_T}) = (X_d, \Sigma_o, f_d, x_{0_d}, \mu_d)$, where $A_{l_T} = (X_l, \Sigma, f_l, x_{0_l}, \mu_l)$. The automaton A_{l_T} is a two-state TIA where $X_l = \{N, Y\}$, $x_{0_l} = N$, $f_l(N, \Sigma \backslash \{\sigma_f\}) = N$, $f_l(N, \sigma_f) = Y$, $f_l(Y, \Sigma) = Y$, and $\mu_l(N, \Sigma \backslash \{\sigma_f\}, N) = \mu_l(N, \sigma_f, Y) = \mu_l(Y, \Sigma, Y) = [0, +\infty)$.

  The states of G_{l_T} have the form $x_l = (x, l)$, where $x \in X_l$ and $l \in \{Y, N\}$. The states of G_{l_T} will be denoted as xl for simplification. Thus, the states of G_{d_T} are defined as $x_d = \{x_1 l_1, x_2 l_2, ..., x_n l_n\}$. A state $x_d \in X_d$ is called Y-certain (or faulty) if $l_i = Y, i = 1, ..., n$, and N-certain (or non-faulty) if $l_i = N, i = 1, ..., n$. If there exist $x_i l_i, x_j l_j, i \neq j, i, j \in \{1, 2, ..., n\}$, where x_i is not necessarily distinct from x_j, such that $l_i = Y$ and $l_j = N$, then x_d is considered uncertain.

- Example
  Consider the TIA $G_T = (X, \Sigma, f, x_0, X_m, \Sigma_o, \mu)$, shown in Figure 3.86, where $X = \{0, 1, 2, 3, 4, 5\}$, $\Sigma = \{a, b, c, f\}$, $f(0, a) = 1$, $f(1, b) = 2$, $f(2, a) = 3$, $f(1, f) = 4$, $f(3, c) = 3$, $f(4, a) = 5$, $f(5, c) = 5$, $x_0 = 0$, $X_m = \emptyset$, $\Sigma_o = \{a, c\}$, $\mu = \{((0, 'a', 1), [1, 2]), ((1, 'b', 2), [3, 4]), ((2, 'a', 3), [2.5, 4]), ((1, 'f', 4), [0, 1]), ((3, 'c', 3), [0.5, 1.5]), ((4, 'a', 5), [2.5, 4]), ((5, 'c', 5), [1, 2])\}$, and $\sigma_f = \{f\}$. The automaton G_{d_T}, shown in Figure 3.87, is obtained through the following commands.

```
syms('a b c f')

# automaton definition G_T
Xt1 = [0, 1, 2, 3, 4, 5]
Et1 =[a,b,c,f]
sigobst1 = [a,c]
X0t1 = [0]
Xmt1 = []
Tt1 = [(0,a,1),(1,b,2),(2,a,3),(1,f,4), (3,c,3), (4,a,5), (5,c,5)]

mut1 = {(0,a,1): P.closed(1,2),
        (1,b,2): P.closed(3,4),
        (2,a,3): P.closed(2.5,4),
        (1,f,4): P.closed(0,1),
        (3,c,3): P.closed(0.5,1.5),
        (4,a,5): P.closed(2.5,4),
        (5,c,5): P.closed(1,2)
       }

G = fsa(Xt1,Et1,Tt1,X0t1,Xmt1,Sigobs=sigobst1,name="$G_{T}$")
GT = tia(G,mut1)

# Diagnoser
ti_draw(GT, 'figure')
gdt = ti_diag(GT,f)
ti_draw(gdt,'figure')
```

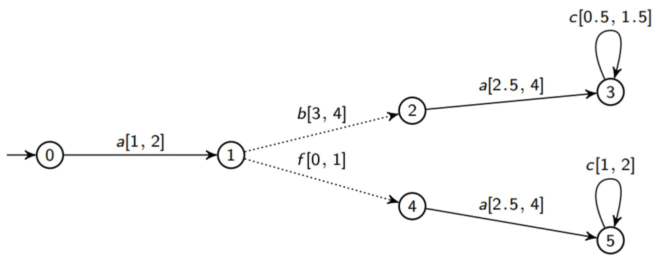


Figure 3.86: TIA from the example in subsection 3.11.1.

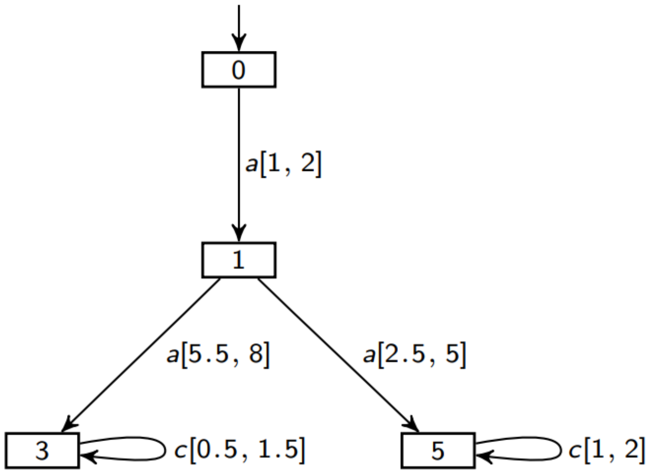


Figure 3.87: Automaton G_{d_T} of the example in subsection 3.11.1.

3.11.2 Timed Test Automaton

- Purpose
  Generates the automaton for fault diagnosis presented in [13].

- Syntax

$$ti\_scc(G_t, fault\_event)$$

  - Input: Time-interval automaton (TIA) and fault event.
  - Output: Timed test automaton.

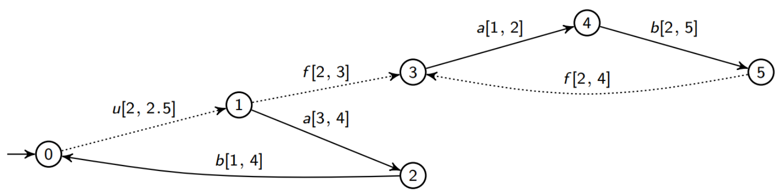
- Description
  Given a TIA $G_T = (X, \Sigma, f, X_0, X_m, \Sigma_o, \mu)$, the construction of the test automaton is done following the steps:

  1. Modify the labeled automaton by replacing transitions labeled with observable events from detectable paths with another transition labeled by the concatenation of these events, without losing the time interval information of the original events.
  2. Store the information of the detectable path containing unobservable events in the states of the TIA obtained in the previous step.
  3. Create the time-interval observer for the TIA obtained in the second step.
  4. Recover the information of the detectable paths from the states of the TIA obtained in step 3 by replacing/adding transitions.
  5. Compute G_{scc_T} and verify diagnosability.

  The diagnosability analysis must be done manually at the end of the process.

- Example
  Consider the TIA $G_T = (X, \Sigma, f, x_0, X_m, \Sigma_o, \mu)$, shown in Figure 3.88, where $X = \{0, 1, 2, 3, 4, 5\}$, $\Sigma = \{a, b, u, f\}$, $f(0, u) = 1$, $f(1, a) = 2$, $f(2, b) = 0$, $f(1, f) = 3$, $f(3, a) = 4$, $f(4, b) = 5$, $f(5, f) = 3$, $x_0 = 0$, $X_m = \emptyset$, $\Sigma_o = \{a, b\}$, $\mu = \{((0,'u', 1), [2, 2.5]), ((1,'a', 2), [3, 4]), ((2,'b', 0), [1, 4]), ((1,'f', 3), [2, 3]), ((3,'a', 4), [1, 2]), ((4,'b', 5), [2, 5]), ((5,'f', 3), [2, 4])\}$, and $\sigma_f = \{f\}$. The TIA G_{scc_T}, shown in Figure 3.89, is computed with the following commands.

```
syms('a b u f')

# automaton definition G_T
Xt1 = [0, 1, 2, 3, 4, 5]
Et1 =[a, b, u, f]
sigobst1 = [a,b]
X0t1 = [0]
Xmt1 = [ ]
Tt1 = [(0,u,1),(1,a,2),(2,b,0),(1,f,3), (3,a,4), (4,b,5), (5,f,3)]

mut1 = {(0,u,1): P.closed(2,2.5),
        (1,a,2): P.closed(3,4),
        (2,b,0): P.closed(1,4),
        (1,f,3): P.closed(2,3),
        (3,a,4): P.closed(1,2),
        (4,b,5): P.closed(2,5),
        (5,f,3): P.closed(2,4)
       }

G = fsa(Xt1,Et1,Tt1,X0t1,Xmt1,Sigobs=sigobst1,name="$G_{T}$")
GT = tia(G,mut1)

# Timed test automaton
gscc = ti_scc(GT,f)
ti_draw(gscc,'figure')
```



"""

Figure 3.88: TIA G_T of the example in subsection 3.11.2.

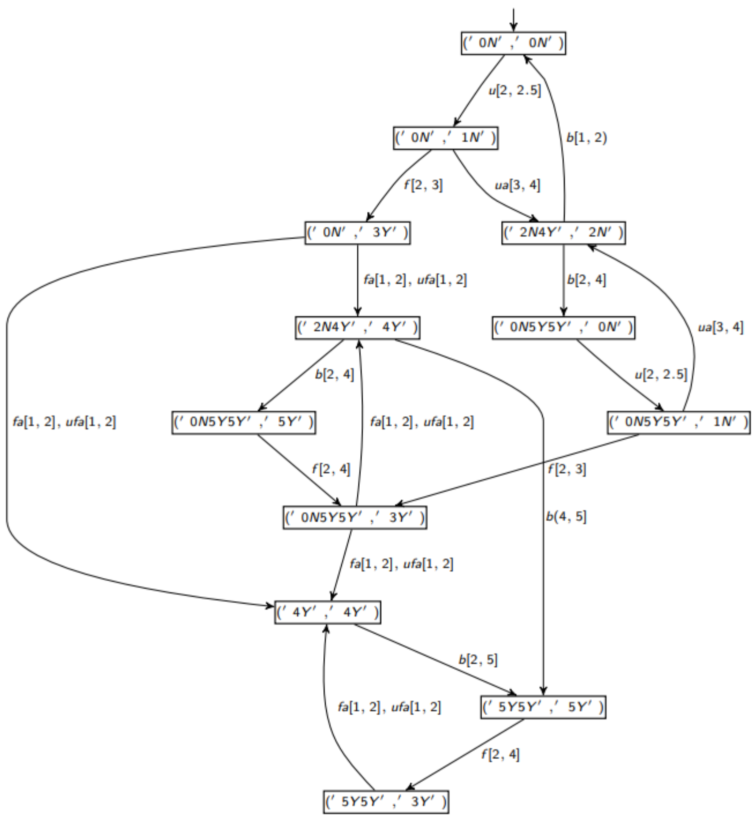


Figure 3.89: Output TIA, G_{scc_T}, of the example in subsection 3.11.2.

Bibliography

[1] WU, Y.-C., LAFORTUNE, S., "Comparative analysis of related notions of opacity in centralized and coordinated architectures", *Discrete Event Dynamic Systems*, v. 23, n. 3, pp. 307–339, 2013.

[2] COUTINHO, L. E. A. A., *A tutorial for the scientific computing program DESlab*. Trabalho de conclusão de curso, 2014.

[3] GARCIA, D. R., *DESlab para desenvolvedores*. Trabalho de conclusão de curso, 2018.

[4] BARBOSA, N. R., *DESlab 1.0*. Trabalho de conclusão de curso, 2025.

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

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

[7] MOREIRA, M. V., JESUS, T. C., BASILIO, J. C., "Polynomial time verification of decentralized diagnosability of discrete event systems", *IEEE Transactions on Automatic Control*, v. 56, n. 7, pp. 1679–1684, 2011.

[8] BARCELOS, R. J., BASILIO, J. C., "Enforcing current-state opacity through shuffle and deletions of event observations", *Automatica*, v. 133, pp. 109836, 2021.

[9] BARCELOS, R. J., BASILIO, J. C., "Ensuring utility while enforcing current-state opacity", *IFAC-PapersOnLine*, v. 56, n. 2, pp. 4595–4600, 2023.

[10] JI, Y., LAFORTUNE, S., "Enforcing opacity by publicly known edit functions". In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pp. 4866–4871, IEEE, 2017.

[11] MARQUES, M., BARCELOS, R., BASILIO, J. C., "Introduzindo Novas Definições de Opacidade de Linguagem de Sistemas a Eventos Discretos Modelados Por Uma Classe de Autômatos Temporizados", , 2024.

[12] REZENDE, C. H., VIANA, G. S., BASILIO, J. C., "Algoritmo baseado na busca de componentes fortemente conexos para verificaçao de diagnosticabilidade com intervalo de tempo", .

[13] REZENDE, C. H., VIANA, G. S., BASILIO, J. C., "Diagnosability of discrete event systems modeled by time-interval automata", *IFAC-PapersOnLine*, v. 56, n. 2, pp. 8660–8665, 2023.