Proceedings of the World Congress on Engineering and Computer Science 2010 Vol I
WCECS 2010, October 20-22, 2010, San Francisco, USA

Product Construction of Finite-State Machines

Samuel C. Hsieh

Abstract— **Two algorithms to construct a product machine from two finite-state machines are presented and analyzed. The first algorithm is simple and correctly produces a product machine, but the product machine may include unreachable states and associated transitions. The second algorithm produces a functionally correct product machine that has no unreachable states.**

Index Terms—**deterministic finite-state machine, product machine construction, theory, unreachable state.**

I. Introduction

The union or the intersection of two regular languages is still a regular language, that is, regular languages are closed under the operations union and intersection, e.g., [1]. In fact, from two given deterministic finite-state machines, a product machine can be constructed such that the product machine simulates the two given machines simultaneously and accepts a language that is the intersection or the union of the languages of the given machines.

This article discusses and analyzes two algorithms to construct a product machine from two given machines. The first algorithm is simple and correctly produces a product machine. However, the product machine may include unreachable states and associated transitions. The second algorithm addresses this problem and produces a functionally correct product machine that has no unreachable states.

II. Basic Definitions

A deterministic finite-state machine (DFSM) consists of a set of states Q, an alphabet Σ, a transition function δ from $Q\times\Sigma$ to Q, a start state s in Q, a set of accept states F, which is a subset of Q. Formally, a DFSM is the 5-tupe $(Q, \Sigma, \delta, s, F)$.

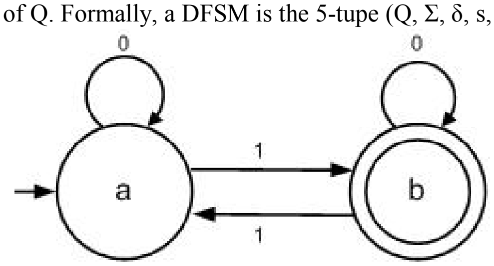

Fig.1 $M_1=(\{a, b\}, \{0, 1\}, \delta_1, a, \{b\})$

Graphically, a finite-state machine is usually shown as a state diagram. As an example, Fig.1 shows a DFSM $M_1=(\{a, b\}, \{0, 1\}, \delta_1, a, \{b\})$, where the set of states is {a, b}, the alphabet is {0, 1}, the start state is the state a, the state b is the only accept state, and the transition function δ_1 is shown by the collection of labeled arrows between the states: for example, since there is an arrow labeled 1 from state a to state b, $\delta_1(a, 1) = b$. The start state is marked by an incoming arrow that does not have a source state, and the accept state is indicated by a double circle. The machine M_1 accepts those strings that have an odd number of 1's and any number of 0's.

Table I. State Transition Table for M_1

	0	1
a	a	b
b	b	a

Often, a transition function is defined with a state transition table. Table I shows the state transition table for M_1. The table defines the transition function δ_1 in a tabular form. For example, the fact $\delta_1(a, 1) = b$ is shown by the value b in the entry [a, 1] of the table (that is, in the row labeled a and in the column labeled 1).

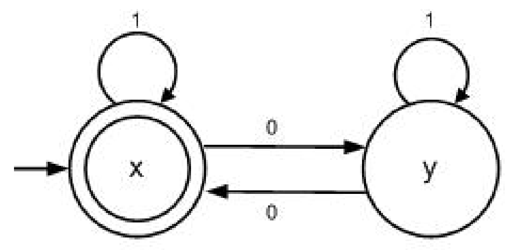

Fig.2 $M_2=(\{x, y\}, \{0, 1\}, \delta_2, x, \{x\})$

Table II. State Transition Table for M_2

	0	1
x	y	x
y	x	y

Fig. 2 shows another machine $M_2=(\{x, y\}, \{0, 1\}, \delta_2, x, \{x\})$. The machine accepts those strings that have an even number of 0's and any number of 1's. The transition function δ_2 is shown as a transition table in Table II.

III. Product Machine Construction

A product machine is constructed from two machines, simulates the behavior of the two machines simultaneously, and accepts either the intersection or the union of the languages of the two machines. A product machine that accepts the *intersection* of the languages of the two machines

Manuscript received July 26, 2010.
S. C. Hsieh is with Computer Science Department, Ball State University, Muncie, IN 47306 (e-mail: shsieh@bsu.edu).

$M_a=\{Q_a, \Sigma, \delta_a, s_a, F_a\}$ and $M_b=\{Q_b, \Sigma, \delta_b, s_b, F_b\}$ is a DFSM $M_p=\{ Q_a \times Q_b, \Sigma, \delta_p, (s_a, s_b), F_a \times F_b\}$.

The set of states of the product machine can be easily found. It is the Cartesian product $Q_a \times Q_b$. Obviously, each state of M_p is a pair of states, denoted by (a, b). In the product machine, the pair is unordered. i.e., (a, b) and (b, a) are considered the same state of the product machine.

The start state of M_p is the pair that consists of the start states of M_a and M_b. The accept states of M_p can also be identified easily: for a state (a, b) of the product machine M_p to be an accept state, one of a and b must be an accept state of M_a and the other must be an accept state of M_b, i.e., both a and b must be accept states in M_a and M_b.

Suppose T_a and T_b are the state transition tables for M_a and M_b. The state transition table T_p for the product machine M_p can be constructed by the following simple algorithm:

```
for each state a in Q_a
   for each state b in Q_b
      for each symbol r in Σ
         T_p[(a, b), r]=(T_a[a, r], T_b[b, r]);
```

Let N_a be the number of states in Q_a, N_b be the number of states in Q_b and N_Σ be the number of symbols in Σ. The running time of this commonly known algorithm is obviously $O(N_a N_b N_\Sigma)$.

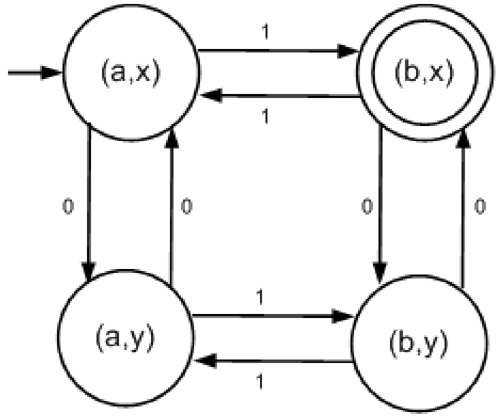

Fig.3 M_3= Product DFSM of M_1 and M_2

Table III. State Transition Table for M_3

	0	1
(a,x)	(a,y)	(b,x)
(a,y)	(a,x)	(b,y)
(b,x)	(b,y)	(a,x)
(b,y)	(b,x)	(a,y)

The machine M_3 shown in Fig. 3 is the product machine constructed from the two machines M_1 and M_2 given in the previous section. Table III shows the state transition table of M_3 constructed using the algorithm given above. Apparently, M_3 accepts those strings having an odd number of 1's **and** an even number of 0's.

A product machine that accepts the *union* of the languages of two machines M_1 and M_2 can be constructed in the same way except that, in order for a state (a, b) of the product machine to be an accept state, at least one of the states a and b must be an accept state of M_a or M_b. As an example, if the three states of M_3 (a,x), (b,x) and (b,y) are accept states, then M_3 will accept the union of the languages of M_1 and M_2. That is, M_3 will accept those strings having an odd number of 1's **or** an even number of 0's.

IV. AVOIDING UNREACHABLE STATES

The algorithm discussed in the previous section may produce a product machine that has states that cannot be reached from its start state. Removing the unreachable states of a machine does not alter the function of the machine.

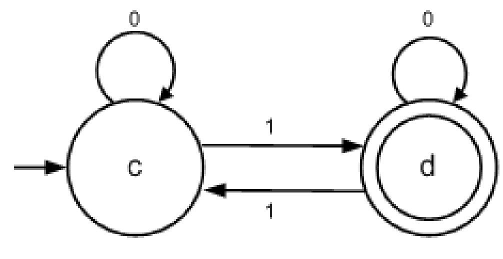

Fig.4 M_4=({c, d}, {0, 1}, δ_4, c, {d})

Table IV. State Transition Table for M_4

	0	1
c	c	d
d	d	c

As an example, consider the machine M_4 shown in Fig. 4 and its state transition table in Table IV. To make the example simple, the machine M_4 is obtained by renaming the states of M_1 and accepts the same language as M_1.

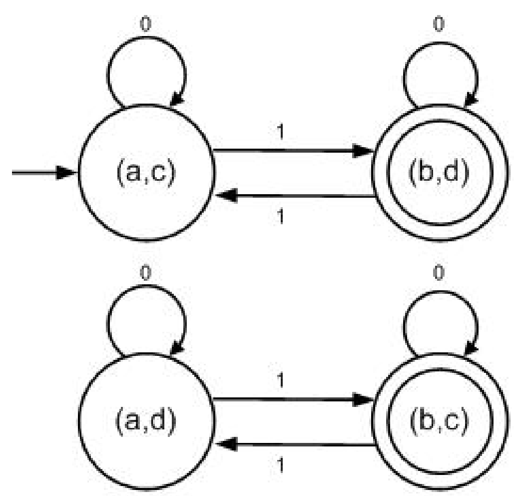

Fig.5 A Product Machine with Unreachable States

The product machine of M_1 and M_4 constructed using the algorithm discussed in the previous section to accept the intersection of the languages of M_1 and M_4 is shown in Fig. 5. Indeed, the product machine functions correctly: it accepts the strings that both M_1 and M_4 accept, but it includes the

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

unreachable states (a, d) and (b, c), which can be eliminated without changing the function of the machine.

An algorithm that constructs a product machine that does not include unreachable states will now be presented. Suppose T_a and T_b are the state transition tables for two machines M_a and M_b whose product machine is to be constructed. The algorithm will produce a product machine M_p. M_p initially has no transitions and only one state: the start state, which is the pair that consists of the start states of M_a and M_b. As the algorithm proceeds, more states and transitions will be added to M_p. When the algorithm terminates, M_p will be the product machine of M_a and M_b with no unreachable states. The algorithm also makes use of a set R, which initially contains the start state of M_p. The purpose of using the set R is to keep track of those states in M_p whose outgoing transitions have yet to be added to M_p. The algorithm is given below.

```
repeat the following
{
 remove a state (a,b) from R;
 for each symbol r in Σ do the following
   {
    if (Ta[a,r], Tb[b,r]) is not in Mp
         add (Ta[a, r],Tb[b, r]) to Mp and to R; //end if
    add to Mp a transition on r from (a,b) to (Ta[a,r],Tb[b,r]);
   } //end for-each
}
until  R is empty;  //end repeat-until
```

The *if* statement in the algorithm guarantees that each state is added to M_p and R only once. It is worth noting that a new state is added to M_p if and only if there is a transition from a state already in M_p to the new state. Since initially the start state is the only state in M_p, all of the states of M_p are reachable from the start state.

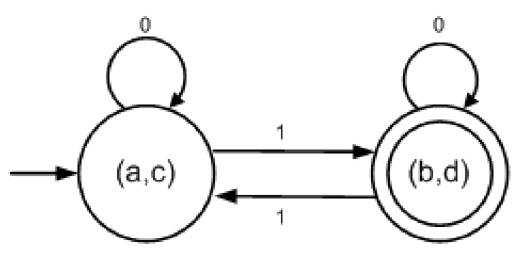

Fig.6 A Product Machine with No Unreachable States

As an example of using the algorithm, suppose the product machine of M_1 and M_4 is to be constructed by the above algorithm. Let T_1 and T_4 be the transition tables of M_1 and M_4. Initially R and M_p contain only the state (a,c). As the algorithm begins, (a,c) is removed from R. For the symbol 0, since the state (T_1[a,0], T_2[c,0]) = (a,c) is already in M_p, (a,c) is not added to M_p or R. The transition from (a,c) to (a,c) on the symbol 0 is then added to M_p. For the symbol 1, since the state (T_1[a,1], T_2[c,1]) = (b,d) is not in M_p, (b,d) is added to M_p and R. The transition from (a,c) to (b,d) on the symbol 1 is then added to M_p. Since R is not empty, another iteration of the *repeat-until* loop begins. The state (b,d) is removed from R. For the symbol 0, since the state (T_1[b,0], T_2[d,0]) = (b,d) is already in M_p, (b,d) is not added to M_p or R. The transition from (b,d) to (b,d) on the symbol 0 is then added to M_p. For the symbol 1, since the state (T_1[b,1], T_2[d,1])=(a,c) is already in M_p, (a,c) is not added to M_p or R. The transition from (b,d) to (a,c) on the symbol 1 is then added to M_p. The *repeat-until* loop then terminates because R is now empty. The resultant product machine M_p is shown in Fig. 6. Obviously, the product machine has no unreachable state and, as expected, it accepts the same language as M_1 and M_4 do.

The *if* statement requires searching the set of states in M_p. Let E be the number of transitions in the product machine (i.e., the number of arrows in its state diagram) and N_Σ be the number of symbols in the alphabet. The number of states is E/N_Σ because every state has N_Σ outgoing transitions. With an efficient implementation, searching a set of E/N_Σ elements takes $O(\log(E/N_\Sigma))$ time. An analysis shows that searching is performed for every transition added, and the algorithm takes $O(E \log(E/N_\Sigma))$ time.

V. SUMMARY

Two algorithms to construct a product machine from two finite-state machines are presented and analyzed. The first algorithm is simple and correctly produces a product machine but the product machine may include unreachable states and associated transitions. The second algorithm produces a functionally correct product machine that has no unreachable states.

REFERENCES

[1] M. Sipser, *Introduction to the Theory of Computation*. Boston: Thomson Course Technology, 2006, ch.1.