AI Magazine Volume 6 Number 1 (1985) (© AAAI)

Toward Better Models Of The Design Process

Jack Mostow

USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

What are the powerful new ideas in knowledge based design? What important research issues require further investigation? Perhaps the key research problem in AI-based design for the 1980's is to develop better models of the design process. A comprehensive model of design should address the following aspects of the design process: the state of the design; the goal structure of the design process; design decisions; rationales for design decisions; control of the design process; and the role of learning in design This article presents some of the most important ideas emerging from current AI research on design, especially ideas for better models of design It is organized into sections dealing with each of the aspects of design listed above

What is design? Why should we study it? How does it relate to AI? Let's address these questions one at a time.

Design can be viewed as a dialectic between the designer and what is possible (Tong, 1984). The purpose of design is to construct a structure (artifact) description that:

1. Satisfies a given (perhaps informally) functional specification
2. Conforms to limitations of the target medium, *e.g.*, is an executable program in a given language, or is a chip layout for some fabrication technology
3. Meets implicit or explicit requirements on performance and resource usage, *e.g.*, time, space, power, cost
4. Satisfies implicit or explicit design criteria on the form of the artifact, *e.g.*, style, simplicity, testability, maintainability, reliability, reusability, manufacturability, modularity, etc.
5. Satisfies restrictions on the design process itself, such as its length or cost, or the tools available for doing the design (*e.g.*, drafting table versus graphic editor).

Thus, design is largely a process of integrating constraints imposed by the problem, the medium, and the designer.

The design process may have other results besides a description of the designed artifact. For example, the designer may be required to produce documentation on how to use the artifact, a justification that the designed artifact is correct, or a design rationale justifying the design decisions.

Design is an important human activity. As a psychological phenomenon, it is an interesting kind of complex problem-solving, as yet incompletely understood. In economic terms, design pervades our high-tech society. It is probably safe to say that billions of dollars are spent annually on various kinds of design. Moreover, design errors cost an untold amount in lives and property: Consider the potential cost of a single design error in a nuclear reactor, a space shuttle, or a missile system. Thus scientific study of design is easily justified by its potential for improving the cost or reliability of design.

Mechanizing design—moving the design process into the machine—offers to improve both cost and reliability. To the extent that design can be automated, the productivity of human designers can be enhanced. To the extent that assumptions involved in design can be explicitly represented and automatically enforced, design errors resulting from violated assumptions can be avoided.

AI addresses the mechanization of complex, knowledge-intensive tasks; design is certainly such a task. Thus, AI is a natural discipline for the study of design, and in fact a number of AI researchers have recently been studying design, especially hardware and software design. The study is still at an early stage in that much of the design process is still poorly understood, let alone automated. We are still developing our models of the design process.

Developing better models is the key research prob-

This report grew out of the July 1984 "Rutgers Workshop on Knowledge Based Design Aids: Models of the Design Process," attended by a few dozen invited AI researchers working on automated design The workshop was organized by Louis Steinberg; the planning committee included David Barstow, Tom Mitchell, and myself I was appointed to write a report placing the presented ideas in a common framework The report became two papers One of them, published in the SIGART Newsletter (Mostow, 1984c), reports what was said in the workshop talks and panels. In contrast, this paper presents a more personal view of the field I am grateful to Michelle Kraus, Bill Swartout, Steve Minton, Bob Balzer, and David Wile for several suggestions which improved the paper Any errors are of course my own lem in AI-based design today. A comprehensive model of design should address the following aspects of the design process:

1. The state of the design. Design involves a series of artifact descriptions at various levels of detail.

2. The goal structure of the design process. If design is a purposive activity, goals guide the choice of what to do at each point. These goals are not artifact descriptions, but prescribe how those descriptions should be manipulated.

3. Design decisions. Given a goal, there may be several plans for achieving it. Design decisions represent choices among them.

4. Rationales for design decisions. The rationale for choosing a particular plan to achieve a goal explains why the plan is expected to work and why it was selected instead of the alternatives.

5. Control of the design process. Guiding design requires choosing which goal to work on at each point and choosing which plan to achieve it with.

6. The role of learning in design. Solving a design problem requires both general knowledge about the domain and specific knowledge about the problem. Learning is a way to acquire such knowledge.

This article presents two kinds of prescriptions for better design models. **Idea #i** indicates an idea successfully exploited in one or more of the research projects described in (Mostow, 1984c). **Issue #i** indicates a direction in which existing work needs to be extended.

Making the state of the design explicit

We need to improve our representations of the design process. To model the way design really occurs, we need to represent the intermediate states in the design process, *i.e.*, the successive descriptions of the artifact being designed. This requires the ability to represent the partial state of a design in which some but not all design decisions have been made. It is also important to represent abstractions of the design process we use for reasoning about it. (Such abstractions are not only significant as psychological constructs but are also useful as views to be manipulated by automated design aids.) For example, one way to abstract the design process is to omit the dead-end branches explored along the way.

A consensus has emerged that:

Idea #1. *An idealized design history is a useful abstraction of the design process.*

Such a structure is useful in reasoning about designed artifacts. It can play a role in several aspects of design:

- Documentation (Balzer 1984). A record of the decisions leading to a design is helpful in developing and maintaining it.
- Understandability (Scherlis & Scott). A transformational derivation of an algorithm neatly captures its structure.
- Debugging (Kedar-Cabelli, Genesereth, Hamscher & Davis). When the actual behavior of a system fails to match its intended behavior, it is easier to localize the reason for the failure given a record of how the system specification was decomposed into successively smaller components and implemented.
- Verification (Lam & Mostow, Barrow). A formal derivation of a design from a specification via a series of correctness-preserving transformations constitutes a proof that the design implements the specification.
- Analysis (Kelly, Singh). A structured history of a design includes the functional specification and actual implementation of its components at various levels of detail. This information permits simulation of the design at multiple levels of abstraction, as well as reasoning about how a change in one part of the design affects others (constraint propagation).
- Explanation (Swartout). To explain its own behavior, a program needs access to the sort of knowledge that went into its design and would be included in a design history.
- Modification (Steinberg & Mitchell, Wile). If the design of an artifact is properly recorded, redesigning it may be largely accomplished by "replaying" its recorded design history.
- Automation (Balzer *et al.* 1976, Fickas 1982, Green *et al.*, Kowalski & Thomas, Smith, Rich *et al.*, Barstow, Kant & Newell, Brown & Chandrasekaran 1984, Subramanyam). Automatic design may be achieved by applying a series of transformations to derive a design from its specification, or by modifying and combining such derivations.

It is common to characterize a design history as a series of transformations leading from specification to implementation. However, further investigation reveals some important differences between different transformational models of design. In the abstract refinement model, each transformation implements a component or decomposes it into subcomponents (Mitchell *et al.* 1981, Smith, Rich *et al.*, Kant & Newell). Since top-down refinement doesn't handle goal coupling well, systems based on abstract refinement typically use constraint propagation to achieve consistency between different parts of the design. In fact, for routine design tasks, the abstract refinement process may be precompiled into a standard structure of parameterized components; in such cases (for example, air-cylinder design), the design process consists totally of reasoning about constraints in order to determine appropriate parameter values (Brown & Chandrasekaran, 1984).

The transformational model (Balzer *et al.* 1976, Green *et al.*, Lam & Mostow, Scherlis & Scott, Mostow 1981) converts a specification into an implementation via a sequence of correctness-preserving transformations from one complete description to another. A single transformation may operate on several components at once. Thus this model is more general than abstract refinement, which is limited to steps that replace a single component with a more detailed description of it. However, this generality comes at the potential cost of increased complexity, since a transformation on a complete description must deal with more information than a transformation that refines a single component.

Idea #2. *A transformation sequence can be viewed as an executable program for implementing the specification.*

If the specification is modified, it may be possible to reimplement it by replaying the derivation of the original implementation (Darlington & Feather 1979, Wile 1983, Mitchell *et al.* 1983). A specification change may necessitate patching the transformation sequence in places where the original design decisions are no longer appropriate (Carbonell, 1981). However, this is much cleaner than patching the end product, where the effects of the revisions may be widespread.

A couple of more specific ideas for representing the states of the design process and the transformations between them are emerging:

Idea #3. *Dataflow graphs with attached assertions are a useful representation for partial algorithm designs.*

First, dataflow graphs abstract away extraneous details found in program-like textual representations. For example, they can capture decisions about which operations are to be performed, without requiring that their order be fully specified. Second, new information can be added to an annotated dataflow graph in small increments. This makes it a good representation for a fine-grained model of algorithm design. Finally, a dataflow graph can be executed on symbolic and test-case data (Kant & Newell, Rich *et al.).* This is a powerful technique for finding places where the algorithm description is inconsistent or incomplete.

Idea #4. *Source-to-source program transformations are a promising technique for representing hardware design methods.*

By encoding circuits as programs, we can apply our knowledge about programming, debugging, etc., to the less-understood domain of VLSI (Mostow, 1983a). In particular, we can apply program transformation techniques to derive hardware designs correctly and mechanically. For example, to implement a multiplication operator, we can use an algebraic transformation to decompose it into a sum of powers of two. Another benefit representing circuits as programs is that it promotes interchangeability of hardware and software implementations of the same module (Subramanyam).

Making the goal structure explicit

Somewhat surprisingly, most of the systems presented at the workshop leave the goal structure of design implicit.

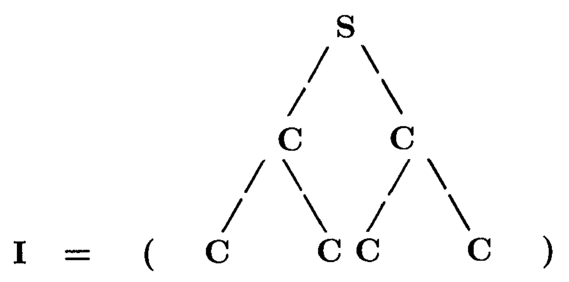
In the pure transformational paradigm, a transformation sequence partly explains how a design was derived, but doesn't explain the purpose of each step. For example, it doesn't distinguish the few transformations that represent key design decisions from the many preparatory steps that serve to transform the design so that a key transformation can be applied. In the abstract refinement paradigm, an unrefined component specification corresponds directly to (and can therefore be viewed as representing) a goal of the form "implement <component>." Subgoals can arise to resolve mismatches between a goal and its initial solution. (In planning, this is called "operator subgoaling.") For example, if a circuit component is implemented using a chip that produces parallel output, but the component needs to produce serial output, the subgoal is "convert the chip output from parallel to serial" (Mitchell *et al.*, 1983). However, other kinds of goals are not represented so directly, *e.g.*, "minimize the number of different chip types used in the circuit," or "design the circuit so as to require a minimum of wiring."

Idea #5. *The goal structure of design can be roughly modeled as a tree.*

The goal tree model makes explicit the goal structure motivating the transformation sequence leading from specification to implementation (Wile 1983, Balzer 1984, Fickas 1982, Mostow 1983, Kant & Newell). Like abstract refinement, this model represents the design history as a kind of tree, but one whose nodes are goals and transformation steps rather than specifications and components. Some goals arising in the course of implementing a high-level program specification might include "reformulate <non-executable construct> into executable code," "optimize <part of program>" and "eliminate <expensive construct>." Since the transformations used to achieve a goal may affect multiple portions of the design, they may give rise to further goals, possibly in a different subtree. Moreover, a transformation may help achieve multiple goals, and goals may be interleaved. However, these phenomena are not explicitly represented in the simple goal tree model. Figures 1 and 2 illustrate the two models.

Idea #6. *An explicit goal structure makes it easier to replay the design process.*

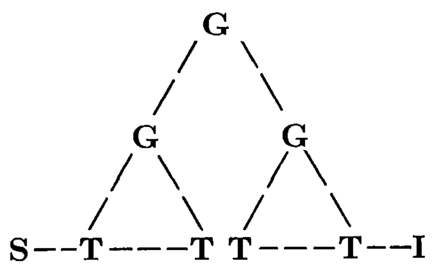
The goal structure motivating the transformation sequence helps the user figure out where to patch it when the replay mechanism encounters a transformation step that no longer applies. For example, if the goal tree includes a goal to eliminate a particular construct, and that construct is absent in the revised specification, the subsequence of transformations under that goal need not be



Abstract refinement decomposes a specification S into components of implementation I.

Figure 1.

replayed. (Wile, 1983) describes a goal tree representation for the development history of a program, and a mechanism for replaying it that lets the user take over at points where replay fails. (Mitchell *et al.*, 1983) describes a similar mechanism used in circuit redesign.



The goal tree model refines a goal into an executable transformation sequence, which converts a specification S into an implementation I.

Figure 2.

Idea #7. *The complete goal structure of a design is too detailed for a human designer to document.*

Thus design should be performed by machine, but guided by the human designer (Scherlis & Scott, Balzer 1981). This human-aided machine design paradigm differs from the machine-aided design paradigm in which the human does the design and the machine provides support tools without understanding the design decisions.

Issue #1. *Goal structure representations should be extended to adequately represent interacting goals.*

A goal tree can represent certain subgoal-supergoal, conjunctive, disjunctive, and sequential relationships, but fails to capture such phenomena as:

- Goal conflicts—two goals cannot both be achieved.
- Goal sharing—achieving a subgoal helps achieve a goal other than its ancestors in the goal tree.
- Goal prerequisites—one goal must be achieved before another goal in a different part of the goal tree (*i.e.*, not its sibling or ancestor).

Issue #2. *Additional kinds of goals in design must be represented.*

While each kind of goal listed below plays an important part in guiding the design process, I know of no design system that explicitly represents all of them.

1. Functionality goals have the form "implement <functional specification>." This kind of goal is represented in the abstract refinement paradigm by an unimplemented specification
2. Performance goals seek to satisfy requirements on the efficiency, cost, reliability, etc., with which the designed artifact satisfies its functional specification. They are often used as criteria for selecting among alternative implementations of a functional specification. Some systems represent such requirements as constraints attached to a functional specification to restrict how it is implemented.
3. Knowledge goals seek to gather information needed to carry out the design. For example, rough design achieves such goals as identifying critical components. These goals ought to be made explicit. The idea of making knowledge acquisition an explicit goal in a problem solver is discussed in (Fox, 1981).
4. Design process goals govern the route taken to arrive at a design, not just the end product itself. For example, limiting the time and cost of the design process is often an important goal, and can be important in explaining the design of the final product. A system that uses such goals explicitly would be reflective, *i.e.*, would reason about its own behavior.

Formalizing these various kinds of goals and representing them explicitly should facilitate clean, flexible control of the design process.

Making the design decisions explicit

The goal tree model represents the goal structure underlying a design, but doesn't explain how a goal is reduced into subgoals, *i.e.*, by what method. It also omits the alternative methods that might have been used to achieve the goal in a different way.

Idea #8. *Decisions can be roughly modeled as choice sets.*

In the explicit choice model, the goal tree is annotated with the competing methods for achieving each goal. For example, (Fickas, 1982) describes a system that constructed an explicit goal tree to record the software development process. Its nodes were program development goals that described how to transform the program description, which was not itself a node in the tree. The system had problem reduction rules (called "methods") for refining goals into subgoals and ultimately into executable program transformations; the leaves of the goal tree formed a sequence of transformations leading from an initial specification-level program description to its final implementation. To select among alternative methods for achieving a development goal, conflict resolution rules called "selection criteria" were used. The goal tree was annotated with the methods applicable to each goal and the selection criteria responsible for which one was actually used.

Issue #3. *Design assumptions and commitments should be made explicit.*

Design can be viewed as a series of commitments and the optimizations that exploit them (Scherlis & Scott). We need to develop representations of the design process, its intermediate states, and its idealized histories, that account for the assumptions and commitments made in the course of design (Balzer, 1984). While systems exist which make assumptions and add constraints, we need to develop mechanisms that explicitly reason about whether to make a particular assumption or commit to a design choice.

Issue #4. *Decision-making should be represented as an explicit goal.*

While systems like Fickas' (Fickas, 1982) partly capture design decisions, listing the set of alternative methods at each point is an incomplete representation of decision-making. We need to develop design models that represent making a decision as an explicit goal which can be reasoned about. For example, given the goal of selecting between two implementation strategies, a designer might create subgoals to construct a rough design based on each, compare them, and choose accordingly Thus learning, in the sense of gathering information about a particular design problem, can be viewed as a deliberate activity in design. More on this later.

For instance, LIBRA (Kant, 1979) used a combination of decision heuristics and algorithm analysis to guide the process of program synthesis by estimating the efficiency of the programs produced by alternative implementation strategies. Such estimates were used to choose what order to work on implementation goals, what order to consider alternative implementations, and which ones to choose. LIBRA used analysis to identify potential bottlenecks in the program, worked on them earlier, and allocated design resources in proportion to the estimated impact of the decision. If there were sufficient resources, it would expand alternatives into more detail in order to obtain more accurate cost estimates for them. One can view this top-down expansion of an implementation alternative as a form of rough design.

In general, before going ahead with rough designs, a designer might estimate how long they would take and whether the expected difference between the alternative implementation strategies warrants the extra effort. The designer might even reason about whether to make such an estimate, and whether to reason about that, and so forth. To cut off all this contemplation and actually make a decision, a processing architecture is needed—a built-in interpreter that knows when to stop thinking about decisions and actually make them. This is true of problem-solving in general, not just design. (Laird & Newell, 1983) provides one example of a general problem solving architecture in which decisions about what to do next are represented explicitly.

A comprehensive model of decision-making would explicitly represent the following processes:

1. Frame a decision to be made. (How does a designer realize a decision is needed?)
2. Generate alternatives (When is it OK to stop?)
3 Establish criteria for comparing them. (How do these relate to higher-level goals?)
4 Evaluate them according to those criteria (How careful should the evaluation be?)
5 Choose an acceptable alternative. (What constitutes enough information to choose?)
6. Retract it if it proves unsatisfactory. (When should a decision be reconsidered?)

Existing design systems tend not to deal explicitly with all these aspects. In particular, the simplistic model of a decision as selecting among an explicit set of alternatives ignores the problems of framing the decision, generating the alternatives, and choosing selection criteria.

Making the design rationale explicit

One can ask different kinds of "why" questions about a design:

1. Why did the designer perform that transformation or solve that subgoal? *Answer: To achieve its supergoal in the goal tree.*
2 Why did the designer think a plan to perform some transformations or solve some subgoals would achieve its goal? *Answer: A proof, or less formal justification, of the plan for achieving the goal*
3 Why did the designer choose one plan rather than another? *Answer: A proof or other explanation of why that plan satisfies a given set of design criteria better than the alternatives.*
4. Why did the designer use a particular set of criteria to compare alternatives? *Answer: A proof or other explanation of why those criteria are appropriate, given the tradeoffs of the design space and the designer's preferences.*

A design rationale explains why a goal was achieved in a particular way. The " correctness" rationale explains why the particular plan used to achieve the goal ought to work. The "appropriateness" rationale explains why that plan was chosen instead of some other plan.

Idea #9. *Rationales are useful in replaying the design history to solve a new problem.*

If a goal in the old problem was solved using some plan, and the reasons it worked also hold true in the new problem, the plan ought to work in the new problem as well. Similarly, if the reasons for selecting that plan are still valid in the new problem, the same plan should be selected; otherwise, some other plan might be more appropriate (Carbonell 1983).

Idea #10. *Formally proving that a component satisfies its specification has uses besides the obvious one of verification.*

The CYPRESS system (Smith) uses a constructive proof to derive a specification of a component from specifications of related components and how it's related to them. CYPRESS has a general Divide and Conquer schema with a Decompose component that splits an input problem into two subproblems and a Compose component that combines their solutions. Given a particular choice of a Decompose operator for some problem (*e.g.*, sorting), CYPRESS deduces what the Compose operator must be, and vice versa.

A proof that a component satisfies its specification can be generalized into a rule for implementing similar components automatically in the future (Mitchell et al, 1984a). The Rutgers Learning Apprentice project uses correctness rationales to learn general rules for solving goals of the form "Design a circuit that satisfies <specification>": it generalizes from an example of a particular hand-designed circuit that satisfies a specification and a rationale for why it does. However, representing appropriateness rationales is much harder, since they involve look-ahead: a design decision is most naturally explained in terms of its eventual impact on parts of the design that have not been completed at the time the decision is made. (Fickas, 1982) included some rules for selecting among alternative designs, but they were rather weak and avoided this problem of representing something that isn't there yet. In principle, given an appropriateness rationale one could generalize it into a rule for selecting among implementations; see Learning and Design, below.

Idea #11. *What's good for explanation is good for design.*

The ability to generate machine explanations of a designed artifact is not an add-on feature; it strongly influences the design process, since it requires a design rationale so detailed that the most feasible way to construct it is for the design to be generated by machine. Building in an explanation capability forces the reasoning behind the design to be made explicit—which tends to lead to a more principled design (Swartout).

Understanding how to control the design process.

How do designers decide what to do next? We need to uncover the reasoning behind such decisions and represent it explicitly.

Issue #5. *We need to explicate strategies for how to handle interacting goals.*

Given two goals, various relationships between them are possible:

- *Independence:* The goals do not affect each other.
- *Cooperation:* Achieving one goal makes it easier to achieve the other.
- *Competition:* One goal can be achieved only at the expense of the other.
- *Interference:* One goal must be achieved in a way that takes the other goal into account.

Some research questions concerning multiple goals are:

1. How can the relationship between two goals be inferred?
2. What control strategies are appropriate to this relationship?
3. If more than one control strategy is appropriate, which one should be used?

Questions 1 and 3 appear to be open problems. Question 2 is partly addressed below by listing some control strategies suited to the various goal relationships.

Independent goals can be achieved in any order, with the same net result. Thus no special control strategy is needed to order them; this decision can be made arbitrarily, or based on other factors, *e.g.*, relationships to other goals.

Cooperative goals should be achieved in whichever order best exploits the relationship among them:

- *Achieve prerequisite first:* If one goal satisfies a precondition for achieving the other, achieving it first may generate information (*e.g.*, variable bindings) needed to solve the second goal.
- *Achieve more general goal first:* If one goal subsumes the other, achieving it first satisfies the less general goal without any additional work. This usually makes more sense than solving the simpler goal first and then trying to extend the solution to cover the more general case—but not always:
- *Learn by solving easier goal first:* If two goals are similar but one is harder, solving the easier goal first may serve as a way to generate additional knowledge useful in solving the harder goal. This strategy can be effective if the designer is capable of learning from experience.

Competitive goals must be integrated according to their relative importance:

- *Sacrifice less important goal:* If one goal completely dominates the other, ignore the less important goal.
- *Relax goal:* If both goals are important, relax one of them to a version that is weaker than the original but compatible with the other.
- *Treat as trade-off:* If the goals are relative preferences rather than absolute predicates, treat the competitive relationship (*e.g.*, between time-efficiency and space-efficiency) as a tradeoff, and choose a compromise solution to optimize (or satisfice) some overall utility function (*e.g.*, minimize a weighted sum of time and space).

Interacting goals can be achieved in several ways, depending on the nature of the interaction. For example, suppose one goal is to implement function F as a VLSI circuit, and another goal is to make the circuit fit on a single chip. Several strategies are possible:

- *Achieve goals sequentially:* First solve one goal, and then transform its solution to achieve the other goal. For example, first implement a circuit to compute F without worrying about area, and then use a compaction algorithm to make it fit on one chip. This strategy runs into difficulty when commitments made in the course of solving the first goal make it hard to solve the second.
- *Defer commitments:* Order the goals so as to start with whichever decisions impose fewest restrictions on the form of the solution. The idea is to postpone decisions not forced by the problem, thereby leaving as much freedom as possible for achieving subsequent goals.
- *Make critical decisions first:* Order the goals so as to start with whichever decisions are most constrained by the problem. For example, design the most constrained component first. Postponing the constrained decision until later in the design would increase the risk of dead ends, since design commitments made in the interim might well render the already highly constrained decision over-constrained. By solving the most constrained problem first, before making such commitments, such dead ends can be avoided. This idea is closely related to but distinct from the notion of deferring commitments. The critical path strategy says to make a decision as early as possible if it is highly constrained by the design task, while the deferred commitment strategy says to postpone a decision as long as possible if it is not.
- *Merge goals:* Conjoin the two goals into a single specification and implement it. In the example at hand, that would correspond to the single goal "Implement-F-on-a-chip," which might be useful given methods for solving that goal directly. For this example, however, the strategy appears useless, perhaps because the two goals address such different aspects of the design (functionality and area-efficiency). Goal merging may be more feasible for goals with a common generalization. For example, the goals "implement a circuit to compute A*x + y" and "implement a circuit to compute x + B*y" can be merged into the single goal "implement a circuit to compute A*x + B*y" (assuming one circuit can be shared for both purposes).
- *Use goal as selection criterion:* Use one goal as a criterion for selecting among different solutions to the other, in the hope that the goal being used as a selection criterion will be achieved as well. For example, as the function F is decomposed into primitives that can be implemented in hardware, choose the smallest implementation for each primitive. With luck, the resulting circuit will fit on one chip. This method of integrating two goals is not guaranteed to solve them both.
- *Combine orderings:* If both goals can be used as selection criteria to order some set of choices, they can be combined erging the orderings into a partial or total ordering. For instance, suppose the choices are alternatives for how to implement some function, and the goals are "optimize time" and "optimize space." Each goal induces a different ordering on how the function is implemented. For example, if a goal constrains some resource consumed by the design, it can be used to order implementation alternatives according to some metric on the amount of resource they consume. Two such goals can be merged into a single selection criterion by using some function to combine the two metrics, *e.g.*, "minimize A*time + B*space", where A and B reflect the relative importance of the two goals. Alternatively, if one goal is absolutely more important than the other, the secondary goal can be used solely to break ties—*i.e.*, to choose among the alternatives ranked best by the primary goal.
- *Use goal to budget:* Decompose one goal into subgoals parallel with the decomposition of the other. For example, given a decomposition of F into parts, split the total chip area into allocations for each part. If each part is implemented so as to fit into its budgeted area, the circuit will fit. This strategy requires a good decomposition—in this case, an accurate estimate of the area required for each part of the circuit. When such estimates fail, some negotiation is needed to adjust the allocations.

In practice, these control strategies are used in combination. For example, consider the interactions between the goals of testability and area-efficiency in the design of a VLSI chip. A designer might achieve these goals sequentially by first transforming the circuit to make it testable (*i.e.*, make each piece of circuit state settable and observable), and then applying a compaction algorithm to produce an area-efficient layout. However, there are several methods for achieving testability, which vary greatly in the amount of chip area they use. Thus area-efficiency should be used as a selection criterion in choosing among them. Alternatively, the designer might use an area budget, allocating a certain amount of chip area for test circuitry. This constraint would then constrain the solution of the testability goal.

It is particularly interesting to study how goals interact prior to the point where they can be integrated with each other or incorporated into a functional specification. For example, what reasoning does a designer use in reformulating a goal like area-efficiency into a selection criterion or budget?

Since design must solve many goals simultaneously, it is important to study and encode general strategies for integrating goals. The list of strategies mentioned here should be refined and extended.

Idea #12. *One way to decompose a design problem is to design components separately and then design interfaces between them.*

Interface design is an important part of bottom-up design, in which components are designed in detail before being assembled into a higher-level structure. The SYS program for designing systolic circuits (Lam & Mostow) provides an example of this process. SYS identifies the primitive computations in a given algorithm and designs cells (at the functional level) to compute them. To fit them together into a circuit that implements the algorithm, it must decide which outputs to connect to which inputs, and what delay elements to insert between cells so that the timing works out correctly. That is, to glue subcircuits together, it must design an interface between them.

A similar process may arise in semi-customized design systems that use libraries of VLSI cells, subroutines, design histories, or domain specifications. For example, the REDESIGN system (Mitchell *et al.*, 1983) deals with digital circuits composed out of commercially available chips. Using a chip to implement an abstract component may require designing an interface to satisfy constraints on the abstract component that are not satisfied by the chip. For instance, if the chip produces parallel output, but the component needs to produce serial output, the interface can be a shift register. An example of the interface problem in the software domain is the cut-and-paste process of melding together different, possibly overlapping or inconsistent formalizations of the same domain (Fickas, 1984b).

An interesting issue is how to design reusable building blocks that can be easily combined. Adopting standard interface conventions is one example of a design strategy that promotes reusability.

Issue #6. *We need to integrate heuristic and algorithmic methods to perform design tasks involving large amounts of search.*

The TALIB system (Kim) illustrates a nice clean approach to integrating algorithms and heuristics: use algorithms that take advice, speeding up as they get more knowledge. TALIB's layout algorithm accepts topological constraints and searches for a layout that satisfies them. Adding constraints can speed up the search.

Roach (1984) investigates the problem of integrating heuristic and algorithmic knowledge to generate circuit layouts.

Idea #13. *Symbolic execution and test-case simulation are a crucial part of the design process.*

Symbolic execution calls attention to missing and inconsistent components (Kant & Newell, Adelson & Soloway). Trying out concrete test cases allows humans to apply knowledge they can't use in the general case—for example, a person can use visual reasoning to draw a convex hull around a particular set of points, and then generalize observations about the example into conclusions useful for designing an algorithm. In particular, such an observation may provide an "Aha!" solution to an unsolved goal for a mind prepared by a previous attempt to solve it. That is, "Aha!" design may consist of creating structure to solve one problem, and then perceiving in it the solution to another (Kant & Newell).

Issue #7. *We need to model compiled goals and choices.*

A comprehensive deep model of a rational design process ought to make its goal structure explicit. However, the fact that a process has some goal doesn't necessarily mean that a system for achieving that goal must represent it as an explicit data structure and manipulate that structure in the course of the process. In both people and programs, certain design concerns are compiled out. For example:

- People are not aware of all the goals in their mental processing; work on human problem-solving has showed how goal-oriented novice behavior evolves into more procedural expert behavior (Larkin, Anderson *et al.).* An accurate *cognitive* model of expert behavior on routine design tasks would quite properly leave implicit many goals that the experts don't think about while designing.
- In the register allocator of a compiler, goal structure and choice points have been compiled out. Rather than trying different assignments of variables to registers in order to find a good one, the allocator uses an algorithmic method to achieve the same effect. That is, some of the constraints on the designed artifact have been compiled into the design method—a characteristic of good design, according to (Tong, 1984).
- The transformational paradigm as a whole compiles in the general correctness goal of making the designed

artifact satisfy its specification. It does this by restricting the set of available implementation operators to correctness-preserving transformations.

Thus a design *system* need not represent all goals of a design explicitly. Nonetheless, a complete explanation of the system (or the design it produces) should recognize these goals and describe how the system achieves them.

Issue #8. *Cognitive models of design deserve further investigation.*

One reason for psychological study of design is to improve our models of the design process (Adelson & Soloway, Brown & Chandrasekaran 1984, Kant & Newell). Top-down, goal-directed processing is only one way to control design. Evidence from protocol studies reveals other control mechanisms at work. For example, designers seem to create demons that remind them to resume some goal when missing required information becomes available (Adelson & Soloway). They have effective strategies for recovering from failures (Brown & Chandrasekaran 1984) and repairing bugs (Sussman 1973, Kant & Newell). While much of design operates breadth-first, expert designers are good at identifying critical components and plunging depth-first to investigate them, sometimes by carrying out a rough design that approximates other elements (Adelson & Soloway, Brown & Chandrasekaran 1984, Kowalski & Thomas).

Another reason for cognitive studies of design is to help us develop paradigms for human-machine cooperation based on a principled division of labor that exploits each party's strengths and compensates for their weaknesses (Mitchell *et al.*, 1984b). (Mostow, 1984b) presents a taxonomy of 15 possible divisions of labor based on who (human or machine) does what (makes a decision or carries it out) when (before, during, or after machine execution). For example, in an interactive transformational model, the human repeatedly selects a transformation and the machine applies it. In an annotated-input model, the human annotates components of the input specification to show how they should be implemented, and the machine later implements them accordingly. Both these models combine human expertise in decision making (hard to explicate in machine-understandable terms) with the machine's ability to manage all the details (hard for humans to remember) involved in carrying out a design decision.

Investigating the role of learning in design

It has been hypothesized that "problem solving and learning are inalienable, concurrent processes in the human cognitive system" (Carbonell, 1981). People acquire various sorts of design knowledge when they solve (or fail to solve) a design problem. This includes both knowledge about the specific problem and general knowledge about design in the problem domain.

Issue #9. *Learning about a specific design problem is an important but poorly understood process requiring further study.*

Human designers deliberately learn about a design problem by doing one or more rough designs. What they learn in the process enables them to produce a better design the next time around.

Design systems exist that incorporate this capability to some degree. The TALIB system (Kim) critiques the layout it designs, generating additional constraints to guide the next iteration. Some other design systems go through a rough-design phase (Brown & Chandrasekaran, Kowalski). Such problem-specific learning is a powerful technique for reducing the search through the design space.

Idea #14. *Search can be viewed as a kind of learning.*

Consider a search through some space. Each time a candidate solution fails, the searcher "learns" not to try that particular solution. (This assumes that the searcher doesn't consider the same solution more than once.) A sophisticated searcher may eliminate a whole branch of the search based on such a failure; that is, it "learns" not to try a more general class of solutions. Moreover, it may also use this information to decide where to look next. Thus learning about a problem can be viewed as an extremely sophisticated form of search, which exploits information about previous attempts to eliminate large portions of the search space and steer subsequent attempts in more promising directions. People use this kind of learning to converge rapidly on a solution; we need to find out how they do it.

Idea #15. *Learning by watching over an expert's shoulder is a fruitful paradigm for acquiring design knowledge.*

For example, Mitchell et al. (1984a) proposes a Learning Apprentice to acquire rules for implementing and decomposing components. It relies on the ability to construct and generalize a proof that a manually designed circuit satisfies its specification (Mahadevan, 1984). The generalized proof is converted into a general rule for implementing (or decomposing) similar components in the future, just as ABSTRIPS (Fikes *et al.*, 1972) generalized its robot plans for future use.

Issue #10. *Techniques for acquiring informal design knowledge need to be developed.*

While generalizing a correctness proof into an implementation rule is an elegant way to acquire one kind of design knowledge, what about other kinds of design knowledge, where such proofs are infeasible?

For example, consider the problem of acquiring the knowledge used to choose which method should be applied to a given goal. Human designers select one design method over another based on such factors as the ease of applying it, the likelihood that it will lead to a dead end (*e.g.*, require unavailable information), and its predicted impact on the eventual design.

It is not feasible to learn these factors by watching an expert designer and formally proving that he or she chose the "right" method (relative to some criterion such as optimal result or minimal design cost). Such a proof would necessarily refer to such external factors as the competing methods for this choice, the methods available to complete the design, and the form of the eventual final design, which is not known at the time the decision is made (Mostow, 1984b).

People use informal rationales to justify design decisions, based on such empirical reasons as "this method has worked well for me in the past." An interesting research problem is how to infer such informal rationales (perhaps given some hints from the designer) and generalize them into heuristics for selecting methods. DeJong (1983) deals with the related problem of understanding the actions of r a character in a story and abstracting them into a general plan.

Issue #11. *We need to find out how to acquire general design knowledge from experience.*

Somehow people manage to acquire knowledge that helps them guide the design process. Automating this acquisition of search control knowledge for design is a challenging research task. It has been studied for certain simpler kinds of problem solving.

For example, Langley (1983) describes the SAGE 2 system, which learns search heuristics for solving puzzles. Given the results of a breadth-first search through a state space, SAGE.2 assigns credit to moves on the solution path. It assigns blame to moves leading off the path, to previously visited states, or to dead ends; the latter two heuristics allow SAGE.2 to start learning before the search terminates, even if it proves unsuccessful. SAGE.2 uses discrimination learning to develop search control rules that tend toward a shortest-path solution by favoring good moves and avoiding bad ones.

Mitchell (1983) describes the LEX system for learning selection heuristics for symbolic integration. Each heuristic specifies a class of integration problems (goals) for which a particular integration operator is appropriate—*i.e.*, the operator leads to a solution (or a shortest-path solution, depending on the learning criterion used). While LEX1 used version space induction to learn heuristics, LEX2 used the technique later adopted in the Learning Apprentice: given an integration problem solved by a sequence of operators found using brute force, prove that this sequence leads to a (shortest-path) solution, and extract from the proof the general conditions under which this will occur.

Learning how to guide the design process will require more sophisticated techniques than those used in SAGE and LEX. For example, whether a method is appropriate for a goal may depend not only on its likelihood of leading to a solution and on the cost of applying it, but also on the resources (time, space, power) used in the resulting design. SAGE.2 and LEX didn't have to worry about differences between alternative solutions to a given puzzle or integration problem. Also, since design involves many interacting goals, it is important to learn which goal to work on next. SAGE 2 operated in a state space with a single goal state and no subgoals. In LEX, the implicit goal was to get rid of the integral signs; multiple integrals in an expression constituted conjunctive subgoals, which were independent and could therefore be solved in any order.

EURISKO (Lenat, 1983) can be viewed as learning heuristics for (among other things) deciding what to work on next. In EURISKO, such heuristics operate by determining the priority of proposed tasks on the agenda. In practice, these heuristics are based on the results of performing similar tasks in the past; tasks that ultimately lead to interesting results are considered successful, and the rules that proposed them and rated them highly are reinforced. In principle, heuristics in EURISKO could control the order in which tasks are performed by using information about which tasks have already been performed. Since EURISKO learns heuristics and has been successfully applied to at least one design problem, it is worth investigating the effectiveness of the EURISKO paradigm for acquiring heuristics to control an agenda-based design process. (Lenat and EURISKO together designed space fleets that won a national competition. The design process was evolutionary in nature; fleets that performed well were considered highly interesting and were selected for further experimentation. EURISKO has also been used to discover "high rise" VLSI configurations, but such discovery is more open-ended than designing an artifact to satisfy a given specification. In such open-ended tasks, the order in which two goals are addressed may not matter very much, assuming both goals are eventually achieved.)

Conclusion

Idea #16. *Design operates in multiple problem spaces.*

It is already a commonplace in AI that design operates at different levels of abstraction. That is, the designed artifact is represented (both functionally and structurally) at more than one level of detail. However, it is now becoming apparent that design also operates in qualitatively different problem spaces (Kant & Newell). That is, the space of artifact descriptions is just one of the problem spaces involved in design. A comprehensive model of design will have to account for the following kinds of entities constructed during the design process.

1. *Artifact descriptions* represent the designed artifact, or a fragment of it, at some level of detail. Functional descriptions represent the artifact in terms of its desired properties. For example, an input-output specification can be used as a functional description of a circuit

or procedure. Structural descriptions represent the artifact in terms of its parts. A schematic is a structural description of a circuit, while executable code is a structural description of a procedure.

2. *Goals* prescribe what to do to the artifact descriptions, *e.g.*—"reformulate <specification> in terms of implementation-level constructs" or "reduce the space used by <structure>." Constraints are attached to component descriptions to represent properties the design should satisfy. This is often achieved by propagating the constraint to a component that can be implemented in a way that satisfies the constraint. A selection criterion is a goal used in selecting among alternatives. Using a goal as a selection criterion is a way to integrate two goals (Issue #5). Typically one goal has the form "implement <functional specification>" and the other goal has the form "make the result satisfy <design desiderata>." The goals are integrated by using the second, vaguer goal as a selection criterion for comparing alternative solutions to the first, more concrete goal. Some researchers prefer to reserve the term "goal" for the more concrete kind, and use terms like "design desiderata" to refer to the fuzzier kind.

I suspect that very high-level design—the initial phase of the design process, which may go on completely in the designer's head, even before the paper and pencil stage—operates mostly within the goal space and between it and the space of fragment descriptions. This is because very high-level goals are hard to integrate at first into a complete specification. Only subsequently does the design problem become sufficiently well structured to allow top-down refinement of the complete specification.

3. Beliefs about a design consist of additional information the designer derives from artifact descriptions by means of *analysis, simulation, symbolic execution, heuristic assumption,* etc. Not every true assertion about a design is a belief—only those the designer explicitly knows (or assumes). Moreover, not every belief is true—for example, a design may be based on simplifying assumptions that only approximate reality (Issue #3). Beliefs represent knowledge that is not initially explicit in the artifact descriptions, but is generated in the course of reasoning about a design and used to guide the design process.

4. *Examples* are useful in discovering properties of a design. For example, simulating the behavior of a design on test data can reveal bugs (Idea #13) or generate performance estimates.

5. *Justifications* show how a belief is derived. A formal proof is derived logically, while an informal rationale may rely on heuristic inference rules or empirical findings. Justifications can be useful in *deriving* (Idea #10), *replaying* (Idea #9), *explaining* (Idea #11), or *generalizing* (Idea #15) the design.

6. Decisions—explicitly represent competition among alternatives. If these are represented as first-class objects, they can be referred to by goals, beliefs, etc. This allows reflective design models in which making a decision (in particular, selecting which operator to apply) can be an explicit goal, solvable using other operators (Issue #4).

A central aim of current AI research in design is to make explicit these various aspects of the design process so that they can be formalized and modeled in the machine. We need good representations for the successive states in the design process; the operators for getting from one state to another; the goals pursued in the course of design; the decisions that are identified, analyzed, and made; the rationales for these decisions; the assumptions on which they are based; and the control strategies for guiding the design process.

Representing these entities explicitly should make it easier to capture different kinds of design knowledge for use in automating design. In fact, a good test of any design system or model is whether it can represent and exploit the various kinds of knowledge people learn about design.

Conversely, a good question to ask about a system that learns to design is what kinds of design knowledge it can acquire. In particular:

- Can it learn new methods for solving design goals? For example, can it acquire new operators for decomposing specifications into components, implementing them, or optimizing the result?
- Can it learn criteria for which method to apply to a given goal? For example, can it learn to predict the relative impact of different methods on the utilization of various resources in the eventual design?
- Can it learn strategies for which goal to work on at each point? For example, can it learn to identify the critical components that should be designed first?

A second question that pertains to a learner is the generality of what it learns:

- Does it learn knowledge specific to an individual design problem?
- Does it acquire knowledge applicable to some broad domain of design?

Third, a learner can be classified according to where it gets the data from which it learns. For instance:

- Does it learn from its own design experience?
- Does it learn by observing the behavior of an expert designer?

These three dimensions define a taxonomy of learning in design. For example, doing a rough design (Idea #9) to identify critical components is a way to learn a strategy for what order to solve goals in a specific problem, based on experience. In contrast, generalizing a correctness proof for a manually designed circuit (Idea #15) is a way to learn a design method applicable over a general domain, based on observation.

The field of machine learning has far to go, and building systems that learn about design is especially difficult because we don't yet know how to represent many of the kinds of design knowledge we might want such systems to acquire. That the area of learning in design is at a very early stage is illustrated by the fact that very few of the $3 \times 2 \times 2$ combinations defined above are as yet instantiated in real systems. As the representational problems are solved over the next several years, more of these learning modes should be explored.

The various aspects of design discussed in this paper must be better understood in order to build design systems whose partnership with human designers is based on a shared model of the design process. This paper has tried to identify these aspects, communicate some of the powerful ideas for dealing with them emerging from current AI research, and point out areas where existing systems are limited and further work is required. If the paper has expanded the reader's model of design by providing useful insight into the nature of the design process, it has served its purpose well.

References

Adelson, B. & E. Soloway (1984) A cognitive model of software design. Tech. Rep. 342, Dept. of Computer Science, Yale University

Anderson, J. R , J. G. Greeno, P. J. Kline, D. M. Neves (1981) Acquisition of problem-solving skill In John R Anderson (Ed.), *Cognitive Skills and their Acquisition* Hillsdale, NJ: Lawrence Erlbaum Associates.

Balzer, R. (1981) Transformation implementation: an example. *IEEE Transactions on Software Engineering* SE-7: 3-14.

Balzer, R., N. Goldman, & D Wile. (1976) On the transformational implementation approach to programming *Proc* 2nd International Conf. Software Engineering. 337-343.

Balzer, R (1984) Capturing the design process in the machine Talk presented at Rutgers Workshop on Knowledge-Based Design Aids: Models of the Design Process

Barrow, H. G. (1984) Proving the Correctness of Digital Hardware Designs. *AAAI-83*:17

Barrow, H. G. (1984) VERIFY: A program for Proving Correctness of Digital Hardware Designs *Artificial Intelligence* (forthcoming).

Barstow, D. (1984) A Perspective on Automatic Programming. *AI Magazine* 5: 5.

Brown, D. C. & B. Chandrasekaran. (1984) Expert systems for a class of mechanical design activity. *Proc. IFIP WG5.2 Working Conf. on Knowledge Engineering in Computer Aided Design.*

Carbonell, J G. (1981) A computational model of analogical problem solving. *IJCAI 7*: 147.

Carbonell, J. G. (1983) Derivational analogy and its role in problem solving. *AAAI-83*:64.

Darlington, J. & M. Feather (1979) A transformational approach to modification. Tech. Rep. 80/3, Imperial College, London.

DeJong, G (1983) Acquiring schemata through understanding and generalizing plans *IJCAI 8*: 462.

Fickas, S (1982) Automating the transformational development of software. Ph D Thesis, Computer Science Dept., University of California at Irvine.

Fickas, S. (1983) Automating software development: a small example. In *Symposium on Application and Assessment of Automated Software Development Tools.*

Fickas, S. (1984a) A Problem Solving Approach to Software Development. Tech. Rep , Computer Science Dept., University of Oregon.

Fickas, S. (1984b) Mechanizing software specification: a proposal. Tech. Rep. CS84-1, Computer Science Dept., University of Oregon.

Fikes, R., P. Hart, & N. Nilsson (1977) Learning and executing generalized robot plans. *Artificial Intelligence 3*, 2: 251

Fox, M. (1981) Reasoning with incomplete knowledge in a resource-limited environment: integrating reasoning and knowledge acquisition. *IJCAI 7*: 313

Freeman, P. (1983) Fundamentals of design. In Peter Freeman & Anthony I. Wasserman (Eds.), *Tutorial on Software Design Techniques.* 4th ed. IEEE Computer Society.

Genesereth, M. R. (1982) Diagnosis using hierarchical design models. *AAAI-82*:278.

Genesereth, M R. (1984) The Use of Design Descriptions in Automated Diagnosis. Stanford Heuristic Programming Project, HPP-81-20.

Green, C , J. Phillips, S Westfold, T Pressburger, S. Angebranndt, B. Kedzierski, B. Mont-Reynaud, & D. Chapiro. (1981) Toward a knowledge-based programming system. Tech Rep. KES.U 81.1, Kestrel Institute

Hamscher, W & R. Davis. (1984) Diagnosing circuits with state: An inherently underconstrained problem. *AAAI-84*:142

Kant, E (1979) A knowledge-based approach to using efficience estimation in program synthesis. *IJCAI 6*: 457.

Kant, E & A Newell (1982) Naive Algorithm Design Techniques: A Case Study. *Proceedings of the European Conference on Artificial Intelligence.*

Kant, E. & A. Newell (1984) Problem Solving Techniques for the Design of Algorithms. *Information Processing and Management.* Pergamon Press Also appears as Tech. Rep. CMU-CS-82-145, Computer Science Dept Carnegie-Mellon University.

Kant, E. & A Newell (1983) An Automatic Algorithm Designer: An initial implementation. *AAAI-83*:177.

Katz, R., P. A Subrahmanyam, & W. Scacci (1984) Design Environments for VSLI and Software. *Journal of Systems and Software 4*, 1:13.

Kedar-Cabelli, S (1983) An artificial intelligence approach to VLSI chip debugging Rutgers AI/VLSI Project Working Paper No. 14

Kelly, Van E. The CRITTER system—automating critiquing of digital circuit designs. *Proc 21st Design Automation Conference IEEE* June 1984 Also, Rutgers AI/VLSI Project Working Paper No. 13.

Kim, J. (1984) Exploiting domain knowledge in IC cell layout. *IEEE Design and Test* August.

Kim, J. Application of domain knowledge in computer aids for IC layout. Ph.D. dissertation (forthcoming). Computer Science Department, Carnegie-Mellon University.

Kowalski, T. J. & Thomas, D. E. (1983) The VLSI design automation assistant: first steps. *Twenty-sixth IEEE Computer Society International Conference* pp. 126-130.

Laird, J., & Newell, A. (1983) A universal weak method. Computer Science Department, Carnegie-Mellon University. Tech. Rep. CMU-CS-83-141,

Lam, M. & Mostow, J. (1983) A transformational model of VLSI systolic design. *Proc. IFIP WG 10.2 Sixth International Symposium on Computer Hardware Description Languages and their Applications* Pittsburgh, PA.

Langley, P. (1983) Learning effective search heuristics. *IJCAI-83.*

Larkin, J. (1981) Enriching formal knowledge: A model for learning to solve textbook physics problems. In John R. Anderson (Ed.), *Cognitive Skills and their Acquisition. Lawrence Erlbaum Associates.*

Lenat, D (1983) The Nature of Heuristics (II and III) *Artificial Intelligence* Vol. 21, March.

Mahadevan, Sridhar. (1984) Verification-based learning of implementation rules for VLSI design. Tech. Rep. (forthcoming) Rutgers AI/VLSI Project.

Mitchell, T., Steinberg, L., Reid, G., Schooley, P., Jacobs, H., & V. Kelly. (1981) Representations for reasoning about digital circuits. *IJCAI-81.*

Mitchell, T. (1983) Learning and problem solving. *IJCAI-83.*

Mitchell, T., Steinberg, L., & S. Amarel. (1984) A learning apprentice for knowledge acquisition in an expert system. Rutgers Digital Design Project Working Paper No. 16.

Mitchell, T., Steinberg, L., Kedar-Cabelli, S., Kelly, V., Shulman, J., & Weinrich, T. (1983) An intelligent aid for circuit redesign. *AAAI-83.*

Mitchell, T., Steinberg, L., & Shulman, J. (1984) VEXED: a knowledge-based VLSI design consultant. Rutgers AI/VLSI Project Working Paper No. 17.

Mostow, D. J. (1981) Mechanical Transformation of Task Heuristics into Operational Procedures Ph.D. thesis, Computer Science Department, Carnegie-Mellon University.

Mostow, J. (1983) Program transformations for VLSI. *IJCAI-83.*

Mostow, J (1983) A problem solver for making advice operational. AAAI-83.

Mostow, J. & Balzer, B. (1984) Application of a transformational software development methodology to VLSI design. *Journal of Systems and Software*: 4.

Mostow, J. (1984) A decision-based framework for comparing hardware compilers. *Journal of Systems and Software*: 4.

Mostow, J. (1984) Rutgers Workshop on Knowledge-Based Design. *SIGART Newsletter* (to appear).

Mostow, J. Memoize: a transformation for speeding up real Interlisp programs (including itself!). In preparation.

Neches, R., Balzer, R , Dyer, D., Goldman, N., & M. Morgenstern. (1983) Information management: a specification-oriented, rule-based approach to friendly computing environments. *Proc. IEEE Conference on Systems, Man, and Cybernetics.*

Novak, G. (1976) Computer Understanding of Physics Problems Stated in Natural Language. *American Journal of Computational Linguistics* Microfiche 53.

Novak, G. (1977) Representations of Knowledge in a Program for Solving Physics Problems. *IJCAI-5.*

Novak, G. & A. Araya. (1980) Research on Expert Problem Solving in Physics, *Proc. First Annual National Conference on Artificial Intelligence.*

Rich, C., Shrobe, H.E., & Waters, R. C. (1979) Overview of the programmer's apprentice. *IJCAI.*

Roach, J (1984) The rectangle placement language. *Proc. 21st Design Automation Conference, IEEE.*

Scherlis, W. & D. Scott. (1983) First steps towards inferential programming. Invited paper, IFIP Congress 83, North-Holland.

Singh, N. (1983) MARS: A Multiple Abstraction Rule-Based Simulator. Stanford Heuristic Programming Project, HPP-83-43.

Smith, Douglas R (1983) A problem reduction approach to program synthesis *IJCAI*

Smith, Douglas R. The design of divide and conquer algorithms. To appear in *Science of Computer Programming.*

Smith, Douglas R (1982) Top-down synthesis of simple divide and conquer algorithms. Tech. Rep. NPS52-82-11, Department of Computer Science, Naval Postgraduate School, Monterey, CA. Revised version submitted for publication and available from the author.

Steinberg, L. & T. Mitchell. (1984) A knowledge based approach to VLSI CAD: The redesign system. *Proc. 21st Design Automation Conference, IEEE.* Also Rutgers AI/VLSI Project Working Paper No. 11.

Subrahmanyam, P.A (1982) Automatable Paradigms for Software-Hardware Design: Language Issues. In J.Rader (Ed.), *IEEE Workshop on VLSI and Software Engineering.*

Subrahmanyam, P.A. (1983) Synthesizing VLSI Circuits from Behavioral Specifications: A Very High Level Silicon Compiler and its Theoretical Basis In F. Anceau (Ed.,) *VLSI 83.* North Holland.

Subrahmanyam, P.A., & S. Purushothaman. (1983) An Algebraic Basis for Specifying and Reasoning about Protocols For Designing Self Timed Circuits. In F. Anceau (Ed.,) *VLSI 83.* North Holland.

Subrahmanyam, P. A. (1983) Overview of a Conceptual and Formal Basis for An Automatable High Level Design Paradigm for Integrated Systems. *1983 IEEE International Conference for Computer Design and VLSI.*

Subrahmanyam, P.A. & S.Rajopadhye (1983) Formal Semantics for a Symbolic IC Design Technique. *1983 International Conference on Computer Design and VLSI.*

Subrahmanyam, P.A., Scachi, W., & R. Katz. (1984) Design Environments for VLSI and Software. To appear in *Journal of Systems and Software.*

Sussman, G. J. (1973) A Computational Model of Skill Acquisition. Ph.D. thesis, MIT Math Department. Available as MIT AI Lab. Tech. Rep. 297, August 1973, and as *A Computer Model of Skill Acquisition* New York: American Elsevier Publishing Company.

Swartout, W.R. (1981) Explaining and justifying expert consulting programs. *IJCAI*

Swartout, W.R. (1983) XPLAIN: A system for creating and explaining expert consulting systems. *Artificial Intelligence* Vol. 21, No. 3, Sept. 83. Available as ISI/RS-83-4, USC-ISI, 4676 Admiralty Way, Marina del Rey, CA. 90292.

Swartout, B. (1983) Explainable expert systems. *Proc. IEEE Conference, MEDCOMP '83.*

Tong, C. (1984) Knowledge-based circuit design. Ph.D dissertation, Department of Computer Science. Stanford University.

Wile, David S. (1983) Program developments: Formal explanations of implementations. CACM 26:11.

Announcement of Publication

Handbook of Statistics
Volumes 8 and 9

The series *Handbook of Statistics* has been started to disseminate information on a very broad spectrum of topics in theoretical and applied statistics. Each volume is devoted to a specific topic and consists of chapters written by prominent workers in that area.

Volume 8 is devoted to Statistical Methods in Biological and Medical Sciences; Volume 9 is devoted to Computational Statistics.

The editor of both volumes is P. R. Krishnaiah. The members of the editorial board for Volume 8 are S. Karlin and C. R. Rao. R. Gnanadesikan and E. J. Wegman are the members of the editorial board for Volume 9.

Suggestions for these volumes should be sent to:

P. R. Krishnaiah, General Editor
Handbook of Statistics
Center for Multivariate Analysis
Fifth Floor, Thackeray Hall
University of Pittsburgh
Pittsburgh, PA 15260 U.S.A.
Phone (412) 624-5814

CSI LISP
from Cognitive Systems

CSI LISP is a new implementation of **T**, which is descended from SCHEME. It was designed without compromise to be a powerful, expressive, and portable language while maintaining an integrated, balanced, and clean design. Built to run on top of Common LISP and LISP-VM, **CSI LISP** provides portability between computers such as VAX, IBM, and Apollo.

LANGUAGE FEATURES
Lexical scoping
Object-oriented programming
Procedures and lexical closures as first-class objects
Portability between computers supporting Common LISP and IBM computers

PACKAGE INCLUDES
Interpreting
Run-time Library
Comprehensive User Guide
Debugging Tools
Language Extensions
Compiling
Source Code (optional)
On-Line Help
Performance Measurement
Maintenance (optional)

CSI LISP is easy to learn, easy to read, and easy to write, because the language includes a compact but comprehensive set of powerful and flexible functions Almost as important for ease of programming is what the language does **not** include; the language is not cluttered with non-mnemonic names or groups of functions having nearly identical behavior.

CSI LISP and **T** are in use today at commercial and academic installations throughout the world. Applications include natural language interfaces, expert systems, and advisory systems in financial, industrial, military, and commercial domains.

For more information, contact:
Ann Drinan
Cognitive Systems
234 Church Street
New Haven, CT 06510
(203) 773-0726