Integrating Multiple Sources of Knowledge into Designer-Soar, an Automatic Algorithm Designer[1]

David Steier and Allen Newell
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Designing algorithms requires diverse knowledge about general problem-solving, algorithm design, implementation techniques, and the application domain. The knowledge can come from a variety of sources, including previous design experience, and the ability to integrate knowledge from such diverse sources appears critical to the success of human algorithm designers. Such integration is feasible in an automatic design system, especially when supported by the general problem-solving and learning mechanisms in the Soar architecture. Our system, Designer-Soar, now designs several simple generate-and-test and divide-and-conquer algorithms. The system already uses several levels of abstraction, generalizes from examples, and learns from experience, transferring knowledge acquired during the design of one algorithm to aid in the design of others.

1. Introduction

The frontier of artificial intelligence research has recently been described as "figuring out how to bring more kinds of knowledge to bear [18]". This paper addresses the question of how to bring more kinds of knowledge to bear in an automatic algorithm design system. A designer should be able to use knowledge about general problem-solving, algorithm design, implementation techniques, the application domain and prior experience. We describe a system, Designer-Soar, that both applies knowledge from these different sources and acquires knowledge for transfer to future problems. We adapt and extend techniques used in Designer [9], an initial implementation of an algorithm design system, and exploit the special properties of the Soar architecture [13].

The focus of this research is on the *design* of algorithms, rather than their *implementation*. We define algorithm design to be the process of sketching a computationally feasible technique for accomplishing a specified behavior [9]. Given such a sketch, a programmer may then proceed to an efficient implementation of the algorithm. Although we focus on the early design stages of the total programming process, we expect similar issues of multiple knowledge sources to arise in later stages as well.

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499, and monitored by the Air Force Avionics Laboratory. The research was also supported in part under a Schlumberger Graduate Fellowship to David Steier. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, Schlumberger, or the U.S. Government.

2. The Need for Knowledge Integration in Algorithm Design

By *knowledge* we mean the information about some domain, abstracted from the representation used to encode it and the processing required to make it available [20]. A *knowledge source* is a system that provides access to a body of knowledge. A knowledge source has a specific *representation* of the knowledge, comprising both symbolic structures and the means for interpreting them to influence actions, when appropriate. The problem of integration of multiple sources of knowledge arises from the diversity of representations of the sources, each of which may differ from the representation used to select actions to attain the goals of the system. It is always much easier to design a system with a single source of knowledge, where the representation for action selection can be directly adapted to it.

The kinds of knowledge relevant to algorithm design are described below. Table 1 gives typical processes in design systems that apply this knowledge.

K1. Weak methods: Designing algorithms requires solving problems. Human problem solvers (but generally not automatic systems) usually manage to make some progress, even if they don't have all the knowledge necessary in the form of powerful domain specific techniques. They resort to *weak methods*, such as generating and testing many solutions, depth-first search, etc. Human algorithm designers show particularly heavy use of *means-ends analysis* [11]. They work to reduce the differences between the current state and the goal state, resulting in problem solving driven by difficulties and opportunities detected.

K2. High-level algorithm schemes: Algorithm designers usually begin to attack a problem using design schemes. A common example is *divide-and-conquer*: splitting a problem into subproblems, solving the subproblems separately, and merging the solutions to solve the original problem [24].

K3. Transformations: Once some procedural representation of the algorithm exists, other knowledge suggests ways to reformulate and refine the procedure into a better solution. One generally applicable transformation is *recursion formation* as used in [15]. Transformations more specific to particular situations have been collected in libraries of rules [3] or programming overlays that show correspondences between plans [21].

K4. Correctness: Knowledge suggesting transformations to apply is complemented by other knowledge asserting the application of a transformation will satisfy some design goals. Particularly in

General area	Knowledge	Typical processes for applying knowledge
Problem solving	K1. Weak methods	Predefined search procedure
Algorithm design and implementation	K2. High-level algorithm schemes	Instantiation of design templates
	K3. Transformations	Application of transformation rules
	K4. Correctness	Testing designs, proofs
	K5. Efficiency	Performance analysis
	K6. Target language and architecture	Application of selection rules
Application domain	K7. Domain definitions	Inference from domain axioms
	K8. Domain procedures	Generalization from examples
Past experience	K9. Learned knowledge (K1 - K8)	Derivational analogy

Table 1: Knowledge sources in algorithm design

derivation systems developed by the program transformation community, transformations are known to preserve desired semantic properties in program descriptions. But sometimes there is no knowledge that any known transformations preserve the desired property. One way to acquire this knowledge is to prove the transformation correct; another is to apply a transformation and test the results by executing the resulting design [26].

K5. Efficiency: Knowledge about efficiency may take several forms. It is useful to know that extra effort devoted to finding a divide-and-conquer algorithm may ultimately yield a more efficient algorithm than a generate-and-test scheme [9]. The *balancing principle*, which applies specifically to divide-and-conquer algorithms, state that the optimal divide step produces subproblems of equal size.

K6. Target language and architecture: Knowledge about the intended target language and architecture is mainly important in the later (coding) stages of program synthesis [3, 4]. However, the availability of certain language features (e.g., bit operations) may influence the choice of algorithm used; architectural features (e.g., parallelism) may create opportunities for using algorithms that would be otherwise impossible.

K7. Domain definitions: As with programming in general [4], algorithm specification and design require domain knowledge. For example, in specifying a sorting algorithm, Clark and Darlington use logical axioms and lemmas to give the semantics of the terms *ordered* and *permutation* [5]. Other types of domain knowledge provide performance constraints, input data characteristics, etc.

K8. Domain procedures: Human designers invariably understand the algorithm specifications procedurally. No one designs a sorting algorithm who does not know how to sort. Novice LISP programmers often solve sample problems by hand, and then map the structure of their hand solution onto LISP [1]. Using examples provides a focus of attention for reasoning, excluding irrelevant attributes and unrealistic situations that might result from exclusive use of an abstract domain theory [17].

K9. Learned knowledge (K1 - K8): Human designers learn from experience, acquiring knowledge ranging from specific subprocedures to general design techniques. The importance of reuse for automation of programming is commonly recognized [2, 6, 7, 19], but we are only beginning to understand how automatic programming systems can learn [8, 25].

Each type of knowledge has been incorporated into at least one system for automating algorithm design or other phases of programming. Table 2 indicates the degree to which such systems (including Designer-Soar) integrate multiple sources of knowledge. The top half of the table lists systems that emphasize algorithm design: Designer-Soar, Designer [10], Cypress [24], Cypress-Soar[2] [25], MEDUSA [16], and STRATA [14]. The second half of the table lists systems that emphasize other parts of programming: DEDALUS [15], PSI/SYN [12], Glitter [7], Φ_{NIX} [4], KBEMACS [28] and DRACO [19].

The table shows that no single system integrates all the sources of knowledge. Weak methods, domain procedures, and learned knowledge are used most infrequently. As expected, the systems that emphasize algorithm design use less knowledge of the target language and architecture than the other systems. Also, those systems most strongly driven to handle difficult real-world problems are the ones that incorporate (or plan to incorporate) the most types of knowledge. This is particularly true of Φ_{NIX}, which is intended to produce usable oil well logging software, and DRACO, which has been used for the analysis of domains such as real-time tactical display systems.

3. The Task of Algorithm Design in Designer-Soar

The problem-solving architecture is critical to a system that permits integrating multiple, diverse knowledge sources. The Soar architecture [13] appears to have the requisite generality. Soar systems have solved problems and learned in domains ranging from the traditional AI toy problems such as the eight-puzzle to more complex knowledge-intensive tasks, such as part of the VAX configuration performed by the R1 expert system [22]. Soar also provides a way to explore transfer of learned knowledge both within a design and between designs.

Soar represents tasks as search in problem spaces: sets of states, with operators that move from state to state, and the free ability to search within the space for a desired state that represents task accomplishment. Knowledge is embodied in productions, which are used to select problem spaces, states, and operators. Productions also implement simple operators, complex operators being treated as

[2]Cypress-Soar and Designer-Soar are both Soar-based algorithm designers. Cypress-Soar assumes the use of a deductive engine to formally derive divide-and-conquer algorithms, while Designer-Soar designs these and other algorithms without such a deductive engine, relying heavily on the use of examples as a source of knowledge.

System	K1. Weak methods	K2. Algorithm schemes	K3. Trans-formations	K4. Correctness	K5. Efficiency	K6. Target language	K7. Domain definitions	K8. Domain procedures	K9. Learned knowledge
Designer-Soar	+	+	+	~	→	→	+	+	+
Designer	~	+	+	~	~	~	+	~	
Cypress		+	+	+	→		~		
Cypress-Soar	+	+	+	~			~		+
MEDUSA		+	+	+	+		~		
STRATA		→	→	→		→	→		
DEDALUS		~	+	+		+	+		
PSI/SYN		+	+	+	+	+	~		
Glitter	~	+	+	~	~		+		~
Φ_{NIX}	+	+	+	+	+	+	+		
PA		~	+	+		+			
DRACO		+	+	+	→	+	+		+

+ = Knowledge applied in system
~ = Knowledge applied to a small degree
→ = Knowledge application planned for system

Table 2: Knowledge integration in algorithm design and automatic programming systems

tasks, which are accomplished in appropriate problem spaces. With insufficient or conflicting knowledge, Soar reaches an impasse and generates a subgoal to resolve it. When subgoals are terminated, Soar learns from the experience by building new productions, *chunks*. The left-hand side of a chunk consists of generalized conditions on the working memory elements used in producing the results of the subgoal. If these conditions become true again, the chunk will fire to automatically apply the knowledge from the previous solution, and avoid the subgoal. Transfer of knowledge occurs because the chunk's conditions abstract away from inessential features of the original situation.

Design tasks are given to Designer-Soar in terms of two sets of problem spaces. One defines the *computational model*; its operators are the primitives in which to express the algorithm. The other defines the *application domain* of the algorithm. The desired behavior of the algorithm can be operationally specified by the system knowing how to perform the task in the domain. Thus, Designer-Soar understands sorting if it can sort sequences in the domain space. The algorithm design task is to express sorting in the computational model, which (for algorithm design, as opposed to programming in a specific language) is a space that has abstract operators that correspond to the capabilities of computers. The total specification of the design task may require additional subspaces to define the operators and additional operational knowledge about how to work within the two main spaces. Additional constraints may come from performance requirements on the algorithm or from resource limitations on the design process itself.

This definition of the task of algorithm design separates the understanding of what the algorithm is to do from the creation of an algorithm within some computational framework. If Designer-Soar does not know how to sort at all, then it must first acquire that understanding, which will occur as a capability within the domain space for sorting, namely, a space of abstract sequences. Designer-Soar designs the algorithm by working in the computational space until it can perform the task (e.g., sorting) in a functionally equivalent way to the domain-space algorithm, while satisfying the given constraints. The chunks that are learned for the target computational spaces implement an algorithm.

4. Knowledge Integration in Designer-Soar

We will discuss knowledge integration in Designer-Soar by the example of designing insertion sort, which Designer-Soar synthesizes in the same form as that created by Cypress [23] and Cypress-Soar [25]. The algorithm is two divide-and-conquer algorithms, one for the top level sort function and one for inserting an element into an ordered sequence (the composition subprocedure). *Sort* takes a sequence of elements to be sorted as input. If the sequence is empty, it is returned directly as already sorted; otherwise the sequence is split into its first element and the rest of the sequence. The first element is then inserted into the result of recursively sorting the remainder of the sequence. *Insert* takes an element and an ordered sequence as input. If the sequence is empty, the function returns a sequence containing only the element; otherwise, a conditional subprogram is called to decompose the input into smaller subproblems. The subprogram compares the value of the element parameter to the value of first element of the sequence parameter. If we assume x_0 corresponds to the smaller element, x_1 to the larger element, and x_2 to the remainder of the sequence, the conditional returns a pair of the form $<x_0,<x_1,x_2>>$. The first parameter is then prepended to the result of recursively calling the insertion function on the second parameter (the nested pair).

The target computational model space has dataflow operators that correspond to the conceptual building blocks for algorithms, such as `test` a data item for some predicate, and `apply` some function to data [11]. Algorithm schemes can be encoded procedurally as higher-level operators that are implemented in terms of these build-

Choice (decision cycle)	Effects of choice	Rationale for choice	Knowledge used
C1. (145)	**Specification:** `Sort` integer sequence	Domain procedure acquired by TAQ	K7, K8
C2. (216)	**Sort scheme:** Divide-and-conquer	Abstract lookahead	K1, K2, K4, K8
C3. (227)	**Sort DivConq form:** Simple decompose	Pre-selected preference	K2, [K5]
C4. (227)	**Sort decomposition:** `FirstRest`	Pre-selected preference	K3, K6
C5. (228)	**Sort decomposability test:** `Length`(*Input*) > 0	`FirstRest` decomposes example input	K4, K7
C6. (285)	**Sort directly-solve:** `Id`	Domain op says empty sequence is sorted	K4, K7, K8
C7. (354)	**Insertion scheme:** Divide-and-conquer	Abstract lookahead	K1, K2, K4, K8, K9
C8. (365)	**Insertion DivConq form:** Simple compose	Pre-selected preference	K2, [K5]
C9. (366)	**Insertion decomposability test:** `Length`(*seq-param*) > 0	Can't decompose empty sequence	K4, K7
C10. (410)	**Insertion directly-solve:** `Cons`	`Cons` returns desired result	K1, K6, K8
C11. (535)	**Insertion decompose scheme:** Conditional	Domain execution shows two possibilities for returning results	K1, K3, K4, K8
C12. (557)	**Insertion decompose predicate:** *int-param* ≥ `First`(*seq-param*)	Need ordered result for composition	K4, K7, K8
C13. (582)	**Insertion decompose true branch:** *int-param* first in returned result	Ensure smallest element moved to front in example	K3, K4, K9
C14. (701)	**Insertion decompose false branch:** `First`(*seq-param*) first in returned result	Ensure smallest element moved to front in new example	K3, K4, K9
C15. (750)	**Insertion composition:** `Cons`	`Cons` returns desired result	K1, K6, K8

Table 3: Insertion-sort design in Designer-Soar

ing blocks. For example, Designer-Soar has `compose` and `decompose` operators that correspond to decomposing problems into subproblems, and composing subproblems solutions to get the answer to the original problem,. These are not implemented directly by productions. When attempting to apply these operators in executing an algorithm, an impasse results, with a corresponding subgoal to acquire the knowledge to implement them. Designer-Soar knows an algorithm when it can select *and* implement the appropriate dataflow operators to compute the correct output given any legal input. This uniform procedural representation of abstractions at levels varying from algorithm schemes down to computational primitives is crucial to knowledge integration in Designer-Soar.

The design of insertion sort is summarized in Table 3. Column 1 labels the design choice and gives the decision cycle at which the choice occured. The decision cycle is the basic unit of problem solving effort in Soar (the entire run takes 883 decision cycles, requiring about 35 minutes on a Sun3/260). Column 2 summarizes the design choices; column 3 gives Designer-Soar's reasons for making the choice. Column 4 lists the types of knowledge used for each choice. We describe the design process in more detail in the following subsections.

4.1. Acquiring the specification and a plan (C1 - C2)

The goal of algorithm design is to be able to know what to do to execute the algorithm on any valid input. Designer-Soar makes design choices while repeatedly executing both the domain procedure and the partially designed algorithm at varying levels of abstraction. The results of the executions are used to detect problems and opportunities that guide the design, so that the design process can be characterized more as means-ends-analysis than as strict top-down refinement.

Designer-Soar first attempts to execute the insertion-sort algorithm (which doesn't exist yet) to see what needs to be done. An impasse is generated because Designer-Soar has not yet learned how to select between the computational operators it could apply as a first step. While resolving this impasse, Designer-Soar learns that the algorithm should have the functionality of the high-level domain operator "`sort` a sequence into nondecreasing order." Designer-Soar already has the knowledge to implement `sort` in domain spaces, but acquires the knowledge to select the `sort` operator for this run by translating an external task description into an internal description of the operator selection knowledge, and then interpreting this description to build a procedural representation of the knowledge as an operator selection chunk [29].

Knowledge that the algorithm must sort is used to select an operator to apply in the computational space. The operator selected implements the first step of divide-and-conquer: a test to check if the input is decomposable. The operator is selected according to the results of a subgoal to evaluate the choice by lookahead, i.e., trying out the operator to see if it leads to a final state. The exact test for decomposability is not yet known, and no concrete example has

been produced to refine it. Therefore, the lookahead takes place in an abstracted version of the computational space, in which the operators can be applied without knowing the missing details[3]. Currently, Designer-Soar only uses type knowledge in abstracted execution, but we expect to propagate efficiency constraints as well.

4.2. Designing the top level sort (C3 - C6)

Given the decision to execute a divide-and-conquer algorithm, Designer-Soar attempts to apply the first step, testing for decomposability. The test is not known, but the system knows that execution on concrete examples is useful for refining tests, so a new execution pass is begun. An example of the input required, a sequence of integers, is incrementally generated by adding elements to an initially empty sequence until it has two elements. Designer-Soar knows that sequences with two or more elements will probably not be boundary cases (in contrast to zero or one elements). To find the test for decomposability, Designer-Soar looks ahead for a possible decomposition operator. We have told the system to select the `FirstRest` operator for decomposition in this case (which leads to insertion sort rather than other sorting algorithms), splitting off the first element from the sequence containing the second element. The precondition for applying `FirstRest` — that the sequence has at least one element — is used as the test for decomposability.

The subproblems from this decomposition are then solved. The first subproblem is an element rather than a sortable sequence, and is passed to the composition as is. The remaining subproblem is a sequence, and test-case execution is recursively invoked to sort it. It is decomposed into an element and an empty sequence. The test for decomposability applied to the empty sequence returns *false*, so it must be sorted directly. Applying the domain operator to sort the empty sequence shows that the computational space operator `Id` (identity) operator has the necessary functionality.

4.3. Designing the insertion algorithm (C7 - C10)

While making these selection and implementation choices, Designer-Soar learns chunks. Because of the execution paths followed so far, the chunks learned encode the entire structure of the top level of the insertion sorting algorithm. However, an impasse arises when the system tries to combine the element and the sorted remaining sequence, because it does not know how to implement the necessary `Compose` operator. It decides to implement the operator by divide-and-conquer, making the selection by the same abstract lookahead planning used for the higher level algorithm; indeed some of the chunks learned then now apply, speeding up the problem-solving, showing the integration of learned knowledge. To refine the insertion subalgorithm further, the 2-element example from the higher-level execution context is used again. We told the system to assume the decomposition for insertion would have to be custom designed and that the composition would be selected from simple known operators. In fact, decompose need not be applied for the current input, which is an element and an empty sequence. The check for the empty sequence is made into a test for decomposability. A comparison to results of execution in domain space shows that the operator `Cons` suffices to insert an element into an empty sequence.

4.4. Designing the decomposition of the insertion algorithm (C11 - C15)

Though Designer-Soar has solved the problem of insertion for the current example, it knows that the purpose of the execution is not only to obtain the answer, but also to exercise the execution paths so that it learns the algorithm. In finding that the test for decomposability returns *false*, it remembers it must come back to find out what happens when a test returns *true*. It generates a new example to force the execution down the untried path, adding an element to the sequence to make it decomposable.

In processing the new example and looking at the results of domain execution, the system discovers that it needs to handle several cases separately for the decomposition. This leads to a conditional algorithm, where inputs are an element and an ordered sequence. Another execution pass refines the predicate of the conditional to compare the value of the element to the value of the first element of the sequence, and also refines the *true* branch to ensure that the smaller of the two elements is moved to the front. Some of the knowledge learned in refining the *true* branch is used together with a new example to refine the *false* branch analogously. While finishing the design, the composition operation of the insertion is refined to `Cons` the element (known to be smallest) to the front.

4.5. Learning

Prior experience is a significant source of knowledge for design. Soar's learning mechanism, chunking, is so tightly integrated into Designer-Soar that the boundary between problem-solving and learning has disappeared: designing an algorithm is equivalent to learning to execute it and the current Designer-Soar requires that chunking be on to run. However, a slightly earlier version of Designer-Soar did permit no-chunking runs so as to isolate the effects of learning. Figure 4-1 shows the cumulative problem-solving effort needed to design two simple algorithms in sequence, with and without chunking. On the left, the first algorithm finds the subset of elements satisfying a given predicate in a given set, the second finds the intersection of two sets. On the right, the two algorithms are insertion sort and merge sort. There is a significant savings from learning in both pairs of algorithms: 28% for the set algorithms and 69% for the sorting algorithms, illustrating that the benefits of learning increase as the designs get more complex. Furthermore, the slope of the learning graph decreases during the design of the second algorithm in each pair, suggesting transfer across, as well as within, the similar designs. We found that without the chunks from the design of insertion sort, the learning run for merge sort takes 860 decision cycles, an increase of 56% over the 551 needed with those chunks.

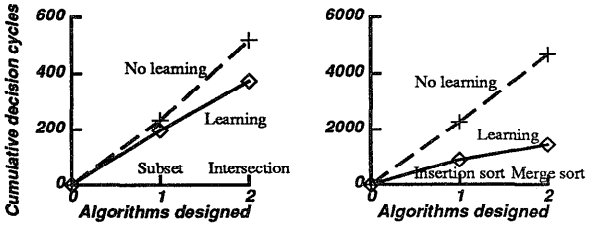

Figure 4-1: Effects of learning in Designer-Soar

5. Summary

Returning to our list of knowledge sources, we summarize the mechanisms used for integrating each source into Designer-Soar. The Soar architecture directly supports access and use of two of the knowledge sources: weak method search (K1) results from Soar's default behavior in knowledge-lean situations, and learned

[3] A similar use of abstraction in Soar has been described for a partial reimplementation of R1, the VAX configuration expert system [27].

knowledge (K9) is applied when chunks fire. The problem spaces that are specific to Designer-Soar support integration of the other sources. Knowledge of the high-level algorithm schemes (K2) and of possible transformations (K3) is encoded in the operators in the computational spaces. Similarly, knowledge about application domain definitions and procedures (K7 and K8) is embodied in the structure of the domain spaces. Concerns of correctness (K4) are addressed by execution in both computational and domain spaces, and means-ends analysis on the results. Though we have not yet focused on knowledge about efficiency (K5) or the target language and architecture (K6), there is a clear role for integrating these sources in terms of selection knowledge in the computational space, or even computational spaces with different functional operators.

Currently, Designer-Soar designs both generate-and-test and divide-and-conquer algorithms, but only simple instances of each. We are now reorganizing Designer-Soar to give it greater generality and robustness. We expect that the results we obtain in integration of multiple knowledge sources, including learning, will have implications not only for algorithm design, but for other applications as well.

Acknowledgement

We thank Erik Altmann, Gregg Yost and Kathy Swedlow for their comments on earlier drafts of this paper.

References

1. Anderson, J. R., Farrell, R., and Sauers, R. "Learning to program in LISP". *Cognitive Science 8*, 2 (1984), 87-129.

2. Balzer, R. "A 15-year perspective on automatic programming". *IEEE Transactions on Software Engineering SE-11*, 11 (1985), 1257-1268.

3. Barstow, D. R.. *Knowledge-Based Program Construction.* North Holland Publishing Company, Amsterdam, 1979.

4. Barstow, D. R. "Domain-specific automatic programming". *IEEE Transactions on Software Engineering SE-11*, 11 (1985), 1321-1336.

5. Clark, K. and Darlington, J. "Algorithm classification through synthesis". *Computer Journal 23*, 1 (1980), 61-65.

6. Dietzen, S. R., and Scherlis, W. L. Analogy in program development. Proceedings of the Second Conference on the Role of Language in Problem Solving, April, 1986, pp. 95-113.

7. Fickas, S. "Automating the transformational development of software". *IEEE Transactions on Software Engineering SE-11*, 11 (1985), 1268-1277.

8. Hill, W. L. Machine learning for software reuse. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, August, 1987, pp. 338-344.

9. Kant, E. "Understanding and automating algorithm design". *IEEE Transactions on Software Engineering SE-11*, 11 (1985), 1361-1374.

10. Kant, E., and Newell, A. An automatic algorithm designer: An initial implementation. Proceedings of The National Conference on Artificial Intelligence, August, 1983, pp. 177-181.

11. Kant, E. and Newell, A. "Problem solving techniques for the design of algorithms". *Information Processing and Management 20*, 1-2 (1984), 97-118.

12. Kant, E. and Barstow, D. R. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. In *Interactive Programming Environments*, McGraw-Hill, New York, 1984, pp. 487-513.

13. Laird, J. E., Newell, A., and Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence 33*, 1 (1987), 1-64.

14. Lowry, M. R. Algorithm synthesis through problem reformulation. Proceedings of the National Conference on Artificial Intelligence, August, 1987, pp. 432-436.

15. Manna, Z. and Waldinger, R. "Synthesis: dreams => programs". *IEEE Transactions on Software Engineering SE-5*, 4 (1979), 294-328.

16. McCartney, R. D. Synthesizing algorithms with performance constraints. Proceedings of the National Conference on Artificial Intelligence, August, 1987, pp. 154-159.

17. Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. "Explanation-based generalization: A unifying view". *Machine Learning 1*, 1 (1986), 47-80.

18. Mostow, D. J. "What is AI? And what does it have to do with software engineering?". *IEEE Transactions on Software Engineering SE-11*, 11 (1985), 1253-1256.

19. Neighbors, J. "The Draco approach to constructing software from reusable components". *IEEE Transactions on Software Engineering SE-10*, 5 (1984), 564-574.

20. Newell, A. "The knowledge level". *Artificial Intelligence 18*, 1 (1982), 87-127.

21. Rich, C. Inspection methods in programming. Tech. Rept. AI-TR 604, Massachusetts Institute of Technology, June, 1981.

22. Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., and Orciuch, E. "R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture". *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 5 (1985), 561-569.

23. Smith, D. R. Top-down synthesis of simple divide-and-conquer algorithms. Tech. Rept. NPS52-82-011, Navy Postgraduate School, November, 1982.

24. Smith, D. R. "Top-down synthesis of divide-and-conquer algorithms". *Artificial Intelligence 27*, 1 (1985), 43-96.

25. Steier, D. M. Cypress-Soar: A case study in search and learning in algorithm design. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, August, 1987, pp. 327-330.

26. Steier, D. M. and Kant, E. "The roles of execution and analysis in algorithm design". *IEEE Transactions on Software Engineering SE-11*, 11 (1985), 1375-1386.

27. Unruh, A., Rosenbloom, P. S., and Laird, J. E. Dynamic abstraction problem solving in Soar. Proceedings of the Third Annual Conference on Aerospace Applications of Artificial Intelligence, October, 1987, pp. 245-256.

28. Waters, R. C. "The programmer's apprentice: a session with KBEmacs". *IEEE Transactions on Software Engineering SE-11*, 11 (1985), 1296-1320.

29. Yost, G. R. TAQ: Soar Task Acquisition System User Manual. In preparation.