Universidade Federal
do Rio de Janeiro
Escola Politécnica

A TUTORIAL FOR THE SCIENTIFIC COMPUTING PROGRAM DESLAB

Lahis El Ajouze Azeredo Coutinho

Projeto de Graduação apresentado ao Corpo Docente do Departamento de Engenharia Elétrica da Escola Politécnica da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro Eletricista.

Orientador: Lilian Kawakami de Carvalho

Rio de Janeiro
Agosto de 2014

A TUTORIAL FOR THE SCIENTIFIC COMPUTING PROGRAM DESLAB

Lahis El Ajouze Azeredo Coutinho

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO ELETRICISTA.

Examinado por:

Prof. Lilian Kawakami de Carvalho, D.Sc.

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

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

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 2014

Coutinho, Lahis El Ajouze Azeredo

Um Tutorial para o Programa Científico Computacional DESLab / Lahis El Ajouze Azeredo Coutinho. – Rio de Janeiro: UFRJ/Escola Politécnica, 2014.

XI, 122 p.: il.; 29, 7cm.

Orientador: Lilian Kawakami de Carvalho

Projeto de Graduação – UFRJ/Escola Politécnica/ Departamento de Engenharia Elétrica, 2014.

Referências Bibliográficas: p. 121 – 122.

1. Discrete Event Systems. 2. Automata. 3. Tutorial. 4. Verification and Validation. I. Carvalho, Lilian Kawakami de. II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Departamento de Engenharia Elétrica. III. Um Tutorial para o Programa Científico Computacional DESLab.

Acknowledgments

I thank God, primarily and at all times, for every single moment so far and all the better ones that are yet to come.

My parents Soraya Cascun and Pedro Paiva, along with my beloved grandmother Mirian Cascun, deserve all my loving and gratitude.

Graduating would also not be possible without some special friends that crossed my way. The most sincere thank you for everyone that somehow helped me throughout these last years of few hours of sleep, tons of work and lots of coffee. It has been great, and it does come to an end. Be faithful.

At long last, I thank the professors responsible for guiding my way here, specially my adviser Lilian Kawakami for all the time shared and the help provided.

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

UM TUTORIAL PARA O PROGRAMA CIENTÍFICO COMPUTACIONAL DESLAB

Lahis El Ajouze Azeredo Coutinho

Agosto/2014

Orientador: Lilian Kawakami de Carvalho

Departamento: Engenharia Elétrica

O programa computacional científico DESLab foi recentemente desenvolvido para análise e síntese de sistemas a eventos discretos (SED) modelados por autômatos. Escrito em Python, o DESLab é um programa gratuito de código aberto. A fim de que seja propriamente difundido pela comunidade científica, é necessário que o programa seja testado, verificado e validado. Além disso, DESLab deve possuir um tutorial. Dessa forma, este trabalho consiste na verificação e validação do programa DESLab, assim como no desenvolvimento de seu tutorial, contendo breves explicações da teoria envolvida e da sintaxe dos comandos, além de exemplos práticos para guiar os usuários.

Palavras-chave: Sistemas a Eventos Discretos, Autômatos, Tutorial, Verificação e Validação.

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

A TUTORIAL FOR THE SCIENTIFIC COMPUTING PROGRAM DESLAB

Lahis El Ajouze Azeredo Coutinho

August/2014

Advisor: Lilian Kawakami de Carvalho

Department: Electrical Engineering

The recently developed scientific computing program DESLab was intended for the analysis and synthesis of discrete event systems (DES) modeled as automata. Written in Python, DESLab is a free open source program. In order to be properly launched to the scientific community, it demands to be tested, verified and validated. Moreover, it should come with a tutorial. In this regard, this work consists in the verification and validation of DESLab, as well as the development of a tutorial, containing brief explanations of the command syntax and the theory involved, along with practical examples to guide the users through its usage.

Keywords: Discrete Event Systems, Automata, Tutorial, Verification and Validation.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introduction

Ever since industry emerged, it has been seeking for constant improvement of operation and optimization of processes. Over the last years, the concept of automation came to light as an alternative capable of reducing labor, energy and material expenses, whilst enhancing quality, precision and accuracy in industry. General Motors established its automation department in 1947 [1], and after that the theory started to be spread out and widely used in computer, manufacturing, traffic, database, software, hybrid and many other types of systems.

Even though systems can be classified in many different aspects, this work will focus on a particular type of a discrete-state, event-driven system known as Discrete Event System (DES) [2]. In order to better comprehend its definition, it is necessary to elucidate a few concepts such as system, event and state, that will be further presented along with a mathematical formalism used to describe DESs. Based on set and graph theories, dealing with this formalism can be very complex depending on the dimension of the system concerned.

Hence, the DES research community began to develop several computing tools to help managing the mathematical intricacy, and some of the most important of them deserves to be highlighted. DESUMA, TCT, SUPREMICA, IDES and libFAUDES are mostly free but not open source software that have been developed in the past few years [3]. Each one of them has its specific advantages, but the urge of integrating automata theory, graph algorithms and numerical calculation in order to manipulate, operate, analyze and visualize an automaton object encouraged the development of a new free and open source scientific computing program called DESLab.

In order to safely launch it to the community, guaranteeing its best performance and accordance to the expected theory results, DESLab needed to be validated and verified [4]. In this regard, it is the purpose of this work to analyse the program features with a view to clarifying whether or not they match the developer primary goals. In order to achieve these goals, all the commands were run and fully tested to evaluate their correspondence to the related theory. Besides, a program demands

a tutorial capable of foreseeing possible complications on the way the software is resorted in order to help users getting started with it. Thus, it has been developed a tutorial explaining the syntax, briefly describing the commands and providing examples of each DESLab functionality.

The work hereby presented consists in the validation and verification of a new scientific computing program called DESLab. Furthermore, it presents the development of the software tutorial, resulting from studies on discrete event systems, automata and graph theory, along with extensive test running and error investigation.

This work is organized as follows. In Chapter 2, the theory of discrete event systems, relevant to this work, is presented, consisting in key concepts of languages and automata. In Chapter 3, the scientific computing program DESLab is presented, along with information about verification and validation, tutorials, graph theory and programming languages. The DESLab tutorial is proposed in Chapter 4, divided in four categories: basic instructions, operations on languages, unary and composition operations and additional instructions. The work is finally concluded in Chapter 5.

Capítulo 2

Discrete Event Systems

This chapter presents a fragment of the theory of discrete event systems that is relevant to the comprehension of this work. It covers simple introductory concepts such as systems, events and states. Moreover, it presents the concepts of languages and automata in a way to mathematically formalize discrete event systems [2].

An exact definition of *system* is not very structured in literature, but there are some representative options that are good enough for understanding this intuitive concept. *Encyclopedia Americana* defined a system as *"an aggregation or assemblage of things so combined by nature or man as to form an integral or complex whole"* [5]. In other words, it can be seen as an entirety of components associated with a function.

Defining an event is also not very simple, but it can be fairly described as an action that occur instantaneously and provoke a variation in the system. Before the occurrence of an event, being it spontaneous by nature or resulted from the meeting of several conditions, the discrete system was at a determined state. Upon the occurrence of the event, here represented by the symbol e, a transition between states occur, taking the system to another state value. An event set is clearly discrete and is here denoted by the symbol Σ.

The proper definition of a DES is better comprehended now that these concepts were introduced. Discrete-state systems are the ones where state variables are elements of a discrete set, here taken as X. In event-driven systems, on the other hand, instantaneous state transitions are forced by the occurrence of asynchronously generated discrete events. These states remain unaltered unless an event occur. That said, a DES is informally defined as a discrete-state, event-driven system.

The discrete sets of events and states give rise to a language-based approach to model these systems and start mathematically formalizing them. Based on set theory, languages are capable, together with the information of the initial state of the system, to describe a DES. For theoretic discusses it is valid and widely used.

However, taking the practical side in account, it is preferred to use discrete event modeling formalisms, that are state transition structures that represent languages and, therefore, discrete event systems. The literature counts with several modeling formalisms, but the one to be presented here will be *automata*.

Languages and automata are conjointly able to model discrete event systems and empower composition operations, help analyzing and synthesizing structures, expressively improving dealing with DESs.

2.1 Languages

The concept of language is based on set theory and is relevant to discrete event system modeling since the sets of states and events are discrete. Specifically focusing on the event set Σ, it is possible to think of a sequence of events. If events are labeled by letters, and Σ is taken as an *alphabet*, the strings that represent sequences of events can be taken as *words*. Given a system, all the sequences of events that are possible to occur can then be faced as the *language* spoken by this system. That line of thinking start explaining language modeling.

2.1.1 Notation and Definitions

As it was already introduced, the event set Σ of a DES is here treated as an alphabet assumed to be finite. Events are generically represented by the letter e, and empty strings (or words) are represented by the symbol ε. Note that an empty alphabet has no elements, that is, not even ε is included. A sequence, string or word is denoted by s, and its length is given by $|s|$. By convention, the length of ε is zero.

Definition 1 *Defined over an event set Σ, a language is a set which elements are finite-length strings formed from the events of Σ.* □

A language defined over a set Σ is a subset of Σ^*, which denotes the set of all possible finite-length sequences of events in Σ, including ε. Particularly, $\emptyset$, Σ and Σ^* itself are languages.

Let $s = abc$ denote a sequence where a, b and $c \in \Sigma^*$. Then, referred to s, a is the prefix, b is a substring and c is the suffix. Besides, s/t denotes the suffix of s after the prefix a. Whenever a is not a prefix of a given s, s/t is not defined.

2.1.2 Operations on Languages

Inasmuch as languages are sets, all the usual set operations are also applicable to them. Besides, new operations are defined as follows.

- Concatenation: Let L_a, $L_b \subseteq \Sigma^*$, then

$$L_a L_b := \{s \in \Sigma^* : (s = s_a s_b) \wedge (s_a \in L_a) \wedge (s_b \in L_b)\}.$$

- Prefix-closure: Let $L \subseteq \Sigma^*$, then

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

- Kleene-closure: Let $L \subseteq \Sigma^*$, then

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

- Projections:

$$P : \Sigma_l^* \to \Sigma_s^*, \; \Sigma_s \subset \Sigma_l,$$

where

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

$$P(e) := \begin{cases} e & \text{if } e \in \Sigma_s \\ \varepsilon & \text{if } e \in \Sigma_l \setminus \Sigma_s, \end{cases}$$

$$P(se) := P(s)P(e) \text{ for } s \in \Sigma_l^*, e \in \Sigma_l.$$

Likewise, it is possible to define the inverse operation, with respect to the corresponding inverse map:

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

where

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

2.2 Automata

The modeling formalism here presented is centered on the automaton, which is a device defined as a six-tuple capable of representing and manipulating languages.

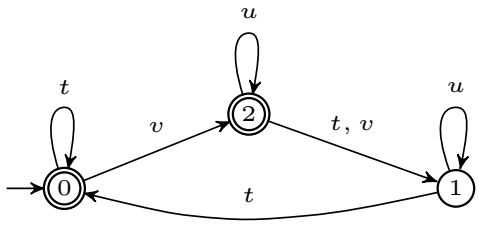


Figura 2.1: State transition diagram from example 1.

Definition 2 *A deterministic automaton, denoted by G, is a six-tuple:*

$$G = (X, \Sigma, f, \Gamma, x_0, X_m) \tag{2.1}$$

where X is the set of states, Σ is the set of events associated with G, $f : X \times \Sigma \to X$ is the transition function, that is usually partial on its domain, $\Gamma : X \to 2^{\Sigma}$ is the active event (or feasible event) function, x_0 is the initial state and $X_m \subseteq X$ is the set of marked states. □

The well-defined rules have another handy result: the *state transition diagram*, which is the oriented graph representation of an automaton and helps shaping up all the math involved with DESs. In the state transition diagrams, the vertices of the graph are circles and represent the different states of the system. The arcs connecting them are labelled by symbols, usually letters, that are the events of the system. A transition can be understood as the combination of a pair of states, an arc connecting them and its label. Note that it is possible to have selfloops, which in this case are transitions that start and end at the same state. The initial state has an arrow pointing to it, and the marked states are double circles.

An example of an automaton and its state transition diagram is provided.

Example 1 *Let G be a deterministic automaton which state transition diagram can be seen in Figure 2.1. The sets of states and events are respectively given by $X = \{0, 1, 2\}$ and $\Sigma = \{t, u, v\}$. The transition function is defined as: $f(0, t) = 0$; $f(0, v) = 2$; $f(1, t) = 0$; $f(1, u) = 1$; $f(2, u) = 2$; $f(2, t) = f(2, v) = 1$, causing the active event functions to be: $\Gamma(0) = \{t, v\}$; $\Gamma(1) = \{t, u\}$; $\Gamma(2) = \{t, u, v\}$. Finally, the initial state is $x_0 = 0$ and the set of marked states $X_m = \{0, 2\}$.*

The generated and marked languages of an automaton are described according to definition 3.

Definition 3 *A language generated by an automaton G is given by:*

$$\mathcal{L}(G) := \{s \in \Sigma^* : f(x_0, s) \text{ is defined}\}, \tag{2.2}$$

While the language marked by G is given by:

$$\mathcal{L}_m(G) := \{s \in \mathcal{L}(G) : f(x_0, s) \in X_m\}. \tag{2.3}$$

□

It is important rephrasing that on definition 3, it is supposed that the transition function f is extended, that is,

$$f : X \times \Sigma^* \to X.$$

Besides, for every G possessing a non-empty set of states X, $\varepsilon \in \mathcal{L}(G)$.

Starting at the initial state, and following along the state transition diagram, all the directed possible paths are represented by the language generated by G, $\mathcal{L}(G)$. The transitions composing each path are labelled by events. The concatenation of these events is the string corresponding to this path, in a way that $s \in \mathcal{L}(G)$ if and only if it corresponds to an admissible path in the state transition diagram. It is important remarking that $\mathcal{L}(G)$ is prefix-closed by definition, and that it is possible having events defined in Σ that do not appear in the state transition diagram. That indicates that these events will not figure in $\mathcal{L}(G)$.

The language marked by G, $\mathcal{L}_m(G)$, is a subset of $\mathcal{L}(G)$ corresponding to all the sequences s in the state transition diagram labelling paths that ends in a marked state, that is, for which $f(x_0, s) \in X_m$. Note that $\mathcal{L}_m(G)$ is not necessarily prefix-closed, since not all states in X are to be marked.

An automaton marking the empty language is said to be the empty automaton. It possesses an empty set of states X and generates $\mathcal{L}(G) = \emptyset$.

Another relevant definition to be presented involves the concept of *blocking*. While inspecting the state transition diagram, whenever a path takes to a state where no further events can be executed, it is called a *deadlock*. Analogously, a path that takes to a set of strongly connected[1] unmarked states is actually taking to a *livelock*. Staring at the big picture, and taking real systems in consideration, the idea of having ends in the state transition diagram confronts the idea of a cycling process, for instance. That indicates the importance of the definition that follows.

Definition 4 *An automaton G is said to be blocking if*

$$\overline{\mathcal{L}_m(G)} \subset \mathcal{L}(G) \tag{2.4}$$

And nonblocking when

$$\overline{\mathcal{L}_m(G)} = \mathcal{L}(G) \tag{2.5}$$

[1]In graph theory, a pair of vertices in a directed graph is said to be strongly connected if there is a path in each direction between them [6] .

□

It is possible that different automata generate and mark the same language. These are *language-equivalent* automata.

Definition 5 *Automata G_1 and G_2 are said to be language-equivalent if*

$$\mathcal{L}(G_1) = \mathcal{L}(G_2) \text{ and } \mathcal{L}_m(G_1) = \mathcal{L}_m(G_2) \tag{2.6}$$

□

Automata theory is not restricted to deterministic automata. It is possible to have an automaton with more than one initial state, as well as having events labelling transitions that are not defined in Σ. Moreover, a single event may label multiple transitions from a state x. Whenever an automaton meets at least one of these three requirements it is said to be nondeterministic.

Definition 6 *A nondeterministic automaton, denoted by G_{nd}, is a six-tuple:*

$$G_{nd} = (X, \Sigma \cup \{\varepsilon\}, f_{nd}, \Gamma, x_0, X_m) \tag{2.7}$$

where the differences are:

- $f_{nd} : X \times \Sigma \cup \varepsilon \to 2^X$*, that is,* $f_{nd}(x, e) \subseteq X$ *whenever it is defined.*

- *The initial state* x_0 *may be a set of states such as* $x_0 \subseteq X$*.* □

Defining the generated and marked languages by a nondeterministic automaton demands a few previous steps. For starts, the transition function needs to be extended to the domain $X \times \Sigma^*$, denoted by $f_{nd}^{ext}(x, \varepsilon)$. Further on, the $\varepsilon - reach$ of a state x is defined to be the set of all states that can be reached from x by following transitions labelled by ε in the state transition diagram. The proper notation is $\varepsilon R(x)$. By convention, $x \in \varepsilon R(x)$. The definition is extended to a subset of states $B \subseteq X$,

$$\varepsilon R(B) = \bigcup_{x \in B} \varepsilon R(x). \tag{2.8}$$

Recursively, $f_{nd}^{ext}(x, \varepsilon)$ is constructed as follows

$$f_{nd}^{ext}(x, \varepsilon) := \varepsilon R(x) \tag{2.9}$$

For $u \in \Sigma^*$ and $e \in \Sigma$,

$$f_{nd}^{ext}(x, ue) := \varepsilon R[z : z \in f_{nd}(y, e) \text{ for some state } y \in f_{nd}^{ext}(x, u)] \tag{2.10}$$

Now that all the requirements have been met, the generated and marked languages of a nondeterministic automaton are defined as follows.

Definition 7 *The language generated by a nondeterministic automaton G_{nd} is*

$$\mathcal{L}(G_{nd}) := \{s \in \Sigma^* : (\exists x \in x_0)[f_{nd}^{ext}(x, s) \text{ is defined}\} \tag{2.11}$$

While the language marked by G_{nd} is given by:

$$\mathcal{L}_m(G_{nd}) := \{s \in \mathcal{L}(G_{nd}) : (\exists x \in x_0)[f_{nd}^{ext}(x, s) \cap X_m \neq \emptyset\}. \tag{2.12}$$

□

It is possible for a deterministic automaton to generate and mark the same languages of a nondeterministic automaton. The automata are then said to be *language equivalent*, and the deterministic automaton is called the *observer*. The procedure for building an observer automaton is proposed.

The observer of a G_{nd} is defined as $Obs(G_{nd}) = (X_{obs}, \Sigma, f_{obs}, x_{0,obs}, X_{m,obs})$ and is built in four steps.

Step 1: Define $x_{0,obs} := \varepsilon R(x_0)$ and set $X_{obs} = x_0$;
Step 2: For each B $\in X_{obs}$ and $e \in \Sigma$, define

$$f_{obs}(B, e) := \varepsilon R(x \in X : (\exists x_e \in B)[x \in f_{nd}(x_e, e)]) \tag{2.13}$$

whenever $f_{nd}(x_e, e)$ is defined for some $x_e \in B$. In this case, add the state $f_{obs}(B, e)$ to X_{obs}. If $f_{nd}(x_e, e)$ 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: $X_{m,obs} := \{B \in X_{obs} : B \cap X_m \neq \emptyset\}$.

The observer automaton is an important tool in the study of partially-observed systems, that are the ones where some events are defined as unobservable and behave exactly like a ε-transition of a nondeterministic automaton.

The model for a partially-observed DES is a deterministic automaton whose set of events is $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$, a partition of Σ, where Σ_o and Σ_{uo} are, respectively, the set of observable and unobservable events. This event set is used to build an observer

automaton just like before. The notion of ε-reach is now generalized by the notion of *unobservable reach*, that is

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

Extended to the set of states $B \subseteq X$,

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

Let $G = (X, \Sigma, f, x_0, X_m)$ denote a partially observed automaton with the partitioned event set. $Obs(G_{nd}) = (X_{obs}, \Sigma, f_{obs}, x_{0,obs}, X_{m,obs})$ is built as follows.

Step 1: Define $x_{0,obs} := UR(x_0)$ and set $X_{obs} = x_{0,obs}$;
Step 2: For each B $\in X_{obs}$ and $e \in \Sigma_o$, define

$$f_{obs}(B,e) := UR(x \in X : (\exists x_e \in B)[x \in f(x_e, e)]) \tag{2.14}$$

whenever $f(x_e, e)$ is defined for some $x_e \in B$. In this case, add the state $f_{obs}(B,e)$ to X_{obs}. If $f(x_e, e)$ 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: $X_{m,obs} := \{B \in X_{obs} : B \cap X_m \neq \emptyset\}$.

2.2.1 Operations on Automata

In order to properly analyze a DES modeled by an automaton it is necessary the definition of a set of operations useful for modifying the state transition diagram. Besides, operations capable of combining two or more automata are primordial once working with components of a system.

The following operations are said to be unary and do not interfere in Σ.

Accessible Part

The accessible part is an operation that deletes all the unreachable states of an automaton G, starting from the initial state x_0. It takes along all the related transitions, and is formally defined as:

$$\begin{aligned} Ac(G) &:= (X_{ac}, \Sigma, f_{ac}, x_0, X_{ac,m}) \text{ where} \\ X_{ac} &= \{x \in X : (\exists s \in \Sigma^*)[f(x_0, s) = x]\} \\ X_{ac,m} &= X_m \cap X_{ac} \\ f_{ac} &: X_{ac} \times \Sigma^* \to X_{ac} \end{aligned}$$

Note that the operation restrains the domain of the transition function to X_{ac}, although it does not alter $\mathcal{L}(G)$ e $\mathcal{L}_m(G)$.

Coaccessible Part

A state $x \in X$ is said to be coaccessible whenever there is a path starting from state x and ending in a marked stated belonging to X_m. The operation erases all the states in G that are not coaccessible, along with all its related transitions. Formally:

$$\begin{aligned} CoAc(G) &:= (X_{coac}, \Sigma, f_{coac}, x_{0,coac}, X_m) \text{ where} \\ X_{coac} &= \{x \in X : (\exists s \in \Sigma^*)[f(x,s) \in X_m]\} \\ x_{0,coac} &:= \begin{cases} x_0 & \text{if } x_0 \in X_{coac} \\ \text{undefined} & \text{otherwise} \end{cases} \\ f_{coac} &: X_{coac} \times \Sigma^* \to X_{coac} \end{aligned}$$

Note that $\mathcal{L}(CoAc(G)) \subseteq \mathcal{L}(G)$. However, $\mathcal{L}_m(CoAc(G)) = \mathcal{L}_m(G)$.

Trim Operation

An automaton that is at the same time accessible and coaccessible is said to be trim, formally defined as:

$$Trim(G) := CoAc[Ac(G)] = Ac[CoAc(G)]. \tag{2.15}$$

Complement

Consider a trim deterministic automaton G that marks $L \subseteq \Sigma^*$ and thus generates $\overline{\mathcal{L}}$. The automaton marking the language $\Sigma^* \setminus L$ is here denoted by G^{comp} and is built in two steps. The first one actually defines a complete automaton (if referred to G), making f to be a total function f_{tot}. In order to do so, a new state x_d is added to X, known as the *dump state*. All undefined $f(x,e)$ in G is then assigned to x_d.

$$f_{tot}(x,e) = \begin{cases} f(x,e) & \text{if } e \in \Gamma_1(x) \\ x_d & \text{otherwise} \end{cases}$$

Setting $f_{tot}(x_d, e) \forall e \in \Sigma$, the complete automaton is built, such that $\mathcal{L}(G_{tot}) = \Sigma^*$ and $\mathcal{L}_m(G_{tot}) = L$

$$G_{tot} = (X \cup x_d, \Sigma, f_{tot}, x_0, X_m)$$

The next step towards building the complement is to change the marking status of all states in G_{tot} by marking all unmarked states, including x_d, and unmarking all marked states. The step yields the definition

$$Comp(G) := (X \cup x_d, \Sigma, f_{tot}, x_0, (X \cup x_d) \setminus X_m)$$

From this point, the operations are named composition operations.

Product Operation

The product operation, said to be the completely synchronous composition, is denoted by $\times$. The product between two automata G_1 e G_2 results in:

$$\begin{aligned} G_1 \times G_2 &:= Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f, \Gamma_{1\times 2}, (x_{01}, x_{02}), X_{m,1} \times X_{m,2}) \text{ where} \\ f((x_1, x_2), e) &:= \begin{cases} (f_1(x_1, e), f_2(x_2, e)), & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined}, & \text{otherwise} \end{cases} \\ \Gamma_{1\times 2}(x_1, x_2) &= \Gamma_1(x_1) \cap \Gamma_2(x_2). \end{aligned}$$

According to the definition of product composition, the transitions of both automata need to be synchronized with a common event, that is, an event $e \in \Sigma_1 \cap \Sigma_2$. An event will then only occur in $G_1 \times G_2$ if and only if it occurs simultaneously in G_1 e G_2.

The states of $G_1 \times G_2$ are denoted in pairs, in which the first component is the current state of G_1 and the second component is the current state of G_2. Besides, the generated and marked languages of $G_1 \times G_2$ are:

$$\mathcal{L}(G_1 \times G_2) = \mathcal{L}(G_1) \cap \mathcal{L}(G_2), \tag{2.16}$$

$$\mathcal{L}_m(G_1 \times G_2) = \mathcal{L}_m(G_1) \cap \mathcal{L}_m(G_2). \tag{2.17}$$

Parallel Composition

The parallel composition is also called a synchronous composition and is represented by $||$. However, it does allow private transitions to occur, synchronizing only transitions labelled by common events. It is considered to be the best way of combining components of a complex system.

The parallel composition between the two automata G_1 and G_2 results in:

$$
\begin{aligned}
G_1||G_2 &:= Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f, \Gamma_{1||2}, (x_{01}, x_{02}), X_{m,1} \times X_{m,2}) \text{ where} \\
f((x_1, x_2), e) &:= \begin{cases} (f_1(x_1, e), f_2(x_2, e)), & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, e), x_2), & \text{if } e \in \Gamma_1(x_1) \setminus \Sigma_2 \\ (x_1, f_2(x_2, e)), & \text{if } e \in \Gamma_2(x_2) \setminus \Sigma_1 \\ \text{undefined}, & \text{otherwise} \end{cases} \\
\Gamma_{1||2}(x_1, x_2) &= [\Gamma_1(x_1) \cap \Gamma_2(x_2)] \cup [\Gamma_1(x_1) \setminus \Sigma_2] \cup [\Gamma_2(x_2) \setminus \Sigma_1].
\end{aligned}
$$

In parallel composition, a common event, that is, belonging to $\Sigma_1 \cap \Sigma_2$ will occur only if both automata execute them at the same time. On the other hand, the private events, that is, the ones in $(\Sigma_1 \setminus \Sigma_2) \cup (\Sigma_2 \setminus \Sigma_1)$, may occur whenever it is possible.

If $\Sigma_1 = \Sigma_2$, then the parallel composition plays the same roll as the product operation.

In order to properly characterize both generated and marked languages of the automaton resulted from the parallel composition, a few definitions are nedded:

$$
P_i : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_i^* \text{ for } i = 1, 2. \tag{2.18}
$$

Based on these projections, the resulting languages are:

$$
\mathcal{L}(G_1||G_2) = P_1^{-1}[\mathcal{L}(G_1)] \cap P_2^{-1}[\mathcal{L}(G_2)], \tag{2.19}
$$

$$
\mathcal{L}_m(G_1||G_2) = P_1^{-1}[\mathcal{L}_m(G_1)] \cap P_2^{-1}[\mathcal{L}_m(G_2)]. \tag{2.20}
$$

Capítulo 3

The Scientific Computing Program DESLab

As a result of the advent of discrete event system, the scientific research community started developing computing tools in order to optimize the work with discrete event systems and all the mathematical formalism to them related. Some of the released tools treat specifically of automata modeling, whereas a few others focus on manipulating formal languages. Among the already established programs in use, some deserve to be remarked. [3]

DESUMA [7], a software developed at Michigan University, integrates the UMDES library with the GIDDES, a GUI for DES visualization developed at Mount Allison University. Targeting discrete event systems modeled as automata, it is able to perform graph operations, fault diagnosis and verification, modular and centralized supervisory control. Nevertheless, it is not an open source software[1], it has limitations for data input of intermediate and high level complexity systems due to GIDDES interface. Having no code input, it implied that all DES algorithms should be run manually.

TCT [9], a Wonham's group development at the University of Toronto, is a program including operations on automata, graph operations and supervisory control synthesis routines. Even though the current version includes BDD routines allowing the process of large systems, the software has some usability drawbacks such as very limited user interface and impossibility of using symbolic labels for events or states.

SUPREMICA [10] bridges the gap between supervisory control theory and industrial applications, preserving the goal of being user-friendly. Hence, its user interface provides a rich set of usability features to aid the supervisor design cycle, apart from automatic generate codes for different programming languages such as

[1] An open-source software (OSS) has its source code available and licensed such that the copyright holder provides the rights to study, change and distribute the software to anyone and for any purpose. [8]

Java bytecode, Ansi C and ABB control builder.

IDES [3], made by Queens University, is a software that simplifies the graphical input of DESs throughout a well-designed graphical user interface. Including operations with automata and supervisor synthesis, the software is not amenable for the development of new algorithms, not providing a direct code input.

The libFAUDES [11] library was implemented in C++ by Erlangen-Nuremberg University for the analyses and synthesis of DESs. Algorithms for operations with automata and specially for centralized, modular and hierarchic supervisory control are included in the library, along with new modules for fault diagnosis and BDD controller synthesis. The new interface for LUA scripting language reduces programming computation effort, even though new algorithm development is still complicated.

As it goes for formal language manipulation software, the ones to be evinced here are GRAIL, VAUCANSON and FADO. GRAIL, implemented in C++ at the University of Toronto is a symbolic computation environment for automaton manipulation, regular expressions and other formal language theory objects. Also developed in C++, VAUCANSON, from the EPITA Research and Development Laboratory, has a high computational performance, providing objects with abstraction level close to the mathematical objects by them represented. FADO, a library developed in PYTHON[2] for symbolic manipulation of automata and other computation models, is mainly intended to facilitate prototyping of new algorithms.

Written in Python, a new scientific computer program developed by Leonardo Bermeo Clavijo, Ph.D., at the Universidade Federal do Rio de Janeiro, was conceived for the development of algorithms for analysis and synthesis of discrete event systems modeled as automata, mainly objectifying to be an unified tool capable of integrating automata, graph algorithms and numerical calculations. The structure of the new software to be further on explained allows it to go beyond the cited programs previously presented, once it is a free open-source software that working conjointly with PYTHON's simplicity permit programmers to quickly develop new functions.

The requirements intended to be satisfy by the time of the conception of DESLab were

- Possibility of interaction with formal language software such as VAUCANSON;
- Possibility of integration with mathematical and graph analysis libraries such

[2]Powerful programming language having efficient high-level data structures and object-oriented programming. It is considered to be ideal for scripting and rapid application development in many areas on most plataforms. [12]

as NUMPY[3] and NETWORKX[4];

- Possibility of visualizing an automaton using Latex, GRAPHVIZ[5] and other scientific visualization of complex networks;

- Possibility of file interchangeability for graphics, graphs, automata and general data.

DESLab uses the automaton as its main computational object, implemented in the class **fsa**. Some other classes are worth to be mentioned. The class **Fsamath** defines a nondeterministic automaton into DESLab, while **Graph** is responsible for all the graph related operations. The class **Graphic** is intimately related to the state transition diagrams generated by DESLab.

The interaction with the program is done either through commands written in a shell or by means of scripts in a PYTHON development environment such as SPYDER[6]. After running codes, the output graphic result is generated by a Latex code and has several options of displacement. The software is free and open source, and can be downloaded from www.dee.ufrj.br/lca.

The work of developing a software is not done until it has been properly validated and verified. These processes concern checking whether or not the software in question meets its specifications and delivers the functionality expected. Even free software should be validated (in a sense of comparing the final product with the project and planning first intended) and verificated in order to prevent usage complications. Barry Boehm had best differentiated the two concepts by associating them with two sentences:

- *"Validation: Are we building the right product?"*

- *"Verification: Are we building the product right?"*

The brilliant explanation easily shows how validation is about ensuring how the final work actually meets the project, whereas verification involves command testing of functional and non-functional requirements. The entire validation and verification process is often called $V\ \&\ V$, and has the ultimate goal of establishing confidence that a software is actually 'fit for purpose' [4]. That done it is reasonably harder for an inattentive user to face errors.

Along with certifying a software, the need of explaining its usage to users is tremendously important since a developer hopes to attain the biggest number of users,

[3]Array-processing package written in Python and designed to manipulate large multi-dimensional arrays. [13]

[4]Python package developed for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. [14]

[5]Open source graph visualization software.

[6]Open source cross-platform IDE for Python language programming

and no one would be comfortable with the unknown. Writing a tutorial involves researching about the software objectives, fundamentals, theory and design. Testing, exemplifying and always searching for the easiest way of explaining procedures is routine for the writer. Moreover, it is determinant the use of empathy in order to predict possible complications a user may have and anticipate its proper explanations. A simple language avoiding ambiguity and tiredness is then thought so that the user can be comfortable and have his way guided through.

Capítulo 4

DESLab Tutorial

4.1 Basic Instructions of DESLab

4.1.1 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;

  - 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;

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

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 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;

Tabela 4.1: 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

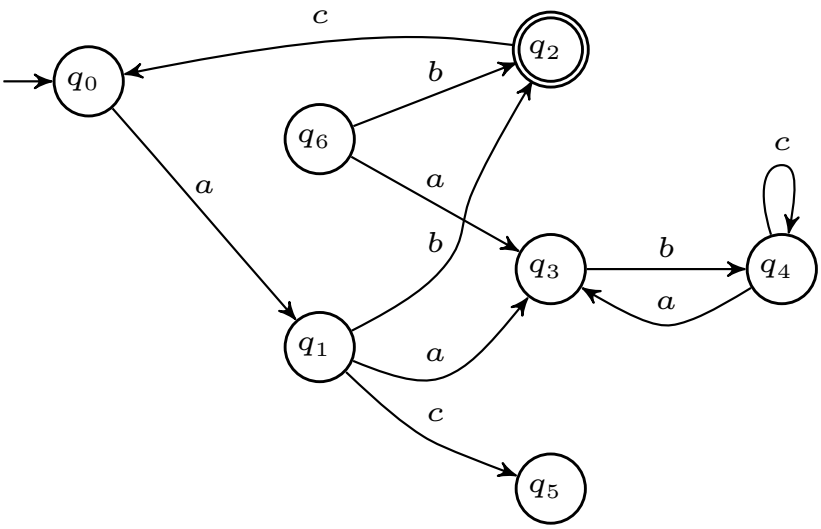
- Σ is the finite set of events associated with G;
- $f : X \times \Sigma \longrightarrow \Sigma$ is the transition function: *f(x,e)=y* meaning that there is an event e that labels the transition from state x to state y;
- $\Gamma : X \longrightarrow 2^{\Sigma}$ is the active event function (or feasible 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.

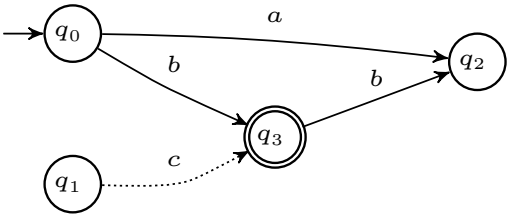
- Example
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 4.1(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$, shown in Figure 4.1(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 4.1(a)) and G_2 (shown in Figure 4.1(b)) are defined into DESLab by writing the following instructions.

```
from  deslab import *
```

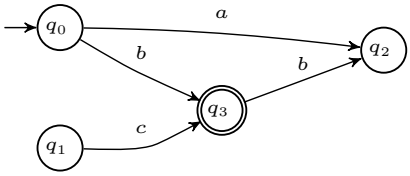
```
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# 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 .

Figura 4.1: Example of the definition of automata into DESLab.

- See also: Drawing a State Transition Diagram

4.1.2 Drawing a State Transition Diagram

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

- Syntax

$$draw(G, mode)$$

  - Inputs

    The input parameters are finite automata of the class fsa and 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.

  - Output
    The output are state transition diagrams according to the mode of exhibition.

- 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 labelled 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 labelling 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.

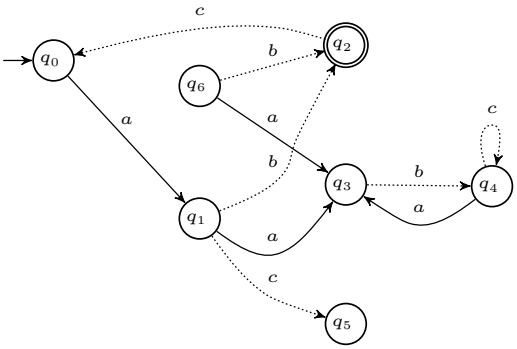
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$, where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) =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

  1. the state transition diagram of G_1 in black and white;

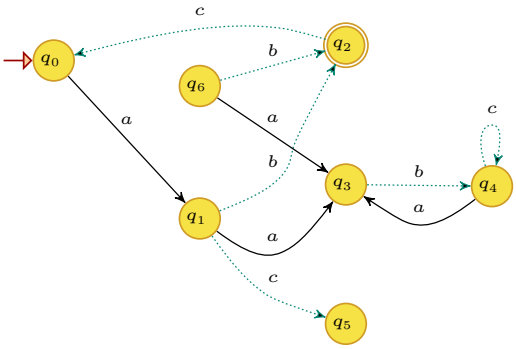
2. the state transition diagram G_1 in colors;
3. the state transition diagram of G_1 in beamer format.

Using DESLab , we can obtain automaton G_1 (shown in Figure 4.2(a), (b) and (c)) displayed respectively in all modes of exhibition required by writing the following instructions.

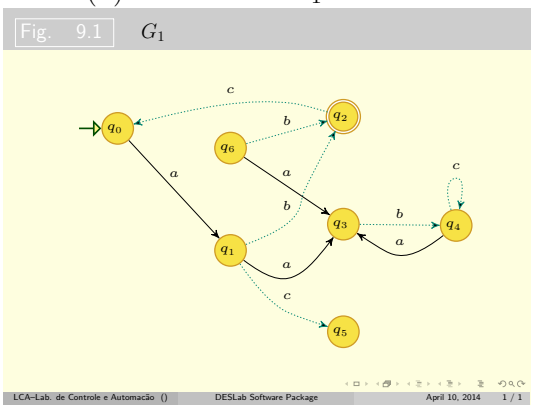
```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c ')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6')]

# 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.

Figura 4.2: Example of state transition diagrams.

- See also: Graph Algorithms

4.1.3 Adding States

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

- Syntax

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

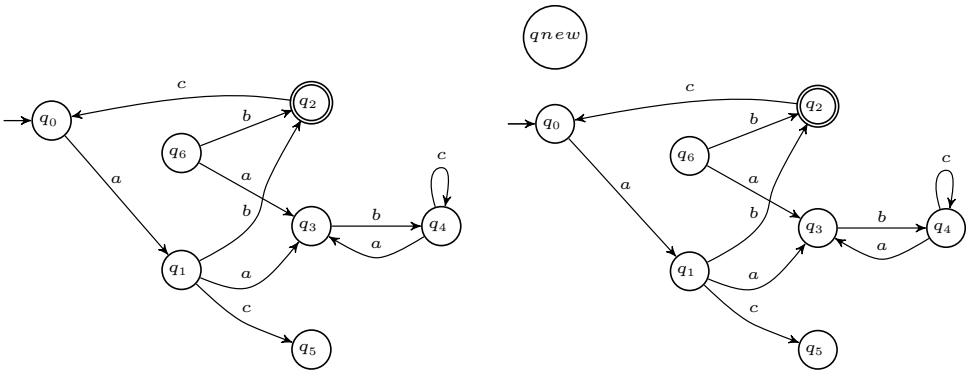
  - Inputs
    The input parameter is the state to be added.
  - Output
    The output is the input automaton with the new state added to its set of states.

- 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 4.3(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 4.3(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.

Figura 4.3: Example of adding a state to G_1.

- See also: Deleting States, Renaming States

4.1.4 Deleting States

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

- Syntax

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

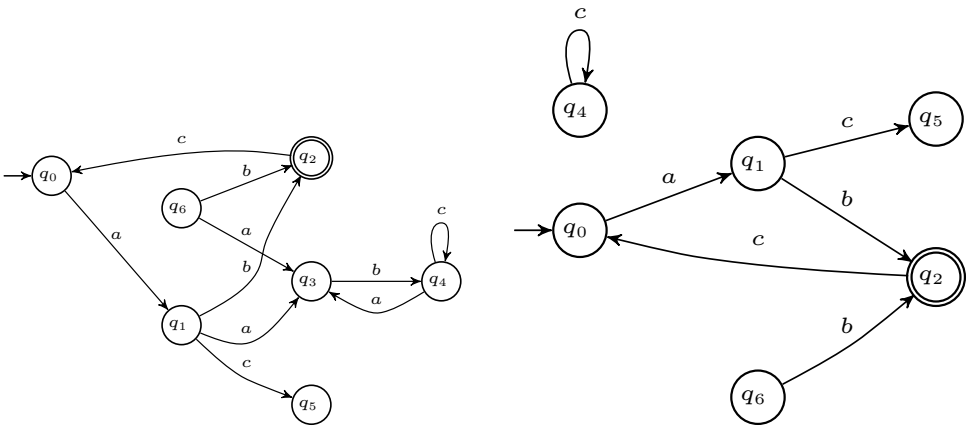
  - Inputs
    The input parameter is the state to be deleted.
  - Output
    The output is the previous automaton without the given state and all the transitions associated to it.

- 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 4.4(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 4.4(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.

Figura 4.4: Example of deleting a state from G_1.

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

4.1.5 Adding Events

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

- Syntax

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

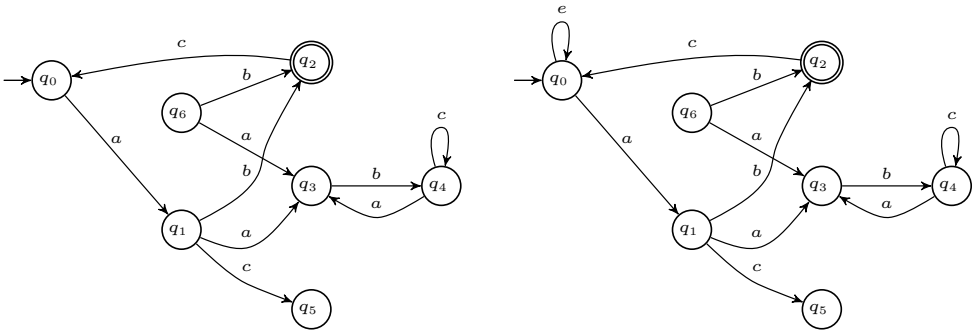
  - Inputs
    The input parameter is the list of events to be added.
  - Output
    The output is the addition of the list of events e to the set of events.

- 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 4.5(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 4.5(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 $q_0 \xrightarrow{e} q_0$.

Figura 4.5: Example of adding an event to G_1.

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

4.1.6 Deleting Events

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

- Syntax

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

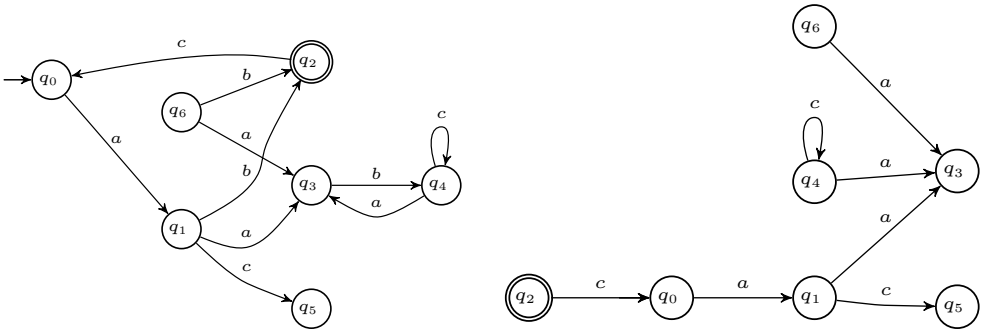
  - Inputs
    The input parameter is the list of events to be deleted.
  - Output
    The output is the input automaton without the events and all the transitions labelled by them.

- 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 4.6(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 4.6(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.

Figura 4.6: Example of deleting an event from G_1.

- See also: Adding Events

4.1.7 Adding Transitions

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

- Syntax

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

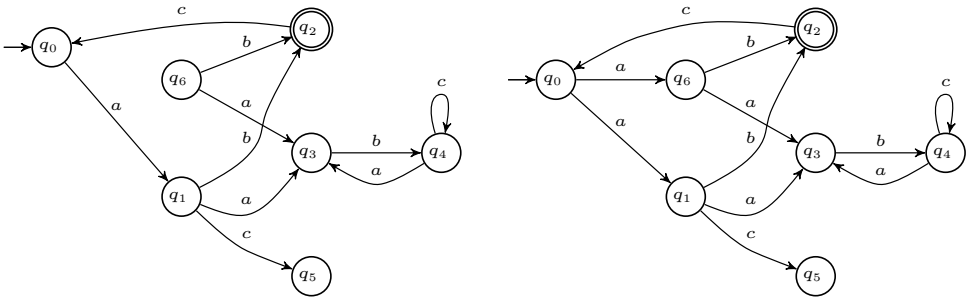
  - Inputs
    The input parameter is the transition to be added to the transition function, given by the states and the event that labels it.
  - Output
    The output is the input automaton with a new transition from state x to state y through the occurrence of the event a.

- 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 4.7(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 4.7(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 $q_0 \xrightarrow{a} q_6$

Figura 4.7: Example of adding a transition to G_1.

- See also: Deleting Transitions, Renaming Transitions

4.1.8 Deleting Transitions

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

- Syntax

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

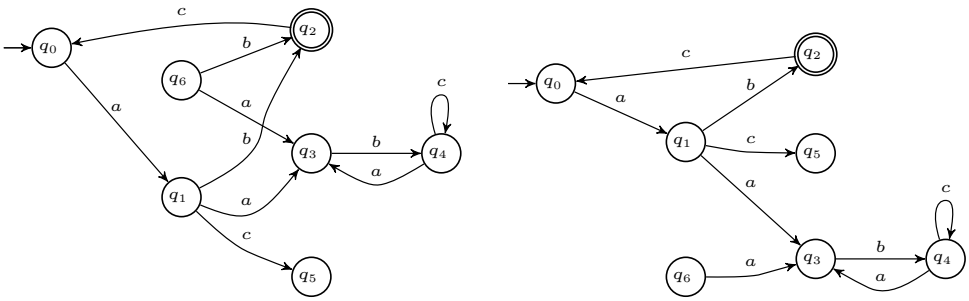
  - Inputs
    The input parameter is the transition to be deleted from the transition function, given by the states and the event that labels it.
  - Output
    The output is the previous automaton without the transition from state x to state y labelled by the event a.

- 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 4.8(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 4.8(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 $q_6 \xrightarrow{b} q_2$

Figura 4.8: Example of deleting a transition from G_1.

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

4.1.9 Adding Selfloops

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

- Syntax

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

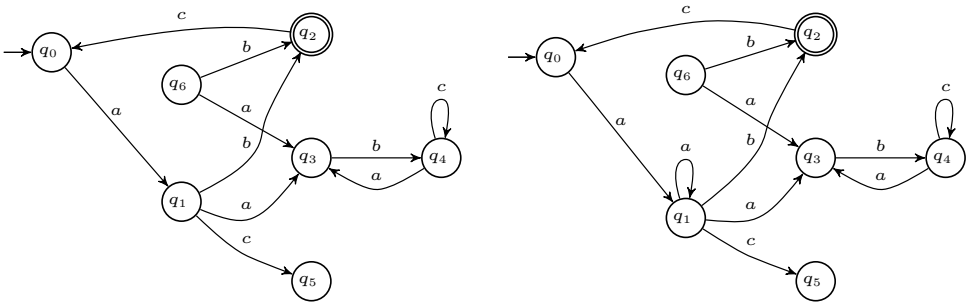
  - Inputs
    The input parameters are the state and the event that characterizes the selfloop to be added.
  - Output
    The output is the addition of a selfloop to the transition function.

- 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 4.9(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 4.9(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 $q_1 \xrightarrow{a} q_1$.

Figura 4.9: Example of adding a selfloop to G_1.

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

4.1.10 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:

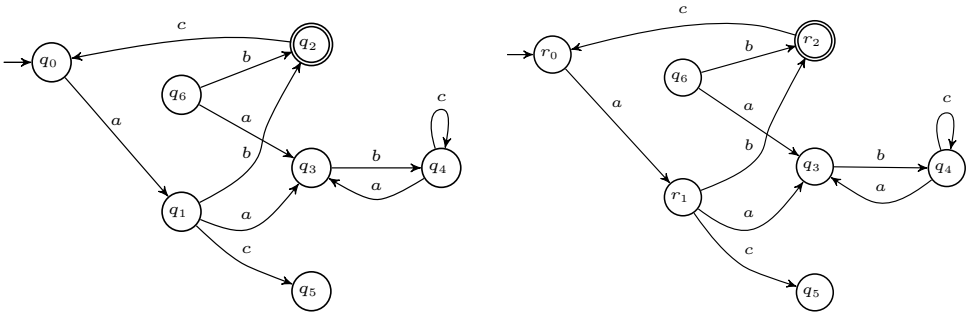
$$\text{mapping} = [(\text{old state label, 'new state label'})]$$

- 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 4.10(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 4.10(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.

Figura 4.10: Example of renaming states of G_1.

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

4.1.11 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:

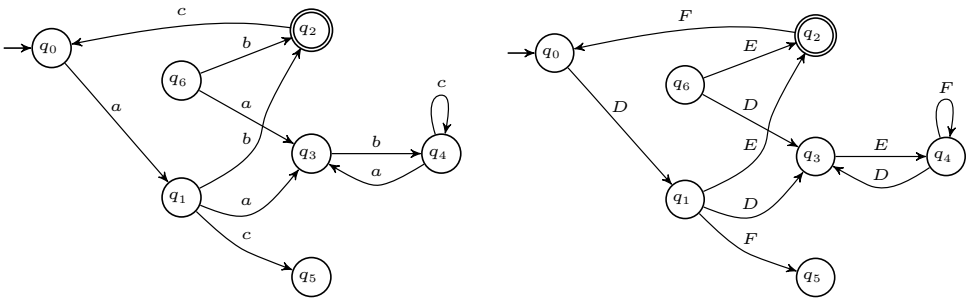
$$\text{mapping} = [(\text{old event label, 'new event label'})]$$

- 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 4.11(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 4.11(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

Figura 4.11: Example of renaming an event from G_1.

- See also: Adding Events, Deleting Events

4.1.12 Renaming Transitions

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

- Syntax

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

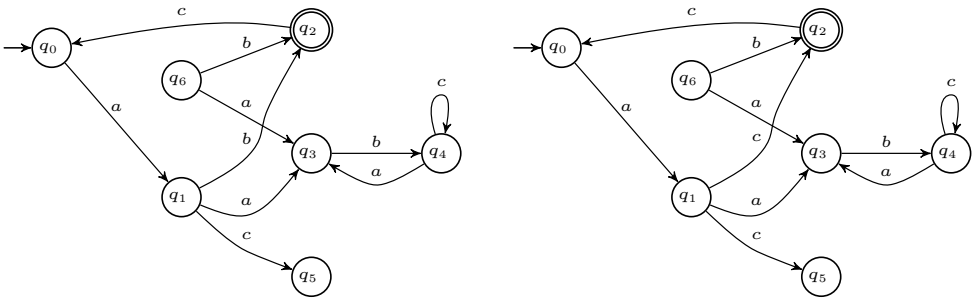
  - Inputs
    The input parameters are the states and events related to the transition to be renamed, where the old event comes first in the pair.
  - Output
    The output is the alteration of the event that labels the specified state transition.

- 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 4.12(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 4.12(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 $q_1 \xrightarrow{c} q_2$.

Figura 4.12: Example of renaming a transition of G_1.

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

4.1.13 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

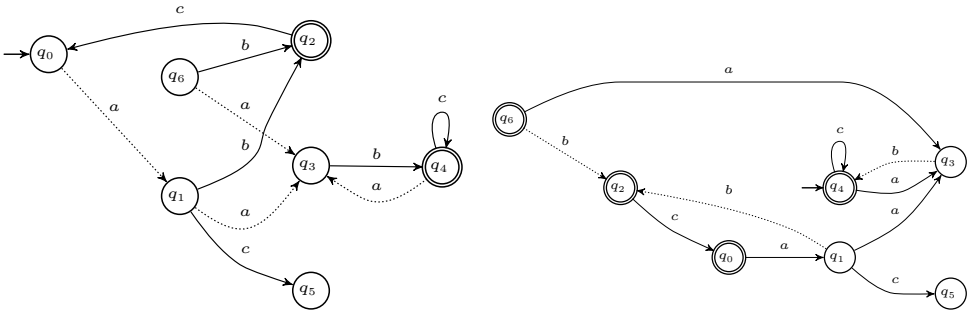
$$G_1 = G_1.setpar(property = value)$$

  - 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 4.13(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 4.13(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.

Figura 4.13: 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

4.1.14 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 4.14(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)
```

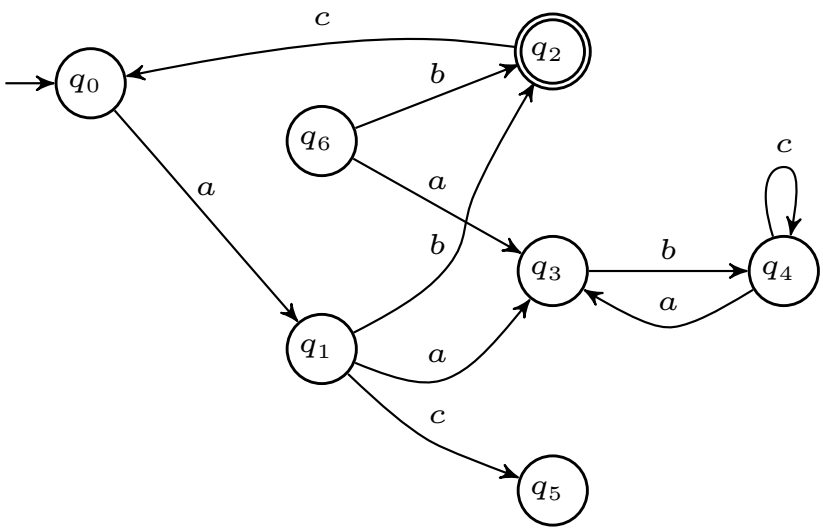


Figura 4.14: Automaton G_1 of the example of the number of states.

- 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

4.1.15 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 4.15(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)
```

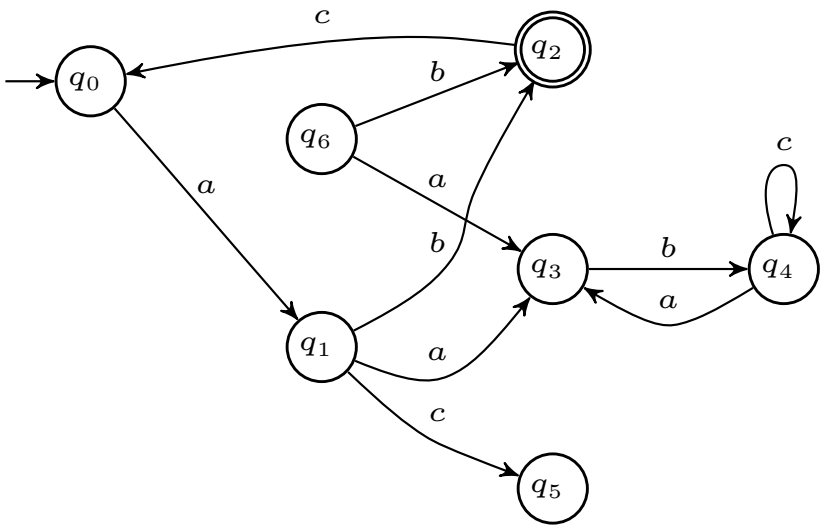


Figura 4.15: Automaton G_1 of the example of the number of transitions.

- 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

4.1.16 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 4.16(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()
```

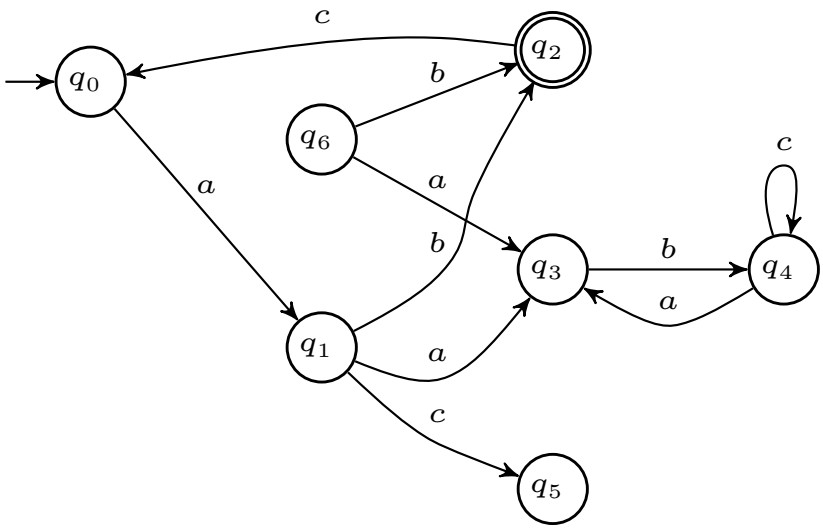


Figura 4.16: Automaton G_1 of the example of listing transitions.

- 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

4.2 Operations on Languages

4.2.1 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

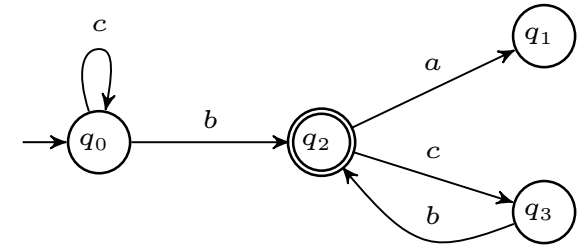
$$\begin{aligned} L_m(G) &= L_m(G_1)L_m(G_2) \\ &= \{s \in \Sigma^* : (s = s_1 s_2) \\ &\quad \wedge (s_1 \in L_m(G_1)) \wedge (s_2 \in L_m(G_2))\} \end{aligned} \tag{4.1}$$

  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 nondeterministic automaton that marks exactly $L_m(G_1)L_m(G_2)$.

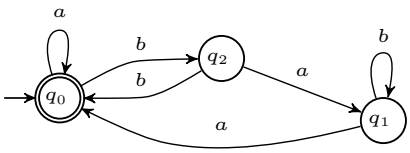
- Example
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 4.17(a) and (b) where $X_1 = \{q_0, q_1, q_2, q_3\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, c) = \{q_0\}$, $f_1(q_0, b) = \{q_2\}$, $f_1(q_2, a) = \{q_1\}$, $f_1(q_2, c) = \{q_3\}$, $f_1(q_3, b) = \{q_2\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ 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 4.17(c)) by writing the following instructions. Note that DESLab will automatically rename the states of G.

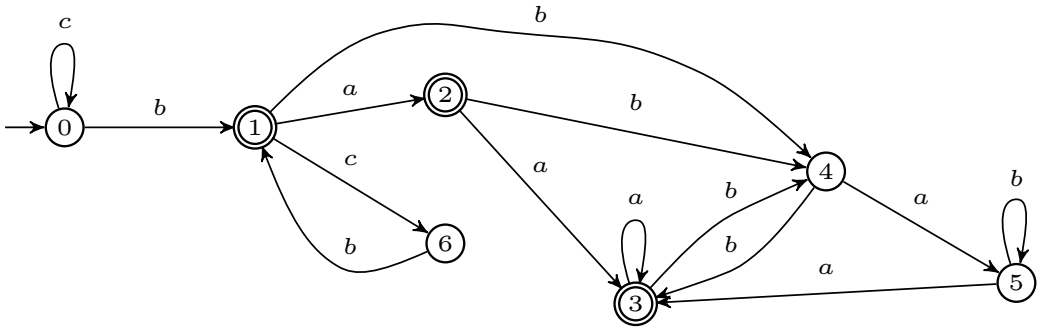
```
from  deslab import *
syms('q0 q1 q2 q3 a b c')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3')]

# 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 $G = G_1 * G_2$

Figura 4.17: Example of the concatenation operation.

- See also: Union

4.2.2 Union

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

- Syntax

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

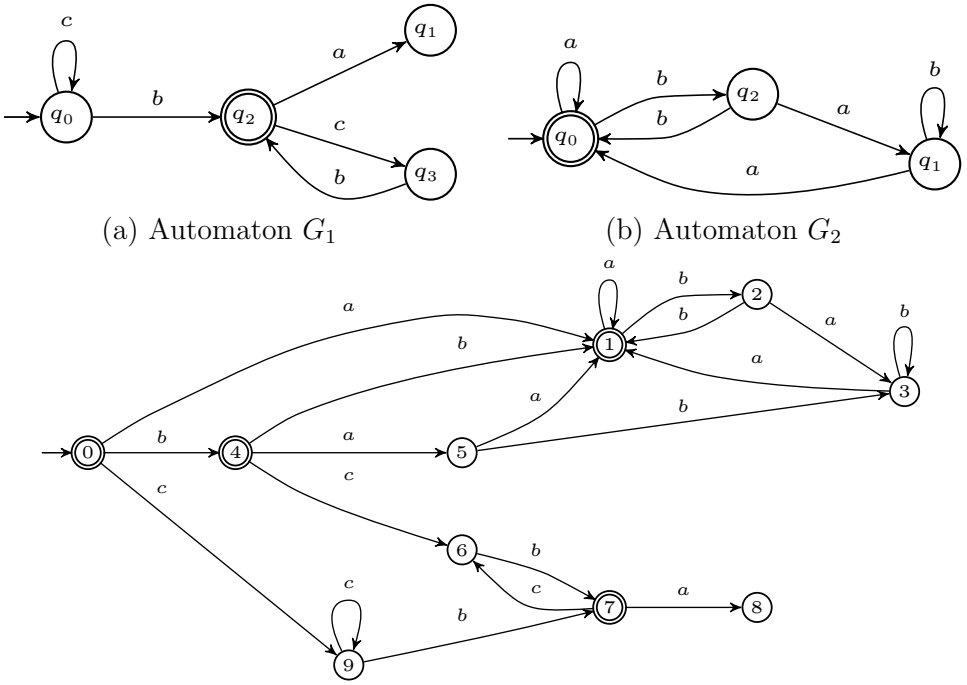
  - Inputs
    The input parameters are automata of the class fsa.
  - Output
    The output is an automaton marked by the language resulted from the union of the input automata marked languages.

- 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 4.17(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 4.18(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='$G_2$')

# union

G = G1+G2

# another possible notation

G = union(G1,G2)

draw(G2,'figure')
```



(a) Automaton G_1

(b) Automaton G_2

(c) Automaton $G = G_1 + G_2$

Figura 4.18: Example of the union operation.

- See also: Concatenation

4.2.3 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

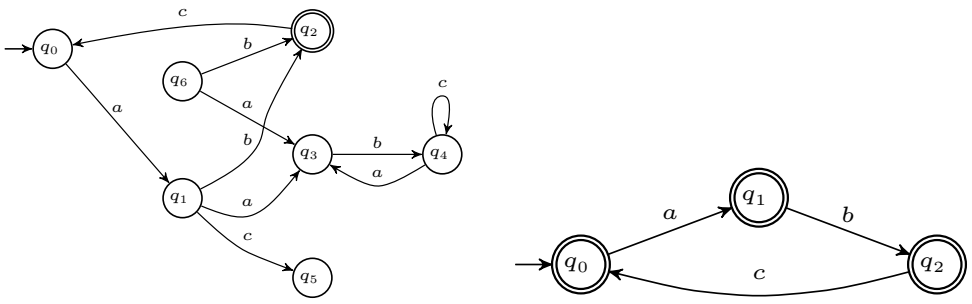
$$\overline{L_1} = \{s \in \Sigma^* : (\exists t \in \Sigma^*)[st \in L_1]\} \tag{4.2}$$

  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 4.19(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 4.19(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 $G = pclosure(G_1)$

Figura 4.19: Example of the prefix closure operation.

- See also: Kleene Closure, Kleene Closure Generator

4.2.4 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

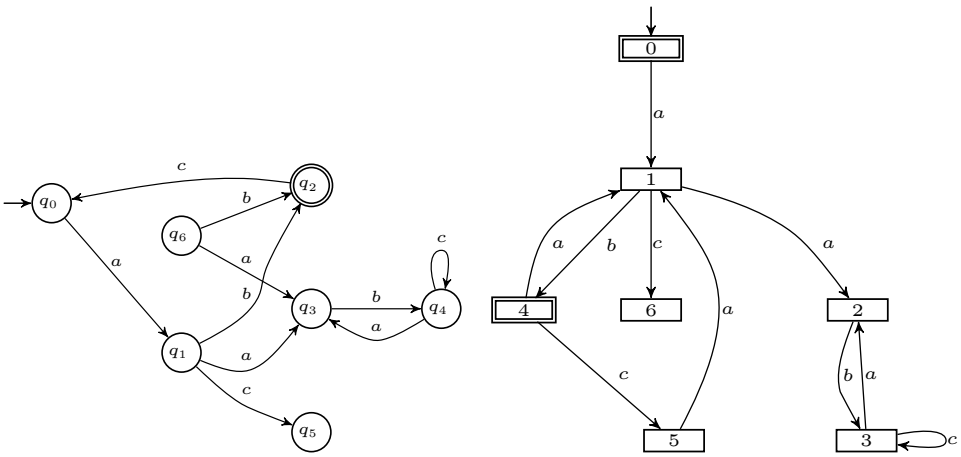
$$L^*_{m,1} := \{\varepsilon\} \cup L_{m,1} \cup L_{m,1}L_{m,1} \cup L_{m,1}L_{m,1}L_{m,1} \cup \cdots \tag{4.3}$$

  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 4.20(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 $G = kleeneclos(G1)$ (shown in figure 4.20(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='$G_1$')

# kleene closure

G = kleeneclos(G1)

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



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

Figura 4.20: Example of the Kleene closure operation.

- See also: Kleene Closure Generator, Prefix Closure

4.2.5 Kleene Closure Generator

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

- Syntax

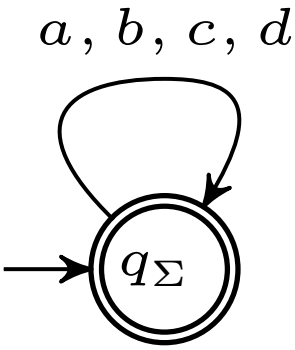
$$G = sigmakleeneclos(\Sigma, x_0 = value, label = value)$$

  - 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 4.21(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 $G = sigmakleenclos(\Sigma_{input})$

Figura 4.21: Example of the Kleene closure generator operation.

- See also: Kleene Closure, Prefix Closure

4.2.6 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 a predefined 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

$$\begin{array}{rcccl} P & : & \Sigma^* & \rightarrow & \Sigma_o^*,\ \Sigma_o \subseteq \Sigma \\ & & s & \mapsto & P(s), \end{array} \tag{4.4}$$

  satisfying the following properties:

$$\begin{array}{rcl} P(\varepsilon) & = & \varepsilon, \\ P(\sigma) & = & \begin{cases} \sigma, \text{ if } \sigma \in \Sigma_o, \\ \varepsilon, \text{ if } \sigma \in \Sigma \backslash \Sigma_o, \end{cases} \\ P(s\sigma) & = & P(s)P(\sigma), s \in \Sigma^*, \sigma \in \Sigma. \end{array} \tag{4.5}$$

  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

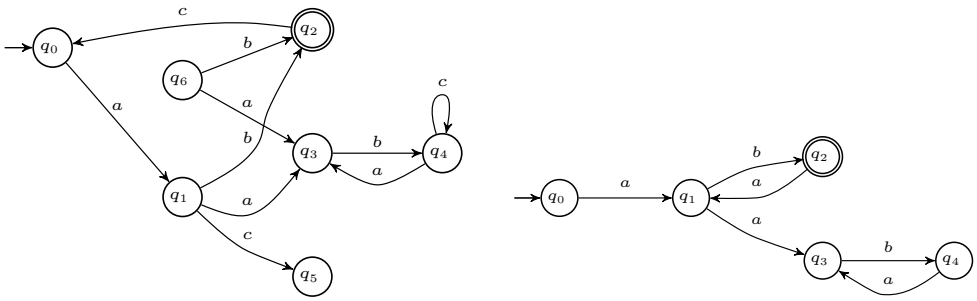
$$P(L) = \{t \in \Sigma_o^* : (\exists s \in L)[P(s) = t]\} \tag{4.6}$$

  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 4.22(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 4.22(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 $G = proj(G_1, Sigma_o)$

Figura 4.22: Example of the projection operation.

- See also: Inverse Projection

4.2.7 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 $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_{m_1})$ 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

$$P^{-1} : \Sigma_s^* \rightarrow 2^{\Sigma_1^*} \tag{4.7}$$

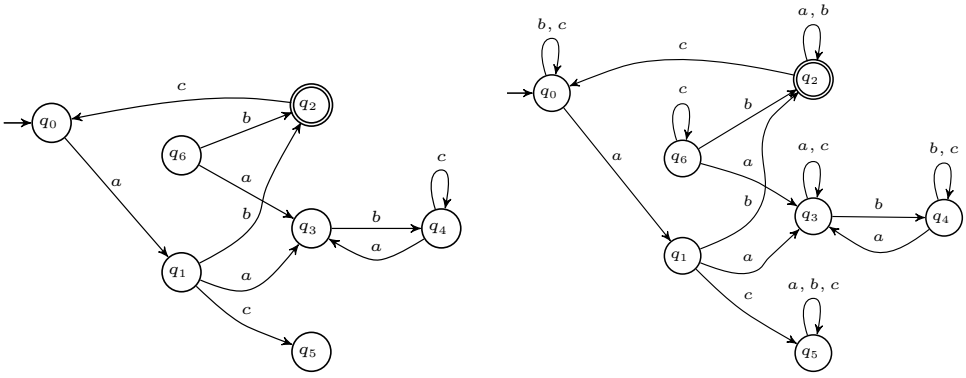
  returns the set of all strings from Σ_1^* that project to the given string. That extended to a language, here the generated (L) and marked (L_m) languages of the output, gives

$$P^{-1}(L) = \{s \in \Sigma_l^* : (\exists t \in L)[P(s) = t]\}, L \subseteq \Sigma_s^* \tag{4.8}$$

- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 4.23(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$, $\Sigma_s = \{a\}$. Using DESLab we can obtain automaton $G = invproj(G_1, \Sigma_s)$ (shown in figure 4.23(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]
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 $G = invproj(G_1)$

Figura 4.23: Example of the inverse projection operation.

- See also: Projection

4.2.8 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

$$L_D = L_{m_1} \backslash L_{m_2} = L_{m_1} \cap L_{m_2}^c \tag{4.9}$$

  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.

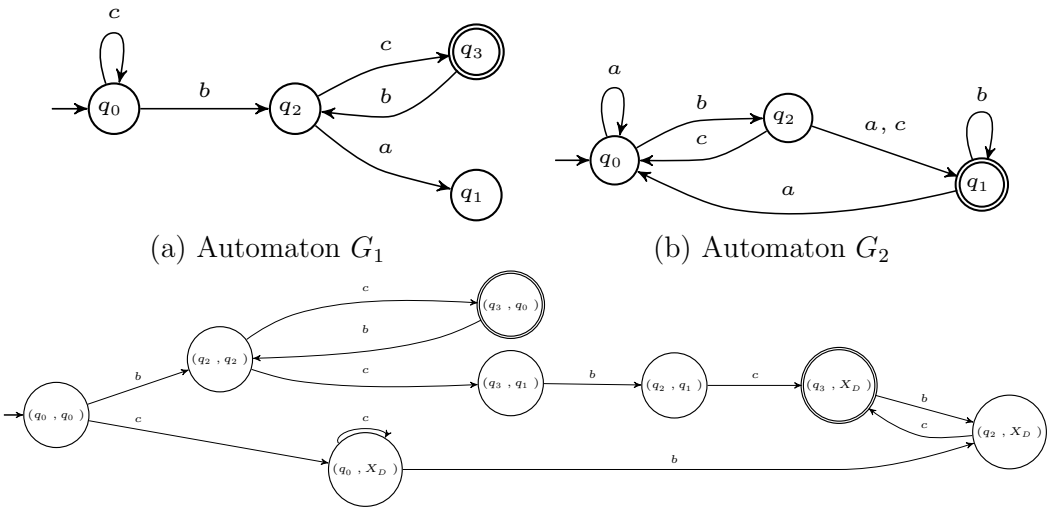
- Example

  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 4.17(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 4.24(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 $G_D = G_1 - G_2$

Figura 4.24: Example of the language difference operation.

- See also: Concatenation, Language Quotient

4.2.9 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:

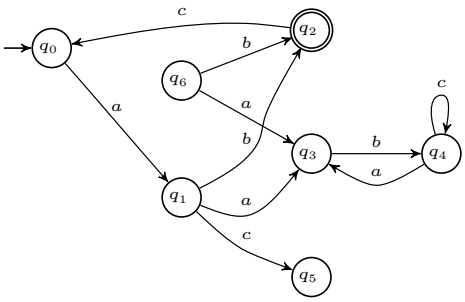
$$L(G_1)/L(G_2) := \{s \in \Sigma^* : (\exists t \in L(G_2))[st \in L(G_1)]\} \tag{4.10}$$

- 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 4.25(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 4.25(c)) by writing the following instructions.

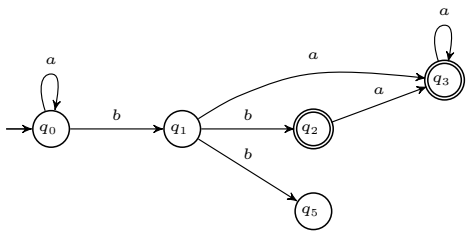
```
from  deslab import *
syms('q0 q1 q2 q3 q4 q5 q6 a b c Gquo')
table = [(q0,'q_0'), (q1,'q_1'), (q2,'q_2'), (q3,'q_3'),
(q4,'q_4'), (q5,'q_5'), (q6,'q_6'), (Gquo,'$G_{quo}$')]
```

```
# 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')
```

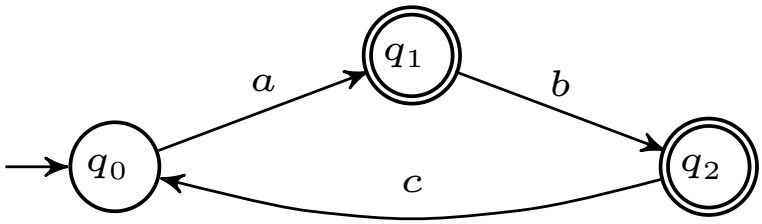
- See also: Language Difference



(a) Automaton G_1



(b) Automaton G_2



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

Figura 4.25: Example of the language quotient operation.

4.3 Unary and Composition Operations

4.3.1 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:

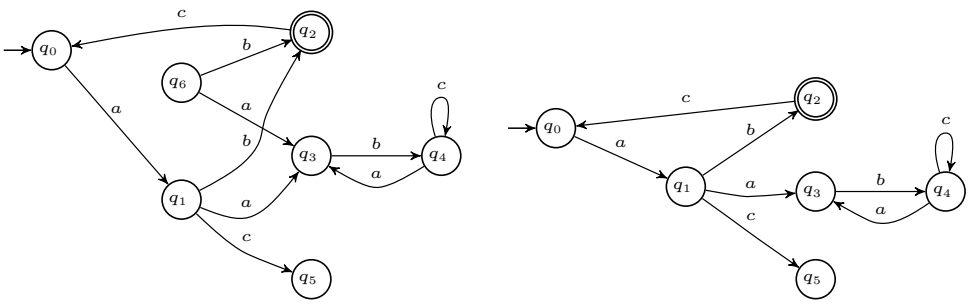
$$\begin{aligned} Ac(G_1) &:= (X_{ac,1}, \Sigma_1, f_{ac,1}, x_{0,1}, X_{ac,m_1}) \text{ where} \\ X_{ac,1} &= \{x \in X_1 : (\exists s \in \Sigma_1^*)[f_1(x_0, s) = x]\} \\ X_{ac,m_1} &= X_{m,1} \cap X_{ac,1} \\ f_{ac,1} &= f_1 \mid X_{ac,1} \times \Sigma_1 \to X_{ac,1} \end{aligned} \tag{4.11}$$

  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 4.26(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 4.26(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) $G = ac(G1)$

Figura 4.26: Example of the accessible part operation.

- See also: Coaccessible Part

4.3.2 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:

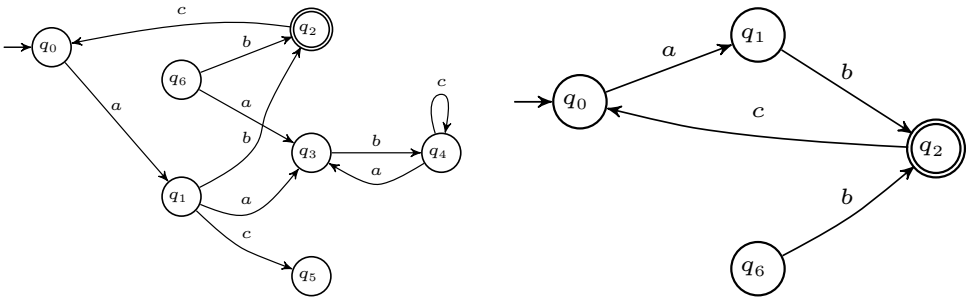
$$\begin{array}{lcl} CoAc(G_1) & := & (X_{coac,1}, \Sigma_1, f_{coac,1}, x_{0,coac_1}, X_{m,1}) \text{ where} \\ X_{coac,1} & = & \{x \in X_1 : (\exists s \in \Sigma_1^*)[f_1(x,s) \in X_{m,1}]\} \\ f_{coac,1} & = & f_1 \mid X_{coac,1} \times \Sigma_1 \to X_{coac,1} \\ x_{0,coac_1} & = & \begin{cases}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

  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 4.27(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 $G = coac(G_1)$ (shown in figure 4.27(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='$G_1$')

# coaccessible part

G = coac(G1)


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



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

Figura 4.27: Example of the coaccessible part operation.

- See also: Accessible Part

4.3.3 Trim

- Purpose
  This operation returns the trim of automaton.

- Syntax

$$G = trim(G_1)$$

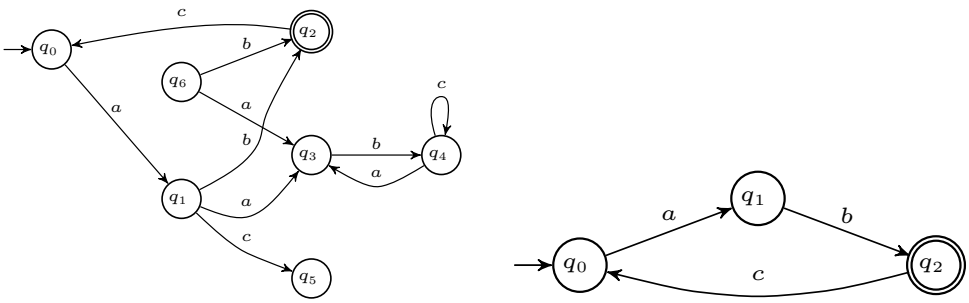
  - Inputs
    The input parameter is a finite automaton of the class fsa.
  - Output
    The output is the resulting of the trim of the input automaton.

- 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 4.28(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 $G = trim(G_1)$ (shown in figure 4.28(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='$G_1$')

# trim

G = trim(G1)

draw(G,'figure')
```



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

Figura 4.28: Example of the trim operation.

- See also: Accessible Part, Coaccessible Part

4.3.4 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.

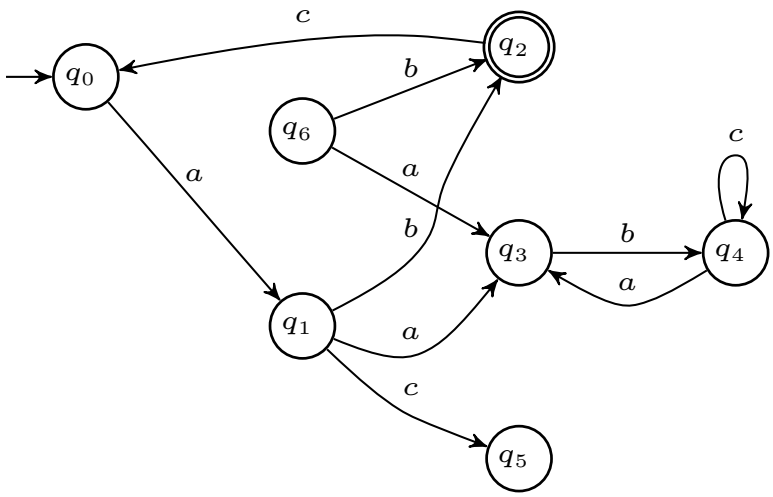
  1. Take the trim part of G_1;
  2. Add a new state x_d to X_1;
  3. Complete f_1. In order to do so, create transitions from every $x \in X_1$ to x_d labbeled by events $e \in \Sigma_1 \backslash \Gamma(x)$;
  4. Change the marking status of the states of G_1 by unmarking the marked states and marking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

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

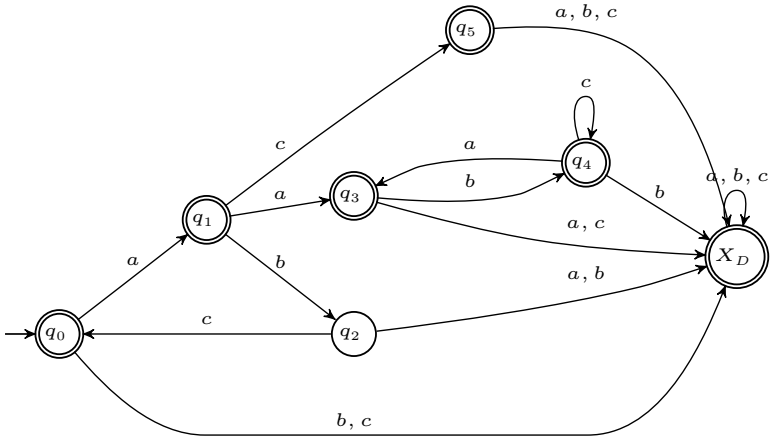
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 4.29(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = G_1$ (shown in figure 4.29(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='$G_1$')

# complement of G1

G = ~(G1)

# another notation

G = complement(G1)


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



(a) Automaton G_1



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

Figura 4.29: Example of the complement operation.

- See also: Complete Automaton

4.3.5 Complete Automaton

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

- 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

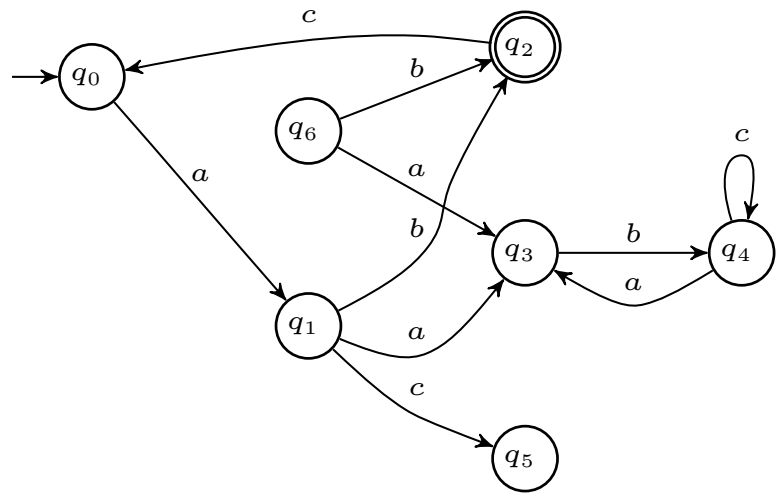
$$f_{complete} = \begin{cases} f_1(x,e) & \text{if } e \in \Gamma_1(x) \\ x_d & \text{if } e \notin \Gamma_1(x) \vee x = x_d, \forall\ e \in \Sigma \end{cases}$$

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

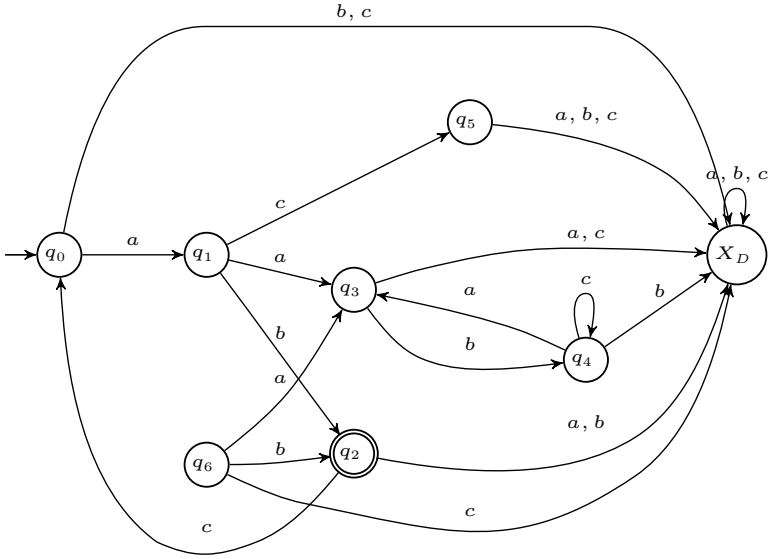
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 4.30(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can obtain automaton $G = complete(G1)$ (shown in figure 4.30(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='$G_1$')

# complete automaton

G2 = complete(G1)

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



(a) Automaton G_1



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

Figura 4.30: Example of the complete automaton operation.

- See also: Comparison Instructions, Empty Automaton, Complement

4.3.6 Empty Automaton

- Purpose
  This operation returns an empty automaton.

- Syntax

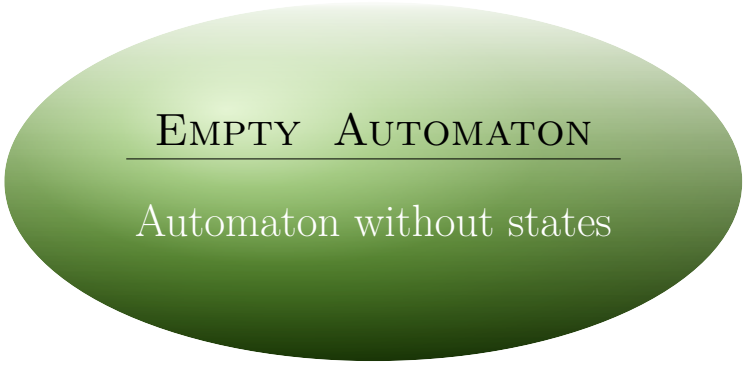
$$G1 = fsa()$$

  - 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 4.31(b)) by writing the following instructions:

```
from  deslab import *

# empty automaton definition G1

G1 = fsa()

draw(G1,'figure')
```



Empty Automaton G_1

Figura 4.31: Example of the empty automaton operation.

- See also: Comparison Instructions, Complete Automaton

4.3.7 Product

- Purpose

  This operation returns the product between automata.

- Syntax

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

  - Inputs

    The input parameters are finite automata of the class fsa.

  - Output

    The output is an automaton that 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 follows.

$$G_1 \times G_2 := Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f, \Gamma_{1\times 2}, (x_{01}, x_{02}), X_{m,1} \times X_{m,2}) \quad (4.12)$$

where

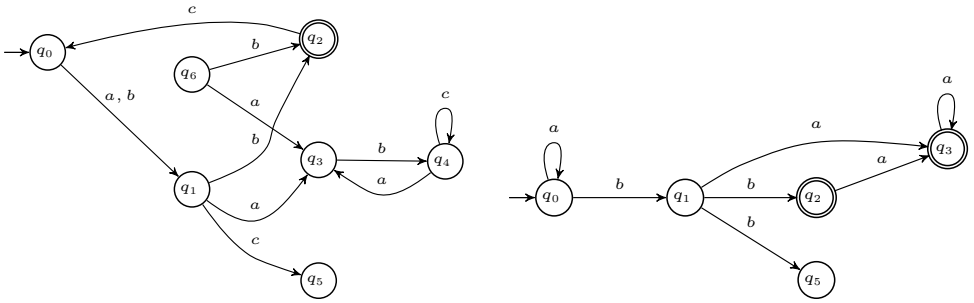
$$f((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), f_2(x_2, e)), & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined}, & \text{otherwise} \end{cases} \quad (4.13)$$

$$\Gamma_{1\times 2}(x_1, x_2) = \Gamma_1(x_1) \cap \Gamma_2(x_2). \quad (4.14)$$

- Example

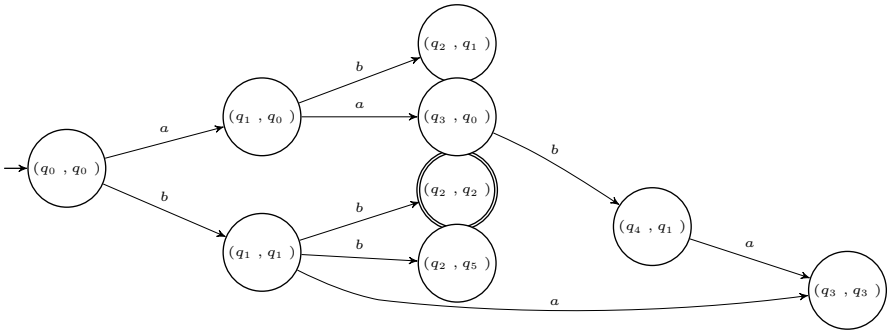
  Consider the deterministic automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 4.32(a) and (b) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_0, b) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ 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 4.32(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 $G = G_1 \times G_2$

Figura 4.32: Example of the product operation.

- See also: Parallel Composition

4.3.8 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.

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

  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:

$$f((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), f_2(x_2, e)), & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, e), x_2), & \text{if } e \in \Gamma_1(x_1) \setminus \Sigma_2 \\ (x_1, f_2(x_2, e)), & \text{if } e \in \Gamma_2(x_2) \setminus \Sigma_1 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

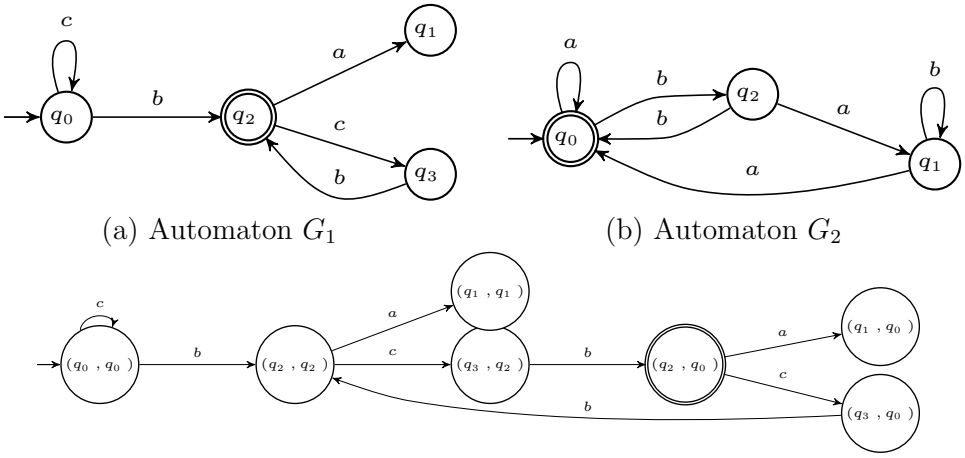
$$\Gamma_{1\|2}(x_1, x_2) = [\Gamma_1(x_1) \cap \Gamma_2(x_2)] \cup [\Gamma_1(x_1) \setminus \Sigma_2] \cup [\Gamma_2(x_2) \setminus \Sigma_1].$$

  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 4.17(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 4.33(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 $G = G_1 \| G_2$

Figura 4.33: Example of the parallel composition operation.

- See also: Product

4.3.9 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

$$f_{obs}(B, e) := UR(x_1 \in X_1 : (\exists x_e \in B)[x_1 \in f_1(x_e, e)]) \tag{4.15}$$

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.

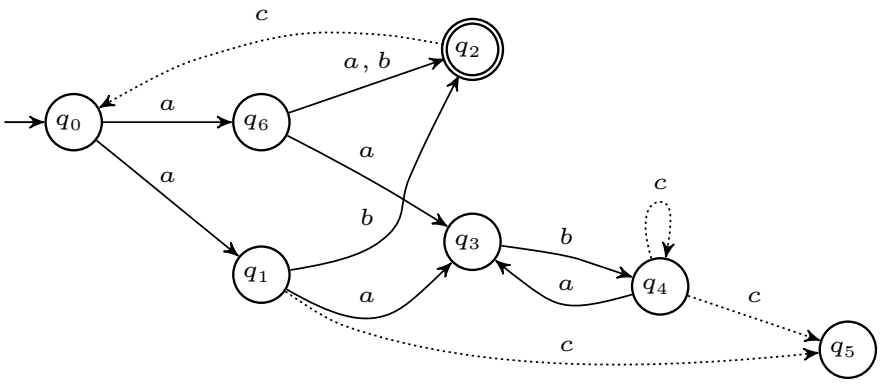
Step 3: Repeat Step 2 until the entire accessible part of G_{obs} has been constructed.

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

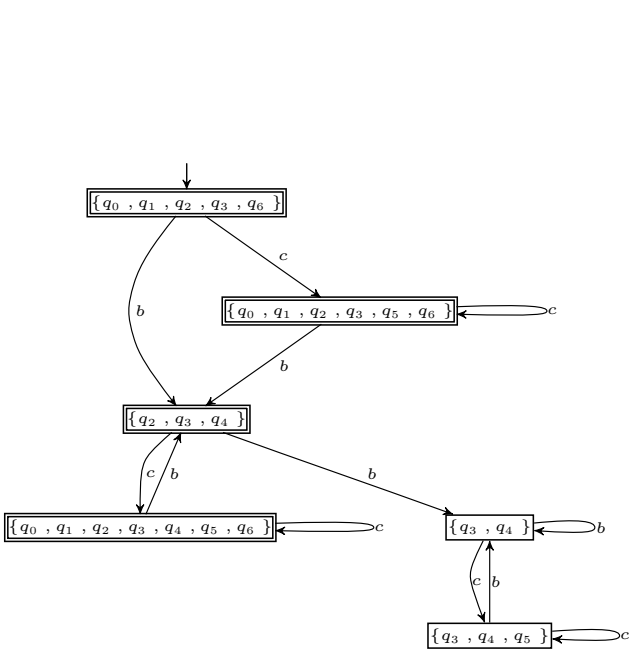
- Example
  Consider the deterministic automaton $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 4.34(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_0, a1) = \{q_6\}$ $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, c) = \{q_5\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $f_1(q_6, a) = \{q_2\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. It is desired that two automata are generated, being the first one according to a given set of observable events. Using DES-Lab we can obtain automata $G_{obs,1}$ and $G_{obs,2}$ (shown in figure 4.34(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)
```

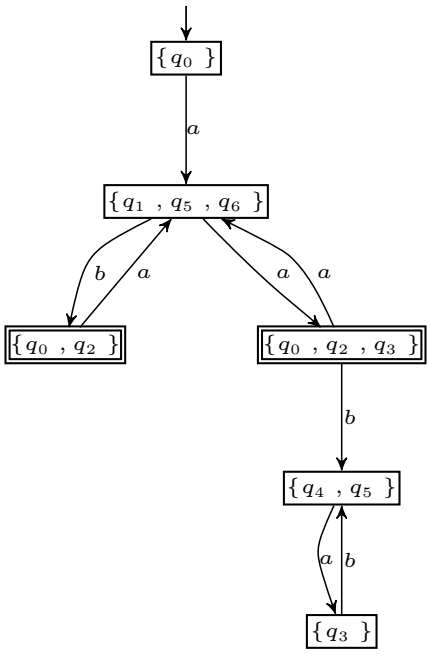
```
draw(G1, Gobs1, Gobs2,'figure')
```



(a) Automaton G_1



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



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

Figura 4.34: Example of the observer operation.

- See also: Projection, Inverse Projection, Epsilon Observer

4.3.10 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 4.3.9. 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

$$f_{obs}(B, e) := \varepsilon R(x_1 \in X_1 : (\exists x_e \in B)[x_1 \in f_{nd}(x_e, e)]) \tag{4.16}$$

  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

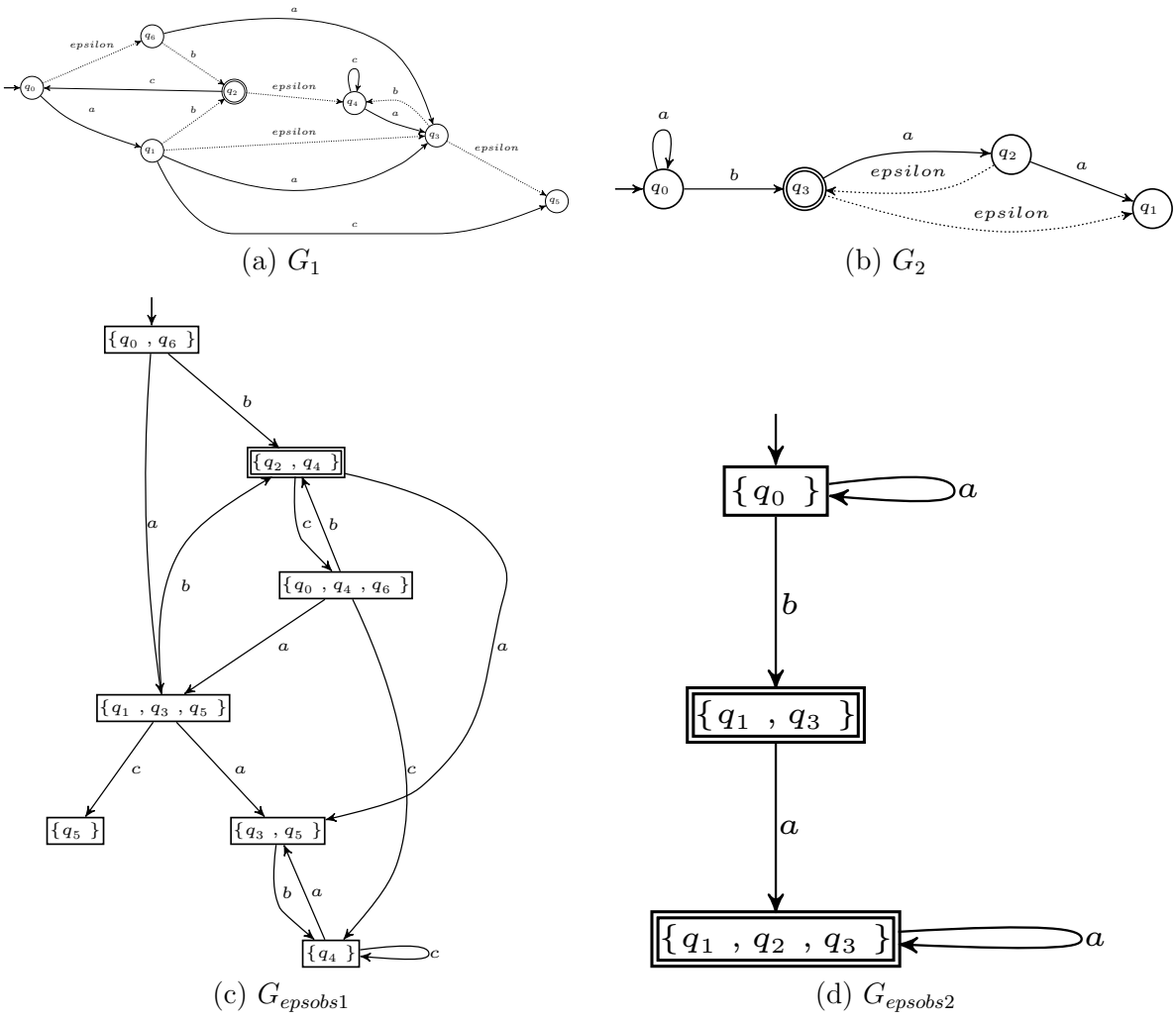
  Consider the automata $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 4.35(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c, \varepsilon\}$,

$Sigobs_1 = \{a, c\}$, $f_1(q_0, a) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $f_1(q_0, \varepsilon) = \{q_6\}$, $f_1(q_2, \varepsilon) = \{q_4\}$, $f_1(q_1, \varepsilon) = \{q_3\}$, $f_1(q_3, \varepsilon) = \{q_5\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$ and $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, X_{0,2}, X_{m,2})$ shown in Figure 4.35(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 4.36(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 4.35(c)), $G_{obs1} = observer(G1)$ (shown in figure 4.36(e)) $G_{epsobs2} = epsilonobserver(G2)$ (shown in figure 4.35(d)), $G_{obs2} = observer(G2)$ (shown in figure 4.36(f)), $G_{epsobs3} = epsilonobserver(G3)$ (shown in figure 4.36(h)), and $G_{obs3} = observer(G3)$ (shown in figure 4.36(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')
```

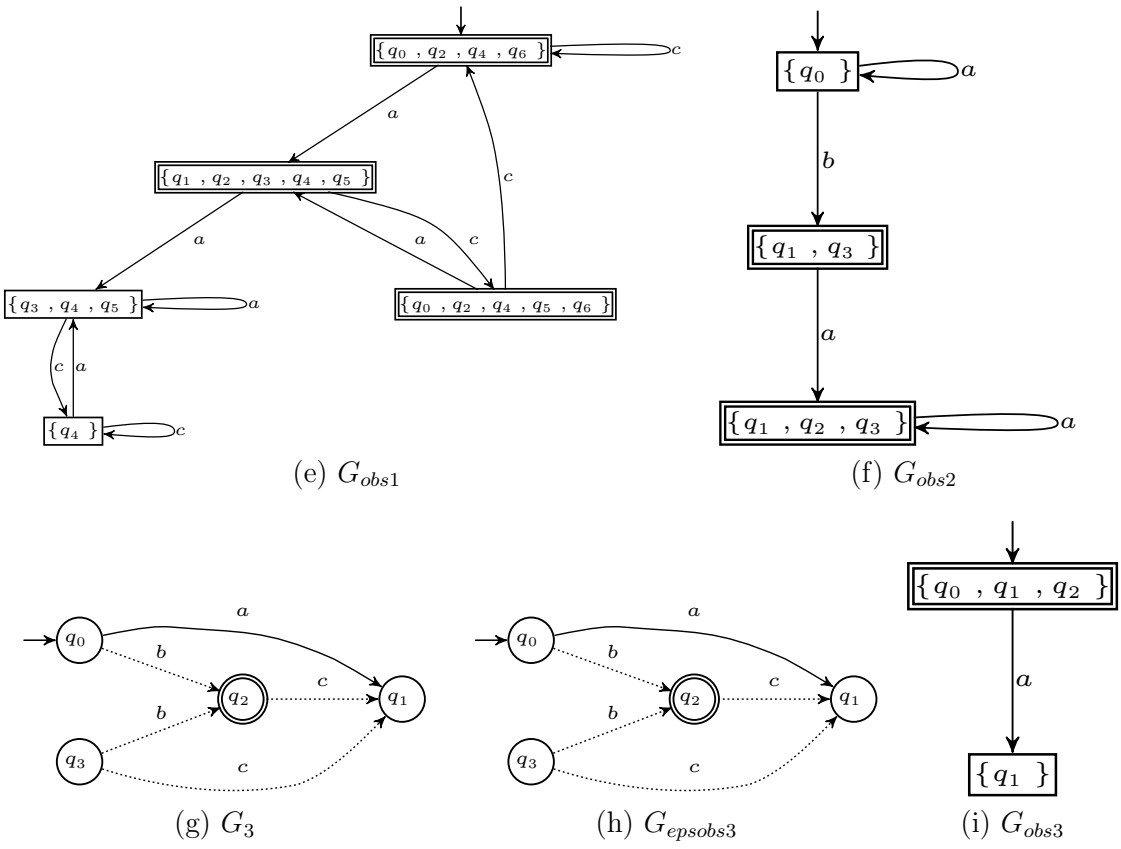
- See also: Projection, Inverse Projection, Observer Automaton



(a) G_1 (b) G_2

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

Figura 4.35: Example of the epsilon observer operation.



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

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

Figura 4.36: Example of the epsilon observer operation.

4.4 Additional Instructions of DESLab

4.4.1 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.

Tabela 4.2: 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.

Tabela 4.3: 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 $L_m(G_1) = \emptyset$
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 4.37(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

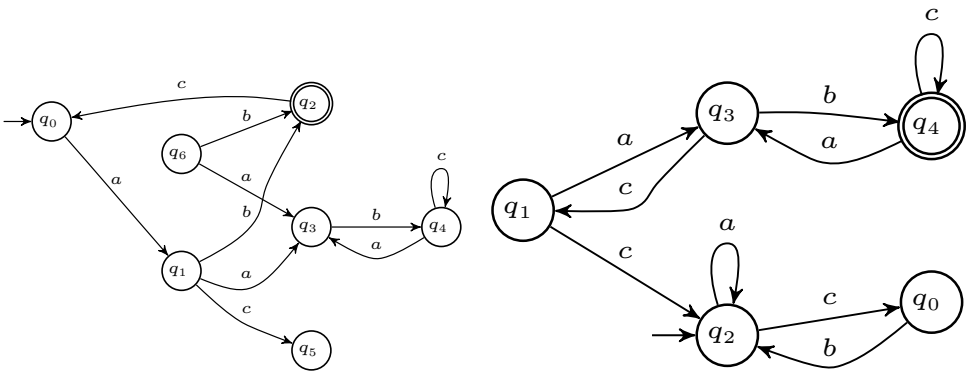
  G2>=G1
  >>>False

  isitempty(G1)
  >>>False

  G1==G2
  >>>False

  G1<>G2 >>>True

  isitcomplete(G1)
  >>>False



(a) Automaton G_1 (b) Automaton G_2

Figura 4.37: Example of the comparison instructions.

- See also: Graph Algorithms

4.4.2 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 4.4.

Tabela 4.4: 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 $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_{m_1})$ 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 $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 4.38(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 investigate the graph of G_1 (shown in Figure 4.38(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='$G_1$')

# All the commands must be run in the console:

# accessing graph of G1

g = G1.Graph # Attempt for the upper case G on 'Graph'

# finding strongly connected components

strconncomps(g)
strconncomps(G1)

# finding states with self loops

selfloopnodes(G1)
 selfloopnodes(g)

# running a depth first search

dfs(g, G1)
```

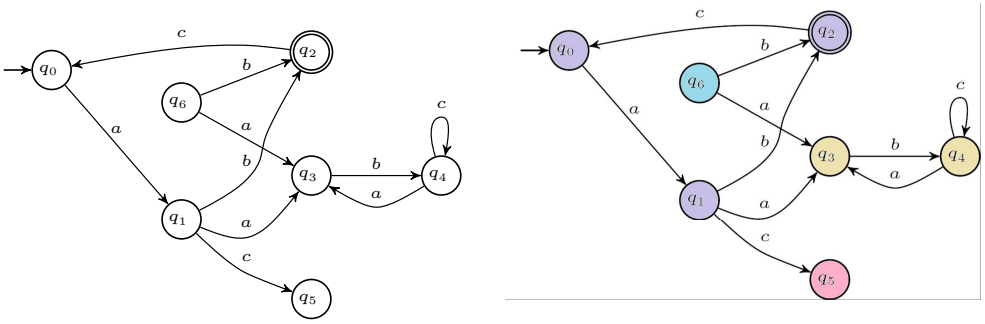
- Console outputs
  The response to the example, which should be plugged in the console, is:

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

  strconncomps(G1)
  >>>[['q1', 'q2', 'q0'], ['q3', 'q4'], ['q5'], ['q6']]

  selfloopnodes(G1)
  >>>['q4']

  dfs(g, G1)
  >>>frozenset(['q1', 'q0', 'q3', 'q2', 'q5', 'q4', 'q6'])



(a) Automaton G_1

(b) Automaton G_1 with the strongly connected components sorted by colors.

Figura 4.38: Example of using graph algorithms.

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

4.4.3 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.

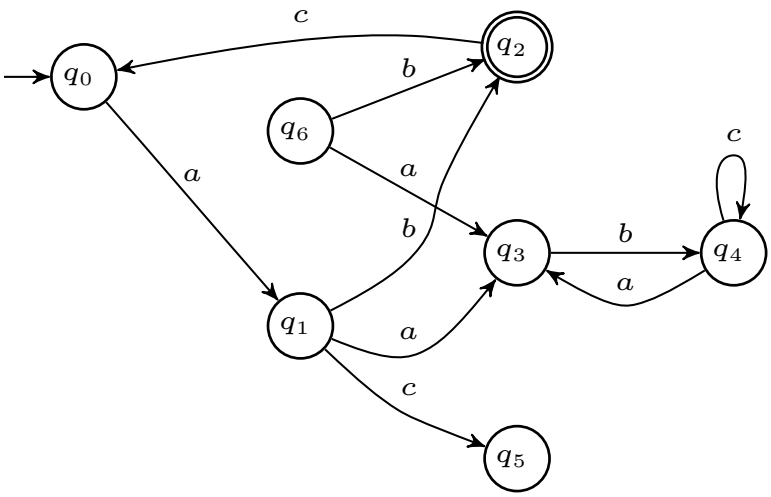
- Description
  Redefining graphical properties can be handy for the user willing to write routines and explore the uses of DESLab. Verifiers and diagnosers, for instance, can be implemented using the commands available. The graphical properties become then relevant since it brings a better presentation 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 4.39(a) where $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$,

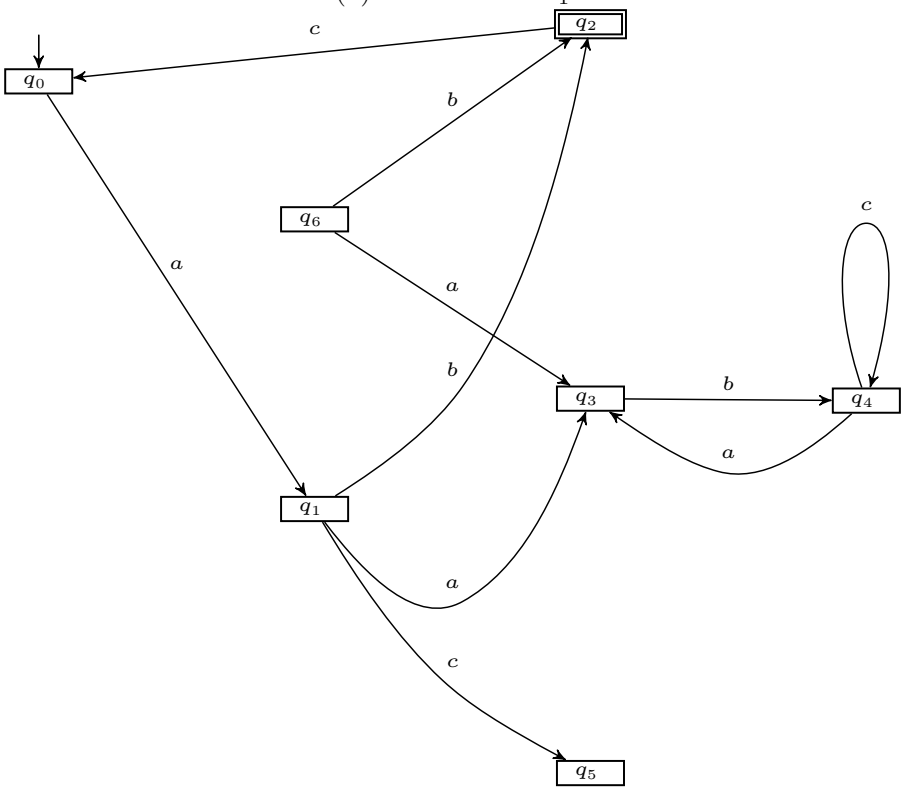
$f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Is is required that the states are represented by rectangles, with left to right orientation and node and rank separation both equal to 1. The requirements can be met by running the following instructions.

```
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.

Figura 4.39: Example of redefining graphical properties.

4.4.4 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.

Tabela 4.5: Syntax of the lexicographical features

Syntax	Brief description
$lexgraph\_dfs(G1)$	list depth first searched states
$lexgraph\_alphamap(G1)$	list the shortest paths to states
$lexgraph\_numbermap(G1)$	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_1 with the string composed by the shortest path conducting to each state.

    Number mapping returns a dictionary associating the accessible states of G_1 with a string composed by a number that represents the order of appearance of the state in the lexicographical depth search.

- Description

  Let $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_{m_1})$ denote a finite state automaton. There are cases when a closely search through the transitions is needed. For 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

[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.

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 $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, X_{0,1}, X_{m,1})$ shown in Figure 4.40 $X_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma_1 = \{a, b, c\}$, $f_1(q_0, a1) = \{q_1\}$, $f_1(q_1, b) = \{q_2\}$, $f_1(q_1, a) = \{q_3\}$, $f_1(q_1, c) = \{q_5\}$, $f_1(q_2, c) = \{q_0\}$, $f_1(q_3, b) = \{q_4\}$, $f_1(q_4, c) = \{q_4\}$, $f_1(q_4, a) = \{q_3\}$, $f_1(q_6, b) = \{q_2\}$, $f_1(q_6, a) = \{q_3\}$, $X_{0,1} = \{q_0\}$, $X_{m,1} = \{q_2\}$. Using DESLab we can 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 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='$G_1$')

# running a lexicographical depth search on G1

lexgraph_dfs(G1)

# running a lexicographical string mapping of G1

lexgraph_alphamap(G1)

# running a lexicographical number mapping of G1

lexgraph_numbermap(G1)
```

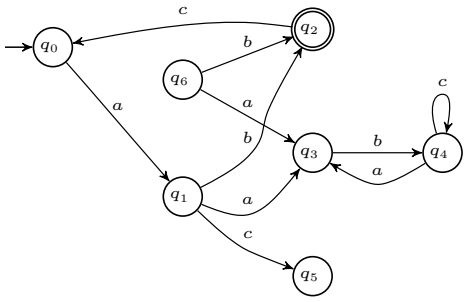


Figura 4.40: Example of running the lexicographical features.

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

  lexgraph_dfs(G1)
  >>>['q0', 'q1', 'q3', 'q4', 'q2', 'q5']

  lexgraph_alphamap(G1)
  >>>'q1': 'a', 'q0': 'epsilon', 'q3': 'aa', 'q2': 'ab','q5': 'ac', 'q4': 'aab'

  lexgraph_numbermap(G1)
  >>>'q1': 1, 'q0': 0, 'q3': 2, 'q2': 4, 'q5': 5, 'q4': 3

- See also: Graph Algorithms

Capítulo 5

Conclusion

This work intended to validate and verify the scientific computing program DESLab, along with developing its tutorial. The relevance is explicit since a new software that has not been properly tested does not inspire confidence to the scientific community, may being passive of error encounter and disagreement of results according to a given expectation. Besides, a tutorial is primordial in a sense of guiding a user through the software usage.

During the work, the mistakes found and barriers encountered were determinant to elucidate the theory regarding verification and validation processes. Commands incapable of running, difficulties faced in daily bases usage showed that a software needs some testing and attention payed before released to costumers indeed.

A possible future plan relevant to this work would be the online displacement of the tutorial, providing it with more interactive features and best aesthetic presentation when implemented in a website such as www.scipy.org[1].

[1]Python-based ecosystem of open-source software for mathematics, science, and engineering

Referências Bibliográficas

[1] RIFKIN, J. *The End of Work: Decline of the Global Labor Force and the Dawn of the Post-market Era*, v. 1. New York, NY, Warner Books, 1996.

[2] CASSANDRAS, C. G., LAFORTUNE, S. *Introduction to discrete event systems*. 2nd ed. New York, NY, Springer, 2008.

[3] CLAVIJO, L. B., BASILIO, J. C., CARVALHO, L. K. "DESLAB: A scientific computing program for analysis and synthesis of discrete-event systems". In: *Discrete Event Systems, 11th International Workshop on*, v. 1, pp. 349–355, Guadalajara, Mexico, 2012.

[4] SOMMERVILLE, I., MELNIKOFF, S. S. S., ARAKAKI, R., et al. *Engenharia de software*, v. 6. 9th ed. Boston, MA, Addison Wesley, 2003.

[5] CAYNE, B. S. *The Encyclopedia Americana*. New York, NY, Grolier, Inc., 1981.

[6] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., et al. *Introduction to algorithms*, v. 2. 3nd ed. London, MIT press Cambridge, 2001.

[7] RICKER, L., LAFORTUNE, S., GENC, S. "Desuma: A tool integrating giddes and umdes". In: *Discrete Event Systems, 2006 8th International Workshop on*, pp. 392–393, Ann Arbor, MI, 2006.

[8] LAURENT, A. M. S. *Understanding open source and free software licensing*. 1st ed. Sebastopol, CA, "O'Reilly Media, Inc.", 2004.

[9] WONHAM, W. M. "W. M. Wonham Home Page". Website, July 2014. `http://www.control.utoronto.ca/DES/`. Visited on 07/18/2014.

[10] AKESSON, K., FABIAN, M., FLORDAL, H., et al. "Supremica-an integrated environment for verification, synthesis and simulation of discrete event systems". In: *Discrete Event Systems, 2006 8th International Workshop on*, pp. 384–385, Ann Arbor, MI, 2006.

[11] MOOR, T., SCHMIDT, K., PERK, S. "libFAUDES—An open source C++ library for discrete event systems". In: *Discrete Event Systems, 2008.*

WODES 2008. 9th International Workshop on, pp. 125–130, Göteborg, Sweden, 2008.

[12] FOUNDATION, P. S. "Python Tutorial". Website, July 2014. `https://docs.python.org/2/tutorial/`. Visited on 07/10/2014.

[13] VAN DER WALT, S., COLBERT, S. C., VAROQUAUX, G. "The NumPy array: a structure for efficient numerical computation", *Computing in Science & Engineering*, v. 13, n. 2, pp. 22–30, 2011.

[14] HAGBERG, A., SWART, P., S CHULT, D. *Exploring network structure, dynamics, and function using NetworkX*. Relatório técnico, Los Alamos National Laboratory (LANL), 2008.