Parsing to Learn Fine Grained Rules*

Subrata Roy and **Jack Mostow**
Rutgers University Computer Science Department
New Brunswick, NJ 08903, USA
ARPAnet address: suroy@paul.rutgers.edu, Mostow@aramis.rutgers.edu

Abstract

The grain size of rules acquired by explanation-based learning may vary widely depending on the size of the training examples. Such variation can cause redundancy among the learned rules and limit their range of applicability. In this paper, we study this problem in the context of LEAP, the "learning apprentice" component of the VEXED circuit design system. LEAP acquires circuit design rules by analyzing and generalizing design steps performed by the user. We show how to reduce the grain size of rules learned by LEAP by using *"synthetic parsing"* to extract parts of the manual design step not covered by existing design rules and then using LEAP to generalize the extracted parts. A prototype implementation of this technique yields finer grained rules with more coverage. We examine its effects on some problems associated with the explanation-based learning technique used in LEAP.

1 Introduction

A knowledge-based approach to design, as explored by many researchers (e.g. [Mitchell *et al.*, 1985b; Kowalski and Thomas, 1983]), requires explicit representation of various kinds of design knowledge. In order to build such a system, one needs to decide on the granularity or the level of detail at which the design knowledge is represented. Ideally the grain size should be coarse enough to allow efficient reasoning and yet fine enough to make important distinctions. The problem of deciding proper grain size manifests itself in different forms depending on the method of acquiring design knowledge. In this paper we investigate the grain size problem for the case in which knowledge is acquired by explanation-based learning [Mitchell *et al.*, 1986; Dejong and Mooney, 1986] from examples supplied by the user.

Engineering design is well-suited for EBL as many design domains possess a detailed theory about the behavior of the structural components and their combinations. Note that this knowledge is more suitable for analyzing the performance of a given design rather than synthesizing a design from primitive structural components. Hence it would be useful for a design automation system to use EBL to learn knowledge suitable for synthesizing artifacts. Variants of EBL have been used to learn circuit design rules in [Mitchell *et al.*, 1985a; Ellman, 1985].

In this paper we address the problem of the large grain size of rules learned by EBL. We demonstrate the use of parsing as an approach to the problem and identify its promises and limitations. Our vehicle of experimentation is LEAP [Mitchell *et al.*, 1985a], the learning apprentice component of VEXED [Mitchell *et al.*, 1985b; Steinberg, 1987].

To begin with, we give a brief description of VEXED and LEAP in the following section. The example introduced to illustrate the operation of LEAP and VEXED is used throughout the paper. Section 3 describes the nature of the grain size problem and Section 4 illustrates our technique with a circuit design example. Section 5 analyzes our approach.

2 VEXED and LEAP

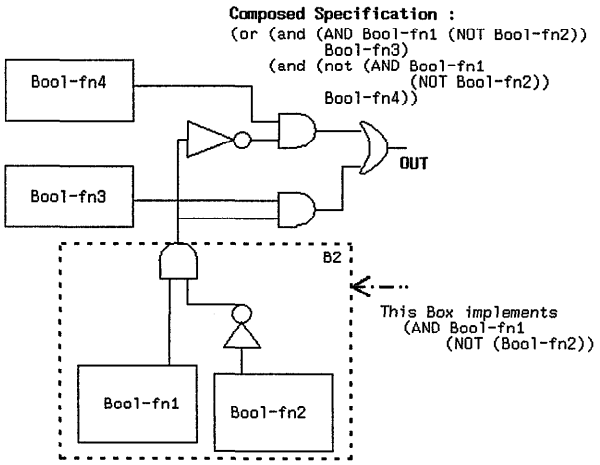
VEXED is an interactive knowledge-based design aid for VLSI circuits. It embodies a model of design based on "top-down refinement plus constraint propagation" [Steinberg, 1987]. The knowledge for synthesizing circuits based on their functional specifications, called synthesis knowledge, is embodied in a set of refinement rules. VEXED also has analysis knowledge for verifying the correctness of a circuit implementation. A refinement rule is used to refine an unimplemented circuit specification (a *generic module*) into a set of primitive components and generic modules. The following is an English paraphrase of a typical refinement rule.

Rule1:
If the specification of module output has the form
(IF (Bool-fn1 > Bool-fn2) THEN Bool-fn3
ELSE Bool-fn4)
Then refine it to the circuit in Figure 1.

The refinement rule essentially decomposes the original problem into subproblems represented by generic modules. The interactions between subproblems are taken care of by symbolic constraint propagation (more details are available in [Mitchell *et al.*, 1985b]). The design is considered to be completed when all generic modules have been refined into primitive circuit components.

At each intermediate stage of the design the system suggests a set of applicable refinement rules and applies the

*This work is supported by NSF under Grant Number DMC-86-10507, by Rutgers CAIP, and by DARPA under Contract Numbers N00014-81-K-0394 and N00014-85-K-0116. The opinions expressed in this paper are the authors' and do not represent the policies, either expressed or implied, of any granting agency.
We are grateful to the other members of the Rutgers AI/Design Project for the stimulating and helpful climate in which this work was conceived.



Identifiers of the form Bool-fn1 represent any boolean function.

Figure 1: Result of a refinement rule

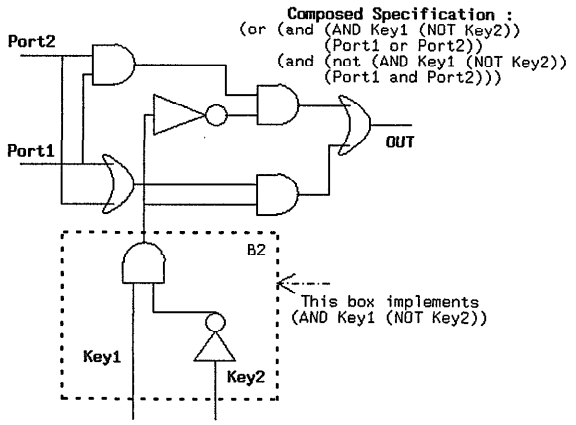


Figure 2: A manual refinement

one chosen by the user. The user may alternatively choose to refine the circuit by hand. In this case LEAP, the learning apprentice component of VEXED, generalizes the manual refinement step into a new refinement rule. LEAP uses a variant of explanation-based learning (called *verification based learning* [Mitchell *et al.*, 1985a]). The example manual refinement is "explained" by constructing a proof for the correctness of the circuit using analysis knowledge. The proof is then generalized to construct a general refinement rule which adds to the synthesis knowledge.

To illustrate, consider the task of implementing a modified 2-to-1 multiplexer controlled by the condition (Key1 > Key2). Key1 and Key2 are 1-bit binary values. Hence (Key1 > Key2) is equivalent to (AND Key1 (NOT Key2)). Depending on the value of (Key1 > Key2) either (Port1 OR Port2) or (Port1 AND Port2) is connected to the output. Hence the output specification for the multiplexer is

Spec1: IF (Key1 > Key2) THEN (Port1 OR Port2)
ELSE (Port1 AND Port2)

Suppose the user chooses to manually refine the module into the circuit in Figure 2.

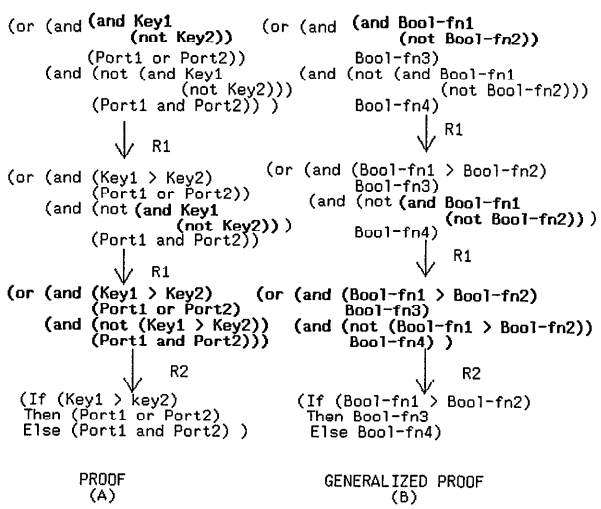
LEAP explains the manual refinement in Figure 2 by proving that the composed specification constructed by "symbolically executing" the circuit on its inputs is equivalent to the desired specification **Spec1**. The proof (Figure 4(A)) uses the rules of equivalence of boolean expressions in Figure 3. These rules are expressed as *rewrite rules* in which the *precondition* expression can be replaced by the *postcondition* expression.

The proof in Figure 4(A) is then generalized by abstracting away details not tested by the preconditions of the rule. The generalized proof tree is shown in Figure 4(B). The learned rule extracted from the generalized proof is **Rule1**, presented earlier as an example of a refinement rule. Next time a similar circuit is being designed, VEXED would suggest this newly learned rule.

R1:
PRE = (AND Bool-fn1 (NOT (Bool-fn2)))
POST = (Bool-fn1 > Bool-fn2)

R2:
PRE = (OR (AND Bool-fn1 Bool-fn2)
(AND (NOT Bool-fn1) Bool-fn3))
POST = (IF Bool-fn1 THEN Bool-fn2
ELSE Bool-fn3)

Figure 3: Verification rules



The rules apply to the boldfaced parts of the expression.

Figure 4: Proofs constructed by LEAP

3 Grain size problem in LEAP

The learning architecture of LEAP causes a problem which limits the usefulness of learned rules in VEXED. This has been identified as the grain size problem in [Mitchell *et al.*, 1985a]. LEAP always learns a single refinement rule from a manual refinement. However, the learned rule may actually be composed of several finer grained rules. For example, **Rule1** can be considered to be composed of

Rule2:
If the specification of the module output has the form
(IF Bool-fn1 THEN Bool-fn2
ELSE Bool-fn3)
Then refine it to the circuit in figure 5.
and
Rule3:
If the specification of the module output has the form
(Bool-fn1 > Bool-fn2)
Then refine it to the circuit in box B2 in Figure 1.

A refinement rule em r is considered to be of larger grain size than another refinement rule $\acute{r}$, if we can construct a tree of rules t containing $\acute{r}$ such that application of t produces the same result as a single application of r. For example, **Rule2** followed by **Rule3** will produce the same result as a single application of **Rule1**. Hence **Rule1** is of larger grain size than both **Rule2** and **Rule3**. Due to the varying size of the manual refinements, the rules learned by LEAP will be of varying grain size. This leads to the following problems limiting the usefulness of the learned rules in VEXED, the performance system.

1. **Less coverage:** Firstly, a large grained rule is not applicable in many situations, even though it contains all the relevant information. For example, consider the specification:

   Spec2: IF (Key1 = Key2) THEN (Port1 OR Port2)
   ELSE (Port1 AND Port2)

   Rule1 does not apply to **Spec2**, even though it is almost the same as **Spec1** and could be implemented simply by changing the components in box B2 in Figure 2. Secondly, larger grained rules may not be able to produce alternative designs. Alternative implementations of the specification **(Bool-fn1 > Bool-fn2)** will not get used if **Rule1** is the only rule available for refining **Spec1**.
2. **Redundancy:** Rules of varying grain size often overlap. For example **Rule2** can be considered to be a part of **Rule1**. If rules are redundant then more rules must be acquired to achieve the same coverage.
3. **Efficiency:** If the rules used by the performance system are too fine grained then a large number of rules need to be applied to complete a design. For interactive systems like VEXED, this means that the user has to make too many choices.

4 Synthetic Parsing

The first two effects of the grain size problem as identified in the previous section suggest that the rules should be as fine grained as possible. However the last effect suggests use of larger grained rules. This apparent conflict can be resolved by splitting the rule learning process into two phases.

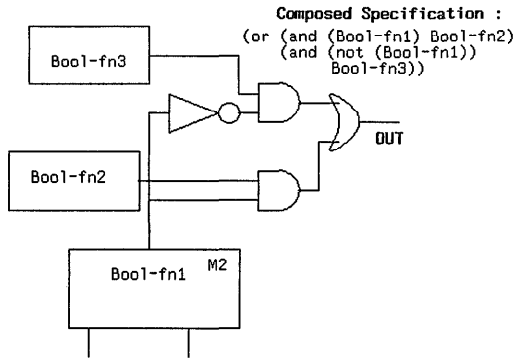


Figure 5: THEN part of a refinement rule

In Phase 1 we try to learn as much as possible from the single training example, i.e. learn rules which apply in more cases and produce a larger range of designs when used along with other existing rules. Hence in this phase the system should try to learn as fine grained a rule as possible. We propose the technique of *"synthetic parsing"* to extract fine grained rules from the manual refinement.

In Phase 2 the objective is to increase the efficiency of the performance system. In interactive systems like VEXED this means that the user has to make fewer control decisions. So we need to reorganize the rules to produce rules of larger grain size. This may be done by forming macro-steps by composing finer grained rules [Huhns and Acosta, 1987] or by storing design plans and "replaying" [Mostow and Barley, 1987] them when necessary.

Our current work implements a prototype for Phase 1.

If a manual refinement step corresponds to a large step, one way to extract a fine grained rule from it is to determine which parts of the step can be accounted for by the existing rules. The part that cannot be accounted for by any existing rule is isolated and generalized into a new rule using LEAP.

Parsing a manual refinement is the process of finding a hierarchy of refinement rules which when applied to the initial module will decompose it into the same circuit as the manual refinement. The hierarchy of existing rules found is called a *parse tree*. If the hierarchy of rules is allowed to contain newly synthesized rules, then the process is called *synthetic parsing* and the hierarchy of rules is called a *synthetic parse tree*.

We assume the user chooses to refine a module manually only if none of the applicable rules refines it toward the desired circuit. Thus the parse tree for the module will require a new rule at the top. So we use a simple bottom-up parser which iteratively replaces a connected set of modules (M1...Mn) by a single module (M), if there is a refinement rule which refines M to the group of connected modules M1...Mn. Given this scheme of parsing, the following questions need to be answered to modify it

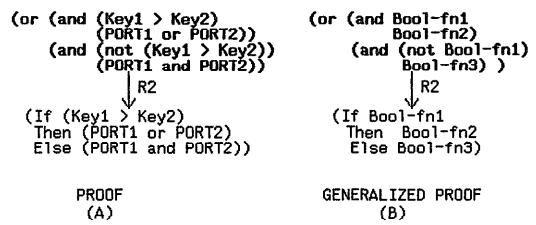


Figure 6: Proof trees for New-rule1

to implement synthetic parsing.

- When to create a new rule? Whenever the basic parsing scheme is in a state where it needs to backtrack, we create a new rule to complete the parse tree.
- How to create a new rule? The new rule created refines the initial module specification to the partially parsed circuit formed at the point when we decided to create a new rule.
- How to choose a good parse tree from which the newly created rule is given to LEAP for further generalization? Currently this is done manually.

In our implementation we have chosen to answer these questions in the most simple manner, since our emphasis in this work is to demonstrate the usefulness of the rule extracted by synthetic parsing rather than the efficiency of synthetic parsing. We have a PROLOG prototype implementation of a synthetic parser whose output can be processed by a simpler version of LEAP, also implemented in PROLOG, which ignores some features of circuits such as timing considered in the original LEAP.

To illustrate, let us assume that **Rule3** is already known to the system, and the user manually refines the specification **Spec1** to the circuit in Figure 2. Synthetic parsing would use **Rule3** to convert the circuit in box B2 of the circuit in Figure 2 to a generic module with output specification (Key1 > Key). Since existing rules cannot parse the circuit any further, synthetic parsing creates a new rule **New-rule1** which refines **Spec1** to the partially parsed circuit obtained by modifying Figure 2 as just mentioned. **New-rule1** is given to LEAP for further generalization. LEAP verifies that the partially parsed circuit implements its composed specification

(OR (AND (Key1 > Key2) (Port1 OR Port2))
(AND (NOT (Key1 > Key2)) (Port1 AND Port2)))

by creating the proof tree in Figure 6(A). The proof tree in Figure 6(A) is generalized to the proof tree in Figure 6(B) from which LEAP creates the rule **Rule2** presented before.

As explained before, **Rule2** is finer grained than **Rule1**, the rule that would have been learned by using LEAP directly on the manual refinement. **Rule2** is more general than **Rule1**, e.g., it applies to **Spec2** as well as to **Spec1**. **Rule2** is also less redundant as it captures only the knowledge missing from the previous set of rules. Thus this example illustrates the use of synthetic parsing to learn rules which are more general and less redundant.

5 Discussion

Parsing can be viewed as explanation [Vanlehn, 1987]. Parsing allows synthesis knowledge (the refinement rules) to explain part of the manual refinement and leaves the rest to be verified by LEAP using its analysis knowledge. Hence learned refinement rules can contribute to explanation of future manual refinements when synthetic parsing is added to LEAP. Parsing also affects some problems related to EBL [Mitchell *et al.*, 1985a] in LEAP.

- **Intractability of verification:** Comparison of the proof (Figure 4) for the complete refinement step and that (Figure 6) of **New-rule1** extracted by the synthetic parser shows that the latter is much smaller. Hence fine grained rules appear to be less expensive to verify and generalize. However this may be offset by the effort required to isolate the fine grained part from the larger refinement step provided by the user.
- **Incomplete theory:** LEAP needs to verify the manual refinement completely before it can learn a new rule. If the analysis knowledge is not complete, LEAP may not be able to learn from a manual refinement, even though most of the refinement step can be verified. By adding synthetic parsing the burden of explanation is shared between analysis knowledge and synthesis knowledge. Hence, even if a part of the manual refinement cannot be explained by analysis knowledge, LEAP still might be able to learn something, provided there is a refinement rule that parses away the problematic part of the refinement step.

5.1 Related work

Other similar systems differ in method, application domain, or purpose.

[Waters, 1985] parses LISP code in terms of programming "cliches," but does not attempt to learn new ones. [Hall, 1986] uses existing rules to explain as much of a circuit design as possible, and learns a new rule from the remainder, but without generalizing. [Vanlehn, 1987] uses existing rules to explain as much of a subtraction protocol as possible, and generalizes the rest into a new rule by an inductive step. In contrast, LEAP's analysis knowledge lets it use EBL for this step.

[Pazzani, 1987] and [Rajamoney, 1988] also use existing rules to explain part of an example. They then use weaker rules to fill the gaps. While their purpose is to complete the explanation in the face of an incomplete theory, ours is to generalize the explanation by omitting parts already explained by existing synthesis rules.

[Dejong and Mooney, 1986] generalizes explanations by replacing subtrees with abstract schemas, just as we parse modules explained by existing synthesis rules. However, the learned structures are used for concept recognition, while LEAP's design rules are used for generation.

[Segre, 1987] generalizes explanations by dropping lower-level details based on a pre-specified measure of the desired tradeoff between the operationality and generality of the robot operator to be learned. In contrast, the grain size of a rule learned in our system is determined by the mismatch between the user's example and the existing rules.

SOAR [Laird *et al.*, 1986] might be viewed as parsing subproblem traces into chunks that "explain" (solve) parts

of subsequent problems. Chunking simplifies future traces by dropping certain subproblem details, such as the preference rules and subgoaling used to guide the search.

5.2 Limitations

In the example considered, the system actually produces two different partial parses of which only one leads to learning **Rule2**. The second partially parsed circuit cannot be verified by LEAP using the rules of equivalence in Figure 3 and hence does not lead to any new refinement rule. Experiments with the prototype parser suggest that if the the manual refinement is large compared to the grain size of existing refinement rules, not only do we get a large parse tree, but also more of them. With many parse trees many new rules would be constructed and it is expensive to identify the ones that are verifiable by LEAP and result in a finer grained rule. Moreover, since different parse trees would bridge the gap in the existing set of refinement rules in various ways, the rules learned are likely to overlap with each other. This defeats the objective of learning non-redundant rules by parsing. These limitations suggest that it would be useful to have heuristics capable of selecting "good" parse trees, perhaps based on the size of the parse tree or the size of the unparsed part of the circuit.

Because a single rule is created for the unparsed portion of the example, the grain size of the learned rule depends on the mismatch between the example and the existing rules. While this approach reduces the combinatorial number of ways in which the example could be decomposed into finer-grained rules, it is sensitive to the order in which examples are presented. In our example, learning New-rule1 depends on already having acquired Rule3; otherwise synthetic parsing will not help. To overcome this limitation without decomposing the unparsed portion, one might try to factor existing rules each time a new rule is acquired [Hall, 1986].

6 Conclusions

We can draw the following conclusions from this work:

- Synthetic parsing combined with LEAP can be used to learn rules which are finer grained than those learned by LEAP alone. Fine grained rules improve coverage and reduce redundancy of refinement rules.
- Synthetic parsing allows the burden of explanation to be shared between synthesis knowledge and analysis knowledge. Hence incompleteness in analysis knowledge may be compensated for by relevant synthesis knowledge.
- Heuristics for selecting "good" parse trees from the many generated by synthetic parsing would be very useful.

References

[Dejong and Mooney, 1986] G. Dejong and R. Mooney. Explanation-based learning: an alternative view. *Machine Learning*, 1(2):145–176, 1986.

[Ellman, 1985] Thomas Ellman. Generalizing logic circuit designs by analyzing proofs of correctness. In *IJCAI85*, pages 643–646, Los Angeles, CA, 1985.

[Hall, 1986] Robert J. Hall. Learning by failing to explain. In *Proceedings AAAI86*, pages 568–572, University of Pennsylvania, Philadelphia, Pa., 1986.

[Huhns and Acosta, 1987] M. N. Huhns and R. D. Acosta. *Argo: an analogical reasoning system for solving design problems.* Technical Report AI/CAD-092-87, MCC, Austin, Texas, March 1987.

[Kowalski and Thomas, 1983] T. J. Kowalski and D. E. Thomas. The VLSI design automation assistant: first steps. In *26th IEEE Computer Society International Conference*, pages 126–130, 1983.

[Laird *et al.*, 1986] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in SOAR: the anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[Mitchell *et al.*, 1986] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[Mitchell *et al.*, 1985a] T. M. Mitchell, S. Mahadevan, and L. Steinberg. LEAP: A learning apprentice for VLSI design. In *IJCAI85*, Los Angeles, CA., August 1985.

[Mitchell *et al.*, 1985b] T. M. Mitchell, L. Steinberg, and J. Shulman. A knowledge-based approach to design. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(5):502–510, September 1985.

[Mostow and Barley, 1987] J. Mostow and M. Barley. Automated reuse of design plans. In *Proceedings of the 1987 International Conference on Engineering Design (ICED87)*, pages 632–647, American Society of Mechanical Engineers, Boston, MA, August 1987.

[Pazzani, 1987] M. J. Pazzani. Inducing causal and social theories: a prerequisite for explanation-based learning. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 230–241, Morgan Kaufmann Publishers Inc., University of California, Irvine, 1987.

[Rajamoney, 1988] S. Rajamoney. Experimentation-based theory revision. In *Proceedings of AAAI Spring Symposium Series- EBL*, pages 7–11, Stanford, CA, 1988.

[Segre, 1987] A. M. Segre. On the operationality/generality trade-off in explanation-based learning. In *Proceedings IJCAI-87*, pages 242–248, Milan, Italy, August 1987.

[Steinberg, 1987] L. Steinberg. Design as refinement plus constraint propagation: the VEXED experience. In *Proceedings AAAI87*, pages 830–835, July 1987.

[Vanlehn, 1987] K. Vanlehn. Learning one subprocedure per lesson. *Artificial Intelligence*, 31(1):1–40, January 1987.

[Waters, 1985] R. Waters. The programmer's apprentice: a session with KBEMACS. *IEEE Transactions on Software Engineering*, SE-11(11):1296–1320, November 1985.