Synthesizing UNIX shell scripts using derivational analogy: An empirical assessment

Sanjay Bhansali and **Mehdi T. Harandi**
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue, Urbana, IL 61801
bhansali@cs.uiuc.edu harandi@cs.uiuc.edu

Abstract

The feasibility of derivational analogy as a mechanism for improving problem-solving behavior has been shown for a variety of problem domains by several researchers. However, most of the implemented systems have been empirically evaluated in the restricted context of an already supplied base analog, or on a few isolated examples. In this paper, we address the utility of a derivational analogy based approach when the cost of retrieving analogs from a sizable case library, and the cost of retrieving inappropriate analogs is factored in.

Introduction

The process of derivational analogy [Carbonell, 1983] consists of storing derivation histories of problems in an episodic memory, retrieving an appropriate case from the memory that shares "significant" aspects with a current problem, and transferring the derivation of the retrieved solution to the new problem by replaying relevant parts of the solution and modifying the inapplicable portions in light of the new context. In recent years, several systems based on the reuse of a design process have been developed, for domains including program synthesis ([Baxter, 1990, Goldberg, 1989, Mostow and Fischer, 1989, Steier, 1987, Wile, 1983]), circuit design [Huhns and Acosta, 1987, Mostow *et al.*, 1989, Steinberg and Mitchell, 1985], mathematical reasoning ([Carbonell and Veloso, 1988]), human interface design ([Blumenthal, 1990]) and blocks-world ([Kambhampati, 1989]). Though these systems have demonstrated the effectiveness of a reuse-based paradigm for improving efficiency of search-intensive tasks, most of them have been tested in the restricted context of an already supplied base analog, or on a few isolated examples, or for "toy" problem domains. Whether the derivational analogy approach would scale up for real-world problems, and how factoring in the cost of retrieving analogs from a sizable episodic memory, as well as the cost of retrieving inappropriate analogs, would affect the system's performance, remain open questions. In this paper, we address the last two issues in the context of a system, APU, that synthesizes UNIX shell scripts from a formal problem specification [Bhansali, 1991, Bhansali and Harandi, 1990, Harandi and Bhansali, 1989]

We describe experiments designed to determine whether automatic detection of appropriate analogies in APU is cost-effective, and whether using derivational analogy does speed-up APU's overall problem-solving performance. Following in the style of Minton [Minton, 1988], we assess APU's performance on a population of real-world problems, generated randomly by fixed procedures. The results of the experiment point to criteria that may be used to determine the viability of using derivational analogy, and also suggest ways of building up a case library.

Overview of APU

The input to APU is a problem specification in the form of pre- and post-conditions, written in a functional, lisp-like notation, augmented with logical and set-theoretic quantifiers. This is transformed by a *hierarchical planner* into a shell-script by a top-down goal decomposition process employing a *knowledge-base* of *rules* and a component library of *subroutines* and *cliches* (or program templates), which represent the primitive operators for the planner. A solved problem is stored together with its derivation history in a *derivation history library.* When a new problem is encountered, an *analogical reasoner* is used to retrieve an analogous problem from the derivation history library, and to synthesize a solution for the new problem by replaying the derivation history of the replayed problem. A potential speedup is achieved by the elimination of search for the right sequence of rules to apply. Details of the program synthesis process are given elsewhere [Bhansali, 1991, Bhansali and Harandi, 1990].

A *concept dictionary* contains the definitions and relationship between various domain-dependent concepts including objects, predicates, and functions. These concepts are organized in an abstraction hierarchy. Rules are written in terms of the most abstract concepts and apply to all instances of the concept in the hierarchy. For example, *lines* and *words* are instances of the abstract concept *text-object*, and *kill* (a process)

and *delete* (a file) are instances of the abstract function *remove*. Thus, e.g., a rule to remove an object would be applicable to both killing a process as well as removing a file. The abstraction hierarchies in the concept dictionary and the polymorphic rules form the basis of the analogical reasoning.

Derivation History

A derivation history of a problem is essentially a tree showing the top-down decomposition of a goal into sub-goals terminating in a UNIX command or subroutine. With each node in the tree the following information is stored:

1. The sub-goal to be achieved. This forms the basis for determining whether the sub-plan below it could be replayed in order to derive a solution for a new problem. The sub-goal is used to derive a set of keys, which are used to index the derivation tree rooted at that node. If these keys match those of a new subproblem, the corresponding sub-tree is considered a plausible candidate for replay.

2. The sub-plan used to solve it (i.e. a pointer to the sub-tree rooted at that node).

3. The rule applied to decompose the problem, and the set of other applicable rules. This is responsible for some speed-up during replay by eliminating the search for an applicable rule.

4. The types of the various arguments. These are used to determine the best analog for a target problem, by comparing them with the types of the corresponding variables in the target problem (next section). If the types of all corresponding variables are identical, it represents a perfect match, and the solution can be copied - a much faster operation than replay.

5. The binding of variables occurring in the goal to the variables in the rule. This is used to establish correspondence between variables in the target and source - expressions bound to the same rule variable are assumed to correspond.

Retrieval of Candidate Analogs

When derivations are stored, they are indexed and retrieved using a set of four heuristics.

Solution Structure Heuristic (H_1). This heuristic is used to detect analogies based on the abstract solution structure of two programs, as determined by the top-level strategies used in decomposing the problem. The top-level strategies usually correspond to the outerlevel constructs in the post-condition of a problem specification. These constructs include the various quantifiers and the logical connectives - not, and, or and implies. As an example an outermost construct of the form:

(¬(EXIST (?x : ...) :ST (and ?constr1 ?constr2)))

is suggestive of a particular strategy for solving the problem: *Find all* ?x *that satisfy the given constraints and delete them.* Therefore, all problems with such a post-condition might be analogous. The other quantifiers and logical connectives result in analogous strategies for writing programs. The *Solution Structure* heuristic creates a table of such abstract keys and uses them to index and retrieve problems.

Systematicity Heuristic (H_2). This heuristic is loosely based on the *systematicity principle* proposed by Gentner [Gentner, 1983] and states that: *if the input and output arguments of two problem specifications are instances of a common abstraction in the abstraction hierarchy, then the two problems are more likely to be analogous.*

The principle is based on the intuition that analogies are about relations, rather than simple features. The target objects do not have to resemble their corresponding base objects, but are placed in correspondence due to corresponding roles in a common relational structure.

To implement this heuristic, APU looks at each of the conjunctive constraints in the postcondition of a problem and forms a *key* for it by abstracting the constants, input-variables, output-variables, and the predicates. The constants and variables in the problem are treated as black-boxes, and unary functions are converted to a uniform binary predicate, since they are deemed unimportant, and the primary interest is in detecting the higher order abstract relations that hold between these objects. This is done by replacing a function or predicate by climbing one step up the abstraction hierarchy. The detection of these higher order relations also establishes the correspondences between the input/output variables of the source and target problem, which is used to prune the set of plausible candidates (using the *Argument Abstraction heuristic*, discussed shortly) as well as in replay.

As an example, the constraints

(≥ (cpu-time ?p) ?t), and
(< (?n (size ?f)))

where $?p, ?f$ are input variables and $?t, ?n$ are output variables would result in the formation of the following key:

(order-rel-op (attribute Constt inp-var) out-var)

suggesting that the two problems might be analogous.

Syntactic Structure Heuristic (H_3). The third heuristic relies on certain syntactic features of a problem definition to detect analogies. The syntactic features include the structure of a specification (e.g. if two problems include a function definition that is defined recursively), as well as certain keywords in the specification (e.g. the keyword :WHEN in a specification indicates that the program is asynchronous, involving a periodic wait or suspension of a process). APU creates a table of such syntactic features which are used to index and retrieve problems.

Argument Abstraction Heuristic (H_4). This heuristic uses the abstraction hierarchy of objects to

determine how "close" the corresponding objects in two problem specifications are. Closeness is measured in terms of the number of links separating the objects in the concept dictionary - shorter the link, the better are the chances that the two problems will have analogous solutions. For example, *lines* and *words* are closer to each other than, say, to a *process*. Therefore, the problem of counting lines in a file is closer to the problem of counting words in a file than to the problem of counting processes in the system.

Interaction of Heuristics The retrieval algorithm resolves conflicts between candidate choices suggested by the above heuristics by using H_2, H_1, and H_3, as the primary, secondary, and ternary indices respectively. Further ties are broken by using H_4, and then choosing one of the candidates at random [Bhansali, 1991].

Experimental Results

In order to address the issues mentioned in the introduction, we need experimental results that provide answers to the following aspects of APU's retrieval and replay techniques:

- How good are the heuristics in determining appropriate base analogs?
- How does the time taken to synthesize programs using analogy compare with the time taken to synthesize programs without analogy?
- How does the retrieval time depend on the size of the derivation history library?

As mentioned earlier, it is not enough to show results on isolated examples; the system must be tested on a population of problems that is representative of real-world problems. However, the limited knowledge-base of our prototype system, precluded testing on a truly representative sample of the space of UNIX programs. Therefore, we decided to restrict ourselves to a subset of the problem domain, consisting of file and process manipulation programs; problems were constructed randomly from this subset by fixed procedures.

Generating the Data Set

We began by constructing a rule-base for 8 problems that are typical of the kind of problems solved using shell scripts in this problem space. The problems included in the set were: *1) List all descendant files of a directory, 2) Find most/least frequent word in a file, 3) Count all files, satisfying certain constraints, in a directory, 4) List duplicate files under a directory, 5) Generate an index for a manuscript, 6) Delete processes with certain characteristics, 7) Search for certain words in a file,* and *8) List all the ancestors of a file.*

To generate the sample set, we first created a list of the various high-level operations which can be used to describe the top-level functionality of each of the above problem - *count, list, delete,* etc. - and a list of objects which could occur as arguments to the above operations - *directory, file, system, line, word,* etc. Then we created another list of the predicates and functions in our concept dictionary which relate these objects, e.g., *occurs, owned, descendant, size, cpu-time, word-length, line-number,* etc.

Next, we used the definitions of the top-level predicates in the concept dictionary to generate all legal combinations of operations and argument types. For example, for the *count* operation, the following instances were generated: *(count file system), (count file directory), (count word file), (count character file), (count line file), (count string file), (count process system).*

In a similar fashion, a list of all legal constraints were generated, using the second list of predicates and functions. Examples of constraints generated are *(occurs file directory), (descendant directory directory), (= int (line-number word file)),* and *(= string (owner file)).* Constraints that were trivial or uninteresting were pruned away, e.g. *(= int int).*

Next we combined these constraints with the top-level operations to create a base set of problems. We restricted each problem to have a maximum of three conjunctive constraints. From this set a random number generator was used to select 37 problems, which together with the initial set formed our sample population.

The final step consisted of translating the high level description of the problems into a formal specification. This was done manually, in a fairly mechanical manner. The only non-mechanical step was in assigning the input and output arguments for each program. This was done using the most 'natural' or likely formulation of the problem.

Experiment 1: Feasibility of Automatic Retrieval

We stored 15 randomly chosen problems from the sample set in the derivation history library. Then, for each of the 45 problems, we ran the retrieval algorithm to determine the best base analog. This was done for various combinations of the heuristics.

To evaluate the heuristics, we compared APU's choice against a human expert's, namely ourselves. To ensure that our choices are not biased by APU's, we compiled our own list of the best analogs for each problem, *before* running APU's retrieval algorithm.

The result of the experiment is summarized in Figure 1. The first column shows which heuristics were turned on during the experiment. The combinations tried were - all heuristics working, all but one heuristic working, and each heuristic working separately[1]

[1]The argument abstraction heuristic cannot be used independently, since it is not used to index problems, but simply to prune the set of candidates retrieved by the other analogs.

Heuristics used	Mismatch	Inferior solns.	Score
All	8	2	43
H_1, H_2, H_3	9	3	42
H_1, H_2, H_4	8	2	43
H_1, H_3, H_4	9	7	38
H_2, H_3, H_4	11	5	40
H_1	13	11	34
H_2	14	8	37
H_3	41	41	4

Figure 1: Performance of APU's retrieval heuristics against a human expert's.

The second column shows the number of problems for which APU's choice did not match ours. However, it would not be fair to judge APU's performance simply on the number of mismatches, since that would imply that the human choices are always the best. Since we could not be confident of the latter, after obtaining the mismatches, we again carefully compared APU's choice against ours to judge their respective merits. We discovered that in a few instances APU's choices were clearly inferior to ours, while in others, it was not clear which of the mismatched choice was better. The former were marked as inferior choices (column 3), and an overall score for each heuristic combination, was determined by subtracting the number of the inferior choices from the total number of problems (column 4).

Discussion. The experiment clearly indicates that using all 4 heuristics, APU's retrieval algorithm performed almost as well as a human. There were only two cases in which APU's choice of an analog was clearly inferior. The reason why APU failed to pick the correct choice for the two cases became obvious, when we looked at the two cases.

Consider the first case, which was to delete all directories that are descendants of a particular subdirectory. This was specified using the post-condition

(¬ (EXIST (?sd: dir) :ST (descendant ?sd ?d)))

where *?d* is an input directory-variable. The best analog for this problem was the problem of listing all the descendant sub-directories of a directory, since both of them involve recursive traversal of the directory structure in UNIX. However, the analog picked by APU was the problem of deleting all files under a given directory, specified with the post-condition:

(¬(EXIST (?f: file) :ST (occurs ?f ?d)))

where *?d* is again an input directory-variable. The reason APU picked this analogy was because *occurs* and *descendant* are groupted under a common abstraction *contained* in APU's concept dictionary. Thus, the systematicity heuristic abstracted both *occurs* and *descendant* to *(contained OUTPUT-VAR INPUT-VAR)*, and considered both to be equally good analogs for the target; the solution-structure heuristic then picked the *delete-files* problem because its outerlevel constructs were closer to the target's.

At a more abstract level, APU's inability to pick the right analog can be explained by the fact that APU's estimation of the closeness of two problems in the implementation domain is based solely on its assessment of the closeness of the two problems in the specification domain. A better organization of the concept dictionary, so that the distance between concepts in the specification domain reflects the corresponding distance in the implementation domain, might avoid some of these missed analogies.

The experiment also shows that H_1 and H_2 are the two most important heuristics - as expected. Rows 4 and 5 show the number of missed analogs when one of the two is turned off. Though the table doesn't show it, the problems for which the analogies were missed were also different, indicating that neither heuristic is redundant.

The result in Row 2 was unexpected, since it seems to indicate that the argument abstraction heuristic is unimportant. This was contrary to our experience when we tried the heuristics on isolated examples. In particular, when the base set of analogs contained several similar analogs, the argument abstraction heuristic was important to select the closest one. The reason we got this result is because of the small set of base analogs - there weren't two analogs sufficiently close as to be indistinguishable without using the argument abstraction heuristic.

Finally, H_3 doesn't seem to contribute much to the effectiveness of the retrieval. This is again due to the nature of the sample space, where most problem descriptions did not have syntactic cues like keywords and recursion.

We remark, that although the heuristics are specialized for the domain of synthesizing UNIX programs, we believe that the general strategy of using the abstract solution structure (H_1), the problem formulation (H_2 and H_3), and generic objects (H_4) should be adaptable to other domains.

Experiment 2: Speed-up using Derivational Analogy

For this experiment, we selected 10 examples at random from our sample set to form the set of source analogs. From the same sample set, we selected another set of 10 examples (again using a random number generator) and measured the times taken to synthesize a program for each of them, once with the analogical reasoner off, and once with the analogical reasoner turned on. This was repeated with 20 different sets of base analogs.

Figure 2 shows the result of one typical run and figure 3 shows the speed-up achieved on the first 5 runs.

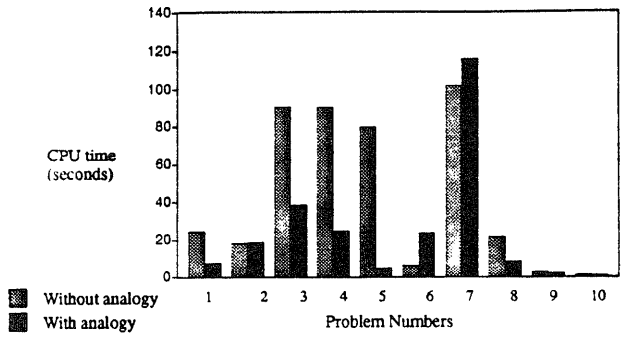

Figure 2: Sample data showing the speedup of program synthesis using derivational analogy

Expt. No.	Time without analogy: t_1	Time using analogy: t_2	Speedup (t_2/t_1)
1	295.69	133.69	0.45
2	514.13	284.03	0.55
3	470.80	256.87	0.54
4	437.17	246.33	0.56
5	488.27	257.09	0.53

Figure 3: Speed-up obtained on 5 different sets of source analogs

Discussion. Figure 3 show that using derivational analogy, the average time to synthesize programs is reduced by half. This is not as great as we had expected based on our experience on isolated examples. Nevertheless, the result is significant, because it demonstrates that derivational analogy is an effective technique in improving problem-solving performance, not only on isolated examples, but on populations of problems too.

There are several factors that affect the generality of these results. First, it is based on the assumption that problems are drawn from the set of basic concepts and constraints with a uniform distribution. However, in practice, we expect a small set of concepts and constraints to account for a large share of the problems encountered in real life. In that case, with a judicious choice of which problems to store in the derivation history library, the number of problems for which close analogs can be found will be much larger than the number of problems without analogs. Consequently, the benefits of using derivational analogy would increase.

There are two potential ways in which replay can improve problem-solving performance. One is by avoidance of failure paths encountered by a planner during initial plan-synthesis. This is the primary source in domains where there are a few general purpose rules and the planner has to search considerably in order to find a solution. The other (less substantial) speed-up is obtained by identifying solution spaces which are identical to a previously encountered one. In such situations, instead of traversing the solution space, the problem-solver can copy the solution directly. This is the primary source of speed-up in systems like APU, where there are a large number of focussed rules. The planner does not spend much time in backtracking, however there are common sub-problems that arise frequently. (Again, we expect the 80-20 rule to apply in most domains, i.e. 80% of the problems would be generated from 20% of the solution space, implying that there would be a large number of problems that are analogous, compared to non-analogous ones.)

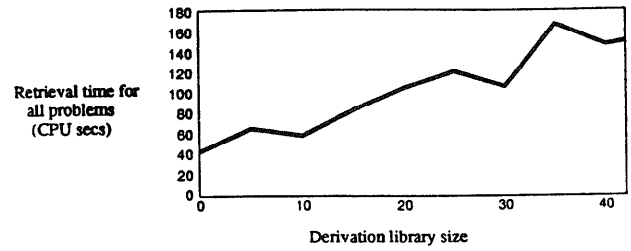

Figure 4: The time taken to retrieve analogs as a function of library size.

A promising observation from figure 2 is that, when target problems do not match analogs in the library, the degradation in performance is small (problems 2, 6, 7), compared to the improvement in performance when problems match (problems 4,5). This suggests that unless the number of mismatches is much larger than the number of matches, derivational analogy would improve problem-solving.

Also, the speedup obtained for larger problems is generally greater than speedup for smaller problems. This could be used to decide when it would be advantageous to use replay, based on an estimation of the size of a user specified problem.

Experiment 3: Retrieval Time

Our third experiment was designed to measure the cost of retrieving analogs as a function of the size of the derivation history library. To measure this, we incrementally increased the size of the derivation history library, (in steps of 5), and measured the time taken to retrieve analogs for all the 45 problems. Figure 4 shows the result of one typical run of this experiment.

Discussion. The figure shows that the retrieval time increases almost linearly with the number of problems in the library. The time taken to search for analogs essentially depends on the average number of problems indexed on each feature. For the retrieval time to converge, the average number of problems per feature

should approach a constant value. For our sample set, we found that this was not true. The average number of problems per feature after each set of 5 problems were added to the library were: 1.88, 1.85, 2.11, 2.5, 2.2, 3.58, 3.2, and 3.28, which corresponds remarkably well with (the peaks and valleys in) the graph.

This provides a clue as to when problems should be stored in the derivation history library: if adding a set of problems to the library increases the ratio problems/feature, it suggests that the new problems are quite similar to the problems already existing in the library, and hence their utility would be low. On the other hand, if the ratio decreases or remains the same, the problems are different from the ones in the library and should probably be added.

Finally, the figure shows that the retrieval time itself is not much - less than 4 seconds on an average - compared to the time to synthesize programs.

Conclusion

The significance of the experiment reported here lies in elucidating some of the criteria that determine the viability of using derivational analogy for improving problem-solving behavior - specifically, when the cost of retrieving the right source analogs and the cost of applying incorrect analogs (called *globally divergent* situations [Veloso and Carbonell, 1991]) is factored in. To summarize, the derivational analogy approach is likely to be cost-effective if

1) The distribution of problems in a domain is such that a large proportion of the problems are analogous compared to non-analogous ones .

2) It is possible to abstract features of a problem-specification, so that the distance between problems in the abstracted feature space reflects the distance between problems in the implementation space.

3) The ratio problems/feature of problems in the case library converges.

4) The average complexity of solution derivation is large compared to the overhead of analogical detection.

The last two conditions also suggest when a solution derivation should be incorporated in a case library. This, in conjunction with an empirical analysis [Minton, 1988] of the utility of stored derivations can be used to maintain a case library that would maximize the benefit/cost ratio of using the derivation history library.

References

[Baxter, 1990] Ira D. Baxter. *Transformational Maintenance by reuse of design histories.* PhD thesis, University of California, Irvine, December 1990.

[Bhansali and Harandi, 1990] Sanjay Bhansali and Mehdi T. Harandi. Apu: Automating UNIX programming. In *Tools for Artificial Intelligence 90*, pages 410–416, Washington, D.C., November 1990.

[Bhansali, 1991] Sanjay Bhansali. *Domain-based program synthesis using planning and derivational analogy.* PhD thesis, University of Illinois at Urbana-Champaign, 1991. (Forthcoming).

[Blumenthal, 1990] Brad Blumenthal. Empirical comparisons of some design replay algorithms. In *Proceedings Eighth National Conference on Artificial Intelligence*, pages 902–907, Boston, August 1990. AAAI.

[Carbonell and Veloso, 1988] Jaime Carbonell and Manuela Veloso. Integrating derivational analogy into a general problem solving architecture. In *Proceedings Case-based Reasoning Workshop*, pages 104–124, Clearwater Beach,Florida, May 1988.

[Carbonell, 1983] Jaime G. Carbonell. Derivational analogy and its role in problem solving. In AAAI, pages 64–69, 1983.

[Gentner, 1983] D. Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.

[Goldberg, 1989] Allen Goldberg. Reusing software developments. Technical report, Kestrel Institute, Palo Alto, California, 1989. Draft.

[Harandi and Bhansali, 1989] Mehdi T. Harandi and Sanjay Bhansali. Program derivation using analogy. In IJCAI, pages 389–394, Detroit, August 1989.

[Huhns and Acosta, 1987] M.N. Huhns and R.D. Acosta. Argo: An analogical reasoning system for solving design problems. Technical Report AI/CAD-092-87, MCC, Microelectronics and Computer Technology Corporation, Austin, TX, 1987.

[Kambhampati, 1989] S. Kambhampati. *Flexible reuse and modification in hierarchical planning: a validation structure based approach.* PhD thesis, University of Maryland, College Park, October 1989.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: an explanation-based approach.* PhD thesis, Carnegie Mellon University, March 1988.

[Mostow and Fischer, 1989] Jack Mostow and Greg Fischer. Replaying transformationals of heuristic search algorithms in DIOGENES. In *Proc. of the* DARPA *case-based reasoning workshop*, Peniscola, Florida, May 1989.

[Mostow *et al.*, 1989] Jack Mostow, Michael Barley, and Timothy Weinrich. Automated reuse of design plans. *International Journal for Artificial Intelligence and Engineering*, 4(4):181–196, October 1989.

[Steier, 1987] David Steier. CYPRESS-Soar: a case study in search and learning in algorithm design. In *IJCAI*, pages 327–330, August 1987.

[Steinberg and Mitchell, 1985] L. I. Steinberg and T. M. Mitchell. The redesign system: a knowledge-based approach to VLSI CAD. IEEE *Design Test*, 2:45–54, 1985.

[Veloso and Carbonell, 1991] Manuela Veloso and Jaime G. Carbonell. Learning by analogical replay in PRODIGY: first results. In *Proceedings of the European Working Session on Learning.* Springer-verlag, March 1991.

[Wile, 1983] D. S. Wile. Program developments: formal explanations of implementations. *Communications of the* ACM, 26(11):902–911, 1983.