Applied Intelligence 16, 205–221, 2002

Applying Learning by Examples for Digital Design Automation

BEN CHOI
Computer Science, College of Engineering and Science, Louisiana Tech University, Ruston, LA 71272, USA
pro@Benchoi.org

Abstract. This paper describes a new learning by example mechanism and its application for digital circuit design automation. This mechanism uses finite state machines to represent the inferred models or designs. The resultant models are easy to be implemented in hardware using current VLSI technologies. Our simulation results show that it is often possible to infer a well-defined deterministic model or design from just one sequence of examples. In addition this mechanism is able to handle sequential task involving long-term dependence. This new learning by example mechanism is used as a design by example system for automatic synthesis of digital circuits. Such systems have not previously been successfully developed mainly because of the lack of mechanism to implement them. From artificial neural network research, it seems possible to apply the knowledge gained from learning by example to form a design by example system. However, one of the problems with neural network approaches is that the resultant models are very difficult to be implemented in hardware using current VLSI technologies. By using the mechanism described in this paper, the resultant models are finite state machines that are well suited for digital designs. Several sequential circuit design examples are simulated and tested. Although our test results show that such a system is feasible for designing simple circuits or small-scale circuit modules, the feasibility of such a system for large-scale circuit design remains to be showed. Both the learning mechanism and the design method show potential and the future research directions are provided.

Keywords: learning by example, learning mechanism, finite state machine, design by example, and digital design

1. Introduction

Learning by examples and design by examples have a lot in common. For learning by examples, a system must be able to (1) generate a model to describe a given set of examples, (2) modify the existing model to capture the information provided by additional examples, and (3) handle noise or inconsistent information. In design by example and in particularly for digital circuit design automation, a system must address the following three problems:

(1) The designer provides input/output examples of a required digital circuit to the system. The system must be able to generate the circuit design.
(2) The designer can provide additional examples to refine the design. The system must be able to take the additional examples and generate a refined design.
(3) The designer may provide two or more sets of examples that are not compatible with each other. The system must be able to inform the designer that there are incompatible sets of examples.

Such design by example systems has not previously been successfully developed mainly because of the lack of a mechanism to implement them. With the knowledge gained from artificial neural network research, it seems possible to apply the knowledge, such as learning by example [1–3], to the design by example system. Learning by example and design by example have similar end results. In learning by example, the end result is a model describing the given examples. In design by example, the end result is a design based on the given examples. However, one of the problems with the artificial neural network approaches is that the resultant model is very difficult to implement in hardware using current VLSI technologies [4–8]. The difficulty is the result

of analog models used in most neural networks. Most neural network models require continuous and signed values for representing weights and internal states (also called activation values). They also require operations such as multiplication, summation, and comparison of the continuous values [9–14]. To efficiently implement the continuous values and operations, analog circuits are required.

Researchers face a number of technical challenges in attempting to use analog VLSI [15–21] and optical computers for implementing neural networks. Researchers in analog VLSI have to overcome problems of variation in chip lithography, dopant density, instability, and imprecision of component parameters [22]. Researchers in optical computing have to overcome problems of limiting dynamic range, and problems of interfacing bottleneck between electronic and optical components. Other attempts to implement neural networks in hardware include simulating the analog networks in digital computers. Some commercial products advertised as Neurocomputers are in fact simulators running in digital computers [10]. It shall be emphasized that these simulators do not exploit in hardware the parallel and distributed computation power of neural network models [22].

This paper describes a mechanism for forming a model that is quite different from the neural network approach. This new mechanism is conceived partially based on automata and switching theories. By using this mechanism, the resultant models are finite state machines. Unlike the neural network approach, the resultant model is very easy (can be directly translated) to be implemented in hardware using current VLSI technologies. In addition, the system developed in this investigation is trained by using sequences of input/output examples. Each step in a sequence consists of an input and an output. The steps in a sequence are ordered in time. From the sequences of input/output examples, the system is able to learn to form a sequential circuit or to capture information of sequential events. Thus, this new approach is well suited as mechanism for digital circuit design automation.

Besides easy to be implemented in hardware, the learning by example system described in this paper has other unique features comparing to other related systems. For instance, training the system developed in this investigation requires a small number of training sequences or examples. The system is often (see example in Section 7) capable of inferring a reasonable finite state machine from just one sequence of input/output examples. In addition, the system is able to handle sequential tasks that require many states. On the contrary, training recurrent neural networks [23, 24] (that are the type of networks needed to model sequence of events) requires a very large number of training sequences. For example, it requires 2400 to 409,600 sequences of examples (epochs) to train a recurrent neural network to determine whether an input sequence contains an even or odd number of 1's. The number of sequences required depends on the length of the input sequence and the training algorithm used. For an input sequence having 10 inputs, for instance, all five training algorithms used in the study by [25, 26] requires more than 6400 training sequences.

1.1. A Simple Design Example

Consider the design of a sequential circuit to recognize the input/output sequence provided in Fig. 1. The sequence shows that at step 0, given input 0, the circuit is required to produce output 0. At step 1, given input 1, the circuit is required to produce output 1. At step 2, given input 0, the circuit is required to produce output 1, and so on. Or, consider the design of a circuit to determine the parity of a communication bit stream—to determine whether there is even or odd number of ones in the bit stream. Given such a requirement, a designer, using the traditional design method [27], needs to spend time drawing a transition diagram, setting up a state table, minimizing the states, and then translating the minimal state table to circuits. Using recent design methods, such as hardware descriptive language (VHDL) [28], the designer still needs to write a program to define the circuit. Either going through the traditional design method or using a hardware descriptive language method requires intuitive and labor-intensive processes.

By using the proposed design by example system, to design a circuit to determine the parity of a communication bit stream outlined above, a designer simply provides, to our design automation system, input/output examples such as the ones shown in Fig. 1.

Step:	0	1	2	3	4	5	6	7	8	9	10
Input:	0	1	0	0	1	0	1	0	1	1	0
Output:	0	1	1	1	0	0	1	1	0	1	1

Figure 1. Input/output examples.

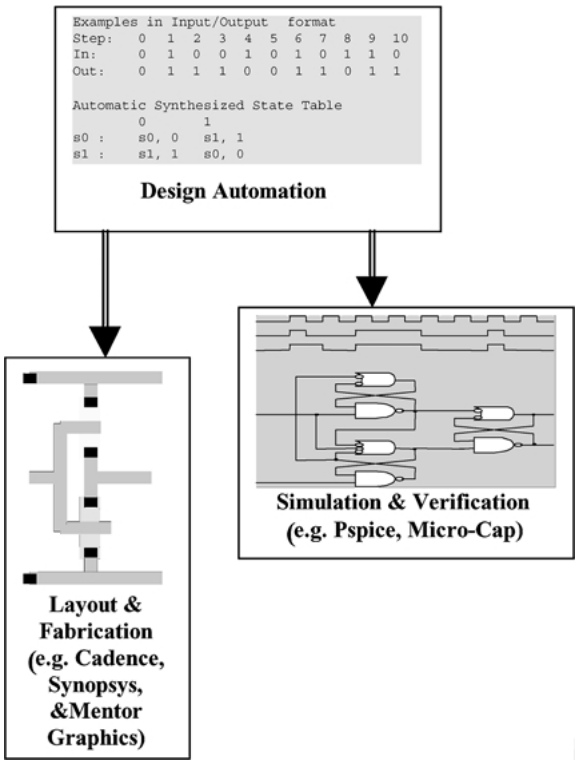


Figure 2. Integrated digital design automation system.

Our design automation system will provide, to the designer, the resultant design in terms of finite state machine that can then be translated to VHDL. The VHDL can then be provided to simulation and verification system and to layout and fabrication systems as outlined in Fig. 2, in which the Design Automation depicts only the input/output of the design automation system. Such an integrated digital design automation system is of central importance to meet the high demands for custom built IC and Application Specific IC (ASIC) [29], and to reduce the time-to-market that typically may take one to two years.

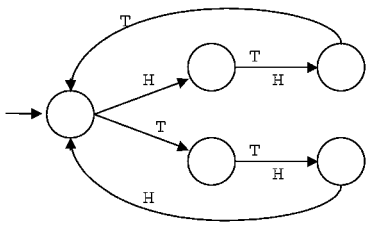
1.2. Organization

This paper is organized into nine sections. Section 1, this section, describes the motivations for proposing a new learning mechanism for learning by example systems and its application as design by example systems for digital design automation. Section 2 further compares this work with other related AI approaches. Section 3 defines formally the representations for the training data or design examples and the inferred models or designs. Section 4 provides a constructive mechanism for creating a new models or design. Section 5 shows mechanism for modifying an existing model or for refining an existing design. Section 6 discusses a mechanism for handling inconsistent information that may result from noise or from designer providing contradictory design examples. Section 7 provides several simple designs and training examples and their simulation results. Section 8 provides performance analysis for the proposed learning mechanism and digital design automation system. And, Section 9 gives the conclusions and future research directions.

2. Background

Other related AI approaches, described below, have their limitations and are not suitable for digital design automation. For example, several systems, outlined below, restrict their training data to special formats and their examples are not in time order. Thus they are not able to learn to form sequential circuits or to capture information of sequential events. For instance, the system described by Porat [30] requires the training data to be arranged into strict lexicographic order before being given to the system. The system also restricts the output to only one bit representing acceptance (denoted by $+$) or rejection (denoted by $-$). For example, using a and b as inputs, some strict lexicographic ordered examples are: $-$a, $+$b, $+$aa, and $-$ab. Another system restricts the training data to only output sequences. For example, using the sample output sequence shown in Fig. 3(b), the method reported in Rouvellou [31] is able to generate the state diagram shown in Fig. 3(a).

Another system [32] restricts the training data to input sequences and their associated classes. An input



(a) Generated state diagram

H T T T T H T H H T H H T H H T T H T H H H T T T T
H H H T T H H H H T H H T H T T H H T T H H etc.

(b) Sample output sequence

Figure 3. Learning from output sequence [31].

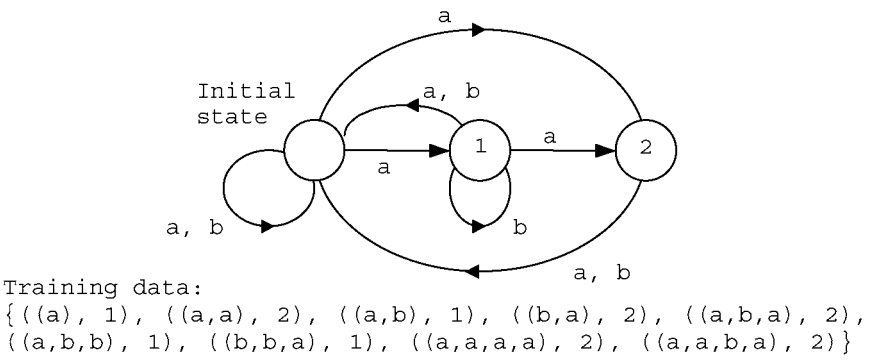


Figure 4. Building a state diagram form input sequences and their classes.

sequence and its associated class is represented by ((i1,i2,i3, ...), c) where i1,i2,i3, ... is the input sequence and c is the associated class. For example, a set of the training data is shown in Fig. 4. By using the training data, the method reported in Biermann [32] is able to construct the state diagram shown in Fig. 4. To construct the state diagram, the method reported in Biermann requires that several inputs be associated with one output. On the other hand, the system developed in this investigation allows each input to be associated with an output.

Some other systems can only learn from reward or punishment information [33–35]. The systems are first given a set of possible actions. They initially choose an action at random. During training, the trainers tell the systems whether the action is favorable. Then the systems update the likelihood for producing the action.

The system developed in this investigation does not pass computational burden to its teacher. It does not ask questions. On the contrary, several systems that pass the computational burden to their teachers have been reported in [36–40]. Those systems require their teachers to compute functions, such as, whether two sets of the training data are equivalent. For example, Angluin [37] provided an algorithm to infer a regular set that is described by a deterministic finite state acceptor. The algorithm uses a teacher that can answer membership queries as to whether an example is a member of the regular set. The teacher also provides counter-examples if the algorithm makes an incorrect guess. Ibarra [39] addressed the same problem but only allowed the teacher to answer equivalency queries. Marron [40] addressed the problem by using one initial example followed by membership queries.

The system developed in this investigation does not perform experiments on its environment. It learns from training data provided by a teacher. On the contrary, the system developed by Rivest [36] requires experiments to be performed on its environment. It also requires a teacher to determine the equivalency between the experiments. Performing experiments on the environment is similar to performing experiments on an unknown machine. Thus, the task for the learning system is similar to that of machine identification [41–43].

By using homing sequences together with experiments, Rivest [36] and Schapire [44] provided a method that does not require a means of resetting an unknown machine to an initial state. They used a notion called diversity, that is, equivalence between experiments or tests. Their notion of diversity can be viewed as equivalency between states, that is, two states are equivalent if every test produces the same value for both states. Their experiments can then be viewed as a distinguishing experiment described in Kohavi [43]. Their system was later implemented by Mozer [45] using neural networks.

3. Representing Training Data and Inferred Models

Before describing the mechanism for inferring a model or a design from examples, it is necessary to decide how to specify training data or examples and how to represent the inferred models or designs. This paper proposes to use input/output sequences to specify training data or examples and to use Finite State Machines to represent the inferred models, which are defined as follows.

3.1. Training Data or Design Examples

An example or a sample is specified by an input and an output, a stimulus and a response, or a condition and an action. A sequence of examples is an ordered set of examples. The order is important to capture our dynamic physical world since one example or sample may depend on some previous example. In terms of machine learning, a sequence of example is a training data set (an epoch). For our formulation, a sequence of example is specified by an input/output sequence.

Definition 1. Let I be a non-empty set of inputs and O be a non-empty set of outputs. An *input sequence* is defined to be $(x_0, x_1, \ldots, x_m) \in I^m$. An *output sequence* is defined to be $(y_0, y_1, \ldots, y_m) \in O^m$. And, an *input/output sequence* is defined to be $((x_0, y_0), (x_1, y_1), \ldots, (x_m, y_m)) \in (I \times O)^m$.

For $k > 0$, the output y_k corresponding to the input x_k may also depend on the previous output y_{k-1}, i.e. perhaps there exists a function F with $y_k = F(x_k, y_{k-1})$. There are special cases that are less complex than this. One special case is when an example does not depend on any previous examples, that is, there exists a function f with, $y_k = f(x_k)$. Another special case is when all the inputs are equal; in this case only the output sequence is of concern. For instance, when repeatedly tossing a coin, we are only concerned with the outcomes that are the y_k.

3.2. Knowledge Presentation and Inferred Models or Designs

This paper proposes to use finite state machines as models to describe sequences of examples. Finite state machines are particularly well suited for capturing the information of a dynamic system. In this investigation, they not only serve as compact ways to describe the given sequences of examples, but also serve as general statements that can be used to predict future outcomes, or as a design based on the given examples.

Definition 2. A *finite state machine* is defined to be a 6-tuple, $(I, O, S, s_0, \delta, \lambda)$, where I, O, and S are non-empty sets, $s_0 \in S$, $\delta : I \times S \to \rho(S)$, and $\lambda : I \times S \to O$.

Here $\rho(S) = \{A \mid A \subseteq S\}$ is the power set of S. The sets I, O, and S are interpreted as the set of inputs, the set of outputs, and the set of states, respectively. The state s_0, called the initial state, is defined as the starting state of the machine. The functions δ and λ are called the state transition function and the output function, respectively. They are usually defined by a table (called a *state table*) having each input in I as a column and each state in S as a row.

In general, this definition of a finite state machine is a *non-deterministic* finite state machine since, for $i \in I$ and $s \in S$, the *next state* $\delta(i, s)$, is defined up to an element in the power set of S. One special case is when $\delta(i, s) = S$ that specifies the next state can be any state in S including the *present state* s. Another case is when, for a state $t \in S$, $\delta(i, s) = \{t\}$, which specifies one single next state. If for all $i \in I$ and for all $s \in S$, the $\delta(i, s)$ specifies only one single next state, then the finite state machine is a *deterministic*.

A deterministic finite state machine will *produce* one output sequence when *given* an input sequence. A non-deterministic finite state machine may produce many output sequences when given an input sequence. The finite state machine *starts* at its initial state s_0, when *given* an input $i \in I$, it *produces* an output $o \in O$ and $o = \lambda(i, s_0)$; it *goes to* next state $T \in \rho(S)$ and $T = \delta(i, s_0)$. For each state $t \in T$, it now starts at t; when *given* an input $j \in I$, it *produces* an output $p \in O$ and $p = \lambda(i, t)$; it *goes to* next state $U \in \rho(S)$ and $U = \delta(i, t)$, and so on.

4. Mechanism for Creating a New Model or Design

Based on the formulations provided in the previous section, a sequence of examples is specified by an input/output sequence and the inferred model or design is presented by a finite state machine. Thus, the process to create a new model involves generating a finite state machine based on a given input/output sequence. The process of creating a new model also involves a process to minimize the complexity of the finite state machine by minimizing the number of states to form a minimal finite state machine.

The proposed process for creating a new model consists of six steps as outlined in Fig. 5. The first step converts an input/output sequence to an incompletely specified state table that represents a non-deterministic finite state machine. The resultant finite state machine captures the information of the input/output sequence. When this finite state machine is given the same input sequence it will generate the same output sequence.

Process for creating a new model
1. Creating an incompletely specified state table
2. Partitioning the states into compatible classes
3. Finding maximal compatibility classes
4. Deriving all prime compatibility classes
5. Finding a minimal closed cover
6. Deriving a minimal finite state machine

Figure 5. Steps for creating new model.

The following describes, as a theorem, the first step for constructing a finite state machine from the information provided by an input/output sequence.

Theorem 1. *Given an input/output sequence* (x_k, y_k), *for* $k = 0, 1, \ldots, m$, *there exists a finite state machine* $(I, O, S, s_0, \delta, \lambda)$ *that will produce the output sequence* (y_k) *when given the input sequence* (x_k).

Proof: This proof is constructive. Let I be the set of distinct entries in (x_k), O be the set of distinct entries in (y_k) and also let O contain the symbol '*', and let S be $\{0, 1, \ldots, m\}$.

The function δ is defined as, for $\forall i \in I, \forall s \in S$,

$$\delta(i, s) := \begin{cases} s+1, & \text{if } i = x_s \quad \text{and} \quad 0 \le s \le m-1 \\ S, & \text{otherwise} \end{cases}$$

The output function λ is defined as, for $\forall i \in I, \forall s \in S$,

$$\lambda(i, s) = \begin{cases} y_s, & \text{if } i = x_s \quad \text{and} \quad 0 \le s \le m \\ *, & \text{otherwise} \end{cases}$$

The initial state s_0 is assigned to state 0. Starting at state 0, when given the input x_0, based on the definition of $\lambda(x_0, 0)$, the finite state machine will produce y_0. Based on the definition of $\delta(x_0, 0)$, it will go to state 1. At state 1, when given x_1, it will produce y_1, and goes to state 2, and so on to the last state m. □

The rest of the steps for creating a new model or design, steps 2 to 6 (Fig. 5), are used as mechanisms for generalization and as mechanisms for creating a simple and compact model. These involve minimizing the number of states in the non-deterministic finite state machine. To do so, step 2 will partition the states into compatible classes. Step 3 will analyze the compatible classes and find maximal compatibility classes. Step 4 will derive subsets of the maximal compatibility classes and determine which classes have special property called prime. The prime compatibility classes are then used in step 5 to form a minimal closed cover that insure that the resulting minimized finite state machine will remain able to reproduce the same output sequence when given the same input sequence. And, step 6 used the result of step 5 to form a minimal finite state machine. Details of these steps for minimizing the number of states in the finite state machine are referred to other publications by the author and others [46–48].

5. Mechanism for Modifying an Existing Model or Design

The previous section described the proposed processes for creating a new model. This section will describe how to modify the model to incorporate the new information provided by a new sequence of examples. It also describes how to check whether a model is consistent with the new sequence of examples. The process outlined in Fig. 6 is developed in this investigation for both updating a model and checking consistency. The process incorporates new information into a model by adding new input symbols, defining previously unspecified outputs, and removing possible next states. Checking consistency insures that the updated model will remain consistent with previous examples.

5.1. Checking Consistency

To determine what to do with the new sequence of examples, our learning by example system must have a method to compare the new examples with the existing models. The new examples come to the learning system in the form of input/output sequences, while the system stores its models in the form of finite state machines (defined by state tables). Thus, we propose a method for comparing an input/output sequence with a finite state machine.

The proposed method for checking consistency is outlined in Fig. 6. The following provides explanations and definitions. The stored finite state machines can be deterministic or non-deterministic. A deterministic finite state machine produces one output sequence in responding to one input sequence. A non-deterministic finite state machine, on the other hand, can produce many output sequences in responding to one input sequence.

Definition 3. A finite state machine is *consistent* with an input/output sequence if in responding to the input sequence, the finite state machine produces at least

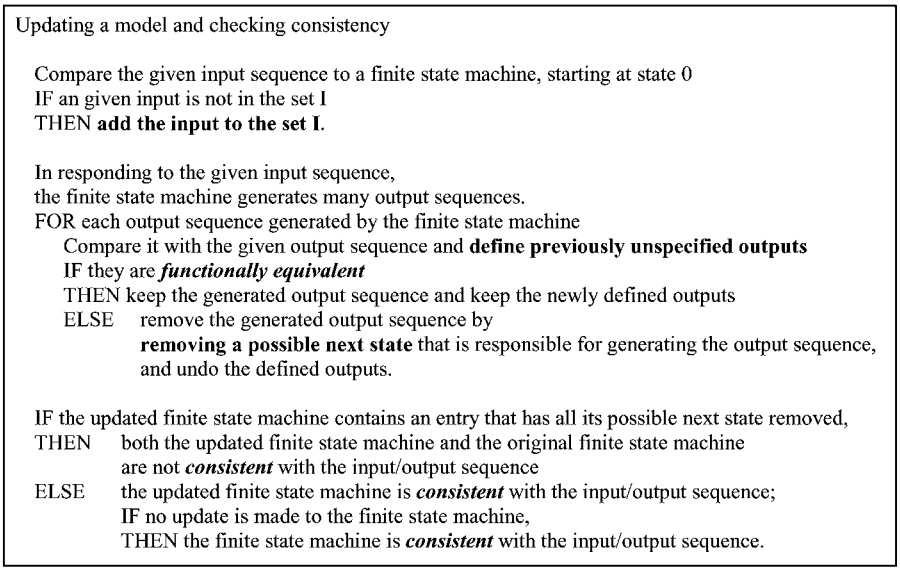


Figure 6. Updating a model and checking consistency.

one output sequence that is functionally equivalent to the given output sequence, and produces no output sequence that is not functionally equivalent to the given output sequence.

Definition 4. An *output sequence* $(p(k))$, for $k = 0, 1, \ldots, m$ is *functionally equivalent* to another output sequence $(q(t))$, for $t = 0, 1, \ldots, n$ if $m = n$ and for $k = 0, 1, \ldots, m$, each output $p(k)$ is functionally equivalent to output $q(k)$.

Definition 5. An *output c* is *functionally equivalent* to an output d (denoted as $c \approx d$) if $c = {}^*$ or $d = {}^*$ or $c = d$.

If the output c is represented by as a string of characters c_p for $p = 0, 1, \ldots, m$, and the output d is represented by a string of characters d_q for $q = 0, 1, \ldots, n$, then $c = d$ when $m = n$ and for all p, $c_p = ?$ or $d_p = ?$ or $c_p = d_p$. Where '*' denotes an unspecified output or called "don't care" in digital design automation, and '?' denotes a "don't care" character or a "don't care" bit that is particularly useful for digital design automation.

Based on the above definitions, our learning by example system will try to modify a model to make the model consistent with the new examples, while insure that the updated model will remain consistent with the previous examples.

5.2. *Adding New Input Symbols*

To allow our model to expand its scope, to generalize, or to cover more instances, an input symbol is added to the set I of the finite state machine when the input is in the input sequence but it is not in I. The process for adding new an input is illustrated by an example.

The following example also illustrates the process outlined in Fig. 6 for updating a model and checking consistency. As an example an input/output sequence (0/11, n/01, 1/?1) is compared to the finite state machine defined by state table shown in Fig. 7(a). The result of each step of the comparisons is shown in Fig. 7(b) while the resulting modified state table is shown in Fig. 7(c). The steps of the comparison are as follows.

(1) Starting at state 0, the finite state machine receives an input 0. It produces an output 1? (shown in sequence (a) in Fig. 7(b)) which is compared with the required output 11. To make the outputs functionally equivalent, the produced output 1?, specified the don't care character ? to 1, is now specified to 11. Since the next state is state 1, the finite state machine goes to state 1.

(2) In state 1, the machine receives an input n that is not in the set I. A new input column is created, which is labeled n (shown in Fig. 7(c)). All the entries in the new column are initialized to unspecified next state and unspecified output; shown as "S, *". The entry at the

State \ Input	0	1
0	1, 1?	1, 00
1	0, 10	0, 01

Next State, Output

a. Original state table.

Given Inputs	Sequence (a)	Sequence (b)	Required Outputs
0	0, 1?	=	11
n	1, 01	=	01
1	*0, 00*	*1, 01*	?1

Current state, Output

b. Sequences produced after adding new input column.

State \ Input	0	1	n
0	1, 11	1, 00	S, *
1	0, 10	0, 01	1, 01

Next state, Output

c. Resulting state table.

Figure 7. Example for updating a model.

current state that is state 1 is then set to "S, 01" in which the next state is the set S while the value of the output is that of the input/output pair "n/01" because at current state the machine is required to produce an output 01. The next state S is treated as next states 0 and next state 1. Thus, there are two *possible next states* for the current state. The next state 0 is checked first while next state 1 is saved in a stack to be checked later. Now the machine goes to state 0.

(3a) In state 0, the machine receives an input 1. It then produces an output 00 which is compared to required output ?1. The first characters of the two outputs can be functionally equivalent if the don't care character is set to 0, but the second characters cannot be functionally equivalent. Thus, the newly produced output sequence (shown in sequence (a) in Fig. 7(b)) is not functionally equivalent to the required output sequence. So the required output is restored back to ?1, and the newly produced output sequence needs to be removed. To do so, the possible next state 0 that is the result of step (2) is removed.

(3b) To generate sequence (b) (see Fig. 7(b)), the process performs backtracking and finds that in step (2) there is another possible next state that needed to be checked. It retrieves from the stack the possible next state that is state 1. The machine now goes to state 1 in which it receives input 1 and then produces output 01 which is compared to the required output ?1. These two outputs are functionally equivalent and the required output is now specified to 01. This completes sequence (b). In the figure, the equal sign "=" in the entry indicates that the value of that entry is equal to that on the left. To complete sequence (b) the process does not need to regenerate those values that are equal to that already generated in the sequence (a). Sequence (b) is functionally equivalent to the required output sequence. Thus, the possible next state 1 is kept. This competes the entire search process and the final result is shown in Fig. 7(c).

The resulting finite state machine has a new input symbol as shown in Fig. 7(c). The don't care output character ? in the original finite state machine is now specified to 1.

5.3. Defining Previously Unspecified Outputs

When a new sequence of examples contains new or more specific information, this information will be incorporated into existing model. One way to do so is by defining previously unspecified outputs in the finite state machine. The proposed method is explained and illustrated by an example.

To incorporate new information into existing model requires comparing the new information with information contained in the models. In our case this is partially accomplished by comparing the given output sequence with the output sequence generated by the finite state machine. Comparing two output sequences in which any output may contain don't care characters or may be an unspecified output requires a method for comparing two outputs and a method for keeping track of which don't care character or which unspecified output has been defined.

An example shown in Fig. 8 will illustrate the proposed method. There are two output sequences say sequence O (o1, o2, o3, o4) and sequence P (p1, p2, p3, p4). To complicate the matter, sequence O makes up of only three distinct output variables namely a, b, and c; those values will change as the result of the comparisons. Sequence O = (a, c, b, c). The initial values are a = 11?, b = ?10, and c = *, where * denotes an unspecified output, and ? denotes a don't care character. Sequence P makes up of only two distinct output variables namely x and y. The initial values are x = ?10 and y = 1?0. Sequence P = (x, y, x, y).

1. Start the comparison with the first pair, o1 to p1. Compare o1 (a = 11?) to p1 (x = ?10) resulting in

```
Sequence O (o1, o2, o3, o4)
       O = (a,  c,  b,  c )
Initial value: a = 11?  b = ?10  and  c = *

Sequence P (p1, p2, p3, p4)
       P = (x,  y,  x,  y )
Initial value: x = ?10  and  y = 1?0

Compare o1: a=11? to p1: x=?10  =>  a=110  x=110
Compare o2: c= *  to p2: y=1?0  =>  c=1?0  y=1?0
Compare o3: b=?10 to p3: x=110  =>  b=110  x=110
Compare o4: c=1?0 to p4: y=1?0  =>  c=1?0  y=1?0
```

Figure 8. Example for comparing two outputs sequences.

that this pair is functionally equivalent and that the values of a and x are both changed. Now they are a = 110 and x = 110 as well. The original don't care character in a is now specified to 0 while that of x is specified to 1.

2. Next compare o2 (c = *) to p2 (y = 1?0) resulting in there are functionally equivalent and the value of c is changed while that of y is unchanged. Now they are c = 1?0 and y = 1?0; the original unspecified output c is now specified to the specific value.
3. Next compare o3 (b = ?10) to p3 (x = 110); where the value of x is the result of the first comparison. The result of the comparison is that they are functionally equivalent and the value of b is changed. Now the values are b = 110 and x = 110.
4. Finally compare o4 (c = 1?0) to p4 (y = 1?0) resulting that they are functionally equivalent and both the values of c and y are unchanged.
5. Since all the corresponding pairs of sequence O and P are functionally equivalent, the final result is that these two sequences are functionally equivalent.

In this investigation, it is required to compare a given output sequence to many output sequences generated by a finite state machine. For the purpose of creating a modified finite state machine that is consistent with the given input/output sequence, a generated output sequence will be removed if it is not functionally equivalent to the given output sequence. The specified don't care characters or previously unspecified outputs, associated with the removed output sequence, will be restored. As described in the last example (Fig. 8) some of the don't care characters may be specified to specific values and some unspecified outputs may be specified. The results of the specification are kept if the two sequences turn out to be functionally equivalent.

5.4. Removing Possible next States

Another way to incorporate new information into existing model is by removing possible next states in the finite state machine. The new information helps the inductive system to distinguish possible outcomes, to limit the choices of many outcomes, or to eliminate non-deterministic states. This will restrict the scope of the model. The model becomes more specific and better defined.

Removing the next state corresponding to an output sequence will remove the output sequence. In order to produce a modified finite state machine that can be consistent with a given input/output sequence, the output sequence generated by a finite state machine will be removed if it is not functionally equivalent to the given output sequence. The proposed process is illustrated by an example.

As an example the input/output sequence (x/L, y/L, z/N) is compared to the finite state machine defined by the state table shown in Fig. 9(a). In responding to the input sequence (x, y, z), the finite state machine produces three state sequences and output sequences

State \ Input	x	y	z
0	0,1, L	1, L	0, M
1	0, K	0,2, L	1, N
2	1, M	0, K	0, M

Possible next states, …, Output

a. State table.

Inputs	(a)	(b)	(c)	Outputs
x	0, L	0, L	0, L	L
y	*0, L*	*1, L*	1, L	L
z	1, N	*0, M*	*2, M*	N

Current State, Output

b. Sequences produced.

Figure 9. Choosing possible states to be removed.

as shown in Fig. 9(b). Compare the required output sequence (L, L, N) to the three produced output sequences resulting in that output sequence (a) is functionally equivalent while output sequence (b) and (c) are not functionally equivalent. Both sequence (b) and (c) must be removed in order to produce a modified finite state machine that is consistent with the given input/output sequence.

There are two possible next states that are responsible for producing sequence (b): the possible next state 1 of present state 0 and input x, and the possible next state 0 of present state 1 and input y. Removing any one of the two possible next states will remove the possibility to produce sequence (b). Similarly there are two possible next states that are responsible for producing sequence (c): the possible next state 1 of present state 0 and input x, and the possible next state 2 of present state 1 and input y. Removing any one of the two possible next states will remove sequence (c).

To remove both sequence (b) and (c), there are two choices: (1) remove the possible next state 1 of present state 0 and input x; that possible next state is responsible for producing both the sequences, or (2) remove both possible next states 0 and 2 of present state 1 and input y. If choice (1) is chosen, then the resulting modified finite state machine will be consistent with the given input/output sequence and the modified finite state machine will remain consistent with those input/output sequences that are consistent with the original finite state machine. If choice (2) is chosen, then the resulting modified finite state machine will not consistent with those input/output sequences that are consistent with the original finite state machine. The reason for the later is that the modified finite state machine will now consist of an undefined next state (for present state 1 and input y), in which all possible next states have been removed.

A general rule for choosing which possible next states to remove is: (1) remove first those next states that are responsible for as few output sequences as possible; if the result of (1) is that all the possible next states of a particular present state and input have been removed, then (2) restore all those possible next states and then remove those next states that are responsible for the whole set of output sequences.

The reason for the general rule is that during the process for creating an updated model, it is necessary to ensure that (1) the updated model is consistent with the new input/output sequence and (2) the updated model remains consistent with those original input/output sequences that are consistent with the original model. To ensure these, the updated finite state machine must have remained at least one next state for every pair of present states and inputs.

6. Mechanism for Handling Inconsistent Information

The previous section described how to update an existing model based on the information provided by new sequences of examples. This section will describe methods that will be used when none of the existing models can be modified to account for the new examples or when the new examples contradict all existing models. The proposed method is to use multiple models. Each model is associated with a confidence factor called weight.

The proposed method for assigning confidence factor to each model or design is outlined as shown in Fig. 10. A number called "weight" is designed to each model. The weight for a newly created internal model equals 1 (as outlined in Fig. 10). Whenever the inductive system receives a new input/output sequence, it will check for consistency of the new sequence against all the existing models (Step (1) in Fig. 10). If a model is consistent with the new sequence, then its weight is increased by one; otherwise its weight is unchanged. (2) If a model can be modified so that the modified model is consistent with the new sequence and that remains consistent with the old sequences, then the newly modified model is added and its weight is one plus the weight of the original model. (3) Otherwise, a completely new model is constructed based on the information provided by the new input/output sequence, and its weight is one.

This process uses other processes described earlier. (1) It detects any inconsistency by using the process for checking consistency (described in Section 5.1).

```
Assigning weight to each model
    The weight for a newly created model equals 1
    FOR each existing model
      IF (1) an input/output sequence is consistent with the model
      THEN increase by one the weight of the model
      ELSE IF the input/output sequence is consistent with
                an updated model
      THEN (2) store the updated internal model,
                its weight equals one plus that of the original model.
    IF the input/output sequence is not consistent with any existing models
    THEN (3) create a new model
              its weight equals one.
```

Figure 10. Process for assigning weight to each model.

(2) It incorporates new useful information into the existing models by using the processes for updating models (outlined in Section 5). And, (3) it records inconsistent information by using the process for creating new model (described in Section 4).

If inconsistent information occurs less frequently than useful information, then the weights of the models that result from inconsistent information will be lower than the weights of the good models. The model having the highest weight is the most favorable model.

7. Training the System and Simple Designs

Five training examples are provided in the following to illustrate the training and the abilities of the learning system. The training examples are: a parity checker, an up-down counter, a sequence detector, a task involving many states, and a simple code breaker.

The training examples are done on a simulation program developed in this investigation to test the learning system. The simulation program is named Archetype. A screen shot of the program is shown in Fig. 11. As shown in the figure the program provides five commands to a user.

7.1. A Parity Checker

In this section we will train the system to perform the task of a parity checker. We will test whether the system is able to learn the task by using only one sequence of input/output examples. We specify what we mean by a parity checker by using the input/output sequence shown in Fig. 12. At any step k, the parity checker is required to produce an output 1 for an odd number of 1's in the input sequence up to k. It produces an output 0, at step k, for an even number of 1's in the input sequence up to k.

Processing the input/output sequence (Fig. 12), Archetype produces the state table (Fig. 13) by using the process for creating a new internal model described

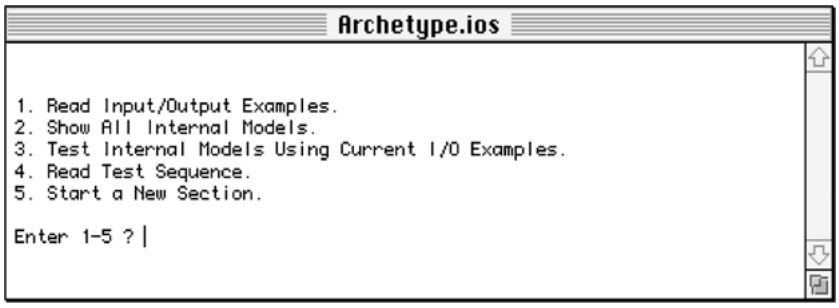


Figure 11. A screen shot of the simulation program.

```
Input/Output Sequence
Step:   0   1   2   3   4   5   6   7   8   9   10
In:     0   1   0   0   1   0   1   0   1   1   0
Out:    0   1   1   1   0   0   1   1   0   1   1
```

Figure 12. Parity checker: I/O sequence.

```
(8 nodes are used by min closed-cover search tree.)
State Table, Weight = 1
         0       1
s0 :    s0, 0   s1, 1
s1 :    s1, 1   s0, 0

New state table added.
```

Figure 13. Parity checker: Resultant state table.

```
Input/Output Sequence
Step:  0    1    2      3     4      5     6     7     8      9
In:    up   up   up     up    down   down  down  down  down   up
Out:   One  Two  Three  Z??o  Three  Two   One   Zero  T??ee  Zero
```

Figure 14. Counter: I/O sequence.

```
(230 nodes are used by min closed-cover search tree.)
State Table, Weight = 1
             up          down
s0 :         s2, One     s1, Three
s1 :         s0, Zero    s3, Two
s2 :         s3, Two     s0, Zero
s3 :         s1, Three   s2, One

New state table added.
```

Figure 15. Counter: Resultant state table.

in Section 4. As demonstrated, the system is able to learn the design of the parity checker by using only one sequence of input/output examples that consists 11 steps. The process requires only 8 nodes in the minimum closed-cover search tree that is provided for evaluating the performance of the process (described in Section 8).

7.2. An Up-Down Counter

In this section we will train the system to perform the task of an up-down counter. We will test whether the system is able to learn the task by using only one sequence of input/output examples (shown in Fig. 14).

This training example also shows that the system is able to handle inputs and outputs that are represented by string of characters (or multiple-bit binary numbers) having various lengths. It also shows that the system is able to take don't care characters. For example, in two of the outputs, i.e., "Z??o" and "T??ee", there are unspecified or don't care characters each of which is denoted by a "?". Don't care characters or don't care bits in a binary number are useful in the design of digital circuits.

Processing the input/output sequence (Fig. 14), Archetype produces the state table shown in Fig. 15. As demonstrated, the system is able to learn the design of a four-state up-down counter by using only one sequence of input/output examples, despite some of outputs in the sequence contains unknown characters.

7.3. A Sequence Detector

In this section we will train the system to perform the task of a sequence detector. Four sequences of input/output examples will be used. One of the sequences contains errors or "noise" so that it is not consistent with other sequences.

Suppose that we are going to design a sequence detector that detects three consecutive 1's in an input sequence and then produces an output 1, otherwise it produces an output 0. We will do so by providing examples to the system. The job of the system is to derive a model of the sequence detector from those examples.

We provide the first sequence of input/output examples as shown in Fig. 16. Processing the input/output sequence, Archetype creates the state table (Fig. 17). As shown in Fig. 17, there are three possible next states "`s0, s1, s2`" for present state "`s2`" and input "`1`". The reason for the three possible next states is that our examples do not provide sufficient information for the system to determine a single next state.

```
Input/Output Sequence
Step:   0   1   2   3   4
In:     0   1   1   1   0
Out:    0   0   0   1   0
```

Figure 16. Sequence detector: I/O sequence 1.

```
 (4 nodes are used by min closed-cover search tree.)
State Table, Weight = 1
                0               1
s0 :            s0, 0           s1, 0
s1 :            s0, 0           s2, 0
s2 :            s0, 0           s0,s1,s2, 1

New state table added.
```

Figure 17. Sequence detector: State table 0.

```
Input/Output Sequence
Step:   0   1   2   3   4   5   6   7   8   9
In:     0   1   1   1   0   1   1   0   1   0
Out:    0   0   0   1   0   0   0   0   0   0
```

Figure 18. Sequence detector: I/O sequence 2.

```
(22 nodes are used in checking and updating the internal model.)
The i/o sequence is consistent with state table 0
```

Figure 19. Sequence detector: Result of sequence 2.

To eliminate uncertainty we need more examples. We provide another set of examples as shown in Fig. 18. Using the process for "checking consistency and updating internal models" described in Section 5, Archetype finds that the new set of examples (Fig. 18) is consistent with the existing state table (Fig. 17). As shown in Fig. 19, only 22 nodes are used in the search tree for checking consistency and updating internal models. The number of nodes used in the search tree is provided for evaluating the performance of the process (described in Section 8).

As shown in Fig. 20, the state table (Fig. 17) is not changed except that its weight is increased by one, which is the result of the process described in Section 6. Although another set of examples is provided, the information contained in the examples is already there in the existing model. Thus the state table is not changed.

Now we provide another set of examples. However, the examples contain two errors. As shown in Fig. 21, the output of step 3 should be a 1 not a 0. The output of step 6 should be a 0 not a 1.

By using the process for checking consistency and updating internal models, Archetype detects that the new input/output sequence is not consistent with the existing state table (Fig. 20). Archetype records the inconsistency by using the process for handing inconsistent information and creates a new state table to capture the inconsistent information as shown in Fig. 22.

```
0 State Table, Weight = 2
        0           1
s0 :    s0, 0       s1, 0
s1 :    s0, 0       s2, 0
s2 :    s0, 0       s0,s1,s2, 1
```

Figure 20. Sequence detector: State table 0 after sequence 2.

```
Input/Output Sequence
Step:   0   1   2   3   4   5   6   7
In:     0   1   1   1   0   1   1   0
Out:    0   0   0   0   0   0   1   0
```

Figure 21. Sequence detector: I/O sequence 3.

We give another set of examples as shown in Fig. 23. Using the process for checking consistency and updating internal models, Archetype updates state table 0 (Fig. 20) by removing possible next states "`s1, s2`" from the list of possible next states associated with

```
(4 nodes are used in checking and updating the internal model.)
The i/o sequence is not consistent with state table 0

(12 nodes are used by min closed-cover search tree.)
State Table, Weight = 1
                0               1
s0 :            s1, 0           s2, 0
s1 :            s0, 0           s1, 0
s2 :            s0, 0           s0,s1,s2, 1

New state table built and added
```

Figure 22. Sequence detector: State table 1.

```
Input/Output Sequence
Step:   0   1   2   3   4   5   6   7
In:     0   1   1   1   1   1   1   0
Out:    0   0   0   1   0   0   1   0
```

Figure 23. Sequence detector: I/O sequence 4.

present state s2 and input 1. The process for removing possible next states is described in Section 5.4. The updated state table is shown in Fig. 24. Archetype does not overwrite the original state table 0. It adds a new updated version of the state table.

Next it checks state table 1, which is the result of the set of examples that contains two errors. It finds that the input/output sequence is not consistent with the state table 1 (Fig. 22).

As the result of the above four set of examples, Archetype builds three state tables as shown in Figs. 20, 22 and 24. State table 2 (Fig. 24) has the highest weight, and is considered as the best model or best design based on the given examples.

These set simulations demonstrate that the system developed in this investigation is able to accept sequences of input/output examples for the design of simple sequential machines. The system allows a designer to provide additional examples to fine tune the design. It also tolerates infrequent errors made by the designer.

7.4. Other Simulation Results

The simulation examples for designing a parity checker, an up-down counter, and a sequence detector, are described in the previous sections. Their simulation results together with that of designing a task involving many states and a simple code breaker are provided in Figs. 25 and 26.

8. Performance Analysis

The performance of the processes for constructing a minimal finite state machine from an input/output

```
(13 nodes are used in checking and updating the internal model.)
The i/o sequence is consistent with an updated version of state table 0

State Table, Weight = 3
        0       1
s0 :    s0, 0   s1, 0
s1 :    s0, 0   s2, 0
s2 :    s0, 0   s0, 1

The updated state table added.

(4 nodes are used in checking and updating the internal model.)
The i/o sequence is not consistent with state table 1
```

Figure 24. Sequence detector: State table 2.

Name of application examples	Number of steps in the input/output sequence	Number of states in the generated state table	Number of nodes used by the minimum closed- cover search tree
An up-down counter	10	4	230
A parity checker	11	2	8
A sequence detector	5	3	4
	8	3	12
A task involving many states	30	30	31
	100	100	101
A simple code breaker	9	1	2
	8	1	2

Figure 25. Summary of simulation results for creating new models.

Name of application examples	Number of steps in the input/output sequence	Number of states in the state table used for checking	Number of nodes used by the process for checking and updating model
A sequence detector	10	3	22
	8	3	4
	8	3	13
	8	3	4
A task involving many states	7	30	7470
	7	100	94690
A simple code breaker	9	1	9
	8	1	6
	8	1	5

Figure 26. Summary of simulation results for checking and updating models.

sequence (Section 4) is limited by the step for finding a minimum closed cover, which has a search space having an upper bound of p^m nodes, where p is the number of prime compatibility classes, and m is the number of classes in the minimum closed cover. Further analysis of the complexity of inferring general rules from given data can be found in Gold [49]. To reduce the search space, the method implemented in this investigation is to use near-minimal finite state machine whenever a minimal solution is impractical [50]. A near-minimal finite state machine is less compact and represents a less general description comparing to a minimal finite state machine. However, the learning process remains sound.

To find a near-minimal finite state machine instead of a minimum one, the only modification needed is to skip the process of deriving all prime compatibility classes (Section 4). The set of all maximum compatibility classes is then used as the prime compatibility classes for finding a minimum closed cover. Since the set of all maximum compatibility classes is usually much smaller than the set of all prime compatibility classes, the solution greatly reduces the search space.

The process for checking consistency and updating models (described in Section 5) has a search space having an upper bound of q^k nodes, where q is the number of incompletely specified next states in a state table, and k is the number of steps in a given input/output sequence. A large number of incompletely specified next states in a state table could result in a very large number of output sequences that can be generated by the finite state machine in responding to a given input sequence.

To solve this problem, two methods are used to reduce the search space by eliminating branches in the search trees. The first method is to use a depth-first search process. The depth-first search process is stopped as soon as an inconsistency is found. Thus, not all the output sequences are required to be generated or compared. The second method is incorporated in the process for removing possible next states (described in Section 5.4). Removing a possible next state corresponding to an output sequence will remove the possibility for generating the output sequence. Since a next state is usually responsible for generating many output sequences, removing one next state will prevent many other corresponding output sequences from being generated.

9. Conclusions and Future Research

This paper describes a new learning by example mechanism that is well suited for digital circuit design automation. The mechanism is conceived partially based on automata and switching theories. By using the mechanism the resultant models are finite state machines that are easy to be implemented in hardware using current VLSI technologies. Our simulation results show that it is often possible to infer a well-defined deterministic model or design from just one sequence of examples. In addition this mechanism is able to handle sequential task involving long-term dependence.

This new learning by example mechanism is used as a design by example system for digital circuit design automation. The designer provides input/output examples of a required digital circuit to the system. The system is able to generate the circuit design. The designer can provide additional examples to refine the design. The system is able to take the additional examples and generates a refined design. The designer may provide two or more sets of examples that are not compatible with each other. The system is able to inform the designer that there are incompatible sets of examples. Several simple sequential circuit design examples are simulated and tested to determine the feasibility of

such a system. Although our test results show that such a system is feasible for designing simple circuits or small-scale circuit modules, it remains to be shown the feasibility of such a system for large-scale circuit design.

Although both the learning mechanism and the design method show potential, much future work remains. This work shows that design by example is a feasible concept for designing small-scale circuit modules. This concept should be extended either for designing larger-scale circuits or for other applications. Future work includes improving the inference mechanism provided in this paper or creating more effective mechanism for inferring finite state machines from given examples. The mechanism provided in the paper may be extended for inferring timed and hybrid automata [51–54] from given examples.

Acknowledgments

Thanks go to Professor Dick Greechie and anonymous reviewers for proofreading and suggesting many changes to make this paper more readable.

References

1. P. Frasconi, M. Giro, M. Maggini, and G. Soda, "Unified integration of explicit knowledge and learning by example in recurrent networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 2, pp. 340–346, 1995.
2. A. Blum and A. Kalai, "Note on learning from multiple-instance examples," *Machine Learning*, vol. 30, no. 1, pp. 23–29, 1998.
3. I. Pitas, E. Milios, and A.N. Venetsanopoulos, "A minimum entropy approach to rule learning from examples," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 22, no. 4, pp. 621–635, 1992.
4. S.P. Eberhardt, R. Tawel, T.X. Brown, T. Daud, and A.P. Thakoor, "Analog VLSI neural networks—implementation issues and examples in optimization and supervised learning," *IEEE Transactions on Industrial Electronics*, vol. 39, no. 6, pp. 552–564, 1992.
5. O. Vermesan, "A modular VLSI architecture for neural networks implementation," *From Natural to Artificial Neural Computation*, vol. 930, pp. 794–799, 1995.
6. S. Espejo, R. Carmona, R. DominguezCastro, and A. RodriguezVazquez, "A CNN universal chip in CMOS technology," *International Journal on Circuit Theory and Applications*, vol. 24, no. 1, pp. 93–109, 1996.
7. *Proceedings of the 1998 Symposium on VLSI Technology Source, Symposium on VLSI Technology* (sponsored by IEEE), 1998.
8. A. Moini, K. Eshraghian, and A. Bouzerdoum, "Impact of VLSI technology on neural networks," *IEEE International Conference on Neural Networks*, pp. 158–163, 1995.
9. D.E. Rumelhart, G.E. Hinton, and R.J. Willians, "Learning internal representations by error propagation," *Parallel Distributed Processing*, vol. 1, 1987.
10. R. Hecht-Nielsen, *Neurocomputing*, Addison-Wesley: Reading, MA, 1990.
11. T. Khanna, *Foundations of Neural Networks*, Addison-Wesley: Reading, MA, 1990.
12. H.C. Anderson, "Neural network machines," *IEEE Potentials*, pp. 13–16, Feb. 1989.
13. J.B. Pollack, "Connectionism: Past, present, and future," *Artificial Intelligence Review*, vol. 3, pp. 3–22, 1989.
14. N.J. Nilsson, *The Mathematical Foundations of Learning Machines*, Morgan Kaufmann Publishers: San Mateo, CA, 1990.
15. S. Bibyk and M. Ismail, "Issues in analog VLSI and MOS techniques for neural computing," in *Analog VLSI Implementation of Neural systems*, edited by C. Mead and M. Ismail, Kluwer Academic Publishers, pp. 103–133, 1989.
16. J. Hootman, (Ed.), *IEEE Micro*, Special Issue on Silicon Neural Networks, Dec., 1989.
17. C. Mead, *Analog VLSI and Neural Systems*, Addison-Wesley: Reading, MA, 1989.
18. C. Mead and M. Ismail, *Analog VLSI Implementation of Neural Systems*, Kluwer Academic Publishers: Boston, 1989.
19. M. Ismail and T. Fiez, *Analog VLSI Signal and Information Processing*, McGraw-Hill: New York, 1994.
20. H.C. Card, D.K. McNeill, and C.R. Schneider, "Analog VLSI circuits for competitive learning networks," *Analog Integrated Circuits and Signal Processing*, vol. 15, no. 3, pp. 291–314, 1998.
21. P.S. Eberhardt, R. Tawel, T.X. Brown, T. Daud, and A.P. Thakoor, "Analog VLSI neural networks: Implementation issues and examples in optimization and supervised learning," *IEEE Transactions on Industrial Electronics*, vol. 39, no. 6, pp. 522–564, 1992.
22. M.W. Roth, "Survey of neural network technology for automatic target recognition," *IEEE Transactions on Neural Networks*, vol. 1, no.1, pp. 28–43, 1990.
23. C.L. Giles, G.M. Kuhn, and R.J. Williams (Guest Editors), "Special issue on dynamic recurrent neural networks, *IEEE Transactions on Neural Networks*, vol. 5, no. 2, 1994.
24. M.I. Jordan, "Attractor dynamics and parallelism in connectionist sequential machine," in *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pp. 521–545, 1986.
25. Y. Bengio, P. Frasconi, and P. Simard, "The problem of learning long-term dependencies in recurrent networks," in *IEEE International Conference on Neural Networks*, 1993.
26. Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, 1994.
27. K.J. Breeding, *Digital Design Fundamentals*, Prentice Hall: Englewood Cliffs, NJ, 1992.
28. K.C. Chang, *Digital Systems Design with VHDL and Synthesis: An Integrated Approach*, IEEE Computer Society: Los Alamitos, CA, 1999.
29. H. Dicken and M. Griffith (Eds.), *ASIC Outlook, 1998: An Application Specific IC Report and Directory*, Integrated Circuit Engineering Corporation, Nov. 1997.
30. S. Porat and J.A. Feldman, "Learning automata from ordered examples," in *Proceedings of the 1988 Workshop on*

Computational Learning Theory (also on *Machine Learning*, 1991), 1988.
31. I. Rouvellou and G.W. Hart, "Inference of a probabilistic finite state machine from its output," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 25, no. 3, pp. 424–437, 1995.
32. A.W. Biermann and J.A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, vol. 21, pp. 592–597, 1972.
33. B.J. Oommen, N. Andrade, and S. Iyengar, "Trajectory planning of robot manipulators in noisy work spaces using stochastic automata," *International Journal of Robotics Research*, vol. 10, no. 2, pp. 135–148, 1991.
34. A.C. Barto and P. Anandan, "Pattern-recognizing stochastic learning automata," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-15, no. 3, pp. 360–375, 1985.
35. K. Lanctot and B.J. Oommen, "Discretized estimator learning automata," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 6, 1992.
36. R.L. Rivest and R.E. Schapire, "Inference of finite automata using homing sequences," in *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, May 15–17, pp. 411–420, 1989.
37. D. Angluin, "Learning regular sets from queries and counter examples," *Information and Computation*, vol. 75, pp. 87–106, 1987.
38. D. Angluin, "A note on the number of queries needed to identify regular languages," *Information and Control*, 1981.
39. O.H. Ibarra and T. Jiang, "Learning regular languages from counterexamples," in *Proceedings of the 1988 Workshop on Computational Learning Theory*, 1988.
40. A. Marron, "Learning pattern languages form a single initial example and from queries," in *Proceedings of the 1988 Workshop on Computational Learning Theory*, 1988.
41. E.M. Gold, "System identification via state characterization," *Automatica*, 1972.
42. L.A. Litteral, "An algorithm for solving the sequential machine identification problem," Thesis, The Ohio State University, 1973.
43. Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill Book Company: New York, 1978.
44. R.E. Schapire, "Diversity-based inference of finite automata," MIT/LCS/TR-414, 1988.
45. M.C. Mozer and J. Bachrach, "SLUG: A connectionist architecture for inferring the structure of finite-state environments," *Machine Learning*, vol. 7, pp. 139–160, 1991.
46. B. Choi, "Automata for learning sequential tasks," *New Generation Computing*, vol. 16, pp. 23–54, 1998.
47. A. Grasselliand and F. Luccio, "A method for minimizing the number of internal states in incompletely specified sequential networks," *IEEE Transactions on Electronic Computers*, vol. 14, no. 3, pp. 350–359, 1965.
48. W.S. Meisel, "A note on internal state minimization in incompletely specified sequential networks," *IEEE Transactions on Electronic Computers*, pp. 508–509, 1967.
49. E.M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, pp. 306–320, 1978.
50. L. Pitt and M.K. Warmuth, "The minimum consistent DFA problem cannot be approximated within any polynomial," in *Proceedings of the 21 Annual ACM Symposium on Theory of Computing*, pp. 421–432, 1989.
51. R. Alur and D.L. Dill, "Theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.
52. C. Largouët and M.O. Cordier, "Timed automata model to improve the classification of a sequence of images," in *Proc. of 14th European Conference on Artificial Intelligence (ECAI'2000)*, 2000, pp. 156–160.
53. A. Puri, "Undecidable problem for timed automat," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 9, no. 2, pp. 135–146, 1999.
54. R. Alur, R.P. Kurshan, and M. Viswanatha, "Membership questions for timed and hybrid automata," in *Proceedings of Real-Time Systems Symposium*, 1998, pp. 254–263.

Ben Choi, Ph.D. & Pilot: He is an Asst. Professor in Computer Science at Louisiana Tech University. Before he joined the academia he worked in the computer industry as a System Performance Engineer at Lucent Technologies—Bell Labs Innovations. He got his Bachelor of Science, Master of Science, and Ph.D. all from The Ohio State University. His major areas of study include Solid State Microelectronics, Computer Engineering, and Computer Science. He has works on general associative memory, parallel and distributed computer architectures, and machine learning. His research interests include hardware and software methods of building intelligent machines. He is a member of IEEE and ACM and is a pilot of airplanes and helicopters.