CYPRESS-Soar: A case study in search and learning in algorithm design

David Steier
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15208 USA

Abstract

This paper describes a partial reimplementation of Doug Smith's CYPRESS algorithm design system within the Soar problem-solving architecture. The system, CYPRESS-SOAR, reproduces most of CYPRESS' behavior in the synthesis of three divide-and-conquer sorting algorithms from formal specifications. CYPRESS-Soar is based on heuristic search of problem spaces, and uses search to compensate for missing knowledge in some instances. CYPRESS-Soar also learns as it designs algorithms, exhibiting significant transfer of learned knowledge, both within a single design run, and across designs of several different algorithms. These results were produced by reimplementing just the high-level synthesis control of CYPRESS, simulating the results of calls to CYPRESS* deduction engine. Thus after only two months of effort, we had a surprisingly effective research vehicle for investigating the roles of search, knowledge, and learning in this domain.*

I Introduction

Good human programmers have at least two remarkable abilities: they manage to produce programs in the face of incomplete knowledge, and they make use of previous experience in solving new problems. How could we get automatic programming systems to produce the same intelligent behavior? Al-based performance systems in other domains compensate for incomplete knowledge by searching through a space of possible solutions, and there exist a variety of mechanisms for learning from experience. However, automatic programming research has so far produced only a few systems that either search or learn, and, to my knowledge, none that do both. This is true despite the field's growing acknowledgement of the importance of both search and learning [2,3,4].

This paper describes a prototype system that both searches and learns while performing part of an automatic programming task. An algorithm design system, previously built within a special-purpose framework, was reimplemented in a more general problem-solving architecture with built-in search and learning capabilities. The previously implemented system is Doug Smith's CYPRESS [11,12,13], which is most noted for its design of divide-and-conquer algorithms. The foundation for the reimplementation is Soar [7,8], an architecture for general intelligence developed by John Laird, Allen Newell, and Paul Rosen bloom. The combined system, CYPRESS-Soar, produces the bulk of three of CYPRESS' sorting algorithm derivations, and takes advantage of the properties of Soar to search and learn while doing so.

In Section II, I describe CYPRESS and its approach to the synthesis of divide-and-conquer sorting algorithms and in Section III I give an overview of the Soar architecture. The remaining sections discuss CYPRESS-Soar, presenting the following results:

- Performance without fixed design strategies: CYPRESS-Soar uses any knowledge available at run-time to decide when algorithm refinement operators should be applied. If the knowledge is unavailable, CYPRESS-Soar automatically falls back on general problem-solving methods, initiating lookahead search to evaluate the possibilities. In contrast, design strategies control operator application in CYPRESS, and any necessary search must be guided by an expert user. (Section IV)
- Transfer of learned knowledge: CYPRESS-Soar knows what goal it is working on, and caches the result of the goal for future use. Because some goals show up more than once, this learning mechanism reduces problem-solving effort, both within the design of a single algorithm, and on later designs of different algorithms. CYPRESS does not learn, and consequently can not take advantage of repeated subgoals. (Section V)

Section VI concludes with a discussion of several issues involved in extending the prototype CYPRESS-Soar system into a more general automatic algorithm designer.

*This research was supported in part by the National Science Foundation under Grant DCR-8412139. and in part by the Defense Advanced Research Projects Agency under Contract F336 15-81-K-1539. Work on Cypress-Soar was begun while the author was visiting Kestrel

II How CYPRESS designs divide-and-conquer algorithms

CYPRESS is a semi-automatic system that derives algorithms from formal specifications. It works by top-down refinement of program schemes, or templates, which represent abstractions such as *divide-and-conquer* and *generate-and-test.* A problem specification is matched against a program scheme, and with the aid of a design strategy, decomposed into specifications of simpler problems. This problem reduction process continues recursively until a specification can be solved directly by primitive operators known to the system. When more than one design strategy is applicable, or more than one operator matches a specification, the user makes a selection among alternatives.

CYPRESS spends most of its time in calls to RAINBOW, its deduction engine. RAINBOW performs a generalized version of theorem-proving known as *antecedent derivation* [11]. Given a set of hypotheses, H and a goal formula, G, RAINBOW tries to give the weakest possible precondition, or antecedent, P such that the hypotheses in H conjoined with P imply G. If P is just *true,* then G is already a valid formula given H. In the context of algorithm synthesis, RAINBOW is used to reason backwards from output conditions to test if a specification is satisfied. If it is not satisfied, the derived antecedent is used as dictated by the active design strategy as the basis for further action. Viewed in problem-solving terms, RAINBOW provides a sophisticated form of means-ends analysis.

The input to CYPRESS is a formal specification of the problem to be solved, giving the input and output domains (types, or sets), and input and output conditions for the problem. A specification of the problem of sorting lists of natural numbers from [13] is

```
SORT: X = Z such that Bag:x=Bag:z A Ordered:z
        where SORT: LIST(N)-->LIST(N).
```

The SORT function maps the input *x* into the output *z*. An implicit input condition, *true* is assumed. The output condition is that the bag (multiset) of elements in *x* is the same as in z, and z is ordered. The specification assumes pre-existing knowledge of the terms "Bag" and "Ordered".

The sorting problem is amenable to a divide-and-conquer solution. The CYPRESS scheme for divide-and-conquer is expressed in a typed functional programming language, a derivative of Backus' FP [1]:

```
F:x    if
          Primitive:x —► Directly_Solve:x []
          --Primitive:! —► Compose • (G x F) • Decompose:x
       fi
```

The scheme abstractly specifics how to compute the value of *F* on input r. If *x* is a base-case input, then solve it directly; otherwise, decompose *x* into two subproblems, recursively solve one and apply an auxiliary function *G* to the other, then compose the results.

To instantiate this scheme for a given specification, CYPRESS creates subspecifications for Directly.Solve, Compose, and Decompose, and then attempts to design algorithms for these subspecifications or to verify that known operators satisfy them. Along the way, the auxiliary function G is refined, usually either to a recursive call to the top-level algorithm, or to the identity operator, Id. The Primitive control predicate is derived as the input condition to Decompose.

The design strategies for instantiating the divide-and-conquer scheme allow the choice of either simple decomposition or simple composition operators. For sorting, choosing a simple decomposition operator leads to insertion-sort and mergesort, while a simple composition operator leads to selection-sort and quicksort. The algorithms for the top levels of the quicksort and partition algorithms are

$$\begin{aligned}&\text{Qsort}.x \Leftarrow \text{if}\\ &\quad x_0 = \text{nil} \vee \text{Rest}.x_0 = \text{nil} \rightarrow \text{Id}.x\ [\!]\\ &\quad x_0 \neq \text{nil} \wedge \text{Rest}.x_0 \neq \text{nil} \rightarrow \text{Append} \cdot \langle \text{Qsort} \times \text{Qsort} \rangle \cdot \text{Partition}.x\\ &\text{fi}.\end{aligned}$$

$$\begin{aligned}&\text{Partition}.x \Leftarrow \text{if}\\ &\quad \text{Rest} \cdot \text{Rest}.x = \text{nil} \rightarrow \text{Partition\_Directly\_Solve}.x\ [\!]\\ &\quad \text{Rest} \cdot \text{Rest}.x \neq \text{nil} \rightarrow \text{Partition\_Compose} \cdot \langle \text{Id} \times \text{Partition} \rangle \cdot \text{FirstRest}.x\\ &\text{fi}\end{aligned}$$

The top level function for quicksort is a divide-and-conquer scheme that uses the simple composition operator Append, while the partition algorithm called by quicksort is the scheme instantiated to use the simple decomposition FirstRest (equivalent to returning the head and tail of a list). The top level function Qsort is the conventional quicksort: If x is of length 0 or 1, it returns x. Otherwise, it partitions x into two sublists, sorts them both and appends the results. The partition algorithm created by CYPRESS differs from the standard partition algorithm in that it is a divide-and-conquer algorithm, and it does not use a partitioning element If there are only two elements in the list, it produces two singleton lists, with the smaller element in the first list. Otherwise, it builds up two lists by recursively partitioning off the rest of the list, and adding each element in turn to the appropriate sublist as determined by its value. The functions implementing Partition Directly Solve and Partition Compose for the partition algorithm were also produced by CYPRESS, but are not shown here.

III Soar

CYPRESS-Soar is built in Soar, an architecture developed to study the computational mechanisms necessary for intelligent behavior [6]. Soar is based on the hypothesis that all goal-directed cognitive activity can be represented as search in a problem space. A problem space is defined by a set of states, and a set of operators to move from state to state. Soar uses knowledge, represented as productions, to generate and select problem spaces, states, and operators to move towards a goal state. The knowledge accumulates in the elaboration phase and is used as a basis for action in the decision phase of each decision cycle, the basic unit of problem-solving effort in Soar-based systems. Several productions may fire in accumulating the knowledge to make a given decision (about 4-5 productions per decision cycle for CYPRESS-Soar).

Often the knowledge directly available in a given situation is insufficient to determine the next thing to do immediately. In Soar, such situations are called impasses, subgoals arise exclusively in response to these impasses. The types of impasses that may arise in Soar systems are determined by the architecture. For example, a common impasse, operator-tie, occurs when several operators are proposed as acceptable for application to a given state, and there is insufficient knowledge to choose between them. The subgoal to resolve an operator-tie impasse would be satisfied when the system acquired knowledge indicating that one of the operators initially causing the tie is actually preferable to all other candidates.

When Soar finishes working on a subgoal, it can learn from its experience by building productions called chunks for use in future problem solving. The conditions of a chunk are the features of the pre-impasse situation that were used to produce the results of the subgoal, where the results are those working-memory elements created in the subgoal (or its subgoals, etc.) that are accessible from a supergoal. The actions of a chunk are based on the results. At first glance one would expect chunking to yield nothing more than rote learning, but generalization does occur because chunks test only relevant attributes of the problem-solving context [7].

The Soar architecture has by now been subjected to extensive study and experimental use in many applications. 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 the part of VAX configuration performed by the RI expert system [10]. Other work has demonstrated that Soar can exhibit the behavior of a wide variety of problem-solving methods [5]. Also, chunking, which was developed from psychological models of human learning [9], has proven to be a powerful mechanism capable of improving performance in many applications. Therefore, while Soar is not yet a complete model for intelligent behavior, it already demonstrates many of the characteristics necessary for such a model.

IV CYPRESS-Soar

One encodes a task in Soar by writing productions implementing one or more task problem spaces. CYPRESS-Soar consists of 195 Soar (Version 4.4) productions. Of these productions, 60 contain Soar's default search control knowledge, and the remaining 135 (comprising about 4500 lines of text) are task-specific. This section describes the problem spaces in CYPRESS-Soar implemented by the task-specific productions. The derivation of the quicksort algorithm discussed earlier is summarized to illustrate the operation of the system.

CYPRESS-Soar follows the same principles of "Soarware engineering" used in the construction of RI-Soar [10]. The top level problem space attempts to apply a single operator (Conf igure-backplane in RI-Soar, Synthesize in CYPRESS-Soar) to solve the problem. Because no productions implement this operator directly. Soar creates a subgoal to implement it, selecting a special problem space associated with this operator. This problem space in turn contains other operators which may be themselves implemented in other problem spaces, or else implemented directly by productions that fire in the appropriate context In this manner, tasks are decomposed into problem spaces in the same way that large conventional programs are broken up into modules. Other problem spaces are evoked to handle search control, such as the selection of operators.

Inn CYPRESS-Soar, the Synthesize operator is implemented by subgoaling into the Synthesize problem space. In this space, operators may be applied to synthesize either a divide-and-conquer algorithm, or a simple conditional. This paper focuses on the creation of divide-and-conquer algorithms, for which the states in the Synthesize space arc the successive refinements of the divide-and-conquer program scheme. The initial state is a completely abstract scheme, and the desired state is a scheme with all of its parts refined to known operations. The Specify-decompose, Specify-auxiliary, Specify-primitive, Specify-compose, and Specify-directly-solve operators map directly onto the parts of the scheme that need to be refined. Specify-ordering chooses a well-founded ordering on the input domain to be preserved by the decomposition operator; this is necessary to guarantee that the synthesized algorithm terminates. Each of these operators is implemented in its own problem space, where the knowledge about divide-and-conquer taken from CYPRESS is used for operator selection and implementation. In the Specify-decompose, Specify-compose, and Specify-directly-solve problem spaces, the Synthesize operator may be recursively invoked to satisfy specifications for complex algorithms.

For example, the Specify-decompose problem space is used to refine the divide-and-conquer algorithm's Decompose operation. For decomposing lists, operators in this problem space might choose FirstRest, which returns the element at the head of the list along with the rest of the list. However, suppose one desires an algorithm that uses a simple method of composing lists, say the Cons operator. Then FirstRest will probably not suffice for the decomposition, because FirstRest does not satisfy a strong enough output condition to guarantee correct results with the chosen composition method. The existence of such constraints imposed by already instantiated parts of the algorithm will make other refinement operators acceptable. In this case, CYPRESS-Soar uses an antecedent derived from the output conditions of the problem specification, the auxiliary operator, and the specification of Cons in conjunction with knowledge about divide-and-conquer to find a stronger output condition for the decomposition specification.

A complete implementation of some of the operators in the Synthesize space would require a separate space for deduction (the kind done by CYPRESS' RAINBOW). In developing CYPRESS-Soar, we wanted to focus on the knowledge involved in making design choices, rather than on deduction, so we simulated RAINBOW'S behavior without implementing it in Soar. CYPRESS-Soar includes rules that return the results of calls to RAINBOW on the particular sets of premises and goal formulae needed for the sorting derivations. While this is inadequate for a fully general design system, it is sufficient for an investigation of search and learning at the design choice level, where the method of deduction does not affect the results.

For the sorting specification, CYPRESS-Soar currently has the knowledge to design insertion-sort, mergesort, and quicksort (though it could easily be extended to design selection-sort). Figure 1 illustrates the behavior of CYPRESS-Soar during its synthesis of quicksort. The first column describes the major choices made in the design. The second column states the design alternative that CYPRESS-Soar selected, and the third column lists any alternatives that were rejected. The fourth column classifies the processes and knowledge involved in making the choices

using the following categories:

- Lookahead: Candidates are evaluated by trying them out to see if they lead to a complete algorithm.
- Derived antecedent. An antecedent is derived from the constraints imposed by previous design choices.
- Domain compatibility. The input or output domain of the proposed refinement be compatible with a domain commitment resulting from a previous design choice or from the specification.
- Operator match: The specification of a known operator matches the specifications set up for the subproblem being solved.
- Preselected preferences: Preferences dictating some of the choices are set up beforehand in order to produce a specific sorting algorithm. Extra attributes added to the specification for each different synthesis trigger these preferences.

The creation of divide-and-conquer algorithms at two levels results from decisions #2 and #8, while the forms of these algorithms are chosen in decisions #3 and #10. The specification for the Partition subalgorithm is first derived in decision #7, but the input condition must be strengthened in order for the specification to be satisfiable. In making decision #15, CYPRESS-Soar suggests candidates for the new input condition, rejecting the first two because they lead to an unsatisfiable subspecification for Directly-solve. With the exception of the synthesis of Directly-solve and Compose for Partition (which require knowledge about conditionals rather than just divide-and-conquer) and the details of the deduction, the behavior in designing quicksort is the same as that of CYPRESS.

Furthermore, this behavior was obtained without the fixed design strategies controlling the synthesis in CYPRESS. CYPRESS-Soar has enough task knowledge so that it can exhibit the characteristic behavior of the design strategies, but it is not constrained to follow a fixed procedure. This is possible for the same reasons that Soar can exhibit the behavior of the weak methods (such as steepest-ascent hill climbing) without having to be programmed explicitly to do so [5]. Soar-based systems propose applying an operator as soon as enough knowledge is available to apply it, as determined by the current state and the preconditions of the operator. The appropriate behavior is thus determined at run-time rather than system design time, by evaluating these operators. If CYPRESS-Soar goes into a subgoal to carry out the evaluation process, the result of the subgoal will be saved as a chunk. Such chunks may produce future behavior corresponding to the effects of one of CYPRESS' design strategies.

V Search and Learning in CYPRESS-Soar

Because of the Soar-based foundation of CYPRESS-Soar, we were able to run several experiments measuring the effects of search control knowledge and learning on the problem-solving effort required for algorithm design. For example, when CYPRESS-Soar has complete search control knowledge, the lengths of solutions reflect only the processing required to fill in all the details of the algorithm. It is possible to remove the search control knowledge from CYPRESS-Soar so that search is required as well, by removing 15 of the 195 productions. CYPRESS-Soar still yields correct algorithms under these conditions, albeit with greater problem-solving effort (a factor of 2 to 4 more decision cycles).

Then using chunking, we can measure the effects on solution lengths of learning from experience, not only on different algorithms, but also with different levels of search control knowledge. In some cases, the effects of learning are more pronounced when search control knowledge is absent. This is true not only because the solutions from which effort can be saved are longer, but also because the larger number of impasses lead to more opportunities for learning.

Figure 2 shows the effects of chunking in CYPRESS-Soar with the search control knowledge removed. The three clusters of bars give the lengths of syntheses of insertion-sort, mergesort, and quicksort under various learning conditions. The first bar in each cluster shows the number of decision cycles used by CYPRESS-Soar with no previous learning and learning off during the run. The second bar displays the solution length again with no previous learning, but with learning on during the run. The last three bars in each cluster give solution lengths with previous learning on each of the three algorithms, with learning off during the run."

The second bars of each cluster in Figure 2 illustrate what is known as within-trial transfer, a relatively rare phenomenon in the machine learning literature. Within-trial transfer results in CYPRESS-Soar because knowledge learned early in the design of an algorithm is applied productively later in the same design. In the runs with full search control, learning has little effect, since there is no search, and the operators perform distinct functions. However, in the runs with minimal search control, the search

---

Further experiments with learning on more than one algorithm or more than one learning trial showed no noteworthy additional reductions.

Design choice	Path selected	Path(s) rejected	Rationale
1. Sort specification:	SORT spec	(None)	*Pre-selected preference*
2. Sort scheme:	Divide-and-conquer	Conditional	*Lookahead*
3. Sort ordering:	Length of one list	Length of two lists	*Domain compatibility*
4. Sort DivConq form:	Simple compose	Simple decompose	*Pre-selected preference*
5. Sort compose:	Append	Cons	*Pre-selected preference, Operator match*
6. Sort auxiliary operator:	Sort	Id	*Domain compatibility*
7. Partition specification :	Sort decompose spec	(None)	*Derived antecedent*
8. Partition scheme:	Divide-and-conquer	Conditional	*Lookahead*
9. Partition ordering:	Length of one list	Length of two lists	*Domain compatibility*
10. Partition DivConq form:	Simple decompose	Simple compose	*Lookahead*
11. Partition decompose:	FirstRest	ListSplit	*Pre-selected preference, Operator match*
12. Partition auxiliary:	Id	Partition	*Domain compatibility*
13. Partition primitive:	Length:*Input* = 2	(None)	*Derived antecedent*
14. Partition compose scheme:	Conditional	Divide-and-conquer	*Lookahead*
15. Partition input condition:	Length:*Input* > 1	*true* Length:*Input* > 0	*Derived antecedent, Lookahead*
16. Partition directly-solve scheme:	Conditional	Divide-and-conquer	*Lookahead*
17. Sort primitive:	Length:*Input* ≤ 1	(None)	*Derived antecedent*
18. Sort directly-solve:	Id	(None)	*Operator match*

Figure 1: Quicksort design choices in CYPRESS-Soar

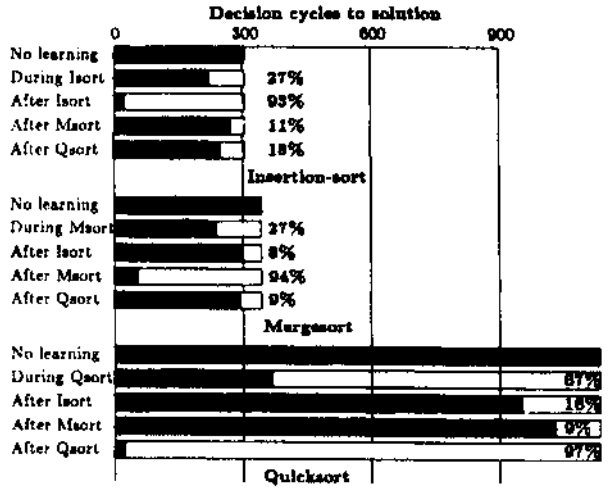

Figure 2: Effects of learning with minimal search control

leads to a large number of similar situations and operator applications, and with the generalization performed by chunking, much redundant problem-solving effort can be saved. This is especially true in in the quicksort synthesis where the system needs to search for the right input condition for partition: the reduction in decision cycles from learning is close to 70%.

In CYPRESS-Soar, the majority of the within-trial transfer results from the need to actually apply an operator after it has been evaluated by lookahead. Since evaluating an operator by lookahead implies computing the result of the operator in the process, after lookahead there will be a chunk that directly creates the new state once the operator is actually selected. The context in which the chunk fires is identical to the one in which the chunk was formed, so the transfer is not very surprising.

The remaining within-trial transfer occurs when an algorithm is synthesized for a subproblem specification in one context, and the same specification shows up again later in another context in the same design. An example of this shows up in the quicksort derivation. In synthesizing partition, CYPRESS-Soar proposed three possible input conditions in searching for the correct one, each time retaining the same output condition. Since the specification for the composition subalgorithm was unaffected by changes in the input condition, the same composition algorithm could be used on each attempt. With minimal search control, the savings from eliminating redundant syntheses of the composition amounted to about one-fifth of the total problem-solving effort of the run without learning. In more complicated algorithms and specifications, one would expect the savings from learning to be even greater.

The last three bars show that CYPRESS-Soar also exhibits *across-trial transfer,* improving performance on subsequent designs of the same algorithm, and *across-task transfer,* applying knowledge learned from the design of one algorithm to subsequent designs of different algorithms. For example, with full search control and no learning, it took CYPRESS-Soar 303 decision cycles to synthesize insertion-sort. As one might expect, it took almost no effort to synthesize quicksort after learning on it, only 20 decision cycles, a savings of 93%. But some of the transfer also occurred after designing the other algorithms: 269 decision cycles after mergesort and 249 after quicksort, savings of 11% and 18% respectively. Reductions of 8-26% in solution lengths were observed across all pairs of algorithms, in both the minimal and the full search control runs.

The transfer occurs mostly because all three algorithms solve the same problem, namely sorting. Each sorting algorithm must decompose lists, and so the same well-founded ordering by list length can be preserved by all three top-level decomposition operators. Other transfer occurs in implementing simple deduction operators, such as negating certain logical expressions. Also, refining *Directly solve* to Id led to transfer between mergesort and quicksort, because in both cases the input is either a single-element or null list. There is no transfer to insertion-sort, because there the input condition specifies only sorting null lists. The representation of the input condition would have to be changed for the matcher, which only fires chunks in the case of an exact syntactic match to the context, to detect that an operator that handled a certain type of input could also handle subsets of that input.

VI Discussion

While a system that designs three algorithms is better than a system that only designs one, CYPRESS-Soar is still not a general automatic algorithm designer, not even within the class of divide-and-conquer algorithms. This is mainly due to the special-case rules for deduction and conditional synthesis, a consequence of the strategic choice in this research to focus first on search and learning in a few divide-and-conquer algorithms. For a general system, one would need to implement additional problem spaces that would perform these functions. We foresee no theoretical barriers to such extensions. The major hurdles to be dealt with are the construction of better interfaces for working with logical formulae in Soar, and the efficiency of a Soar-based deduction engine.

Perhaps most important is that with the existing chunks and the ability to precisely measure across-task transfer, CYPRESS-Soar forms a unique experimental vehicle with which to explore the potential for learning in this domain. The degree to which a Soar-based system can apply chunks to improve its performance depends on how often similar situations are repeated as subgoals while problem-solving. The repetition may be less frequent than it could be because CYPRESS-Soar does not currently break down the deduction into subgoals. It is likely that more transfer would occur if the deduction engine were implemented completely within Soar. More fundamentally the representation used by CYPRESS-Soar may need to capture abstractions common to the algorithms in the syntax of the representation language. On the other hand, it may be the case with these sorting algorithms that no further transfer is possible; that the design processes needed for their creation are just not very similar.

While much work remains to be done, it is encouraging that the current results were obtained in CYPRESS-Soar with only two months' work. This demonstration that a formal theory of design is fully compatible with a general framework for intelligent action was possible only because of the strong foundations available in the work on CYPRESS and Soar. It is also encouraging that the issues raised in the course of developing CYPRESS-Soar have seemed to be worthwhile research topics; in addressing them, we expect to gain useful insights about algorithms and the processes involved in their design.

Acknowledgements

I am most grateful to Doug Smith and the Kestrel Institute for providing me with the opportunity and the environment to begin this research, and to Allen Newell for numerous discussions during the development of CYPRESS-Soar. Doug Smith, Allen Newell, Elaine Kant, John Laird, Craig Knoblock, Dorothy Setliff and Oren Etzioni also made useful comments on earlier drafts of this paper.

References

1. Backus, J. Tan programming be liberated from the von Neumann style?: a functional style of programming and its algebra of programs". *Communications of the ACM 21,8* (August 1978), 613-641.
2. Balzer, R. "A 15-year perspective on automatic programming". *IEEE Transactions on Software Engineering SE-11,* 11 (November 1985).
3. Barstow, D. R. "Domain-specific automatic programming". *IEEE Transactions on Software Engineering SE-*1 1 , (November 1985).
4. 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.
5. Laird, J. E. Universal subgoaling. In *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies,* Kluwe Academic Publishing, Hingham, MA, 1986.
6. Laird, J. E.. Newell, A, and Rosenbloom, P. S. "Soar An architecture for general intelligence". *Artificial Intelligence* (1987). in press.
7. Laird, J. E., Rosenbloom, P. S*. and Newell. A. Towards chunking as a general learning mechanism. Proceedings of AAAI-84, The American Association for Artificial Intelligence, Austin, Texas, August, 1984, pp. 188-192.
8. Laird, J. E., Rosenbloom, P. S., and Newell, A. "Chunking in Soar. The anatomy of a general learning mechanism". *Machine Learning 1,*1 (1986).
9. Rosenbloom, P. S. The chunking of goal hierarchies. In *Universal subgoaling and chunking: The automatic generation and learning of go hierarchies,* Kluwer Academic Publishing, Hingham, MA, 1986.
10. Rosenbloom, P. S.. Laird, J. E., McDermott. J., Newell, A., and Orciuch, E. "RI-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.
11. Smith, D.R. Derived preconditions and their use in program synthesis. In *Sixth Conference on Automated Deduction,* Springer-Verlag, 1982. Lecture Notes in Computer Science 138.
12. Smith, D.R. "The design of divide-and-conquer algorithms". *Science of Computer Programming 5* (1985), 37-58.
13. Smith, D.R. "Top-down synthesis of divide-and-conquer algorithms". *Artificial Intelligence 27,*1 (1985), 43-96.