# Analysis and Empirical Studies of Derivational Analogy[†]

Brad Blumenthal [‡]
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, IL 60680
brad@bert.eecs.uic.edu

Bruce Porter
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
porter@cs.utexas.edu

October 18, 1994

**Abstract**

Derivational analogy is a technique for reusing problem solving experience to improve problem solving performance. This research addresses an issue common to all problem solvers that use derivational analogy: overcoming the mismatches between past experiences and new problems that impede reuse. First, this research describes the variety of mismatches that can arise and proposes a new approach to derivational analogy that uses appropriate adaptation strategies for each. Second, it compares this approach with seven others in a common domain. This empirical study shows that derivational analogy is almost always more efficient than problem solving from scratch, but the amount it contributes depends on its ability to overcome mismatches

[†]Support for this research was provided by the National Science Foundation under grant IRI-8620052, Apple Computer Corporation, Hughes Corporation, Digital Equipment Corporation, and GTE Research. This research was conducted in the Department of Computer Sciences, University of Texas at Austin.

[‡]Additional support has been provided by the Medical Research Council of Great Britain.

and to usefully interleave reuse with from-scratch problem solving. Finally, this research describes a fundamental tradeoff between efficiency and solution quality, and proposes a derivational analogy algorithm that can improve its adaptation strategy with experience.

# 1 The derivational analogy technique

Derivational analogy is a technique for reusing problem solving experience to improve problem solving performance. Since its proposal [Car83, Car86], it has been applied to a number of domains, including circuit design, matrix manipulation, non-linear planning, and more recently, computer program transformations, DC circuit problem solving, and the design of human-computer interfaces [Mos89, MB87, MF89, HA87, Kam89a, CV88, Bax90, HL90, Blu90a]. While this diversity shows that derivational analogy is widely applicable, it makes it difficult to determine how and why derivational analogy works.

The goal of the research described here is to examine derivational analogy in two ways. First, the research analyzes the fundamental structure of the problem addressed by derivational analogy and the inherent limitations in the technique. Second, the research empirically evaluates the relative success of various approaches to the derivational analogy technique by implementing existing and original derivational analogy techniques and testing them on a number of problems in the same domain. Although there are a number of difficult issues that a derivational analogy algorithm must address, this paper concentrates on the problem of reducing the differences between prior problem solving experiences and new problems to facilitate reuse.

In principle, the derivational analogy technique of recording and reusing experience can be applied to any kind of problem. However, in practice derivational analogy has been applied to problems in design, search-based heuristic problem solving, and hierarchical non-linear planning. Problem solving in these three areas typically proceeds by top-down decomposition of the problem combined with instantiation of atomic solution steps. Design reuse is representative of the domain areas that derivational analogy has been applied to, and following Mostow [Mos89], the two terms are used interchangeably.

A derivational analogy algorithm proceeds through four phases: recording an experience, retrieving a *recorded experience*[1] when faced with a *new*

[1] Terms commonly used in this paper appear in italics when they are first introduced.

Derivational Analogy:
  Record one or more experiences;
  Retrieve a recorded experience;
  Reuse the retrieved experience on a new problem;
  Improve Reuse;

Retrieve Experience:
  Select recorded experience from memory;
  Establish correspondences between recorded experience and new problem;
  Evaluate correspondences;
    If necessary, select a different recorded experience
        or establish new correspondences;
  Return recorded experience and correspondences;

Reuse Experience:
  **While** there are unaddressed goals left in the new problem
      and unused goals left in the recorded experience
    **begin**
      *1. Select goal(s)*
      Choose next goal to be reused from the ordered list
        of goals in the recorded experience;
      Use the recorded goal ordering information to attempt to select
        a corresponding goal (or goals) from the new problem;
      *2. Address goal(s)*
      Determine which of the goal (or goals) selected from
        the new problem can be addressed with the
        recorded problem solving information;
      **If** an appropriate goal (or goals) is (are) selected and
          the recorded problem solving information can be applied **then**
        The problem solving information is applied and reuse succeeds; **else**
        Adapt the recorded experience and/or the new problem
          so that reuse can continue; If adaptation fails,
          fall back on a from-scratch problem solver.
      **fi**
    **end while**

Improve Reuse:
  Evaluate solution quality and efficiency;
  Record evaluation on both the recorded experience and the new problem;
  Alter retrieval and/or reuse strategies based on evaluation;

Figure 1: The skeletal derivational analogy algorithm.

*problem*, reusing the recorded experience to solve the new problem, and evaluating and improving the performance of the retrieval and reuse phases (see figure 1).

First, a derivational analogy algorithm must have a record of some problem solving experience. The simplest way to produce such a record is to have a knowledge engineer construct it. A more automatic way is to have the algorithm record the actions of a person solving solving a problem. Alternatively, instead of working with a person, a derivational analogy algorithm might record the actions taken by a from-scratch problem solver (*e.g.*, a non-linear planner [Kam89b, Vel90], a system for solving circuit analysis problems [HL90], or an automatic human-interface design system [Blu90a]).

Second, a derivational analogy algorithm must select one of its recorded experiences based on features of the new problem. The recording, indexing, and retrieval of experience is a difficult problem, and some preliminary solutions have been suggested in the derivational analogy research [BH91, Kam90b, Vel90] and in the knowledge acquisition and case-based reasoning literature [PBH90, Ham90, Kol87].

Third, the derivational analogy algorithm must adapt the selected recorded experience for reuse on a new problem that is similar, but not identical, to the original problem. This ability to adapt to differences between a recorded experience and a new problem *during the course of problem solving* is what distinguishes derivational analogy from other methods for reusing recorded experience. Compared to derivational analogy, most other methods concentrate on reusing groups of steps without adaptation [LRN86, MKKC86, DM86] or on doing adaptation after all of the recorded experience has been applied [Ham90]. This is not to say that the other parts of the derivational analogy algorithm do not present formidable problems. However, the better a derivational analogy algorithm is at adaptation, the less important it is to select just the right recorded experience.

In attempting to solve a new problem by reusing a recorded experience, derivational analogy iteratively performs two steps: *selecting goals* and *addressing goals* (see figure 1). During each iteration of derivational analogy, one or more goals are selected from the new problem that are considered similar to a goal in the recorded experience.[2] Some algorithms select goals by simply reusing the order in which goals were selected in the recorded

[2]Different algorithms for derivational analogy use different criteria for similarity. However, this paper is more concerned with adaptation than goal selection. Therefore, the algorithms tested in the experiments described in section 4 use a common similarity criterion.

experience. Others select goals by reusing the *rationale* behind the order in which the goals were selected.

Once goals in the new problem are selected, they are addressed by reusing the problem-solving information (*i.e.,* rules, variable bindings, and constraints) that were applied in the recorded experience. However, because the recorded experience and the new problem are not identical, this information might be inapplicable to the selected goals. Such a failure surfaces as an unsatisfied precondition for a rule, an illegal variable binding, or a constraint violation. If the steps of selecting and addressing goals are unsuccessful, either because the recorded goal ordering information does not select any goals from the new problem, or the recorded problem solving information cannot be applied to the goals that are selected, then derivational analogy attempts to adapt either the new problem or the recorded experience so that reuse can continue. When differences between the recorded experience and the new problem cause reuse to fail, a derivational analogy algorithm must fall back on *from-scratch* problem solving, either a human expert or a search-based, heuristic problem solver.

The last step in the basic derivational analogy algorithm is to analyze the results of reusing the experience and to use that analysis to change the performance of the derivational analogy algorithm in subsequent problems. Such modifications may involve changes to the features used for indexing the recorded experience [VC89] or improvements to the adaptation process [HL91].

Because of its crucial role in derivational analogy, the ability of a derivational analogy algorithm to adapt a recorded experience or a new problem to enable reuse is the focus of this research. We will measure this ability in two ways. First, the total amount of effort expended by a derivational analogy system to solve a problem (including the effort expended during any from-scratch problem solving) is a measure of that system's *efficiency*. Second, the proportion of the new problem that is solved by reuse, rather than by from-scratch problem solving, is a measure of that system's *autonomy*.

## 2 Obstacles to reuse – The need for adaptation

A derivational analogy algorithm attempts to adapt a recorded experience and a new problem when it encounters an obstacle to reuse. An obstacle is a *mismatch* between the structure of the solution stored in the recorded experience and the structure of the desired (but unknown) solution to a new

problem. As noted above, derivational analogy algorithms have typically been applied to domains where problem solving proceeds by problem decomposition. In such domains, the goals addressed in the process of solving a problem form an and-tree structure, with subgoals represented as children nodes of their supergoal. The *topology* of such a goal tree is the structure of the tree as determined by its overall depth, its breadth at each level, and the relationships between the parent and children nodes. When the desired solution to the new problem matches the solution in the recorded experience, the two goal trees have the same topology, and corresponding nodes represent corresponding goals. When the topologies are not the same, a mismatch occurs and adaptation is required.

## 2.1 A testbed for studying adaptation

In order to present concrete examples of the kinds of obstacles that a derivational analogy algorithm encounters, this section introduces the domain of automated metaphoric human interface design, which is used as the testbed domain for comparing derivational analogy systems.

Briefly, a metaphoric interface is one that uses features from the real world to present the appearance and behavior of the objects and operations that a computer application makes available to the user. Perhaps the most familiar metaphoric interface is the desktop metaphor for operating systems. This interface uses pictures of pieces of paper to represent files, pictures of file folders to represent directories, and animations of putting pictures of pieces of paper into pictures of folders to represent putting files into directories.

The domain of human interface design has a number of advantages as a testbed for comparing the various derviational analogy algorithms. Most important for the purposes of the empirical evaluations, this domain exhibits a large number of mismatches among different interface design problems, while still retaining enough similarity among problems that reuse is beneficial.

A second advantage of the interface design domain is that the quality of the solutions produced in this domain is sensitive to the order in which goals are addressed. The reasons for this sensitivity are discussed more fully elsewhere [Blu90a]; however, one example involves the design of *direct manipulation* interfaces [HHN86]. If an operation in an interface acts on an entity that is displayed on the screen, it is often desirable to implement the gesture for that operation as a direct manipulation action on that entity. In terms of automated interface design, this requires that the screen area for that entity be allocated before the interface to the operation is designed.

Otherwise, a non-direct manipulation operation will be designed instead. The implication of this sensitivity for our empirical study of derivational analogy algorithms is that the performance of the algorithms can be measured not only in terms of autonomy and efficiency, but also in terms of solution quality (as discussed in section 5).

Even in domains that have some formal criteria for correctness, solution quality may be an issue. While a circuit design can be formally shown to implement the logic given in a specification, it cannot necessarily be formally shown to be the best circuit in terms of size, power consumption, heat dissipation, ease of manufacturing, and so on.

To illustrate the variety of mismatches that arise among different interface design problems, we present two interfaces generated by MAID, a from-scratch problem solver. [3] The interfaces are for a simple data manager application, which maintains a list of records, each of which has a name field, an address field, and a phone number field. In addition, the application allows the user to add and delete records and browse through the records one at a time, either forwards or backwards.

One metaphoric interface that MAID designs for the data manager application has some of the appearances and behaviors of a note pad (figure 2 shows the appearance of part of this interface). This particular interface design presents the information in a data manager record in the way that a name, address, and phone number would canonically be written at the bottom of a notepad page. To create this design, MAID adds entities to the interface corresponding to notepad pages, the spine holding the pages together, extra entries for names, addresses, and phone numbers on the notepad page, *etc.*

A second metaphoric interface that MAID designs for the data manager application uses the characteristics of a Rolodex[4] to determine the appearance and behavior of the interface. The design of this interface uses the appearance of the top Rolodex card on a Rolodex to present a data manager record and introduces a number of new entities to the application, including a spindle, a frame, and a face-down bottom card (see figure 3).

As may be clear from a cursory glance at the note pad and Rolodex examples, there are a number of differences between the two interface designs. For example, the spindle in the Rolodex corresponds to the spine

[3] MAID is an acronym for Metaphoric Application Interface Designer. It is described more fully elsewhere [Blu90b, Blu90a, Blu90c].

[4] Rolodex is a trademark of the Rolodex Company.

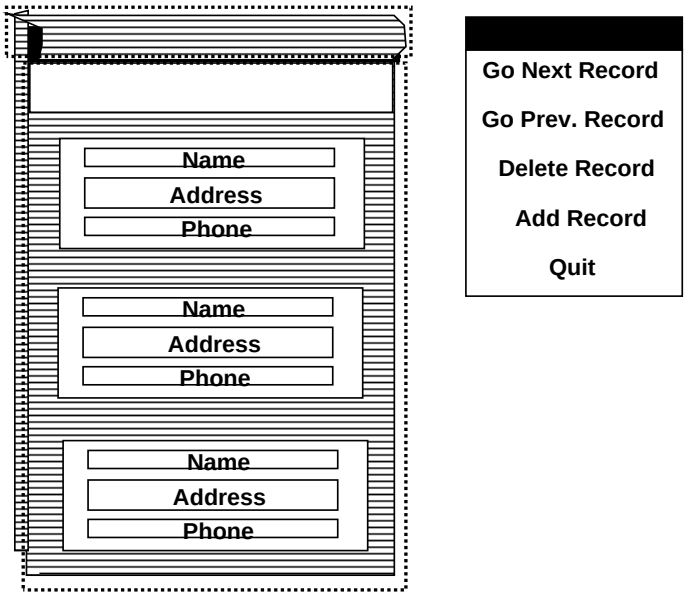


Figure 2: A MAID metaphoric interface design for the data manager application using characteristics from the note pad. Hashed lines indicate mouse-sensitive regions.

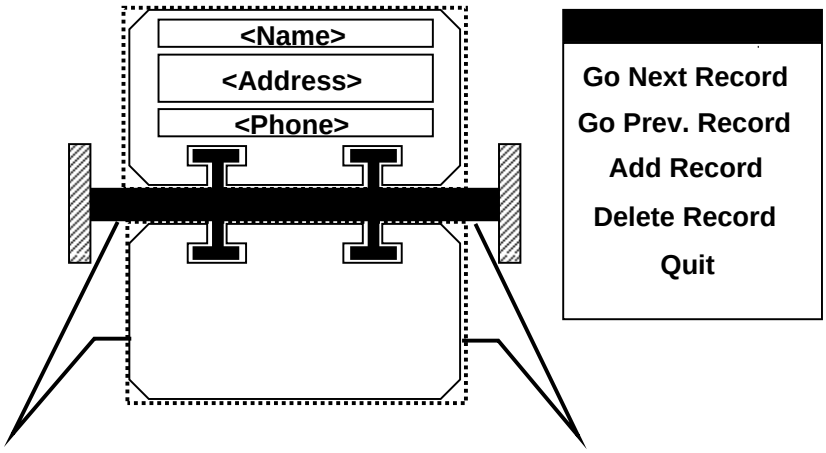


Figure 3: Part of the design of the interface to the data manager application using the Rolodex metaphor.

on the notepad, but the Rolodex frame does not correspond to anything in the notepad. If the Rolodex design is used as the recorded experience when designing the notepad interface, then a derivational analogy algorithm must adapt to this difference. Another example is the notepad page, which does not correspond to anything in the Rolodex-inspired interface. Besides adding an extra entity that must be adapted to, the notepad page introduces a different organization of the problem that must be adapted to. The next section enumerates the kinds of differences that a derivational analogy must adapt to and gives examples from the human interface design domain.

## 2.2 Topology mismatches – a bird's eye view

Upon examining the topologies of various solution trees, it becomes clear that there are several distinct kinds of mismatches occurring during the reuse of recorded experience. Furthermore, the empirical evidence gathered in the course of this research (reported in section 4) convincingly argues that a derivational analogy algorithm must be sensitive to these differences if it is to give the best performance on any but the most trivial applications of derivational analogy. In particular, different strategies are necessary to cope with the various kinds of mismatches, and a strategy that is appropriate for one kind of mismatch may be disastrously expensive when applied to a different kind of mismatch.

Looking at the complete solution trees for the recorded experience and the new problem solution, there are essentially three ways the two topologies can fail to match. There can be goals in the new problem that fail to match goals in the recorded experience, goals in the recorded experience that fail to match goals in the new problem, and goals in both the recorded experience and the new problem that fail to match goals in the other. These mismatches are called *detours*, *pretours*, and *combinations* respectively.

There are a number of ways these mismatches can manifest themselves between different solution trees, some of which may not be as straightforward as the descriptions and diagrams presented here. However, the linear order imposed on the goals by a sequential problem solver makes detours, pretours, and combinations adequate for describing the mismatches encountered by a derivational analogy algorithm. Describing mismatches in terms of the solution trees gives a better picture of the difficulties involved in overcoming these obstacles, and the linear order imposed during sequential problem solving collapses a number of different solution tree mismatches into the categories described in the next sections.

In the discussion of mismatches, and in the implementations of replay algorithms described in section 4, the match criteria is loosely based on BOGART's [MB87]. Two goals match if: 1) the goals are of the same type, 2) the goals address the same or similar entities, and 3) the super-goals of the two goals satisfy conditions 1) and 2). Determining whether the two entities being addressed are similar is a domain-specific problem. In the MAID domain of metaphoric human-computer interface design, two entities are similar if they are instances of the same class of entity or if there is a metaphoric mapping between the two entities. Metaphoric mappings are either given by the designer as part of the initial problem statement, or established by the MAID design rules (for a detailed discussion of how MAID automatically establishes new mappings see [Blu90c, Blu90b, Blu90a].

### 2.2.1 Detours

When a new problem contains goals that do not correspond to anything in the recorded experience, these goals are referred to as detours. Since nothing in the experience indicates when or how these goals should be addressed, the derivational analogy program must notice these new goals and address them appropriately.

In practice, there are two kinds of detours that may be encountered. The less troublesome of the two is a *horizontal detour* which occurs when the extra goal in the new problem is a sibling of a goal that matches some goal in the recorded experience (see figure 4).

As an example, in the note pad design, MAID addresses the goal of designing the note pad spine before it addresses the goals for the various operations (goals of addressing objects are usually chosen before goals of addressing operations so that direct manipulation operations can be implemented). When this design is used as the recorded experience to solve the Rolodex design problem, a horizontal detour is encountered. The goal of designing the Rolodex spindle matches the goal of designing the note pad spine, but the goal of designing the Rolodex frame does not match any goal on the retrieved experience.

If a derivational analogy algorithm fails to recognize this sort of detour, then the extra goal and all of its descendants may not be addressed at all. If they are eventually noticed (*e.g.*, as goals left unaddressed after reuse has finished), they will have to be addressed in some other way (*e.g.*, by calling on a person or calling on a from-scratch problem solver). It can be argued that leaving such goals unaddressed is a reasonable course of action for

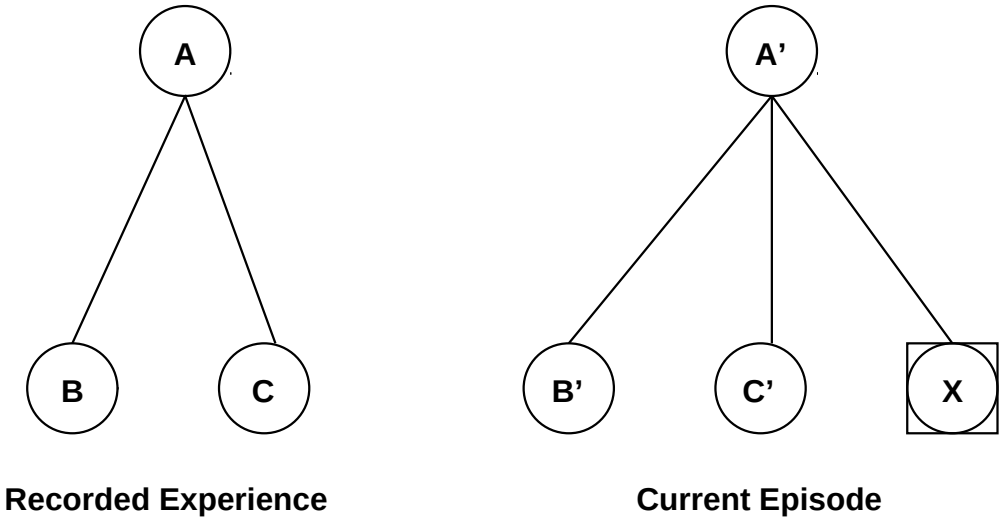


Figure 4: A schematic description of a horizontal detour. The goal (or set of goals) marked X in the new problem does not correspond to anything in the recorded experience.

a derivational analogy algorithm since nothing in the experience indicates how such goals should be handled. However, depending on the nature of the domain, this strategy of ignoring horizontal detours may have adverse effects on the quality of the resulting problem solution (as described briefly in section 5 and in detail elsewhere [Blu90a]).

A more troublesome kind of detour is the *vertical detour*. Like a horizontal detour, this sort of detour occurs when there are some extra goals in the new problem that do not correspond to anything in the recorded experience. The difference is that in a vertical detour these new goals are spliced between a parent goal and its subgoals in the recorded experience (see figure 5).

An example of a vertical detour is encountered when the Rolodex design is used as the recorded experience for the note pad design. The goal of designing an entry on the notepad corresponds to the goal of designing a Rolodex card (since a notepad entry and Rolodex card both record one name, address, and phone number). However, these goals are not aligned in the solution trees, as shown in figure 6.

The reason that vertical detours are more difficult to handle is that the intervening, unmatched goals in the new problem must be addressed

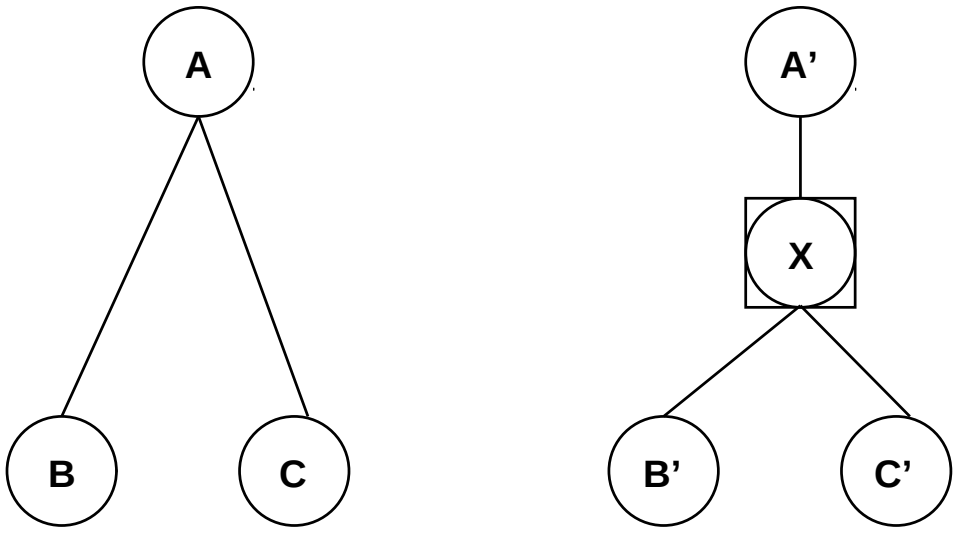


Figure 5: A schematic description of a vertical detour. The goal (or set of goals) marked X in the new problem does not correspond to anything in the recorded experience.

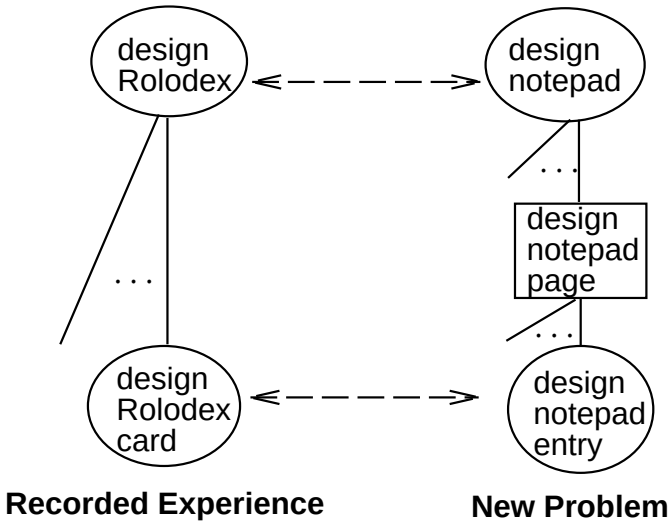


Figure 6: An example of a mismatch between the solution trees for designing records with a Rolodex interface versus a notepad interface. The goal of designing a note pad page does not correspond to anything in the Rolodex design.

(somehow) in order to spawn the goals below the detour that do correspond to goals in the recorded experience. In the example above, the goal of designing the note pad entry is not added to the list of pending goals until the goal of designing the note pad page is addressed. If a vertical detour is not addressed, it may block the appearance of a large number of goals in the new problem that could be addressed by derivational analogy. In addition, it should be emphasized that the number of goals in a detour, as well as any other mismatch, is arbitrary. There can be a large number of goals that must be addressed by the adaptation mechanism before derivational analogy can resume.

### 2.2.2 Pretours

If the recorded experience contains goals that do not match any goals that occur in the new problem, such goals are referred to as *pretours*[5] (see figure 7). Since these goals cannot be reused to address goals in the new problem, it is up to the derivational analogy algorithm to skip these goals and continue reuse at some appropriate later point.

An example of a pretour occurs when the note pad design is used as the recorded experience for the Rolodex design, that is, when the roles are reversed from the previous example of a vertical detour. In this case, the goal of designing a note pad page (from the recorded experience) must be ignored. If there were always just one intervening goal, then simply skipping a goal might be an effective adaptation method. However, as there may be many intervening goals, a derivational analogy algorithm needs an efficient way of determining how many goals to skip.

Although, technically, there are horizontal and vertical pretours in the same sense that there are horizontal and vertical detours, in practice there is essentially no difference between the two. In both cases, goals in the recorded experience must be ignored, and since goals in the recorded experience are considered in a linear order, they can be ignored in the same way. This is an example of how the linear order imposed by a sequential problem solver can conflate apparently different kinds of topology mismatches into a single class for the purposes of a derivational analogy adaptation algorithm.

[5] This is a contraction of "previous detour," indicating that such goals are similar to detours, but in the previously recorded experience.

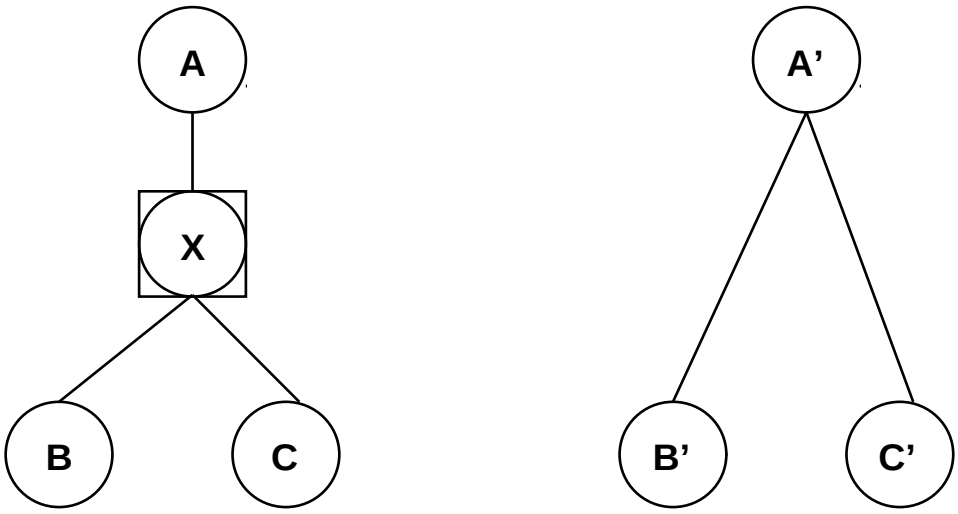


Figure 7: A schematic description of a pretour. The goal (or set of goals) marked X in the recorded experience does not correspond to anything in the new problem.

### 2.2.3 Combinations

When both a pretour in the recorded experience and a detour in the new problem exist at the same time, the situation is referred to as a *combination* (see figure 8). Combinations are especially difficult to adapt to because they are difficult to distinguish from pretours and detours, and many of the strategies that work for pretours and detours are not effective for combinations (see section 2.3).

One example of a combination occurs when the Rolodex interface design is used as the recorded experience for the note pad interface. The goals for designing the Rolodex frame constitute a pretour. In addition, the goals for designing the note pad page constitute a vertical detour, and the goals for designing the top entry on the note pad page constitute a horizontal detour. Thus, combinations can include pretours and both vertical and horizontal detours.

If a combination is assumed to be a detour, and goals are addressed somehow in the new problem, then the extra goals in the recorded experience are never skipped. Reuse is never resumed because the goal in the recorded experience that the system is attempting to reuse does not correspond to any goal in the new problem.

If a combination is assumed to be a pretour, and goals in the recorded experience are skipped, then the detour is never adapted to, and none of the goals in the detour, or the goals that might be spawned by goals in the detour, are ever addressed. To successfully adapt to a combination mismatch, a derivational analogy algorithm must ignore the goals making up the pretour and somehow address the goals making up the detour.

## 2.3 Topology mismatches – what the algorithm sees

While mismatches are best described in terms of complete solution trees, during reuse, a derivational analogy algorithm does not have the entire solution tree to examine. This section briefly describes the information that a derivational analogy algorithm has available to determine what kinds of mismatches are encountered and how to adapt to them during the course of reusing a recorded experience.

In the case of vertical detours, pretours, and combinations, all the algorithm can determine initially is that no goal in the new problem appropriately matches the goal selected from the recorded experience. This is what makes these kinds of mismatches difficult to distinguish and difficult to adapt

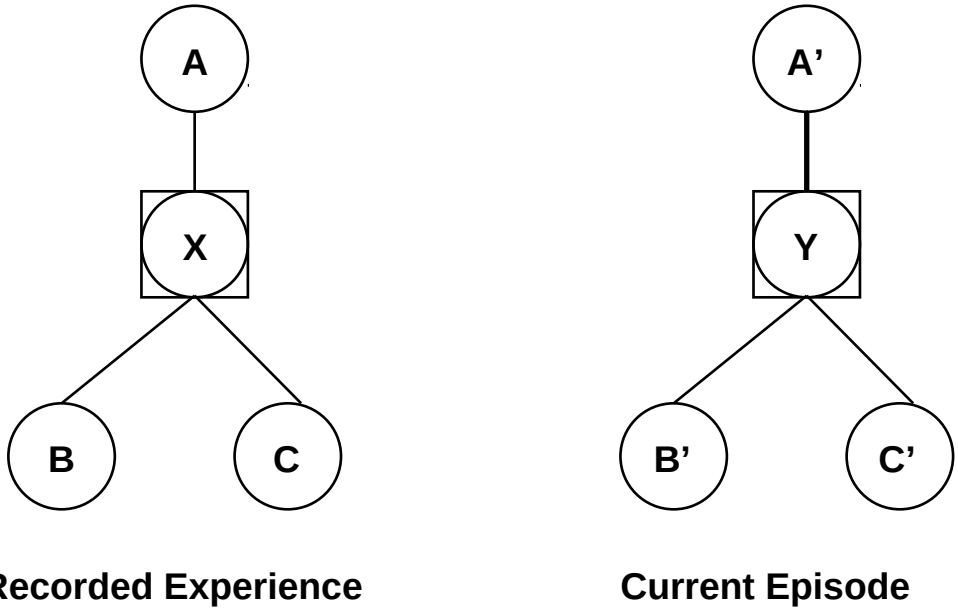


Figure 8: A schematic description of a combination mismatch. The goal (or set of goals) marked X in the recorded experience does not correspond to anything in the new problem, and the goal (or set of goals) marked Y in the new problem does not correspond to anything in the recorded experience.

to. As adaptation progresses, however, a sufficiently sensitive algorithm can distinguish, and appropriately adapt to, each kind of mismatch.

In the case of horizontal detours, an appropriate procedure for matching goals from the recorded experience to goals in the new problem can detect such mismatches by selecting more than one goal from the new problem. However, it is up to the derivational analogy algorithm to handle such a mismatch in an appropriate fashion.

Most implemented derivational analogy algorithms do not distinguish various kinds of mismatches and, therefore, only use one approach to adaptation regardless of the type of mismatch being faced. When one of these algorithms successfully adapts to mismatches, it is because the algorithm's approach is fortuitously appropriate for the mismatches encountered. When an approach is not appropriate for a given mismatch, these derivational analogy algorithms may produce extremely inefficient results or fail to solve the new problem at all.

A derivational analogy algorithm that can distinguish various kinds of topological mismatches and apply different, appropriate approaches to each kind can produce more efficient, successful results. One such algorithm is described in section 3.5. The difficulty faced by such an algorithm is that only incomplete information about the type of mismatch encountered is available while the algorithm is solving a new problem.

# 3 Approaches to adaptation in derivational analogy algorithms

Derivational analogy algorithms use two strategies to adapt a recorded experience and a new problem to eliminate mismatches between them. *Local adaptation* refers to the strategy that the derivational analogy algorithm uses when there is no goal in the new problem that appropriately corresponds to one particular goal in the recorded experience. *Recovery* is the strategy that the derivational analogy algorithm uses when it cannot find an appropriate correspondence for *any* goal in the recorded experience.

To show the range of local adaptation and recovery strategies employed by current derivational analogy algorithms, five programs are presented here. Each of these programs is described in turn, with an analysis of the local adaptation and recovery strategies that it employs.[6]

[6]Some of the programs described here assume that their local adaptation strategies are sufficient, and therefore do not explicitly include a separate recovery strategy.

The first four programs have been described in the literature and all use the recorded experience to guide local adaptation and recovery: the BOGART/VEXED system of Mostow and Barley [MB87], the extensions to PRODIGY done by Carbonell and Veloso [CV88, Vel90], the PRIAR system of Kambhampati [Kam89b], and the work on internal analogy in the context of the RFermi system done by Hickman [HL90].

The fifth one is the REMAID[7] system for reusing human interface design experience [Blu90a] and is one of the results of the research described here. REMAID embodies three novel approaches to the problem of adaptation and recovery. First, it was designed to distinguish the various kinds of topology mismatches described in section 2 and to use different adaptation strategies for each. Second, REMAID uses the state of the new problem, rather than the recorded experience, to more efficiently and successfully guide adaptation. Third, REMAID makes novel use of a search-based problem solver (in addition to other strategies) both to help distinguish various kinds of mismatches and guide adaptation. Some of the other algorithms use some sort of search-based problem solver to handle parts of the new problem that cannot be addressed by reusing the recorded experience, thereby adapting to detours. However, only REMAID uses such a problem solver to help analyze the differences between the recorded experience and the new problem.

Seven derivational analogy algorithms based on the various recovery and adaptation strategies found in the literature have been implemented so that they can all be applied to a common domain. Together with REMAID, these implementations have been used in the empirical evaluations described in section 4 to determine the relative strengths and weaknesses of each approach. In order to test these approaches in the same domain, namely that of automated human interface design, a number of compromises had to be made among the various algorithms. Thus, while the implementations inspired by approaches in the literature are not complete reimplementations, they faithfully embody the applicable recovery and adaptation strategies from the literature in a way that can be meaningfully compared.

### 3.1 BOGART

The BOGART derivational analogy program records the design goals and rules chosen by a human designer who is using the VEXED circuit design program. To solve a new problem, the designer provides BOGART with an

[7] REMAID is an acronym for Replaying Episodes of MAID. MAID is described in section 2.1.

initial match between a goal in the recorded experience and a goal in the new problem. BOGART then applies the recorded goal choices and design rules in the order in which they were recorded.

BOGART has the simplest local adaptation strategy of the programs presented here. When BOGART encounters a goal in the recorded experience that does not correspond appropriately to any goal in the new problem, it simply skips the goal in the recorded experience and continues with the next goal in the recorded experience. After trying to reuse every goal in the recorded experience, BOGART returns to the top of the list of goals and retries all of the recorded goals that failed to be reused. The rationale for this is that a goal may not have been reused because of a constraint violation that was repaired by addressing a later goal. BOGART repeats this process until it tries to reuse every goal remaining in the recorded experience without success.

BOGART's adaptation strategy is to simply skip goals in the recorded experience that do not match a goal in the new problem. This strategy is adequate for addressing pretours, since it can effectively ignore any goals in the recorded experience that do not correspond to goals in the new problem. However, BOGART depends on the user to address detours manually and to restart reuse.

BOGART's recovery strategy is equally simple. When it has made a pass through the recorded experience and none of the goals can be reused, BOGART simply halts and waits for the user to provide it with a new recorded experience.

HOBART[8] is an implementation of the BOGART approach to adaptation and recovery. HOBART skips any goal in the recorded experience that cannot be reused to address a goal in the new problem. It continues to cycle through the list of goals in the recorded experience until none of them can be reused to address a goal in the new problem, and then it halts.

### 3.2 PRODIGY extensions

Part of the PRODIGY system is a program for doing matrix manipulations such as Gaussian elimination. Carbonell and Veloso's extensions to the PRODIGY system [CV88] reuse a recorded experience that includes information about each goal that was selected, such as

[8] Hobart is a registered trademark of the Hobart Food Machinery Company. The names of the implementations were chosen for purely arbitrary reasons.

- what objects in the problem the goal operated on (*e.g.*, what row of the matrix is operated on),
- what step of the problem the goal addressed (*e.g.*, scale a row),
- the order in which the goals were addressed,
- the rule choices that produced the solution,
- the variable bindings that were used when instantiating a problem solving rule, and
- the justifications for each of these decisions.

During reuse, the PRODIGY system selects a goal in the new problem that matches the next goal in the recorded experience. It then checks that the various justifications in the recorded experience are still valid for the goal in the new problem. If the justifications still hold, the recorded rule is applied and reuse continues. If the justifications no longer hold, then PRODIGY follows what Carbonell and Veloso call the *satisficing approach*. This involves either establishing the current goal by other means, or patching the mismatch that caused reuse to fail by adding the creation of the patch as a new goal.

One way of implementing this approach is as a local adaptation strategy that would try different rules than the one recorded to see if one can be found to address the current goal in the new problem. This local adaptation strategy can adapt to a limited class of combination mismatches (those where there are the same number of goals in both the recorded experience and the new problem, and simply using different rules is sufficient adaptation).

A second way of implementing this approach is as a recovery strategy that would call on some other from-scratch problem solving program to address one goal in the new problem when the derivational analogy algorithm has failed to reuse all of the goals remaining in the recorded experience. Then the algorithm could attempt to restart reuse. This would adapt to vertical detours and any horizontal detours that are detected.

To test these strategies, three algorithms were constructed by implementing the two methods outlined above and adding them to the basic HOBART algorithm, both separately and together (recall that HOBART uses an adaptation strategy of skipping a goal and trying the next one).

The implementation of HOBART with the PRODIGY-inspired adaptation strategy (alternate problem solving rules) is called "PROBART." The

implementation that has the recovery strategy (select and address one goal with a problem solving system) is called "POSSIBLY." The implementation with both the adaptation strategy and the recovery strategy is called "PROBABLY."

### 3.3 PRIAR

Technically, Kambhampati's PRIAR system [Kam89b] is closer to an implementation of Gentner's structural analogy ideas [Gen83] than to an implementation of a derivational analogy algorithm. In particular, PRIAR tries to determine what part of the recorded solution, rather than the solution process, is applicable and tries to modify the solution to fit the new problem. However, it exhibits a relevant recovery strategy.

The from-scratch problem solver for PRIAR is a non-linear planner. The reuse component uses a *validation structure* to formally determine which steps from the recorded plan are needed, and which goal conditions in the new problem must be solved by new steps added by the PRIAR recovery mechanism [Kam89a]. This recovery mechanism repairs the recorded plan by taking any goal conditions in the new problem that are left unsatisfied by the plan in the recorded experience and posting them as new goals. These new goals are then addressed by the same non-linear planner that produced the recorded plan. PRIAR's recovery strategy is interesting in that it addresses detours by using the same from-scratch problem solver that generated the recorded experience. The PRIAR system does not perform local adaptation in the course of problem solving; the purpose of the validation structure is to determine exactly which steps in the recorded experience should be reused and which should be eliminated.

PRIAR's recovery strategy inspired two implementations that use a from-scratch problem solver to address any goals left unaddressed in the new problem without attempting to restart reuse. The first, called "BRIAR," uses the HOBART algorithm until there are no more goals in the recorded experience that can be used on existing goals in the new problem, and then it invokes a from-scratch problem solver to address any leftover goals. Unlike the PRODIGY-inspired systems, the PRIAR-inspired recovery strategy does not try to restart reuse when new goals have been added to the new problem agenda by the from-scratch problems solver. The second algorithm, called "PYRE," uses the PRODIGY-inspired adaptation strategy (using alternate problems solving rules) and the PRIAR-inspired recovery strategy (using the from-scratch problem solver to finish solving the problem).

| *Algorithm* | *Adaptation Strategy* | *Recovery Strategy* |
|---|---|---|
| HOBART | Skip goal | Halt |
| PROBART | Alternate problem solving rule | Halt |
| POSSIBLY | Skip goal | From-scratch problem solver for one goal; restart reuse |
| PROBABLY | Alternate problem solving rule | From-scratch problem solver for one goal; restart reuse |
| BRIAR | Skip goal | From-scratch problem solver for all remaining goals |
| PYRE | Alternate problem solving rule | From-scratch problem solver for all remaining goals |

Table 1: Summary of algorithms tested: strategy for adapting to single recorded rule failures, and strategy for recovering when the recorded experience can suggest no more goals.

Table 1 summarizes the strategies used by the six implementations described so far. There are two local adaptation strategies: simply skip the recorded goal and try it later or use alternate problem solving rules. There are three recovery strategies: halt, use a from-scratch problem solver for one goal then restart reuse, or use a from-scratch problem solver for all remaining goals.

### 3.4 Internal analogy

The work of Hickman and her colleagues uses techniques from derivational analogy to improve performance without having to resort to a previous problem [HL90]. This approach is referred to as *internal analogy* and depends on regularities *within* the problems in a domain to allow solutions to subproblems to be reused during the course of solving a single larger problem.

Hickman's program determines if a set of problem solving steps is appropriate to reuse by calculating the *information content* of the original subproblem and comparing that to the information content of the new subproblem. The information content of a subproblem is a measure of the number of bound variables in the left hand side of the rule addressing the subproblem. Hickman has shown some preliminary success with a program that simply uses the information content metric to determine which previous

subproblem solutions can be reused.

Like PRIAR, Hickman's program does recovery by simply using its from-scratch problem solver, the RFermi program. There is no explicit recovery strategy here; in the course of doing problem solving, this program either finds appropriate goals to be reused, or it does not. If there are no more appropriate goals to be reused, the RFermi program simply finishes the problem solution.

The approach taken from Hickman's internal analogy algorithm is the strategy of using the list of addressed goals in a problem as the source for reusable experience. An algorithm embodying this strategy was implemented (in a program called LASH) to determine the viability of internal analogy in the domain used for the empirical evaluations. Instead of looking at a separate recorded problem solving experience, LASH uses the list of addressed goals as its recorded experience. LASH uses the same adaptation and recovery strategies as the REMAID system (described in the next section).

### 3.5 REMAID

Unlike the other algorithms described and implemented here, REMAID was designed with topology mismatches in mind; where possible, its strategies distinguish the various mismatches and adapt to them appropriately [Blu90a]. REMAID's capability to distinguish and adjust to detours, pretours, and combinations allows it to efficiently continue reuse when mismatches between the recorded experience and the new problem are found.

The REMAID adaptation strategy is based on the philosophy that the new problem, not the recorded experience, should be used to guide adaptation when mismatches occur. One way that this philosophy manifests itself is that REMAID, unlike other derivational analogy algorithms, does not strive to conform to the ordering of goals in the recorded experience. Instead, REMAID dynamically reorders its recorded experience in response to mismatches, in a fashion that is sensitive to the state of the new problem. Whereas other algorithms try to make the new problem as similar to the recorded experience as possible, REMAID modifies its recorded experience, not the new problem, to adapt to different kinds of mismatches in appropriate ways. Put more informally, when REMAID is confronted with a mismatch, it steps back and takes a fresh look at the new problem to determine how to proceed.

By contrast, BOGART follows the ordering of goals in its recorded ex-

perience as closely as possible. The adaptation strategies that PRODIGY pursues focus on adapting the new problem so that as much of the recorded experience as possible can be reused, including the ordering of goals. Although not specifically concerned with goal ordering, PRIAR concentrates on reusing as many of the plan steps in its recorded experience as possible. In these other algorithms, this adherence to the recorded experience during reuse makes it difficult to distinguish and efficiently adapt to various kinds of mismatches.

The philosophy of attending to the new problem to guide adaptation is implemented in REMAID by calling on a from-scratch problem solver in a unique fashion. Like RFermi and PRODIGY, REMAID can use a from-scratch problem solver to handle detours by addressing goals that cannot be addressed by reusing recorded experience. However, unlike other derivational analogy systems, REMAID also uses the from-scratch problem solver to select goals from the new problem that are used as a guide to distinguish and efficiently adapt to the various kinds of mismatches. Thus, REMAID uses a from-scratch problem solver not only to work more autonomously (since goals that cannot be addressed via reuse can still be addressed without the aid of a human expert), but also to be more sensitive to the new problem by determining what to attend to when mismatches occur.

As with the sections describing the other programs, the rest of this section concentrates on the adaptation and recovery strategies implemented in the REMAID system. REMAID follows the cycle of selecting and addressing goals described in section 1. In particular, REMAID selects a goal to be addressed by choosing a goal to be reused from the recorded experience and reusing the recorded rationale to select a goal (or goals) from the new problem. Then it analyzes its selection using both the similarity criteria (common to all of the algorithms described here) and the preconditions of the problem solving rules applied to the goal from the recorded experience.

There are three situations that this analysis can immediately detect. In the simplest case (figure 9, case 1), REMAID selects only one similar goal from the new problem, and the recorded problem solving information is appropriate for addressing that goal. In this case, there is no mismatch; REMAID assumes that the new problem is similar enough to the recorded experience that no adaptation is needed, and reuse continues.

### 3.5.1 Horizontal detours

A slightly more complex case occurs when REMAID selects more than one goal, and the similarity criteria and recorded problem solving information are successfully applied to at least one of those goals. In this case (figure 9, case 2), there is a horizontal detour; REMAID assumes that either new goals or different characteristics are present in the new problem that were not in the recorded experience. The presence of a goal that actually does match the goal from the recorded experience, along with some extra goals, indicates that the mismatch is a horizontal detour.

In this situation, REMAID must first determine which of the extra goals selected are new goals and which of the extra goals correspond to other goals in the recorded experience that have different characteristics that cause them to be chosen at different times. REMAID does this by matching each of the extra goals against the unused goals left in the recorded experience.[9] Any goal that matches some other goal in the recorded experience is assumed to have some different characteristics in the new problem that cause it to be selected in the new problem even though it was not chosen in the recorded experience. These goals are not immediately addressed on the assumption that they will be addressed when the matching goal in the recorded experience reaches the front of the list of recorded goals. The rationale is that there may be some difference between the recorded experience and the new problem, but that the selection of a matching goal from the new problem indicates that the difference is not substantial enough to warrant changing the order in which the goals are addressed.

Any extra goals that do not match a goal in the recorded experience are assumed to be part of a horizontal detour. REMAID first addresses the goal corresponding to the one that was chosen in the recorded experience, then REMAID addresses the goals in the horizontal detour by calling on a from-scratch problem solver. The recorded goal is removed from the recorded experience, and reuse continues with the next goal in the recorded experience.

### 3.5.2 Vertical detours, pretours, and combinations

The most complex case (figure 9, case 3) occurs when the REMAID algorithm selects zero or more goals and the recorded problem solving informa-

[9] This is not as expensive as it may sound. In particular, since no new goals are ever added to the recorded experience, the goals recorded there can be efficiently indexed.

Basic Operations:

PS-SELECT: The from-scratch problem solver's goal selection procedure.
PS-ADDRESS: The from-scratch problem solver's procedure for addressing a goal.

R-SELECT: Select a goal using rationale from recorded experience.
R-FILTER: Check goals chosen by R-SELECT using similarity and precondition checks (returns one or zero goals).

RECOGNIZE-AS-UNUSED: Search unused goals in recorded experience and determine that goal selected from new problem matches one.
ROTATE-UNUSED: Reorder goals in recorded experience so that selected goal is first.

REMAID algorithm:

**Case:**

**1.** R-SELECT chooses one goal **and**
R-FILTER passes that goal: *(no mismatch)*
Reuse goal; no adaptation needed.

**2.** R-SELECT chooses more than one goal **and**
R-FILTER passes one goal: *(horizontal detour)*
Reuse goal to address goal passed by R-FILTER;
**For** each goal, G, chosen by R-SELECT
**and not** passed by R-FILTER:
**If** RECOGNIZE-AS-UNUSED G
**then** Ignore G (G will be addressed later).
**else** PS-ADDRESS G.

**3.** R-SELECT chooses zero or more goals **and**
R-FILTER passes zero goals: *(vertical detour, pretour, or combination)*
Goal **:=** PS-SELECT;
**If** RECOGNIZE-AS-UNUSED Goal *(pretour)*
**then** ROTATE-UNUSED Goal and resume reuse.
**else**
**Loop**
PS-ADDRESS Goal;
Goal **:=** PS-SELECT;
**If** RECOGNIZE-AS-UNUSED Goal *(vertical detour or combination)*
**then** ROTATE-UNUSED Goal;
**break** (resume reuse).
**else** continue loop;



Figure 9: The REMAID derivational analogy algorithm.

tion is not appropriate for any of the goals selected. As noted in section 2.3, from this information alone, it is impossible to tell whether the mismatch is a vertical detour, a pretour, or a combination. At this point REMAID has to pursue a strategy that is flexible enough to adapt to each kind of mismatch in an appropriate way.

As noted earlier, the philosophy behind the REMAID adaptation strategy is that the *new problem*, not the recorded history, should determine how adaptation should proceed. To realize this adaptation strategy, REMAID calls on the goal selection mechanism of a from-scratch problem solver to choose a goal from the new problem to be addressed. It then attempts to use this goal as an index into the list of goals in the recorded experience to determine which goal should be reused next. It does this by matching the goal chosen by the from-scratch problem solver to the list of unused goals in the recorded experience. If it finds a match, then a pretour has been encountered, and the list of unused goals is rotated so that the matching goal is at the front. The rationale for rotating the list of unused goals is to restart reuse in the context of the next appropriate goal. REMAID then addresses the chosen goal from the new problem, removes the matching goal from the list of unused goals, and restarts reuse with the next goal in the recorded experience. This strategy adapts to pretours by reordering the recorded experience to correspond to the state of the new problem.

If the match algorithm does not find a match between the goal chosen by the from-scratch problem solver and some goal on the recorded experience, then a vertical detour or a combination has been encountered. In this case, REMAID continues by letting the from-scratch problem solver choose a rule to address the chosen goal. REMAID then calls the from-scratch problem solver again to choose another goal. This process repeats until a match is found between the goal chosen by the from-scratch problem solver and some goal on the recorded experience. If the match eventually found involves the first unused goal on the recorded experience, then the mismatch is a vertical detour.

If the matching goal eventually found by the from-scratch problem solver involves a goal other than the one on the front of the recorded experience, then a combination of pretours and detours has been encountered. The adaptation strategy uses the from-scratch problem solver to address the goals in the detour component of the combination in the fashion described above for vertical detours. As the from-scratch problem solver chooses each goal, this strategy uses that goal as an index into the list of goals in the recorded experience in an attempt to adapt to the pretour component of the

combination in the manner described above for adapting to pretours.

Because REMAID was developed with an analysis of topology mismatches in mind, it can effectively distinguish and adapt to different kinds of mismatches between the recorded experience and the new problem. The strategy inspired by this analysis is to use the new problem rather than the recorded experience to guide adaptation when a mismatch occurs. Like other derivational analogy algorithms, REMAID uses a from-scratch problem solver to address goals that cannot be addressed by reusing goals from the recorded experience. However, unlike other algorithms, REMAID also uses a from-scratch problem solver to select goals that can be used to guide the modification of the recorded experience to adapt to mismatches.

The appendix presents a detailed example of REMAID's handling of a mismatch that arises during replay.

# 4 Empirical evaluation

This section presents empirical data demonstrating the performance of REMAID and the other implemented strategies for adaptation and recovery when applied to problems in the domain of automated human interface design. The data collected support several hypotheses. Some are unsurprising: derivational analogy is generally an effective technique, and increased flexibility in adapting recorded experience to new problems increases both efficiency and autonomy. More informatively, the data also indicate that calling a from-scratch problem solver to help adapt to mismatches can be a successful strategy, but that there may be a trade-off between efficiency and autonomy unless the execution of the derivational analogy program and the execution of the from-scratch problem solver are intelligently interleaved.

## 4.1 The experiment

All of the programs in this experiment use interface design problems solved by the MAID program for their recorded experiences. The MAID program is currently capable of designing five interfaces to the data manager application.[10] Since each design can be used as the recorded experience for any other design, there are 25 possible derivational analogy problems, including five trivial cases where the same interface is designed in both the

[10]This is limited by the amount of knowledge entered about real-world entities in the knowledge base, not by any inherent limitations of the MAID program.

recorded experience and the new problem.

Of the eight implementations presented in section 3, seven reuse experience recorded while solving one problem to help solve a different problem. In this section, the implementations based on strategies in the literature (HOBART, PROBART, POSSIBLY, PROBABLY, BRIAR, and PYRE, summarized in table 1) are referred to as "the basic six" derivational analogy algorithms; the seventh program is REMAID. These seven programs were run on all 25 derivational analogy problems. Data were collected from the 20 non-trivial cases on the total amount of effort expended in solving each problem, the proportion of each problem addressed by reusing recorded experience, and the proportion of effort that was useful in solving each problem.[11] The eighth program is an implementation of the strategy in Hickman's internal analogy approach, which reuses goals within the new problem. This algorithm was run on all five designs, and similar statistics were gathered.

Those programs (POSSIBLY, PROBABLY, BRIAR, and PYRE) that rely on a from-scratch problem solver used MAID.

### 4.2 Results: autonomy

Table 2 quantifies the autonomy of the various programs, that is, the amount of the new problem that each program addresses through reuse, rather than through from-scratch problem solving. For each program, the data in this table show the average proportion of the 20 derivational analogy problems that was addressed through reusing recorded experience and how much was addressed by calling a from-scratch problem solver. The programs are sorted in increasing order of autonomy, with MAID being the least autonomous in terms of reuse (since it solves the entire problem through from-scratch problem solving).

The data presented here indicate that, in terms of autonomy, the PRODIGY-inspired recovery strategy[12] as exemplified by POSSIBLY is about as effective as the PRODIGY-inspired local adaptation strategy[13] as exemplified by PROBART and PYRE. Taken together, the two strategies increase autonomy still more. By being sensitive to the different kinds of mismatches,

[11] Because of irrelevant technical details, meaningful CPU times were unavailable. Instead, counts of the number of goals addressed and the numbers of rules used and reused were collected.

[12] Exhaust all reusable goals, address a goal with the from-scratch problem solver, then try reuse again.

[13] Try alternate problem solving rules each time a mismatch occurs and attempt to start reuse again immediately.

| *Program* | *% Of Total Goals Addressed* | |
|---|---|---|
| | *% Addressed By Reuse* | *% Addressed By MAID* |
| MAID | 0% | 100% |
| LASH | 20 | 80 |
| HOBART | 43 | 0 |
| BRIAR | 43 | 57 |
| PROBART | 63 | 0 |
| PYRE | 63 | 37 |
| POSSIBLY | 64 | 36 |
| PROBABLY | 75 | 25 |
| REMAID | 76 | 24 |

Table 2: Autonomy results for each of the programs when run on 20 derivational analogy problems (note that HOBART and PROBART do not always finish solving the problems).

particularly horizontal detours, the REMAID algorithm is able to perform as well as the best algorithms in terms of autonomy.

The results for the strategy of simply skipping any goals that are not reusable, as exemplified by HOBART and BRIAR, indicate the number of reusable goals *between problems* that are easily found in this domain. Finally, the LASH program uses the same adaptation strategies as REMAID; its poor performance is simply due to the lack of regularity *within* problems in the interface design domain.

### 4.3 Results: efficiency

Since one goal of reusing recorded experience is to improve efficiency, it would be plausible to infer that the most autonomous programs are also the most efficient. However, this is not the case. Table 3 quantifies the average amount of effort that each program spends solving the interface design problems, as well as the proportion of that effort that is productive (as opposed to effort spent trying rules that are not applicable, *etc.*) For comparison, the programs are again presented in order of increasing autonomy, along with their place in order of efficiency.

These data indicate the different amounts of autonomy, that is the different proportions of the new problem that can be addressed by reuse, afforded by different interleaving strategies. In general, more goals in the

| *Program* | *Total Effort Expended (as a % of MAID's Effort)* | *Useful Effort Expended (as a % of Total Effort)* | *Efficiency Rank* |
|---|---|---|---|
| MAID | 100% | 20% | 5 |
| LASH | 112 | 18 | 6 |
| HOBART | 15 | 25 | N/A |
| BRIAR | 58 | 23 | 3 |
| PROBART | 12 | 46 | N/A |
| PYRE | 35 | 28 | 2 |
| POSSIBLY | 234 | 9 | 7 |
| PROBABLY | 83 | 11 | 4 |
| REMAID | 35 | 47 | 1 |

Table 3: Efficiency results for each of the programs when run on 20 derivational analogy problems (note that HOBART and PROBART do not always finish solving the problems, so their rank is omitted).

new problem are addressed through reuse when the entire recorded experience is re-examined every time the from-scratch problem solver is called to address a goal (as it is in the POSSIBLY and PROBABLY algorithms). Using this approach, however, it may be very expensive to determine that the from-scratch problem solver must be called again, and the from-scratch problem solver often must be called several times in a row. One way to avoid this inefficiency is to abandon the recorded experience as soon as the easily reusable goals are exhausted, as BRIAR and PYRE do; however, this sacrifices autonomy.

The REMAID system uses the from-scratch problem solver not only to address goals in the new problem, but also to provide guidance about how to adjust the recorded experience. By intelligently interleaving reuse and calls to a from-scratch problem solver, REMAID's adaptation strategy handles mismatches both autonomously and efficiently.

REMAID's advantage over the other programs is not uniform across all of the derivational analogy problems, though. At one extreme, the basic six programs expend far less effort in the degenerate case where the same design is used for both the recorded experience and the new problem.[14] On

[14]The reason for this is that REMAID's approach is capable of using one goal from the recorded experience to select multiple goals from the new problem. While this strategy is

the other hand, REMAID is much more efficient at handling vertical detours, since the other strategies either halt reuse and turn to from-scratch problem solving or pursue a very expensive recovery strategy.

Given that there are crossover points between the efficiency of REMAID and the efficiency of other programs, it may sometimes make more sense to use a simpler derivational analogy strategy when the recorded experience and the new problem are very similar in size and topology. However, determining these crossover points is an empirical issue for each domain, and it may be impossible to tell how similar two problems are without actually attempting reuse.

### 4.4 Finding the crossover points

To further understand the performance of the various derivational analogy programs, a second application was described in the MAID formalism and used to run a set of experiments similar to those described above. This application allows users to browse through records of inventory that are checked in or checked out, specify a particular item to be checked out, check out an item, or check in an item. In addition to an interface using no real-world characteristics, three interfaces were designed using characteristics from a library (*e.g.*, books and shelves), a rental car company (*e.g.*, an express check-in form and drop box), and a video rental store (*e.g.*, a membership card and check-out form).

Since there are four possible interface designs that MAID can produce for the reservation application, there are 16 possible derivational analogy problems including the four trivial cases of using the same design for both the previous experience and the new problem. Three different descriptions of this application were constructed embodying varying degrees of similarity among the four possible interfaces. As a result, the experiments determine just how much adaptation is required for the flexibility of the REMAID program to make up for the additional overhead of that algorithm.

### 4.5 Experimental conclusions

The data from the experiments using the reservation application are described in detail elsewhere [Blu90a]. This section summarizes the main

capable of detecting horizontal detours, it is also slightly more expensive than the other approaches. Generally, REMAID's efficiency at adapting to mismatches outweighs this expense.

conclusions.

Although there are cases where simpler derivational analogy programs perform more efficiently than REMAID, the more important point is that REMAID shows much less variance in efficiency over the range of examples. While it may not perform as well as the simpler programs when there are very few mismatches, the advantage of using the simpler programs in these cases is not nearly as great as the advantage of using REMAID when there are a larger number of mismatches.

A rough calculation based on the empirical data indicates that if vertical detours are the only kind of mismatch encountered, then REMAID gains the advantage when the number of extra goals it can reuse (compared to PYRE and BRIAR) is about one-fifth of the total number of goals. Furthermore, the data indicate that if pretours are the only kind of mismatch, then REMAID gains the advantage when there are about twice as many goals in the recorded experience as there are in the new problem. This situation may not be terribly common, but it does occur, especially if the experience recorded while solving a design problem is reused while solving a subset of a similar design problem.

## 4.6 Discussion

The foremost lesson of the empirical data is that in terms of efficiency, derivational analogy is a successful technique. Although some of the implementations employ very simple adaptation and recovery strategies, all but one of the programs using previous experience expend less effort than the MAID problem solver. Even the simplest of the basic six derivational analogy approaches, HOBART, reuses over 40% of the design goals.

Most derivational analogy programs have been empirically tested against some from-scratch problem solver, but very little empirical data have been published that compare various approaches in the same domain to determine their strengths and weaknesses. The data presented here indicate that a flexible technique for adapting to mismatches between a recorded experience and a new problem increases both the efficiency and autonomy of derivational analogy. Further, using a from-scratch problem solver for recovery from mismatches is a promising technique, but there is a trade-off between efficiency and autonomy unless a program can intelligently interleave reuse and calls to the from-scratch problem solver.

A number of strategies embodied in the REMAID program contribute to the efficiency and autonomy of its design process. By using the from-scratch

problem solver to help guide the adaptation of the recorded experience, REMAID can adapt to vertical detours. Because REMAID can find more goals to reuse than programs that cannot adapt to vertical detours (and because REMAID is intelligent about interleaving from-scratch problem solving and reuse), this additional autonomy also increases REMAID's efficiency.

By using the from-scratch problem solver's goal selections as indices into the recorded experience to determine where reuse should be restarted, REMAID modifies the order of the recorded goals in response to mismatches between the recorded experience and the new problem. In this way, REMAID adapts to pretours more efficiently than other programs. By combining this technique with its strategies for handling detours, REMAID can adapt to combinations that no single strategy can cope with, thereby improving efficiency and autonomy.

# 5 A fundamental limitation and a proposal

The discussion thus far concentrates on the capabilities of the various derivational analogy algorithms. This section focuses on a fundamental, inherent limitation of the derivational analogy technique as it is currently being pursued in the research. Simply put, the problem is how to integrate innovation with the reuse of experience [Blu90a].

This is not an implementation problem, but rather a problem with the fundamental behavior of derivational analogy. The reason for this is that the derivational analogy technique attempts to minimize problem solving effort by attempting to solve a new problem in a way that is as similar to a recorded experience as possible. The more closely a derivational analogy algorithm follows its recorded experience, the more likely it is to overlook mismatches between the recorded experience and the new problem. Conversely, the more sensitive the derivational analogy algorithm is to such mismatches, the less efficient it will be in reusing its experience.

An example might make this problem a bit clearer. In the BOGART/VEXED system for circuit design [MB87], if the previous experience includes a goal specification like (NOT (EQUAL A B)), a designer might have chosen to use the NOT-DECOMP rule to decompose this into a module implementing (EQUAL A B) and an inverter that takes the the output of that module as input.

If a subsequent circuit design problem contains (NOT (AND A B)) in the same context, BOGART will successfully apply the same NOT-DECOMP

rule which will decompose the goal into a module for (AND A B) and an inverter. Although this looks like a successful use of experience, a from-scratch design might have preferred to use the NOT-AND-DECOMP rule, which uses a NAND gate to implement goals of the form (NOT (AND ...)).

This illustrates how a program might produce an inferior solution by concentrating on reusing past decisions without attending to the important differences between the recorded experience and the new problem. In some task domains, such as matrix manipulation, all correct answers are of equal quality. However, in other task domains, such as human interface design, the path a designer takes to a solution directly affects the quality of that solution.

To address this fundamental problem with derivational analogy, Carbonell [Car86] suggests pursuing an *optimizing approach* (as opposed to the satisficing approach described in section 3.2) by having the derivational analogy algorithm attend to the justifications for the decisions in the recorded experience. When a decision in the recorded experience was arbitrarily made, he suggests exploring alternatives in the new problem; when a decision led to a failing path, he suggests checking the reasons for failure to see if they exist in the current situation.

The problems with pursuing the optimizing approach are twofold. First, in many domains optimality is a global characteristic of a problem solution; it cannot be preserved by simply attending to decision criteria that are local to each step [Kam90a]. Even if a derivational analogy algorithm is given optimal recorded experience, any changes to that experience (such as the changes necessary for adaptation to a new problem) may violate the global optimality criteria, regardless of what local criteria are attended to.

The second, more practical problem with pursuing the optimizing approach is the cost involved. To explore the paths that failed previously, a derivational analogy algorithm must examine all of the preconditions that caused a path to fail previously. The algorithm must also examine all of the preconditions that succeeded previously, to verify that none of them have been rendered unsatisfiable by some mismatch in the new problem. The union of these sets of preconditions is exactly that set of preconditions that a from-scratch problem solver would check if derivational analogy were not being used. Thus, even ignoring the expense of retrieving and matching a recorded experience for reuse, the algorithm would not gain any efficiency by pursuing the optimizing approach.[15]

[15] In the case where a failure detected in a subtree of a goal tree can be propagated up

To overcome the fundamental limitation of derivational analogy, a program must detect when a new problem differs from a recorded experience in some significant way and then adapt to these differences. Most current programs for derivational analogy have simple approaches to these two steps. First, their criterion for deciding when a new problem differs from a recorded experience is simply that reuse fails. This criterion may detect that a solution path is a dead end, but it will not detect that a solution path is not optimal. Second, programs cope with differences either by skipping them or calling a from-scratch problem solver.

We propose an alternative in which the derivational analogy program learns new rules of the form: "When you notice a particular difference between the recorded problem specification and the new problem specification, and you encounter a particular kind of problem solving choice, then here are some additional rules that you want to attend to, regardless of whether they are in the recorded experience." These rules would augment, not replace, the problem-solving knowledge recorded in the program's prior experience. Using these new kinds of rules, a derivational analogy program could both detect differences between problems that would not necessarily hinder reuse and produce better quality solutions.

There are two representations that a program would need in order to accomplish this sort of learning. The first would be a representation for the differences between two problems. In the domain of circuit design, this would be differences between the number and kind of logical operations called for. In the domain of interface design, this might be differences in the number and type of entities in the application description and the real-world entity description.

The second representation needed would capture the differences between two corresponding steps in two separate problems. This could simply be the differences in what rules were recorded as useful in each problem.

The algorithm might learn these new kinds of rules by running a derivational analogy algorithm on a new problem using a variety of recorded experiences, and by running a from-scratch problem solver on the same new problem. The solution produced with derivational analogy would then be compared to the solution produced by the from-scratch problem solver. Differences in the solutions would be noted and, where possible, propagated

---

to a decision made at a goal higher up the tree, the optimizing approach may increase efficiency. However, it is unclear whether the increase outweighs the extra effort. In any case, this technique cannot be used when the domain includes rules with consequents conditional on their antecedents as described elsewhere [Blu90a].

back up to difference links between the problem specification for the recorded experience and the problem specification for the new problem.

Continuing the circuit design example, the program would compare the rule used to address the (NOT (AND ...)) problem using derivational analogy and a recorded experience of (NOT (EQUAL ...)) with the rule used by a from-scratch problem solver (*e.g.*, a person). The program would note that the two rules were different and would determine that the rule used by the from-scratch problem solver (NOT-AND-DECOMP) was not used in the recorded experience because the EQUAL failed to satisfy one of the preconditions. This occurrence of the EQUAL could be traced up to the problem specification, and the difference between it and the AND could be used to build the new rule: "When you notice a difference between an EQUAL operation in the recorded problem specification and an AND operation in the new problem specification, and you encounter a use of the NOT-DECOMP rule involving that EQUAL in the recorded solution, then you should consider using the NOT-AND-DECOMP rule instead."

Some first steps have been made in this direction in the work of Hickman and Lovett [HL91] and Veloso [Vel90]. Hickman and Lovett's approach to derivational analogy relaxes the constraint for considering the recorded experience to be appropriate to reuse. Veloso's approach learns to select the most appropriate recorded experience. However, neither of these programs focuses on learning to improve the adaptation strategy itself.

# 6 Conclusions

This paper has presented both an analytic and an empirical examination of derivational analogy. The basic derivational analogy algorithm has been outlined, and the kinds of mismatches between recorded experience and new problems (detours, pretours, and combinations) have been described in detail, along with the particular difficulties that each kind of mismatch presents to a derivational analogy algorithm.

Various strategies for dealing with these mismatches are presented in the literature. The research presented in this paper has extracted these strategies and implemented them so that they can be applied to a common domain. These implementations have been run on a number of different problems involving the reuse of experience, and several conclusions have become apparent from the resulting empirical data. Among these conclusions are that derivational analogy is a generally successful technique and that us-

ing a from-scratch problem solver to assist a derivational analogy program may increase both efficiency and autonomy. However, care must be taken to intelligently interleave reuse with from-scratch problem solving, or the resulting program will be less efficient than from-scratch problem solving.

The REMAID algorithm for coping with mismatches has been shown to be successful because it intelligently interleaves reuse with from-scratch problem solving. It does this by adapting the recorded experience to the new problem rather than *vice versa* and by recognizing different kinds of mismatches and adapting to them appropriately. To do this, it uses a from-scratch problem solver in a novel way, to aid in recognizing different kinds of mismatches as well as to help adapt to detours.

The empirical data show that the REMAID system exhibits better performance when there are significant mismatches between the recorded experience and the new problem. The data further show that although it involves more overhead, the REMAID system shows less variance in its performance than other derivational analogy programs.

Despite this success, REMAID, like all existing derivational analogy algorithms, succumbs to a trade-off between efficiency and solution quality, although to a lesser extent than other algorithms. This trade-off has been shown to be not just an implementation problem, but a fundamental limitation of static derivational analogy adaptation strategies. The next step is to develop algorithms that can improve their adaptation strategies as they gain (meta-)experience in reusing recorded problem solutions.

## Acknowledgements

We would like to thank Jack Mostow, Angela Hickman, Subbarao Kambhampati, and Raymond Mooney for discussions and comments on this work. We would like to thank Liane Acker and Jeff Rickel for comments on early drafts of this paper. Finally, we would like to thank two anonymous reviewers for many helpful suggestions.

## References


[Bax90] Ira D. Baxter. Transformational maintenance by reuse of design histories. In *Proceedings of Fourth International Workshop on Computer-Aided Software Engineering*. IEEE Computer Society, December 1990.

[BH91] S. Bhansali and M. Harandi. Synthesizing unix shell scripts using derivational analogy: An empirical assessment. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 521–526, 1991.

[Blu90a] Brad Blumenthal. *Applying Design Replay to the Domain of Metaphoric Human Interface Design*. PhD thesis, University of Texas Artificial Intelligence Lab, Austin, TX, 1990. (Available as Technical Report AI-90-145).

[Blu90b] Brad Blumenthal. Incorporating metaphor in automated interface designs. In *Proceedings of the Third IFIP Conference on Human-Computer Interaction*, Cambridge, England, 1990.

[Blu90c] Brad Blumenthal. Strategies for automatically incorporating metaphoric attributes in interface designs. In *Proceedings of the Third User Interface Software and Technology Workshop*, Snowbird, Utah, 1990.

[Car83] Jamie G. Carbonell. Derivational analogy and its role in problem solving. In *Proceedings of the National Conference on Artificial Intelligence*, pages 64–69, Washington, DC, 1983.

[Car86] Jamie G. Carbonell. Derivational analogy: A theory of reconstructive problem solving. In R. S. Michalski, Jamie G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume II, chapter 14. Morgan Kaufmann, Los Altos, CA, 1986.

[CV88] Jamie G. Carbonell and Manuela Veloso. Integrating derivational analogy into a general problem solving architecture. In *Proceedings of the First Workshop on Case-Based Reasoning*, 1988.

[DM86] Gerald DeJong and Raymond Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2), 1986.

[Gen83] Dedre Gentner. Structure mapping: A theoretical framework for analogy. *Cognitive Science*, 7, 1983.

[HA87] Michael N. Huhns and Ramon D. Acosta. Argo: An analogical reasoning system for solving design problems. Technical Report AI/CAD-092-87, Microelectronics and Computer Technology Corporation, Austin, TX, 1987.

[Ham90] Kristian Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45(1-2), 1990.

[HHN86] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. In Donald A. Norman and Stephen W. Draper, editors, *User-Centered System Design*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[HL90] Angela Kennedy Hickman and Jill Larkin. Internal analogy: A model of transfer within problems. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, Cambridge, MA, 1990.

[HL91] Angela Kennedy Hickman and M.C. Lovett. Partial match and search control via internal analogy. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, Evanston, IL, 1991.

[Kam89a] Subbarao Kambhampati. Control of refitting during plan reuse. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989.

[Kam89b] Subbarao Kambhampati. *Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach*. PhD thesis, Computer Vision Laboratory, Center for Automation Research, University of Maryland, College Park, MD, 1989.

[Kam90a] Subbarao Kambhampati. Personal communication at the Eighth Annual National Conference on Artificial Intelligence, 1990.

[Kam90b] Subbarao Kambhampati. Mapping and retrieval during plan reuse: A validation structure based approach. In *Proceedings of the Eighth Annual National Conference on Artificial Intelligence*, Boston, MA, 1990.

[Kol87] Janet Kolodner. Extending problem solver capabilities through case-based inference. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 167–178, 1987.

[LRN86] John Laird, Paul Rosenbloom, and Alan Newell. Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1), 1986.

[MB87] J. Mostow and M. Barley. Automated reuse of design plans. In *Proceedings of the 1987 International Conference on Engineering Design*, pages 632–647, 1987. (Available as Rutgers University Technical Report ML-TR-14).

[MF89] J. Mostow and G. Fisher. Replaying transformational derivations of heuristic search algorithms in DIOGENES. In *Proceedings of the Second Workshop on Case-Based Reasoning*, pages 94–99, Pensacola, FL, May 1989.

[MKKC86] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[Mos89] Jack Mostow. Design by derivational analogy: Issues in the automated replay of design plans. *Artificial Intelligence*, 40(1-3), 1989.

[PBH90] Bruce Porter, E. Ray Bareiss, and Robert Holte. Concept learning and heuristic classification in weak-theory domains. *Artificial Intelligence Journal*, 45(1-2):229–263, 1990.

[VC89] Manuela Veloso and Jamie G. Carbonell. Learning anologies by anology – the closed loop of memory organization and problem solving. In *Proceedings of the Second Workshop on Case-Based Reasoning*, 1989.

[Vel90] Manuela Veloso. Replaying past experience within a general problem solving and learning architecture. Technical report, Carnegie-Mellon University, Pittsburgh, PA, 1990.