University of Wisconsin Computer Sciences Technical Report 858 (June 1989)

Using Explanation-Based Learning to Acquire Programs by Analyzing Examples*

Richard Maclin
Jude W. Shavlik

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706

maclin@cs.wisc.edu
(608) 262-6613

Abstract

A number of problems confront standard automatic programming methods. One problem is that the combinatorics of search make automatic programming intractable for most real-world applications. Another problem is that most automatic programming systems require the user to express information in a form that is too complex. Also, most automatic programming systems do not include mechanisms for incorporating and using domain-specific knowledge. One approach that offers the possibility of dealing with these problems is the application of *explanation-based learning* (EBL). In the form of EBL used for this project, *explanation-based learning by observation*, the user enters a description of a specific problem and solution to that problem in a form comfortable to him or her. Using domain-specific knowledge, the system constructs an explanation of the solution to the problem using the actions of the user as guidance. Next, the goal stated by the user is generalized with respect to any domain information about possible goals of actions performed by the user in the solution. Then the explanation is reconstructed with respect to the generalized goal. Finally, the explanation is transformed into a general solution which can be used to solve problems that are conceptually similar to the specific problem presented. This approach promises to overcome the problems with standard automatic programming methods discussed above.

* This research was partially supported by a grant from the University of Wisconsin-Madison Graduate School.

Introduction

Automatic programming has long been a goal of artificial intelligence (AI) researchers [Rich86]. After peaking in the mid-70's this research has been slowed by three critical problems. One, the combinatorics of search have prevented the developed techniques from being applied to realistic programming problems. Even in a system with complete knowledge of a programming language and a perfect specification of its construct's preconditions and effects, the process of designing a program can be computationally intractable. Two, specifying the desired behavior of the resulting program completely and unambiguously is a complicated task. Three, the need to incorporate and utilize large amounts of domain-specific knowledge has been recognized, but no satisfactory method to do so has been developed.

Machine learning researchers, especially those in *Explanation-Based Learning* (EBL), address issues that directly relate to the problems that have hindered automatic programming research. In EBL [DeJong86, Mitchell86], a specific problem's solution is generalized into a form that can be later used to solve conceptually similar problems. The generalization process is driven by the explanation of why the solution worked. Knowledge about the domain allows the explanation to be developed, and then generalized, thereby producing a general algorithm from the solution to a specific problem.

The PLEESE (Program Learning by Explaining External Solutions of Examples) project applies EBL techniques to the problem of automatic programming. Specifically, PLEESE uses a form of EBL called *explanation-based learning by observation*. In this form of EBL, an external agent provides the learning system with a description of a specific problem and a solution to the specific problem. The system then examines the solution and uses knowledge about the domain to construct and explanation of the solution. From this the system produces a general solution which can be applied to conceptually similar problems and which can be used in constructing more complex problems and solutions.

PLEESE is designed to be simple to use. A user presents an initial state consisting of programming constructs (arrays, stacks, variables, registers, etc.) and values stored in those constructs. An example of the program desired is demonstrated by performing actions known to the system (adding two values together, zeroing some value, etc.) on the initial and subsequent states until some final state is reached. The user either explicitly presents a goal or the system assumes that the changes resulting in the final state are the goal. The system then explains the solution using domain knowledge about the actions selected by the user. This is done by constructing a proof of the goal. The system may also try to generalize the goal if none was presented by the user and reconstructs the explanation with the new goal in mind. Finally, the system generalizes the goal to produce a solution that can be used to solve conceptually similar problems. This approach allows PLEESE to do automatic programming and deal with the problems discussed above.

The EBL approach allows PLEESE to restrict the amount of search performed. The solution sketch provided by the user is used as an outline to construct the explanation of the specific problem. Automatic theorem-proving systems have to construct proofs blindly, thereby preventing their use on all but the simplest of problems. PLEESE uses the solution sketch presented by the user as the outline for the explanation it constructs. The system focuses on verifying the effects of the actions taken by the user and filling in any details not included by the user. The solution sketch acts as a strong bias to greatly reduce the search space that must be explored by PLEESE.

PLEESE also limits the search by focusing on learning as much from the specific case as possible rather than trying to understand the whole of the general case. This is because the solution to the specific case may be much simpler than for the general case. The system attempts to learn as much as possible by generalizing the specific problem. This involves examining the explanation of the solution and relaxing the constraints on the explanation to form a solution to a more general class of problems. If the user chooses a good specific problem then the system will produce a solution covering a large number of similar problems.

The PLEESE approach also allows the user to present the information required by the system in a way that is more natural for the user. The user expresses the solution in terms designed for the user's ease, and then the system reformulates this description to produce a more general form that is convenient for the system. For example, since the language is targeted for human users it may not state details that can be deduced or may be ambiguous. In these cases the system will use its domain knowledge to augment the user's description with the missing details. For example, a user might demonstrate a block problem by manipulating a graphic representation of the blocks with a mouse. The pre- and post-conditions of each action (picking up a block, putting down a block) would be filled in by the system from its knowledge about the domain of blocks to create a complete explanation of the actions.

The user may have also left out details of or omitted entirely the goal of the problem. In these cases the system examines the actions in the solution to determine the effect(s) of these actions. The system then uses domain knowledge about actions and their effects to complete the goal of the system. For instance, if the user performed a set of sequential actions, such as examining each of the elements of an array in turn, the system would recognize this as a case of iteration, and would reform the goal with respect to the iteration. The system would then use the extended goal to reform the explanation with respect to the goal. Once the explanation of the solution has been completed the system then generalizes the explanation.

Finally, the PLEESE approach represents a method to use domain knowledge. At each step of the process, PLEESE is augmenting or generalizing the situation presented by the user with respect to the domain knowledge about programming in the system. Domain knowledge is used to construct and augment the explanation of the solution, in generalizing the goal presented by the user, and in generalizing the explanation of the solution.

The major focus of the PLEESE project is the process of going from explanations of specific solutions to useful general-purpose algorithms. This project builds on previous work on generalizing the structure of explanations [Shavlik88]. Most research in EBL involves relaxing constraints on the items in a specific problem's explanation. The internal structure of the explanation itself is not generalized. However, this precludes the acquisition of concepts where a general iterative or recursive process is implicitly represented by a fixed number of applications in the specific problem's explanation. Since programming so heavily involves iteration and recursion, extensions of the algorithms developed for the BAGGER system [Shavlik89] are particularly appropriate for efficiently learning computer programs. PLEESE generates explanations which are used as input to the BAGGER system. The general rules produced by BAGGER can then be converted into general programs.

Following is a short description of standard EBL techniques along with techniques for generalizing explanation structures and a discussion of a method for determining the goal of a user in a problem situation. Next there is a description of an initial implementation of the PLEESE approach and an example of how it works. This is a followed by a description of other work in automatic programming similar to the PLEESE approach. The paper concludes with a short discussion of further research topics.

An Overview of Explanation-Based Learning by Observation

In an *explanation-based learning by observation* (EBLO) system, a sketch of a solution to a specific problem is presented by a user. This sketch is used to guide the construction of an explanation of the solution. This section provides a description of how an explanation is generated, and how the explanation is then generalized in an explanation-based learning system. In EBLO, knowledge about a domain is used to construct an explanation of how a specific problem was solved by the user. Once the system has an understanding of how the solution steps interact to solve a problem, as well as knowing what upon what facts and assumptions they depend, the EBLO system can generalize the solution technique. The resulting generalization can then be used to solve conceptually similar problems.

One common method of representing domain knowledge is in the form of predicates. In this representation an explanation takes the form of a proof of the goal of the specific example. EBLO systems avoid the combinatorial explosiveness of theorem-proving by using the sketch of the solution provided by the user. The sketch is used as an outline by the theorem-prover to greatly limit the amount of information the system has to consider in constructing the proof. The actions of the user are annotated with domain knowledge to construct the explanation.

An interesting question in an EBLO system is whether the user must enter a complete goal as part of the description of the problem. Since construction of a predicate goal can be extremely costly it would be useful if the system was not only able to complete the solution of the user, but also if the user was able to finish an incomplete goal presented by the user. A first approximation of this process is to determine the effects of the actions of the user. The effects are determined by examining the domain knowledge about the actions of the user. The system then determines if the goal presented accounts for all of the effects. If not, the goal can be annotated with the unaccounted for effects. This process is extremely limited though if there is more than one action, since no relationship between the actions is determined. A further process would be to examine the effects and actions of the user to determine some structure for the actions. An example is the recognition that the actions of adding each member of an array to a register might be the result of an iterative process. This structure is compared against a domain of possible structures, and the goal augmented for that structure (e.g. adding an iterative structure for the adding-array-elements goal). The explanation is then reconstructed with respect to the more specific goal. This explanation would then be used as an input to an EBL algorithm.

EBL algorithms take an explanation of a specific problem and produce a general method for solving similar problems. This process is done by converting constants in the original explanations into constrained variables and then unifying the variables to retain the interactions of the specific problem (see [DeJong86, Mitchell86] for details). These algorithms produce a rule that can be used to efficiently solve similar problems. Some EBL algorithms also generalize the structure of their explanations. The BAGGER algorithm [Shavlik87, Shavlik88] is an example of a system that generalizes structure. BAGGER recognizes cases of implicit recursion or iteration in its explanations and produces solutions containing that recursion or iteration. For example, if the specific problem presented is a solution for stacking 3 blocks, a standard EBL algorithm would produce a solution for stacking *any* 3 blocks. BAGGER instead would produce a solution for stacking any *N* blocks, thus providing a more general solution. The general solution from the BAGGER algorithm can then be used by the PLEESE system to solve similar problems and as a subpart of more complex problems.

The PLEESE Approach

The goal of the PLEESE approach is to observe users solve specific problems and to produce general programs that solve similar problems. This approach uses the EBL techniques discussed in the last section to generalize the specific problem presented by the user. The process of generalization of the explanation (including structural generalization) is performed by the BAGGER system [Shavlik89]. An overview of the PLEESE approach is presented in figure 1, and the following is a description of the steps of the process.

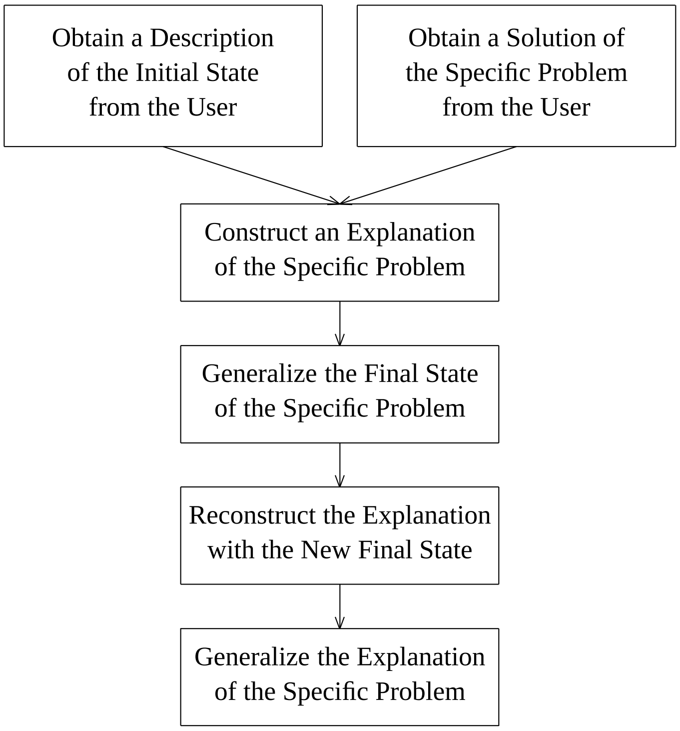

Figure 1. Overview of PLEESE

Step 1. Obtain a Description of the Initial State from the User. The user defines the set of constructs, initial values and possibly the goal for a specific problem. Objects and values are defined as predicates. These initial objects and values together represent the initial state of the problem.

Step 2. Obtain a Solution of the Specific Problem from the User. The user demonstrates the solution to the problem by performing actions on the initial state. The solution to the problem is a record of the set of actions performed along with the states of the problem. Each action has a corresponding rule(s) determining its effect in terms of predicates. These rules are used to produce each of the successive states of the problem. The goal state of the problem is simply the final state produced.

Step 3. Construct an Explanation of the Specific Problem. Knowledge about the domain of programs is used to construct an explanation of the specific solution. The solution sketch (the set of actions performed) provided by the user is used as an outline for the explanation. This outline is then filled in by PLEESE using the domain knowledge (the rules describing the results of user actions). Initially, if no goal of the problem is provided, then the goal is construed to be the changes resulting from the initial state to the goal state. Then an explanation in the form of a proof of the goal is constructed (whether the goal is explicitly or implicitly defined).

Step 4. Generalize the Final State of the Specific Problem. Once the initial explanation of the specific problem is finished, PLEESE attempts to generalize the goal of the problem. This is done by searching the explanation tree and reformulating the set of actions using knowledge about a small set of basic programming constructs and methods. For example, if the user entered a set of actions adding each value in an array of four elements to a register the set of actions could be reformulated using construct which stated that every element of the array should be added to the register. This generalization process focuses on a small number of "primitive" reformulations, since even a very small "primitive" set can formulate a large number of possible programs.

Step 5. Reconstruct the Explanation with the New Final State. After the goal is generalized an expanded explanation must be constructed reflecting the new goal. The rules used in the original explanation are augmented to reflect the generalized goal. For example, if the generalized goal were to add all four values of some array to a register then a new adding rule that counted how many values of the array had been added so far would be constructed. (For more on this see the specific example). After the rules have been augmented, the expanded explanation of the problem is constructed using the new rules and the old explanation as a guide.

Step 6. Generalize the Explanation of the Specific Problem. The explanation produced from the generalized goal is used as input to the BAGGER system. The focus of the BAGGER system is the generalization of the structure of explanations. The BAGGER system is thus able to recognize recursive or iterative structures in explanations and to produce generalizations using the recursive/iterative structure. The rules produced by BAGGER can be used to produce code directly.

In each of the steps 3-6 the original solution presented by the user is transformed into a more general form. This makes it possible to produce the most general possible algorithm from the specific problem presented by the user.

An Example

This section contains a simple example to demonstrate the working of the PLEESE system. The specific example presented is the summing of the values in an array of four elements, while the general program desired is one to sum any number of array elements. All of the output reproduced here is produced by PLEESE.

Problem states in PLEESE are represented using prolog-style clauses. For example, the value of an array in a particular state is represented by the clause *(array–value <a> <p> <v> <s>)* which says that position <*p*> of array <*a*> has value <*v*> in state <*s*>. The example used will be one with an array A of four elements 1 to 4 with values 4, 11, 3 and 8 respectively. Predicates to represent this information can be found in table A.1 in the appendix.

The operations performed by the user in demonstrating a specific example are mapped directly to corresponding actions in the representation. The user clears a register so that it could be used to sum a set of values. The process of clearing the register <*r*> is mapped to the action *(clear <r>)*. Similarly, to add an array value to a register, the action is *(add <r> <a> <p>)*

which says that the value of position <*p*> of array <*a*> should be added to the register <*r*>. Each action is defined by a set of rules determining what changes result from a particular action. A *(clear R0)* in state S0 creates a new state S1 in which the value of R0 is now 0. Actions also define what values are NOT affected by a particular action. The actions used in the simple example are *clear*, which zeroes a register value and *add* which adds the value of some array element to the value of a register. Rules defining these actions can be found in table A.2 in the appendix.

Given the initial state S0, the set of actions presented to PLEESE are:

Action	New State
(clear R0)	S1 = (do (clear R0) S0)
(add R0 A 1)	S2 = (do (add R0 A 1) S1)
(add R0 A 2)	S3 = (do (add R0 A 2) S2)
(add R0 A 3)	S4 = (do (add R0 A 3) S3)
(add R0 A 4)	S5 = (do (add R0 A 4) S4)

which says that register R0 is cleared, then A[1] is added to the register, followed by A[2], A[3], and A[4] in that order.

Assuming no goal was defined, the system would examine the effect(s) of the actions. From this examination the system determines the only changes are made to the value of register R0, so the change to R0 is made the initial goal of the problem. PLEESE then constructs an explanation of the changes to register R0. The system uses the set of actions presented by the user as an outline for the explanation. This outline is expanded with domain knowledge contained in the rules defining the results of the actions. This explanation is shown in figure 2. The arrows in the figure represent predicates pointing to their implications. Dotted arrows represent implications involving frame axioms, i.e. proofs demonstrating that some value has not changed since the original state.

The system then examines the explanation and the set of actions and attempts to generalize the set of actions. The effect of the actions is to put (+ (+ (+ (+ 0 A[1]) A[2]) A[3]) A[4]) in register R0 which can be rewritten as (+ 0 A[1] A[2] A[3] A[4]). This form is then compared to a set of computer primitives and replaced with the form (+ 0 (FOR EVERY I IN 1,2,3,4 A[I])). From this generalized form along with the fact that array A only has 4 elements the system is able to produce augmented rules which simulate the FOR EVERY by marking each array element as it is added and counting the number of array elements added. These augmented rules can be found in table A.3 of the appendix.

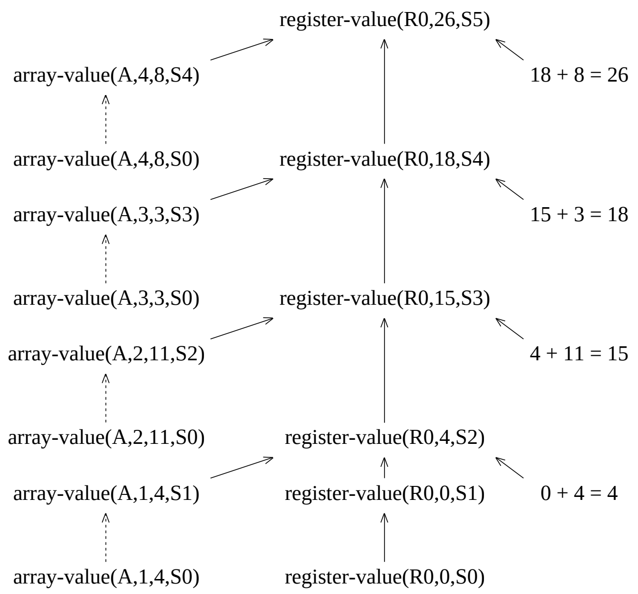

Figure 2. Explanation of the Specific Problem

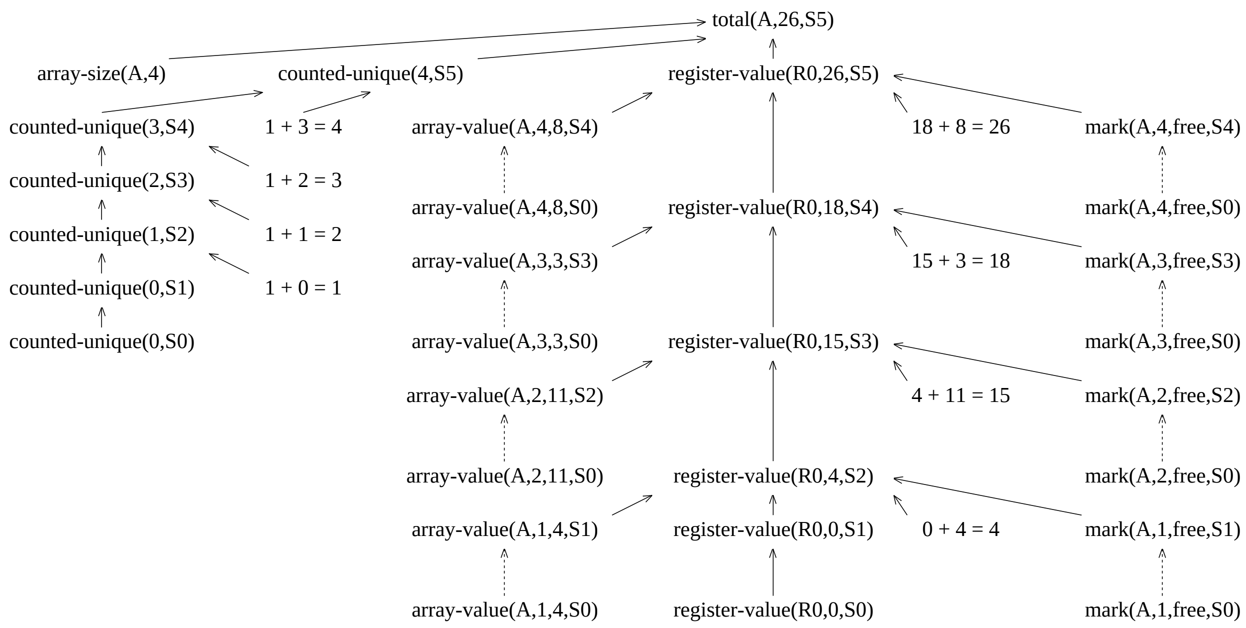

Figure 3. Augmented Explanation of the Specific Problem

The augmented rules are then used to produce a new explanation of the specific problem using the same states. This explanation is shown in figure 3. Note that the only differences between the old and the new explanation are that the new explanation has been augmented with predicates to perform the accounting for the FOR EVERY construct (e.g. predicates to count how many add operations and predicates to mark each element of the array after it has been added).

This general explanation is used as input to the BAGGER2 system which generalizes the structure of the explanation to recognize recursive and iterative elements. The rule produced by BAGGER2 is:

```
(TOTAL ?A ?E ?C) ←
(ARRAY-SIZE ?A ?B)
(CALL COUNTED-UNIQUE-REGISTER-VALUE7 (AND (COUNTED-UNIQUE ?B ?C) (REGISTER-VALUE ?D ?E ?C)))
```

which determines the total of array by determining the size of the array and the calling the recurrence COUNTED-UNIQUE-REGISTER-VALUE7. A recurrence is a recursive rule form produced by the BAGGER2 system. The CALL predicate forms is a special form that takes as argument the name of some rule to call and a form to unify with the consequent of the recurrence (the (AND ..) argument on line 3) in the example. Another predicate used is the MATCH predicate which says that the two arguments should be unified. The recurrence COUNTED-UNIQUE-REGISTER-VALUE7 is defined by the rule:

```
 1: (AND (COUNTED-UNIQUE ?V1 (DO (ADD ?V2 ?V3 ?V4) ?V5))
 2:      (REGISTER-VALUE ?V2 ?V6 (DO (ADD ?V2 ?V3 ?V4) ?V5))) ←
 3: (OR (AND (MATCH ?V5 (DO (CLEAR ?V2) S0))
 4:          (= ?V1 1)
 5:          (ARRAY-VALUE ?V3 ?V4 ?E2 S0)
 6:          (= ?V6 (+ 0 ?E2)))
 7:     (AND (= ?V1 (+ 1 ?E3))
 8:          (OR (AND (MATCH ?V5 (DO (ADD ?E4 ?E5 ?E6) (DO (CLEAR ?E7) S0)))
 9:                   (<> ?V4 ?E6)
10:                   (ARRAY-VALUE ?E5 ?V4 ?E8 S0)
11:                   (= ?V6 (+ ?E9 ?E8))
12:                   (CALL COUNTED-UNIQUE-REGISTER-VALUE7
13:                          (AND (COUNTED-UNIQUE ?E3 (DO (ADD ?E4 ?E5 ?E6) (DO (CLEAR ?E7) S0)))
14:                               (REGISTER-VALUE ?E4 ?E9 (DO (ADD ?E4 ?E5 ?E6) (DO (CLEAR ?E7) S0))))))
15:              (AND (MATCH ?V5 (DO (ADD ?E4 ?E5 ?E6) ?E10))
16:                   (= ?V6 (+ ?E8 ?E11))
17:                   (CALL MARK3 (MARK ?V3 ?V4 FREE (DO (ADD ?E4 ?E5 ?E6) ?E10)))
18:                   (CALL ARRAY-VALUE4 (ARRAY-VALUE ?V3 ?V4 ?E11 (DO (ADD ?E4 ?E5 ?E6) ?E10)))
19:                   (CALL COUNTED-UNIQUE-REGISTER-VALUE7
20:                          (AND (COUNTED-UNIQUE ?E3 (DO (ADD ?E4 ?E5 ?E6) ?E10))
21:                               (REGISTER-VALUE ?E4 ?E8 (DO (ADD ?E4 ?E5 ?E6) ?E10))))))))
```

This recurrence introduces an add action each time it is called recursively. If the count is 1 then a clear action is added and the recursive process terminates (the conjunct on lines 3-6). If the count is not 1, then the count is decreased by 1 and the procedure is called recursively. The rule defines two possible recursive conditions: 1) an unmarked array value is located and its value determined, the rule is then called recursively (the conjunct on lines 15-21 containing the second recursive call to the rule); and 2) if only one more add action is to be performed calculate the result directly (the conjunct on lines 8-14 containing the first recursive call). Note that the first recursive call could be encompassed in the second, but the present system does not look for this type of reduction. Note also that the above recurrence requires two other recurrences MARK3:

```
(MARK ?A ?B FREE (DO (ADD ?V1 ?A ?V2) ?V3)) ←
(OR (AND (MATCH ?V3 (DO (CLEAR ?E1) S0))
         (<> ?B ?V2))
    (AND (MATCH ?V3 (DO (ADD ?E1 ?A ?E2) ?E3))
         (<> ?B ?V2)
         (CALL MARK3 (MARK ?A ?B FREE (DO (ADD ?E1 ?A ?E2) ?E3)))))
```

and ARRAY-VALUE4:

```
(ARRAY-VALUE ?A ?B ?C (DO (ADD ?V1 ?V2 ?V3) ?V4)) ←
(OR (AND (MATCH ?V4 (DO (CLEAR ?E1) ?E2))
         (ARRAY-VALUE ?A ?B ?C ?E2))
    (AND (MATCH ?V4 (DO (ADD ?E1 ?E2 ?E3) ?E4))
         (CALL ARRAY-VALUE4 (ARRAY-VALUE ?A ?B ?C (DO (ADD ?E1 ?E2 ?E3) ?E4)))))
```

generated by BAGGER2. These perform the processes of verifying (through frame axioms) that an array value has not been marked and determining the value of some array position respectively.

Each step involves generalizing the original problem presented by the user.

Related Work in Automatic Programming

The approach taken in PLEESE draws on existing automatic programming methods including automatic theorem proving (e.g., [Good84, Manna86]) and program specification using traces (e.g., [Bauer75, Phillips77]) or examples [Summers77]. PLEESE addresses a number of the problems encountered by these methods.

The PLEESE approach is most like automatic theorem proving in that both involve deriving a proof (explanation) for an algorithm and then using that proof to write a program for the algorithm. One problem automatic theorem proving has is that the process of searching the space of possible proof structures is combinatorially explosive. As the problems addressed become "realistic", the proofs involved become intractable. The PLEESE approach avoids this problem because the process of creating an explanation of a specific problem is significantly constrained by the solution presented by the user.

A second problem with automatic theorem proving is the production of specifications [Balzer77]. Since these specifications are logically oriented, even the simplest of procedures may have a large and unwieldy specification, which the user has to construct and enter. PLEESE addresses this problem by allowing the user to state a specification (goal) implicitly in terms of the actions in the specific problem, and then the system produces a general specification during its generalization of the specific problem's explanation.

Specification using traces was originated from work done on automatic programming methods using examples of *i/o* pairs as specifications. The problem with *i/o* pairs is that for complex operations the amount of search needed to find the proper algorithm is prohibitive. Traces were seen as a method of including information concerning the control structure of the algorithm in the specification. Trace specifications are by nature more costly since the programmer has to give more details about the algorithm, but the tradeoff helps make the search process more tractable.

Automatic programming methods using examples or traces as specification have the advantage that such specifications do not have to be complete (i.e. one example or trace might not define all of the possible outcomes of the algorithm). One difference between most automatic programming systems and PLEESE is that the former tend to focus on presenting enough examples to derive a complete specification of the algorithm. This runs into the problem that a complete specification may require a large number of examples. Such systems are not able to take advantage of situations where a large part of an algorithm can be represented by a small number of examples. PLEESE focuses on the idea of learning as much as possible from a single example and not worrying whether the specification derived is complete. It can do this because it has a domain theory with which is is able to explain how these examples can be solved. Previous approaches were much like similarity-based learning algorithms [Michalski75, Quinlan86], in that both require a large number of training examples due to their inability to explain how single examples are solved. Also, the generalizations they make are unjustified, while generalizations in an EBL system are justified by a domain theory.

Most approaches using traces focus on a subset of the aspects of the algorithm presented. Systems, for example, might focus on flow of control or changes to data structures while ignoring other aspects. Thus these systems would run into problems when attempting to learn algorithms whose actions could not be well expressed in the trace methodology. The PLEESE system allows users to focus on what they believe to be the important aspects of the algorithm and counts on domain knowledge and other information to help it figure out how to fill in the details. Systems using trace methodologies also generally did not focus on proving their programs and therefore are unable to verify the correctness or coverage of the programs produced.

Others ([Hill87, Steier87]) have applied EBL to automatic programming. The EBL algorithms underlying their approaches are not capable of extracting recursive or iterative concepts from examples where the recursion or iteration is implicitly represented.

Current Research Directions

The most obvious extension to the system is to test it on more complex domains. One domain being looked at is sorting programs. Sorting programs have the nice property that while they each share a similar goal, there are many different ways to satisfy that goal. The algorithm learned by the system should be dependent on what type of sorting example is presented to the system. If the user presents a sorting example that shows an insertion sort, the system should learn an insertion sort method, but if the user sorts the example with a quicksort the system should produce a quicksort method. Another domain being investigated is graph algorithms. Graph algorithms are appealing because it is possible to specify a large number of very different algorithms (connectivity, depth-first search, matching, etc.) using a small number of constructs and operators (nodes, edges, visits to nodes). Graph algorithms also offer the possibility of designing a graphical interface for the user to demonstrate the programs, which should greatly reduce the task of the user.

The ability to infer the goal of the user when information about the goal is incomplete or missing is crucial. The goal of a problem expressed in terms such as predicate logic grows exponentially with the complexity of the program. This would make entry of such a goal by the user unacceptable. An alternative is to focus on inferring the goal from information provided. A simple approach to this problem would be to store a set of known transformations which the actions could be compared to. Each transformation would define how the goal should be generalized if its set of actions is recognized. A problem with this approach is that it is limited by the set of transformations included by the user. This approach is useful if a relatively small number of transformations but in a complex domain the number of different possible transformations may make this solution impossible. A more appealing approach would be to use the explanation of the changes from the initial to the final state to determine the goal. The system could then explore the explanation tree for simple generalizations such as recognizing that some action is repeatedly or recursively applied and making this information explicit in the goal. The system could also interact with the user concerning sections of the explanation that it is unable to generalize. This approach would allow the system to generalize the goal without having to include a large number of domain-specific transformations. That would decrease the amount of information the user would have to provide.

An issue raised by the current system is the role of multiple examples, which is not currently addressed. The system may interpret the solution to a specific example from a viewpoint other than that intended by the user. In addition, the system may under-generalize or over-generalize its explanation of the specific solution. Finally, it may be impossible to present a single example that will cover all of the possible cases. The system is being extended so that in these situations further examples will be used to *refine* rather than replace the existing solution. A simple method to do this might be to allow the disjunction of the rules learned in the further examples. The problem with this approach is that the rules might become to unwieldly or inefficient, which defeats the purpose of the system. A better approach would also generalize the resulting rules to produce single general rules. The use of multiple examples will allow the user to interactively correct solutions with counter-examples and will also allow the user to build large complex programs from multiple simple cases.

Another interesting issue raised in PLEESE is the possibility of automatically parallelizing the code produced. During construction of the explanation of the solution PLEESE identifies dependencies among portions of the explanation produced. Since dependencies among sections of the explanation are known then it should be possible to construct parallel code for those sections that are independent. A more complex problem would be to identify the type of dependency that holds between two sections. Code could then be produced in those cases where the dependencies matched existing compiler techniques for producing parallel code where dependencies are present.

A goal of PLEESE is to provide a mechanism for producing programs that can be used by non-programmers. These people would program by showing how their program would work on some specific cases. One step toward this goal would be the introduction of a more powerful interface based on a graphical and mouse representation of the possible programming constructs. Another step would be to introduce methods to improve the readability of the code produced. Also, the system could introduce more powerful methods of interacting with the user, possibly by including some method for analyzing sections of the code produced for its effects (generating explanations of the interesting section and then providing a means to explain the explanation to a user). The system could provide the user with complex debugging and testing facilities, some method of maintaining a library of test examples, and methods such as the multiple examples method mentioned above for refining the code produced by the system.

Conclusion

The system developed, PLEESE, is an interactive automatic programming system using EBL techniques. The user enters a solution to a specific programming problem. The system uses the solution presented by the user to produce an explanation of the solution using domain knowledge about actions in the solution. The system then generalizes the goal of the problem with domain knowledge about patterns of actions. Finally the system generalizes the explanation which involves both relaxing constraints on the explanation and recognizing situations involving implicit iteration and recursion. The system is heavily dependent on the BAGGER system which takes the general explanation produced by PLEESE and produces a general rule which solves conceptually similar problems.

EBL gives PLEESE a number of advantages over other automatic programming systems. EBL is used by PLEESE to control the amount of search performed in producing general solutions. The system focuses on understanding a user generated solution to a specific problem rather than trying to solve the general case which greatly limits the search space involved. The PLEESE system also focuses on the problem of allowing the user to enter information to the system in a form comfortable to them. Each step of the PLEESE process augments the description presented by the user to produce a general form to solve similar problems. Finally, EBL allows PLEESE to integrate a large amount of domain knowledge in a domain-independent way. The PLEESE approach provides a promising way to simplify the interaction of user and computer in producing programs.

Appendix: Domain Knowledge for Arrays and Registers

The predicates in Table A.1 describe the initial state of the example presented. The initial state consists of the predicates defining the array A and the values of its 4 elements.

Table A.1 Initial State Predicates

Predicate	Description
(array-size A 4)	Array A has 4 elements.
(array-value A 1 4 S0)	A[1] = 4 in state S0
(array-value A 2 11 S0)	A[2] = 11 in state S0
(array-value A 3 3 S0)	A[3] = 3 in state S0
(array-value A 4 8 S0)	A[4] = 8 in state S0

The rules in Table A.2 implement the actions of clearing the value of a register and adding the value of an array element to a register. The rules define how to determine the value of a register or array element in a particular state. Note that the rules describing how to determine the value of an array element are frame axioms - the rules state that the value before and after each action remain the same. These rules are used to produce the initial explanation of the users problem solution.

Table A.2 Original Action Rules

Rule	Description
(register-value ?r 0 (do (clear ?r) ?s)) ← (register-value ?r ?v ?s)	A register has the value 0 after a clear
(array-value ?a ?p ?v (do (clear ?r) ?s)) ← (array-value ?a ?p ?v ?s)	All array values remain the same after a clear action.
(register-value ?r ?v (do (add ?r ?a ?p) ?s)) ← (register-value ?r ?vr ?s) (array-value ?a ?p ?va ?s) ?v = ?vr + ?va	After an add the register holds the sum of old register value plus the array value.
(array-value ?a1 ?p1 ?v (do (add ?r ?a2 ?p2) ?s)) ← (array-value ?a1 ?p1 ?v ?s)	All array values remain the same after an

The rules in Table A.3 are the set of rules produced when the goal is generalized in the example problem presented. Since the example problem's goal was generalized to include the idea that EACH element of the array should be added to the register, the rules are altered to reflect this information. In this case this is done by including rules to mark each array value after it is added to the register and counting how many array elements have been marked. These new rules are then used to produce the second explanation of the problem with respect to the generalized goal.

Table A.3 Augmented Action Rules

Rule	Description
(mark ?a ?p marked (do (add ?r ?a ?p) ?s))	Array element is marked after it is added.
(mark ?a ?p free S0)	An array element is free in the initial state.
(mark ?a ?p1 ?m (do (add ?r ?a ?p2) ?s) ← (mark ?a ?p1 ?m ?s) ?p1 <> ?p2	An array element different from the one added retains its mark in the new state.
(mark ?a ?p ?m (do (clear ?r) ?s)) ← (mark ?a ?p ?m ?s)	An array element retains its mark after a clear action.
(counted-unique 0 S0)	The count is 0 initially.
(counted-unique ?ip (do (add ?r ?a ?p) ?s)) ← (counted-unique ?i ?s) ?ip = 1 + ?i	The count is incremented after each add action.
(counted-unique ?i (do (clear ?r) ?s)) ← (counted-unique ?i ?s)	The count is unchanged by a clear action.
(register-value ?r 0 (do (clear ?r) ?s)) ← (register-value ?r ?v ?s)	A register has the value 0 after a clear.
(array-value ?a ?p ?v (do (clear ?r) ?s)) ← (array-value ?a ?p ?v ?s)	All array values remain the same after a clear action.
(register-value ?r ?v (do (add ?r ?a ?p) ?s)) ← (mark ?a ?p free ?s) (register-value ?r ?vr ?s) (array-value ?a ?p ?va ?s) ?v = ?vr + ?va	If the array element is unmarked the an add sets the register value to the old register value plus the array value.
(array-value ?a1 ?p1 ?v (do (add ?r ?a2 ?p2) ?s)) ← (array-value ?a1 ?p1 ?v ?s)	All array values remain the same after an add action.
(total ?a ?v ?s) ← (array-size ?a ?n) (counted-unique ?n ?s) (register-value ?r ?v ?s)	For an array of size n add n different the result is the final register value.

References

[Balzer77] R. Balzer, N. Goldman and D. Wile, "Informality in Program Specifications," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* Cambridge, MA, August 1977, pp. 389-397.

[Bauer75] M. Bauer, "A Basis for the Acquisition of Procedures from Protocols," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tiblisi, Georgia, U.S.S.R., August 1975, pp. 226-231.

[DeJong86] G. F. DeJong and R. J. Mooney, "Explanation-Based Learning: An Alternative View," *Machine Learning 1*, 2 (1986), pp. 145-176.

[Good84] D. Good, "Mechanical Proofs About Computer Programs," *Philosophical Transactions of the Royal Society of London 312,* 1522 (1984), pp. 389-409. (Also appears in *Readings in Artificial Intelligence and Software Engineering,* C. Rich (ed.), Morgan-Kaufmann)

[Hill87] W. L. Hill, "Machine Learning for Software Reuse," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987, pp. 338-344.

[Manna86] Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis," in *Readings in Artificial Intelligence and Software Engineering,* C. Rich (ed.), Morgan Kaufman, Inc., Los Altos, CA, 1986, pp. 3-34.

[Michalski75] R. S. Michalski, "Synthesis of Optimal and Quasi-optimal Variable Valued Logic Formulas," *Proceedings of the 1975 International Symposium on Multiple-Valued Logic*, Bloomington, IN, 1975, pp. 76-87.

[Mitchell86] T. M. Mitchell, R. M. Keller and S. Kedar-Cabelli, "Explanation-Based Generalization: A Unifying View," *Machine Learning 1*, 1 (1986), pp. 47-80.

[Phillips77] J. Phillips, "A Framework for the Synthesis of Programs from Traces using Multiple Knowledge Sources," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* Cambridge, MA, August 1977.

[Quinlan86] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning 1*, 1 (1986), pp. 81-106.

[Rich86] C. Rich and R. C. Waters, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman, Inc., Los Altos, CA, 1986.

[Shavlik87] J. W. Shavlik and G. F. DeJong, "BAGGER: An EBL System that Extends and Generalizes Explanations," *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, WA, July 1987, pp. 516-520.

[Shavlik88] J. W. Shavlik, "Generalizing the Structure of Explanations in Explanation-Based Learning," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, January 1988. (Also appears as CSL Technical Report UILU-ENG-87-2276.)

[Shavlik89] J. W. Shavlik, "Acquiring Recursive Concepts with Explanation-Based Learning," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989, pp. 688-693.

[Steier87] D. Steier, "CYPRESS-SOAR: A Case Study in Search and Learning in Algorithm Design," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987, pp. 327-330.

[Summers77] P. D. Summers, "A Methodology for LISP Program Construction from Examples," *Journal of the Association for Computing Machinery 24*, (1977), pp. 161-175.