[60] C. Tong and P. Franklin. Learning a satisficing compiler for circuit design. In *Proceedings of the Eleventh IJCAI*. Morgan Kaufmann, 1989.

[61] G. G. Towell, J. W. Shavlik, and M. O. Noordewier. Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI)*, pages 861–866. AAAI Press, 1990.

[62] D. Wilkins. Knowledge base refinement using apprenticeship learning techniques. In *Proceedings of the Tenth IJCAI*. Morgan Kaufmann, 1987.

[47] A. Prieditis and J. Mostow. Towards a PROLOG interpreter that learns. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI)*. Morgan-Kaufmann, 1987.

[48] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[49] P. Riddle. An approach for learning problem-reduction schemas and iterative macro-operators. In *Proceedings of the Workshop on Change of Representation and Inductive Bias*. Morgan-Kaufmann, 1988.

[50] S. Roy and J. Mostow. Parsing to learn fine grained rules. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI)*. Morgan-Kaufmann, 1988.

[51] S. Russell. *Analogical and Inductive Reasoning*. PhD thesis, Stanford University., 1986. Appeared also as The Use of Knowledge in Analogy and Induction, Morgan Kaufmann, 1989.

[52] A. Segre. A learning apprentice system for mechanical assembly. In *Proceedings of the Third IEEE International Conference on Artificial Intelligence Applications*, pages 112–117, 1987.

[53] J. Shavlik. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5:39–70, 1990.

[54] R. Smith, H. Winston, T. Mitchell, and B. Buchanan. Representation and use of explicit justification structures for knowledge base refinement. In *Proceeding of the Ninth IJCAI*. Morgan Kaufmann, 1985.

[55] L. Steinberg. Design = top down refinement plus constraint propagation plus what? In *Proceedings of the 1987 IEEE International Conference on Systems, Man, and Cybernetics*, volume 2, pages 498–502, October 1987. Also available as Rutgers AI/VLSI Project Working Paper No. 67 and CAIP-TR-057.

[56] L Steinberg. Design as refinement plus constraint propagation: the VEXED experience. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI)87)*, pages 830–835. Morgan-Kaufmann, 1987.

[57] G. Tecuci and Y. Kodratoff. Apprenticeship learning in imperfect domain theories. In Y. Kodratoff and R. Michalski, editors, *Machine Learning III*. Morgan Kaufmann, 1990.

[58] C. Tong. Goal-directed planning of the design process. In *Proceedings of the Third IEEE Conference on AI Applications*, February 1987. Available as Rutgers AI/VLSI Project Working Paper No. 41.

[59] C. Tong. *Knowledge-Based Circuit Design*. PhD thesis, Stanford University, 1988. Also available as Rutgers Technical Report LCSR-TR-108.

[33] S. Minton and G. Carbonell. Strategies for learning search control rules: An explanation-based approach. In *Proceedings of the Tenth IJCAI*. Morgan Kaufmann, 1987.

[34] T. Mitchell. *Version Spaces: An Approach to Concept Learning*. PhD thesis, Stanford University., 1978.

[35] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-Based Generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[36] T. Mitchell, S. Mahadevan, and L. Steinberg. LEAP: A learning apprentice for VLSI design. In *Proceedings of the Ninth IJCAI*, pages 573–580. Morgan-Kaufmann, 1985. Also reprinted in Machine Learning: An Artificial Intelligence Approach, Volume III, edited by Y. Kodratoff and R. Michalski., Morgan Kaufmann, 1990.

[37] T. Mitchell and L. Steinberg. A knowledge based approach to design. In *IEEE Proceedings of the Workshop on Principles of Knowledge Based Systems*. IEEE Computer Society Press, 1984.

[38] T. Mitchell, P. Utgoff., and R. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, 1983.

[39] J. Mostow. Design by derivational analogy: Issues in the automated replay of design plans. *Artificial Intelligence*, 40(1-3):119–184, 1990. Special Issue on Machine Learning.

[40] J. Mostow, M. Barley, and T. Weinrich. Automated reuse of design plans. *International Journal for Artificial Intelligence in Engineering*, 4(4):181–196, 1989.

[41] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*. Ohmsha, 1990.

[42] S. Norton and K. Kelly. Learning preference rules for a VLSI design problem-solver. In *Fourth IEEE Conference on AI Applications*, March 1988.

[43] D. Ourston and R. J. Mooney. Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI)*, pages 815–820. AAAI Press, 1990.

[44] M. Pazzani. Refining the knowledge base of a diagnostic expert system: An application of failure-driven learning. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI)*, pages 1029–1035. Morgan-Kaufmann, 1986.

[45] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1992.

[46] P. Politakis and S Weiss. Using empirical analysis to refine expert system knowledge bases. *Artifical Intelligence*, 22:23–48, 1984.

[18] J. Jourdan, L. Dent, J. McDermott, T. Mitchell, and D. Zabowski. Interfaces that learn: A learning apprentice for calendar management. In S. Minton, editor, *Machine Learning Methods for Planning and Scheduling*. Morgan-Kaufmann Press, 1991. forthcoming.

[19] S. Kedar-Cabelli and T. McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the Fourth International Machine Learning Conference*. Morgan-Kaufmann, 1987.

[20] V. Kelly. *The CRITTER System: An Artificial Intelligence Approach to Digital Circuit Design Critiquing*. PhD thesis, Rutgers University., 1985.

[21] J. Laird, M. Hucka, E. Yager, and C. Tuck. Correcting and extending domain knowledge using outside guidance. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 235–243. Morgan Kaufmann, 1990.

[22] J. Laird, P. Rosenbloom, and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1(1), 1986.

[23] P. Laird and E. Gamble. Extending EBG to term-rewriting systems. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI)*. Morgan-Kaufmann, 1990.

[24] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[25] S. Mahadevan. Verification-based learning: A generalization strategy for inferring problem-reduction methods. In *Proceedings of the Ninth IJCAI*. Morgan Kaufmann, 1985.

[26] S. Mahadevan. Using determinations in EBL: A solution to the incomplete theory problem. In *Proceedings of the 6th International Workshop on Machine Learning*, pages 320–325. Morgan-Kaufmann, 1989.

[27] S. Mahadevan. *An Apprentice-based Approach to Learning Problem-Solving Knowledge*. PhD thesis, Rutgers University, 1990.

[28] S. Mahadevan. PED: A technique for refining determination-based theories. In *Proceedings of the AAAI Symposium on Abduction*, 1990.

[29] S. Marcus. *Automating Knowledge Acquisition for Expert Systems*. Kluwer Academic Publishers, 1988.

[30] M. McFarland and T. Kowalski. Incorporating bottom-up design into hardware synthesis. *IEEE Transactions on Computer-Aided Design*, 9(9):938–950, September 1990.

[31] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison Wesley, 1980.

[32] R. Michalski. A theory and methodology of inductive inference. In R. Michalksi, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 83–134. Tioga Press, 1983.

[4] W. Cohen. Generalizing number and learning from multiple examples in explanation-based learning. In *Proceedings of the Fifth International Machine Learning Conference*. Morgan-Kaufmann, 1988.

[5] W. Cohen. The generality of overgenerality. In *Proceedings of the Eighth International Workshop on Machine Learning*. Morgan-Kaufmann, 1991.

[6] T. Davies and S. Russell. A logical approach to reasoning by analogy. In *Proceedings of The Tenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1987.

[7] R. Davis. Applications of meta-level knowledge to the construction and use of large knowledge bases. In R. Davis and D. Lenat, editors, *Knowledge-based systems in Artificial Intelligence*. McGraw Hill, 1981.

[8] G. DeJong. An approach to learning from observation. In *Machine Learning: An Artificial Intelligence Approach, Vol. II*. Morgan Kaufmann, 1986.

[9] G. Dejong and R. Mooney. Explanation-Based Learning: An alternative view. *Machine Learning*, 1(2), 1986.

[10] L. Dent, J. Boticario, J. McDermott, T. Mitchell, and D. Zabowski. A personal learning apprentice. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI)*, pages 96–103. MIT Press, 1992.

[11] T. Ellman. Generalizing logic circuit designs by analyzing proofs of correctness. In *Proceedings of the Ninth IJCAI*. Morgan Kaufmann, 1985.

[12] L. Fu and B Buchanan. Inductive knowledge acquisition for rule based expert systems. Technical Report KSL-85-42, Stanford University., 1985. Knowledge Systems Laboratory Report.

[13] A. Golding, P. Rosenbloom, and J. Laird. Learning general search control from outside guidance. In *Proceedings of the Tenth IJCAI*, pages 334–337. Morgan Kaufmann, 1987.

[14] R. Hall. Learning by failing to explain. *Machine Learning*, 3(1), 1988.

[15] B. Hayes-Roth. A blackboard architecture for control. *Artifical Intelligence*, 26(3):251–321, 1985.

[16] W. Hill. Machine learning for software reuse. In *Proceedings of the Tenth IJCAI*, pages 338–344. Morgan Kaufmann, 1987.

[17] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In *Formal Languages: Perspectives and Open Problems*. Academic Press, 1980. Also appeared as SRI Techreport CSL-111.

central assumption in the VEXED design system is that the user initially provides VEXED with the specifications of a circuit module to be designed. However, in reality, coming up with an accurate specification is a very important part of the design task (consider completely specifying the initially fuzzy task of designing a house, for example). Furthermore, LEAP uses a specification language that is based on a LISP style syntax, which may be awkward for non-specialists to understand.

Domain experts can also help pinpoint the kinds of design knowledge that change relatively rapidly due to changes in the underlying technology, and thus are good candidates for learning, versus design knowledge that remains fairly static and can therefore be programmed into the system. Another benefit of involving experts is that it forces attention on aspects of the knowledge acquisition problem, such as the "friendliness" of the user interface etc, that are all too frequently ignored. Finally, it allows an experimental analysis comparing the quality of solutions produced by the expert with those learned by the system.

10 Acknowledgements

We thank the following members of the AI/VLSI group at Rutgers who were vital to the success of the VEXED and LEAP projects: Saul Amarel was one of the original proponents of learning apprentices; Mike Barley developed the BOGART program for replaying manual design decisions; William Cohen developed the ADEPT technique for learning iterative problem-reduction operators; Smadar Kedar and Rich Keller provided valuable help during the initial development of VBL; Van Kelly developed the CRITTER program as part of his thesis; Sunil Mohan implemented the rule language in VEXED; Armand Prieditis developed the PROLEARN program which was used to implement VBL and PED; John Roach developed a program for laying out circuits created by VEXED; Subrata Roy developed a translator for mapping LEAP operators onto VEXED operators; Jeff Shulman implemented large portions of the VEXED system; Chris Tong provided many stimulating discussions on knowledge-based design; and finally, Timothy Weinrich reimplemented the CRITTER circuit analysis program for use in VEXED and LEAP. Finally, we thank the reviewers of this paper for many useful comments.

References

[1] V. Barker and D. O'Conner. Expert systems for configuration at Digital: Xcon and beyond. *Communications of the ACM*, 32(3):298–318, March 1989.

[2] H. Barrow. VERIFY: A program for proving the correctness of digital hardware designs. *Artifical Intelligence*, 24(1-3), 1984.

[3] W. Clancey. Classification problem solving. In *Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI)*. Morgan Kaufmann, 1984.

9.3 Limitations of Our Work and Directions for Future Research

One major problem is that the learning techniques we have proposed – VBL and PED – do not scale up very well. VBL suffers from all the limitations of explanation-based learning. For example, VBL needs to verify a given refinement step in order to learn from it, but in general circuit verification is known to be an intractable problem [2]. One direction for future work is to see how VBL scales up to deal with very complex circuits, such as microprocessors. Clearly, some additional constraints are needed such as organizing verifications in a series of abstraction levels [2], or limiting the types of refinement.

Similarly, PED assumes a circuit cost theory that is representable as a set of determinations. However, some circuit cost metrics such as area are difficult to formulate in this manner. One possibility is to investigate alternative representations of circuit cost theories, such as a hand crafted evaluation function that estimates area of circuits implemented using a fixed layout scheme, such as programmable logic arrays or gate arrays [30].

The design model used in LEAP – top-down refinement plus constraint propagation – is impoverished [55]. Design often requires an optimization step whereby the existing solution is compacted in some way. It would be interesting to investigate a LEAP-like apprentice system for a more complex design system than VEXED.

Another problem is that the architecture of a learning apprentice, as embodied in the LEAP system, is limited. LEAP is intended to be a *passive* learner, which watches the user, and learns by recording and subsequently explaining his or her steps. One limitation of this architecture is the low bandwidth of communication between LEAP and its user. Human apprentices are able to communicate with their teachers by asking specific questions, and using the resulting answers to complete their knowledge. Adding such a capability to LEAP would greatly improve its ability to deal with more complex design situations. For example, the process of verification in VBL could be made interactive, with the user filling in part of the proof for LEAP, although this could pose a non-trivial burden on the user. Similarly, a user could assist PED in completing gaps in a domain theory, as is done in the DISCIPLE sytem [57].

The learning task faced by LEAP may be overly ambitious. It seems much easier to program certain types of design knowledge into VEXED, such as a routine for compiling boolean expressions into programmable logic arrays, than to learn this knowledge. On the other hand, it may be easier for LEAP to fine tune its programmed design knowledge, such as learning control information on when to use its existing set of operators, than to manually enter this knowledge.[20] The point here is that while a learning apprentice that can acquire all types of design knowledge may be years away, we may be more successful in building an apprentice for a non-trivial subtask of the overall design problem.

Finally, one important direction for future research is to actively involve domain experts in the design and testing of knowledge acquisition systems. Domain experts sometimes have a very good feel for where AI based problem-solving aids will be of help. For example, a

[20]Interestingly, introductory textbooks on VLSI design such as [31] mostly provide the building blocks for design, such as circuits for implementing generic combinational and memory functions. However, they seem to contain little information on ways of controlling the search through the design space of possible implementations using the standard building blocks.

operators.[19] The second hypothesis was addressed in Section 4 and Section 6. Finally, the last hypothesis was addressed in Section 5 and Section 6.

9.2 Experimentally Validating Learning Apprentices

Much work remains to be done to "really solve" the problems addressed in this paper. The goal of building a learning system that can automatically construct the knowledge base used in an expert system remains one of the prime motivations underlying machine learning. It is also pragmatically important given the widespread use of expert systems in the industrial environment.

Clearly, our paper contains only suggestive but not convincing evidence in support of learning apprentices. What would it take to empirically substantiate the claim that learning apprentices can automate significant portions of the knowledge acquisition task? We discuss below some pieces of evidence needed to make the case for learning apprentices.

- One hallmark of a mature technology is its effectiveness across *multiple* domains. Expert systems and neural networks are two recent technologies that have clearly crossed this hurdle. In section 8, we described learning apprentice systems for several domains including circuit design, scheduling, robot assembly, software development, calendar management, and medical diagnosis. However, most of these systems are *prototypes*, and not fully functioning software in regular commercial use.

- Another important piece of evidence in support of learning apprentices is its use in developing a *state of the art expert system.* That is, we need an experimental study showing how a particular learning apprentice system was vital to the development of a commercially successful expert system like XCON [1].

- There are several alternate approaches to knowledge acquisition, ranging from direct questioning of an expert to rule induction from a database of classified instances. To fully make the case for learning apprentices, we need *comparative studies* showing situations where the learning apprentice approach clearly outperforms other approaches.

- One advantage to be leveraged from a learning apprentice is by distributing multiple copies across a user community. The ensemble of such systems will be exposed to many more training instances than any one system. However, challenging knowledge integration issues are involved in combining the rules learned from multiple users. One useful experimental study would be to show that the benefits of having multiple copies of a learning apprentice clearly outweigh any difficulties in integrating their learned knowledge.

[19] The work by Dent et al. on the CAP calendar apprentice [10] clearly shows that learning apprentices can effectively collect training instances during normal use.

ANA-EBL represents a "batch" approach, while PED is an "incremental" technique that extends the domain theory only when it is unable to explain a given training instance.

Recently, there have been several successful attempts of inductive learning of horn-clause programs by building generalizations of examples from relational data – the most well-known of which are FOIL [48], GOLEM [41], and FOCL [45]. These can be viewed as filling gaps in an incomplete theory by using examples. However, the only gaps that they can fill are those that correspond to the top-level predicate. In contrast with PED, these techniques are suitable to batch learning situations with little or no theory, but with a large number of positive and negative examples. Unlike PED, FOCL can also recover from incorrect theories, using the information theoretic gain over the set of examples as a heuristic to identify and repair incorrect parts of the domain theory. None of these techniques, however, have any semantic constraints similar to determinations on the kinds of clauses they can add to their theory.

9 Conclusions and Future Work

This section summarizes the main results of the paper, and identifies problems for future research.

9.1 Summary

The paper attempts to establish the following three hypotheses:

1. A learning apprentice based on a top-down refinement problem-solver can collect training examples of problem-solving knowledge that it needs to learn through the normal course of its use. By "normal use", we mean that the system does not have an explicit training phase, but instead learns as it is being used to perform its normal task.

2. Given a domain theory for checking the correctness of a refinement step, general problem-reduction operators for boolean circuit design can be learned by using VBL from training instances collected by a learning apprentice. Furthermore, using this approach a learning apprentice can automatically acquire a database of boolean circuit design operators comparable to a manually created set.

3. Given training instances of preference rules collected by a learning apprentice, and an incomplete determination-based theory of circuit cost, rules extending the cost domain theory as well as general preference rules for ranking alternative operators can be learned using the PED technique.

Some evidence for the above hypotheses was provided in the paper (for more experimental details, see [27]). In particular, Section 3 addressed the first hypothesis by showing how the design of LEAP allows it to capture training instances during normal use. However, we hasten to add that the data used in the experimental evaluation in Section 6 was not collected by LEAP during normal use, but was instead created by instantiating handcoded

8.3 Learning Control Knowledge

There has been a lot of work on control rule learning in the context of autonomous problem solvers including SOAR and PRODIGY [22, 33]. PRODIGY and SOAR can learn a variety of control rules including preference, select and reject rules. However, since they employ EBL, the control rules learned by these systems are deductive consequences of their domain theories, which is not necessarily true in LEAP.

Golding et al. [13] describe a SOAR-based system that, like LEAP, learns control knowledge from outside guidance. However, unlike LEAP, this system assumes that its domain theory is sufficient to explain the advice given to it by a user.

Kelly and Norton [42] describe a system called PR-LEAP that learns preference rules in VLSI design. One difference is that LEAP can learn preference rules from an incomplete cost theory, whereas PR-LEAP requires a complete domain theory. Furthermore, PR-LEAP only applies to "leaf" steps in the refinement tree that yield circuits composed of primitive components.

8.4 Learning from Incomplete Domain Theories

Hall [14] describes a technique called "learning by failing to explain", for dealing with incomplete verification theories. The basic idea is to explain as much of the circuit as possible, and storing the unexplained portions as a new design rule. A critical assumption of this approach is that the failure of the explanation can be localized to particular submodules in the circuit. It is unclear whether this assumption is true in all situations when the explanation fails because of missing verification rules.

Laird et al. [21] describe a system called ROBO-SOAR that is able to correct its domain knowledge by actively seeking outside guidance. One difference between this system and LEAP is that the latter uses determinations to explicitly specify possible gaps in the domain theory.

Ourston and Mooney [43] describe a system called EITHER that can fill in missing rules and antecedents in an incomplete theory, and remove incorrect antecedents. It uses a form of Occam's razor to choose among different possible modifications to the theory. The KBANN algorithm of Towell et al. [61] can also learn from incorrect and incomplete domain theories by using them to initialize a neural net, and training it on the examples using the backpropagation algorithm. Both of these approaches, however, have only been demonstrated in prepositional horn clause theories, and do not capture determination type knowledge.

Cohen [5] describes a technique (ANA-EBL) that extends explanation-based learning to solve theory specialization tasks. A theory specialization task is defined as specializing an over-general theory to make it consistent with the training examples. The over-general theory may be inconsistent and incorrect. Cohen shows that his system can handle incomplete theories such as determinations by manually replacing the determination with the set of all possible completions of that determination, and then using the examples to choose the correct subset of completions. Viewed as a technique for completing partial domain theories,

Replay Systems A different approach to the problem of acquiring design knowledge is taken by "replay" systems such as BOGART [39, 40]. BOGART records the manual refinement steps contributed by a user in a hierarchical "design plan" structure. At any point in the design, the user can examine the design plan, and replay or retract a step in it. This service is particularly helpful for large designs, which may have many copies of the same module. LEAP's ability to replay manual design steps is limited, since it discards potentially useful information whenever it abstracts a manual refinement step into a general operator. For example, LEAP may ignore the fact that a module in the user's refinement was primitive (meaning it was selected from the catalog of library modules in VEXED). When replaying the step, LEAP may create a module with more general specifications (for example, it may generalize the encoding). As the experiment on VBL in Section 6 showed, operators learned by LEAP create finer grained refinement steps than the manually created operators. A finer grained refinement is not always desirable since it increases the number of steps required to implement a given module, which might considerably increase the amount of problem-solving search required.

8.2 Learning Problem-Solving Operators

Cohen's system ADEPT [4] can learn macros of verification rules such as those used in VBL to verify circuits. One advantage of ADEPT is that it is capable of generalizing parameters in a refinement step, such as the number of module inputs. In other words, it addresses the generalization-to-N problem [53] unlike VBL. On the other hand, VBL can deal with more circuit features (such as `data_value` and `encoding`). Also VBL outputs generalized problem-reduction operators, not just macros of verification rules.

Tong and Franklin [60] describe a technique for refining correct problem-reduction operators into efficient ones. One limitation of their approach is that they assume fanout free circuits (an output cannot be connected to more than one input). Several VBL-learned operators, such as EQUAL-OR-DECOMP, have fanouts greater than 1. Furthermore, their method relies on being able to generate all possible refinements, which grows exponentially with the number of possible operators.

Ellman [11] describes a technique for generalizing logic circuit designs into problem-reduction operators by using proofs of correctness. Although Ellman's technique is very similar in flavor to VBL, there are several important differences. First, in Ellman's system, the verification was manually supplied by the user, and not automatically generated as in LEAP. Second, the proof of correctness amounted only to composing the specifications of the submodules, but not any simplification. Third, Ellman's system did not represent the operating conditions of a circuit, but only its functionality. Finally, unlike Ellman's technique, VBL is incorporated as part of a learning apprentice system, and is able to exploit key architectural features of LEAP to simplify its task. On the other hand, Ellman showed in his paper how the circuit topology could itself be generalized by embedding it in the module specification, which cannot be done by VBL.

is that the Dipmeter Advisor is improving a heuristic classification problem-solver whereas LEAP is improving a hierarchical planner.

ARMS [52] is a learning apprentice system for assembly tasks in robotics. It learns schemas for assembling configurations, such as revolute joint. ARMS is not based on an interactive performance element. Hence, training instances are sequences of primitive actions. ARMS thus has a more difficult explanation problem, since there is a considerable gap between the abstraction level of the goal and the primitive solution sequence. ARMS also has to confront the *grain size* problem, since it has to abstract the primitive solution sequence, before generalizing it to form a schema. Otherwise the schemas it learns will always map a goal specification to a primitive solution, requiring many more schemas, and creating many redundancies between different schemas. Finally, ARMS assumes that a complete domain theory is available to explain the given training instance.

LASR is a learning apprentice system for software reuse proposed by Hill [16]. LASR is intended to acquire implementation methods for software design. Hill suggested a VBL-like technique for acquiring operators for implementing data structures such as stacks. One difference between LEAP and LASR is that LASR only acquires implementation methods for software design, and not control knowledge regarding when the methods are useful.

ODYSSEUS [62] is a learning apprentice designed to acquire and improve the knowledge of a heuristic classification system. ODYSSEUS uses an abduction-based technique to fill gaps in the domain theory. Its strategy differs significantly from the way PED fills gaps in the domain theory, because PED relies on determinations to constrain the gap-filling process. Another difference between LEAP and ODYSSEUS lies in the representation of domain knowledge. In ODYSSEUS, domain knowledge is primarily in the form of tuples representing facts, such as `finding(diplopia,yes)`. This simplifies the knowledge base refinement problem, since adding new knowledge means only adding instantiated facts. In LEAP, new knowledge takes the form of Horn rules, which complicates the refinement problem.

DISCIPLE [57] is a multi-strategy, domain-independent learning apprentice system that learns problem decomposition rules. It integrates several learning strategies including EBL, analogy, induction, and learning from queries. It learns from incomplete domain theories by using heuristics to construct partial explanations, and having them critiqued by the user. Unlike LEAP, DISCIPLE does not learn rules to control the search space of feasible decompositions. Also, LEAP differs from DISCIPLE in its use of determinations to constrain the gap filling process. Finally, it is unclear how the explanation generation process in DISCIPLE is controlled. Since it queries a user to fill gaps, it needs to be able to focus on good partial explanations to ask good questions.

CAP [10] is a learning apprentice system for calendar management. CAP learns scheduling rules for arranging meetings during its normal use as an electronic calendar. One unique feature of CAP is that it is the first apprentice system that has been in routine use over a fairly long period (over six months). CAP uses inductive methods (ID3 and backpropagation) to generalize training instances into rules. Thus, it needs a fairly large database of instances (on the order of hundreds of examples) before it becomes effective. In contrast, LEAP relies on knowledge-intensive learning methods, and can produce a good set of rules from far fewer training instances.

- *The learning techniques* can be of two types, *knowledge-intensive* or *knowledge-poor*. The former consists of techniques such as explanation-based learning [35], in which a domain theory is used to explain and then generalize training instances. Knowledge-poor learning techniques, such as version spaces [34], inductively infer an appropriate concept from training instances using a generalization language.

Using these criteria to categorize our work, we would say that LEAP is a knowledge acquisition system built around a synthesis problem-solving model, which acquires knowledge passively, and which uses knowledge-intensive learning techniques. We now compare LEAP with some other knowledge acquisition systems.

SEEK [46], INDUCE [32], and RL [12] are three related knowledge acquisition systems intended for use in heuristic classification tasks. These systems learn rules from a database of classified instances prepared by their programmers. Thus, unlike LEAP they require an explicit training mode. Also, since these systems are non-interactive, they have to address the credit assignment problem of figuring out which rules are incorrect in a sequence of rule firings leading to an incorrect classification. Finally, unlike LEAP these systems rely solely on inductive learning techniques.

TEIRESIAS [7] interacts with an expert in tracking down incorrect or missing rules in a medical diagnosis system. In case of missing rules, the expert supplies the rules which the system then modifies using a rule model. BB1 [15] is a general blackboard architecture for problem-solving and learning. Unlike LEAP, BB1 does not acquire new problem-solving operators, but only control information regarding when they should be used. However, BB1 can represent more powerful forms of control knowledge than LEAP can, such as "design the most constraining module first". Both these knowledge acquisition systems use an active question-and-answer dialog session with a human user; thus, in both these systems, the user plays a much more active role than in LEAP.

MORE, MOLE, SALT, KNACK, and SIZZLE are a set of related knowledge acquisition systems [29] that acquire some part of their knowledge by interacting with an expert through a question-and-answer dialog session (although MORE, MOLE, and SALT also incorporate techniques for compiling rules from basic causal knowledge acquired from the expert). While MORE and MOLE are primarily intended for knowledge acquisition in heuristic classification tasks, SALT acquires knowledge for solving synthesis tasks. KNACK and SIZZLE are specialized to data presentation and sizing tasks, respectively. These systems are more powerful than LEAP in being able to correct erroneous rules in their knowledge base. However, they do require an active training session with an expert.

Pazzani [44] describes a failure-driven knowledge-intensive learning technique for acquiring knowledge for a diagnostic expert system called ACES. His system assumes completeness of the domain theory, as opposed to LEAP which can deal with limited kinds of incompleteness in the domain theory.

A lot of recent work on knowledge-intensive approaches to the knowledge acquisition problem has been based on the learning apprentice model. We briefly summarize these systems and relate them to LEAP.

Smith et al. [54] describe a learning apprentice in the domain of geology based on an expert system called the *Dipmeter Advisor*. One difference between this system and LEAP

7.3 Separating Correctness and Control

A third feature in the design of LEAP is the partitioning of implementation knowledge from control knowledge. One reason for this partition is that implementation knowledge is monotonic – adding a new problem-reduction operator does not require that existing ones be revised. This is important in a learning apprentice system, such as LEAP, since one attractive scenario for such systems is distributing multiple copies of them in a user community. While this was not done in the case of LEAP, the problem of combining multiple operator sets will need to be addressed for systems that do get distributed. Although combining sets of operators may not be as easy as unioning them – doing so will undoubtedly cause some grain size problems and create redundancies – nevertheless, the property of monotonicity makes this problem easier for the case of implementation knowledge than for control knowledge. This is because control knowledge, which specifies efficient ways of implementing circuits, is non-monotonic since adding more implementation choices may override existing efficient solutions.

A second reason for this partition is that it simplifies the task of explaining a refinement step. The explanation step in VBL involves explaining the user's refinement step. By partioning the implementation knowledge from the control knowledge, VBL is only required to explain why the user's step is *correct*. This is more tractable than having to prove that the user's step is efficient, which would require considering alternative problem-reduction operators.

A third reason for this partition is that it leads to a more compact set of problem-reduction operators. For example, many different refinement steps can be viewed as instances of the same operator if the control knowledge specifying when a refinement was useful is removed from an operator. This is why the number of operators learned by LEAP was 20 percent less than the number of handcoded operators in the experiment described earlier.

8 Related Work

8.1 Knowledge Acquisition

We distinguish previous work in knowledge acquisition in the following three ways:

- *The problem-solving model* refers to the manner in which the problem solver (or expert system) uses the rules learned by the knowledge acquisition system. In *classification* tasks [3], such as diagnosis, problem-solving consists in selecting a solution from a set of enumerated candidates. In *synthesis* tasks, such as design, problem-solving occurs by constructing a solution from primitive pieces.

- *The acquisition model* refers to the mode of interaction between a knowledge acquisition system and a user. A knowledge acquisition system can operate in a *passive* mode, whereby it learns by silently observing the user solve problems. Alternatively, it can operate in an *active* mode, where it engages the user in a question-and-answer dialog session to extract rules.

The Grain Size Problem The above benefits are offset to a certain extent by the *grain size problem*. Depending on the situation, LEAP may collect example refinement steps that are more or less fine grained. For example, a serial to parallel converter module can be refined in several steps, first by selecting a shift register, next by choosing a particular class of registers, and finally by choosing a specific register circuit. However, it is possible that a user considers this as a single step, and directly implements a serial to parallel converter with a particular circuit.

The grain size problem impacts each of the above benefits. If a refinement step corresponds to a sequence of problem-reduction operators, some of which are already in the system, then training examples no longer correspond to knowledge missing from the system. In such a situation, every time LEAP learns a new operator, it has to examine whether the new operator is actually a sequence of existing operators, which may involve exhaustively generating operator sequences. Also, if sequences of operators have to be analyzed for every training example, then the savings from not having to address the credit assignment problem is lost. Finally, the argument for the tractability of explanations in a learning apprentice loses force if the training examples correspond to large refinement steps.

The grain size problem in learning problem-reduction operators in circuit design has been addressed by Roy and Mostow [50]. Their solution was to "parse" the refinement suggested by the user using previously learned problem-reduction operators to the extent possible, before trying to learn from the refinement step. The advantage is that redundancies in the problem-reduction operators are avoided, as well as the fact that more general operators can be learned from a given refinement step.

7.2 Use of Knowledge-Intensive Learning Methods

A second feature of LEAP is the use of knowledge-intensive or analytical learning methods. One benefit of using analytical learning techniques in a learning apprentice is the possibility of learning from noisy data. For example, we envision LEAP learning from a community of university students who are themselves learning about VLSI design. It is quite probable that students will create faulty circuits. Since LEAP uses VBL – an explanation-based learning technique – to generalize refinement steps, LEAP is insensitive to errors in training data for learning problem-reduction operators. This is not true for learning control knowledge, as we discuss next.

The Imperfect Domain Theory Problem Although analytical learning techniques offer a number of advantages, they require that the system begin with a perfect domain theory. This is an unrealistic assumption in general. For example, in Section 5 we showed that LEAP's theory of circuit cost metrics is incomplete. Since LEAP relies on user's control decisions to complete its circuit cost theory, it can learn incorrect cost rules if it is given incorrect control decisions.

Finally, PED cannot deal with high level control decisions such as choosing a synchronous circuit versus an asynchronous circuit because it currently explains control decisions in terms of power or delay costs. High level control decisions, such as synchronous versus asynchronous clocking, cannot be evaluated solely in such terms. For example, synchronous clocking is desirable because it reduces the possibility of timing errors in the circuit. On the other hand, asynchronous clocking can lead to simpler circuitry since clock signals do not have to be routed to all components in the circuit.

Theory Refinement PED can fill only limited types of gaps in a domain theory. In particular, it assumes the gaps are specifiable as determinations, which means it can deal only with limited forms of incompleteness. Also, it assumes the determinations are correct. PED makes a closed world assumption (predicate completion) regarding the implicative portion of the domain theory. Finally, even with the closed world assumption, there are certain combinations of determinations (such as a chain of two determinations) that PED cannot refine using a single training instance. A more detailed explanation of PED's limitations is given in [27].

7 Architectural Features of LEAP

This section considers some central architectural features of LEAP and analyzes how they affect the knowledge acquisition task faced by LEAP.

7.1 Interactive Nature

An important feature of LEAP is that it incorporates a learning element in conjunction with an interactive problem-solver. There are several benefits in doing so:

- Training examples correspond to knowledge missing from the system. This is because the user intervenes in the problem solving only when the system is missing some knowledge that is relevant to the particular task at hand. For example, in the LEAP system, the user intervenes only when it is unable to refine a module, or when it cannot rank the set of alternative implementations.

- Training examples correspond to single problem-solving steps. This occurs because the user provides two kinds of advice: one, the user manually refines circuit modules, creating instances of problem-reduction operators; second, the user selects an operator from several alternatives, thus giving the system a training example of a preference rule it lacks. Because of the interactive dialogue between the user and the system, learning apprentice systems, such as LEAP and ODYSSEUS [62], as well as other interactive knowledge acquisition systems, such as TEIRESIAS [7] and BB1 [15], do not face the difficult problem of credit assignment faced by a system such as SEEK [46]. Also, verifying the correctness of single problem-solving steps is easier than verifying complete designs [2].

can be constructed, the table already reveals some generic limitations in using PED to learn control knowledge in LEAP.

One problem with our approach to learning preference rules for low power or low delay boolean circuit design is the specificity of the resulting rules. For example, the preference rule illustrated in Figure 5 (on page 8) applies only to designing a restricted type of `AND` expression because the explanation used to generate the rule was constrained by the details of the two circuits. However, it is likely that even for different types of `AND` expression, a NOR gate implementation would be faster than one implementing the negation of the `AND` expression.

To learn more general rules, a more abstract theory of preference is needed. For example, NMOS designers use general heuristics like "Prefer NOR gate implementations to NAND gate ones when low delay circuits are needed". This piece of control knowledge is more abstract than the preference rules our approach is capable of learning. In order to learn control knowledge of this form we need to develop more abstract ways of describing refinements, and incorporate these in a larger theory for explaining preferences. For example, consider the following determination:

$$\text{ref_type(Op1,R1)} \wedge \text{ref_type(Op2,R2)} \wedge \text{applicable(Op1,Op2,S)} \succ \text{prefer(Op1,Op2,S,G)}$$

It states that the type of refinement `R1` and `R2` of operators `Op1` and `Op2` determines which one is preferable for refining specification `S` given design goal `G`. Given suitable training examples, and some knowledge for computing the type of refinement represented by a given operator, PED can learn preference rules of the following form:

$$\text{ref_type(Op1,nor)} \wedge \text{ref_type(Op2,nand)} \wedge \text{applicable(Op1,Op2,S)} \Rightarrow \text{prefer(Op1,Op2,S,speed)}$$

This rule states that a NOR gate implementation is preferable to a NAND gate implementation whenever speed is the design goal. Learning such rules is preferable to directly encoding them because in general there are many such rules, only a portion of which may be relevant to a particular design context. Of course, the above rule can be wrong if we compare a very inefficient NOR gate implementation with a very efficient NAND gate implementation. The success of the approach depends on whether there is a succinct description of possible types of refinement, and how it can be integrated with a more accurate theory for delay estimation.

Another problem with the LEAP control learner is that it is sensitive to the order in which training instances are processed. For example, suppose LEAP had been given the `And(Or,And)` example in Table 19 (row 6) before the `Or(And,And)` example (row 3). LEAP would fail to explain the `And(Or,And)` control decision because two modules with unknown delay values are involved (both the `And` module and the `Or` modules are created by `operator_fxydecomp` in its implementation of `And(Or,And)`). One partial solution to the sensitivity problem is to let LEAP store control decisions that it was unable to explain, and reanalyze these examples as it acquires more circuit cost and preference knowledge (this approach is similar to Hall's rule reanalysis technique [14]).

The table illustrates the points mentioned above regarding PED's uniqueness as a control learner. First, it shows that LEAP learns two types of knowledge from each control decision, namely new cost rules and preference rules. Second, it depicts across-trial learning – delay rules learned by PED for the `Or(And,And)` example (row 3) are reused in explaining the `And(Or,And)` example (row 6). Third, it shows how the second extension to PED (search) described in Section 5.3.2 is sometimes needed to explain a given control decision (rows 7 and 8). The table, however, does not illustrate the use of learned preference rules in explaining subsequent decisions, because the examples in the table were selected to emphasize the learning of *new* cost rules and preference rules. If the examples in the table were given to LEAP after learning, the learned preference rules would filter out all but one of the operators (namely, the operator chosen by the user in each example) because for each of the other operators LEAP learned a preference rule that ranks the user's operator over that operator.

Description	Design Goal	Operators Applicable	Results
And(Or,Or)	Speed	**op_nors3**, op_notdecomp	2 delay rules added 1 pref. rule added
Or(And,And)	Power	**op_nors3**, op_notdecomp	1 power rule added 1 pref. rule added
Or(And,And)	Speed	**op_3nand**, op_notdecomp, op_fxydecomp	4 delay rules added 2 pref. rules added
And(And,And)	Speed	op_notdecomp, **op_nors3**, op_fxydecomp	2 delay rules added 2 pref. rules added
And(And,And)	Power	op_notdecomp, **op_nors3**, op_fxydecomp	1 power rule added 1 pref. rule added
And(Or,And)	Speed	op_notdecomp, op_fxydecomp, **op_3nand**	2 delay rules added 2 pref. rules added Learned rules used
Equal(Or,Or)	Speed	**op_equal_or**, op_notdecomp	2 delay rules added 1 pref. rule added Search used
Equal(Or,Or)	Power	**op_equal_or**, op_notdecomp	1 power rule added 1 pref. rule added Search used

Figure 19: Examples to Test Control Learning using PED. Operator chosen by user is shown in boldface.

6.4 Limitations of PED

Control Learning It is difficult to gauge the effectiveness of PED across the entire domain of boolean circuits from the table in Figure 19. Although more examples of control learning

can divide possible target concepts into those that require global information from the whole problem-solving episode versus those that do not. For example, "correct refinement step" is a target concept that does not require examining the entire problem-solving episode. It is possible to prove that X is a correct refinement step, independent of the larger problem-solving context in which it appears. On the other hand, "optimal refinement step" is a target concept that requires globally examining all possible refinements. Because of efficiency considerations, VBL is practically restricted to "local" target concepts, such as "correct refinement step". Also, VBL assumes that the verification domain theory is complete, tractable, and correct. Finally, VBL assumes training instances correspond to single problem-solving steps.

- *Preconditions on the steps:* Each of the steps in VBL makes a number of important assumptions. The first step (compose) of VBL assumes circuits with no feedback. The second step of VBL (verify) assumes that the term rewrite rules are terminating (they always simplify an expression) and confluent (every expression has a unique normal form). Finally, the last step in VBL (generalize) assumes a binary notion of operationality.

- *Quality of output:* VBL produces correct operators; however, the operators could yield suboptimal refinements. VBL does not address the issue of producing optimal refinement steps [49, 60, 42]. The PED technique addresses this issue in the context of the LEAP system.

6.3 Evaluation of PED

We now turn to evaluate the strengths and limitations of the PED technique.

Control Learning One difficulty in testing the effectiveness of PED is that, unlike other techniques for learning control information [22, 33, 38], PED not only adds new control knowledge, but also refines the domain theory used to explain control decisions. Thus, we have to evaluate PED not only in terms of how it enhances LEAP's ability to rank competing refinement operators, but also in terms of how it increases LEAP's ability to explain subsequent control decisions. Furthermore, the problem-solver in LEAP (VEXED) is an interactive one, which makes any measurement of the reduction in search time with control learning difficult. Instead, we simply illustrate PED's ability in learning control knowledge, and its ability to use new domain theory rules learned during past control decisions to explain subsequent instances.

Figure 19 describes a sequence of control decisions on which PED was tested. Each row in this figure describes the specification of the module being refined, the design goal, the set of applicable operators returned by LEAP, the operator selected by the user in boldface, and the results of learning. The result of each example is described by how many new rules for circuit cost were added to the domain theory, as well as how many new preference rules were added.

6.2 Limitations of VBL

The VBL technique has several important limitations, which we describe below.

Handcoded operators Not Learnable by VBL We list below some categories of handcoded operators not learnable by VBL, and discuss possible extensions to VBL to enable it to learn these operators.

- *COMB-SHIFT Operators:* These operators implement modules that compute functions of delayed (that is, earlier) inputs. VBL cannot learn these handcoded operators because they involve reasoning about time delays. To learn these operators, the domain theory given to VBL needs to be extended to deal with verification of timing constraints.

- *MEMORY Operators:* These operators apply to modules whose implementation requires the use of memories. In the COMB-SHIFT operators above, the output at a given time is a function of the inputs at some previous time instant, which is a fixed interval in the past. However, in the MEMORY operators, the output at a given time may depend on the input values at varying previous time instants, such as "the last time the LOAD signal was HIGH". To learn such operators, VBL's domain theory needs to be extended to deal with verification of finite state machine circuits.

- *COMBINE Operators:* These are optimization operators that compress a set of circuit modules into a smaller set while preserving the overall function computed by the circuit. Some forms of optimization operators can be viewed as inverses of refinement operators [14]. Since VBL uses a bi-directional rewrite strategy, it can be modified to learn optimization operators that are inverses of refinement operators. Such an operator is of the form: IF the circuit description is of the form `X` THEN transform it to the form `Y`. Here `Y` is a single module, and `X` is one particular implementation of the module. However, there are other optimization operators, such as a *structure sharing* operator that merges two submodules that compute the same function of the same inputs, that would require extending VBL's domain theory for it to learn them.

- *ITERATIVE Operators:* These operators involve looping and recursive constructs. Typically, they are parameterized by the number of input bits. Such rules are used to construct N-bit multipliers and adders. VBL cannot learn these operators since they involve iteration. It would be useful to extend VBL to incorporate some techniques that have been developed for learning iterative constructs [53]. In particular, Cohen's system ADEPT [4] is capable of learning design operators involving iterative constructs.

Other Limitations of VBL We can classify other limitations of VBL into the following categories:

- *Restrictions on inputs to VBL:* The inputs to VBL (like other instances of explanation-based learning) are a target concept, a domain theory, and a training instance. We

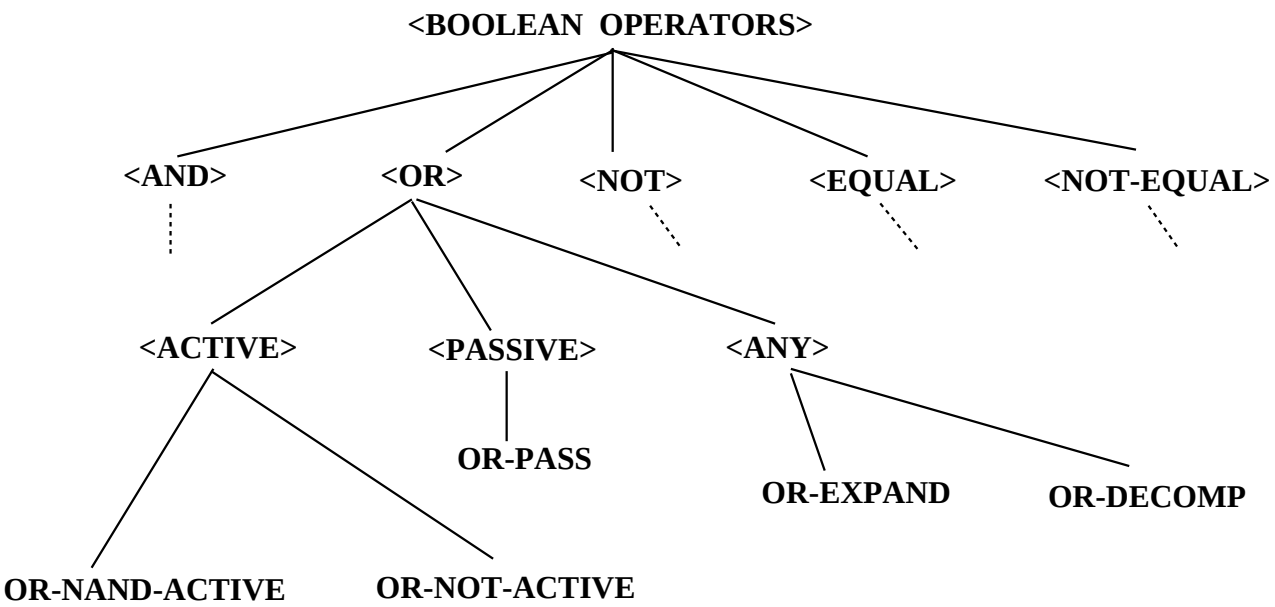


Figure 17: A portion of the hierarchical structure of the handcoded operators

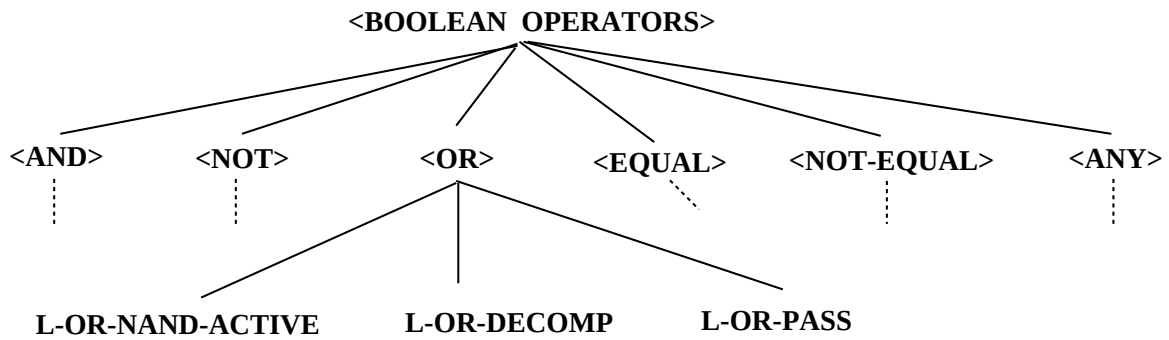


Figure 18: A portion of the hierarchical structure of LEAP-learned operators

boolean functions. Second, at level 2, the distinction between operators based on encoding has disappeared. Handcoded operators that applied to either active or passive encoding have been generalized to apply to arbitrary encodings. Finally, the number of operators for implementing disjunctions (OR's) has been reduced from five to three. The remaining two handcoded operators are subsumed by the three LEAP operators. Once again, we have detailed only the structure under `OR`; the structure for the other functions is similar. We have prefixed the letter `L` onto the LEAP operator names to distinguish them from the corresponding handcoded operators.

- *Erroneous Operators:* If VBL fails in proving the correctness of a training instance, this may be because the training instance was incorrect to begin with (of course, it may also be because VBL has an incomplete domain theory). This feature of VBL can be used to detect errors in manually created operators. The number of such errors will, of course, depend on the extent to which the manually created operators have been debugged by testing them on sample problems. For the example set of 25 operators used in Experiment 1, VBL detected one buggy operator, namely `AND-Q1`. The error was an incorrect encoding condition on one of the internal ports of the circuit. Given larger operator sets that have not been rigorously manually debugged on sample problems, we expect that VBL will detect more erroneous operators.

Varying the Grain Size of Learned Operators The results of the above experiment show that VBL can learn operators of generality comparable to or greater than the existing handcoded operators. However, the added generality may not always be desirable. For example, one operator learned by VBL suggests implementing any boolean function `F` by refining it into a circuit consisting of ¬`F` followed by an inverter. This operator applies in many situations where there are better refinement alternatives.

One solution to this problem of "correct but frequently wasteful" operators is to learn less general operators, by embedding some of the control knowledge in the operator precondition itself. This can be achieved by augmenting the explanation of a refinement step (which so far has only focussed on correctness) to include an explanation of why the refinement step is a useful thing to do.

One interesting experiment that we carried out was to modify the target concept used in VBL from "correct refinement step" to "useful refinement step". A refinement step is considered useful if it is correct, and the submodules are non-interacting. One simple way to enforce non-interaction among submodules is to instantiate the internal encodings of a circuit to a fixed value.[18] The result of this experiment was that VBL learned a different set of operators that were less general than before (because the port encodings in example refinement steps were not generalized). This experiment, described in more detail in [27], demonstrates that LEAP can learn operators of varying generality.

[18] For the class of circuits described in this paper, submodules can interact only through the encoding on ports. However, more generally, submodules can also interact along other dimensions, such as timing.

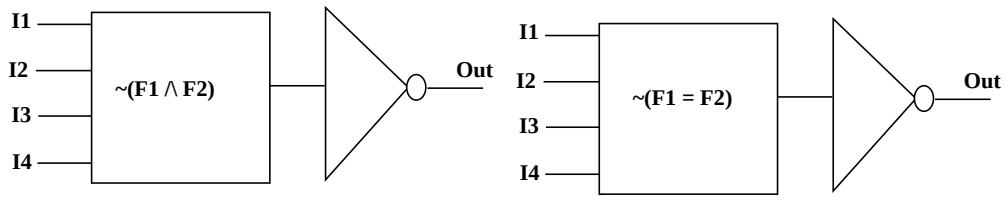


Figure 16: RHS of Two handcoded operators

general operator could have been learned had VBL formed a different explanation from the rules in its domain theory. This dependence of the generality of the learned knowledge on the particular explanation is a characteristic of most EBL techniques. Note that in all but 3 cases, the VBL learned operator has a more general `encoding` constraint than the handcoded operator. In the 3 cases where they are equivalent, the handcoded operator and the instance of it input to VBL made no specific commitment to either active or passive encoding.

Global Comparison of Handcoded Operators with Learned Operators Above we compared each handcoded operator individually with the corresponding LEAP-learned operator. It is also useful to step back and make a more global comparison of the knowledge base of handcoded and learned operators. We can do this in several ways, as described below.

- *Compaction of the Operator Set:* One way to compare the LEAP and handcoded operator sets is by counting the number of distinct LEAP operators that were learned from the training instances of handcoded operators given to VBL. In particular, VBL learned 20 distinct operators from the 25 examples described in the above experiment. An example will make this more concrete. Figure 16 describes the RHS of two different handcoded operators, `AND-NOT-ACTIVE` and `EQUAL-NOT-ACTIVE`, for implementing a conjunction and equality. When VBL is given an instance of one of these operators, it learns an operator that states that any boolean function `F` can be implemented by a circuit comprised of a module computing ¬ `F` in series with an inverter. This LEAP operator covers both handcoded operators. The LEAP operators represent a 20 percent compaction of the handcoded operators. One interesting way of viewing this compaction is that it highlights those manually created handcoded operators that implicitly contain some control information. Although these results were obtained on a small operator set, we believe that the same phenomenon will be observed for larger operator sets.

- *Hierarchical Structure of the Operator Set:* Another useful global comparison of the handcoded and LEAP-learned operator sets can be done by comparing the hierarchical structure of the operator sets based on the generality of the LHS of the operators. Figure 17 depicts the hierarchical structure of the handcoded operators described earlier. Figure 18 depicts the hierarchical structure of the LEAP operator set, again based on the generality of the LHS of an operator. Several differences are apparent. First, at level 1, a new category has emerged, namely operators that are applicable to arbitrary

an equivalent or more general LHS and RHS than the corresponding handcoded operator. Thus, extrapolating from this table, we can conclude: *A LEAP operator is usually of a* **finer grain size** *than the corresponding handcoded operator: that is, it applies to* **more** *modules, and proposes a* **smaller** *refinement step in those cases.*[17]

Handcoded Operator	Comparison with Corresponding LEAP Operator		
	Generality of LHS		Generality of RHS
	Data_value	Encoding	
AND-NOR-ACTIVE	=	>	>
AND-NOT-ACTIVE	>	>	>
AND-DECOMP	>	=	>
AND-EXPAND	=	=	=
AND-Q1	=	>	>
AND-PP	=	>	>
EQUAL-NOT-ACTIVE	>	>	>
EQUAL-OR-DECOMP	<	>	<
EQUAL-SWITCHED-8	<	>	?
EQUAL-SWITCHED-8-NOT	<	>	?
EQUAL-SWITCHED-0	<	>	?
EQUAL-SWITCHED-0-NOT	<	>	?
NEQUAL-SWITCHED-8	=	>	>
NEQUAL-SWITCHED-8-NOT	=	>	>
NEQUAL-SWITCHED-0	=	>	>
NEQUAL-SWITCHED-0-NOT	=	>	>
NOT-AND-DECOMP	<	>	?
NAND-RULE	>	>	>
NOT-DECOMP	>	>	>
NOR-DECOMP	>	>	>
OR-NAND-ACTIVE	<	>	?
OR-NOT-ACTIVE	>	>	>
OR-DECOMP	>	=	>
OR-EXPAND	=	>	>
OR-PASS	=	>	>

Table 1: Comparing handcoded operators with LEAP learned operators. A > indicates that the LEAP operator is more general than the corresponding handcoded operator. < indicates it is less general. = indicates equivalent generality. ? indicates no ordering is possible

In seven cases, the LHS of the VBL learned operator is more specific than the handcoded operator. This is because the generality of the VBL operator depends on the particular explanation or proof constructed for the training example. In all of these cases, a more

[17]A refinement step `r1` is smaller than a step `r2` if for every module that both `r1` and `r2` apply to, the circuit proposed by `r1` can be refined into the one proposed by `r2`.

6.1 Evaluation of VBL

One way to probe VBL's effectiveness in acquiring problem-reduction operators is by using it to relearn the manually created set of operators for boolean circuit design used in the VEXED design system. If VBL can learn these operators from examples, it provides a good empirical demonstration for the effectiveness of VBL as a knowledge acquisition tool.

Twenty five training examples of refinement steps were collected for the experiment, each of which was created by instantiating a different problem-reduction operator in the VEXED knowledge base. We use the term *handcoded operators* to refer to the manually created operators in the VEXED knowledge base.

Figure 17 on page 29 illustrates the structure of the handcoded operators. At the first level, operators are categorized based on the type of boolean function to which they are applicable (AND, OR etc.). At the second level, operators under a given function are discriminated by the type of encoding to which they apply (passive or active). For brevity, only the operators under the functor OR have been detailed. The leaves of the tree represent names of specific operators. To ensure that VBL was tested on a broad enough sample of the handcoded operators, we included operators from each of the function classes. Commutative variants of operators were eliminated – for example, if two operators for implementing an AND of two functions differed in the order in which they assigned the two functions to submodules, only one of them was chosen. Finally, certain classes of handcoded operators, such as iterative operators, were not selected since these cannot be learned by VBL. We describe the classes of handcoded operators not learnable by VBL in the next section.

After running VBL on these training instances, we compared the operators learned by VBL with the handcoded operators (from which each training instance was derived) along several dimensions:

- *Generality of LHS:* Is the LHS of the learned problem-reduction operator more or less general than the LHS of the handcoded operator from which it was derived? The LHS of an operator consists of constraints on two features:
  - *Data_value:* This defines the class of data value mappings to which the operator applies.
  - *Encoding:* This defines the mapping from boolean values to voltages for each of the inputs and outputs of a module.
- *Generality of RHS:* Is the circuit produced by the learned problem-reduction operator more or less general than that produced by the handcoded operator? We define a circuit `C1` as being more general than a circuit `C2` if there exists some sequence of refinements of `C1` that will produce `C2`.

The results of this experiment are summarized in Table 1, which lists the 25 handcoded operators, and compares them to LEAP operators learned by VBL from refinement steps created by instantiating the handcoded operators.[16] The results in the table can be summarized as follows. In 72 percent (18 times out of 25) of the cases, the learned operator has

[16] A detailed description of these operators is given in [27].

each nonprimitive module by an additional step. This results in a circuit whose delay can be estimated, thus enabling PED to explain this control decision. This search process can lead to exaggerated upper bounds for preferred implementation if the non-primitive modules are implemented inefficiently, which in turn lead to exaggerated lower bounds on the rejected implementations (because LEAP assumes that an upper bound on the preferred implementation must be a lower bound on the rejected implementation). Thus, this extension to the control learning algorithm is safe to use only if optimal implementations of non-primitive modules are derivable using VEXED's problem-reduction operators.

Using a limited amount of search to compute circuit cost estimates is an instance of a general technique called *rough design*, which is analyzed in great detail in Tong's thesis [59, 58].

6 Experimental Results

This section describes some experimental results on using VBL and PED in LEAP to acquire design knowledge. Before plunging into the details, we need to clarify the claims under scrutiny. In essence, we want to examine LEAP's ability to recreate design knowledge that was handcoded in the VEXED problem solver. Note that this claim is far humbler than the one we would really like to test, namely LEAP's ability to acquire sufficient expertise in a substantial class of real-world VLSI design problems.

Our inability to test this grander claim is partly because of limitations in VEXED's model of design. Basically, VEXED assumes design is a progressive refinement of circuit modules whose function can be formally specified at implementation time. However, many key aspects of real-world VLSI design, such as layout optimization, cannot be viewed in such a top-down manner.

Even within the scope of circuits implementable using VEXED, we restrict our attention to NMOS VLSI realizations of combinational boolean circuits. This additional restriction stems from some inherent limitations of the two learning techniques used in LEAP, VBL and PED, which we discuss in detail later in this section. However, we should emphasize that even this restricted class of circuit design problems is a richer domain than those previous EBL studies have investigated, such as blocks world.

In sum, the experimental results we present are not as strong as we would have liked. They demonstrate "proof of concept" of the learning apprentice approach, but are not a "real world" experiment. However, our results can be viewed as a *scaled-down* version of a more complete analysis. That is, we expect that the *form* of our results – for example, the different ways in which learned and handcoded rules can be compared – will be quite useful in future studies of the learning apprentice approach.

We return to discuss the issue of experimental validation of learning apprentices in Section 9.

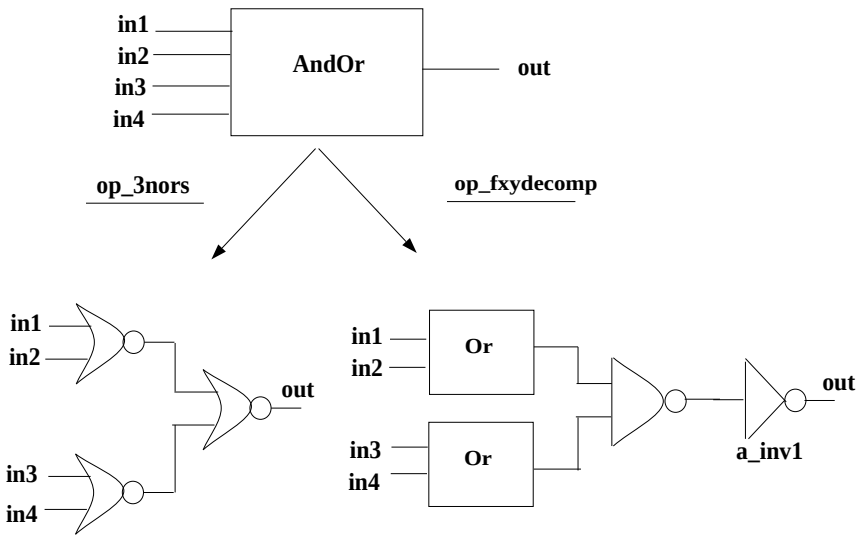


Figure 14: Using Partial Delay Information to Prune Paths

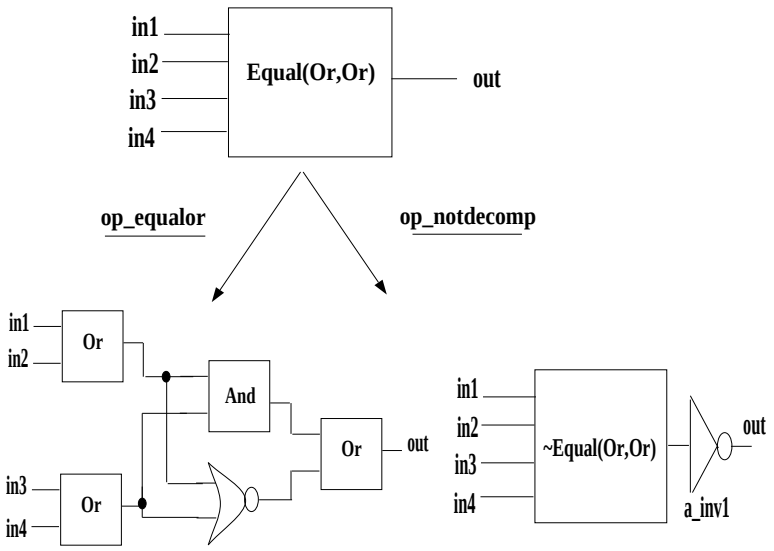


Figure 15: Using Search to Compute Estimates

spec(M1, (not (and (or X1 Y1) (or X2 Y2))), active) $\wedge$ spec(M2,(not X),active)
$\rightarrow$ rise(lb,M1,M2,4)

The first rule states that a lower bound on the fall time of a module computing the $\neg$AndOr function driving an inverter is 1 time unit, provided the module outputs are encoded in active logic. The second rule provides a corresponding lower bound for rise time.

Finally, PED adds both these rules to its theory of delay, and reexplains the training instance in order to form a general preference rule. This time, it is successfully able to explain the control decision, and returns the preference rule shown earlier in Figure 5. This rule states that if the module being implemented computes the AndOr function, and the output is encoded in active logic, then the operator `op_3nors` is to be preferred to the operator `op_notdecomp` when the design goal is to minimize delay.

Note that LEAP was only able to generalize the encoding of the module inputs from the training instance in Figure 12. Further generalization is not possible because the delay explanation is constrained by the details of the two circuit implementations, such as the fact that active logic NOR gates were used. We discuss this issue in more detail in Section 6.

5.3.2 Elaborations on the basic PED technique

We now describe two ways of dealing with cases where more than one module may have unknown cost. Both extensions have been implemented in LEAP.

Using Partial Delay Information to Prune Paths The first solution is to attempt to explain the control decision while ignoring modules whose cost is not known. Consider, for example, the control decision illustrated in Figure 14. The user in this case chose the implementation on the left (using NOR gates). If PED is given this example, it is able to explain it without knowing the delay characteristics of the `OR` modules. LEAP estimates an upper bound on the delay of the NOR gate implementation as 5 time units (for both rise and fall time). Next, it tries to show that the lower bound delay estimate for the circuit on the right is at least 5 time units. However, while doing so, it computes the lower bound delay for the partial path involving the NAND gate and the inverter to be 6 units for rise time, and 9 units for fall time, both of which already exceed the given upper bound estimate.[15]

Adding Search to Compute Cost The second solution is to use search to make additional refinements to the implementations being compared. This is necessary whenever PED cannot estimate bounds on both the implementations being compared. For example, consider the control decision illustrated in Figure 15. In this case, PED cannot estimate bounds on either of the two implementations.

In case LEAP fails to explain a particular control decision because it is unable to estimate upper bounds on the preferred implementation, then the preferred implementation is refined using the VEXED problem solver. In particular, for the example illustrated in Figure 15, the implementation suggested by `operator_equalor` is further refined by refining

[15]This is similar to the pruning that occurs in branch and bound algorithms.

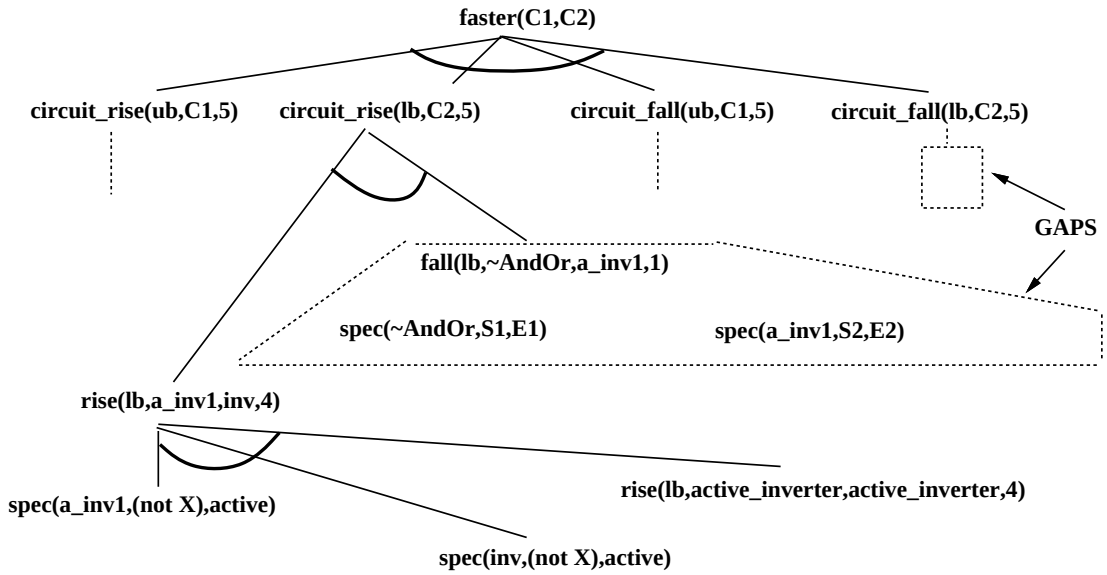


Figure 13: A Portion of the Delay Explanation

the explanation similar to the first one above (shown in Figure 13 only as a dotted empty box for brevity).

At this point, since there are no other known ways of proving that circuit `C1` is faster than `C2`, PED attempts to fill the gaps in the delay explanation in Figure 13 by assuming the correctness of the user's control decision (that is, by assuming that circuit `C1` is indeed faster than `C2`). Note that the explanation structure above is similar to that shown in Figure 11 in that there are two instantiated determinations in an explanation path from the target concept to the example description, and that all other paths have failed.

Next, using predicate completion, PED concludes that `fall(lb,¬AndOr,a_inv1,1)` and `rise(lb,¬AndOr,a_inv1,4)` (instances of the right-hand sides of the determinations representing the gaps) must follow from the training instance description and the instantiated target concept. Next, in order to apply the single instance generalization rule, PED proves the following instances of the left-hand side of the two determinations.

spec(¬AndOr,(not (and (or X1 Y1) (or X2 Y2))), active)

spec(a_inv1,(not X), active)

Here PED has computed the input output relation and encoding of the `¬AndOr` and `a_inv1` modules using the information given in the description of the control decision in Figure 12. Finally, using single instance generalization twice, PED forms the following two rules regarding the rise and fall time of the ¬AndOr module:

spec(M1, (not (and (or X1 Y1) (or X2 Y2))), active) ∧ spec(M2,(not X),active)
→ fall(lb,M1,M2,1)

```
The specification of the module And_Or is:
For all ports P, (encoding P I) = (nmos_boolean (false low) (true high))
(data_value out I) = (and (or (data_value in1 I)
                              (data_value in2 I))
                          (or (data_value in3 I)
                              (data_value in4 I)))
```

User's choice: `Prefer op_3nors to op_notdecomp if minimizing delay`

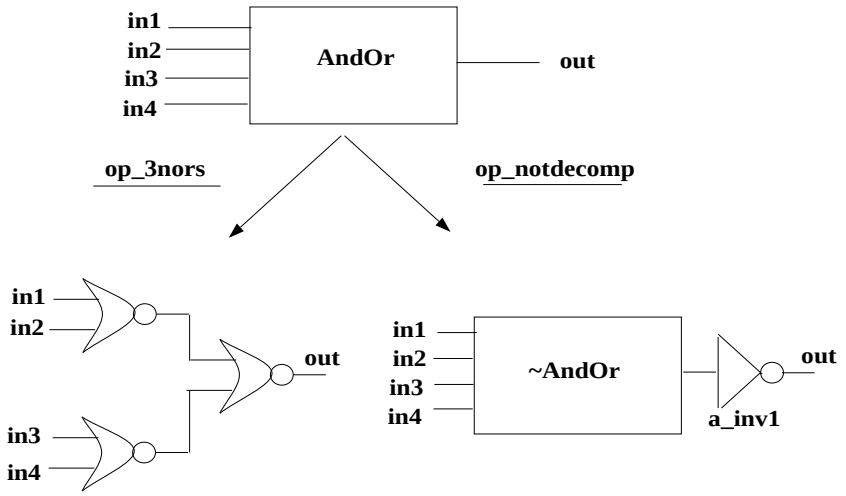


Figure 12: A manual control decision recorded by LEAP

the fall time of the ¬AndOr module driving the inverter `a_inv1`.[13]

Since PED does not know the delay characteristics of the ¬AndOr module, it cannot calculate its fall time, and so the explanation fails at this point. However, using the determination for fall time given in Figure 9, PED records a gap in the explanation as shown in Figure 13.[14] Note that the structure of the determination constituting this gap is of the form $P(a, X) \succ Q(a, c)$, exactly of the form described in Step 2 of the PED algorithm in Figure 10.

PED turns to compute the fall time of circuit `C1`, which also turns out to be 5 time units. Then, PED tries to show that the fall time of circuit `C2` is lower bounded by 5 time units. This step fails for the same reason as above, that is, PED does not know the delay characteristics of the ¬AndOr module. However, PED is able to construct a second gap in

[13]The computation of circuit delay switches from rise time to fall time (and vice versa) any time a module inverts its inputs (that is, a low to high signal transition in the module input produces an opposite signal transition in the module output).

[14]In the description of PED we separated the explanation and gap detection phases into two separate steps (Steps 1 and 2 in Figure 10). However, in practice it is more efficient to merge these two steps.

conclude that both A and B must be true. Similarly, we can also conclude that $Q1$ and Qn must be true since all other definitions of A and B fail.

If Step 2 is successful, then Step 3 tries to prove instances of the left-hand side of each of the determinations $P_i \succ Q_i$ in the path computed in Step 2 from the training example description. If Step 3 is successful, then Step 4 applies the single instance generalization rule and adds an implicative rule derived from each determination $P_i \succ Q_i$ to the implicative portion of the domain theory $T_{\Rightarrow}$. Finally, Step 1 is reinvoked to see if the training example can be shown to satisfy the target concept using the augmented domain theory.

The new rules added to the domain theory by PED logically follow from the training example description, the target concept instance, the domain theory, and the particular closed world assumption described above. A detailed analysis of PED, including its implementation in PROLOG, can be found in [26, 27, 28].

5.3.1 Learning Preference Rules In Circuit Design

We illustrate how PED can be used to learn preference rules in LEAP using an example. Figure 12 shows a control decision pertaining to two ways of implementing a particular boolean function. One way is to use a set of 3 NOR gates using the operator `op_3nors`. The second way, recommended by operator `op_notdecomp`, is to decompose the boolean function into a module computing the negation of the function, and invert its output. The user has selected the `op_3nors` as being preferable to `op_notdecomp` given that the design goal is to minimize delay.

The key idea behind the use of PED in LEAP is the following. Assuming that LEAP is only given delay estimates of primitive components (NAND gates, NOR gates, and inverters), LEAP cannot explain why the NOR gate implementation is better than the ¬AndOr implementation since it cannot calculate the delay characteristics of the ¬AndOr module. However, by assuming the correctness of the user's control decision, PED can learn new rules characterizing the rise and fall times of the ¬AndOr module. Using these new rules, PED can successfully explain the example, and create a new preference rule.

Figure 13 details a portion of the explanation structure constructed by PED to explain the user's control decision in Figure 12. A broken line indicates steps in the explanation that have been omitted for brevity. The predicate `faster` defines a circuit `C1` as being faster than a circuit `C2` if an upper bound (ub) on rise time of `C1` can be shown to be a lower bound (lb) on the rise time of `C2`, and similarly for fall time. Here, `C1` is the circuit on the left in Figure 12 composed of three NOR gates. `C2` is the circuit on the right composed of an inverter `a_inv1` and the ¬`AndOr` module.

PED is able to successfully compute an upper bound (5 time units) on the rise time of the NOR gate circuit `C1` because it knows the delay characteristics of NOR gates (this computation is not shown for brevity). Next PED attempts to show that the rise time of circuit `C2` is at least 5 time units. First, PED estimates the rise time of module `a_inv1` (an inverter) driving the default load `inv` (another inverter) to be at least 4 time units, using known facts about NMOS active logic inverter delays. This leaves 1 time unit remaining for

The interesting part is when Step 1 fails. Step 2 looks for explanation paths containing determinations that link the target concept with the example description. Each such determination represents a gap in the theory that if filled may allow the training instance to be explained. Gaps are filled by extracting implicative rules from the determinations using the single instance generalization rule. However, this requires proving instantiations of the left-hand side P and right-hand side Q of a determination $P \succ Q$ from the instantiated target concept and the training example description. Showing that P follows from the example description is straightforward, since such a path has been shown to exist in Step 2. However, showing that Q holds is difficult because the path linking the instantiated target concept C' to Q is a forward inference path from Q to C', and not vice versa.

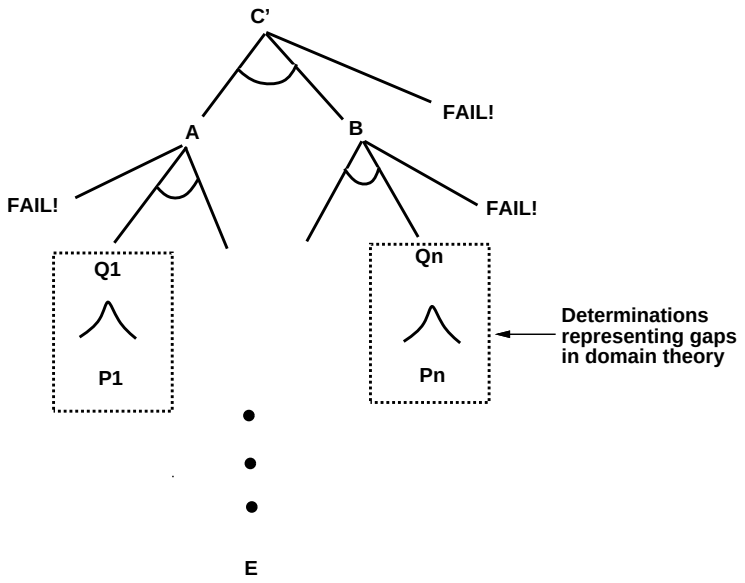


Figure 11: Filling Gaps in Determination-based Theories

Thus, Step 2 looks for determinations $P \succ Q$ that occur on a path leading from the instantiated concept C' to the example E, such that all other paths not involving these determinations can be shown to fail for the particular example E. Figure 11 illustrates such a path. Here an arc under a node crossing a set of links represents a conjunction of those links. Links under a node not crossed by an arc represent alternative explanation paths. In the figure, all alternative explanation paths (except the one containing the determinations) of C', A, and B fail.

If such a path exists, then an appropriately instantiated form of the right-hand side of each such determination Q' can be concluded from C' under a certain type of closed world assumption. This involves using predicate completion [24], a procedure for adding "only-if" rules to a logic program containing only "if" rules. In a simplified form, it asserts that the disjunction of a set of sufficient conditions for a predicate $P \leftarrow C_1, \ldots, P \leftarrow C_n$ is also a necessary condition for P. For the example path illustrated in Figure 11, since C' is known to be true, and all other paths except the one shown fail, using predicate completion we can

Inputs: A domain theory T consisting of a Horn clause theory $T_{\Rightarrow}$ and a set of determinations $T_{\succ}$, a target concept C (a predicate), an example description E (list of ground predicates), and an operationality condition O (list of predicates).

Outputs: A set of implicative rules R derived from some of the determinations in $T_{\succ}$, and an operational sufficient condition S of C, such that $E \Rightarrow S'$ and $S' \Rightarrow C'$ hold in the augmented Horn theory $T_{\Rightarrow} + R$, but not in the original Horn theory $T_{\Rightarrow}$. S' and C' are instantiations of S and C from the example E.

Method:

1. **Step 1:** Show that the example E is an instance of concept C using only $T_{\Rightarrow}$. If successful, use explanation-based learning to return an operational sufficient condition S of concept C.

2. **Step 2:** Otherwise compute an explanation path from the instantiated concept C' to the example description E that contains a set of instantiated determinations $P_i(a_i, Y_i) \succ Q_i(a_i, c_i)$ such that all other paths leading from C' to E can be provably shown to fail.

3. **Step 3:** If Step 2 is successful, then using predicate completion $Q_i(a_i, c_i)$ (for every i) must follow from the example description E and the instantiated target concept C'. Next, show that an instance of the LHS of the determination $P_i(a_i, b_i)$ (for every i) follows from the example description E using the Horn clause theory $T_{\Rightarrow}$.

4. **Step 4:** If Step 3 is successful, use the single instance generalization rule and add the rule $P_i(X_i, b_i) \rightarrow Q_i(X_i, c_i)$ (for every i) to the Horn clause domain theory $T_{\Rightarrow}$. Otherwise, fail.

5. **Step 5:** Go to Step 1.

Figure 10: The PED procedure

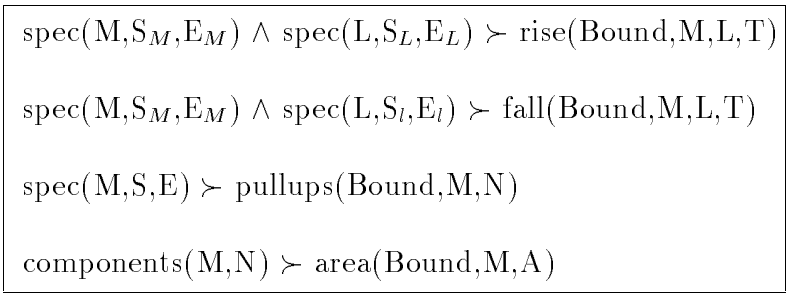


Figure 9: Example Determinations for Estimating Circuit Cost

pullups N in a module M is determined by the module specification (S,E). The number of pullup transistors of different ratios in the circuit is a rough measure of the (static) power consumed by an active logic NMOS circuit. Finally, the last determination states that bounds on the area A of a module M is determined by the number of components N in M. In other words, it expresses a correlation between area and the number of (primitive) components in the circuit.

Note that the above determinations express approximate rules that may be wrong in some situations. For example, the third and fourth determinations ignore power and area consumed by routing. The accuracy of these rules can be improved by adding more conditions. For example, we can make the area rule more accurate by including a factor that approximates routing area (such as number of inputs and outputs of a module). Furthermore, the purpose of a cost theory is not to accurately estimate the cost of a given circuit, but rather to discriminate between alternative circuits. This is why the above determinations specify bounds on the actual values.

5.3 The PED Technique

Since the circuit cost domain theory used by LEAP contains determinations, a generic implementation of explanation-based learning (such as PROLEARN [47] or PROLOG-EBG [19]) cannot directly be used. We now describe an extension of such implementations of explanation-based learning to theories containing determinations, namely the PED technique.

Let C be a target concept and E be an instance of C.[11] Let the domain theory T be composed of a Horn clause[12] theory $T_{\Rightarrow}$ and a set of determinations $T_{\succ}$. We are interested in situations when the fact that E is an instance of C cannot be proved using the rules in $T_{\Rightarrow}$.

Figure 10 describes the PED procedure in algorithmic form. Step 1 tries to prove that the example satisfies the target concept definition, and if successful, uses standard EBL [35] to return an operational sufficient condition of the concept.

[11] For simplicity, we are ignoring the arguments of each predicate.

[12] A Horn clause is an expression of the form $H \leftarrow B_1, \ldots B_n$, where H and B_i are positive (that is, not negated) atomic predicates.

- A standard (2 input) NMOS active logic NAND gate driving a standard NMOS active logic inverter has a rise time of 8 time units and a fall time of 2 time units.
- A standard (2 input) NMOS active logic NOR gate driving another standard NMOS active logic NOR gate has a rise time of 4 time units and a fall time of 1 time unit.

The above set of rules can be continually extended to cover increasingly abstract modules, by "backing up" the known delay values of more primitive components to cover more abstract components. Ideally, LEAP could be given a "base" set of delay rules for primitive components, and be made to learn the rest from examples of control decisions provided by its users. We describe below a generic form of knowledge called determinations that facilitates this process.

5.2 A Determination-based Theory of Circuit Cost

Determinations are a form of relevance knowledge [6]. Intuitively, an attribute P determines attribute Q if knowing that P holds for some object tells us something about whether Q holds for that object. A more precise definition is as follows:[9]

Definition 1 *Let P(x,y) and Q(x,z) be any two first-order sentences, where x represents the set of variables that occur free in both P and Q, while y and z represent the set of free variables that occur only in P and Q respectively. We say P(x,y) totally determines Q(x,z), or P(x,y) $\succ$ Q(x,z), iff*

$$\forall y, z[[\exists x P(x, y) \wedge Q(x, z)] \Rightarrow \forall x[P(x, y) \Rightarrow Q(x, z)]]$$

For example, let $P(x, y)$ denote $Nationality(x, y)$, meaning that individual x has nationality y. Also let $Q(x, z)$ denote $Language(x, z)$, meaning that x speaks language z. Then, the above definition states that if there exists an individual x whose nationality is y, and who speaks a language z, then all individuals of nationality y speak language z. One useful property of determinations is the single instance generalization rule. That is, from $P(x, y) \succ Q(x, z)$ and $P(A, B) \wedge Q(A, C)$ the implication $\forall x P(x, B) \rightarrow Q(x, C)$ follows.

Using Determinations to Estimate Circuit Cost Determinations are useful in expressing correlations between attributes that are known at abstract levels in the design process and circuit cost metrics that can be accurately estimated only at lower levels. Several examples of such determinations are given in Figure 9.[10] S denotes the input output relation of a module, and E denotes the encoding on its output port.

The first determination states that bounds (lower or upper, instantiated by the variable Bound) on the rise time T of a module M are determined by the specification (S_M,E_M) of M and the specification (S_L,E_L) of its load L. The second determination expresses a similar relationship for fall time. The third determination states that bounds on the number of

[9] See Russell's thesis [51] for a description of other types of determination.

[10] Capitalized arguments of predicates represent variables.

Final Output: Step 3 finally results in the problem-reduction operator shown earlier in Figure 4. Note that this operator is more general than the example refinement step (shown in Figure 6) in two respects: first, the operator applies to conjunctions of any two (binary) boolean functions, whereas the example is restricted to a conjunction of two disjunctions; second, the operator applies to arbitrary encodings (except that ports connected by a wire must have the same encoding), whereas the example requires positive encoding; finally, port names, such as `In1`, in the operator are variablized, whereas port names in the refinement step, such as `in1`, are constants.

In summary, VBL generalizes refinement steps into problem-reduction operators using three steps: the **compose** step determines the specification of the circuit proposed by the refinement step; the **verify** step proves that the circuit specification satisfies the original module specification; and finally, the **generalize-and-simplify** step creates a general problem-reduction operator that is consistent with the training example, and which satisfies the operationality criterion. A detailed experimental analysis of VBL is given in Section 6.

5 Learning Preference Rules for Low Cost Circuit Design

LEAP incorporates a second technique called PED to learn control knowledge for choosing among competing problem-reduction operators. PED is an extension of explanation-based learning to theories containing a form of partial knowledge called determinations [6]. First we motivate the need to extend earlier techniques for learning control knowledge that are based on explanation-based learning [33, 42, 22].

5.1 The Control Knowledge Problem in LEAP

One popular approach to learning preference rules is based on explaining instances of control decisions using a domain theory. However, this approach is not possible in LEAP because its circuit cost theory is incomplete. This is because cost metrics can only be estimated accurately once a module has been completely implemented. For example, delay involves computing rise and fall times. These are determined by the amount of current the driving module can source and sink, and the impedance of the load module, both of which can be determined accurately only at the most detailed implementation level.[7]

One solution is to assume that the rise and fall time of all primitive components are known, and can be expressed as a set of rules of the following form:

- A standard NMOS active logic inverter driving another standard NMOS active logic inverter has a rise time of 4 time units and a fall time of 1 time unit.[8]

[7] Actually, even then one can only approximately estimate certain factors such as parasitic capacitance.

[8] The time unit depends on the particular technology being used. For NMOS VLSI, it is a nanosecond [31].

requirement, considerable simplification of the leaves constituting the "frontier" of the proof tree is often necessary. The details of this simplification process are given in [27]. Briefly, in the PROLOG implementation of VBL, problem-reduction operators for boolean circuit design are represented as PROLOG clauses:

```
implements(operator,module,circuit) ← conditions
```

Here, `operator` denotes a particular learned operator, `module` denotes a module specification, `circuit` denotes a refinement of the module into a circuit, and `conditions` denotes a conjunction of constraints on the variables in the head of the clause. Prior to simplification, the PROLOG clause representing a problem-reduction operator output by VBL could have a list of `conditions` tens of pages long! The simplification consists of applying a sequence of truth-preserving transformations to the expressions in the body of the PROLOG clause representing a reduction operator learned by VBL. Transformations include rules for evaluating expressions and propagating resulting values across the clause, eliminating tautologies etc. Here is an example of a simplification rule in VBL:

```
simplify(X is A*B + C,true) :-
    integer(A),
    integer(B),
    integer(C), !,
    call(X is A*B + C). % evaluate expression using PROLOG interpreter
```

The first argument of the predicate `simplify` is the expression to be simplified. The second argument is the result of the simplification. Basically, this rule takes an arithmetic expression of the given form, evaluates it, and propagates the value of the variable `X` globally.

Reordering: After simplification, the next step is to reorder the leaves of the explanation structure such that they can be executed linearly by the PROLOG interpreter. Otherwise, certain goals in the body of the learned operator cannot be executed because they involve calls to predicates with unbound arguments. It is important to note that often predicates in PROLOG are not strictly logical predicates, in that certain arguments have to be instantiated for the predicate to return a value. This problem is most transparent with arithmetic predicates, such as `X > Y`, and with predicates that access data structures, such as `functor(Term,Functor,N)` where the term `Term` has `Functor` as its outermost function symbol of `N` arguments.

The basic strategy behind the reordering is as follows. The input to the reorderer is a list of predicates in the body of the clause. At every step, the reorderer searches for the first predicate in the body that be evaluated using a list of variables which are known to be bound at that point. The reorderer has information regarding the inputs and outputs of every predicate. If the predicate can be evaluated, then its outputs, which will be bound when it is executed, can be added to the list of bound variables. Given the updated list, the reorderer repeats this process.

shown earlier in Step 1. As a second example, from the following operating condition on the port `nor2_in1`

```
(encoding nor2_in1 I) = (nmos_boolean (false low) (true high))
```

and the fact that `nor2_in1` is connected to port `in1` (see Figure 6), the below composed specification can be obtained.

```
(encoding in1 I) = (nmos_boolean (false low) (true high))
```

Verification of this specification is also easy since the exact same assertion appears in the module specification itself.

Although the verification step for our example is straightforward, for larger circuits it can be quite complex involving tens of steps, and dozens of conditions that need to be checked. The complete set of rewrite rules used by VBL (given in [27]) contains about fifty rules. To keep the verification tractable, VBL exploits the directional nature of term rewrite rules. In particular, if the set of term rewrite rules needed in a verification has the property that they always reduce the size of an expression (where size can be defined in several ways, such as depth), then verification using term rewrite rules is linear in the size of the expression. However, in general, VBL also uses rules for checking commutativity and associativity, which are not size reducing, and have to be handled specially. Also, in the worst case, boolean circuit verification could involve an exponential number of steps because the expressions may be exponentially long (in the number of module inputs).

We note in passing that term rewrite systems are particularly suited to representing simplification rules in formal domains. Laird and Gamble [23] describe a general explanation-based learning algorithm for term rewrite systems. Huet and Oppen [17] present a comprehensive survey of term rewrite systems.

Step 3: Generalization and Simplification The goal of step 3 is to create a general problem-reduction operator from the verification generated in the previous step. This is achieved by creating a generalized proof structure that mirrors the structure of the proof of the example, and forming a "macro" of the generalized proof structure by computing constraints on the variables in the proof. This is similar to the generalization step in PROLOG-based implementations of explanation-based learning, such as PROLOG-EBG [19] and PROLEARN [47]. However, both PROLOG-EBG and PROLEARN do not incorporate any general simplification rules for compressing (or otherwise transforming) the resulting generalization. As we will see below, simplification is an important component of VBL and is incorporated as part of the operationality criterion in VBL.

Simplification: The *operationality criterion* [35] is a constraint on the generalization step. Here it states that it should be possible to test the applicability of a learned problem-reduction operator to a circuit module "efficiently", that is by unifying the IF condition of the operator with the module specification. In other words, checking whether an operator is applicable should not require rewriting or any other form of reasoning. To satisfy this

Step 2: Verify: Verifying that the composed specifications computed in step 1 meet the original module specification is done either by showing that a particular composed specification is logically equivalent to a particular original module specification, or by rewriting a particular composed specification to a tautology. We illustrate each of these methods below. For the `data value` feature, the following assertion needs to be proved:

```
(data_value out I) =
     (not (or (not (or (data_value in1 I) (data_value in2 I)))
              (not (or (data_value in3 I) (data_value in4 I)))))
     ⇔
(data_value out I) = (and (or (data_value in1 I) (data_value in2 I))
                          (or (data_value in3 I) (data_value in4 I)))
```

Such equivalences are proved in VBL by simplifying both the antecedent and the consequent expressions into a common form using rewrite rules. Two rules that are useful for the current example are shown below:

De Morgan's Law: `(or (not X) (not Y))` → `(not (and X Y))`

Double Negation Rule: `(not (not X))` → `X`

Figure 8 illustrates the application of the above rewrite rules to prove that the circuit satisfies the `data_value` specification of the module. In general, verification proofs constructed by VBL are sequences of rewrites, one sequence for each feature (such as `data_value` and `encoding`). The proof in Figure 8 illustrates a special case where the circuit specification can be rewritten into a form that matches the module specification. More generally, the two specifications are rewritten into a common intermediate form.

```
(not (or (not (or (data_value in1 I) (data_value in2 I)))
         (not (or (data_value in3 I) (data_value in4 I)))))

                ⇓        DeMorgan's law

(not (not (and (or (data_value in1 I) (data_value in2 I))
               (or (data_value in3 I) (data_value in4 I)))))

                ⇓        Double Negation rule

(and (or (data_value in1 I) (data_value in2 I))
     (or (data_value in3 I) (data_value in4 I)))
```

Figure 8: Part of the verification for the example refinement step

To complete the verification, the operating conditions of the circuit must also be shown to meet those of the original module. This is trivial for the composed encoding expression

1. **Compose:** Given a module specification `P`, and its refinement into a circuit `C`, compute the specification `Pc` of circuit `C` from the specifications of each submodule in `C` and the circuit connectivity.

2. **Verify:** Prove that the circuit specification `Pc` satisfies the module specification `P`.

3. **Generalize-and-Simplify:** Form a general problem-reduction operator from the verification that satisfies the operationality criterion that testing the applicability of the operator to a circuit module should be efficient.

Figure 7: The VBL Technique

2. If port `p` is the output port of a submodule `M`, substitute `E` for all occurrences of `(data_value p I)` if the specification of `M` contains the constraint `(data_value p I) = E`. (A similar rule can be stated for `encoding`.)

Carrying out the above steps for the circuit in Figure 6, we obtain the following expression for the `data_value` condition:

```
(data_value out I) =
     (not (or (not (or (data_value in1 I) (data_value in2 I)))
              (not (or (data_value in3 I) (data_value in4 I))))))
```

VBL deals with operating conditions (specified here by the `encoding` conditions) by similarly substituting for internal port references, and obtaining derived conditions. For example, from the constraints

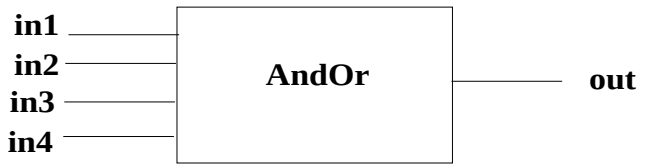
```
(encoding nor1_in1 I) = (nmos_boolean (false low) (true high))
(encoding nor2_out I) = (nmos_boolean (false low) (true high))
```

and given that the port `nor1_in1` is connected to port `nor2_out`, applying the above rules results in the following (trivially true) derived condition:

```
(nmos_boolean (false low) (true high)) =

(nmos_boolean (false low)(true high))
```

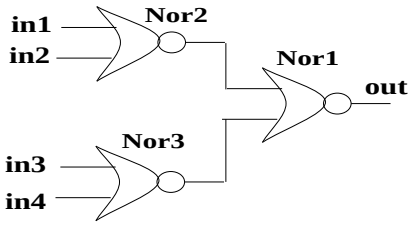
A similar expression can be derived starting from the encoding constraint on the second input port `nor1_in2` of module `Nor1`.



Specification of the module:

(data_value out I) = (and (or (data_value in1 I) (data_value in2 I))
(or (data_value in3 I) (data_value in4 I)))
(encoding in_j I) = (nmos_boolean (false low) (true high)) $(1 \leq j \leq 4)$
(encoding out I) = (nmos_boolean (false low) (true high))

Refinement of the module:



where each NOR gate is specified as follows:

(data_value nor_k_out I) = (not (or (data_value nor_k_in1 I)
(data_value nor_k_in2 I)))
(encoding nor_k_in1 I) = (nmos_boolean (false low) (true high))
(encoding nor_k_in2 I) = (nmos_boolean (false low) (true high))
(encoding nor_k_out I) = (nmos_boolean (false low) (true high))
$(1 \leq k \leq 3)$

Figure 6: A manual refinement step recorded by LEAP

The first two steps correspond to the explanation step in explanation-based learning, and the third step corresponds to the generalization step. We illustrate each of these steps for the example refinement in Figure 6.

Step 1: Compose: The composed specification of the overall circuit is obtained by repeatedly substituting expressions for port references in the `data_value` and `encoding` expression of submodule `Nor1` (the outermost submodule in the circuit) until we obtain a resultant expression in terms of the ports `in1` to `in4` of the `AndOr` module. Two rules guide the substitution process:

1. If input port `p1` of one module is connected to output port `p2` of a different module, substitute `(data_value p2 I)` for all occurrences of `(data_value p1 I)`. (A similar rule can be stated for `encoding`.)

```
IF: The specification of the module is:
(data_value Out I) = (and (or (data_value A I)
                              (data_value B I))
                          (or (data_value C I)
                              (data_value D I)))
(encoding Out I) = (nmos_boolean (false low) (true high))

And the design goal is minimum delay

THEN: Prefer op_3nors to op_notdecomp
```

Figure 5: A Preference Rule Learned by LEAP

the Lisp implementation of VEXED did not allow for control knowledge to be added in the form of preference rules.

A second implementation of LEAP was carried out in Prolog (using Quintus Prolog on a Sun workstation) [27]. There were several reasons for doing a Prolog implementation. One, the learning techniques could be more rapidly developed and tested, essentially by focusing on a smaller subset of circuits (only boolean circuits) and a smaller set of circuit descriptors (only `data_value` and `encoding`) than that in the Lisp version. Two, new capabilities, such as control knowledge in the form of preference rules, could be added to the system relatively painlessly, by eliminating from the Prolog implementation many of the "bells and whistles" in the Lisp version. Finally, both VBL and PED could be

implemented by extending PROLEARN, a generic implementation of explanation-based learning in Prolog [47]. The experimental results described in Section 6 were obtained using this implementation.

Thus, in terms of its PROLOG implementation, LEAP is basically a research demonstration system, which has so far been tested on small boolean circuits with less than a couple of dozen components.

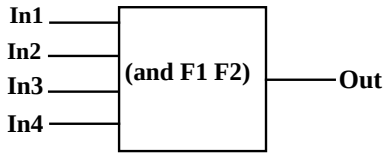
4 Learning Operators for Boolean Circuit Design

This section describes how general problem-reduction operators for boolean circuit design can be learned from specific refinement steps using a technique called VBL [25]. Figure 6 illustrates a specific refinement step recorded by LEAP from which it learned the general problem-reduction operator shown earlier in Figure 4. We will use this as a running example to illustrate the steps in VBL. This refinement step decomposes a module computing a conjunction (AND) of disjunctions (OR) into a circuit comprised of three `nor` gates. The encodings of all the ports, including inputs, output, and ports internal to the circuit, are specified as `active` (in other words, `true` is mapped to `high` and `false` is mapped to `low`).

Figure 7 summarizes the main steps in VBL. The structure of VBL – "verify, then generalize based on the verification" – is an instance of explanation-based learning [9, 35].

Operator AND-NOR-ACTIVE:

IF: The module being implemented is of the form:



where its specification is:

```
(data_value Out I) = (and (F1 (data_value In1 I) (data_value In2 I))
                          (F2 (data_value In1 I) (data_value In2 I))
```

THEN: Refine it into the following circuit:

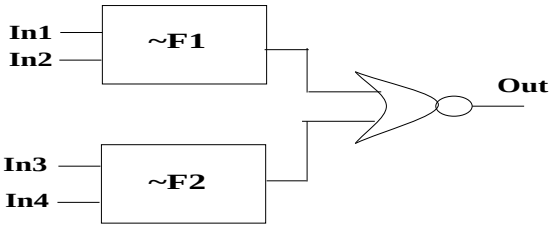


Figure 4: Problem-Reduction Operator Learned by LEAP

occurs when LEAP suggests multiple ways of implementing a particular module, and the user selects one of these alternatives. In this case, the system is getting an example of control knowledge for selecting among competing operators. Corresponding to these two learning scenarios, the learning element in LEAP is comprised of two techniques, VBL and PED, for generalizing user-provided examples into general design knowledge. VBL is a technique for learning general problem-reduction operators for circuit design from particular refinement steps. PED is a technique for learning control knowledge in the form of preference rules from specific advice about which operator to apply to implement a circuit module. Section 4 and Section 5 describe these techniques in detail.

3.4 The User

As described above, a user simultaneously controls the problem-solving process in VEXED, and supplies LEAP with examples from which to learn. One caveat that we must mention at this point is that LEAP has not had any *real* users, apart from its designers.

3.5 Implementation Status

Two separate implementations of LEAP have been developed, one in Lisp and the other in Prolog. The Lisp implementation was built upon an existing implementation of the VEXED problem solver [37], which incorporated a symbolic constraint propagation system called CRITTER [20]. Only the VBL technique was implemented in the Lisp version [36]. Also,

Specification of the module:

Input/output relation:
(data_value out I) = (and (or (data_value in1 I) (data_value in2 I))
(or (data_value in3 I) (data_value in4 I)))

Operating Conditions:
(encoding out I) = (nmos_boolean (false low) (true high))
(encoding in_j I) = (nmos_boolean (false low) (true high)) $(1 \leq j \leq 4)$

Figure 3: Specification of a Module in LEAP

knowledge [37]: Given two alternative refinements of a particular module, which one should be preferred? Tradeoff knowledge of this form can be represented in the form of *preference rules* [22, 33, 42].

Figure 5 illustrates a preference rule learned by LEAP. The IF part of a preference rule consists of a module specification, and a design goal. The THEN part specifies a preference for one operator over another. The particular preference rule described in the figure ranks an operator proposing a NOR gate implementation (`op_3nors`) of a conjunction over one proposing a different decomposition (`op_notdecomp`), when the design goal is to minimize delay. In Section 5 we describe how LEAP learns this preference rule from a control decision provided by a user.

Analysis Knowledge Several forms of analysis knowledge are used in LEAP to compute properties of a given, possibly partial design. *Symbolic simulation knowledge* is used in computing an overall specification of a circuit from the specifications of its components and the circuit connectivity. *Verification knowledge* is used to decide whether a given circuit meets its specifications. Finally, *circuit cost knowledge* is used to estimate the cost of a given circuit implementation. Examples of analysis knowledge are given in Section 4.

3.3 The Learning Element

Two opportunities for learning arise in LEAP. The first is when it is unable to refine a particular module, or when the user is unhappy with any of the refinements it suggests. In either of these situations, the user can manually refine the module by using a circuit editor, and create an implementation of the module. This provides the system with a training example of a problem-reduction operator that it lacks. The second opportunity for learning

also partially orders the set of applicable operators using control knowledge in the form of preference rules. The user selects the best operator (or picks one of the best operators), which the system then applies, refining the module into a set of interconnected submodules. Each operator application results in a module being refined into a set of submodules, and a new set of constraints on the interfaces between the submodules. These constraints are propagated to other parts of the circuit by a program called CRITTER [20]. Each submodule represents a new design subproblem, one that is (possibly) simpler than the original problem. Problem-solving activity terminates when all the remaining submodules have been refined into primitive components, and all the interface constraints between modules have been satisfied.

3.2 The Knowledge Base

The knowledge base in LEAP is comprised of the following parts.

Implementation Knowledge Implementation knowledge consists of a set of problem-reduction operators that represent legal ways of decomposing a circuit module. A module is described by its functional specification, which consists of a set of *input/output relations* and a set of *operating conditions*. Input/output relations specify the functionality of a module, whereas operating conditions specify the set of input signals for which the module must compute this function.

Figure 3 illustrates this style of representation. In this example, the input/output relation is specified by the `data_value` condition, which specifies that the value of the signal at time `I` on the `out` port of the module is a particular boolean function of the values of the signals on the input ports `in1` through `in4`. The operating conditions are specified by the `encoding` conditions on the four inputs `in1` through `in4`. For example, the encoding of the input `in1` is specified as being in *active logic*, that is boolean *true* is mapped to a *high* signal, and boolean *false* is mapped to a *low* signal. In general, module specifications also include timing, phase, and data type constraints. In this paper we will consider only `data_value` and `encoding` constraints, as the learning techniques were only been tested on these.

Figure 4 describes a problem-reduction operator learned by LEAP for implementing a conjunction of two boolean functions. The IF component of a problem-reduction operator represents the class of module specifications to which it applies. The THEN component proposes a particular circuit implementation of the module described by the IF component. The operator in Figure 4 suggests implementing the conjunction (AND) of two binary boolean functions by implementing the negation of each boolean function, and combining their outputs using a NOR gate.[6] In Section 4 we will describe how this operator was learned from a particular refinement step provided by a user.

Control Knowledge Problem-reduction operators for circuit design define the "legal moves" in the search space of possible implementations. Control knowledge is needed to make the search for an implementation tractable. One kind of control knowledge is *tradeoff*

[6] Capitalized names in Figure 4, such as `I`, `In1` and `F1`, represent variables.

degree that the learning methods are explanation-based analytical methods, the apprentice can accommodate incorrect or inconsistent training examples by simply ignoring training data that is inconsistent with its initial knowledge base (for example, by rejecting user-provided circuit designs that do not appear to correctly implement the desired function). However, when inductive learning methods are required, errors in user decisions can well lead to misguided learning.

3 LEAP: A Learning Apprentice for Digital Circuit Design

So far, we have discussed learning apprentices in abstract terms. We now describe a particular learning apprentice system for digital circuit design called LEAP. Figure 2 illustrates the main components of LEAP. We describe each of these components in detail below.

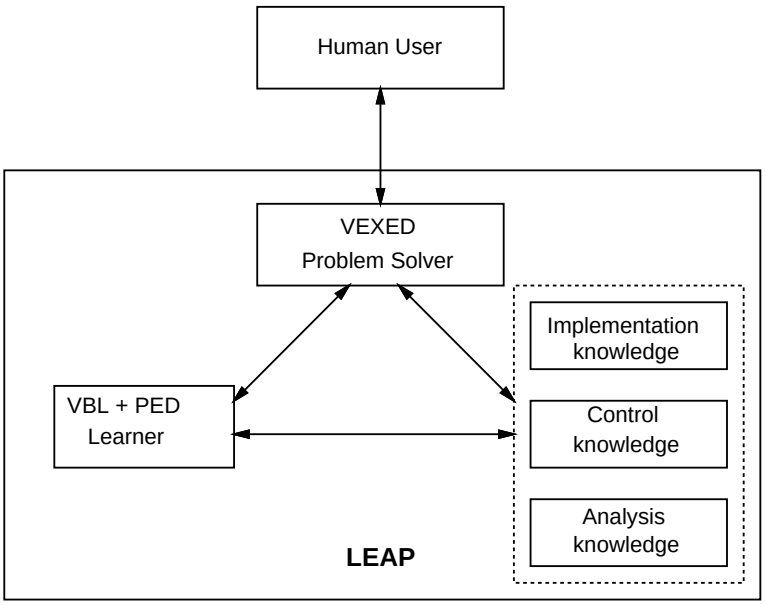


Figure 2: Structure of the LEAP Apprentice

3.1 The Problem Solver

The problem solver in LEAP is called VEXED.[5] It is based on a model of circuit design as top-down refinement with constraint propagation used to avoid interacting subproblems [37, 56]. Problem-solving in VEXED starts with a user supplying the system with the specifications of a generic circuit module. VEXED then displays to the user all the problem-reduction operators in the knowledge base whose preconditions match the module specification. VEXED

[5] An acronym for Vlsi EXpert EDitor.

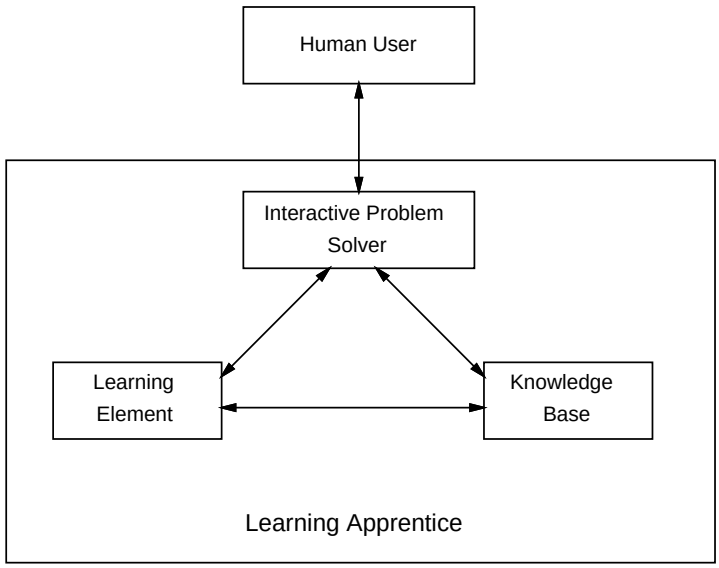


Figure 1: Model of a Learning Apprentice System

- *Explanation is easier than problem solving.* An apprentice has to only explain the correctness of the observed user's solution. It need not be capable of generating these solutions on its own. As has been pointed out before [8], explaining a given solution is often easier than generating that solution.

- *Finer-grained training examples are available.* A user typically solves problems in multiple steps, such as decomposing a circuit module into submodules, then subsubmodules, etc. Learning rules from such fine-grained training examples is easier than learning from a case library of problem solution pairs that represent only the final and initial problem solving state (for example, pairs of completely designed circuits and their corresponding functions, with no intermediate decomposition).

- *Training instances of control decisions are easily available.* A user tells the apprentice which operator to apply in cases where the apprentice is unable to select among the alternatives. In contrast, autonomous learning programs that acquire control knowledge on their own must themselves conduct an extensive search in order to obtain training examples of control decisions.

There are also limitations and difficulties to organizing a knowledge acquisition system based on a learning apprentice. First, an apprentice requires the assistance of humans who are experts (or at least, more knowledgeable than the apprentice) in the particular domain. Second, the behavior of the apprentice should not impose additional demands on the user in order to allow it to learn – from the user's perspective, the system's interface should be no different than if it were simply assisting in solving problems without learning. Third, there is a problem of knowledge integration, in that different users may provide conflicting advice to the apprentice, and potentially cause the apprentice to learn inconsistent information. To the

suggesting feasible decompositions of abstract circuit modules). PED[4] is an extension of explanation-based learning to domain theories containing determinations, a form of partial knowledge [6]. LEAP uses PED to learn preference rules for chosing among competing reduction operators.

Both VBL and PED were designed to take advantage of the learning apprentice context. VBL uses a circuit verification theory that can efficiently explain the correctness of a given circuit refinement step, even though it is too weak to efficiently generate useful circuit refinements. Likewise, PED relies on control decisions collected from LEAP's users to fill gaps in a partial theory dealing with circuit cost (i.e., circuit area, power, speed). LEAP subsequently uses its enhanced cost theory to learn control rules from decisions that it otherwise could not have generalized.

We present experimental results on the effectiveness of VBL and PED in acquiring design knowledge. The results show that VBL can relearn a significant portion of a manually created knowledge-base of operators for boolean circuit design. They also show that PED can refine gaps in a partial theory of circuit cost, and simultaneously learn new control rules from control decisions made by the user.

This paper is organized as follows. Section 2 describes a generic model for learning apprentices. Section 3 describes the LEAP learning apprentice in detail. The two learning techniques used in LEAP, VBL and PED, are described in Section 4 and Section 5, respectively. Experimental results demonstrating their effectiveness is presented in Section 6. Section 7 analyzes the architectural features of LEAP. Section 8 describes related work. Finally, Section 9 summarizes, and outlines problems for further research.

2 Architecture of a General Learning Apprentice

Figure 1 illustrates a general model for learning apprentice systems. There are four main components: a *user* who utilizes the system to solve problems; an *interactive problem solver* that helps the user solve problems and whose performance is being improved by learning; a *knowledge base* consisting of a set of problem-solving operators, control information, and (possibly) a theory of the domain; and a *learning element* consisting of a set of techniques for generalizing from the specific advice provided by the user. The arrows in the diagram show the information flow in a learning apprentice. The user interacts with the problem-solver, which uses the knowledge base to provide problem-solving advice to the user. The learning element observes the user's actions through the problem solver, in order to capture training examples of the user's problem solving steps. The learning element also accesses the knowledge base to obtain background knowledge to guide learning, and updates the knowledge base by adding newly learned knowledge.

We conjecture that learning apprentices will provide a more successful approach to learning than approaches that attempt to learn without observing a user (for example, systems for autonomously learning control knowledge, or systems that attempt to learn directly from a database of cases). The reasons include:

[4] An acronym for Partially Evaluating Determinations.

1 Introduction

Knowledge-based expert systems, such as XCON [1], are increasingly used as high performance problem solvers in real-world applications. However, constructing such expert systems often requires creating and maintaining a large knowledge base of rules. This development and maintenance cost has become a limiting factor in the propagation of expert systems. For example, the XCS system [1] used by Digital to configure computers has a staff of approximately sixty people who maintain and update the knowledge base used by this system.

This paper explores an alternative to the manual development and maintenance of knowledge bases. In particular, we consider interactive systems that learn throughout their lifetime, by generalizing from the observed actions of their users. Humans learn skills in many domains by apprenticeship. For example, novice automobile mechanics learn their trade by helping an expert repair cars, having the expert critique their efforts, and simply observing the actions of the expert in situations that they do not yet understand. We conjecture that such apprentice training methods are well-suited to interactive knowledge-based systems. We report here on the design and evaluation of a particular such system, called LEAP. LEAP is an example of a *learning apprentice system*, as defined previously in [36]:

> 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.

Since the concept of learning apprentices was first proposed in [36], several prototype systems have been developed in a variety of domains [16, 52, 62, 54, 18], offering evidence of the promise of this approach to knowledge acquisition. This paper describes a learning apprentice for digital circuit design called LEAP.[1] It empirically analyzes the effectiveness of LEAP as a knowledge acquisition system, and assesses the extent to which its architectural features help or hinder its function. More generally, the paper uses LEAP as case study to assess the feasibility of learning apprentices as an approach to overcoming the knowledge acquisition bottleneck in expert systems.[2]

LEAP assists the user in the top-down design of digital circuits, helping to transform a functional specification into a digital circuit implementation. LEAP suggests feasible ways of decomposing circuit modules into submodules, as well as the recommended way of choosing among competing decompositions. It automatically learns knowledge that allows it to provide both types of advice, by generalizing from the observed actions of its users.

LEAP uses two knowledge-intensive learning techniques, called VBL and PED, to acquire its two types of advice-giving rules. VBL[3] is a technique based on explanation-based learning [9, 35] that acquires problem-reduction operators for boolean circuit design (that is, for

[1] An acronym for LEarning APprentice.

[2] Section 8 contains a detailed analysis of the differences between LEAP and other learning apprentice systems.

[3] An acronym for Verification-Based Learning.

Abstract

We explore here the feasibility of *learning apprentice* programs: interactive knowledge-based assistants that learn by observing and analyzing the problem-solving steps of their users. In particular, we describe a learning apprentice for digital circuit design, called LEAP. LEAP learns feasible ways of decomposing circuit modules into submodules, as well as the recommended method when there are competing feasible decompositions. VBL is an explanation-based learning technique used in LEAP to infer problem-reduction operators for decomposing circuit modules. PED is a general extension of explanation-based learning to incomplete domain theories containing determinations. PED is used in LEAP to learn control rules for ranking alternative decompositions as well as to extend LEAP's partial theory of circuit cost. An experimental study shows that by using this approach LEAP can learn a significant subset of a manually created knowledge base for boolean circuit design. The experimental study also reveals some limitations of LEAP, and more generally suggests directions for further research in building effective learning apprentice systems.

An Apprentice-Based Approach to Knowledge Acquisition[1]

Sridhar Mahadevan
IBM T.J.Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
(sridhar@watson.ibm.com)

Tom Mitchell
School of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
(mitchell@cs.cmu.edu)

Jack Mostow
Department of Computer Science
Rutgers University
Piscataway, NJ 08855
(mostow@cs.rutgers.edu)

Lou Steinberg
Department of Computer Science
Rutgers University
Piscataway, NJ 08855
(lou@cs.rutgers.edu)

Prasad Tadepalli
Department of Computer Science
Oregon State University
Corvallis, OR 97331
(tadepall@cs.orst.edu)

July 30, 1992

[1]The research reported here was carried out at Rutgers University and at Carnegie-Mellon University, and was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract number N00014-85-K-0116, in part by the National Science Foundation (NSF) under Grant Number DMC-8610507, and in part by the Center for Computer Aids to Industrial Productivity (CAIP), an Advanced Technology Center of the New Jersey Commission on Science and Technology, at Rutgers University, Piscataway, New Jersey. The opinions expressed in this paper are those of the authors and do not reflect any policies, either expressed or implied, of any granting agency.