INTEGRATING MULTIPLE SOURCES OF KNOWLEDGE INTO DESIGNER-SOAR, AN AUTOMATIC ALGORITHM DESIGNER

Technical Report AIP - 52

David Steier & Allen Newell

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pa. 15213

The Artificial Intelligence and Psychology Project

Departments of Computer Science and Psychology
Carnegie Mellon University

Learning Research and Development Center
University of Pittsburgh

Approved for public release; distribution unlimited.
INTEGRATING MULTIPLE SOURCES OF KNOWLEDGE INTO DESIGNER-SOAR, AN AUTOMATIC ALGORITHM DESIGNER

Technical Report AIP - 52

David Steier & Allen Newell

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pa. 15213

June 1988

This research was supported by the Computer Sciences Division, Office of Naval Research and DARPA under Contract Number N00014-86-K-0678, and 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 representint the official policies, either expressed or implied, of the Defense Advance Research Projects Agency, Schlumberger, or the U.S. Government. Reproduction in whole or in part is permitted for purposes of the United States Government. Approved for public release; distribution unlimited.
## Integrating Multiple Sources of Knowledge into Designer-Soar, an Automatic Algorithm Designer

**Personal Author(s):** David Steier and Allen Newell

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.
Integrating Multiple Sources of Knowledge into Designer-Soar, an Automatic Algorithm Designer

David Stier 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.

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

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F30615-87-C-1499, and monitored by the Air Force Avionics Laboratory. The research was also supported in part under a Schlumberger Graduate Fellowship to David Stier. 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.

To Appear.

To Appear.

To Appear.
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, states 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 sub-procedures to general design techniques. The importance of reuse of an abstract domain theory is commonly recognized [2, 6, 7, 19], but we are only beginning to understand how automatic programming systems can learn [8, 25].

Table 1: Knowledge sources in algorithm design

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

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 [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], @NEX [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 @NEX, 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 domain 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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Designer-Soar</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Designer</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cypress</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Cypress-Soar</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>MEDUSA</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>STRATA</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>DEDALUS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PSI/SYN</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Gliser</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Φ_Ner</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PA</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DRACO</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

+ = 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
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 occurred. 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 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
4.4. empty sequence. Figure check for the empty sequence is made into a test for decom-
current input, which is an element and an empty sequence. The
known operators. In fact, decompose need not
higher-level execution context is used again. We told the system to
solving. showing the integration of learned knowledge. To refine
of the chunks learned then now apply, speeding up the problem-
by
necessary Compose operator. It decides

4.3. Designing the insertion algorithm (C7 - C10)
While making these selection and implementation choices,
Soar's learning mechanism, chunking, is so tightly integrated into
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. Fur-
thermore, 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.

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 decom-
possibility returns false, it remembers it must come back to find out
what happens when a test returns true. It generates a new example
force the execution down the untried path, adding an element to
the sequence to make it decomposable.

4.5. Learning
Prior experience is a significant source of knowl
dge 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. Fur-
thermore, 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
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 or even computational spaces with different functional operators.

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

References