AI Magazine Volume 3 Number 3 (1982) (© AAAI)

Towards the Principled Engineering of Knowledge

Mark Stefik and Lynn Conway

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Abstract

The acquisition of expert knowledge is fundamental to the creation of expert systems. The conventional approach to building expert systems assumes that the knowledge exists, and that it is feasible to find an expert who has the knowledge and can articulate it in collaboration with a knowledge engineer. This article considers the practice of knowledge engineering when these assumptions can not be strictly justified. It draws on our experiences in the design of VLSI design methods, and in the prototyping of an expert assistant for VLSI design. We suggest methods for expanding the practice of knowledge engineering when applied to fields that are fragmented and undergoing rapid evolution. We outline how the expanded practice can shape and accelerate the process of knowledge generation and refinement. Our examples also clarify some of the unarticulated present practice of knowledge engineering.

Thanks especially to Daniel Bobrow for helping us to discover, refine, and articulate many of these ideas. We are also grateful to John Seely Brown, Douglas Lenat, Christopher Tong, and Michael Williams for their thorough reviews of drafts of this article. Thanks also to the members of the KBVLSI project: Alan Bell, Harry Barrow, Daniel Bobrow, Harold Brown, Phil Gerring, Gordon Foyster, Gordon Novak, Christopher Tong, and Narinder Singh, who have participated in the knowledge engineering and expert systems aspects of the project. The synergistic combination of their contributions to the project, merging ideas from a wide variety of viewpoints, has given us all a sense of excitement and common purpose.

Thanks to the Xerox Corporation for providing the intellectual and computational environment in which this work could be done. This research is being conducted as part of the KBVLSI project in collaboration with the Heuristic Programming Project at Stanford University. The Stanford component of the research is funded by the Defense Advanced Research Projects Agency.

A STRONG MOTIVATION for AI research on *expert* systems is that these problem domains provide an appropriate level of complexity for studying problem solving. Toward this end, knowledge acquisition is sometimes considered a necessary burden, carried out under protest so that one can get on with the study of cognitive processes in problem solving. In this article we argue that the two activities—knowledge acquisition and cognitive modeling—are necessarily interwoven, and provide interesting opportunities when taken together. Knowledge acquisition shapes cognitive modeling because *operational* knowledge contains assumptions and directions for its use, that is, an implicit processing model. In return, problem solving models can profoundly shape knowledge acquisition by providing a framework for the articulation and creation of domain expertise. This introduces the theme of this article, that one can *engineer* bodies of knowledge for various purposes, such as learnability, or efficient use in problem solving. To the knowledge engineering slogan "knowledge is power," we add "knowledge is an artifact, worthy of design."

The organization of this article is as follows: We first consider the practice of VLSI design and find difficulties with the building of expert systems for that area using conventional methods. The second section relates some experiences in the transformation of design practices in VLSI design communities. These experiences suggest that knowledge embedded in these transformed methods gives practitioners a

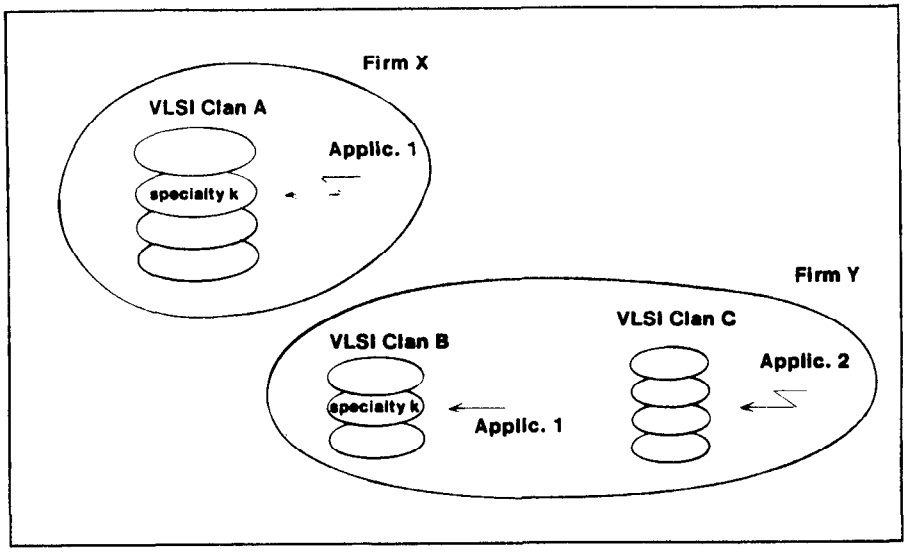

Figure 1 The demographics of VLSI clan structure To a first approximation, the design community is divided into a number of clans Each clan is divided by specializations of labor, and works on a particular type of application (e g , microprocessor chips) Secrecy barriers between firms inhibit flow of knowledge between clans, although some specialties may find common practice in several firms

"cognitive advantage " In the third section we suggest some principles and measurements that can be applied to the engineering of knowledge, in order to impart advantages for cognitive processing We then present examples of the principled engineering of knowledge drawn from our experiences with VLSI design methodology. Finally, we offer speculations on possible roles for knowledge engineers working on new bodies of knowledge.

A Shift in Viewpoint from Experts to Clans

Over the past decade there have been tremendous advances in the fabrication of integrated circuits (Robinson, 1980a). Circuits have become smaller and manufacturing costs have dropped dramatically. Design is becoming the dominant cost (Robinson, 1980b) with the current round of miniaturization, which goes by the name of VLSI for very large scale integration. This is leading to a substantial interest in understanding design processes.

The tendency to specialize and the shifting of the technological base are forces for diversity in the integrated circuit design community. To a first approximation, the community can be viewed as a collection of sometimes independent and sometimes competing *clans* having different practices, identifiable by their tools and methods Digital system architects and integrated circuit designers often specialize in different kinds of systems and circuits, such as microprocessors or digital signal-processing chips (just as mechanical system designers may specialize in domains such as aircraft or automobiles). The picture is further complicated by the traditional stress on secrecy within the integrated circuit industry—designers in different firms find themselves initiated into the local craft practices of their particular firms. Cultural drift occurs, gradually widening the gap between practices of different firms. Therefore, many separate clans in different firms use different methods to work different parts of the space of possible designs (see Fig. 1).

The practice of VSLI design has further evolved and fragmented in response to shifts in the technological base of integrated circuit processing-technology As companies have explored and invested in different fabrication technologies, the design community has become divided by another dimension, that of the particular technology of implementation (nMOS, CMOS, I^2L, etc.).

Within each clan in the community, expertise is further split according to specialized divisions of labor For example, microprocessor design has traditionally had four levels of specialization system architecture, logic design, circuit design, and finally, layout design. Such division of expertise among cooperating specialists is observed in many problem domains. However, in integrated circuit design the specializations of expertise often carry over many practices from earlier, non-integrated, circuit technologies. The layered accumulation of past practices has led to a situation where most integrated system architects are unable to understand circuit layouts, and most layout designers are unable to understand the function of the chip as a whole

All these factors—the different design domains, the evolution of the underlying fabrication technologies, and the different specialized divisions of expert practice—have led to a state where design practices appear to be extremely com-

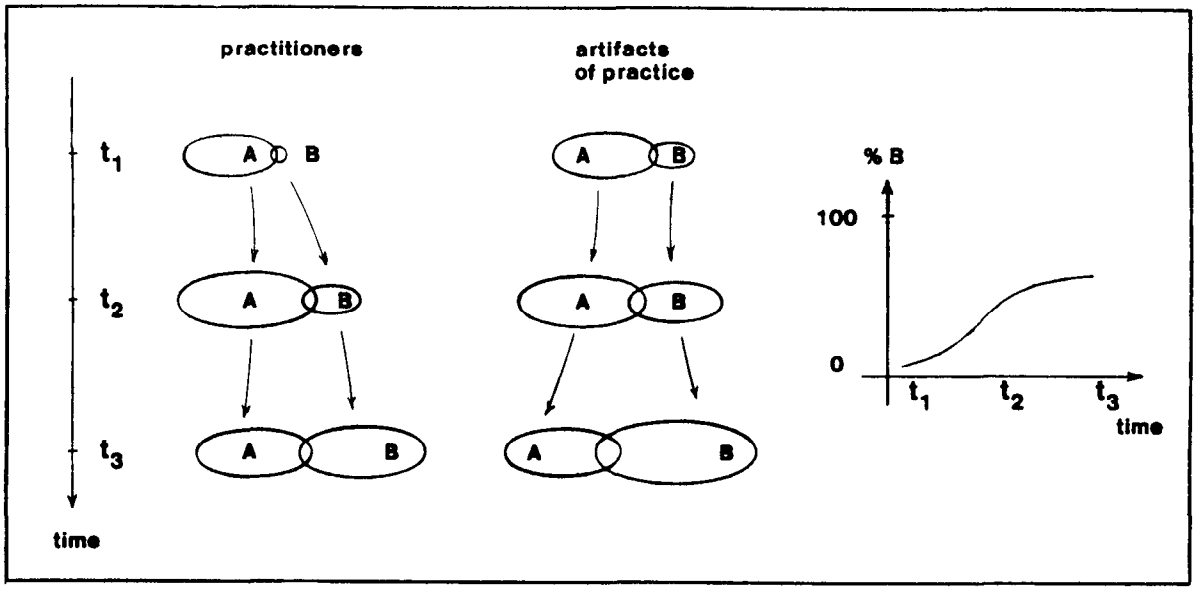

Figure 2 Knowledge diffusion and evolution This figure shows two competing technologies (e g , sailing ships and steam ships) labeled A and B Social historians of technology (Sahal, 1981) measure the population of practitioners and their artifacts over time In this example, technology B is gradually displacing technology A as indicated by the size of the population diagrams and by the slope of the S-curve on the right Actual diffusion of technology can follow more complicated patterns as new areas open up, and as groups displace each other or expand to compete in other areas

plex and in a constant state of tumultuous change (if complexity is measured by summing the observed knowledge, and change is measured by the differences in knowledge observed over time).

From the perspective of conventional thought in the knowledge engineering community, such a problem domain is not ready for an expert system. The knowledge is changing too rapidly, and community practice is too fragmented. Across clans, practice and knowledge vary radically and there is a widely shared belief that there are many open questions and opportunities for developing design methods. If durable *expert* knowledge about how to design VLSI systems exists at all, it has not been widely recognized in the design community. This lack of convergence is in conflict with the conventional approach to building expert systems, which assumes that the knowledge exists and that it is feasible to find an expert who has the knowledge and can articulate it in collaboration with a knowledge engineer (Feigenbaum, 1977; Barstow and Buchanan, 1981; Duda and Gaschnig, 1981; Davis, 1982).

Conventional knowledge engineering, even as applied in areas such as medicine, has historically dealt with selected subsets of knowledge that are relatively stable over time, and that are not highly fragmented into different clusters of specializations carried by competing clans By repeated application of these conventional methods, the field has evolved a thought style that fails to recognize the presence and significance of the fragmentation, competition, and transformation phenomena inherent in the underlying evolution of the knowledge itself.

The Design of Design Knowledge

Over the past three or four years, a new clan of VLSI system designers has been emerging, using design methods described in the Mead and Conway text on VLSI design (Mead and Conway, 1980) Courses based on this book are now offered in over one hundred universities and by a number of commercial training organizations. As Mead-Conway designers have succeeded in completing interesting designs in substantially less time than practitioners of other methods, the phenomenon has attracted considerable attention (Marshall, Waller, and Wolff, 1981), and the methods have propagated rather rapidly

It is from the success of the Mead-Conway work on the design of VLSI design methods that we gain confidence in the new line of thought stressed in this article, namely, that knowledge can be designed, and that reusable principles can be developed for the principled practice of knowledge engineering (see Fig. 2)

The Mead-Conway methods can be visualized as a covering by one simple body of knowledge of the previously separate bodies of knowledge used by the system architect, logic designer, integrated circuit designer, and chip layout designer. Those existing layers of specialized knowledge had incrementally accumulated over many years, without reorganization; while tracking a large accumulation of technol-

ogy change. Mead and Conway seized the opportunity inherent in the accumulated technology for a major restructuring and redesign of digital system design knowledge.

When using the methods, an individual designer can conceptualize a design and make all the decisions from architecture to chip layout. Furthermore, it becomes possible to explore the design space and optimize an overall design in ways precluded when the design is forced through the usual sequence of narrow specialties. The new knowledge is applicable in a wide variety of design domains. We suggest that the knowledge embedded in this method gives practitioners a *cognitive advantage,* characterized as a simpler cognitive mapping from architectural concepts down through layouts in silicon. We will return to this claim and its implications in the next section.

Conway has given an account of the process by which the textbook and the new methods were created, tested, and then integrated into the design community (Conway, 1981). Much of that account deals with ways of refining new methods by promoting their experimental use in the design community, and then responding to feedback from designers. A great deal of novel infrastructure was created to encourage the substantial amount of exploratory use required to debug, evaluate, and refine design methods, and to stimulate the diffusion of methods into the engineering community. The ideas for creating and refining methods *per se* were described anecdotally in terms of a generate-test-revise cycle.

The Mead-Conway example illustrates the deliberate design of a system of knowledge intended to replace an existing body of *ad hoc* design practice. Conway's account deals mainly with the social dimensions of the phenomenon—the evolving demographics of the knowledge during its testing, refinement, and propagation. As part of a recent knowledge engineering enterprise, we have begun to obtain interesting insights into properties of the Mead-Conway knowledge itself We have also begun to further engineer that knowledge under the guidance of certain new principles. These knowledge engineering activities are the subject of the following sections of this article.

Developing Principles for the Engineering of Knowledge

The VLSI System Design Area at Xerox PARC and the Heuristic Programming Project at Stanford University have undertaken a collaborative Knowledge-based VLSI Design (KBVLSI) Project. The aim of the project is to explore possibilities for application of AI methods and expert system technology towards the creation of an expert assistant for the VLSI designer. The project's leaders considered this to be a difficult application area, one that would test the state-of-the-art of expert system technology. On the other hand, VLSI design was also seen as an application area of strategic importance, one that promised great leverage of any successes that might occur.

Because of the observations discussed above, the project leaders chose to focus on mechanization of the Mead-Conway VLSI design methods. When the project began, that design community had not produced a formal body of design knowledge, from the knowledge engineering point of view. The community's methods were relatively simple, and a descriptive textbook existed. Most of the embedded knowledge was informal, and was communicated in the traditional manner—by way of examples.

It was clear from the examples that the designers worked within multiple design levels ranging from abstract system descriptions to chip layouts. However, most of the levels were not recorded in a formal notation, and were only informally shared within the design community. During KBVLSI project efforts to formalize these abstraction levels, we gained insight into how certain of the levels were different from those traditionally used in integrated circuit design. We then began to study the general properties of sets of abstractions, hoping to find bases for comparing and understanding the relative utility of different sets of abstractions, and to perhaps even find principles for *designing* sets of abstractions. In this section we describe some results of that study.

The combinatorics of problem decomposition The importance of effective problem decomposition for taming large search problems has been recognized in AI for many years. This idea was quantified by Minsky, who explained the combinatorial advantage of introducing *planning islands* for reducing search by what he called a "fractional exponent:"

> In a graph with 10 branches descending from each node, a 20 step search might involve 10^{20} trials, which is out of the question, while the insertion of just four ... *sequential subgoals* might reduce the search to only 5×10^4 trials, which is within reason for machine exploration. Thus it will be worth a relatively enormous effort to find such "islands" in the solution of complex problems. Note that even if one encountered, say 10^6 failures of such procedures before success, one would still have gained a factor of perhaps 10^{10} in over-all trial reduction. (Minsky, 1961, pp. 441–442)

The islands in this example decompose the search into a set of subproblems. Although the search reduction is dramatic, it depends heavily on the placement of the islands. For example, if the islands were located at levels 16, 17, 18, and 19 in the tree, the search would still require 10^{16} trials. Merely breaking a problem into subproblems is not nearly as powerful as breaking it into well-spaced subproblems.

Languages and problem decomposition. Although this enumeration illustrates the power of well-spaced subproblems, it gives no advice about how to find them. It is intuitively clear that big steps are better than little ones, but how do we find the steps? Must the decomposition process be determined anew for every problem? This section presents two ideas that bear on this. The first idea is that a language that describes suitable abstractions of problems, can guide the decomposition of problems into subproblems. This idea can be iterated to yield an ordered set of languages providing

intermediate abstractions The second idea is that the order of the set of languages matters. The languages should be arranged to yield a low degree of backtracking in problem solving When an ordering of languages can be found that provides low backtracking across a broad spectrum of problems in a domain, the languages can be used to effectively guide problem decomposition in the domain.

The first idea can be illustrated by the problem of finding a route from Palo Alto to the San Francisco Airport. A methodical *street at a time* approach would search a widening circle of city blocks until the airport was found. In contrast, if a map is available that shows only main roads and connections, then the search can be confined to the points on the map. The map helps us to decompose the original problem into separate routing subproblems through intermediate points (e.g., Embarcadero and El Camino, the Embarcadero entrance to the Bayshore Freeway, and the airport exit from the Bayshore Freeway). The search of the map can be expressed in terms of a language whose "terms" are the points on the map and whose "syntax" is the set of rules for connecting adjacent points into routes Such languages should preserve some important characteristics of the problems, but suppress much of the detail For example, the map would be of no use for decomposing problems if it failed to show the freeway exits, or if it showed minor streets but omitted the main traffic arteries

This idea of abstraction can be applied iteratively in problem solving, as in the hierarchical planning systems reviewed in Stefik, Aikins, Balzer, et al., (1982). Abstraction can take several forms, such as detail suppression, or implementation relationships. We use the term implementation to indicate relationships between abstract constructs of completely different types In such cases it is convenient for the abstraction levels to be reified in terms of distinct languages. Search reduction results when there is an ordered sequence of languages such that an abstract construct can be implemented in terms of expanded constructs at lower levels The search for candidate solutions in the abstract languages, and the early elimination of some of them, yield substantial economies for problem solving. By eliminating a particular abstract construct from consideration, a problem solver avoids pursuing the members of an equivalence class of detailed solutions

Even if an abstracted problem is not a perfect morphism of the original, its solution may prove useful as a guide. What matters is that a language (or ordered set of languages) provides a guide to decision making so that the average retraction of decisions is low. For example, in multiple languages for a top-down design process, each language facilitates the composition of constructs that need to be implemented in the language at the next level down. Typically, much knowledge must be brought to bear in making the implementation decisions In any particular problem, knowledge gained during the implementation process may suggest the need to reconsider some of the decisions made at a higher level, that is, a repartitioning of subproblems. For languages to be generally effective across problems in a domain, the relative rate of such retraction must be low. This amounts to a uniformity requirement on decision making –on average, the same kinds of decisions must be critical for all problems, so that making them first will efficiently guide problem decomposition.

When the languages successfully guide the partitioning of subproblems, we say that the languages exhibit the *planning island effect.* We claim that the influence on a problem solver is akin to that proposed in the Whorfian hypothesis- language shapes the patterns of habitual thought (Whorf, 1956). For example, a designer who systematically carries a design through several implementations in different languages is guided by an "invisible hand" that determines the kinds of decisions that are made at each step

A comparison of design methods In creating their textbook, Mead and Conway worked to simplify VLSI design practice by reducing the amount of knowledge required, and by restructuring the form of the knowledge. Traditional integrated circuit design processes have four levels of specialization System architects perform the highest level of design, specifying the function blocks, registers, and data paths of a design The next level is carried out by logic designers, who work out the details of the logic implementing the functional blocks. Circuit designers then specify the circuit devices and interconnections to be used to implement the logic designs. Finally, layout designers specify the geometric patterns to be conveyed into the various layers of the integrated circuit chips to implement the devices and interconnections. Implicit in this division of labor is a set of informal abstraction levels, one per specialty, that convey a planning island effect into the design process (see Fig. 3).

In contrast with traditional practice, the Mead-Conway methods bypass the requirement for Boolean logic gate representation as an intermediate step in design They thus eliminate an unneeded step in the design process, a step that often introduces unwanted complications while precluding important design constructs. The methods also advocate the consistent use of simple charge-storage methods instead of cross-coupled gates for saving state between register transfer stages A simple "primitive switches" design step, which can generate not only logic gates when needed, but also steering logic and charge-storing registers, replaced *both* the logic-gate step and the detailed electrical circuit-design step of previous methods Mead and Conway also proposed simplified electrical models, timing models, and layout rules for guiding design under the resulting methods. The methods are sufficiently simple to learn and to use so that an individual designer can now rapidly carry a design from architecture to layout in silicon, whereas previously a team of specialists would have been required.

We hypothesize that further analysis of the new system of abstraction levels embedded in the Mead-Conway methods, as contrasted with the traditional levels, will reveal the sources of the advantages of the methods in terms such as the planning island effect. However, in order to conduct

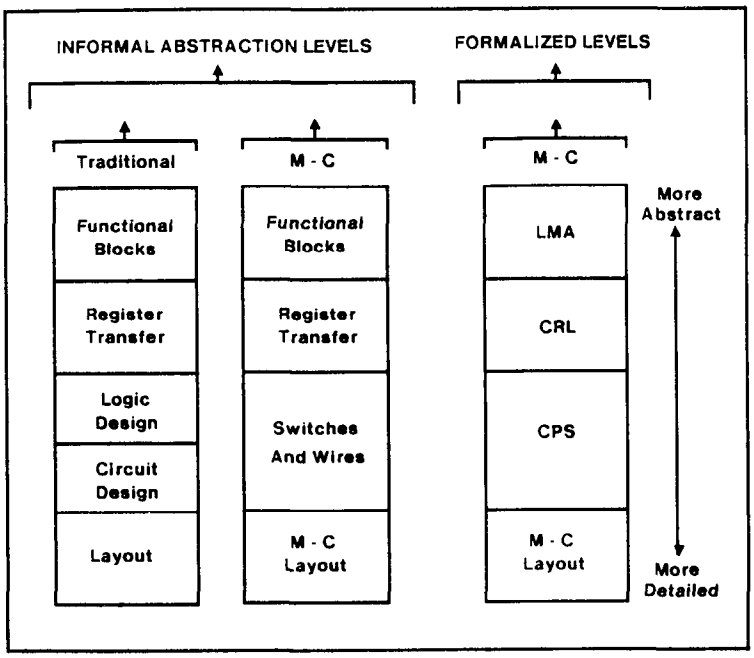

Figure 3 Comparison of relative placements of abstraction levels
The sequences of levels are shown for traditional IC design methods, informal Mead-Conway methods, and for re-engineered, formalized Mead-Conway methods

such analyses, and also in order to embed the methods in an expert system, we must formalize these abstraction levels.

Creating synthesis languages. The design practices described in the previous section are largely informal, both in the Mead-Conway methods and in more traditional methods. By this we mean that the rules of design are not written down in terms of a formal language having precise rules of syntax Although informal notations seem to accommodate open-ended specifications, they are usually inadequate as documentation, either for a designer of a large project, or for teams of designers.

Many formal languages for describing hardware have been proposed. For example, there are many proposed logic descriptions and register transfer descriptions in the *hardware description language* literature. However, these hardware description languages have had very limited acceptance in actual design practice. We believe that the reason for this is that the languages were designed for the wrong purposes. They were designed for documenting, describing, and verifying the properties of *existing* hardware.

These observations lead us to take a closer look at the properties of notations used in the integrated circuit design culture. One widely used formal notation in integrated circuit design is the artwork or *layout* notation. This notation describes integrated circuits in terms of "colored rectangles" (representing material on a chip) that can be composed to build up large designs. Combined with the layout notation is a set of *composition rules,* called layout design rules. Designs created under these rules are guaranteed to have adequate physical spacing on a chip.

The layout language has several important properties which make it useful for the synthesis of designs First, primitive terms can be combined to form larger terms and subsystems ("design by composition") Second, there are rules of composition that define the allowed compositions of these terms These rules apply both to composite objects and primitive terms. Third, there is a well characterized set of *bugs* that are avoided when the composition rules are obeyed. At the layout level, these bugs correspond to the function and performance problems caused by inadequate physical spacing. The composition rules provide a simple *shallow model* of composition that is based on a *deep model* of electrical properties and fabrication tolerances (Lyon, 1981).

With these properties in mind, we have created the set of synthesis languages characterized in Figure 4. The set of languages can be viewed as a re-engineering and then formalization of the Mead-Conway abstraction levels, with the inclusion of a new type of top abstraction level (see Fig. 3) Each language provides a vocabulary of terms and a set of composition rules that define legal combinations of the terms. The concerns of each language are characterized by specific classes of *bugs* that can be avoided when the composition rules are followed.

Collectively, the synthesis languages factor the concerns of a digital designer. (See Stefik, Bobrow, Bell, et al., 1982 for a discussion and more details) The *linked module abstraction* (LMA) language is concerned with the sequencing of computational events. It describes the paths along which

Description Level	Concerns	Terms	Composition Rules	Bugs Avoided
Linked Module Abstraction LMA	Event Sequencing	Modules Forks Joins Buffers	Token Conservation Fork/Join Rules	Deadlock Data not Ready
Clocked Registers and Logic CRL	Clocking 2 Phase	Stages Register Transfer Transfer Functions	Connection of Stages	Mixed Clock Bugs Unclocked Feedback
Clocked Primitive Switches CPS	Digital Behavior	Pull-Ups Pull-downs Pass Transistors	Connection of switch networks Ratio Rules	Charge Sharing Switching Levels
Layout	Physical Dimensions	Colored Rectangles	Lambda Rules	Spacing Errors

Figure 4 Synthesis languages of an expert system for aiding VLSI design Each language has a set of *terms* that can be composed to form systems and a set of *composition rules* that define legal combinations of the terms The concerns of each language are characterized by specific classes of *bugs* avoided when the composition rules are followed

data can flow, the sequential and parallel activation of computations, and the distribution of registers. The LMA level provides a simple covering of ideas from many sources including Petri nets and the design of speed-independent modules. The LMA composition rules preclude bugs of starting computations before the data are ready, and deadlock bugs that arise from the use of shared modules. The *clocked registers and logic* (CRL) language is concerned with the composition of stages of combinational logic and registers. The CRL rules preclude various bugs related to clocking in a two-phase system The *clocked primitive switches* (CPS) language distinguishes between different uses for logic, such as steering, clocking, and restoring, and is concerned with the *digital* behavior of a system The composition rules of this language prevent bugs of non-digital behavior caused by charge sharing and invalid switching levels.

The characteristics of these *synthesis languages* stand in contrast to the hardware description languages (i e , *analysis languages*) mentioned earlier. The logic description languages are too isolated and the register transfer (RT) languages are incomplete and insufficiently formalized. For example, it is difficult to find clocking specifications in a typical RT description. The composition of partial RT descriptions does not yield a *test of correctness* for clocking. Those hardware description languages provide no composition rules, optimization rules, or bug characterizations, and fail to provide enough leverage for designers.

We believe that the practice of creating synthesis languages for different design domains may eventually be understood in terms of a relatively small number of common principles. To return to the map example, the process of making maps is not radically different for different cities Our search for such abstract synthesis languages has been aided by our interest in their formal properties. For example, the articulation of nearly independent *concerns* arises, in part, from generalizing about categories of design bugs. The characterizations of design bugs arise from the articulation of composition rules. The composition rules arise from the need to determine when the compositions of terms are valid.

Quantifying the abstraction power of synthesis languages. Given a set of synthesis languages, we would like to be able to quantify their utility for creating well-spaced planning islands. The *spacing of islands* is a metaphor for branching factors between levels. We have found it useful to define two such factors, termed the *choice factor* and the *expansion factor* The choice factor is a measure of the alternatives in decision making. It is defined as the average number of possible alternative implementations for a primitive term at the next lower level. The expansion factor is a measure of the expansion of detail. It is defined as the average multiplicative increase in the amount of information for specification of a primitive term at the next lower level If there are typically 20 ways to implement an LMA term in CRL and the average increase in the amount of information is 200, then the choice factor is 20 and the expansion factor is 200 (see Fig 5)

In the Minsky example for computing the power of planning islands, we saw factors on the order of 10^4. When the computation is extended to consider *multiple levels,* the factors for the individual pairs of levels can be multiplied to obtain factors for the entire set of languages For example, with four levels an average choice factor of 22 provides a total choice factor of 10^4

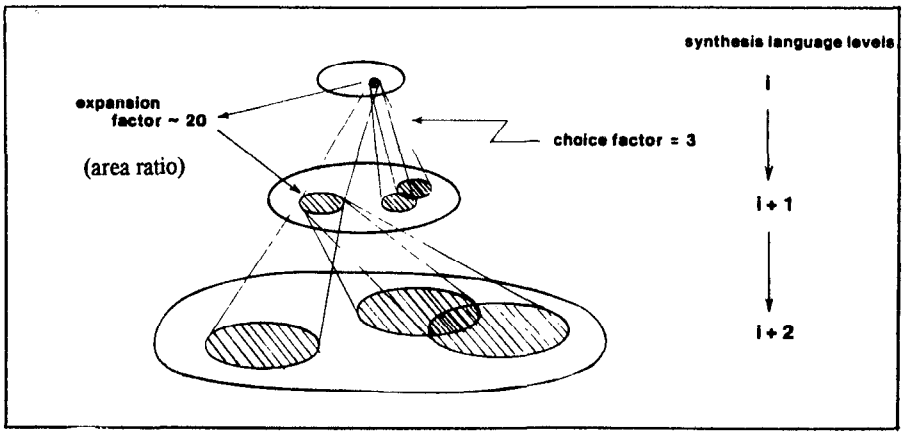

Figure 5 Choice and expansion factors for synthesis languages The *choice factor* and the *expansion factor* are two measures of the abstraction power of a synthesis language The choice factor measures the number of alternatives in decision making, and the expansion factor measures the expansion of detail For an ordered set of levels, the total choice and expansion factors of the set correspond to the products of the individual factors

Accurate quantifications of the choice and expansion factors of the synthesis languages being developed for the KBVLSI project are still a ways off and it is clear that the quantification of these factors depends on a careful information-theoretic analysis As we complete our knowledge bases and expand our experience with these levels, we will be interested in developing systematic means of applying the new measures to our work, and will perhaps further tune our abstraction levels in response to the results.

Examples of the Engineering of Knowledge

By suggesting that *knowledge* is subject to design, we place knowledge engineering among the sciences of the artificial (Simon, 1981). Designed objects are *artificial* in that they are man-made and shaped to suit a designer's purposes for use in some environment As an engineering practice develops, engineering principles emerge that account for the constraints imposed by designer goals and an environment. Since there can be antagonistic goals (Tong, 1982), the principles need to account for examples of tradeoffs. Although no substantial body of knowledge engineering principles has yet been articulated, a partial picture of some of its elements is starting to appear. This section presents several examples of the engineering of knowledge from the KBVLSI project, and the reasons for the shaping of knowledge that we have found compelling. These examples suggest that a reusable body of practice may eventually emerge

Example: Composition and optimization. The "design by composition" model characterizes a design process that is dominated by the composition of terms. The terms can be primitive in some synthesis language, or they can be previously created composite terms known to be correct (relative to some classes of bugs). Observations of practicing system architects and circuit designers confirm that this technique is a significant part of typical practice. This section argues that knowledge about design should be engineered to separate composition knowledge from optimization knowledge (see Fig. 6).

The following composition rule about clocking is taken from rules at the CRL-level:

> Data outputs from a stage must be valid during the opposite clocking interval than the data input to that stage

This rule, combined with others, prevents creation of stages having distinct input lines holding data valid on different clocks (mixed clock bugs) and also creation of unclocked feedback loops. This insures correct alternation of clocks on successive stages as shown in Figure 6a. Figure 6b shows two versions of a circuit for a memory cell. The optimized version omits a clocking switch, thus violating a composition rule. But, given some assumptions about output line loading and clock speed, the optimized circuit can be shown to be correct. The proof observes that a signal going through two inverters is restored to its original value. A more general form of the argument would accomodate any "identity transform" (e.g., as implemented by an even number of serially-connected inverters) The composition rule by itself employs the worst case assumption that data changes on lines, and misses this optimization. If the composition rule had to account for all possible optimizations, it would need many more *exception clauses* as in:

> All of the data inputs to a stage must be valid during the high interval of the same clock unless (1) they are derived from an unclocked stage yielding an identity transform and the loading of the line is . and the capacitance is less than ... and the speed of the unclocked stage is ... or (2) . .

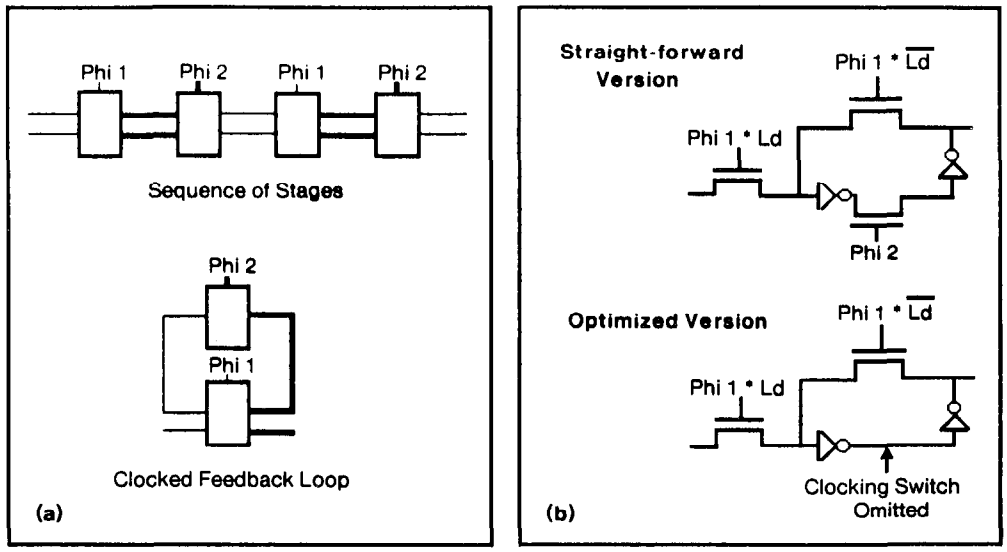

Figure 6 Optimization example from the CRL level Figure 6a shows the usual composition of *stages* at the CRL level The lines labeled Phi1 and Phi2 represent clock lines The key observation is that the clock lines alternate for successive stages Figure 6b shows two versions of a memory cell circuit The optimized version violates the composition rule The text argues that the optimization of the cell is correct, but that its correctness depends on properties of the memory circuit that are not true in general The price of simplicity in the composition rules is that they make *worst case assumptions* But later optimizations can take account of special cases

A serious disadvantage of this approach is that it *diminishes the leverage conferred by multiple abstraction levels* Verifying the optimization clauses in the complicated form of the rule is not generally possible from *only* a CRL description, because the capacitance information is not known until another level of implementation is done (a layout). As a consequence, designs could not always be verified to be free of clocking bugs *at the CRL level.* This would diminish the effectiveness of the CRL level in producing planning islands.

An alternative is to use the simple composition rules and to have a separate pass in the design process that uses *optimization rules* to identify and introduce optimizations. This has several advantages By keeping the composition rules simple, it is easier to get them right because the special cases are isolated We have found examples of optimization conditions like these at every level of description in our work. In most cases, an optimization combines information from more than one of our description levels.

The factoring of optimization knowledge helps to defuse the argument that "simplified bodies of knowledge must miss something." Our approach to this is to first formalize the knowledge in terms of languages, for which we can be precise about exactly what they cover. The languages can then be engineered to have appropriate properties for synthesis, as discussed in the previous section. Finally, separate bodies of optimization knowledge can be developed that extend the total coverage of the design knowledge by characterizing opportunities for performance tuning.

This example of the engineering of design knowledge illustrates the influence of a problem solving model on the acquisition and design of knowledge The current framework admits the possibility of an approach to design that separates concerns of functional correctness (via composition) from performance tuning (via optimization). This reflects cognitive economics by enabling the effective use of planning islands in composition, and by admitting a design process in which only the critical portions of a design are optimized.

Example: Coverage and simplicity. Two important attributes of a body of knowledge are its coverage and its simplicity By coverage we mean a measure of the cases in the field of interest for which the knowledge is adequate. In design knowledge for VLSI systems, coverage refers to the kinds of digital systems and integrated circuit technologies that can be adequately characterized

The search for simplicity is endemic in science (e.g., Occam's razor). In our knowledge engineering, we have employed several kinds of simplicity measure:

1 *basis simplicity*—the number of kinds of basic elements;

2 *expression simplicity*—the length of the average (most common) expressions;

3. *composition simplicity*—the number and simplicity of the rules for combining terms with other terms

The first measure is used when we try to reduce the number of primitive terms by defining some constructs in terms of others. The second measure is used to counteract excessive

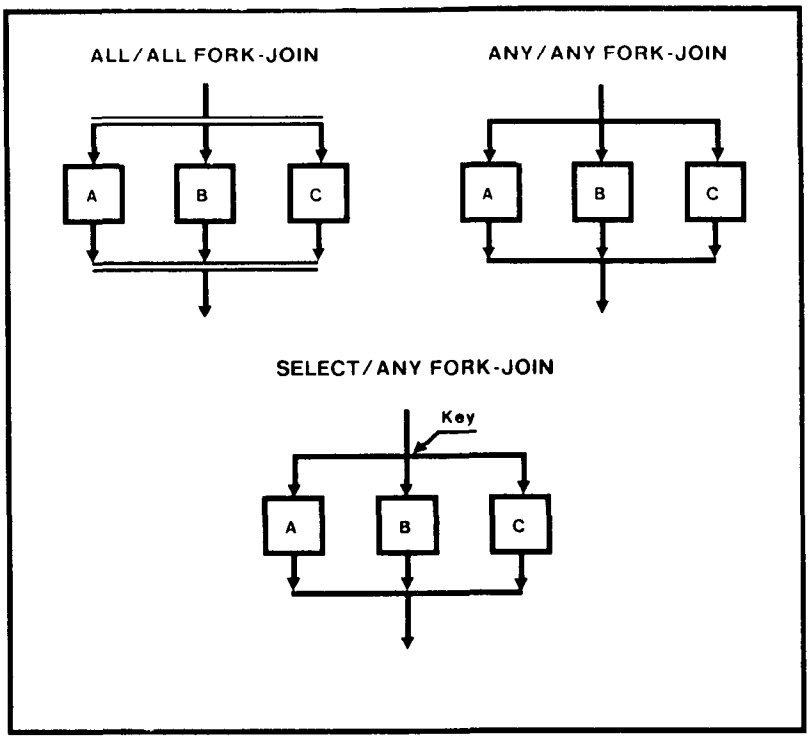

Figure 7 Common fork-join combinations used in the LMA language *Forks* are elements that map control and data from one module to many modules They are annotated graphically as the downward-branching trees in the examples above *Joins* map control and data from many modules to one module The all/all combination is used to start a set of operations going in parallel It indicates completion after all of the operations are complete The any/any combination starts one of several operations and finishes when it is complete A select/any combination uses a key to select a particular operation It is used to implement if-then and case statements in the LMA language

use of the first measure For example, we would argue for the continued use of $\wedge$ (conjunction) and $\vee$ (disjunction) in introductory logic courses in spite of the fact that logical expressions can be written with fewer *kinds of terms* using less familiar connectives. The third measure attempts to account for the interfacing effects in the design by composition model. Terms should be excluded if their composition rules are excessively baroque

These concepts about coverage and simplicity can be illustrated by the knowledge engineering of elements of the LMA language (Stefik, Bobrow, Bell, et al., 1982). The LMA language provides a formal means for synthesizing digital systems in which the sequencing of operations is given primary attention. The sequencing is specified in terms of modules that carry out instructions and links between them that determine the flow and control of information. Flow of control is described in terms of a token-passing protocol between elements. *Forks* are a type of link that enables one module to pass data and control to several other modules (fanout); *joins* are a type of link that combines data and control from several modules into one (fanin). Forks and joins are typically used in fork-join combinations as shown in Figure 7.

The selection of the forks and joins included in the LMA language was intended to provide a small basis set of elements with substantial coverage. In the current set four kinds of forks (any-fork, all-fork, synchronizing-all-fork, and select-fork) and two kinds of joins (all-join and any-join) are included The fork and join vocabulary is interesting from the knowledge engineering point of view in that it illustrates some of the *kinds of arguments* that can be used in deciding what to include in a description language. These arguments arose in the consideration of the possible kinds of "select-forks" for LMA. At one point, we created a chart of possible characteristics as follows:

Possible Selection Characteristics

1 Outside selection by key.

2 Self-selection by ready status

3. Priority-based selection by precedence rules

Possible Synchronization Characteristics

1 All selectees started at once

2 Selectees started when ready

Possible Termination Characteristics

1 At least N selectees activated.

2 No more than N selectees activated

In the current LMA model for select-forks, we chose "outside selection by key" as the selection characteristic and "exactly one selectee activated" as the termination condition If we allowed more than one module to be selected by its ready status, a select fork would start an unpredictable number of modules, perhaps dependent on timing. This would also mean that select-forks would not conserve tokens We have discovered that the rules for composing non-token-conserving elements are remarkably baroque, and that designs that use such elements seem considerably more difficult to understand (*composition simplicity*). Most of the design examples that we considered could be easily described using only the simple token-conserving version of the select-fork The common cases are analogous to *if-then* and *case* statements in conventional programming languages (*expression simplicity*). In addition, the more complex variations of the select-fork can be described in terms of the simpler version and other LMA elements (*basis simplicity*)

Example: Embedding practice in synthesis languages Stacks are familiar storage devices that provide last-in-first-out access to data They are basic to many fundamental algorithms in computer science There are a variety of digital architectures that can be used to create stacks (e.g., Guibas and Liang, 1982) For example, one architecture is like a software implementation, and uses a counter to keep track of pushes and pops. Another architecture uses "marker bits" instead of a counter, to mark the top of the stack. Other architectures resemble large shift registers which either shift the data all at once, or allow it to ripple from one end to the other during pushes and pops These architectures differ in ways that substantially effect the amount of storage needed, the amount of control logic, the fanout of the control lines, and the performance characteristics of a large stack

We observe that practicing designers do not share a common architectural notation adequate for synthesizing or describing these examples. This gap in design knowledge often makes it difficult to share or understand designs. The LMA notation appears expressive enough to admit architectural comparisons and abstract enough to provide leverage for exploring design alternatives. For example, all of the stacks mentioned above have been described in LMA (Stefik, Bobrow, Bell, et al., 1982). From these descriptions one can answer such questions as "how much storage is needed per element of capacity?", "what fanout of control logic is required?", and "what determines the minimum delay between successive push commands?"

The availability of languages can provide opportunities for representing bodies of *ad hoc* practice For example, to describe the design of bit serial circuits for implementing digital filters, one would begin by collecting examples of the design practice. This practice would be partitioned into primitive and composite terms, and composition methods drawn from the *ad hoc* fragments In this example, the practice would include a set of composition rules for combining active elements and buffers according to data rate requirements, as well as some theory about the tradeoffs in this design area. A language like LMA would be used to describe the components. Composition knowledge and tradeoff knowledge would be described in other suitable languages. Throughout this process a knowledge engineer tries to identify concerns that can be isolated and details that can be suppressed. The example illustrates two points:

1 one can describe the terms of an architectural practice as constructs in a synthesis language like LMA, and

2 one can augment the practice and create an embedded architectural language by also creating *composition rules* for the terms

The base languages simplify the process of representing the specialized languages

Significance of these examples. Our interest in formalizing knowledge about VLSI design is akin to other current efforts in AI aimed at formalizing particular bodies of knowledge, such as the physics of fluids (Hayes, 1979) or reasoning about time (e.g., Allen, 1981) This article has been concerned with the design of a body of knowledge in order to give it particular properties

The examples above illustrate that knowledge can be engineered to meet particular objectives Sometimes there are tensions between multiple objectives in the design of knowledge. The composition and optimization example illustrated a tension between simplicity and coverage We sought to keep knowledge simple to facilitate composition, without sacrificing coverage of special cases essential for circuit performance Our approach was to partition the composition knowledge from the optimization knowledge so that they can be applied separately.

The second example illustrated three measures of simplicity that can be employed in the design of synthesis languages These measures reflect a tension between minimizing the number of primitive elements in a language, and keeping short the length of common expressions in the language.

The third example illustrated the idea that design practice can be systematically embedded in synthesis languages, when there is an appropriate match between the important distinctions in the practice and the features emphasized in the language.

These examples also illustrate progress in coping with the difficulties discussed in the first section of this article In that section we noted some characteristics of VLSI design that made prospects for expert systems seem premature given the conventional methods of knowledge engineering.

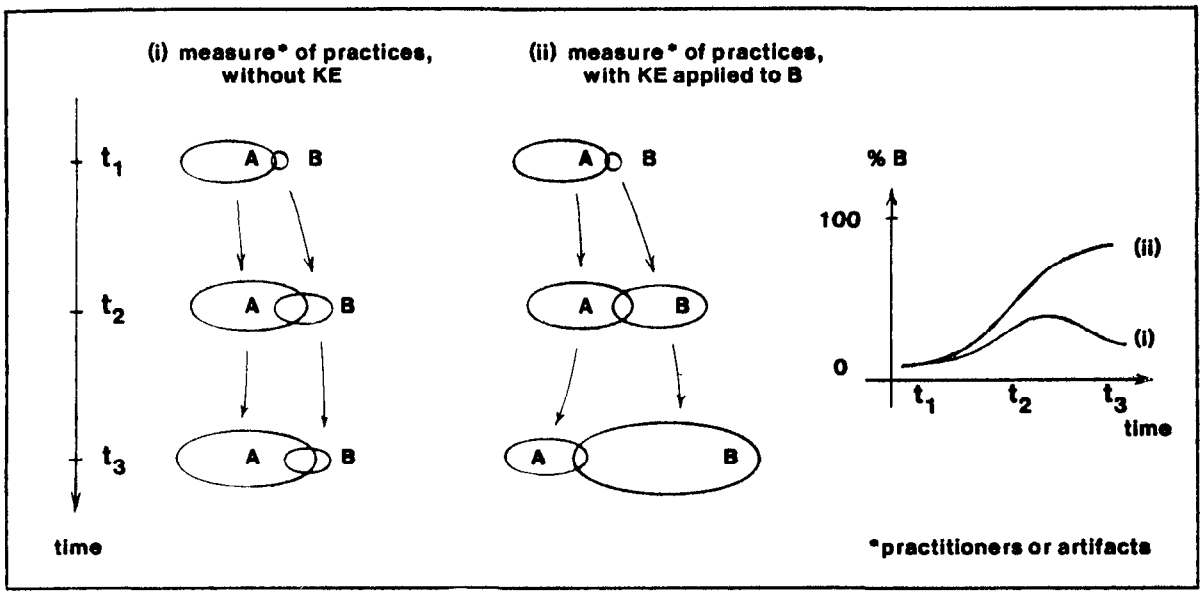

Figure 8 Knowledge engineering mediating the transformation of knowledge The processes that underlie the diffusion of technology and knowledge depend on a variety of factors including properties of the knowledge itself Does it provide economic advantages? Is it too complex to apply? Can it propagate through a particular culture? Knowledge engineering can potentially augment the infrastructure in which these natural transformation, displacement, and diffusion processes operate

The main difficulties were fragmentation of the design community and rapid evolution of the design knowledge. The fragmentation problem can be eased by the use of common languages to represent digital systems in uniform notations. The rate-of-change problem can be eased by the use of languages for abstraction which cover the range of concerns of existing design methods, and which provide insulation from changes in fabrication technology. In contrast, the libraries of standard layout-level cells in current CAD systems are obsoleted quickly by changes in technology. In a multi-level approach, libraries of abstract constructs will span many technologies, and only the implementation rules need be changed as technology shifts.

Speculations on the Potential Impact of Knowledge Engineering

Some AI researchers (e.g., Nilsson, 1982) caution against too great an involvement with the knowledge of "expert" fields, lest AI researchers lose their identities by becoming absorbed by the fields. In contrast, we sense opportunities in substantial involvement The struggle to formalize and mechanize knowledge in difficult problem areas can strongly stimulate the production of new hypotheses regarding the foundations of AI and knowledge engineering. Such problem areas also provide empirical contexts for the experimental testing of those new hypotheses.

Our examples of the engineering of knowledge have highlighted roles for AI specialists. In particular, we have focused on opportunities for exploiting synergy between research on knowledge acquisition and research on problem solving processes. There are possibilities, however, for a much wider range of participation in knowledge engineering. Practitioners in a particular field can apply the techniques to the simplification and refinement of their methods, enabling more efficient application and easier propagation of their knowledge. Cognitive scientists can develop refined models of human information processing by studying the processing, propagation, and evolution of knowledge having known properties. AI specialists, cognitive scientists, and social scientists can collaborate to develop techniques of demography, ethnography, and analysis for identifying areas of *ad hoc* practice ripe for knowledge engineering

The generation, selection, and diffusion of knowledge depend on a variety of social, ecological, and economic factors, including the properties of the knowledge itself. Social structures often mediate the generation process through complex membership and career feedback processes (Latour and Woolgar, 1979). Social networks of knowledge carriers, sometimes invisible to outsiders, can provide a means for rapid diffusion of new knowledge (Crane, 1972). Technological diffusion and displacement are increasingly being scrutinized under quantitative methods, resulting in useful new insights and models of the underlying cultural and economic processes (Sahal, 1981) As we better understand these natural processes, we can propose and test how they might be modified by knowledge engineeering.

We believe that the merging of knowledge engineering into the existing cultural infrastructure can enable great increases in the rates and extents of knowledge generation and diffusion processes (as suggested in Fig. 8). A common literacy regarding the representation and mechanization of practical knowledge would encourage placement

of more effort into the design of knowledge, rather than its routine application. Knowledge engineered for good cognitive matching to receiving cultures will diffuse more rapidly. Knowledge engineered for more efficient computational processing will provide cognitive advantages. Of course, for these results to occur, the field of knowledge engineering must itself successfully integrate into our culture under the operation of natural displacement and diffusion processes!

Special opportunities are presented when knowledge engineering takes on bodies of knowledge of strategic importance, such as design methods in critical technologies. Design methods occupy a central cognitive position for the engineer, much as systems of natural law hold for the physicist. Periods of rapid knowledge displacement among engineers correspond in form to the large-scale cognitive model displacement-processes described by Kuhn (1962) as shifts of paradigm in natural science. As engineered knowledge conveys advantages to its human and machine carriers, the field could modulate and accelerate the currently *ad hoc* natural processes of knowledge generation and diffusion. Our knowledge engineering explorations may ultimately help us to understand the causes, measures, and indeed methods for initiating and controlling, large-scale shifts in the production and application of knowledge

References

Allen, J F. (1981) *An interval-based representation of temporal knowledge* *IJCAI* 7, 221–226

Barstow, D R., & Buchanan, B G (1981) Maxims for knowledge engineering. Tech. Rep HPP–81–4, Computer Science Dept , Stanford University (Also AI Memo 10, Schlumberger-Doll Research Laboratory, Ridgefield, Conn.)

Conway, L (1981) The MPC adventures: Experiences with the generation of VLSI design and implementation methodologies. *Proceedings of the Second Caltech Conference on Very Large Scale Integration,* 5–28 (Also reprinted as Tech Rep VLSI–81–2, Xerox Palo Alto Research Center.)

Crane, D. (1972) *Invisible colleges: Diffusion of knowledge in scientific communities* Chicago: University of Chicago Press

Davis, R. (1982) Teiresias: Applications of meta-level reasoning. In R Davis & D B Lenat (Eds), *Knowledge-based systems in artificial intelligence.* New York: McGraw-Hill

Duda, R O , & Gaschnig, J G. (1981) Knowledge-based expert systems come of age *BYTE* 6(9):238–281

Feigenbaum, E A (1977) The art of artificial intelligence: I. Themes and case studies in knowledge engineering *IJCAI 5,* 1014–1029

Guibas, L J., & Liang, F M (1982) Systolic stacks, queues, and counters. *Proceedings of the Conference on Advanced Research in VLSI,* 155 164.

Hayes, P J. (1979) The naive physics manifesto In D Michie (Ed), *Expert systems in the micro-electronics age* Edinburgh: Edinburgh University Press.

Kuhn, T S. (1962) *The structure of scientific revolutions* Chicago: University of Chicago Press

Latour, B , & Woolgar, S (1979) *Laboratory life: The social construction of scientific facts.* Beverly Hills, Calif : Sage Publications.

Lyon, R. F. (1981) Simplified design rules for VLSI layouts *LAMBDA The Magazine of VLSI Design,* First Quarter, 54–59

Marshall, M , Waller, L , & Wolff, H (1981) The 1981 Award for Achievement *Electronics* 54(21):102 105

Mead, C., & Conway, L (1980) *Introduction to VLSI systems* Reading, Mass : Addison-Wesley

Minsky, M (1961) Steps toward artificial intelligence In E A Feigenbaum & J Feldman (Eds), *Computers and thought* New York: McGraw-Hill.

Nilsson, N J (1982) Artificial intelligence: Engineering, science, or slogan? *The AI Magazine* 3(1):2–9

Robinson, A L (1980a) New ways to make microcircuits smaller *Science* 208:1019–1026.

Robinson, A. L. (1980b) Are VLSI microcircuits too hard to design? *Science* 209:258–262

Sahal, D (1981) *Patterns of technological innovation.* Reading, Mass.: Addison-Wesley

Simon, H A (1981) *The sciences of the artificial* (2nd ed) Cambridge, Mass : The MIT Press

Stefik, M., Aikins, J , Balzer, R , Benoit, J , Birnbaum, L., Hayes-Roth, F., & Sacerdoti, E (1982) The organization of expert systems: A prescriptive tutorial *Artificial Intelligence* 18:135-173

Stefik, M , Bobrow, D , Bell, A , Brown, H , Conway, L., & Tong, C. (1982) The partitioning of concerns in digital system design *Proceedings of the Conference on Advanced Research in VLSI,* 43–52.

Tong, C (1982) A framework for design. Memo KB-VLSI–82 16 (Working paper), Knowledge-based VLSI Design Group, Xerox PARC

Whorf, B. L (1956) The relation of habitual thought and behavior to language In B. Whorf, *Language, thought, and reality* Cambridge: Technology Press