Program Derivation Using Analogy

Mehdi T. Harandi
Dept. of Computer Science
Univ. of Illinois at Urbana-Champaign

Sanjay Bhansali
Dept. of Computer Science
Univ. of Illinois at Urbana-Champaign

Abstract

We present a methodology for using analogy to derive programs based on a derivational transformation method. The derived programs are deductively closed under the rules in the knowledge base, and the emphasis is on speeding up the derivation of a solution. We describe certain heuristics to find a good source analogue to the target problem efficiently, show how the derivation trace of that program can be used to guide the derivation of the new program.

1 Introduction

Analogical reasoning is a mechanism for using past experience in solving a problem to solve a new, similar problem. Almost all the systems that perform analogical reasoning fall under one of two broad categories. The first category of systems use analogy to solve a problem for which the domain theory is incomplete [Winston, 1980, Falkenhainer *et al*., 1986, Greiner, 1988]. In such systems it would not be possible to solve the problem by a deductive mechanism, using only the known facts about the domain, and it is necessary to fill in the missing parts by analogy from a different situation or domain. The other category of systems use analogy to speed up the derivation of a solution. The final solution is deductively closed under the given domain theory, i.e. even without analogy, it would have been possible to arrive at the solution. Such systems have mostly been used for theorem-proving and program construction [Kling, 1971, Dershowitz, 1983] though there are exceptions [Davies and Russel, 1987].

1.1 Unix Programming by Analogy

The work presented here falls into the second category. It uses analogy to derive a program using past experience in solving a similar problem. The domain of programming is the Unix operating system environment. Unix programming is very similar to conventional programming. Its piping, sequencing, input-output redirection, shell control constructs, etc. are similar to the control structures available in high level programming languages. However, Unix has a much richer set of primitive commands. These can be considered similar to a library of standard subroutines in conventional programming. Thus one works with a richer set of building blocks than is available in general programming.

1.2 Problem Specification

The system described here is designed to work on top of an automatic programming system which takes a semi-formal specification of a problem and produces the correct sequence of Unix commands or a shell script to solve the problem. A problem specification in our system has the following form:

```
INPUT: Primitive-objects
  WHERE:<type-spec>  SUCH-THAT:Constraints>
OUTPUT: Primitive-fn(arguments)
  WHERE:<type-spec>  SUCH-THAT:<constraints>
EFFECTS: Priraitive-fn(arguments)
  WHERE:<type-spec>  SUCH-THAT:<constraints>
```

Here, INPUT and OUTPUT specify the input and outputs of a program, and EFFECT specifies any side-efllfects that a program may have. The WHERE slot describes the domains for the various fields, and SUCH-THAT specifies the constraints on the field values. For example, to specify a program that prints the name of all files greater than 10K and the owner of the file, the problem specification would be:

```
INPUT: dir WHERE: (Directory dir)
OUTPUT: (List f u)
  SUCH-THAT: (and (Belongs f dir)
                  (> (Size 1) 10K)
                  (Owner u f)
  WHERE: (and (File f) (User u))
```

In the above examples, *Belongs, Owner,* etc. are pre-defined predicates/functions and *files, directories, users,* etc. are pre-defined objects known to the system. In addition, it is possible for a user to define new relations, functions and objects and use them in a problem specification. For example a user may define an ANCESTOR function as :

```
#DEFINE ancestor
 INPUT: f SUCH-THAT: (<> 1 //) WHERE: (File f)
 OUTPUT: (LIST d)
  SUCH-THAT:(= d (Union(Parent d)
                        (Ancestor(Parent d))))
  WHERE: (Directory d)
```

1.3 The Rulebase

The system has three different levels of rules, each representing a different degree of generality. These rules are used to generate a Unix program for a given specification. The steps taken in decomposing a problem and the set of rules applied in the process form the basis for solving analogical problems.

At the highest level the system has rides that encode domain-independent *high level strategies* for problem solving. These rules are encoded in the form of stereo-typed templates of shell scripts, similar to the concept of programming cliches in KBEmacs [Waters, 1981], with the appropriate commands filled in for solving the particular problem. Examples of such strategies are recursively-solve, divide-and-conquer, and loop-over-objects.

The second level of rules are related to solving problems in the operating system domain. These rules encode common sense reasoning about problem solving, e.g. *To count the number of objects of type A, map the objects of type A into objects of type B, count the number of objects of type B, and apply the inverse mapping relation to the output of count.*

The third category of rules are those that are specific to the Unix operating system. This is the largest part of the rule base, and forms an index to the commands available in Unix. An example of such a rule would be: *To list all sub-objects of a directory use command* ls.

2 Finding Analogues

Before a system can apply an analogy, it has to discover a source analogue. Unlike some systems, we do not assume that the analogues to be matched are specified by the user. Therefore, the system has to find a source analogue given the target problem. To do this task it has some heuristic knowledge built in.

To get an intuitive feel for the kind of knowledge required to find analogue matches, we consider an example. Suppose we want a program that removes all the files bigger than 10K. Suppose we have already solved a problem to delete all processes with cpu times greater than 1000 sec. A program to solve this would be: *form a list of all the processes, select all processes whose cpu time is greater than 1000 sec, retrieve their process-id's and kill the processes.* Seeing this solution, we can get an insight on how to solve the problem of deleting all files bigger than 10 K : *Form a list of all the files, select those whose size is greater than 10 K, retrieve the file names and remove the files.*

Both the above problems involve deleting objects and comparing numbers. But this is not the main reason that the analogy worked. To see this, consider another problem of changing the names of all files that do not have write access by appending a suffix *.read* to them. This problem has nothing to do with deleting objects or comparing numbers, but the same program structure can be used to solve this problem: *Form a list of all files, select those files that do not have write access, retrieve their names, say* name , *and change it to* name.read .

The reason the analogy worked in both the cases is that they arc both instances of the search-and-process paradigm. They involve a search for a particular object among several similar objects and then some processing on a particular attribute of that object. Abstractly, each of the above problem can be stated as:

Apply C(x) where x 6 D such that P(x)

where *C* is an abstract function, *D* is the type of the input and *P* is the predicate characterising those *x* on which the operator *C* is to be applied.

The success of the analogical reasoning system is, therefore, contingent on detecting the occurrence of such paradigms from the problem specification. At the same time, note that Problem 2 seems to be "more" analogous to Problem 1 than to Problem 3. Thus, given both Problems 1 and 3 in the knowledge base, to solve Problem 2, we would like Problem 1 to be retrieved rather than Problem 3. Thus, the analogue matcher should return not just a possible source analogue, but the best possible source analogue in the knowledge base. The next section gives some heuristics on how this can be done.

2.1 Heuristics to Detect Analogies from Problem Specifications

2.1.1 Functional abstraction heuristic

If two problems involve instantiations of the same abstract function, then it might be possible to replace the program fragment corresponding to that function in the source program by an analogous program fragment in the target program.

The first example above illustrates this. Both deleting a file and killing a process are instantiations of the abstract function *Remove* . If we know the program fragment for finding the object, we can append the code corresponding to deleting a file or killing a process to get one or the other program.

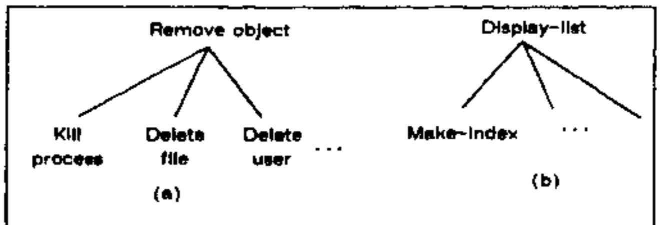


Fig. 1 Examples of functional abstraction hierarchies

Such analogies can be detected by building an abstraction hierarchy for the various functions as illustrated in Fig. la. Note, that most of the functionality provided by Unix has already been captured in our specification language. For example, instead of specifying kill process or remove directory the user would simply say:

EFFECT:(Remove /foo) WHERE:(Directory /foo)
or, EFFECT-.(Remove pr) WHERE: (Process pr)

However, one feature of Unix (and also our system) is that it allows a user to invent new commands and hence, not all the functions can be captured by our specification language. But the system can still build the abstraction hierarchy dynamically from the specification of the problem. Thus, suppose a user creates a new command *make-index* defined as :

```
INPUT: foo  WHERE: (File foo)
OUTPUT:(List w n) SUCH-THAT:...
 WHERE:(and (Word w)(Int n))
```

then, the command gets automatically incorporated into the abstraction tree shown in Fig. 1(b), since the output of the command is to display a list. So, when we get a new problem *make-table* , defined as

```
INPUT:  foo  WHERE: (File foo)
OUTPUT:(List a b c) WHERE:... SUCH-THAT:...
```

we can immediately access the *make-index* command, and if there is some similarity, use that command to generate a program to create the table.

2.1.2 Same higher order relationship between arguments or Systematicity heuristic

This heuristic is based on the systematicity principle proposed by Gentner [Gentner, 1983]. Slightly abusing the definition, we can say that if the input and output arguments of two problem specifications are related by the same abstract relationship, then the two problems are more likely to be analogous.

Again, the first example in section 4.1 provides an illustration of the application of this heuristic. It is instructive to look at the formal specification of the two problems:

```
PI.-INPUT: d WHERE: (Directory d)
   EFFECT:(Remove f)
    WHERE:(File f)
    SUCH-THAT:(and (Belongs f d)
                   (> (size f) 10))
P2:INPUT: u WHERE: (User u)
    EFFECT: (Remove p)
     WHERE: (Process p)
     SUCH-THAT: (and (Belongs p u)
                     (< (cpu-time p) 15))
```

In this example, the two arguments of each problem are related by the same relation *Belongs* . But more interesting is the second constraint, which says that a unary function of the output argument of each problem is related by the abstract relation *comparison* (instantiated to > and < respectively). If we view each unary function as an abstract relation called Attribute with two arguments - the name of the attribute and the original argument - then, the above constraints *((> (Attribute(f, size), I0)and(< Aitribute(p, cpu - time), 15))* can be seen as instances of the following second-order relation [1]*:*

Rel2 (Rell (argl, propertyl), Re IO)

Therefore, according to the systematicity heuristic the two problems may be analogous.

To detect analogous problems based on the systematicity heuristic, the system would have to determine the order of each relation in the problem specification and then match the highest order relation from the two problems. If they belong to the same abstract relation then the corresponding arguments of the relation have to be matched recursively until the zero-order relations are matched.

[1]The order of a relation is defined as follows: Constants are order 0. The order of a predicate is 1 plus the maximum of the order of its arguments.

2.1.3 Similar syntactic structure heuristic

For those functions that are defined dynamically by the user, there is another clue which can suggest that two problems might have analogous solution: the structural similarity of the function definition. Thus, if both functions are recursive then their solutions would be very similar. For example a program to find ancestors defined in section 1.2 might consist of a shell file called ancestor. When this file is called with an argument, the shell script checks that the argument is not '/' (end-of-recursion test), prints the parent of the argument and invokes the same file again with the parent as the argument. Having solved this, it is easy to determine a program for descendants defined below:

```
Descendants:-
 INPUT: (List dir) WHERE: (Directory dir)
 OUTPUT: (List t)
  SUCH-THAT:(= f (Union(Sub-objects dir)
                    (Descendant
                      (Sub-objects dir))))
  WHERE:(File-objects f)
```

The program would involve creating a similar shell file, that checks that its argument list is non-null and then for each argument it prints the sub-objects under it, and calls the file recursively with the sub-object as the argument.

It may be remarked that problems having a similar definition structure can generally be solved by using *strategy* rules, since both are inherently syntactic in nature.

2.1.4 Argument abstraction heuristic

As in the case of functions, we can envision an abstraction hierarchy for certain objects which appear as arguments in the problem specification. In some cases if two problems have arguments that belong in the same abstraction hierarchy, the two problems may be analogous. For example, *count number of paragraphs, count number of lines* and *count number of words* are analogous. All involve finding a way of recognizing a text-object by finding its terminator (white space for a word, a newline for a line, a blank line for a paragraph), mapping them to a countable object and counting that (using the rule given in section 1.3). However, in other cases it may not work, e.g. a page is not recognized by a delimiter but in terms of the number of lines (typically 24).

2.2 How the Heuristics Interact with each other

In general each of the above heuristic will suggest several, and possibly different, problems as a potential analogue of the target problem. Therefore the matcher needs to decide what importance should be attached to each suggested alternative, and in what order they should be tried.

The most important heuristic seems to be the systematicity heuristic, and if there are any problems supported by that heuristic the matcher tries them first. Next in importance are the definitional structure and the functional abstraction heuristic. In case of ties, the problem with the maximum number of support from the lower heuristic is used to decide which problem should be tried. The fourth heuristic is not really strong enough to be used as a basis for deciding on a particular problem as the source analog, and is used only as a support for strengthening the candidature of a problem suggested by the other heuristics.

3 Derivational Analogy

3.1 Problems with Direct Solution Transformation

Most analogy systems are based on a solution transformation method [Kling, 1971, Dershowitz, 1983, Carbonell, 1983a]. In this method the system retrieves the solution of a problem which is similar to the current problem, and perturbs the solution, guided by some heuristics, until it satisfies the requirements of the new problem. Such systems ignore the reasoning behind the various steps used in deriving the original solution. Though this method works in some domains, it is inadequate in program construction, and even when they work they often result in inefficient and inelegant solutions.

As a simplistic example, suppose we have program which counts the number of occurrences of each word in a file. This program first forms a list of all the words occurring in the file and then for each word finds how many times it occurs in the file. The list of all words in the file is formed by writing each word of the file in a separate line and then deleting duplicate lines.

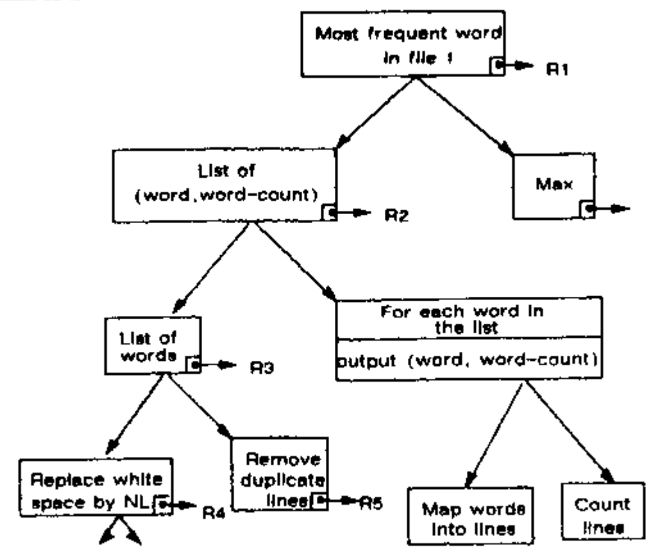
Now, suppose we want to derive a program to count the number of occurrences of each character in a file using the above. Using a solution transformation method on the final program, one would produce a program that finds a list of all characters in the file by listing each character on a separate line and deleting duplicate lines, which is extremely inefficient, since there are only about a 100 or so characters in a text file, which are known a priori by any programmer.

We use a derivational transformation method based on [Carbonell, 1983b]. Such a method avoids the kind of inefficiency encountered above, because it records the reasons for each step in the program derivation. Therefore, it can figure out that the reason for writing each word in a separate line and then deleting duplicate lines is to get a list of unique words in the file; and, since it knows of a direct method of listing all possible characters in a file, it can generate a more efficient program.

3.2 Derivation Trace

A derivation trace consists of the subgoal structure of the problem, a pointer to each rule used in decomposing the problem at each node of the tree and the final solution. Fig. 2a shows the derivation tree for a program to find the most frequent word in a file.

Fig. 2b and 2c show some of the rules in English used to derive the sub-tree below each node, and the Unix program for the problem.



Most frequent word in file 1
R1
List of (word,word-count)
R2
Max
List of words
R3
For each word in the list
output (word, word-count)
Replace white space by NL
R4
Remove duplicate lines
R5
Map words into lines
Count lines

Fig 2a. Partial derivation tree for a program to find the most frequent word in a file.

R1 : To find the most frequent object, form a list of all the objects and the object-count and take the maximum from the list.

R2 : To form a list of all objects and their counts, form a list of unique objects and for each object in the list find its count and output the object and the count.

R3 : To form a unique list of objects, form a list of all the objects and remove duplicates.

R4 : To form a list of text-objects in a text stream, replace the object-terminator by a newline character.

Fig 2b. Sample rules used in the derivation in fig. 2a.

```
cat file1 | tr ' ' '\012' | tr '      ' '\012' > /tmp/f1
 FOR word in 'sort /tmp/f1 | uniq'
 DO
   count = 'grep $word /tmp/f1 | wc -l'
   echo $word $count
 DONE
 <Code to find maximum>
```

Fig 2c. Unix program to compute a list of words and the number of times the word occurs in the file.

3.3 The Derivational Transformation Method

The derivational transformation method can best be understood by going through the derivation of an example. The source program is the same as shown in Fig. 2. The target problem is:

Find the most frequent file name in a system.

Assume that the system has selected the problem in Fig. 2 as the best possible analogue to this problem, based on the heuristics given in section 4. The first step in generating a solution to this problem is to see if there is a direct Unix command to do this. The rules pertaining to Unix commands are stored separately from the other rules and are always searched first. Since there isn't a direct command to solve the problem, the system tries to generate a program by an analogue transformation of a previous solution.

The first rule used in Fig. 2a is rule R1. This rule is applicable to the target problem with *object* being instantiated to file-name. So the system uses the same rule to get the top-level decomposition of the problem.

Next, the system goes down the left branch of both the source and target trees, and repeats the process, till it comes to a node where the rule is not applicable. This happens when it tries to apply rule R4, which is applicable only for text-objects in a text-stream. At this stage, the system reverts back to its usual problem solving mode and tries to solve the problem by first principles. (Eventually, it will discover the *ls* -r command which lists all objects under a directory and the *grep* command to remove extraneous information about sub-directories, links, etc.)

The other nodes of the sub-tree referring to removal of duplicate lines, mapping the file-names into lines and counting lines, and taking the maximum from the list remain valid for this problem, and hence, that part of the program remains the same.

The final program resulting from this method is shown in Fig. 3.

```
<Code to get list of flle~name9>
FOR word ln 'sort /tmp/fl | unlq'
DO
   count = 'grep Sword /tmp/fl | we -l'
   echo Sword $count
DONE
<Code to find maximum>
```

Fig 3. Unix program to find the most frequent file-name under a directory.

Note, that at each step the system checks for a direct method of solving the problem; thus, it can take advantage of a more efficient method of solving the problem, if it exists.

3.4 Algorithm

We now give the an outline of the algorithms used for deriving a program by analogy. The general algorithm for solving a problem is given first.

```
SOLVE(P)
1. Search the command-library to see if P can be solved
   directly by using a Unix command. If so, then return that
   command as the solution for P, else continue.
2. Find P', the best possible analogous problem to P, using
   the analogue-matching heuristics. If such a problem cannot
   be found go to step 4.
3. Call the ANALOGY algorithm on P and P' If it returns a
   solution, return that solution, else go to step 4.
4. Using the rules in the knowledge-base, decompose the problem
   into sub-problems. In general this will return a set D, each
   element of which represents a way of decomposing the problem.
   While D is non-empty repeat :
      Choose any decomposition d E D and delete it from D.
      Let d = (p1,P2,—,Pn) , where each p, represents a
      sub-problem of P. For each p_i, call SOLVE(p_i), and let
      S_i denote the solution returned. If none of the S_i is
      FAIL, and the all the S_i's are compatible (see notes
      below), then return the sequence (S_1, s_2,...,S_n) as the
      solution for P.
5. Return FAIL.
```

In step 4 above, after each of the sub-problem is solved, it is still necessary to check that the solutions do not interact destructively, and invalidate the solution for P. Therefore, after a potential solution is obtained, it is passed to a plan analyzer, which checks that the subgoals achieved by each sub-solution is not destroyed by the other sub-solutions. If necessary, the plan analyzer re-orders the solutions. However, if it is not possible to achieve the goal even after re-ordering, it reports failure, and the algorithm has to consider some other decomposition scheme and/or backtrack.

Before presenting the ANALOGY algorithm we define two terms. Two problems are said to be *directly analogous* if the same rule R applies with the same instantiation of parameters for the two problems. They are *possibly-analogous* if the same rule R is applicable to both the problems with a different instantiation of the parameters.

```
ANALOGY(P,P')
1. Compare P and P \
   (a) If they are DIRECTLY-ANALOGOUS perform the analogical
       substitutions in S, the program fragment correponding
       to P , to get S' the program fragment corresponding to P '
   (b) If they are POSSIBLY-ANALOGOUS go to step 2.
   (c) Return FAIL.
2. Let R be the rule used to decompose P into sub-problems
   p1,P2,.-.,p_n • Use the same rule to decompose P' into
   sub-problems p1,p'2 ...,Pn'-
3. For each sub-problem p_i' do :
   Check the command-library to see if it can be solved directly
   using a Unix command. If so, record that as the solution, s_i .
   Else, let s_i = ANALOGY (p_i-,Pi'). If* = FAIL, let si zr
   SOLVER'). If SOLVER) also returns FAIL, then return FAIL.
4. Check that the solution set (s_1, s_2, •*> S_n) is compatible.
   If so, return it, else return FAIL.
```

Note that the algorithm checks whether the sub-problems obtained after decomposition can be solved directly using a Unix command (step 3). Hence, if a more efficient solution exists for the problem, the system would discover it.

Thus, the system integrates traditional problem-solving methodology with analogical reasoning, without sacrificing the possibility of finding a better solution for a problem.

4 When to use Analogy?

As has been mentioned, the solutions obtained by the analogical process could have been obtained deductively from the rules in the knowledge base without using analogy. Therefore, a natural question to ask at this stage is: under what conditions should one try to use analogy to solve the problem?

Let $t_A(P)$ be the time to solve a problem P using purely analogical reasoning and $t_D(P)$ be the time taken to solve the same problem using direct problem-solving methods. For analogical reasoning to be justified we want that:

$$t_A(P) \ll t_D(P)$$

The time taken to solve a problem using analogy depends on the time taken to find the analogy and the number of rules used in the derivation of the solution. Thus, assuming that all rule applications take the same amount of time, we may say:

$$t_A(P) \propto t_m + Cn$$

where t_m is the time to find an analogue source problem, n is the number of rules used to solve the problem, and C is a constant. Solving a problem using direct problem-solving techniques requires a search through the knowledge base of rules. If k is the average number of rules searched for each rule that is finally applied then

$$t_D(P) \propto k^n$$

Therefore, for such problems the use of analogy is justified if the sum of the time to find analogs and the time to apply the rules is much smaller than the time to search the knowledge base for the rules.

However is most of the cases, analogy alone is insufficient to solve the problem. Let p be the average proportion of rule that are matched directly using analogy and $q = 1 - p$ be the average proportion of rules that are obtained through a search of the knowledge base. The time to solve a problem using analogy is now given by:

$$t_A(P) \propto t_m + C(pn + k^{qn})$$

For analogy to be justified,

$$t'_m + pn \ll k^n - k^n q$$

where $t'_m = t_m/C$.

The above equation suggests that analogy will be useful if:

1. The time to find analogue matches is small.
2. The degree of similarity of the source problem to the target problem is high.
3. The number of rules that have to be searched (and hence k) is large.
4. The size of the problem (and hence n) is large.

The time to find analogue matches does not depend on the size of the knowledge base of rules but only in the size of the knowledge base of stored programs. To reduce this factor it is important that the problems should be stored judiciously. Some of the plausible heuristics that the system can use to decide whether a problem should be stored in the knowledge base are: (i) The number of rules used in deriving the problem is large; (ii) The time taken to solve the problem is large; (iii) The length of the final solution is large; (iv) The degree of match of the problem with some previous problem is low.

The fourth criteria is important to reduce redundancy in the knowledge base.

5 Conclusions

This work presents a paradigm for using analogy to derive programs from a problem specification, using the derivation of an earlier program. It differs from previous work on deriving programs using analogy in two respects. First, it maintains the derivation history of a program to determine when an analogical transformation can be validly made. Secondly, it does not depend upon an external user to provide a source analogue. It uses certain heuristics to efficiently find a good source program that is analogous to the new program. The program derivation process combines ordinary problem solving with analogical reasoning to find the solution of a problem as efficiently as possible. The program derived is deductively closed under the rules in the knowledge base. Thus, the soundness of the solution does not have to be verified.

The system presented here is under development as part of an automatic programming project to automate Unix programming. The success of the design can only be judged after it has been applied on a large knowledge base, but it seems reasonable to believe that the system would perform better as the size of the knowledge base increases, and deriving programs by a search through the knowledge base becomes more and more computationally expensive.

Acknowledgements: We thank Gregg Collins for early criticism of the work. The paper has benefited from comments of Brian Falkenhainer and members of the KBPA group.

References

[Carbonell, 1983a] J. G. Carbonell. Learning by Analogy: Formulating and generalizing plans from past experience, *in Machine Learning: An Artificial Intelligence Approach,* R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, eds. Morgan Kaufmann, Los Altos, 1983, pp. 137-161.

[Carbonell, 1983b] J. G. Carbonell. Derivational Analogy and its Role in Problem Solving. *AAAI-83,* 1983, pp. 64-69.

[Davies and Russel, 1987] T.R. Davies and S.J. Russel. A Logical Approach to Reasoning by Analogy. *IJCAI-81,* 1987.

[Dershowitz, 1983] N. Dershowitz. *The Evolution of Programs.* Birkhauser, Boston, 1983.

[Falkenhainer *et ai,* 1986] B. Falkenhainer, K. Forbus, and D. Gentner. The Structure Mapping Engine. *AAAI-86,* August, 1986.

[Gentner, 1983] D. Gentner. Structure-mapping: A Theoretical Framework for Analogy. *Cognitive Science,* vol. 7(2), 1983, pp. 155-170.

[Greiner, 1988] R. Greiner. Learning by Understanding Analogies. *Artificial Intelligence* 35. 1988, pp. 81-125.

[Kling, 1971] R.E. Kling. A Paradigm for Reasoning by Analogy. *Artificial Intelligence* 2, 1971, pp. 147-178.

[Waters, 198l] R.C. Waters The Programmer's Apprentice. *IEEE Transactions on Software Engineering,* 11(11), 1981, pp. 1296-1320.

[Winston, 1980] P.H. Winston. Learning and Reasoning by Analogy. *Communications of the ACM,* vol. 23, December, 1980, pp. 689-703.