GENERALIZING LOGIC CIRCUIT DESIGNS BY ANALYZING PROOFS OF CORRECTNESS*

Thomas Ell man
Department of Computer Science
Columbia University
New York, New York 10027

Abstract

This paper presents a method of learning to solve design problems by generalizing examples. The technique has been developed in the domain of logic circuit design. It involves the use of domain knowledge to analyze examples and produce generalized circuit designs. The method utilizes proofs of design correctness to guide the process of generalization. Our approach is illustrated by showing it can generalize a rotational shift register into a schema describing devices capable of computing arbitrary permutations.

Introduction

Research in machine learning has identified two contrasting approaches to the problem of learning from examples. The traditional "empirical" approach is based on the idea that an intelligent system can learn from examples without having much prior knowledge of the domain of application. This approach has involved looking at a large number of examples in order to identify similar features. It usually relies on syntactic methods of matching instances and correlating the common features. Examples of this approach include [Winston 72]; [Michalski 80], among others. The alternative "analytical" approach takes a different point of view. It assumes that generalization requires a great deal of background knowledge of the domain under study. It typically relies on intensive analysis of a single example in order to generalize. The work reported in this paper takes the analytical approach. It has been applied to the problem of learning to design logic circuits. The method involves analyzing single examples of circuit designs and producing generalized designs.

In order to illustrate the technique, consider the circular shift register circuit shown in Figure 1. This device is capable of four operations, rotate right, rotate left, load and no-operation. The operations are controlled by the two bit "select" line (Figure 2). The circuit has been designed using d-type flip-flops, labeled "DFF", and multiplexers, labeled "MUX". A human novice would be capable of generalizing from this example provided he understands the principle of operation behind the circuit. For example, he must know that a d-flip-flop can store one bit of information and that its output is equal to its input delayed by one time unit. The multiplexers are used to route one of four inputs to an output determined by values on their select lines. A novice who understands the operation of this circuit could probably produce similar designs which compute any permutation of four bits. This merely requires connecting the d-flip-flop outputs to the multiplexer inputs in a manner consistent with the chosen permutation. Generalization from the single example is possible because all the other permutations can be implemented using the same principle of operation.

*This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0165.

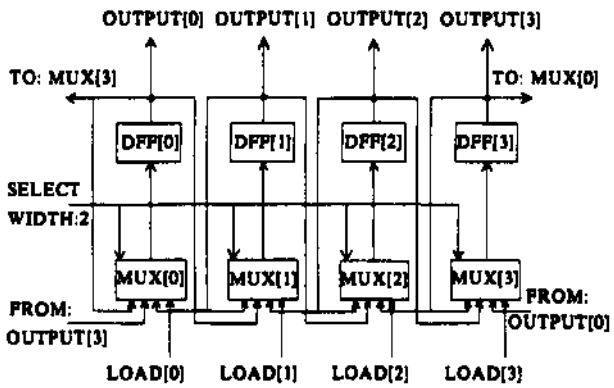

Figure 1: Circular Shift Register

SELECT-CODE	OPERATION
(0 0)	No Operation
(0 1)	Rotate Right
(1 0)	Rotate Left
(1 1)	Load

Figure 2: Control Codes for Circular Shift Register

This paper will describe a program that attempts to model the behavior of the human novice. The program is able to successfully generalize the shift register into a circuit schema capable of implementing any permutation. The system is given enough background knowledge about the operation of devices like multiplexers and d-flop-flops so that it can understand the operation of the shift register. This knowledge takes the form of rules which can be used to prove that the example design is correct. The original example is generalized into a schema describing all circuits that can be verified using the same proof of correctness.

This research is similar in spirit to previous work on analytical methods of generalization. These analytical approaches include "goal-directed learning" [Mitchell 83a], "explanatory schema acquisition" [DeJong 83], "derivational analogy" [Carbonell 83] and "learning from precedents" [Winston 83]. The research reported here is also related to the work on "circuit redesign" reported in [Mitchell 83b].

The approach taken there involves designing a new circus by analogy with a previously designed circuit. Our work is different mainly in that it focuses on generalization, rather than analogy. The technique of explanatory schema acquisition reported in [DeJong 83] is similar to ours, although the domain of application is quite different. Our work also differs by focusing on design problems and generalising both designs and specifications. Other related work includes [Minton 84; Mostow 83a; Mostow 83b; Salzberg 83; Silver 83].

The Learning Task

Our learning program is envisioned as a component of a complete system for designing circuits according to explicit specifications. The problem solving module for such a system would take circuit specifications as input and produce circuit designs as output. The learning module deals with both specifications and designs. It is intended to take as input a pair (S,D) consisting of specifications and a design which correctly implements the specifications. The goal of the learning process is to produce a generalized design schema (S*,D*) consisting of generalized specifications and a generalized design. The learning system must generalize the original pair subject to the constraint that the general design correctly implements the general specifications. The entire process has the following four steps.

1. A sample specification and design is input by a teacher.
2. A correctness proof is built to verify the design.
3. The proof is used to guide generalization of tne example.
4. The generalized schema is used to solve new problems.

We have chosen to focus on the third step which involves using the proof to facilitate generalization. There are several reasons for concentrating on the generalization step. A great deal remains unknown about precisely how causal reasoning may be used to enable generalization. Furthermore, correctness proofs may take a variety of forms and the choice may impact on the extent to which the proofs are a useful aid for generalization. This reasoning suggests investigating the generalization process first and letting the proofs be designed to fit the requirements of generalization. Our generalization program uses proofs built by hand as input. The task of automatically building explanations has not yet been implemented. The task of building proofs is similar to other understanding problems, and has been studied before. The CRITTER system [Kelly 82], is an example.

The Circular Shift Register Example

The generalization program works by analyzing three pieces of information (S,D,PJ. Two of the inputs are the specifications S and the design D as described above. The third input "P" is a proof tree which verifies that the design correctly implements the specification. The specifications for the circular shift register are shown in Figure 3. These specifications contain a list of inputs and outputs, as well as clauses describing the behavior of each of the four output lines. Each clause specifies the value of an output line at time "T" as a function of the values on the input lines at an earlier time, "T - 1". The "Case" statement is used to specify the circuit's behavior for each of four control states of the SELECT lines. The design of the circular shift register is represented by the data in Figure 4. This representation describes the electrical components and the wire connections between their ports.

Declarations of Inputs and Outputs

Wire	Direction	Width (Bits)
SELECT	Input	2
LOAD[0:3]	Input	1
OUTPUT[0:3]	Output	1

Description of Circuit Behavior

```
{This clause describes the behavior of Output[0].
Clauses describing the other outputs are analogous.}

(Equals OUTPUT[0].(T)
        (Case SELECT.(T - 1)
              ( (0 0)  OUTPUT[0].(T - 1) ) {No operation}
              ( (0 1)  OUTPUT[3].(T - 1) ) {Rotate Right}
              ( (1 0)  OUTPUT[1].(T - 1) ) {Rotate Left}
              ( (1 1)  LOAD[0].(T - 1) ))) {Load}
```

Figure 3: Specifications for Circular Shift Register

Declarations of Circuit Components

```
DFF[0:3]   {Four D-Flip-Flops.}
MUX[0:3]   {Four Multiplexers.}
```

Wire Connections Between Components

```
{Only the wire connections involving stage zero are shown.
Connections involving the other stages are analogous.}

{Connect Multiplexers to D-Flip-Flops.}
(CONNECT DFF[0].INPUT        MUX[0].OUTPUT)

{Connect Multiplexer Input[0]'s for No-Operation.}
(CONNECT MUX[0].INPUT[0]     DFF[0].OUTPUT)

{Connect Multiplexer Input[1]'s for Rotate Right.}
(CONNECT MUX[0].INPUT[1]     DFF[3].OUTPUT)

{Connect Multiplexer Input[2]'s for Rotate Left.}
(CONNECT MUX[0].INPUT[2]     DFF[1].OUTPUT)
```

Figure 4: Design Statements for Circular Shift Register

Proofs of correctness are represented as trees. Figure 5 shows a proof tree that verifies a clause describing the behavior of one stage of the shift register. The leaves represent facts about the design and the root contains the specification to be verified. Hence the tree represents a derivation of the specification taking the design statements as assumptions. This proof tree uses four derivation rules. Two of them, the "Dff-Rule", and the "Mux-Rule" describe the behavior of components. The Dff-Rule asserts that a d-flip-flop output signal at time "T" is equal to the d-flip-flop input signal at time "T - 1". The Mux-Rule describes how a multiplexer can be used to implement a case statement. The "Connect-Rule" asserts that two connected wires have the same signal values at all times. (Ignoring propagation delay.) Finally, the "Replace-Rule" allows equal quantities to be substituted for each other in expressions.

Using the Proof to Generalize

Our generalization procedure is intended to produce a schema (S*,D*) describing all circuit designs that can be proven correct using the original correctness proof. The

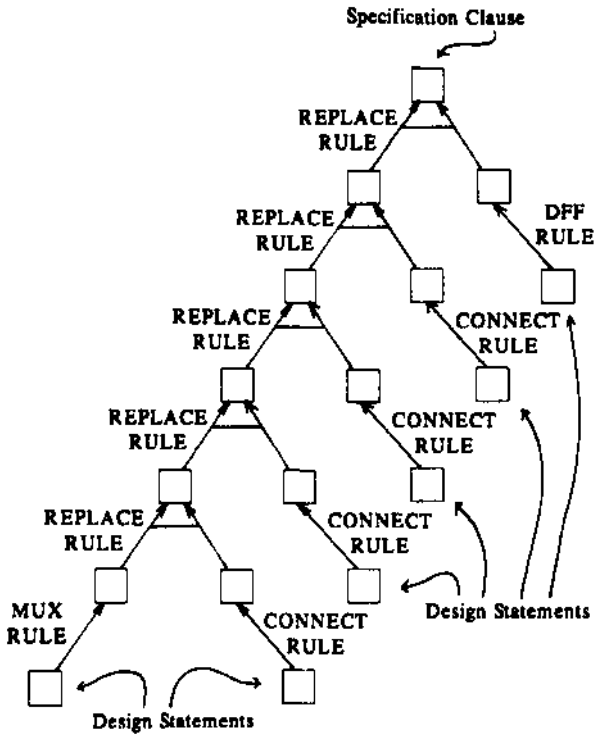

Figure 5: A Portion of the Correctness Proof Tree

proof tree contains information which may be used to identify constraints that must be preserved as the example is generalized. For this purpose, the proof was designed to be processed in both "forward" and "reverse" directions. Running in the forward direction, the tree takes a design at the leaves and produces a specification at the root. In the reverse direction, the proof starts with a specification and produces a design. This suggests that the proof tree could be used to do circuit design by analogy, although that is not the direction taken here. (See [Carbonell 83].) There are four major steps involved in this method of generalization:

Generalization Procedure

1. Generalize the specification.
2. Propagate the generalized specification through the tree.
3. Obtain constraints on the design at the leaves.
4. Apply problem independent constraints to the design.

The first step involves systematically removing information from the specification. This is done by changing constants appearing in the specification into variables. The result is shown in Figure 6. The specification now has three types of free parameters. The indices associated with the output wires have been generalized. The boolean control codes and the time values have also been changed into variables. Only on the right hand side of the equality have constants been generalized. The left side was left alone due to a requirement that all output lines have their behavior specified by some clause. After all four clauses have been generalized, the specifications can express arbitrary permutations of four bits. In fact, the specifications can express movement of data other than permutations. They can also express arbitrary time delays.

Some plausible generalizations have not been made. For example, it might be desirable to generalize the length of the shift register or the number of different operations the device can perform. These quantities do not appear explicitly in the specifications. They cannot be generalized using the technique of changing constants to variables. Generalizing these quantities would require a more complex representation for the specifications as well as a more sophisticated procedure for generalizing the specifications.

Description of Circuit Behavior

```
{This clause describes the behavior of Output[0].
Clauses describing the other outputs are analogous.}

(Equals OUTPUT[0].(T)
        (Case SELECT.(?TIME-01)
              ( ?VALUE-01  OUTPUT[?INDEX-01].(?TIME-02) )
              ( ?VALUE-02  OUTPUT[?INDEX-02].(?TIME-03) )
              ( ?VALUE-03  OUTPUT[?INDEX-03].(?TIME-04) )
              ( ?VALUE-04  LOAD[?INDEX-04].(?TIME-05) )))
```

Figure 8: Generalized Specifications

Once the specifications have been generalized, they can be propagated down through the proof tree. This is achieved by having a procedure for each rule which computes the "pre-conditions" for that rule. Given a "post-condition" on the result of a proof rule, the procedure finds "pre-conditions" on the antecedents of the rule which guarantee that the post-condition will be true. Each of the proof rules must be written in forward and backward versions. For example, the "Replace Rule" involves eliminating variables when running in the forward direction, and introducing variables when running in the backward direction. This method of backward constraint propagation has been applied in other learning systems such as [Utgoff 83; Minton 84], and the method is formalized in [Dijkstra 76]. After the specifications have propagated through the tree, constraints on the circuit design are obtained at each of the leaves.

The final step involves applying some problem-independent constraints to the design statements generated at the leaves of the proof tree. These constraints require that the circuit design meet some general requirements that apply to all designs. For instance, one constraint requires that no input wire be connected to more than one output from another device.

The Generalized Design

A portion of the final design schema is shown in Figure 7. One part of this schema is a list of constraints on the parameters of the generalized specifications. The schema in Figure 7 lists constraints on the time variables "?Time-01", "?Time-02", "?Time-03", etc. These constraints assert that the general design can only implement a one unit time delay. When these variables were first introduced, they allowed the specifications to express arbitrary time delays. Now it turns out that the original specifications were over generalized. The time values were constrained as they propagated through the proof tree. This is a consequence of

the fact that the proof tree does not represent a reasoning process sufficiently general for implementing arbitrary delays.

The schema in Figure 7 also lists statements describing connections between inputs of a multiplexer and outputs of d-flip-flops. These connections are not specified exactly. They depend on the parameters "?Index-01", "?Index-02", "?Index-03", "?Value-01", "?Value-02", and "?Value-03" which appear in the generalised specifications. (The expression "(Number ?Value-01)" represents the integer corresponding to the two bit vector "?Value-01".) These variables fell through the proof tree without having their values constrained. Therefore, the multiplexer inputs may be connected to any of the d-flip-flop outputs. These degrees of freedom allow the schema to implement an arbitrary permutation of the four bits, and an arbitrary choice of control codes. The "Index" parameters determine which permutations can be computed. The "Value" parameters determine the corresponding control codes. This design schema can also be used to implement data rearrangement operations other than permutations.

Wire Connections Between Components

```
(CONNECT MUX[0].OUTPUT  DFF[0].INPUT)

(CONNECT DFF[?INDEX-01].OUTPUT
         MUX[0].INPUT[(NUMBER ?VALUE-01)])

(CONNECT DFF[?INDEX-02].OUTPUT
         MUX[0].INPUT[(NUMBER ?VALUE-02)])

(CONNECT DFF[?INDEX-03].OUTPUT
         MUX[0].INPUT[(NUMBER ?VALUE-03)])
```

Constraints on Generalized Variables

```
?TIME-01 = T - 1
?TIME-02 = T - 1
?TIME-03 = T - 1
?TIME-04 = T - 1
?TIME-05 = T - 1
```

{Connections and constraints are shown for stage zero only.}

Figure 7: A Portion of the Generalized Design

Conclusion

It has been shown that domain knowledge can be used to enable an intelligent system to generalize from a single example. In the area of design problems, a proof of correctness is a useful vehicle for applying domain knowledge to the task of generalization. The proof enables the generalizer to capture a chain of reasoning used to understand the original design. The resulting generalization represents all designs which can be verified using the same proof of correctness.

Acknowledgement

This paper and the work it reports have benefited greatly from numerous discussions with Michael Lebowitz.

References

[Carbonell 83] Carbonell, J. G. Derivational analogy in problem solving and knowledge acquisition. Proceedings of the International Machine Learning Workshop, Champaign-Urbana, Illinois, 1983, pp. 12 - 18.

[DeJong 83] DeJong, G. F. Acquiring schemata through understanding and generalizing plans. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, 1983.

[Dijkstra 76] Dijkstra, E., W. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

[Kelly 82] Kelly, V., Steinberg, L., The CRITTER System: Analyzing Digital Circuits by Propogating Behaviors and Specifications. Proceedings of the Second National Conference on Artificial Intelligence, Pittsburgh, PA, 1982.

[Michalak! 80] Michalski, R. S. "Pattern recognition as rule-guided inductive inference." *IEEE Transactions on Pattern Analysis and Machine Intelligence S,* 4 (1980), 349 - 361.

[Minton 84] Minton, S. Constraint-Based Generalization. Proceedings of the Fourth National Conference on Artificial Intelligence, Austin, Texas, 1984.

[Mitchell 83a] Mitchell. T. M. Learning and problem solving. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, 1983.

[Mitchell 83bl Mitchell, T. M., et al. An Intelligent Aid for Circuit Reaesign. Proceedings of the Third National Conference on Artificial Intelligence, Washington, DC, 1983.

[Mostow 83a] Mostow, J. Operationalizing advice: A problem-solving model. Proceedings of the International Machine Learning Workshop, Champaign-Urbana, Illinois, 1983, pp. 110 - 116.

[Mostow 83b] Mostow, J. Program Transformation for VLSI. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, 1983.

[Salsberg 83] Salzberg, S. Generating Hypotheses to Explain Prediction Failures. Proceedings of the Third National Conference on Artificial Intelligence, Washington, DC, 1983.

(Silver 83] Silver, B., Learning Equation Solving Methods rom Worked Examples. Proceedings of the International Machine Learning Workshop, Champaign-Urbana, Illinois, 1983, pp. 99 - 104.

[Utgoff 83] Utgoff, P. E. Adjusting Bias in Concept earning. Proceedings of the International Machine Learning Workshop, Champaign-Urbana, Illinois, 1983.

[Winston 72] Winston, P. H. Learning structural descriptions from examples. In P. H. Winston, Ed., *The Psychology of Computer* Vision, McGraw-Hill, New York, 1972.

[Winston 83] Winston, P. H, Binford, T. O., Katz, B., wry, M. Learning Physical Descriptions from Functional Definitions, Examples, and Precedents. Proceedings of the Third National Conference on Artificial Intelligence, Washington, DC, 1983.