Knowledge Engineering Issues in VLSI Synthesis

W. H. Wolf
T. J. Kowalski
M. C. McFarland, S.J.

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper explores VLSI synthesis and the role that traditional AI methods can play in solving this problem. VLSI synthesis is hard because interactions among decisions at different levels of abstraction make design choices difficult to identify and evaluate. Our knowledge engineering strategy tackles this problem by organizing knowledge to encourage reasoning about the design through multiple levels of abstraction. We divide design knowledge into three categories: knowledge about modules used to design chips; knowledge used to distinguish and select modules; and knowledge about how to compose new designs from modules. We discuss the uses of procedural and declarative knowledge in each type of knowledge, the types of knowledge useful in each category, and efficient representations for them.

1. INTRODUCTION

The VLSI design domain[1] is well-suited to the exploration of design because of the large body of work on the computer representation and manipulation of VLSI designs. In this paper we present and justify one approach to the knowledge engineering problem for VLSI.

We base our views about VLSI knowledge engineering on our experience with VLSI synthesis programs, notably Fred, a chip planning database,[2] the Design Automation Assistant a knowledge-based synthesis program,[3] and BUD, an intelligent partitioner for ISPS descriptions.[4] Our goal is the automatic design of large (100,000 transistor) systems whose quality as measured by performance and cost is competitive with human-produced designs. We view the design problem as one of successive refinement of an algorithmic description of a processor guided by user-supplied constraints on cost and performance. The synthesis procedure implements the algorithm's data and control flow as a structure built of modules and wires, and finds a layout that implements that structure.

Doubtless the synthesis of high-quality designs is difficult—VLSI design is a composition of a large number of subproblems, many of which are NP-hard. Further, synthesis is in some important respects fundamentally different from the diagnosis problems to which rule-based expert systems are typically applied. Diagnostic systems try to infer behavior of a system from a partial description of its behavior and/or structure; synthesis systems try to build a good implementation from a specification, a process that usually requires search. In this respect the problem more closely resembles the problem attacked by Dendral[5] —finding candidate molecular structures for organic compounds. VLSI synthesis is particularly complex because decisions about architecture, logic design, circuit design, and layout cannot be fully decoupled. Lacking perfect foresight, a synthesis system must be able to reason across multiple levels of abstraction, through deduction and search, to predict or estimate the results of bottom-up implementations.

A synthesis system's ability to make tradeoffs based on bottom-up design information requires not only specific pieces of knowledge, like the size of a particular design, but an organization of knowledge that allows the system to extract and manipulate that knowledge. As in any design system, we judge the value of our knowledge engineering scheme by two criteria: effectiveness, or whether the scheme expresses what synthesis needs to know; and efficiency, or how much it costs to compute the knowledge. The relative importance of effectiveness and efficiency will vary for different tasks; decisions that require the examination of a large number of candidate designs may be satisfied with simple, quickly computable information about the designs, while other decisions are made by detailed examination of a few designs. In the rest of the paper we develop a knowledge engineering scheme and judge it by these two criteria.

2. HORIZONTAL AND VERTICAL REPRESENTATIONS

The partitioning of the digital system design process into levels of abstraction goes back at least to Amdahl, *et al.*[6] and, more concretely, to Bell and Newell,[7] who divided digital system design into four levels of abstraction: processors, programs, logic, and circuits. Bell and Newell emphasized that their taxonomy was dependent on the existing technology and general understanding of computer science, and was likely to change with time, as it did in Siewiorek, Bell, and Newell.[8] A simplified form of their taxonomy was reflected in the SCALD CAD system used to design the S-1 processor.[9] The Carnegie-Mellon Design Automation Project advocated a similar top-down, successive refinement approach for automatic design.[10] More recently, Stefik *et al.*[11] have updated the Bell and Newell paradigm for the VLSI domain.

Gajski and Kuhn have proposed a more comprehensive model for understanding design methodologies.[12] They divide the universe into representations—structural, functional, and geometrical—each of which includes several levels of abstraction. Walker and Thomas have expanded this model to detail

the various levels of abstraction in each representation.[13]

We characterize the levels of abstraction model as *horizontal*: a description level categorizes all the knowledge about a particular phase of design, but the complete description of any particular design requires reasoning at several different levels. Using levels of abstraction as an organizing principle, as in Palladio,[14] limits one's ability to consider bottom-up knowledge. We have organized our knowledge into three groups, with knowledge about modules organized *vertically*—knowledge about a module at all levels of abstraction is contained in the module description. Our methodology is more akin to that of the Caltech Silicon Structures Project,[15] which advanced the "tall thin man" paradigm as an embodiment of the simultaneous consideration of problems at multiple levels of abstraction. We believe that a vertical classification scheme has some distinct advantages.

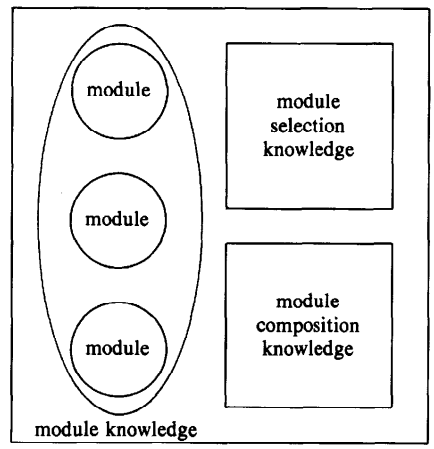
First, a vertical categorization enhances one's ability to analyze tradeoffs. One radical example of the effect of low-level knowledge on high-level decisions is the relation between pinout and architecture. Fabrication, bonding, and power dissipation limitations set a maximum number of input/output pads available on a chip; the resulting upper bound on the amount of communication between the chip and the world is a strong constraint on many architectures. A more subtle example is the relative cost of barrel shifters in nMOS and CMOS—the shifter's higher cost in CMOS may force a different architectural implementation for some algorithms. We must be able to make design decisions by looking deeply into the details of the available implementation choices.

Second, simplified models to describe a particular level of abstraction exclude useful and important designs. One example in what Stefik *et al.* call the *Clocked Primitive Switches* level is the precharged bus (where the parasitic capacitance of a bus temporarily stores a value that is picked up during a later clock phase). This circuit design technique violates a fundamental precept of strict clocking methodologies—that a wire is memoryless—but, when applied with the proper precautions, works. Further, precharged busses are commonly used and are often the only way to improve chip performance to an acceptable level. A strict clocking methodology that has been extended to include precharging is described by Noice *et al.*,[16] but the extensions require explicit verification of the propriety of the precharging circuit, complicating this once simple methodology. Methodologies, like those used to guarantee clocking correctness, simplify a problem enough to allow quick solutions of a wide variety of problems. But to produce high-quality designs, a synthesis system must be aware of the limitations of its methodologies and be able to collect and analyze knowledge to circumvent its limitations.

Figure 1 shows that we partition our design knowledge according to the tasks of synthesis. Each category also uses a distinct knowledge representation scheme. We divide knowledge into three categories: knowledge about particular modules that can be used in a design, which we represent procedurally; knowledge used to distinguish implementations of a module, which we represent declaratively; and knowledge about the composition of designs from modules, which we represent both procedurally and declaratively. In the next three sections we will describe the knowledge in each of these categories and efficient

level	components
PMS	processors, memories, switches, links
program	memories, instructions, operators, controls
logic	registers, operators, latches, logic gates
circuit	transistors, resistors capacitors

horizontal categories
(from Bell and Newell)



vertical categorization
of module knowledge

Figure 1. TWO CATEGORIZATIONS KNOWLEDGE

representations for it. Our enumeration of useful knowledge is not meant to be complete or final, but our experience tells us that this taxonomy is useful.

3. KNOWLEDGE ABOUT MODULES

We divide knowledge about modules into two distinct topics: module designs themselves and methods for evaluating modules. (A module may be a type of component or a class of components, like the class of adders of width n). The design of a module, or of an algorithm for designing a class of modules, is a form of expert knowledge. The ability to compute certain important properties of a module's design is an orthogonal type

of knowledge. The Fred database takes advantage of this orthogonality by using an object-oriented description for modules, as does Palladio. We build a general-purpose set of measurement methods to answer fundamental queries, and build on top of these utilities procedural descriptions of specific modules. The objects that describe these modules are kept in a database that can be searched using selection functions.

Measurement of a module's properties is the best understood topic in VLSI design. Algorithms exist to measure almost every conceivable property of interest. (There is little point in recasting these algorithms in declarative form.) Unfortunately, most synthesis systems have used simple look-up tables or crude built-in approximations to measure candidate designs. Tables are insufficient to describe parameterized module designs; built-in approximations make it difficult to justify decisions to later stages of designs, and inconsistencies may result if different design procedures use different approximations. We have had good success with evaluating candidate designs based on the answers to a few fundamental queries:

- *Physical properties*—Measurements of the values of the electrical elements in the circuit. Values for transistors (length and width) are easy to measure. Parasitic values associated with layout elements (transistors, contact cuts, wires) require more effort.
- *Speed*—Delay is the real time required to propagate logic signals through networks. The details of delay calculation differ among circuit technologies, but all require measurement of the circuit element values and calculation of delays based on those values. Methods for calculating delay for MOS technologies are described by Osterhout.[17]
- *Clocking*—A related, but different type of knowledge describes the clocking behavior of the module, particularly, the clock phases on which the inputs and outputs are valid and the delay in clock cycles from an input to an output. Once clock signals are declared and the clocking behavior of primitive components is known, standard longest path algorithms can be used to compute the clocking delay from inputs to outputs.[18]
- *Shape*—If the module's physical extent is modeled as a set of rectangles, a request for the shape of a module can be used to derive measures such as area, aspect ratio, and minimum required spacing. Fred uses a simple form of compaction[19] to estimate the shape of a module from its constituent components and wires. Compaction also tells us about the locations of the input/output ports for the module.

These queries are usually enough to derive the required knowledge about a module; in a few cases it may be necessary to supply special-purpose methods for calculating some parameter either for performance reasons or, occasionally, because the approximations used in the standard methods are inadequate for the peculiarities of a particular module.

The complementary component of module knowledge is the design of the module itself. We describe module designs procedurally rather than declaratively. There are many design tasks that can be done algorithmically: layout compaction,[20] transistor sizing,[21] and clocking[22] are examples. As with measurement procedures, there is little point in reformulating these algorithms declaratively. The most mundane part of the module design, the basic structural description of the components, wires, and layout elements that implement the module, could be described declaratively, but we choose to use a procedural representation for consistency and ease of use with existing design procedures. Another pragmatic reason for preferring procedural description of a module's structure is that most designers know procedural languages but are unfamiliar with strongly declarative languages.

4. KNOWLEDGE ABOUT MODULE SELECTION

Some information about a module is easily changed with changes in its parameters; other data is static across versions of the module. Often, we can use static information to make an initial selection of modules, and look at the dynamic information (which generally takes longer to compute) only when making final, detailed design decisions. Example of simple questions that greatly prune the search space of modules are "Does the module implement the function I am interested in?", "Is the module implementable in a technology compatible with my design?", and "Is the floor plan of this module compatible with my current physical design?" The distinction between static and dynamic data is not always clear-cut, but we can use it to our advantage to speed the initial search of the module design space.

Fred segregates static, discriminatory knowledge about modules into an associative database to select candidate modules for an implementation. An associative database that supports deduction is powerful enough to support queries used in module selection but simple enough to run quickly. The user and author of the database contents must come to an agreement on the meaning of the predicates in the database. We have found these categories useful in initial module selection:

- *Functionality*—A description of functionality includes a statement of the gross function of the module (adder, shifter, *etc.*) and an enumeration of particular operating characteristics of the module. Synthesis often requires functional information like "Does this latch have a reset signal?" or "What are the feasible bit widths of this shifter?" Such knowledge describes how a module deviates from the ideal behavior for a module that implements the pure function or how it is customized for a particular task.
- *Signal characteristics*—Modules must be compatible in the way they represent logical signals as electrical signals. The important parameters of a signal are:
  - signal level (voltages for logic 0 and 1);
  - signal polarity (active high or low);
  - signal duality (whether the circuit requires/produces both true and complement signals).
- *Technology families*—The most common technological decisions concern fabrication technology and circuit family. The description should allow the synthesis system to distinguish both particular technologies and families of technologies; a module generator may, for instance, be able to produce modules for a number of CMOS technologies. Examples of CMOS circuit families are full complementary, pseudo-nMOS, domino,[23] and zipper.[24] The database should also describe the compatibility of families; for example, domino CMOS circuits may be used to drive fully complementary circuits, but not the reverse.

All this information can be derived from the module descriptions

before design starts and stored in the database. Once facts about the modules have been coded as patterns, such as *(technology adder-1 cmos2.5)*, the database can be searched using standard pattern matching techniques. A pattern like *(and (function ?x add) (technology ?x cmos))* will return the modules that can do an addition and are implemented in CMOS in the bindings of the variable *?x* to the names of the candidate modules.

The associative database mechanism makes it easy to support two useful forms of record-keeping for design decisions. Both methods rely on having the database apply a standard pattern set that is used along with the current pattern specified by the designer. First, a designer can add patterns that express design decisions like fabrication technology. Modules not meeting the criteria will be filtered out by the standard patterns. Similarly, the synthesis program can load the standard pattern set with a design style description that will enforce a set of externally determined choices—circuit family, layout style, *etc.* In both cases the history of changes to the standard pattern set can be used to trace design choices.

Most of the standard techniques described in the literature can be used to speed up the pattern matching search. Because the categories of knowledge, and therefore the first names in the database patterns, are static, they can be organized into relations to speed the search. For efficiency reasons the database should also include ordering criteria to order the search for maximum efficiency; often a few standard categories will greatly restrict the search space.

5. KNOWLEDGE ABOUT MODULE COMPOSITION

The previous sections have described knowledge about particular modules; we also need knowledge about how to put together modules to build new designs. We categorize knowledge about module design into three fields: general composition rules, which describe the basic operations that are used to build a module from components; optimization transformations, which transform one design into another, presumably better design; and search rules, which help the synthesis program search the space of candidate designs. Each of these types of knowledge is used differently, and so requires a different representation.

Examples of general composition rules are that a wire be connected to at least one input port and one output port, or that wires of incompatible clock phases not be connected. Simple composition operations are easy to specify and frequently executed to build and rebuild test designs. For these reasons we choose to represent them as compiled functions. We use the composition functions to build more complex transformations on the design.

Optimization transformations are more intricate. They must recognize a subset of the design that meets some criteria and then transform it into another implementation with the same functionality that is at least as good. The recognition criteria for optimizations are often structural (remove a multiplexer with all its inputs tied to the same signal) but may look at other properties of the design (if a logic gate with minimum-size output transistors is driving a wire with a capacitance of at least 10 pF, replace the gate with a high-power logical equivalent). Experience with the DAA has shown that pattern matching algorithms like those found in production systems such as the OPS family[25] are a good engine for driving transformations. Optimizations stored as patterns are easy to describe and to change; further, optimizations specific to a particular technology or design style can easily be loaded into the system.

The representation of search heuristics is a more complicated issue. Some heuristics cannot easily be formulated as rules; an example is the cost function used by the DAA to evaluate the effect of coalescing functions into a module. Although the result of the cost analysis can be used to drive a rule, writing the cost function itself as rules is both cumbersome and expensive in computation time. As a result, the DAA evaluates the cost function procedurally and uses the result to control rule firing. In general, most predicates that can be used as indicators in guiding search are sufficiently complicated that they should be calculated with procedures—efficiency in calculating their values is of particular concern because of the large size of the search space. However, predicates can be evaluated by rules that decide how to modify the candidate design. Implementing the final decision-making process as rules gives the standard advantages of rule-based systems: rules can easily be changed during experimentation, and special-purpose rules can be added dynamically to customize the search.

6. SYNTHESIS VERSUS ANALYSIS

Hardware synthesis is different from the diagnosis and debugging problems explored by several investigators. Analysis uses knowledge to infer the functionality and performance of a circuit, while synthesis uses knowledge to gauge the quality of an implementation decision. Exploration of the differences between the two problems helps to illustrate the limitations of rule-based systems in synthesis.

Examples of analysis systems are the circuit understanding program of Stallman and Sussman;[26] hardware error diagnosis programs described by Davis and Shrobe,[27] and Genesereth;[28] and a hardware design debugger described by Kelly.[29] Analysis most closely resembles local design optimization, in that an existing design must be analyzed by looking for particular traits. Both concentrate on local analysis of the design, which can be easily implemented as rules.

Synthesis, on the other hand, requires global knowledge of the search space, and several factors limit the utility of rule-based systems for global search. Figure 2 shows the design space for a floating-point arithmetic algorithm as generated by BUD, using an

$area * time^n$ objective function for several values of n. The search space is unpredictable; decisions on how to change the design cannot be made based on simple, local criteria. Two factors argue against using production systems to drive searches through such a space. One is efficiency; the size of the search space for an interesting design is extremely large, and the space may change with design decisions. Another is the difficulty of expressing synthesis decisions as patterns—consider the relative difficulties of explaining how to travel from Murray Hill NJ to New York using procedures ("go from the south exit, turning

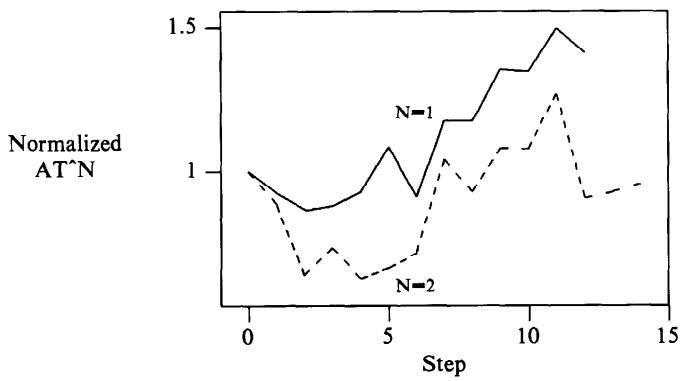


Figure 2. THE SEARCH SPACE OF A SIMPLE DESIGN

left at the light, continue until you find the entrance to I-78 North...") and rules that describe what to do at each intermediate state along the path. Although local transformations may be carried out by rules, the global nature of the search required argues for procedural control of the search strategy, either by rule-based systems that allow control of the search process or by direct coding of the procedures.

7. PLANNING—AN OPEN PROBLEM

One important topic with which we lack direct experience is planning and control of design. Planning is important because many implementation decisions are deferred: later design procedures must know the goals and rationale of the earlier procedures; the assumptions and estimates made during initial design must be verified; and if the design is found to be unsatisfactory, some plan must be formed to correct the problem.

We see two important problems in planning for synthesis. The first is to identify a minimal set of knowledge about design decisions required to detect errors and establish criteria for correcting them. The second problem is how to control the procedures used to solve design subproblems. Not all synthesis algorithms are well-suited to explaining their results or how to change the design with minimal impact on its other properties. The control of synthesis algorithms will probably require expert knowledge about those algorithms encapsulated in rule-based systems.

8. CONCLUSIONS

We have discussed our partitioning of concerns in VLSI design. We believe that it is important to encourage examination of design decisions deeply, particularly because the problem is so poorly understood. So we prefer a vertical organization of knowledge that emphasizes complete descriptions of modules that can be used in the design of a chip.

In general, attacks on individual subproblems encountered during synthesis are best made by well-known algorithms. Traditional AI methods are best suited to the local control of the composition of modules and to diagnosing problems encountered during synthesis. The daunting problem of VLSI synthesis lies in balancing declarative and procedural techniques to converge on a quality design.

REFERENCES

[1] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts (1980).

[2] Wayne Wolf, "An Object-Oriented, Procedural Database for VLSI Chip Planning," *Proceedings, 23rd Design Automation Conference*, ACM/IEEE, (June, 1986).

[3] Thaddeus J. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, Hingham MA (1985).

[4] Michael McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions," *Proceedings, 23rd Design Automation Conference*, ACM/IEEE, (June, 1986).

[5] Bruce G. Buchanan and Edward A. Feigenbaum, "Dendral and Meta-Dendral: Their Applications Dimension," *Artificial Intelligence* **11**(1,2) pp. 5-24 (1978).

[6] G. M. Amdahl , G. A. Blaauw , and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development* **8**(2) pp. 87-101 (April, 1964).

[7] C. Gordon Bell and Allen Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York (1971).

[8] Daniel P. Siewiorek, C. Gordon Bell , and Allen Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York (1982).

[9] Thomas M. McWilliams and Lawrence C. Widdoes, Jr., "SCALD: Structured Computer-Aided Logic Design," *15th Design Automation Conference*, pp. 271-277 IEEE Computer Society Press, (1977).

[10] Stephen W. Director , Alice C. Parker , Daniel P. Siewiorek , and Donald E. Thomas, Jr., "A Design Methodology and Computer Aids for Digital VLSI Systems," *IEEE Transactions on Circuits and Systems* **CAS-28**(7) pp. 634-645 (July, 1981).

[11] Mark Stefik, Daniel G. Bobrow, Alan Bell, Harold Brown, Lynn Conway, and Christoper Tong, "The Partitioning of Concerns in Digital System Design," *Proceedings, 1982 Conference on Advanced Research in VLSI, MIT*, pp. 43-52 Artech House, (January, 1983).

[12] Daniel D. Gajski and Robert H. Kuhn, "Guest Editor's Introduction: New VLSI Tools," *Computer* **16**(12) pp. 11-14 (December, 1983).

[13] Robert A. Walker and Donald E. Thomas, "A Model of Design Representation and Synthesis," *22nd ACM/IEEE Design Automation Conference*, pp. 453-459 IEEE Computer Society Press, (June, 1985).

[14] Harold Brown, Christopher Tong, and Gordon Foyster, "Palladio: An Exploratory Environment for Circuit Design," *Computer*, pp. 41-56 IEEE Computer Society, (December, 1983).

[15] Stephen Trimberger , James A. Rowson , Charles R. Lang , and John P. Gray, "A Structured Design Methodology and Associated Software Tools," *IEEE Transactions on Circuits and Systems* **CAS-28**(7) pp. 618-634 (July, 1981).

[16] David Noice, Rob Mathews, and John Newkirk, "A Clocking Discipline for Two-Phase Digital Systems," *Proceedings, International Conference on Circuits and Computers*, pp. 108-111 IEEE Computer Society, (1982).

[17] John K. Osterhout, "Crystal: A Timing Analyzer for nMOS VLSI Circuits," *Proceedings, Third Caltech Conference on VLSI*, pp. 57-69 Rockville MD, (1983).

[18] Kurt Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, Berlin (1984).

[19] M. W. Bales, *Layout Rule Spacing of Symbolic Integrated Circuit Artwork*, Masters thesis, University of California, Berkeley (May 4, 1982).

[20] Wayne Wolf, *Two-Dimensional Compaction Strategies*, PhD thesis, Stanford University (March 1984).

[21] J. P. Fishburn and A. E. Dunlop, "TILOS: A Posynomial Programming Approach to Transistor Sizing," *Proceedings, ICCAD-85*, pp. 326-328 IEEE Computer Society, (November, 1985).

[22] Nohbyung Park and Alice Parker, "Synthesis of Optimal Clocking Schemes," *Proceedings, 22nd Design Automation Conference*, pp. 489-495 IEEE Computer Society, (June, 1985).

[23] R. H. Krambeck, C. M. Lee, and H. F. S. Law, "High-Speed Compact Circuits with CMOS," *IEEE Journal of Solid-State Circuits* **SC-17**(3) pp. 614-619 IEEE Circuits and Systems Society, (June, 1982).

[24] Charles M. Lee and Ellen W. Szeto, "Zipper CMOS," *IEEE Circuits and Devices Magazine* **2**(3) pp. 10-17 (May, 1986).

[25] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin, *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, Massachusetts (1985).

[26] Richard M. Stallman and Gerald J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence* **9**(2) pp. 135-196 (October, 1977).

[27] Randall Davis and Howard Shrobe, "Representing Structure and Behavior of Digital Hardware," *Computer* **16**(10) pp. 75-82 (October, 1983).

[28] Michael R. Genesereth, "The Use of Design Descriptions in Automated Diagnosis," pp. 411-436 in *Qualitative Reasoning About Physical Systems*, ed. Daniel G. Bobrow, MIT Press, Cambridge MA (1985).

[29] Van E. Kelly, *The Critter System—An Artificial Intelligence Approach to Digital Circuit Design Critiquing*, PhD thesis, Rutgers University (January, 1985).