UNIVERSITY OF MICHIGAN
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
LECTURE NOTES FOR EECS 661
CHAPTER 2: UNTIMED MODELS OF DISCRETE EVENT SYSTEMS

Stéphane Lafortune

September 2004

2.1: LANGUAGES AND AUTOMATA

References for Chapter 2: Textbook, Chapter 2 (and the references therein).

2.1: Languages and Automata

Languages

E: finite set of *event* symbols (or "alphabet")

$E = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$

s: finite sequence of events from E, or word, or string, or *trace*

$s_1 = \sigma_2\sigma_3\sigma_1\sigma_1\sigma_5$

$|s|$: length of trace s (number of events, including repetitions); $|s_1| = 5$

$\sigma_i \in s$ denotes that σ_i appears in s

ϵ denotes the *empty* trace; $|\epsilon| = 0$

Concatenation of traces (in the obvious manner):

If $s_2 = \sigma_5\sigma_4$, then $s_1 s_2 = \sigma_2\sigma_3\sigma_1\sigma_1\sigma_5\sigma_5\sigma_4$.

ϵ is the identity element for concatenation: $s_1\epsilon = \epsilon s_1 = s_1$

σ^n denotes $\sigma\sigma\cdots\sigma$ (n times)

Notions of *prefix*, *suffix*, and *subtrace*:

$\sigma_2\sigma_3\sigma_1$ is a prefix of s_1

$\sigma_1\sigma_5$ is a suffix of s_1

$\sigma_3\sigma_1\sigma_1$ is a subtrace of s_1

prefixes and suffixes are also subtraces

Prefix-closure of a trace: it is the set that contains all the prefixes of the trace

$\overline{s_2} := \overline{\{s_2\}} = \{\epsilon, \sigma_5, s_2\}$

E^* is the *Kleene closure* of E.

It is the *set of all finite traces of elements of E, including ϵ.*

This set is countably infinite.

A *language* over E is a subset of E^*; i.e., any $L \subseteq E^*$ is a language.

Thus $\emptyset$, E, and E^* are languages.

Note: $\epsilon \notin \emptyset$. $\{\epsilon\}$ is a nonempty language containing only the empty trace.

Operations on languages:

- All the usual *set operations*: union, intersection, difference (denoted by "\"), complement (w.r.t. E^*)
- *Concatenation*: Let $L_1, L_2 \subseteq E^*$, then

$$L_1 L_2 := \{s \in E^* : (s = s_1 s_2) \wedge (s_1 \in L_1) \wedge (s_2 \in L_2)\} .$$

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

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

  Thus the prefix-closure $\overline{L}$ of L is the language consisting of all the prefixes of all the traces in L.
  Example: If $L = \{abc, cde\}$ then $\overline{L} = \{\epsilon, a, ab, abc, c, cd, cde\}$.
  If $L = \emptyset$ then $\overline{L} = \emptyset$, and if $L \neq \emptyset$ then $\epsilon \in \overline{L}$.
  In general, $L \subseteq \overline{L}$. L is said to be *prefix-closed* if $L = \overline{L}$.

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

$$L^* := \{\omega \in E^* : \omega = \omega_1\omega_2\cdots\omega_k, k \geq 0, \omega_i \in L\}.$$

  The * operation is *idempotent*: $(L^*)^* = L^*$. Also, $\emptyset^* = \{\epsilon\}$ and $\{\epsilon\}^* = \{\epsilon\}$.

- The *post-language* of L after trace s is:

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

  By definition, $L/s = \emptyset$ if $s \notin \overline{L}$.

More notation: $L^+ := LL^*$.

Two languages L_1 and L_2 are said to be *nonconflicting* if $\overline{L_1 \cap L_2} = \overline{L_1} \cap \overline{L_2}$.

L is $M-$*closed* if $\overline{L} \cap M = L$.

Finite Representation of Languages

- E: finite
- E^*: countably infinite
- 2^{E^*} (the power set of E^*, i.e., the set of all languages): uncountable
- We would like to represent *languages* "finitely".

  If a language is finite, we could always list all its elements; but this is rarely practical.

  If a language L is infinite, we could try to represent it as:

$$L = \{s \in E^* : s \text{ has property P}\}$$

where P could for instance specify that a trace should have the same number of σ_1 events as σ_2 events. This is often useful, but is not amenable to analysis when calculations involving finding subsets or supersets of L have to be performed (see Chapter 3).

- More preferably, we would like to use *discrete event modeling formalisms* that would require us to specify only a finite number of "objects" in order to represent a particular language.
  Finite-state automata and Petri nets are two such formalisms.
  Then we would like to know how much of 2^{E^*} can a particular formalism represent; it cannot represent all of it because this set is uncountable and we are only specifying a finite number of objects.
  Also of interest would be the properties of the class of languages represented by a given formalism (e.g., closed under union).

- Computer scientists have developed a hierarchy of (finite) representations of languages (cf. Chomsky) in a field called Formal Language Theory.
  We are primarily interested in the simplest class of languages in this hierarchy, termed the class of *Regular Languages* and denoted $\mathcal{R}$.
  Note that $\mathcal{R} \neq 2^{E^*}$.
  We will use the notion of *Deterministic Finite-State Automata* to define $\mathcal{R}$.

Automata

A *Deterministic Automaton*, or simply *automaton*, is a six-tuple

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

where

X is the set of *states*

E is the finite set of *events* associated with the transitions in G

$f : X \times E \rightarrow X$ is the *transition function*: $f(x, e) = y$ means that there is a transition labeled by event e from state x to state y; in general, f is a *partial* function on its domain

$\Gamma : X \rightarrow 2^E$ 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*.

Remarks:

- If X is a finite set, we call G a *deterministic finite-state automaton*, often abbreviated as DFA.
- The automaton is said to be *deterministic* because f is a function over $X \times E$.
- The fact that we allow the transition function f to be partially defined over its domain $X \times E$ is a variation over the "standard" definition of automaton in the computer science literature that is quite important in DES theory.
- Formally speaking, the inclusion of Γ in the definition of G is superfluous in the sense that Γ is derived from f.
- Proper selection of which states to mark is a modeling issue that depends on the problem of interest.

The automaton G operates as follows. It starts in the initial state x_0 and upon the occurrence of an event $e \in \Gamma(x_0) \subseteq E$ it will make a transition to state $f(x_0, e) \in X$. This process then continues based on the transitions for which f is defined.
For the sake of convenience, f is always extended from domain $X \times E$ to domain $X \times E^*$ in the following recursive manner:

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

Now think of the automaton as a *directed graph* and consider all the (directed) paths that can be followed from its initial state; consider among these all the paths that end in a marked state.
This leads us to the notion of the languages *generated* and *marked* by the automaton.

- The language *generated* by G is

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

- The language *marked* by G is

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

- $\mathcal{L}(G)$ is always prefix-closed.
- $\mathcal{L}(G) = E^*$ when f is a *total function*.
- Automata G_1 and G_2 are said to be *equivalent* if

$$\mathcal{L}(G_1) = \mathcal{L}(G_2) \qquad \text{and} \qquad \mathcal{L}_m(G_1) = \mathcal{L}_m(G_2) \ .$$

Accessibility and Coaccessibility of Automata

G represents two languages: $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$. This is central to the modeling of discrete event systems.
In general: $\mathcal{L}_m(G) \subseteq \overline{\mathcal{L}_m(G)} \subseteq \mathcal{L}(G)$.

About X:

- Since we use an automaton to model two languages, we can delete all the states that are not *accessible* or *reachable* from x_0 by some trace in $\mathcal{L}(G)$. Note that when we "delete" a state, this means also deleting all the transitions that are *attached* to that state.
- Formally,

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

- Clearly, the Ac operation has no effect on $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$. Thus from now on we will always assume, without loss of generality, that an automaton is *accessible*, i.e., $G = Ac(G)$.

About X_m:

- A state is *coaccessible* if it can reach a marked state.
- Taking the coaccessible part of an automaton means building

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

- The $CoAc$ operation clearly affects (i.e., shrinks) $\mathcal{L}(G)$ but it does not affect $\mathcal{L}_m(G)$. If G is coaccessible (i.e., $G = CoAc(G)$), then $\mathcal{L}(G) = \overline{\mathcal{L}_m(G)}$.
- An automaton that is both accessible and coaccessible is said to be *trim*. $Trim(G) := CoAc[Ac(G)] = Ac[CoAc(G)]$.
- Coaccessibility is very useful to model *deadlock*, or more generally, what we will call *blocking*:

  An automaton is said to be *blocking* if

$$\mathcal{L}(G) \neq \overline{\mathcal{L}_m(G)}$$

  which necessarily means that $\overline{\mathcal{L}_m(G)}$ is a proper subset of $\mathcal{L}(G)$.

About E:

- Formally, we can include in E events that do not appear in $\mathcal{L}(G)$, since E is a parameter in the definition of an automaton. This can however lead to some confusion, as in such a case, the automaton is not entirely represented by its transition function f, something that we find convenient. Thus, from now on, unless explicitly stated otherwise, we will assume that E in the definition of automaton G consists only of those events that appear in the traces in $\mathcal{L}(G)$.

UMDES-LIB:

- refer to the commands: `create_fsm, acc, co_acc, write_ev, write_st, equiv`.

Complement Operation

Given: $G = (X, E, f, \Gamma, x_0, X_m)$ that marks the language $K \subseteq E^*$.
Desired: G^{comp} that marks the language $E^* \setminus K$.
G^{comp} is built in two steps as follows.

1. Complete the transition function f of G and make it a total function, f_{tot}.

   1.1. $X \cup \{x_d\}$ ["dead" or "dump" state]

   1.2.
   $$f_{tot}(x,e) = \begin{cases} f(x,e) & \text{if } e \in \Gamma(x) \\ x_d & \text{otherwise.} \end{cases}$$
   Moreover, set $f_{tot}(x_d, e) = x_d$ for all $e \in E$.

   1.3. $G_{tot} = (X \cup \{x_d\}, E, f_{tot}, x_0, X_m)$
   and $\mathcal{L}(G_{tot}) = E^*$ and $\mathcal{L}_m(G_{tot}) = K$.

2. $G^{comp} = (X \cup \{x_d\}, E, f_{tot}, x_0, (X \cup \{x_d\}) \setminus X_m)$.
   Clearly, $\mathcal{L}(G^{comp}) = E^*$ and $\mathcal{L}_m(G^{comp}) = E^* \setminus \mathcal{L}_m(G)$, as desired.

Nondeterministic Automata

- We extend the definition of automata to allow for two new elements:
  1. The event set is augmented to
  $$E_\varepsilon = E \cup \{\varepsilon\} \quad .$$
  A transition labeled ε is to be interpreted as some internal event of the automaton that is not observed by the outside world.
  2. $f(x, \sigma)$ is no longer required to be a single state but can now be a *set of states*.

  The resulting object is called a *Nondeterministic Automaton*. Formally, a *Nondeterministic Automaton*, denoted by G_{nd}, is a six-tuple
  $$G_{nd} = (X, E_\varepsilon, f_{nd}, \Gamma, x_0, X_m)$$
  where these objects have the same interpretation as in the definition of deterministic automaton, with the two differences that:
  1. f_{nd} is a function $f_{nd} : X \times E_\varepsilon \to 2^X$, that is, $f_{nd}(x, e) \subseteq X$ whenever it is defined.
  2. The *initial* state may itself be a set of states, that is $x_0 \subseteq X$.

- Nondeterministic automata generate and mark languages similarly to automata.

  To describe these languages formally, we start by extending the domain of f_{nd} to traces of events. Let u be a trace of events and e an event; then

$$f_{nd}(x, ue) := \{z : z \in f_{nd}(y, e) \text{ for some state } y \in f_{nd}(x, u)\} \ .$$

  Note that by convention, $x \in f_{nd}(x, \varepsilon)$.

  We define:

$$\begin{aligned} \mathcal{L}(G_{nd}) &= \{s \in E^* : \exists x \in x_0 \ (f_{nd}(x, s) \text{ is defined })\} \\ \mathcal{L}_m(G_{nd}) &= \{s \in \mathcal{L}(G_{nd}) : \exists x \in x_0 \ (f_{nd}(x, s) \cap X_m \neq \emptyset)\} \ . \end{aligned}$$

- *Question?:* Do nondeterministic automata have more expressive power than automata?

  Answer: No! Any nondeterministic automaton can be transformed into an equivalent automaton, i.e., an automaton that generates and marks the same languages.

  Proof: Deferred to section on observer automata.

2.2: COMPOSITION OF AUTOMATA

2.2: Composition of Automata

Product

Symbol for Product: $\times$

Input: $G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1})$ and $G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$.

Output: $G_1 \times G_2 := Ac(X_1 \times X_2, E_1 \cap E_2, f, \Gamma_{1\times 2}, (x_{01}, x_{02}), X_{m1} \times X_{m2})$
where

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

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

Properties:

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

Comments:

- Property (2) shows how we can "implement" the intersection of languages using automata.
- **UMDES-LIB:** `product`.

Parallel Composition

Symbol for Parallel Composition: $||$

Input: $G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1})$ and $G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$.

Output: $G_1 \mid\mid G_2 := Ac(X_1 \times X_2, E_1 \cup E_2, f, \Gamma_{1||2}, (x_{01}, x_{02}), X_{m1} \times X_{m2})$
where

$$f((x_1, x_2), \sigma) := \begin{cases} (f_1(x_1, \sigma), f_2(x_2, \sigma)) & \text{if } \sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, \sigma), x_2) & \text{if } \sigma \in \Gamma_1(x_1) \setminus E_2 \\ (x_1, f_2(x_2, \sigma)) & \text{if } \sigma \in \Gamma_2(x_2) \setminus E_1 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In a parallel composition, a common event, i.e., an event in $E_1 \cap E_2$, can only be executed if the two automata both execute it simultaneously. Thus the two automata are "synchronized" on the common events. (For this reason, this operation is also called *synchronous composition*.) The other events, i.e., those in $(E_2 \setminus E_1) \cup (E_1 \setminus E_2)$, are not subject to such a constraint and can be executed whenever possible.

Properties of $||$:

Let us define the *natural projections* $P_i : (E_1 \cup E_2)^* \to E_i^*$ for $i = 1, 2$ as follows:

$$\begin{aligned} P_i(\epsilon) &= \epsilon \\ P_i(\sigma) &= \begin{cases} \sigma & \text{if } \sigma \in E_i \\ \epsilon & \text{if } \sigma \notin E_i \end{cases} \\ P_i(s\sigma) &= P_i(s)P_i(\sigma) \;\text{ for }\; s \in (E_1 \cup E_2)^*,\ \sigma \in (E_1 \cup E_2) \end{aligned}$$

and the corresponding inverse maps $P_i^{-1} : E_i^* \to 2^{(E_1 \cup E_2)^*}$ as follows:

$$P_i^{-1}(t) = \{s \in (E_1 \cup E_2)^* : P_i(s) = t\} \quad .$$

The projections P_i and their inverses P_i^{-1} are extended to languages in the usual manner: for $L \subseteq (E_1 \cup E_2)^*$,

$$P_i(L) := \{t \in E_i^* : \exists s \in L(P_i(s) = t)\}$$

and for $L_i \subseteq E_i^*$,

$$P_i^{-1}(L_i) := \{s \in (E_1 \cup E_2)^* : \exists t \in L_i(P_i(s) = t)\} \quad .$$

Note that $P_i[P_i^{-1}(L)] = L$ but $L \subseteq P_i^{-1}[P_i(L)]$. (These properties are true for any natural projection.)

We have the following properties for parallel composition:

1. $P_i[\mathcal{L}(G_1||G_2)] \subseteq \mathcal{L}(G_i)$, for $i = 1,\ 2$.
2. $\mathcal{L}(G_1||G_2) = P_1^{-1}[\mathcal{L}(G_1)] \cap P_2^{-1}[\mathcal{L}(G_2)]$
3. $\mathcal{L}_m(G_1||G_2) = P_1^{-1}[\mathcal{L}_m(G_1)] \cap P_2^{-1}[\mathcal{L}_m(G_2)]$
4. $G_1||G_2 = G_2||G_1$, up to a renaming of the states
5. $G_1||(G_2||G_3) = (G_1||G_2)||G_3$

Comments:

- We can also define a $||$ operation on languages. In view of the above, the proper definition is:
  for $L_i \subseteq E_i^*$ and P_i defined as above,
  $$L_1||L_2 = P_1^{-1}(L_1) \cap P_2^{-1}(L_2) \quad .$$
- If $E_1 = E_2$, then the parallel composition reduces to the product, since all transitions are forced to be synchronized.
- If $E_1 \cap E_2 = \emptyset$, then there are no synchronized transitions and thus G is the *concurrent* behavior of G_1 and G_2. This is often termed the *shuffle* of G_1 and G_2.
- **UMDES-LIB:** `par_comp`.

2.3: OBSERVER AUTOMATA

2.3: Observer Automata

- Consider a DES modeled by (possibly nondeterministic) automaton $G_{nd} = (X, E \cup \{\varepsilon\}, f_{nd}, \Gamma, x_0, X_m)$.
- Partition the set of events E of G as

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

  where

  - E_o is the set of *observable* events (i.e., recorded by the sensors);
  - E_{uo} is the set of *unobservable* events (i.e., not recorded by the sensors). Note that ε transitions are also unobservable, by definition of ε.

- Objective: estimate the state of G_{nd} from traces of *observed* events only.

  Tool: *Observers* [G_{obs}].

  UMDES-LIB: refer to the command `obsvr`.

Procedure for Building Observer G_{obs} for G_{nd}

Let $G_{nd} = (X, E \cup \{\epsilon\}, f_{nd}, x_0, X_m)$ be a nondeterministic automaton and let $E = E_o \cup E_{uo}$. Then $G_{obs} = (X_{obs}, E_o, f_{obs}, x_{0,obs}, X_{m,obs})$ and it is built as follows.

Step 0: Replace all the transitions of G_{nd} labeled by events in E_{uo} by ε-transitions. Let the modified automaton still be denoted by G_{nd}.

Step 1: Start with $X_{obs} = 2^X \setminus \emptyset$.

Step 2: For each state $x \in X$ define

$$UR(x) := f_{nd}(x, \varepsilon) \quad .$$

Read UR as "unobservable reach" since ε transitions are not "observed". It is assumed here that we are working with the extension of function f_{nd} to strings in $(E \cup \{\varepsilon\})^*$, as described earlier.

For a set B, define

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

Step 3: Define $x_{0,obs} = UR(x_0)$.

Step 4: For each $S \subseteq X$ and $e \in E$, define

$$f_{obs}(S, e) \;=\; UR(\{x \in X : \exists x_e \in S\ [x \in f_{nd}(x_e, e)]\})$$

Step 5: $X_{m,obs} = \{S \subseteq X : S \cap X_m \neq \emptyset\}$.

Step 6: In practice, the above is performed in a breadth-first manner so that only the accessible part of G_{obs} is constructed. The resulting state space X_{obs} is a subset of 2^X. Note that the empty subset of X need not be considered, since it is never an accessible state of X_{obs}.

The important properties of G_{obs} are that:

1. G_{obs} is a *deterministic* automaton with event set E_o.
2. $\mathcal{L}(G_{obs}) = P_o[\mathcal{L}(G_{nd})]$
   where P_o is the natural projection $P_o : E \rightarrow E_o$.
3. $\mathcal{L}_m(G_{obs}) = P_o[\mathcal{L}_m(G_{nd})]$.
4. 2. and 3. show that nondeterministic automata have the same modeling power as deterministic automata.
5. Let $f_{obs}(x_{0,obs}, t) = S$ where $t \in P_o[\mathcal{L}(G_{nd})]$.
   Then $x \in S$ iff there exists $s \in \mathcal{L}(G_{nd})$ such that $x \in f_{nd}(y, s)$ for some $y \in x_0$ and $P_o(s) = t$.

   Hence, S is the set of all states G_{nd} could be in after observing t, namely, S is the *state estimate* of G_{nd} after t.

Except for the inclusion of unobservable events, the above construction is the standard conversion of a nondeterministic automaton to a deterministic one that you can find in books on automata theory.

2.4: REGULAR LANGUAGES AND FINITE-STATE AUTOMATA

2.4: Regular Languages and Finite-State Automata

The Class of Regular Languages

- *Definition:* A language K is said to be *regular*, i.e., $K \in \mathcal{R}$, if there exists a (deterministic) *finite-state* automaton G that marks it, i.e, $\mathcal{L}_m(G) = K$.
- Not all languages are regular:

$$\{a^n b^n : n = 0, 1, 2, \ldots\} \notin \mathcal{R}.$$

  Intuition: We need to memorize the number of a's to do the right number of b's; but the number of a's can be arbitrarily large, so any finite number of states will not suffice.

  This can be formally proved using the Pumping Lemma:

  Pumping Lemma (1961): Let L be an infinite regular language. Then there exist subtraces x, y, and z such that (i) $y \neq \epsilon$ and (ii) $xy^n z \in L$ for all $n \geq 0$.

  Intuition: Since L has infinite cardinality, then there must be a cycle in any finite-state automaton that marks it.

- $\mathcal{R}$ can also be defined using the notion of *regular expressions*, which are a means of representing languages using events (including ε) and the following three operations: *concatenation*, *or* (denoted +), and *Kleene-closure* (*).

Properties of the Class of Regular Languages

Theorem: The class $\mathcal{R}$ is closed under:

1. Union
2. Concatenation
3. Kleene-closure
4. Complementation (w.r.t. E^*)
5. Intersection

Proof: Sketch.

1. Create a new initial state and connect it, with two ϵ transitions, to the two initial states of the respective automata.
2. Connect the marked states of G_1 to the initial state of G_2 by ϵ transitions. Unmark all the states of G_1.
3. Add a new initial state, mark it, connect it to the old initial state by an ϵ transition. Then add ϵ transitions from every marked state to the old initial state.
4. Use the complement operation.
5. Take the product of the two automata.

State Space Minimization

- For $K \in \mathcal{R}$, define $||K||$ to be the minimum of $|X_A|$ among all finite-state automata A, with complete transition function, that mark K. The automaton that achieves this minimum is called the *canonical recognizer* of K.

  Examples:

  $||\emptyset|| = ||E^*|| = 1$.

  If $E = \{a, b\}$ and $L = \{a\}^*$, then $||L|| = 2$.

- $|| \cdot ||$ has nothing to do with $\subseteq$ for languages.

  Also, $\subseteq$ does not imply a "subgraph" relationship among the canonical recognizers.

  This "subgraph" idea is very useful so we formalize it:

- *Subautomaton Relation:* Consider two automata with same event set E: $G_1 = (X_1, E, f_1, x_{o1})$ and $G_2 = (X_2, E, f_2, x_{o2})$. (Here we ignore marking.) We say that G_1 is a subautomaton of G_2, denoted

$$G_1 \sqsubseteq G_2$$

  if

$$f_1(x_{01}, s) = f_2(x_{02}, s) \text{ for all } s \in \mathcal{L}(G_1) \quad .$$

  Note that this condition implies that $X_1 \subseteq X_2$, $x_{01} = x_{02}$, and $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$.

Algorithm for Identifying Equivalent States

Step 1: Flag (x, y) for all $x \in X_m$, $y \notin X_m$.

Step 2: For every pair (x, y) not flagged in Step 1:

Step 2.1: If $(f(x, e), f(y, e))$ is flagged for some $e \in E$, then:

Step 2.1.1: Flag (x, y).

Step 2.1.2: Flag all unflagged pairs (w, z) in the list of (x, y). Then, repeat this step for each (w, z) until no more flagging is possible.

Step 2.2: Otherwise, that is, no $(f(x, e), f(y, e))$ is flagged, then for every $e \in E$:

Step 2.2.1: If $f(x, e) \neq f(y, e)$, then add (x, y) to the list of $(f(x, e), f(y, e))$.