Understanding and Automating Algorithm Design

ELAINE KANT

(Invited Paper)

Abstract—**Algorithm design is a challenging intellectual activity that provides a rich source of observation and a test domain for a theory of problem-solving behavior. This paper describes a theory of the algorithm design process based on observations of human design and also outlines a framework for automatic design. The adaptation of the theory of human design to a framework for automation in the DESIGNER system helps us understand human design better, and the implementation process helps validate the framework. Issues discussed in this paper include the problem spaces used for design, the loci of knowledge and problem-solving power, and the relationship to other methods of algorithm design and to automatic programming as a whole.**

Index Terms—**Automatic programming, automating algorithm design, human problem solving, program synthesis, protocol analysis.**

I. The Algorithm Design Task

A. Design as an Intellectual Activity

Algorithm design is the process of coming up with a sketch, in a very high level language, of a computationally feasible technique for accomplishing a specified behavior. The design process combines cleverness in problem solving, knowledge of specific algorithm design principles, and knowledge of the subject matter of the algorithm (e.g., geometry, graph theory, physics). When people design algorithms, their design repertoire includes discovery and visual reasoning in addition to the (ideally) disciplined application of problem-solving techniques.

Human design is a rich source of ideas for a framework for automatic algorithm design. Observing the human design process and attempting to capture the basic ideas in an automated system both helps us understand how people structure and use their knowledge about design and also validates our observations and framework. The DESIGNER project included such a study of human design and an initial version of an automated system [16]–[18], [27]. The goal of the project is to create an automatic design system that can apply existing design principles as well as exhibit some creativity. The observations of human design are to be incorporated, but the automatic system should take the strengths and weaknesses of both computers and people into account. We are not trying to model human problem-solving behavior as an end in itself.

The next section of this paper (I-B) presents a sample algorithm design problem (finding the convex hull) and identifies some questions to be addressed for a theory of the design process. Section II summarizes our observations of human designers working on the convex hull and other algorithms. Section III then takes a different cut at the observations of design. Rather than concentrating on the theory of human design, it lays out a framework for automated design and discusses where the problem-solving power in the framework lies. Several sources of power are identified: the ability to search in multiple spaces (relying on knowledge from the domain as well as knowledge about algorithm design), the ability to recognize anticipated and unanticipated but useful results, knowledge about the time performance of different operations and algorithms (efficiency knowledge), and the ability to execute partial algorithm descriptions on examples. In Section IV, the status of the DESIGNER implementation is summarized. Finally, this framework for design is compared in Section V with other approaches to automating algorithm design and with automatic programming as a whole.

Manuscript received June 3, 1985; revised June 27, 1985. This work was performed while the author was at Carnegie-Mellon University and was supported in part by the Defense Advanced Research Projects Agency under Contract F336 15-81-K-1539 and in part by the National Science Foundation under Grant DCR-8412139.

The author is with Schlumberger-Doll Research, Ridgefield, CT 06877, on leave from Carnegie-Mellon University, Pittsburgh, PA 15213.

B. A Challenge: The Convex Hull Problem

Consider the problem of finding a convex hull, which has applications, for example, in algorithms for vision and graphics. The input to the problem is a set of points (in a plane). The desired output is the smallest-area convex polygon that encloses the input points (its vertices are a subset of the input points).

If you were shown some points on a blackboard or piece of paper, you would probably have no trouble sketching their convex hull. (And in fact a picture such as the one in Fig. 1 helps many people to understand the problem.) Suppose instead you, or an automatic design system, were asked to create an algorithm suitable for (later) encoding as a computer program in a conventional high-level language. The goal is to sketch out an algorithm in the terms you would use for describing it to a colleague or to a programmer, without worrying about the low-level implementation details.

To understand the questions that must be addressed by a theory of algorithm design, you might observe your problem-solving behavior as you work on this problem. For example, how do you make sure you understand the problem? Does a picture help? Do you write down any formal problem descriptions? If so, in what language? Do you create a variety of examples or counter-examples?

What process do you use in coming up with an algorithm? Do you draw analogies to other algorithms? Draw on general knowledge of algorithm design principles? How do you decide when your algorithm is complete and at a sufficient level of detail?

Once you have an algorithm, how do you convince yourself that it is correct? Did you design it by applying correctness-preserving transformations? Did you use geometric or other mathematical theorems? Find proofs for conjectures? Will you test your algorithm on sample data? Explain the algorithm to yourself or a friend in words? Write pseudocode?

Some related questions concern the quality of the algorithm. How do you decide when it is good enough? What does it mean to be a good algorithm? Do you know what the run-time or space performance of your algorithm will be? Did you worry about what the distribution of data would be in creating the algorithm? In determining performance?

The next section describes how some human designers solved the convex hull problem, with observations that attempt to answer the types of questions asked here. The goal of the section following that one is to incorporate the observations of human design into a framework for artificial intelligence programs that could perform the same feat. Since the design of complex algorithms is currently best accomplished by human beings, observing their performance would appear to be a profitable starting point for automating the design process. However, since the talents of computers are not those of people, it is reasonable to search for a different method if the goal is total automation of design or a novel mixture of human and machine design. This issue is discussed in Section V.

II. Methods for Designing Algorithms

The theory of human design presented here is based on the analysis of a set of protocols from approximately 15 sessions with computer science faculty, graduate students, and undergraduates. (A methodology for protocol analysis is described elsewhere [8], [21].) Our designers were independently given the task of creating algorithms to find convex hulls, closest pairs of points, and intersecting line segments. Several protocols have been analyzed in great detail while the others have been gone over more lightly and used primarily as confirming evidence.

Before summarizing the features of human design, some caveats on the general applicability of the observations are in order. 1) We observed the design of individual algorithms whose complexity is due to a requirement for cleverness rather than to the information processing overload of combining an overwhelming number of small but straightforward parts. 2) The algorithms depend on applying an appropriate set of operations rather than on designing a specialized data structure. 3) Our study did not include any interaction between people and design aids other than pencil and paper or blackboard. (However, no one volunteered any feelings that a calculator, computer, or any other automated device would have been of any help in designing their algorithm.) 4) The design sessions we observed were on the scale of hours rather than the months spent by research algorithm designers. Other processes than those we observed may take place in such long time periods.

Our observations may be at least partially valid in a wider context despite the caveats. Other researchers have studied the design process in software engineering and have made observations similar to ours [1], [13]. Also, there is anecdotal evidence that similar problem-solving techniques are used in the design of algorithms that are highly dependent on clever representations.

We observed several major classes of behavior in our designers. In addition to understanding these processes (behaviors), there is an issue of control—how the processes are ordered, including how problems are selected and the role of evaluation. After the different processes are discussed, the issue of control will be considered.

The processes that we observed our designers draw upon include the following.

1) *Understand* the problem.

2) *Plan* a solution around a kernel idea and *refine* or elaborate the kernel stucture.

3) *Execute* the partially specified algorithm.

4) *Notice* and formulate any difficulties or opportunities.

5) *Verify* that the structure is a solution (i.e., meets its specifications).

6) *Evaluate* the solution (e.g., for efficiency).

The explanations of these processes draw on all of our observations and those of our colleagues who have studied the design process in software engineering. However, illustrations are taken primarily from the stories of two particular designers from our study, D1 and D2, who tried to solve the convex hull problem. Each part of a story is prefaced by an abbreviated form of the designer's name and a sequence number for future reference. For example, the first step of Designer 1's story is labeled [D1.1].

A. Understand the Problem

In classical discussions of problem solving [23], one important problem-solving process is understanding the problem (perhaps by listing properties of the objects in question) and considering reformulations of the problem. Some of our designers (but not D1 or D2) did draw a picture of a convex hull (or whatever) early on, which may have led to some unverbalized observation of or reasoning about properties of convex hulls and seemed to have convinced them that they understood the problem.

[D6.1] D6 drew the picture shown in Fig. 1.

[D3.1] D3 wondered whether using polar coordinates might not be a useful way to think about the problem.

B. Plan and Refine the Solution

Assuming that a problem specification has been understood, design begins with a *kernel idea* or solution plan, quickly selected from those known to the designer. Depending on the designer's background, the idea may vary in sophistication from generate and test to more complex strategies such as divide and conquer or dynamic programming. The designer lays out the basic steps of the chosen idea and follows through with it unless the approach proves completely infeasible.

[D1.1] D1 had the initial idea that the algorithm should be one that generated all points in the input in some arbitrary order and tested each to determine whether it was on the hull. This had the potential of running in linear time (proportional to the number of input points).

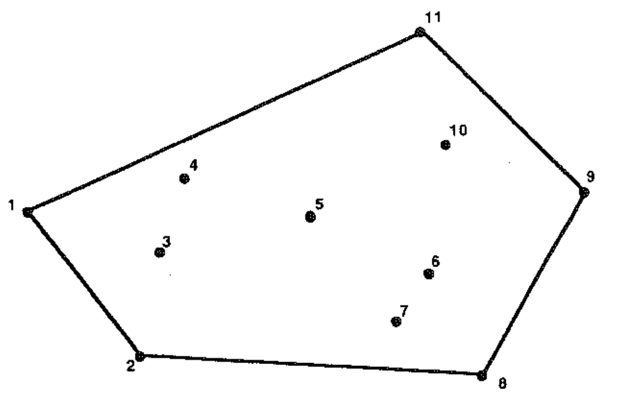

Fig. 1. Initial example drawn by D6.

[D2.1] D2 decided to try a divide and conquer algorithm (the special form of divide and conquer in which the inputs are divided into subsets, the algorithm is recursively applied to each, and the results are merged back together).

At this early stage it is not clear which kernel idea will lead to a better algorithm.

After formulating a plan, the designer *refines* the basic steps of the kernel idea. By and large, this elaboration proceeds by *stepwise refinement*. The designer may lay down the major components, effectively decomposing the problem into subparts, or may add new inputs or assertions about details of the structure. The refinement steps 1) may be suggested by *knowledge* appropriate to the problem and task domain or 2) may be a natural result of attempting to *execute* an algorithm.

[D2.2] An example of the application of appropriate knowledge about algorithm design principles is D2's expansion of the notion of using a divide and conquer algorithm into a sequence of specific steps: divide the input point sets into subproblems, find the convex hulls recursively, merge the subsolutions back into a convex hull.

[D2.3] Furthermore, D2 recognized from previous experience with geometric algorithms that a likely possibility for the divide step of the divide-and-conquer algorithm was to sort the points along one of the coordinates and use the median as a dividing line.

In the absence of the knowledge that suggests the proper refinements, the designers *search* by trial and error: they hypothesize algorithm steps and try them out by executing the partially specified algorithm.

[D1.2] D1 had no idea how to test whether a point was on the hull and decided to try the proposed algorithm on a specific figure to find the test.

[D1.3] D1 then drew the picture shown in Fig. 2.

The refinement process is hardly one of pure top-down design.

[D1.4] Such a point-on-hull test did not reveal itself, but another related test did, and D1 proceeded to modify the hypothesized algorithm to exploit the new test.

D1 was able to proceed with the new test because it was closely related to the old. The new test, described in Section II-D [D1.7], was a test for whether line segments rather than points were on the hull. Since polygons can be specified by sequences of either points or segments,

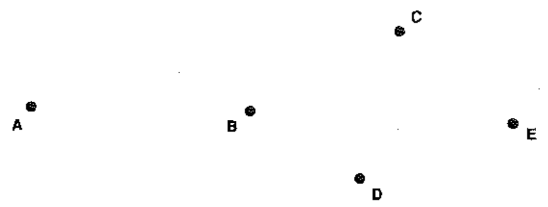

Fig. 2. Initial example drawn by D1.

changing the test might seem to be only a matter of changing the view of polygons used in the problem.

Most design falls somewhere between having the correct knowledge and searching. At some steps the designer knows what to do and knows what the implications of the refinement step are: other times, search is required.

[D2.4] D2 did not find the merge step as obvious as the divide step. Most people do not.

C. Execute and Analyze the Algorithm

Trial execution of algorithms is often used as a technique for making inferences about the algorithm developed so far. We observed two extemes of execution of partially specified algorithms—one on concrete data (which we call *test-case execution*) and the other on symbolic exemplars (which we call *symbolic execution*). Both forms of trial execution help elaborate the algorithm description by exposing difficulties and opportunities. We found it useful to view execution as a technique for selectively propagating constraints (which we call *assertions*) by moving them around in the order in which steps of the algorithm are executed. This limits the reasoning that might otherwise be necessary to find contradictions and make inferences. A more detailed description of some different types of execution and static analysis, collectively referred to as *developmental evaluation*, is available elsewhere [27].

D. Notice Difficulties and Opportunities

Designers notice problems both in their algorithms as described abstractly and in pictures that they draw to help them design. While executing the proposed algorithm, difficulties (missing steps, inconsistencies between parts of the algorithm) may arise, leading the designer to further refinements. Thus, we say that the designer's refinement process is *difficulty driven*.

[D1.5] In D1's algorithm, a difficulty arose when the test involving line segments was combined with the generator of points and D1 had to modify the algorithm to accommodate this.

Here one assertion propagated by the execution process (that a point is produced by generating over the input set) contradicts another assertion (that a line segment should be the input to the test rather than the point it is handed).

[D1.6] D1 eventually changed the kernel idea from generate and test to a greedy algorithm that attempted to generate the hull points in the order they occurred on the hull polygon, using backup to handle guessing failures. This could also be thought of as generating the hull segments in order.

Algorithm execution also can expose opportunities for improvement or modification of an algorithm.

[D2.5] After working on a sample problem, D2 realized that the merge step would be easier if the two subsolutions shared a common point and went back and modified the divide step to ensure that that would happen.

Most people draw *example figures* during algorithm design. The examples are used initially for understanding the problem, and later for reasoning about the task domain (using visual reasoning in at least the geometric domain) and in trying out the partially developed algorithms in test-case execution. Often, the designers notice things about the sample figures that they were not looking for. When what the designers notice turns out to be useful in developing their algorithm, we say that they have made a *discovery*.

[D1.7] In looking at Fig. 3, D1 realized that if a line segment had points on both sides of it, that segment could not be on the convex hull. D1 actually was executing an algorithm with a test for points being on the hull or not; the line segment in the figure was recording the fact that the points A and B had been generated so far.

[D2.6] D2 created Fig. 4 in attempting to find a merge step by considering all segments that could connect vertices of the two hulls and testing which were in the merged hull. D2 knew that this brute force search would be too expensive, but had no other ideas. The picture reminded D2 of another unrelated algorithm (for finding a minimal cost tour) in which a shorter path replaced two adjacent segments. D2 then applied a similar idea to the merge step, replacing segments a-d and d-e by segment a-e (D2's picture was not actually labeled). The generalization D2 made was that a concave angle in the merged hulls was to be replaced by a segment connecting the two end points of the angle.

The discoveries are described in more detail in a previous paper [17].

Some other observations the designers made would have allowed only small optimizations.

[D1.8] D1 noticed that points are always on the same side of the (directed) line segments of the hull.

While discovery is not a voluntary process that can be planned as a design step, it does arise from the process of making observations of more than the immediate symbolic representation of the current algorithm description. The discoveries in our study all occurred when the designer was looking at a sample figure created for one reason and then recognized a geometric property, or key step from another algorithm, that would solve an outstanding goal. That goal was not the one the designer was currently worrying about (finding a test for a point being on the hull; finding a way to tell if a segment was on the merged hull), but it was usually not completely unrelated (finding a segment test rather than a point test; finding a different type of merge step). Thus, discovery could be characterized as serendipitously satisfied goals.

Both key observations in the problem domain and knowledge of design principles are usually necessary for clever design. Most algorithms published in papers or given as exam problems have at least one good observation or trick that is novel at the time of the design; otherwise we would probably say the algorithm is "obvious" or is

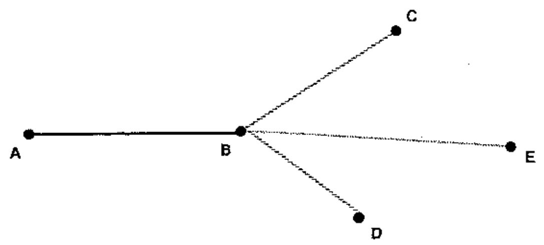

Fig. 3. D1's discovery of a segment-on-hull test.

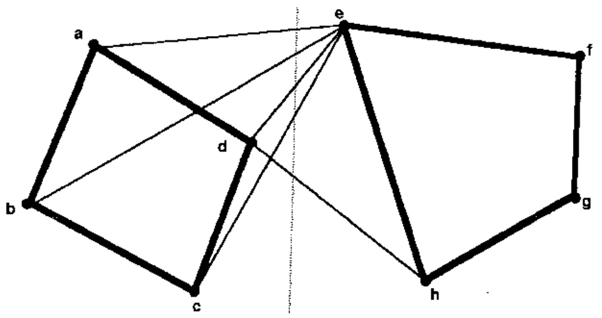

Fig. 4. D2's discovery of a merge operation.

"just" a brute-force algorithm. Each of the tricks must be stumbled upon as a discovery unless it is already known to the designer. Good tricks are eventually refined into principles, but everything is a trick the first time each designer encounters it.

Although there is an element of chance in the discoveries, there is no lack of readiness on the part of the designer. The designer can be prepared both with immediate goals to exploit the observations and with a good understanding of design principles to fit the discovery into an overall algorithm. The more experienced and disciplined the designers, the better prepared they are for the discovery. An "experienced" designer is one with knowledge not only about algorithm design but also about problem domains. Domain knowledge can be derived either from past attempts at the problem or from experience with similar algorithms and similar domains (or from different domains but with the ability to reformulate problems in terms of other domains).

E. Verify Correctness

Our designers determined whether their algorithms were correct primarily by testing them on specific examples and observing whether there were any difficulties. Developmental evaluation can in fact be made to do the job of formal verification (if termination is also checked). To do this, the algorithm is executed on symbolic objects and all assertions are propagated to determine whether the results of the algorithm (and its subparts) match the specifications. If a specification includes performance constraints, then verification must also include an evaluation (see Section II-F) to determine whether the solution is efficien enough (in time or space complexity) according to the expectations.

During the initial algorithm design, the designers ignored "details" such as base cases, initializations, boundary conditions, degenerate inputs, and unresolved notes to themselves, but they were more careful about this i they were attempting to determine if the algorithm was complete or correct.

[D1.9] When D1 was asked for an algorithm summary during a pause, the response was that it was not an algorithm yet because the case of the first point not being on the hull had not yet been tested.

The heuristic that many designers follow is to get an algorithm for the general case first, then worry later about modifying the algorithm to take the exceptions into account. Although some methodologies claim to eliminate the concern with special cases (for example, [12]) they require that the specification or invariant be precisely stated before design begins. This is often difficult to accomplish. For more complex algorithms, handling the exceptions can itself require a major problem-solving activity and may yield new insights into the problem or solution.

F. Evaluate Plans, Refinements, and Solutions

The descriptions of the processes used in design did not detail how plans, refinement steps, and overall solutions are evaluated. Evaluation can be based on specific knowledge about the algorithm design principles being applied or on an analysis of the cost of the algorithm and its subparts.

If the designer has the appropriate rules about the algorithm design principle(s) and the domain, then the refinement process can be smooth and top down. For instance, the appropriateness of the kernel ideas selected by the designers depends on the quality of their knowledge of algorithm design principles. One can really observe here what expert-systems researchers call domain-specific knowledge. Generate and test is usually the fall-back idea, which is sometimes very efficient (linear in the input size) and sometimes not. After an algorithm based on a kernel approach is sketched out, or after the approach seems to be failing, some designers go on to an alternative approach.

[D1.10] After completing the revised algorithm for generating segments and testing whether they were on the hull, D1 determined that the run time of the algorithm was proportional to the cube of the number of input points. Declaring that this algorithm was only a "first shot," D1 went on to consider a dynamic programming approach and eventually to try divide and conquer.

[D4.1] In another problem involving finding intersections of line segments, another designer, D4, noted that there was a straightforward approach having to do with considering all pairs of segments, which was N squared. However, D4 felt that there ought to be some way to use sorting in the solution to get an $N \log N$ algorithm.

When experts (people with a strong background in algorithms and in the subject matter of the problem) design, they consider a variety of alternative refinements, select the best (remembering the rest for possible later use), and apply it to advance the design with one more level of detail in the refinement process. What is "best" is based on efficiency in the cases of algorithm design we studied, but is based on ease of implementation or modification in other cases. In expert design, the breadth-first process tends to be followed for all aspects of the design at a given level,

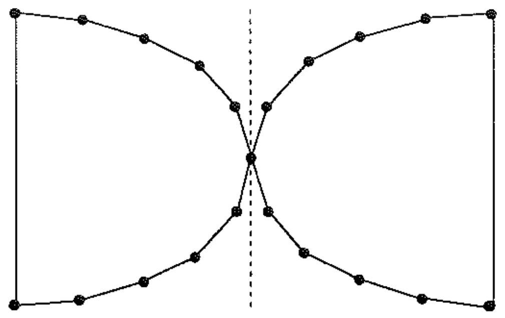
Fig. 5. A worst-case input for divide and conquer.

with interactions between the different parts of the design predicted and taken into account.

In contrast, if the designer's idea is naive (e.g., use sorting somehow), then the technique of executing hypothesized algorithm parts is more likely to be followed in a depth-first search from which the designer may never successfully return. (The idea may not have been wrong, but the designer may not have had the knowledge to carry it through.) Experts as well as novices are prone to a satisficing style of design when they are under pressure and do not have time for more exploratory design. Of course they are better at it since they have more experience, can make better predictions (e.g., about what kernel ideas will work or about interactions among different parts of the algorithm), and guess right more often.

Even when performance constraints are not explicitly specified, the designer often evaluates an algorithm or algorithm step's performance relative to other alternatives or to known or estimated lower bounds. Extreme cases of inputs may be tested to estimate worst case performance. Complexity analysis may be carried out in parallel with execution and verification by more experienced designers, or may be an explicit subtask of a conscious evaluation.

[D2.7] After discovering the way to merge by removing concave angles, D2 estimated the run time of the divide and conquer algorithm by arguing that even for the worst possible input, the merge time was linear in the number of points on the two subhulls and therefore the overall run time was acceptable.

[D5.1] Although D2 did not draw an additional figure to analyze the worst case, both D1 and another designer, D5, did. Even though their algorithms differed from that of D2 and from each other, they were both concerned with the same potential problem and drew similar pictures (see Fig. 5).

D2's final design was an $N \log N$ divide-and-conquer algorithm. It had a prepass step to sort all the points according to their X coordinates. The basic algorithm was to divide the input through the point closest to the median, recursively find the convex hulls of the two resulting point sets, and merge the solutions back together by eliminating concave angles (starting from the shared point). The base case is two- or three-point input sets, which can be made into convex hulls immediately.

G. Control Issues

The design processes described in the preceding sections do not always run to completion and do not take place in any fixed order. Evaluations within each step, as described in Section II-F, may cause the designer to terminate one approach and go on to another. The ordering of the design processes (including when they begin and end) seems to arise naturally out of the mechanisms of trial execution.

Selecting a problem to work on is a natural consequence of the problems exposed by trial execution. The character of the elaboration process appears to be an progressive deepening that takes each of the constructs in the algorithm a little further, sometimes backing up to higher levels to keep the overall picture in mind. However, the development of the different parts of the algorithm is not always even. If one aspect of the algorithm is a potential problem (i.e., other parts of the design depend on it and the outcome is uncertain), then it is more likely to be expanded to ensure that the algorithm as a whole is feasible. If it has an obvious solution or refinement and the implications of that decision seem well understood, at least at the current level of detail, it is not considered further. (Of course the assumptions may be wrong.) New components of the design are refined in the order they are executed, subject to the two previous considerations.

Verification and complexity analysis also seem to be achieved in part by propagating assertions during execution. Thus, other processes that contribute to control fit in nicely with this basic mechanism and can occur at the same time.

In short, design processes are applied as appropriate. Control is not a special source of intelligence. It comes out of responding to the data and out of the problems and opportunities arising during execution.

III. Locating the Problem-Solving Power

An important question to ask about any agent that exhibits intelligent behavior is where the knowledge and problem-solving power lie. Knowing the loci of intelligence gives us some clues for how to produce similar behavior automatically. Thus, we have attempted to formalize the problem-solving behavior we observed in our designers in terms of concepts that lend themselves to automation.

One common view of problem-solving behavior is that it is basically search in a *problem space*, with knowledge used to limit search. A problem space [21] is a set of *states* for describing problems and (partial) solutions. There are also *operators* that can be applied to a state to produce a new state. Starting from some initial state, the problem solver tries to discover a state that contains a solution by searching—by trying out different sequences of operator applications. In difficult problems, the problem solver may not have enough knowledge to proceed directly to a solution state and may try various paths, some of which lead to dead ends, and may return to earlier remembered or reconstructed states. Often, the problem solver does have *search control knowledge* that guides the selection of which operators to apply. In short, the problem solver gradually explores the space, acquires knowledge about it, and eventually may proceed down an appropriate path.

Given this view, we can ask more specifically where the problem-solving power in algorithm design is located. Sections III-A to III-E discuss several sources of power: the ability to search, the availability of multiple problem spaces, the ability to recognize relevant information, the ability to execute partially specified algorithms, and the use of knowledge about the efficiency of algorithms.

Some power lies in the knowledge carried by the problem spaces themselves—in what objects and operators they have available and in the heuristics they have for when and how to apply the operators. In one problem-space view, problem solving is a process of repeatedly changing a *context* by selecting a *goal* to achieve, a *problem space* to work in to attack that goal, a *state* within that space to work on, and an *operator* (and instantiations of its *arguments*) to transform the state [19]. Different types of knowledge can be associated with the selection process for each element of the context. Sections III-F to III-I illustrate the different types of knowledge available in several problems spaces related to design.

A. The Power of Search

In design, as in most tasks requiring intelligence, both search and knowledge are needed. Search is the backup for missing knowledge and can never be completely eliminated. It can take place at the very high level, such as searching for a kernel idea for an algorithm, or at the very low level, such as deciding how to instantiate an operator argument. Although at any level knowledge limits search when possible and gives clues about how to explore the problem spaces in a reasonable way, the ability to search is, in itself, a source of power.

In design, for example, search permits the creation of algorithms by trial and error in the absence of complete knowledge. Algorithm components can simply be hypothesized and then the algorithm as a whole tested to see if the specifications are satisfied. If only the objects and operators that formally specify and manipulate algorithm descriptions are available (i.e., there is no other model of the problem domain), the designing an algorithm requires the use of formal definitions of the concepts used in the problem specification and, recursively, of its subcomponents. However, more power than this is available to human designers and can be made available for automated design through the use of multiple problem spaces.

B. The Power of Multiple Problem Spaces

From our observations we conclude that each designer works in several different problem spaces during design (similar observations are described for other tasks [21]). The details of the problem spaces differ from designer to designer, but there is a remarkable consistency in the types of problem spaces used. The spaces used by our designers seem useful for a mechanical designer as well, so we include them in our framework for automatic design.

We observed our designers working in four spaces, two of which are extensions of another space. The two main spaces were 1) an *algorithm design space* that carries the knowledge of what is achievable in standard computer systems and of domain-independent algorithm design principles, and 2) an *application domain space*, such as one for geometric and visual reasoning. (The algorithm design space is also a domain space relative to design as a whole.) The two extension spaces have the same objects as the first two spaces plus additional objects and/or different sets of operators. They are described as separate spaces because the functions they perform are so different from the main spaces that they require a substantial number of new operators and because only the results of searches in those spaces, not the internal details, are available to the original spaces. 3) An *algorithm execution space* is an extension of the algorithm design space that has as new objects data items that carry information in the form of assertions about their execution history and has new operators that execute components in the design. 4) An *example generation space* is an augmentation of a task domain space in which figures are marked as standard examples, degenerate cases, counter-examples and the like, and in which there are new operators to produce the examples.

The necessity for different problem spaces is a result of the requirements of different types of knowledge. For example, what is possible or efficient in the domain (problem space) of algorithms for conventional digital computers is sometimes quite different from the way people reason visually or from what can be done with analog devices. (Consider solving the convex hull problem by pounding nails into a board to represent the input points and then stretching a rubber band over the nails and letting go.) The problem spaces that express such knowledge differ in the objects and operators included, the properties of objects or relationships between objects, and heuristics for how to control the applications of operators.

Having knowledge represented in a domain space as well as in an algorithm space gives the designer the power to create algorithms even in the absence of formal axioms about specification concepts such as being inside a polygon. The problem can be solved by generating constructs in the algorithm space and testing the proposed algorithms on examples to see if they work. This technique relies on the ability to generate examples to use as test cases. Example generation depends on knowledge of the domain space as well as knowledge of the goals in the algorithm space (say to determine whether a typical or degenerate example is desired). If a domain concept is not formally axiomatized, the designer cannot do any formal symbolic reasoning such as full verification or correctness-preserving derivation. However, by making some conjectures about the domain and validating them with test-case execution, the designer can reason formally about the rest of the algorithm.

[D2.8] Having knowledge from the domain space of what line segments were on the merged hull allowed D2 the hope of finding an operation that would test whether proposed segments were correct.

For each of the problem spaces relevant to design, we can ask what knowledge is available for recognizing when context elements should change: how does a system recognize when goals are satisfied or when new goals should be attempted, when the problem space should be changed to work on the different type of goals, what state to expand within a problem space, and what operator to apply and how to instantiate the operator. Examples of the different types of knowledge contained in problem spaces will be given in Sections III-F through III-I.

First, some aspects of problem-solving power that cut across problem spaces are discussed. This power can be cast as knowledge that allows the designer to avoid search.

C. *The Power of Recognition Knowledge*

The ability to recognize objects and to recognize the applicability of operators is a major source of power in problem solving. The search process is not driven by an algorithm that selects context elements in a fixed order but rather by recognition rules that observe when some context element should change: for example, when a goal has been satisfied or when an operator would help change the state in a desired way. The conditions for recognition can be symbols in the algorithm design space or visual images from the domain space. These clues can involve goals, points of view, or (if the recognition rule was learned) other objects in the problem-solving context whose inclusion as a clue was only accidental to the formation of the recognition rule. A very large number of recognition rules may be present. However, the conditions that are monitored must be computationally simple, involving only straightforward matching.

An example of the role of recognition is its use in discovery, a key process in algorithm design. Discovery depends on generating examples to work with and then noticing properties about them or reasoning about them. The recognition processes usually take place in the domain space, but what is noticed depends on the goals of the problem solving (and the content of the recognition knowledge).

Recognition is also important in example generation, which is constrained by the goals of the problem solving (is it to be an average case, degenerate case, initial or base value, counter-example, used by efficiency analysis, etc.) but depends on knowledge of the domain and recognition of successful construction of the example in terms of domain properties.

[D1.11] D1 first generated points *A*, *C*, *D*, and *E* in Fig. 2 as an initial test-case example, then noticed that the example was degenerate, since all points were on the hull, and added a fifth point (*B*) in the center to remedy the difficulty. The points were not labeled at that time.

Nonsymbolic recognition and processing (such as visual reasoning) is clearly important in designing computational geometry algorithms, but it is really important in all domains, such as that of algebraic problem solving? At least for some people, it is. Built-in visual operators are better at some types of processing and provide another perspective on a problem. They may suggest approximations or fortuitously counterpose objects that would not be related by a general symbolic reasoning process. Unfortunately, the process of visual reasoning is not well understood at this time.

D. *The Power of Execution Knowledge*

Trial execution in algorithm design serves the purposes of controlling the order of the refinement process (see Section II-C) and limiting the inferences made as well as being a vehicle for the more usual functions of debugging and verification (see Section II-E and [6]).

The nature of creative algorithm design requires some mechanism for inference, whether it is a full theorem prover, small set of simplification rules, or something in between. Making all possible inferences during algorithm design would be very expensive computationally. Execution is a way to focus attention on certain assertions in the algorithm description space and on certain parts of pictures in the domain space so that inference and recognition only have to take place over a smaller set. The execution techniques limit the inferences and constraint propagations to those most likely to be useful for the current stage of the design. Avoiding the extensive search of theorem proving or uncontrolled inferencing through execution is a form of knowledge about design. This topic is discussed more thoroughly in other papers [7], [27].

E. The Power of Efficiency Knowledge

Efficiency knowledge serves as an evaluation function throughout the algorithm design process, not just as an evaluation of complete designs. Information about potential run time or space use serves as a rough guideline in the selection of a kernel idea (illustrated in [D1.1][1]), in the evaluation of refinements (D2 knew that the merge step had to be linear to get the desired overall performance [D2.6, D2.7]), and after an algorithm sketch is complete (D1 decided that cubic performance was probably not the best possible [D1.10]).

Efficiency knowledge can take many forms, including assertions about the run time of specific operations or algorithms, assertions about the intrinsic complexity of problems, rules for how to analyze algorithms and combine the run times of the individual operations, and rules for setting constraints on what performance must be reached on a subpart of an algorithm to guarantee overall performance. Efficiency knowledge is generally contained in an algorithm design space, but some of it is specific to particular application domains. Also, some information about operation costs, and in fact, some heuristics about what design principles or implementation choices are best, are built around assumptions about a cost model for the target architecture.

F. The Algorithm Design Space

In algorithm design, it is sometimes difficult to come up with *any* reasonably effective solution,[2] although some problems have simple brute force solutions. (Consider the problem of finding the closest pair of points in a point set. You can probably see a simple algorithm for solving problem immediately.) Since algorithm design involves searching in a space not dense in solutions, dead ends are a serious problem, and knowledge of what design principles and domain facts are relevant is almost a necessity (as is the ability to reason and recognize in other spaces). Such knowledge can help decompose the problem or select and instantiate operators in the problem space.

Designers have variants of the algorithm design space that depend on their assumptions about the target architecture as well as on their overall knowledge of design principles. If the algorithms were to be programmed on an architecture with pipelined or distributed processing or associative retrieval, the representations for algorithms and heuristics for how to design might be greatly different. Some designers make (at least implicit) assumptions about the target architecture from the beginning of an algorithm design, but it is preferable to stay independent of the target as long as possible.

The knowledge in the algorithm description space includes facts about mathematics, logic, arithmetic, and algorithm design principles. The knowledge can be in the form of both object descriptions and operators on those objects. Other knowledge can be represented by rules about when to change the problem-solving context.

1) Objects and Operators: The basic objects for describing algorithms in the algorithm design space are *components* that specify fundamental types of processing. These components may test whether a property holds, generate the elements of a set one at a time, achieve an input/output relationship, apply a domain operator, select a subpart of a compound object, or modify a memory of objects.

The algorithm components are connected by *links* that allow flows of data and/or control. Components may be augmented with *assertions* about their properties or about their relationship to other objects or operators in any of the problem spaces. These assertions specify exactly what property to test, what set to generate over, what operator to apply, and so on. For example, a selection criterion might be to pick the bottom, left point from a set of points. New components can be defined in terms of old ones by adding additional standard inputs or outputs or by adding assertions, or a component can be defined as configuration of other components.

The assertions associated with components may also include information about the types of data objects expected as inputs or outputs or other preconditions or postconditions of processing, the ordering constraints on a generator, the initialization of a memory, expectations or conclusions about the time complexity of the algorithm (component), constraints on the order of execution of the algorithm components, and notes about the algorithm (such as it has not yet been tested for the initial point lying inside the hull).

Since algorithms usually manipulate some sort of data, there are also representations for the common mathematical concepts such as numbers or symbols and of sequences or sets of other objects. Assertions about these objects can be attached to descriptions of the object type or to *items* that represent specific data.

The number of combinations of pairs from a set of elements is proportional to N squared. [D4.1]

Divide and conquer algorithms can often have run time of N log N. [D2.7, D1.11]

The operators in the algorithm description space are

[1]In the remainder of this paper, labels following descriptions of bits of knowledge refer back to parts of the design story where they are used.

[2]See Section V-D or [2] for a comparison to the search problem in program synthesis.

simple (syntactic) editing operations that add or modify components, links between components, and assertions. The knowledge is all located in the rules that suggest instantiations of the type of components to create, the specific components to link, and the details of the assertions to be added.

2) Operator Selection: Selecting an operator (and instantiating it by selecting values for its arguments) can be made more effective through the use of knowledge about general algorithm design principles and about algorithms in a particular domain of application. This knowledge will be expressed here as rules. Other such knowledge, for example how to handle specific problems raised during execution (the equivalent a difference table for means-ends analysis), also limits the amount of search necessary for operator selection.

The following set of rules about operator selection and instantiation is merely a representative sample of the knowledge that an algorithm designer (human or otherwise) might have (not every designer has the same knowledge, of course). Many other rules also would add their suggestions and vetoes about what to do. If there is no consensus about what operator to apply, the fall back is search through the suggested possibilities.

If a component needs to be refined and its output is a subset of its input, refine the component to an element-by-element generate-and-test algorithm. [D1.1]

If a component needs to be refined and its output is a structure that must satisfy certain constraints, refine it to an algorithm that builds a minimal structure and then adds units of structure until the constraints are satisfied. [D1.6] (An instance of this rule is suggested in [3].)

If an algorithm looks at part of the input many times to do the same kinds of tests, try saving information rather than recomputing, say with dynamic programming. [D1.10]

If the characteristics of subproblems produced by the divide step of a divide-and-conquer algorithm are unknown, than add the assertion that they are two equal-sized subproblems.

If the characteristics of subproblems produced by the divide step of a divide-and-conquer algorithm are unknown, and if the set being divided is a set of points in two dimensions, then refine the divide step to split the points into the two sets defined by a line through the median of the points sorted on one axis. [D2.3] This rule has a bit of domain-specific knowledge although it is in the algorithm space.

If a component is missing a link to a required input, look for a component that has an output with the same type (or having that type as a subpart or superpart) and connect the two components. This rule would be used only if there were no suggestions for more specific inputs.

3) Changing State: The state in a problem space context changes primarily as a direct result of the successful application of an operator that modifies the algorithm description. If the operator application fails, and if there were competing suggestions about what operator to apply, then alternative operators still apply and another will be tried. In addition to either failing or succeeding, an operator may return a difficulty or opportunity. This becomes another goal to be worked on, perhaps in a different problem space. After processing of the new goal is complete (which may change the state of objects visible from the algorithm description problem space), the rules that caused the original operator to be selected may or may not be retriggered. If they are, the operator application can be retried.

4) Changing Problem Spaces: One of the benefits of having multiple problem spaces is the ability to reduce search by working toward the same goal in a different space. Some examples of rules that can cause space changes are as follows.

If a component needs to be refined, and its output is a construct in space X, create examples of it and notice their properties. If this rule is applied, it will cause a transfer first to the example generation space and then to the domain space *X*. [D1.2]

If a configuration of components has not been shown to achieve the specifications of the component of which it is a refinement, then symbolically execute it. [D1.9]

If a configuration of components has not been shown to achieve the specifications of the component of which it is a refinement, and if symbolic execution has already been tried or is known in advance to be too complex to be informative, then execute the configuration on a concrete example. [D1.2]

5) Goal Satisfaction and Creation: Recognition of when goals have been achieved or nearly achieved, of when to give up on a goal and declare failure, of when to create new goals, and so on, is crucial to enabling discoveries. Strict enforcement of hierarchical subgoaling would not allow the same flexibility and creativity in goal satisfaction and creation as recognition allows. This sort of goal change knowledge can also serve as design heuristics. For example, some rules that express this knowledge are as follows.

If an exponential algorithm is created, try to improve it or find an alternative unless it can be shown that the problem is itself exponential. [D2.6]

If all objects added to a set have a common assertion, hypothesize that that property holds for all elements in the set and try to substantiate the hypothesis.

If a component is defined by assertions that are appropriate for the level of detail currently desired (however that is determined!), then consider the component acceptable.

If a component is not considered to be refined to an acceptable level of detail, then create a goal to refine it.

G. The Application Domain Space

Algorithm designers need knowledge about their task domain as well as about algorithm design in general. As an example of a problem space describing a task domain, consider the knowledge about geometry that can be used in solving the convex hull problem.

Objects that are manipulable in the geometric domain include points, lines, segments, angles, and polygons. Special properties of object types or of specific objects may also be recorded. For example, the degenerate case of the object type polygon could be a point or line segment, and a triangle would be the boundary case. For a specific geometric object, properties would include being convex or being above or below a line.

The operators in the geometric domain include accomplishing such functions as drawing a line segment between two points and recognizing that a polygon is convex.

Any symbolic descriptions of the objects in a figure and assertions about the objects or their relationships are available to the other spaces. For example, in the algorithm space assertions may serve as test predicates, comparison or ordering relationships, or criteria for extraction from a compound object. Operators are available by reference for execution, say to build a polygon in the example generation space or associated with a component in the algorithm space and run during test-case execution, but their internal workings are not available.

The domain space also includes recognition knowledge, expressed here in the form of rules. If these rules are applied to a figure in the current focus of attention, they may cause recognition and/or the construction of a new object just as an operator application might. For example,

If two line segments share a common end point, perceive the figure defined by that pair of segments as an angle. [D2.6]

H. The Execution Space

The problem space in which execution occurs is an augmentation of the algorithm description space. The execution space is responsible for avoiding the blind search that might result from arbitrary suggestions for operator instantiations. Because the partial algorithm descriptions are checked frequently by execution, meaningless guesses are not allowed to propagate and waste search time.

In the execution space, the object type item represents the data processed by the algorithm that flow over the links between components. The items can represent either specific objects from the domain space (point A) or symbolic objects ("a point"). Items can be augmented by properties that are known to be true of them at a given point in the algorithm execution history. Some example properties are that a point has been determined to be on or off the hull, or that point is the one most recently added to a memory.

The operators in this space control the sequencing of component execution and carry out component execution. If assertions needed to carry out the operators are missing, a difficulty is returned and a new goal to handle the difficulty is created.

Some instances of rules that suggest new goals to work on are as follows.

If the input for test-case execution is uninstantiated, set up a goal to get an example input. This will cause a transfer to the example generation space. A particular point set would be an example input for the convex hull problem. [D1.3]

If test-case execution shows that applying some operation will make progress toward a solution of the problem but not solve it completely, try modifying the description in the algorithm design space to apply the operation repeatedly, perhaps inside a loop.

I. The Example Generation Space

The example generation space is also an augmentation of another space, the domain space. Objects must be augmented by properties that describe their typical instances, degenerate instances, boundary cases, and so on, if such information is not already present in the domain space. For instance, sequences consisting of repeated copies of the same element are not typical instances. Some sample operators are those that add and remove elements from examples. Some sample rules are as follows.

When creating an input set for a generate-and-test algorithm, if all elements currently in the set satisfy the test, then add another element. [D1.10]

When creating an example for test-case execution of an algorithm that has not yet been checked for correctness, pick nondegenerate objects and constructors.

IV. The DESIGNER Implementation

To test the ideas in the design framework described here, a preliminary version of the DESIGNER system has been implemented. This system covered the representation and execution of algorithms (the algorithm design and execution problem spaces), a simple version of the geometric domain problem space, and some simple rules for generating examples and changing problem spaces. The system included some simple knowledge about generate and test and divide conquer, but did not have a sophisticated understanding of algorithm design principles. Some simple knowledge about estimating algorithm efficiency was also represented. Although DESIGNER is capable of approximating some pieces of the designs of the convex hull algorithms described in this paper, it is far from being an independent automatic designer. Much was not attempted in the initial system. Our ideas on visual reasoning and complex recognition are still in flux and have not yet been implemented. The handling of large databases of algorithms and principles, reasoning by analogy, and learning have not been addressed.

The preliminary system was implemented in a combination of MACLISP, a simple (locally designed) frame system with several forms of inheritance, and OPS5 [10]. The frame system was used to represent objects in the algorithm and domain spaces and to maintain historical context. For example, algorithm components, links, items, and assertions were represented as objects, as were points and line segments. The operators in the problem space were represented by MACLISP functions, Search control was specified by rules about when and how to modify the problem space context (what operators to apply, how to instantiate them, etc.), which were implemented in OPS5. Also, a method for recording protocols of human designers and relating them to a history of operations in a design session of the automatic system was implemented. More details of the implementation are described elsewhere [18], [27].

Experimentation with the preliminary system lead to a better understanding of how to implement an automated designer more effectively. Some significant revisions to the details of the framework have been made since the original version, and a new and cleaner implementation in a more sophisticated programming environment is being considered.

V. Design Automation Strategies

This section summarizes our theory of human design and compares the framework based on that theory to some of the other approaches suggested for fully or partially automating algorithm design and for automatic programming. It also discusses how the methods might be extended to handle the problems in other contexts, such as interactive design.

A. *Summary of Human Design*

Several of our designers succeeded in creating convex hull algorithms. The algorithms and key discoveries of designers D1 and D2 have already been described. D1's generate-and-test algorithm had a disappointing worst case run time proportional to the cube of the number of input points. But D1 would never have been able to design the anticipated linear algorithm; it can be shown that the problem of finding a convex hull is related to the problem of sorting, so under conventional assumptions it must be an $N \log N$ problem. Eventually D1 went on to try a divide-and-conquer approach that, with a little help from the experimenters, became a successful $N \log N$ algorithm similar to D2's. Some other designers successfully recreated some convex hull algorithms that they had heard or read about but did not remember very clearly. (Many interesting convex hull algorithms have been described in the literature [20].) Still other designers failed to find any algorithm at all. We also gave our designers some other problems. They were asked for algorithms to find the closest pair of points from a given set or the intersection points of a set of vertical and horizontal lines. Most designers quickly suggested brute force algorithms (which have worst-case run time that is the square of the size of the input) but were unable to find any of the faster algorithms.

The methods observed in human design are quite varied. Selecting and sticking with a kernel idea provides a necessary focusing of attention, and using trial execution as an assertion propagation mechanism continues that focus and avoids the extensive search process that unlimited inference or search through the network of all refinements would entail. Of course if specific knowledge about the domain or about algorithm design is available, it can be used to limit search by suggesting refinements directly. A powerful source of creativity is the use of visual reasoning about specific examples, which paves the way for discoveries about key concepts in algorithms. Although our current set of studies of human designers has provided many good ideas for a theory of design, we would like to do more studies on other types of algorithms and with even more expert algorithm designers.

In general, the designers' successes were highly correlated with their interest in and background in algorithm design. Some problems that they had stemmed from an incomplete (or totally absent) understanding of design principles such as divide and conquer (which is very relevant to the examples we gave). Other problems seemed to be due to impatience with methodically following a design strategy. In some cases, the designers tried to mix aspects of the design from two different approaches. This typically failed when they tried to mix subparts of different types of principles but succeeded when they tried to reuse facts or conjectures from the geometric domain that were learned in an earlier design.

B. *Automatic Programming*

Automatic programming is that ever receding goal of automating the programming of everything the user wants with a minimal amount of specification. Automatic programming encompasses 1) algorithm design, 2) program synthesis, and 3) the problem of managing complexity in programming in the large.

Algorithm design has been defined in Section I-A as the process of producing a computationally feasible program sketch (that is relatively complete and consistent) from a specification of what is to be accomplished. We refer here to the hour-level form of algorithm design, not research design. This routine design often precedes program synthesis.

Program synthesis is the process of choosing data structures and access functions to transform a given algorithm specification into concrete code in a conventional programming language. Like algorithm design, program synthesis requires intelligence, especially to produce extremely efficient code, but it probably can be achieved with more straightforward techniques.

As has been pointed out by others [2], [11], full-fledged automatic programming requires the incorporation of domain knowledge as well as detailed coding knowledge. Furthermore, programming in the large must be supported by effective bookkeeping.

Few concrete results in the area of automatic programming that encompass all three of these aspects have been presented. Perhaps the notion of working in multiple spaces, and in a domain space in particular, may prove valuable in automating the entire programming process.

C. *Formal Derivation*

The formal derivation approach has been proposed for both algorithm design and program synthesis [5], [22], [26]. Formal derivation methods share with the design methods described here a refinement strategy based on a few, largely syntactic, transformations, but differ in that the transformations always preserve correctness. It is assumed that the specifications are correct and complete, and since the transformations require and guarantee correctness, then the intermediate states and the results are also correct and internally consistent. The operations of the transformations—defining new constructs, expanding definitions ("unfolding"), and noticing instances of definitions that have arisen after rearrangement and simplification of the algorithm constituents ("folding")—are similar to some of the operations that we have noted in human design.

One way that the formal approach differs from the framework for design described here is that it requires that terms be defined by axioms or equations and does not allow the use of terms defined only in a domain space. Also, in the formal approach, transformations are instantiated via axioms about the domain or algorithmic constructs; in the framework for design described here, they can be instantiated by similar knowledge based on formal definitions, by arbitrary selection, or by guesses based on observations in the domain space. As discussed earlier, designers can sometimes derive algorithms even if they do not have formal definitions of all the concepts. They need only have operators in the domain space that recognize the concepts, more primitive operators in the domain space that can construct the structures they want to recognize, and techniques for implementing the constructive operators in the algorithm space. In contrast, the formal derivation approaches often have problems with controlling the search process and with creating useful auxiliary definitions—the "aha" or "eureka" steps are often definitions inserted by human interaction. These problems result from there being no clues in the formal approach about how to introduce the right interesting knowledge. The use of guesses in systems that do not require correctness-preserving transformations also poses a potential problem for search control. However, the guesses are only used when no better strategies are available, and the use of trial execution provides a check on unlimited search.

Another way the formal approach, with its requirement for consistency and completeness, differs from our framework is in the handling of boundary conditions and base cases. The formal approach requires that these be defined early on, almost the opposite of the human approach and our framework. Getting the details of the boundary conditions right is one cause of the search problem in formal systems—there are many ways to define these conditions, and selecting the precise specifications or introducing conditionals and filling out the details adds complexity.

For some people, the discipline of taking care of details with a standard methodology releases their creativity. On the other hand, many people find it difficult to state invariants precisely if they must be absolutely correct. Getting the main idea of the invariant is crucial to solving the problem, but stating it formally to avoid such problems as fencepost errors makes it tedious and not obviously productive. For these people, getting the details right immediately is extremely difficult; the overhead of internalizing this methodology is prohibitively high.

Formal derivation systems are being augmented with more detailed knowledge about design techniques so that the search control can be more goal oriented [9] and also with knowledge about example generation [4]. However, this still does not postpone settling all the details (having a domain space lets you finesse formalizing them) or say where the creative definitions come from (cross-fertilization from domain spaces and other algorithms).

D. Program Synthesis by Refinement

The program synthesis problem is complementary to that of algorithm design, although we would expect that many of the same problem-solving techniques are used. The stage at which the algorithm design process stops—when an algorithm is "understood"—should provide an appropriate specification or starting place for program synthesis.

The standard stepwise refinement paradigm in program synthesis [14], [24] assumes a knowledge base of rules that transform abstract constructs into more concrete and more efficient ones. The paradigm involves search over the space of programs defined by that knowledge; no creativity is introduced. The search problem is a bit different since once an algorithm is well defined, the program synthesis problem is usually to find a more detailed program in a standard programming language by selecting concrete data structures and accessing operations. Usually the search space is dense in correct solutions that vary in efficiency, reliability, modifiability, and so on [2]. Past research has investigated the control of the search by efficiency (for example, [15]). Such control is not a definitive solution, but many approaches have been prototyped fairly successfully.

As in most expert systems, in stepwise refinement it is assumed that all the knowledge about how to refine programming constructs is present in the refinement rules. In contrast, the design framework presented here allows the discovery of new programming techniques and algorithms because both the hypothesize and test method and correctness-preserving transformations can be applied. The price paid is that more search at the lower levels and more checking by trial execution is required, and this search is not as easily controlled by efficiency rules as is stepwise refinement for program synthesis.

E. Inductive Inference

Inductive inference from examples is another technique that has been explored, but more for the construction of small programs than for the design of algorithms or large systems. Unambiguously specifying the input/output behavior of algorithms with examples is easier than so specifying the behavior of large programs. However, the inductive approaches usually rely on problem solving using a small set of schemata, with little ability to improvise if none of the schemata match. If the target language is a logical equation-based language with a search mechanism built into the interpreter, then this approach may work [25]. But it is unlikely to produce clever algorithms in conventional languages. Incorporating an inductive inference capability into a program synthesis system makes sense; expecting it to solve the entire program synthesis problem does not.

F. *Program Synthesis by Design*

We hope that algorithm design research will result in aids for program synthesis that avoid hand coding of all the refinement rules. The initial knowledge base requirements should be simplified considerably as a result of the more generic problem-solving abilities such as trial execution, with its low-level means–ends analysis and search, and domain space reasoning. The operators that do not preserve correctness, when judiciously applied, also allow for progress in the absence of complete knowledge (the idea of program synthesis by debugging was actually suggested a decade ago [28]). Putting in more of this creativity should make the automatic programming process more flexible and robust and may even produce better programs.

G. *Interactive Tools*

An interesting question to ask is whether studies of human design suggest any other tools to aid in the design process. Are there some interactive tools that might help people design? Or is there some novel mix of human and machine power that could lead to even better designs?

The conventional wisdom is that people have better insight whereas machines are better at the details. Following this wisdom, the machine could suggest the full range of possible approaches at any one step and the person could decide which to follow, providing the search control.

We could augment this plan by observing that execution is a powerful technique in design. Programs are good at methodically following algorithms for execution, but people frequently see what they expect and miss some of the problems. This would suggest machine support for execution of designs. The execution would expose problems and inconsistencies that people might skip over and the people could suggest some solutions to the problem or suggest new directions to follow.

In addition, the machine support could include a set of rules that continuously monitor simple features of the design, providing a check that preserves almost-correctness but does not guarantee a complete validation. In effect, this makes the machine a sounding board for human design, just as colleagues act as sounding boards. People explain their ideas to others so that they are forced to look at their design from other perspectives (with different assumptions) and go through the design one more time in explaining it.

Building the human/machine communication interface is the hard part of following through with these plans. The two agents must speak the same language in terms of the constructs used in the algorithms and evaluations of algorithms and in describing the design process itself. Each agent must be able to track what the other is doing, which requires both explanation and understanding systems. Building the interface may therefore turn out to be even harder than full automation.

H. *Other Design Tasks*

There are a variety of other design tasks, such as engineering design or VLSI design. Although each of these tasks has its own unique characteristics, we anticipate that some of the concepts discussed in this paper will be relevant to these tasks.

VI. Conclusions

The essence of the framework of design presented here lies in its informality and its use of multiple problem spaces, including example generation and trial execution based on both the domain space and an algorithm design space. These techniques provide a focus of attention to limit search and enable the discovery of key concepts. The framework shares problem-solving techniques with many of the other approaches, but rather than having a single monolithic plan of attack, it shifts techniques depending on the knowledge available.

Several areas need further formalizing and testing. The theories of the processes of discovery and visual reasoning must be extended, made computationally feasible, and incorporated into the framework. Learning and database issues should be explored further. For example, what are the appropriate organization and retrieval techniques for large amounts of information so that kernel plans and key ideas in algorithms and derivations are accessible when relevant? Being able to learn automatically depends on appropriate accessing and on general problem-solving techniques.

The interactions between search, domain knowledge, and programming knowledge seem important in tasks of any appreciable difficulty, including automatic programming and the next generation of expert systems, but several questions about these interactions are still unresolved. For example, it is not well understood how to determine when to stop refining at a given level, how problem spaces are created from problems descriptions, and so on.

Understanding the design process impacts other branches of artificial intelligence. Those that include design tasks, discovery, visual reasoning, the use of examples, and interaction between different types of knowledge could be compared to algorithm design in their organization of knowledge and use of problem-solving techniques. Answering the questions posed for design should shed some light on the general issues in other domains. A side effect of automation—the formalization of algorithm design, analysis, and optimization principles—could also be useful in teaching. Our observations of human design show that examples are useful in the absence of knowledge and therefore probably necessary to teach the knowledge, but having explicit principles is more efficient for the designer.

In summary, the theory of design presented here is a good start toward understanding human algorithm design. The attempt to define a framework for mechanical design based on a formalization of the theory lays a substantial part of the foundation for automation.

Acknowledgment

The research described here is joint work with A. Newell and D. Steier. Many of the ideas I draw on are theirs, but the opinions expressed here are my own. M. A. Wolf recorded and transcribed many of the protocols. Thanks to J. Bentley for helping instigate these studies and thanks to the designers who participated in the experiments for their interest and cooperation; they shall remain anonymous for obvious reasons. Programming support for the implementation and additional input has been provided by B. Bell, L. Covi, B. Kim, R. Kovacs, D. Kulkarny, D. Marshall, J. Muller, E. Pervin, E. Schwabe, M. Taylor, and D. R. Thompson as well as Wolf and Steier. A. Newell, D. Barstow, D. Steier, S. Greenspan, and the referees made valuable comments on earlier versions of this paper.

References

[1] B. Adelson and E. Soloway, "A model of software design," in Chi, Glaser and Farr, Ed., *The Nature of Expertise*, Chi, Glasser, and Farr, Eds. Hillsdale, NJ: Lawrence Erlbaum, to be published.

[2] D. R. Barstow, "A perspective on automatic programming," *Artificial Intell. Mag.*, vol. 5, no. 1, Spring 1984.

[3] ——, "The roles of knowledge and deduction in algorithm design," in *Automatic Program Construction Techniques*, A. W. Biermann, Ed. New York: Macmillan, 1984, ch. 10, pp. 201–222.

[4] W. Bibel and K. M. Horning, "LOPS—A system based on a strategical approach to program synthesis," in *Proc. Int. Workshop Program Construction*, France, Sept. 1980.

[5] A. W. Biermann, Ed., *Automatic Program Construction Techniques*. New York: Macmillan, 1984.

[6] B. Chandrasekaran and S. Radicchi, Eds., *Computer Program Testing*. Amsterdam, The Netherlands: North-Holland, 1981.

[7] D. Cohen, "A forward inference engine to aid in understanding specifications," in *Proc. AAAI-84*, Amer. Ass. Artificial Intell., 1984.

[8] K. A. Ericsson and H. A. Simon, "Verbal reports as data," *Psychol. Rev.*, vol. 87, no. 3, pp. 215–251, May 1980.

[9] M. S. Feather, "A system for assisting program transformation," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 1, pp. 1–20, 1982.

[10] C. L. Forgy, *OPS5 User's Manual*, Dep. Comput. Sci., Carnegie-Mellon Univ. Pittsburgh, PA, Tech Rep. CMU-CS-81-135, July 1981.

[11] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a knowledge-based software assistant," Kestrel Inst., Tech. Rep. RADC-TR-83-195, Aug. 1983.

[12] D. Gries, *The Science of Programming*. New York: Springer-Verlag, 1981.

[13] R. Jeffries, A. A. Turner, and P. G. Polson, "The processes involved in designing software," in *Cognitive Skills and Their Acquisition*, J. R. Anderson, Ed. Hillsdale, NJ: Lawrence Erlbaum, 1981, ch. 8.

[14] E. Kant and D. R. Barstow, "The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis," in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrober, and E. Sandewall, Eds. New York, McGraw-Hill, 1984, pp. 487–513.

[15] E. Kant, *Efficiency in Program Synthesis*. UMI Research Press, 1981.

[16] E. Kant and A. Newell, "Naive algorithm design techniques: A case study," in *Proc. European Conf. Artificial Intell.*, Orsay, France, July 1982. Reprinted in *Progress in Artificial Intell.*, L. Steels and J. A. Campbell, Eds. Ellis Horwood, 1985.

[17] ——, "Problem solving techniques for the design of algorithms," *Inform. Processing and Management*, vol. 20, no. 1, Spring 1984.

[18] ——, "An automatic algorithm designer: An initial implementation," in *Proc. AAAI-83*, Amer. Ass. Artificial Intell., 1983.

[19] J. E. Laird, "Universal subgoaling," Technical Report CMU-CS-84-129 Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-C S-84-129, May 1984.

[20] D. T. Lee and F. P. Preparata, "Computational geometry—A survey," *IEEE Trans. Comput.*, vol. C-33, Dec. 1984.

[21] A. Newell and H. Simon, *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[22] H. Partsch and R. Steinbruggen, "Program transformation systems," *ACM Comput. Surveys*, vol. 15, no. 3, Sept. 1983.

[23] G. Polya, *How to Solve it*. New York: Doubleday-Anchor, 1957.

[24] C. Rich and H. E. Shrobe, "Initial report on a Lisp programmer's apprentice," in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrober, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 443–463.

[25] E. Y. Shapiro, "An algorithm that infers theories from facts," in *Proc. IJCAI-81*, 1981, pp. 446–451.

[26] D. R. Smith, "Top-down synthesis of simple divide and conquer algorithms, Naval Postgraduate School, Tech. Rep. NPS52-82-011, Nov. 1982.

[27] D. M. Steier and E. Kant, "The roles of execution and analysis in algorithm design," *IEEE Trans. Software Eng.*, this issue, pp. 1375–1386.

[28] G. J. Sussman, *A Computer Model of Skill Acquisition*. New York: American Elsevier, 1975.

Elaine Kant received the B.S. degree in mathematics from the Massachusetts Institute of Technology, Cambridge, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

She is currently a member of the Professional Staff at Schlumberger-Doll Research, Ridgefield, CT, on leave from Carnegie-Mellon University, where she is an Assistant Professor of Computer Science, teaching courses in artificial intelligence, software engineering, and algorithm design. Prior to joining the faculty at C-MU, she was a Research Computer Scientist at Systems Control, Inc. Her research interests center around understanding and automating the design, construction, and analysis of algorithms and programs, with a bias toward artificial intelligence approaches. She is author of a book on efficiency estimation in program synthesis, coauthor of a book on programming expert systems in OPS5, and has written a variety of other articles on topics in automatic programming, software engineering, and algorithm design.