LEAP: A Learning Apprentice for VLSI Design[1]

Tom M Mitchell
Sridhar Mahadevan
Louis I Steinberg

AI/VLSI Project
Computer Science Department
Rutgers University
New Brunswick, NJ 08903

ABSTRACT

It is by now well-recognised that a major impediment to developing knowledge-based systems is the knowledge acquisition bottleneck the task of building up a complete enough and correct enough knowledge base to provide high-level performance. This paper proposes a new class of knowledge-based systems designed to address this knowledge-acquisition bottleneck by incorporating a learning component to acquire new knowledge through experience. In particular, we define *Learning Apprentice Systems* as the class of *interactive* knowledge-based consultants that directly assimilate new knowledge by observing and analyzing the problem solving steps contributed by their users through their *normal* use of the system. This paper describes a specific Learning Apprentice System, called LEAP, which is presently being developed in the domain of VLSI design We also discuss design issues for Learning Apprentice Systems more generally, as well as restrictions on the generality of our current approach

I Learning Apprentice Systems

It is by now well-recognised that a major impediment to developing knowledge-based systems is the knowledge acquisition bottleneck: the task of building up a complete enough and correct enough knowledge base to provide high-level performance. In an effort to reduce the cost and increase the level of performance of current know ledge-based systems, a number of researchers have developed semi-automated tools for aiding in the knowledge acquisition process. These tools include interactive aids to help pinpoint and correct weaknesses in existing sets of rules (e.g., |1, 2]), as well as aids for the acquisition of new rules (e.g., |S|). Others have studied the automated learning of rules from databases of stored cases, but with few exceptions (e.g., |4, 5]), work on machine learning has not yet led to useful knowledge acquisition tools.

This paper proposes a new class of knowledge-based consultant systems designed to overcome the knowledge acquisition bottleneck, by incorporating recently developed machine learning methods to automate the acquisition of new rules. In particular, we define *Learning Apprentice Systems* as the class of *interactive* knowledge-based consultants that directly assimilate new knowledge by observing and analyzing the problem solving steps contributed by their users through their *normal* use of the system. This paper discusses issues related to the development of such Learning Apprentice Systems, focusing on the design of a particular Learning Apprentice System (called LEAF*) for VLSI circuit design.

One key aspect of Learning Apprentice Systems as we define them is that they are designed to continually acquire new knowledge without an explicit "training mode". For example, the LEAP system provides advice on how to refine the design of a VLSI circuit, while allowing the user to override this advice and to manually refine the circuit when he so desires. In those cases where the user manually refines the circuit, LEAP records this problem solving step as a training example of some rule that it should have had. LEAP then generalizes from this example to form a new rule summarising this refinement tactic.

In task domains for which Learning Apprentice Systems are feasible, we expect that they will offer strong advantages over present architectures for knowledge-based systems. Many copies of n Learning Apprentice System distributed to a broad community of users could acquire a base of problem-solving experience very large compared to the experience from which a human expert learns For example, by distributing copies of LEAP to a thousand circuit designers, the system (collection) would quickly be exposed to a larger number of example circuit designs than a human designer could hope to see during a lifetime. Such a large experience base would offer the potential for acquiring a very strong knowledge base, provided effective learning methods can be developed.

The following section describes the design of the LEAP Learning Apprentice system for VLSI design, focusing on its mechanism for capturing training examples, and on its methods for generalizing from these examples to form new rules. The final section discusses some of the major choices made in the initial design of LEAP, limitations on the applicability of our initial approach, and several basic issues that we see as central to developing Learning Apprentice Systems in a variety of task domains.

II LEAP: A Learning Apprentice for VLSI Design

LEAP is currently being constructed as an augmentation to a knowledge-based VLSI design assistant called VEXED |6|. VEXED provides interactive aid to the user in implementing a circuit given its functional specifications, by suggesting and carrying out possible refinements to the design. A large part of its knowledge about circuit design is composed of a set of *implementation rules,* each of which suggests some legal method for refining a given function. For example, one implementation rule states that *IF the required function is to convert a parallel signal to a serial signal, THEN one possible implementation it to*

•This material is bated on work supported by the Defense Advanced Research Projects Agency under Research Contract N0O014-81-K-0394, and by the National Science Foundation under grant DCS83-51523. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation, or the U.S. Government.

use of shift registers. In this section, we will see how LEAP is designed to learn. This section describes the VEXED system, the type of training examples that it can capture from its users, and two generalization methods that allow LEAP to form general rules from these examples.

A. The VEXED Design Consultant

VEXED is a prototype knowledge-based design consultant that provides a convenient editor and user interface which helps the user design digital circuits beginning with their functional specifications and leading to their implementation. VEXED maintains an agenda of design subtasks (e.g., "implement the module that must multiply two numbers") which initially contains the top-level task of implementing the entire circuit. VEXED repeatedly selects a subtask from the agenda, examines its implementation rules to determine whether it can suggest possible implementations for the corresponding circuit module, then presents any such suggestions to the user. The user may select one of the suggested implementation rules, in which case that rule is executed to refine the module. Alternatively, the user may disregard VEXED's suggestions and instead use the editor to manually refine the circuit module. It is in this later case that LEAP will add to its knowledge of circuit design, by generalizing from the implementation step contributed by the user to formulate a new rule that summarizes a previously uncatalogued implementation method.

As an example of this kind of learning scenario, suppose that at some point during the design VEXED and the user are considering the task of implementing a particular circuit module. In the present example, this circuit module must compute the boolean product of sums of four particular input signals which appear in the context of the larger circuit. Assume further that these input signals are regular streams of boolean values arriving every 100 nanoseconds, remaining stable for approximately 70 nanoseconds, and encoded in positive logic**. Assume furthermore, that the stream of input values for Input1 is known to be an alternating stream of logical ones and zeros. The exact definitions of the function to be implemented and of the signals for which it must work are given in the top half of figure II-1***.

Function to be Implemented:
Inputs: Input1, Input2 Input3, Input4
Outputs: Output
Function: (Equals (Value Output(i))
(And (Or (Value Input1(i)) (Value Input2(i)))
(Or (Value Input3(i)) (Value Input4(i)))))

Where Input Signals Satisfy:
(Datatype Input1(i)) = Boolean
(Value Input1(i)) = i Mod 2
(Encoding Input1(i)) = Positive-Logic
(Start-Time Input1(i)) = i·100 + t_0
(Duration Input1(i)) = 75 nsec
(Datatype Input2(i)) = Boolean
(Value Input2(i)) = unknown
(Encoding Input2(i)) = Positive-Logic
(Start-Time Input2(i)) = i·100 + t_0
(Duration Input2(i)) = 65 nsec.
(Datatype Input3(i)) = Boolean
(Value Input3(i)) = unknown
(Encoding Input3(i)) = Positive-Logic
(Start-Time Input3(i)) = i·100 + t_0
(Duration Input3(i)) = 58 nsec.
(Datatype Input4(i)) = Boolean
(Value Input4(i)) = unknown
(Encoding Input4(i)) = Positive-Logic
(Start-Time Input4(i)) = i·100 + t_0
(Duration Input4(i)) = 75 nsec.

User's Solution:

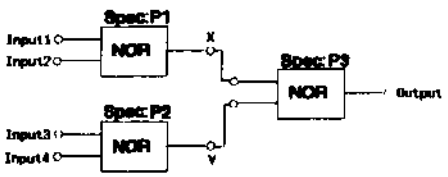


Figure II-1: A Training Example for LEAP

Given this information about the module to be implemented, the system searches its set of implementation rules for advice regarding possible refinements of this circuit. In this case, the system may have a rule that suggests implementing the circuit module using an AND gate and two OR gates. Suppose, however, that the user disregards the advice of the system in this case, choosing instead to implement the module using the circuit shown in figure II-1. This implementation contributed by the user provides the system with precisely the kind of training example that LEAP needs for learning a new implementation rule. In general, then, each training example consists of (1) a description of the function to be implemented, (2) a description of the known characteristics of the input signals, and (3) a circuit entered by the user to implement the given function for the given input signals.****

Given such a training example, there are two kinds of changes that one might expect the system to make to its knowledge base. First, LEAP has the opportunity to acquire a new implementation rule that can be used in subsequent cases to suggest the user's NOR-gate circuit where it is a *potsiblc* implementation. Second, the system also has an opportunity to learn a fragment of control knowledge for selecting between the NOR-gate implementation and the previously known AND-OR gate implementation, depending on which is *preferred* according to some cost criterion. In VEXED, we have cleanly separated out these two kinds of knowledge. Implementation rules characterise only the *possible* correct implementations, while a separate body of control knowledge will be used to select the

**That is, a logical one is encoded as five volts, and a logical sero ai sero volts.

***Signals, or "datastreams" In VEXED are described as an array of data elements, each defined in terms of its Value, Start-Time, Duration, Datatype, and Encoding.

****Although in this example the user's circuit has been refined down to the gate level, in general it need only be one step more refined than the sub-module it is implementing.

preferred implementation of the user's circuit, or possible alternatives. In our work to date and in this paper we consider learning only of new implementation rules that characterize the general conditions under which the user's circuit can be correctly used

IF the Function to be Implemented is of the form:
Inputs: Input1, Input2, Input3, Input4
Outputs: Output
Function: (Equals (Value Output(i))
(And (Or (Value Input1(i)) (Value Input2(i)))
(Or (Value Input3(i)) (Value Input4(i)))))

Where Input Signals Satisfy:
((Datatype Input1(i) = Boolean)
((Encoding Input1(i)) = Positive-Logic)
((Datatype Input2(i) = Boolean)
((Encoding Input2(i)) = Positive-Logic)
((Datatype Input3(i) = Boolean)
((Encoding Input3(i)) = Positive-Logic)
((Datatype Input4(i) = Boolean)
((Encoding Input4(i)) = Positive-Logic)
(Length (Intersection (Interval Input1(i))
(Interval Input2(i))
(Interval Input3(i))
(Interval Input4(i)))) > 3 nsec

THEN one possible implementation is:

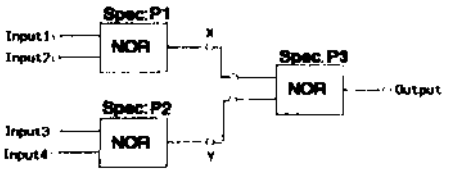
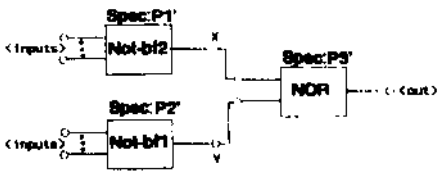


Figure II-2: Inferred Rule with Generalised Left-Hand Side

Given this training example, the most straightforward method of acquiring a new implementation rule is to create a rule that suggests the given circuit, can be used to implement the given module function in precisely this context (e.g., whenever the input signals are precisely the same as in the training example). Such a rule would clearly be so specific that it would add little of general use to the system's knowledge of implementation methods. A better approach would be to generalise the preconditions (left hand side) of the implementation rule, so that it characterises the general class of input signals for which the given circuit correctly implements the specified function. Such a generalised rule is shown in figure II-2 and the method for producing such generalizations in LEAP is described in the following subsection. A further step in generalizing the implementation rule would be to generalize the user's circuit as well as the function it implements. (e.g., the essential idea behind the NOR-gate implementation can be used to implement a class of functions related to the one encountered in this training example). Such a generalization of the implementation rule is shown in figure II-3, and the method used by LEAP for generalizing the rule in this fashion is described in subsection 2.3

IF the Function to be Implemented is of the form:
Inputs: <inputs>
Outputs: <out>
Function: (Equals (Value <out>(i))
(And <bool-fn2> <bool-fn1>))

THEN one possible implementation is:



With Specifications of the three modules as follows:

P_1': (Equals (Value X(i)) (Not <bool-fn2>))

P_2': (Equals (Value Y(i)) (Not <bool-fn1>))

P_3': (Equals (Value <out>(i)) (Not (Or X(i) Y(i))))

Figure II-3: Inferred Rule with Generalized Right-Hand Side

B. Generalizing the Rule Left-Hand Side

LEAP computes a justifiably general rule precondition by using its theory of digital circuits to analyze the single training example. In particular, LEAP first explains (verifies) for itself that the circuit does in fact work for the example input signals, then generalizes from this example by retaining only those features of the signals that were mentioned in this explanation. It is this set of signal features that is required for the explanation to hold in general, and which therefore characterizes the class of input signals for which the circuit will correctly implement the desired function. This explain-then-generalise method for producing justifiable generalizations from single examples is based on our previous work on goal-directed generalization [7|, and is also similar to the generalization methods employed in [8, 9, 10].

To illustrate this generalisation method, consider again the training example introduced above in figure II-1. LEAP begins by verifying that the example circuit will operate correctly for the example input signals. In order to do this, it examines its definitions of the primitive components that make up the example circuit. Figure 2-4 shows the description of the primitive NOR gate used in the present example circuit. The *Operating Conditions* in this description summarise characteristics of the input signals that are required for the component to have a well defined output. For example, the constraint "*(Length (Intersection (Interval Input1(i)) (Interval Inputt(i)))) > S nsec.*" indicates that for the NOR gate to operate correctly, its inputs

```
inputs  Input1. Input2
Outputs  Output

Operating Conditions
  (Equals (Datatype InputI(i)) Boolean)
  (Equals (Encoding Input1(i)) Positive-Logic)
  (Equals (Datatype Inpui2(i)) Boolean)
  (Equals (Encoding Input2(i)) Positive-Logic)
  (Length (Intersection (Interval InputI(i))
                        (Interval Input2(i)))) > S nsec.

Mapping.
  (Equals (Value Output(i))
          (Not (Or (Value InputI(i)) (Value Input2(i)))))
  (Equals (Encoding Output(i)) Positive-Logic)
  (Equals (Start-Time Output(i))
          (+ 10 (Latest (Start-Time InputI(i))
                        (Start-Time Input2(i)))))
  (Equals (Duration Output(i))
          (Length (Intersection (Interval InputI(i))
          (Interval Input2(i)))))
```

The *Operating Conditions* describe minimum requirements on input signals to assure the component will produce a well-defined output. The *Mapping* describes how features of the output signal depend on the inputs.

Figure II-4: Known Behavior of a NOR Gate.

must overlap in time by at least 3 nanoseconds*****.

These Operating Conditions of the individual circuit components are constraints that must be verified for the example circuit and the given input signals. Some of these operating conditions can be tested directly against the descriptions of the global circuit inputs (e.g., the operating conditions for the left-most NOR gates in the example circuit can be tested against the known characteristics of the circuit inputs). The operating conditions associated with components internal to the example circuit must be restated in terms of the equivalent constraints on the global circuit inputs. These constraints are therefore *propagated* to (reexpresaed in terms of) the global inputs of the circuit network, then tested to see that they are satisfied by the example input signals. For instance, the constraint *"(Length (Intersection (Interval X(i)) (Interval Y(i)))) > S nsec.** which follows from the operating conditions of the right-most NOR gate, is reexpresaed in terms of the four global circuit inputs to produce the equivalent constraint *"(Length (Intersection (Interval Input1(i))(Interval Input2(i))(Interval Input3 (i))(Interval Input4(i)))) > S nsec.*,"****** By propagating the constraints arising from the operating conditions of the circuit components, as well aa the original constraint on the circuit output (e.g., that

*****The *Initial* of a data element is defined here as the time interval beginning at the *Start Time* of the data element, and continuing for the *Duration* of that data element.

******This constraint propagation step is performed in the VEXED system by a set of routines called CRITTER [11] which is able to propagate and check signal constraints in loop-free digital circuits, by examining the function definitions of the primitive circuit elements.

it produce the . .. i -UP ..I ,.. i » I ih> inputs) | | \|' . „, venfy that the user-imr«>du<rd U P U I I will correctly implement the desired function for the given inputs More important)) the constraints that are propagated to the inputs of the circuit network characterize precisely the class of inputs for which the circuit will operate correctly, and therefore constitute the desired general preconditions for the newly acquired implementation rule.

In summary, the procedure for computing the generalized preconditions for the new rule is to (1) propagate each constraint derived from the operating conditions of each primitive circuit component, along with constraints on the global circuit output, back to the global inputs to the circuit network, then to (2) record the resulting constraints on the global inputs, with appropriate substitution of variable names, as the generalized preconditions for the new implementation rule. Figure II-2 illustrates the resulting generalization for the training example from figure II-1. Notice that in comparing this generalized rule with the original training example, values of several features of the circuit inputs have been generalized Only the constraints on Datatype and on Signal-Encodings remain intact, while the detailed values for the signal Start-Times and Durations have been replaced by the general constraint on overlapping time intervals.

C. Generalising Rule Right-Hand Side

The previous section describes how LEAP is able to generalize the left-hand side (LHS) of the rule by determining the class of input signals for which the given circuit will work This section describes how LEAP can also generalize the right-hand side (RHS) of the rule; that is, generalize the circuit schematic along with the functional specifications to be implemented

The key to generalizing the RHS is to first verify that the circuit correctly implements the desired function. This verification can then be examined to determine the general class of circuits and functional specifications to which the same verification steps will apply. This method, which we call Verification-Based Learning, is described more generally in [12|. That paper discusses the general applicability of this method to learning problem-decomposition rules, or planning schema. Here we discuss the application of this method to generalizing circuit implementation rules, and illustrate the method using the training example and rule discussed above.

1. Step 1: Forming the Composed Specification from Rule RHS

The first step in the process of inferring a general circuit design rule from a training example is that of verification: ensuring that the function computed by the user's circuit meets the original circuit specification.

We can derive a description of the circuit's function from its structure by composing the functions of the submodules constituting the circuit, according to the configuration in which they are interconnected. For the user's NOR-gate circuit, this *composed specification* is given as

```
(EQUALS (VALUE Output(i))
        (NOT (OR (NOT (OR (VALUE Input1(i))
                          (VALUE Input2(i))))
                 (NOT (OR (VALUE Input3(i))
                          (VALUE Input4(i))))
)))
```

Note that, in general, the composed specification will be a *syntactically reexpressed* version of the original specification. For example, the above composed specification is not syntactically identical to the functional specifications* in the training example, even though it does represent the same boolean function. This frequently occurs in VLSI circuits in which, for example, functional specifications in terms of AND and OR boolean expressions are often implemented in terms of NAND and NOR gates.

2. Step 2: Verifying the Circuit Function

To verify the correctness of the user-suggested NOR-gate circuit, LEAP must show the equivalence between the *composed specification* for this circuit and the *original specification* of the circuit being implemented. Thus, it seeks to verify that

```
(IMPLIES <composed-spec> <original-spec>)
```

or in this case

```
(IMPLIES
    (NOT (OR (NOT (OR (VALUE Input1(i))
                      (VALUE Input2(i))))
             (NOT (OR (VALUE Input3(i))
                      (VALUE Input4(i))))))

    (AND (OR (VALUE Input1(i))
             (VALUE Input2(i)))
         (OR (VALUE Input3(i))
             (VALUE Input4(i)))))
```

LEAP verifies that the composed specification meets the original specification by determining a sequence of algebraic transformations which, when applied to the composed specification, will yield the original specification. Each transform has a *precondition* which describes the class of situations to which it can be applied, and a *postcondition* which specifies the result of the transformation. The two transforms that will be used for the current example in the circuit domain are given below.

De-Morgan's Law

Precondition:
`(NOT (OR <bool-fn1> <bool-fn2>))`

Postcondition:
`(AND (NOT <bool-fn1>) (NOT <bool-fn2>))`

Remove-Double-Negation:

Precondition:
`(NOT (NOT <bool-fn>))`

Postcondition:
`<bool-fn>`

Here "<bool-fn>" represents an arbitrary boolean function. Shown below is the verification as a sequence of transformations.

VERIFICATION

```
(NOT (OR (NOT (OR (VALUE Input1(i))
                  (VALUE Input2(i))))
         (NOT (OR (VALUE Input3(i))
                  (VALUE Input4(i))))))
                  ↓
              De-Morgan
                  ↓
(AND (NOT (NOT (OR (VALUE Input1(i))
                   (VALUE Input2(i)))))
     (NOT (NOT (OR (VALUE Input3(i))
                   (VALUE Input4(i))))))
                  ↓
          Remove-Double-Negation
                  ↓
(AND (OR (VALUE Input1(i))
         (VALUE Input2(i)))
     (NOT (NOT (OR (VALUE Input3(i))
                   (VALUE Input4(i))))))
                  ↓
          Remove-Double-Negation
                  ↓
(AND (OR (VALUE Input1(i))
         (VALUE Input2(i)))
     (OR (VALUE Input3(i))
         (VALUE Input4(i))))
```

3. Step 3: Determining the Generalised Composed Specification

Given the verification tree shown above, the next step is to determine the genera) class of expressions for which this sequence of verification steps will correctly apply. This is essentially a problem of viewing the transformation sequence as a

[illegible] operator sequence to determine the necessary pre-conditions for the operator sequence. LEAP accomplishes this by back-propagating the precondition of each transform in the sequence, to determine the necessary conditions on the starting expression. This process is described in greater detail in [12]. The sequence shown below illustrates this back-propagation, and indicates the resulting generalisation of the composed specification.

COMPUTING THE GENERALIZED COMPOSED SPECIFICATION

```
(AND (OR (VALUE Input1(i))
         (VALUE Input2(i)))
     (OR (VALUE Input3(i))
         (VALUE Input4(i))))
              ↓
      Remove-Double-Neg
              ↓
(AND (OR (VALUE Input1(i))
         (VALUE Input2(i)))
     (NOT (NOT <bool-fn1>)))
              ↓
      Remove-Double-Neg
              ↓
(AND (NOT (NOT <bool-fn2>))
     (NOT (NOT <bool-fn1>)))
              ↓
         De-Morgan
              ↓
(NOT (OR (NOT <bool-fn2>)
         (NOT <bool-fn1>)))
```

Notice that the final expression in the above sequence describes the generalised composed specification for which the verification will correctly apply. From it, we see that the important feature of the two submodule specifications P_1 and P_2 (the two leftmost NOR gates in figure II-1) is that they both compute the *negation of some boolean function,* while the specifications of the third component cannot be generalised.

GENERALIZED SPECIFICATIONS OF SUBMODULES

```
P1': (EQUALS (VALUE X(i)) (NOT <bool-fn2>))
P2': (EQUALS (VALUE Y(i)) (NOT <bool-fn1>))
P3': (EQUALS (VALUE <out>(i))
             (NOT (OR X(i) Y(i))))
```

4. Step 4: Determining the Generalised Original Specification

Having determined the generalised specifications of the circuit submodules, the RHS of the new rule can now be formed. However, LEAP must also produce a corresponding generalisation of the original functional specification in the rule LHS. This generalised original specification can be computed in a relatively straightforward manner, either by reapplying the sequence of verification transforms to the generalised composed specification or by using the variable bindings generated when computing the generalized composed specification. Following either of these two approaches, the result is that the new original specification becomes

```
(EQUALS (VALUE Output(i))
        (AND <bool-fn2> <bool-fn1>))
```

Comparing the generalized original specification above with the original specification of the circuit implementation in figure 11-1, it is seen that a generalisation of the original specification has been achieved from a conjunction of disjunctions to a *conjunction of any boolean functions.*

5. Step 5: Forming the New Implementation Rule

We have shown in the last few paragraphs how the original specification of a circuit module as well as the functional specifications of each of the submodules P. in its implementation could be generalized. The final step is to form the new implementation rule which is based on these generalized specifications. The preconditions for this new rule are formulated to require (1) that the function to be implemented match the generalised original specification, and (2) that the input signals satisfy the constraints that are determined as shown in the previous subsection. The right-hand side of the new rule is formulated so that it produces the submodules with their corresponding submodule specifications P_i. For the present example, the new implementation rule formed in this fashion******* is shown in figure 11-3.

III Discussion

The previous section describes in some detail how LEAP captures training examples from its users, and how it forms genera] rules from these examples. This section discusses more broadly the architectural issues involved in designing knowledge-based systems that can incorporate such learning methods. In particular, we discuss the major design features of LEAP that appear important to the design of Learning Apprentice Systems more generally. Three design features that have a major impact on the capabilities of LEAP are: (1) the interactive nature of the problem solving system, (2) the use of analytic methods for generalizing from examples, and (3) the separation of knowledge about when an implementation technique *can* be used from knowledge about when it *should* be used.

A. Interactive Nature of the Apprentice Consultant

A fundamental feature of LEAP is that it embeds a learning component within an *interactive* problem-solving consultant. This allows it to collect training examples that are closely suited to refining its rule base. In particular, training examples collected by a Learning Apprentice "have two attractive properties:

1. Training examples focus only on knowledge that is missing from the system. The need for the user to intervene in problem solving occurs only when the system is missing knowledge relevant to the task at hand, and the resulting training examples therefore focus specifically on this missing knowledge.

••"•••Notice that in this rule, there are no final constraints that must be satisfied by the input signals. This is because the left-most circuit modules in the figure are defined so abstractly, that they pose no constraints on the signal formats of their inputs.

2 The training examples are single problem solving steps This is in contrast to the type of training examples used by other rule learning systems such as Meta-DENDRAL 4 and INDUCE-PLANT [5], in which training examples are complete problem solutions By working with training examples that are single steps, LEAP circumvents many difficult issues of credit assignment that arise in cases where the training example corresponds to a chain of several rules.

While to first order, LEAP acquires training examples that correspond to single rule inferences, this is only approximately true We expect that LEAP will encounter training examples in which its existing rules will correspond to finer-grained decisions than the user thinks of as a single step For instance, the system may have a sequence of rules to implement a serial-parallel converter by first selecting a shift register, then a general class of shift registers (e.g dynamic), and only then a specific circuit, while the user may think of the whole series of decisions as a single step, implementing the converter with a specific circuit.

In such cases, LEAP could just go ahead and learn the larger-grained rule that will follow from the user's training example, but doing so could cause a number of problems One problem is that it will result in a rule set with rules of greatly varying grain. Such inconsistency in grain is likely to lead to redundancy and lack of generality in the rules. A second potential problem associated with large-grain training examples is that our analytical methods of generalization may be too expensive to use on steps of large grain. Since the methods depend on constructing a verification of the step, there is reason to fear the cost may grow very quickly as the size of the step gets large compared to the size of the transformations used in the verification process.

Thus, the question of how to handle grain size mismatch may be an important issue for future research One possible direction would be to develop methods for examining a training example that corresponds to a large step, then determining which existing rules correspond to parts of this inference step, leaving only the task of acquiring the missing finer grain rules

B. Use of Analytical Methods for Generalization

A second significant feature of the design of LEAP is that it uses analytical methods to form general rules from specific training examples, rather than more traditional empirical, data-intensive methods. LEAP's explain-then-generalize method, based on having an initial domain theory for constructing the explanation of the example, allows LEAP to produce justifiable generalizations from single training examples. This capability is particularly important for LEAP since it is not at all clear how LEAP could tell that two different training examples involving different circuit specifications and different resulting circuits, were in fact two examples of the same rule.

One significant advantage of the analytical methods involves learning in the presence of error-prone training data. An issue that seems central to research on Learning Apprentice Systems, and one that LEAP must confront immediately, is that the users who (unwittingly) supply its training examples are likely to make mistakes. In particular, since we hope to first introduce LEAP to a user community of university students who are themselves learning about VLSI design, the issue of dealing with error-prone examples is a major one. Our initial plan for dealing with this problem is to have LEAP learn general rules only from those training example circuits that it can verify in terms of its knowledge of circuits Since its generalization method requires that it explain an example circuit before it can generalize it, LEAP will be a very conservative learner Since it will be unable to verify incorrect circuit examples that it encounters, there is little danger of it learning from incorrect examples******** This method of dealing with errorful data is attractive, but may be insufficient if we need to include empirical learning methods along with analytical methods for generalisation

While analytical generalization methods offer a number of advantages, they require that the system begin with a domain theory that it can use to explain/validate the training examples This requirement, then, constrains the kind of domain for which our approach can be used In the domain of digital circuit design, the required domain theory corresponds to a theory for verifying the correctness of circuits In certain other domains, such a theory may be difficult to come by For example, in domains such as medical diagnosis the underlying theory to explain/verify an inference relating symptoms to diseases is often unknown even to the domain experts. In such domains, the system would lack a domain theory to guide the analytical generalization methods, and would have to rely instead on empirical generalization methods that generalize by searching for similarities among a large number of training examples. In fact, our present methods for utilizing domain theories to guide generalization are limited to cases where there is a strong enough theory to "prove" the training example is correct. One important research problem is thus to develop methods for utilising more approximate, incomplete domain theories to guide generalisation, and for combining analytical and empirical generalization methods in such cases. One new research project that is interesting in this light is an attempt to construct a Learning Apprentice for well-log interpretation |13|. In this domain, the underlying theory necessary to learn new rules involves geology and response of well-logging tools. Since these theories as inherently approximate and incomplete, that research project must face the issue of generating and utilizing approximate explanations of training examples to infer general rules.

C. Partitioning of Control and Basic Domain Knowledge

A third significant feature in the design of LEAP is the partitioning of its knowledge base into (1) implementation rules that characterize *correct* (though not necessarily preferred) circuit implementations, and (2) control knowledge for selecting the *preferred* implementation from among multiple legal options This partitioning is important because it helps in dealing with the common problem that when one adds a new rule to a knowledge base one must often adjust existing rules as well.

The first of these two parts of the knowledge base has the convenient property that its rules are logically independent; that is, when one adds a new implementation rule characterising a new implementation method, it does not alter in any way the correctness of the existing implementation rules. Thus, when a new implementation rule is added, the only portion of the knowledge base that might require an update is the control knowledge for selecting among alternative implementations. This

••••••"Even this it not quite true. Since its domain theory is only approximate (at will probably be true for Learning Apprentice Systems in general), there may be incorrect circuits that it succeeds in verifying (say, because is overlooks parasitic capacitances).

logical independence of implementation rules is also important when combining sets of rules that may have been learned from various users by different copies of the Learning Apprentice While the problem of combining multiple rule sets learned from different sources is in principle simply a matter of forming the union of the rule sets, in fact the resulting set of correct rules may be overly redundant and disorganized. Thus, we anticipate that we may have to develop methods for merging and reorganizing sets of correct rules to make them more manageable.

To date, we have only considered learning the first type of knowledge. In some sense, learning these rules is easier than learning the control knowledge, because the complexity of explaining a training example is much less for implementation rules than for control rules. To explain/verify an example of an implementation rule, the system need only verify the correctness of the circuit fragment mentioned in the training example. However, to learn a control rule that characterises when some implementation is *preferred,* it is necessary to compare this implementation with all the alternative possibilities. Thus, the complexity of constructing the explanations is quite different in these two cases. In the longer term, we see learning of control knowledge as an important task for LEAP, and a task for which it can easily capture useful training examples.

IV Conclusion

We have presented the notion of a Learning Apprentice System as a framework for automatically acquiring new knowledge in the context of an interactive knowledge-based consultant The initial design of a Learning Apprentice for VLSI design has been described. In particular, we have detailed the methods that LEAP employs for learning new implementation rules, and for generalizing both the left and right hand side of these rules. Whereas previous attempts at automatic knowledge acquisition have met with little success, the proposed Learning Apprentice System differs in two important respects. it utilizes more powerful analytical learning methods, and it is restricted to interactive knowledge-based systems which can easily capture useful training examples. We are currently completing our initial implementation of LEAP, and intend to test it on a user community of students in a VLSI design course to gather data and further insights on this initial design.

V Acknowledgments

We thank several people who provided useful criticisms of earlier drafts of this paper: Rich Keller, Yves Kodratoff, John McDermott, Jack Mostow, Reid Smith, and Timothy Weinrich. We also thank the members of the Rutgers AI/VLSI project for many useful discussions regarding the design of LEAP, and for creating the VEXED system on top of which LEAP is being constructed. Schlumberger-Doll Research has made the STROBE system available as a representation framework in which VEXED and LEAP are being implemented. This research is supported by the Defense Advanced Research Projects Agency under Research Contract N00014-81-K-0394, and by the National Science Foundation under grant DCS83-51523.

References

1 Davis. R "Applications of meta level knowledge to the construction and use of large knowledge bases/ in *Knowledge-based Systems in Artificial Intelligence,* Davis, R. and Lenat, D., eds , McGraw-Hill, New York, 1981

[2] Pohtakis, P., *Using Empirical Analysis to Refine Expert System Knowledge Bases,* PhD dissertation, Rutgers University, August 1982.

[3] Kahn, G., Nowlan, S., and McDermott, J "A Foundation for Knowledge Acquisition " In *Proceedings of the IEEE Workshop of Principles of Knowledge-Based Systems* IEEE, December, 1984, 89-96.

|4] Buchanan, B. G. and Mitchell, T M., "Model-directed learning of production rules," in *Pattern-Directed Inference Systems,* Waterman, D. A and Hayes-Roth, F, eds., Academic Press, New York, 1978.

|5] Michalski, R S. and Chilausky R L. "Knowledge Acquisition by Encoding Expert Rules Versus Computer Induction from Examples. A Case Study using Soybean Pathology." *Intl. Jrnl for Man-Machine Studies.* 12 63 (1980) .

[6] Mitchell, T.M., Steinberg, L.I., and Shulman, J.S. "A Knowledge-Based Approach to Design." In *Proceedings of the IEEE Workshop of Principles of Knowledge-Based Systems.* IEEE, December, 1984, 27-34, Revised version to appear in *IEEE Transactions on Pattern Analysis and Machine Intelligence,* September, 1985.

|7| Mitchell, T. "Learning and Problem-Solving." In *IJCA1-8S.* August, 1983, 1139-1151.

[8] DeJong G. "Automatic Schema Acquisition in a Natural Language Environment." In *Second National Conference on Artificial Intelligence.* Pittsburgh, PA, August, 1982, 410-413.

[9] Salzberg, S. and Atkinson, D.J. "Learning by Building Causal Explanations." In *ECAI-84* September, 1984, 497-500.

[10] Minton, S. "Constraint-Based Generalization." In *AAAI-84.* Austin, Texas, August, 1984, 251-254

|11] Kelly, Van E. The CRITTER System - Automated Critiquing of Digital Circuit Designs." In *Proceedings of the 21st Design Automation Conference.* IEEE, June, 1984, 419-425, Also Rutgers AI/VLS1 Project Working Paper No. 13

[12] Mahadevan, S. "Verification-Based Learning: A Generalization Strategy for Inferring Problem-Decomposition Methods." In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.* August, 1985.

[13] Smith, R.G., Winston, H.A., Mitchell, T.M., and Buchanan, B.G. "Representation, Use and Generation of Explicit Justifications for Knowledge Base Refinement." In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.* August, 1985.