APPLICATIONS OF EXEMPLARY PROGRAMMING

BY WILLIAM S. FAUGHT

THE RAND CORPORATION

SANTA MONICA, CALIFORNIA

INTRODUCTION

Is there an easy way for a computer user to create new software?

One of the main obstacles to effective computer usage is the difficulty of developing software to perform a new task. Computer users, once they discover or identify a task, are faced with the difficult job of software specification and development. The current sequence of task analysis, program requirement specification, coding, and debugging has four bottlenecks: (1) if the user is not a programmer, he must find one; (2) the programmer must translate his mental model of the task into an algorithm; (3) the programmer must translate the algorithm into appropriate programming language statements; (4) the programmer must specify and debug the details of the algorithm, such as initial specifications, branch conditions, and terminating conditions.

In this paper, we discuss programming by example, and show how it can be useful in solving the above problems in three applications: (1) as a software generator for non-programmers in a specific domain; (2) as a specification technique for programmers (DWIT); and (3) as a specific software tool for program development (AUDIT).

We first give a brief overview of the EP paradigm and the EP system. We then describe three application areas in which programming by example could help solve software specification problems. Next, we characterize the class of tasks for which programming by example is suitable. Finally, we analyze how programming by example helps solve software specification problems in general.

THE EP PARADIGM

In the EP project, we have turned the normal programming sequence around. Instead of specifying an algorithm, a program, and an example for debugging, the programmer specifies an example, and the EP system infers the algorithm and the program; i.e., the programmer specifies the program only by giving an example of the desired sequence of actions the program is to perform. The idea is that the user shows EP how to do a task by performing the task himself.

The paradigm is as follows: The user performs some task on a computer, e.g., transferring a file from computer to computer, retrieving information from a data base, or editing a data file. EP watches over the user’s shoulder, creating a trace of the interaction between the user and the system he is using. When the task is done, EP constructs an algorithm or model of the interaction. Part of this construction may involve questions to the user or advice from the user. EP then constructs a program from the model and stores it in a library for subsequent use.

As EP watches, it builds a trace or protocol of the interaction. The trace is a verbatim description of the interaction. EP stores the trace to use with an editing facility for correcting mistakes in the protocol. All advice from the user also enters the trace.

From the trace, EP constructs a model of the interaction. The model can be thought of in two ways: as a generalization of the trace or as an interpretation (or explanation) of the user’s actions. For example, if the trace contains a sequence of repetitive actions, the corresponding structure in the model might be a FOR or WHILE loop. User advice both directs the model construction and constrains the space of potential explanations. EP can either generate an agent (a program) in a particular language to perform the task, or interpret the model directly to perform the task. EP stores the trace, model, and agent in a library for later use.

We plan to extend the current EP system so that models (and agents) can be modified with multiple examples of protocols. In that case, the user provides additional protocols in two ways: (1) the user performs an entirely new protocol, creating a new trace and model, and then tells EP to consolidate the two examples into a single model; (2) the user performs the old model in a “careful” mode, verifying each action before EP performs it; when a change is required, the user takes over from EP in performing the task; EP then consolidates this new (partial) protocol with the old trace and model.

EP can also aid the user in performing repetitive tasks. If EP is watching the user perform a sequence of steps in a repetitive task, the user need only perform enough steps until EP can construct a model for the repetition (perhaps lacking an exit condition). The user can then ask EP to perform the remainder of the sequence, optionally verifying each step as it is performed.

The EP paradigm has two implicit assumptions. The first is the requirement for a performance language. The user must demonstrate the task to EP by performing the task in-
USER INTERACTIVE TASKS

User-interactive tasks are those in which a person uses a computer operating system and its applications programs directly to accomplish some tasks. Several examples are:

1. computer network tasks
   a. transferring files
   b. logging in to remote systems to read mail
2. operating systems tasks
   a. file maintenance
3. data base retrievals
   a. periodic retrieval of time-oriented information
4. edit macros in one- and two-dimensional editors
5. tutorials.

There is a need for writing programs to free computer users from these tasks. The tasks tend to be repetitive and detail-prone. Computer users tend to be inefficient at such tasks. Further, these tasks are labor demanding: The computer user must (typically) wait for the computer to complete each step before typing in the next action. Delays can take several seconds up to a minute. Meanwhile, the user’s attention is held in limbo: No other tasks can be started because of the unpredictability of the computer’s response, but the response can be delayed enough to lose the user’s attention and increase the likelihood of his errors.

The characteristics of these tasks make it difficult to justify writing a separate program for each task. The tasks tend to be (relatively) fast-changing. A manager may want information on accounts receivable from Utah for one month and Nevada the next. Or a systems maintainer may have different features he wishes to exercise each day, but the features change each month. The tasks also tend to be person-specific. Each person has his particular data base item to retrieve, or remote site to retrieve mail from, or edit macros to write. Finally, most of the tasks involve complicated, detailed input/output specifications to interact with the various computer systems. It would be difficult to justify writing (and rewriting) programs for such tasks, and most programmers would dislike writing such programs.

EP, however, seems ideally suited for creating software for these tasks. With EP, users can create their own software without a separate specification loop. The EP paradigm assumes that the user will be task-intelligent and program-naive: The user knows how to do the task, but does not know how to manipulate programming constructs and code to build a program. With EP, the user simply shows an example of his task; EP builds the program. He then has a set of personalized agents for accomplishing his tasks. Further, tasks with simple control structures are easy to demonstrate. The programs needed to accomplish the tasks tend to be relatively simple, consisting of straight-line code, branches, loops, and variable instantiation. There are few complex control structures necessary like parsers or production systems. (Note that the user would find it difficult to demonstrate such control structures to EP.)

An additional bonus to the EP paradigm in this application is the notion of capturing expertise. The user may not know how to accomplish a task, so he may ask an expert for advice. The expert or the expert and the user together can then perform the task with EP watching and create an agent to do the task. Later, the user has the expertise “captured” in the agent—the agent can perform the task itself. Additionally, the expert may be the user himself. The user may have had to consult a manual to perform some seldom-performed task. By constructing an agent, perhaps with comments to himself included, the user can capture his temporary expertise in the agent and free himself from remembering the details.

The current EP system has several features to aid users in these applications. The first is a library of the user’s agents (programs) for storing and retrieving agents that the user created. The second is a “text” feature. The user can insert text comments into the agent for documentation. These comments are typed to the user as the agent is running, describing the agent’s actions. Finally, EP has a simple knowledge base of information about the various systems that EP’s users interact with. This knowledge base is a source of heuristics for inferring branch conditions and loop boundaries.

Problems we anticipate in providing users with an EP software capability center on software maintenance. Users may create software with little documentation and distribute the software to non-expert users. Also, application systems may change and require the agent to be updated. EP has several features which help alleviate the problem. First, the user can watch EP interact with the applications programs or system and can see where EP’s agent is in error. Second is the text feature described above for putting documentation in the agent. Finally, the agent can be executed in a “careful” mode which asks the user for confirmation of EP messages to the system before it sends them to the system. The user can stop any invalid action before it occurs.

DWIT (Do What I want)

Since the advent of programming languages, computer programmers have searched for ways to rid themselves of the compile-load-execute sequence which kept them distant from their programs. Interpreted languages and interactive
debuggers redirected the programmer’s efforts from dealing with an entire program to developing a single function. INTERLISP, an interactive version of LISP, is an example of an intensive effort to provide the programmer with a “friendly” environment in which to implement his ideas in programs. Its facilities include an interactive editor, the BREAK package for interactive debugging, the DWIM (Do What I Mean) facility for correcting the user’s errors in using the system, and a programmer’s assistant for UNDOing and REDOing the programmer’s actions by retaining a history of them (Teitelman, 1969). The main purpose of these facilities is to get the programmer “closer” to the program—closer in terms of developing the program interactively, rather than in batch mode.

However, none of these systems allows the user to develop the program completely interactively. There is still a vestige of the compile-load-execute sequence at some level. For example, in INTERLISP, the programmer must still specify an entire function before any (semantic) debugging can take place. The user is consigned to a sequence of developing an algorithm, writing the function with the editor, constructing a test environment, and running the function to debug it.

Programming by example provides a way to reduce this distance by allowing the programmer the ultimate interaction—at the program statement level. The programmer types to an interactive system with an interpreted programming language (e.g., INTERLISP). The user types each program statement to the program interpreter with appropriate arguments. The program specification system watches what the user types and creates an agent to perform the same function. When the user is done with the task, the agent is translated into the programming language and stored in the user’s core image.

The important feature of the paradigm is that the user debugging each program statement as he types it in. The user verifies that the statement did what he intended as the interpreter executes the statement. When the user is done with the function, he is assured that it is bug-free, at least for the example he used (which is likely to be generic to the task). (Note that the user must be a competent programmer in the performance language).

To program a function, the user simply types the program statements to the interpreter with example arguments. By doing so, he shows DWIT both the program’s data structures and the control flow. DWIT stores the data structures in a variable list. The user’s task is to transform the initial variable list, consisting of the input parameters and (implicitly) all global variables, into results to be returned as the value of the function or to set global variables to. Control structure is demonstrated by the sequence of actions that the user takes. At the simplest level, the paradigm can be thought of as an extended macro feature with registers (the variable list) and with the ability to infer branching and looping control structures.

Although most of the function’s actions can be shown by example, DWIT has a few commands to ease both DWIT’s and the user’s job. The user can demonstrate data structures precisely by his input statements. Control structures, however, require a few aids due to their implicit nature. The problem is that the user can do some actions in his mind that do not appear in the protocol. Conditions are the prime example. If DWIT were clever enough, it could infer the conditions and ask the user for verification. For simple cases, this would suffice. For complex cases, however, the user needs a way to tell DWIT the proper condition. The user does this with a special command, telling DWIT the branch point and the condition.

The second type of command helps the user with repetitive examples. The purpose is to require the user to type in a particular control path only once. When the path is to be repeated, the user can specify where to start in the path, and DWIT will continue execution from there. Options such as a “careful” mode and stopping after each loop constrain the execution.

In addition to providing instant debugging of his program statements, this paradigm aids the user in several ways. First, the user has an example at hand to prompt his actions. By seeing his actions performed in the example, he is prompted to perform the next action. Second, the user has a trace of the example’s execution path that provides a history of what action has taken place. Finally, many times the user can type in just part of the action he wants to take place and DWIT can complete it.

DWIT can be given additional programming heuristics to aid the user. First, DWIT could infer some conditions at branch points, based on the syntax of program statements. Second, DWIT could know the parameter types of each function and could attempt to correct the user’s statements, e.g., correcting the order of function parameters based on the parameters’ types. Third, DWIT could generate some functions on its own. The user could type “LOCATE <expr>” where <expr> is a performance language expression that the current variable contains. DWIT would then locate the occurrence of <expr> and generate the function which the user could have typed to extract <expr> from the current variable.

The user’s protocol provides two byproducts: documentation and a test case. The examples at each step can be inserted into the generated function as comments. Also, the initial arguments and the final returned result(s) can provide global documentation as to the calling sequence and returned value of the function. The protocol also serves as a test case for future modifications. These modifications can be tested on the example for one test of the function’s correctness.

A prototype version of a feature similar to DWIT (called the Mind Channel) has been implemented in the current version of EP. Two questions remain: What types of functions are most suitable to development with DWIT, and how can DWIT provide enough documentation and flexibility so that large-scale systems can be developed incrementally. We will return to these questions later.

AUDIT - A PROGRAM GENERATOR FOR DATA FILE AUDITING

Two features of programming by example are useful for a program generator. One feature is instant verification of
program actions. Basically, the user attempts to do an action, sees its effects, and instantly verifies its correctness. A second feature is the specification system's attempt to do part of the specifying. The user may type an input and an output and ask the system to synthesize the intervening function.

We will argue that this type of cooperation provides a useful shortcut to program specification in certain domains. Two types of shortcuts are possible. The first concerns data structures. The specification system can deduce data formats from examples supplied by the user. The second shortcut is in control structure. If the users executes a partial program that he has developed, he need not fill in little-used branches until (and unless) he actually needs them.

The domain we will consider is data auditing. In this domain, the user has one or more data files that he wants to verify for accuracy or peruse for special cases. The file may originate from an errorful source and require format checking. Or the user may want to compute a frequency distribution of the values appearing in various fields or verify that the values are within acceptable ranges. Statisticians typically perform several such audits on their data files before performing statistical analyses.

The current method for data auditing is to write a separate program for each type of audit, using either a general purpose language like SAIL or PL/1 or a special purpose auditing language like BRIGHT (Goldberg, 1978). The user defines the structure of the file, records within the file, and fields within the records. He then specifies the fields he wants audited, their locations, the expected ranges, and whether he wants a distribution of values. He then runs the program and examines the output, which is in a tabular form.

Specification by example provides an alternate method for data auditing. (The hypothetical specification system will be called AUDIT.) The first task is to specify the format of the data. The user reads in the first few lines or records of the file (using an interpreted performance language). AUDIT attempts to parse the records and creates a parse description which defines the format. This is an interactive task, with AUDIT making hypotheses and the user confirming or denying them. The user could name the fields for easy reference.

The user then tells AUDIT what he wants done with particular fields, e.g., whether to distribute the values or verify a range. As AUDIT reads through the data file, it looks for incorrect data formats (according to its parse) and out-of-bound values. The user has the option of stopping AUDIT whenever an error condition is found, or having AUDIT record the error and continue.

AUDIT's main feature is that it does most of the detailed calculations for field specification. Initially, it counts columns or parses fields for its first set of assumptions. As it reads more examples, it parses them, looking for format errors. If the user asserts that the record is correct, AUDIT automatically extends its assumptions about the field definitions.

The other important feature is that the user can interactively develop programs for manipulating the fields. The user could create a program to select records satisfying particular conditions from one file and write them onto another. He can also edit the file, making the same correction on each record, or reformat the file. AUDIT saves the user effort in that it continues auditing data until new data is found. The program will stop whenever it encounters an unknown condition and ask the user to construct the proper branch condition and path.

**ISSUES OF SPECIFICATION BY EXAMPLE**

We can analyze the utility of specification by example in several ways: by the advantages to the user over other software specification methods, by the classes of programs amenable to demonstrating examples, and by comparing this method to other program synthesis research.

**Software specification problems**

In general, programming by example can help solve software specification problems in a number of ways.

First, programming by example makes specification easier because the user can demonstrate a task rather than specify an algorithm. The user's model may not be represented in an algorithm; he may know the task only by performing it. For example, the task representation may be in kinesthetic memory (Arnold, 1960), or in 'sequential action patterns' (Faught, 1978). Even if a person learned to do a task from an algorithmic specification, his representation of the task may eventually be 'compiled' into a performance-oriented representation (Dennett, 1969).

Also, in demonstrating a task, the user can depend upon context and perceptual cues from a real-time example. The example provides context by putting the world in a familiar and complete state, whereby the user can specify the next step. Further, it may be easier to specify what action should be taken in one particular case than to generalize over several cases. One attempt to accommodate this difficulty has been the development of production system languages (e.g., RITA (Anderson & Gillogly, 1976)) that enable their users to specify actions in IF-THEN rules.

Second, programming by example makes specification more accurate than coding in a programming language. The trace provides an example that the model must a priori account for. If this trace is correct (which may be easier for the user to verify than an algorithm) then the model must be correct for at least one example. Also, the user gets immediate verification of the correctness of his input to the system. If it is wrong, he can correct it on the spot and eliminate this source of bugs. Finally, the specification system will decide on implementation details (to some extent). If the user can perform the task on the system, the synthesized program can, at the minimum, perform the task in the same way. The program also has the option of performing some of the actions internally, rather than as side effects. The user, however, must decide how to perform the task given the real-time interactive languages available to him on the system.
Third, programming by example makes program specification less tedious. The motivation behind the paradigm is to acquire a program specification with the simplest and minimum input from the user. To that end, the specification system can use its accumulated knowledge to infer the specification from a minimal set of traces in two ways:

1. The system can use its knowledge of programming constructs to flesh out partial descriptions. For example, when the system detects a repetitive sequence of actions, it will hypothesize a loop. It constructs the body of the loop, as would a programmer building an algorithm. It then attempts to fill in the initial and exit conditions, using its knowledge base of programming constructs and the trace. The point is that the specification system does the hand simulation to generate the conditions, not the user.

2. The specification system can use its knowledge of the domain to supply tests for error conditions and error correction procedures, conditions that will probably never be demonstrated.

Finally, programming by example eliminates one aspect of debugging and provides one form of documentation for the program. Since the model must account for the trace, the user is assured that in at least one case the resulting program is correct. Debugging is moved to the level of program modification. The user’s task is to extend the model to account for more tasks or more cases. However, he still retains the original trace to test, so that later changes must be compatible with the earlier example. These saved traces also provide a form of documentation in terms of an example created by the user and hopefully suggestive of the purpose of the program.

**Classes of programs amenable to programming by example**

The basis of programming by example is the completeness of the user’s task example. The user must be able to demonstrate the task in a step-by-step sequence of actions on the target system. Thus, tasks which require complex control structures, such as context-free parsers and language translators, would be impossible to demonstrate. The control structure should also be natural or easy for the user to show. This probably eliminates target programs that have a control structure like a command interpreter or a production system. However, with an extension to allow the user to suggest control structures such as command interpreters, the specification system could direct the user’s demonstration and learn the control structure under its own guidance. Thus, the set of programs most amenable to programming by example should be sequential, for the most part, with simple looping and branching.

The paradigm is also dependent upon the user having some performance language available to perform the task. Thus, tasks that require special purpose languages that have no interpreter, such as programming in assembler language, are not suitable.

As discussed above, user-interactive tasks seem ideally suited to software specification by example. Users need programs to free themselves of repetitive, detailed interactions with applications programs. Yet writing such programs cannot be justified due to their fast-changing specifications. Because the user is most in tune with his needs and is closest to the problem definition, an EP-like facility can provide the user with a library of individualized software. Examples of such tasks are: computer network accessing, operating systems interactions, file auditing and maintenance, data base retrievals, editing macros, and tutorials.

One limitation of EP’s application may be the limits of the performance language. In this case, EP may have to provide the user with its own performance language, such as DWIT and AUDIT.

Tasks that are rich in input/output interaction allow the specification system to do much of the user’s work in formatting the data such as in AUDIT. Also, these tasks are easy to show because most of the algorithm is side effects, and therefore can be demonstrated in the performance language, such as pure functions in DWIT.

Certain task characteristics inhibit software specification by example. For instance, some tasks requiring special I/O, such as two-dimensional text editors and other graphics, are especially difficult for the specification system to understand because the user’s actions may depend upon the content of the screen. The general solution to this would require the system to simulate the screen internally; the screen then becomes the context for the user’s actions. The same problem occurs with standard screen or terminal conventions. For example, backspaces are handled differently by most systems. In order for a specification system to understand their effects, it must simulate the effect of the backspace on each system. Of course, this applies to all of the other editing commands on each system.

Report generation and business data processing in general would necessitate additional features to a specification system because of the complexity of the task specification. To keep from having to show an impossibly long trace, the task must be broken into functions. The system would then need additional bookkeeping facilities to help the user merge the program pieces.

For some tasks, such as matrix multiplication and sorting, it is easier to describe the algorithm than to demonstrate, step-by-step, the execution path of the required action sequence. For these tasks, demonstrating an example would be inappropriate.

**Other program synthesis research**

Programming by example computations has been extensively studied (cf. Bauer, 1975; Bierman et al., 1975; and Bierman & Krishnaswamy, 1976). Bierman’s system, the autoprogrammer, is an excellent example of the scratchpad concept. The programmer tries out a line of (assembly language) program by typing it to the autoprogrammer interpreter and interactively debugs the line. The autoprogrammer then constructs a complete listing of the code. EP is also
related to the QBE system developed by IBM for data base retrieval. This system retrieves information based on examples of the type of information desired (Girdonsky and Neudecker, 1976).

We have extended the concept of programming by example in several ways. First, we have outlined the specific assumptions about the relative advantages of examples to other program specification media. Second, we have shown ways that the specification task can be shortened by using a knowledge base of domain and programming knowledge to fill in implicit branches and loops. Third, we have defined the characteristics of tasks that make the EP paradigm amenable. Finally, we have defined several prototype systems to serve particular known needs and analyzed how those needs can be met.

The main difference between EP and other automatic programming projects is how the user specifies a task. In EP, the user shows an example by performing the task live on a computer (as opposed to simulating what the system "would have said"). In Balzer's SAFE project (Balzer, 1978) and Heidorn's automatic business programming work (Heidorn, 1976), task specification is in natural language. In Manna and Waldinger's project (Manna & Waldinger, 1978), task specification is a series of input/output specifications that the user provides—a type of simulation. Finally, in Green's PSI project (Green, 1977), the main emphasis is on incremental program specification by multiple means, including natural language and example. As with Waldinger, the example is not live; the user must provide both input and output specification.

Another difference is in task domain. EP attempts to mesh with current computer technology to provide a useful tool for computer users. Therefore we have focused on typical interactive tasks, such as operating system, network, and file maintenance interactions.

Finally, EP attempts to provide an immediate software tool, but one which can be incrementally improved as more AI techniques are developed. Therefore, we have attempted to find techniques in which EP can use as much information as it has but require as little information as possible to be viable. This characteristic will make EP readily available for new domains.

ACKNOWLEDGMENTS

This work has benefited substantially from comments and suggestions by Don Waterman, Phil Klahr, and Steve Tepper. The research reported here has been supported, in part, by the Defense Advanced Research Projects Agency under contract MDA903-78-C-0029.

REFERENCES

Dennett, D. C., Content and consciousness, New York; The Humanities Press, 1969.
Teitelman, W., "Toward a programming laboratory," International Joint Conference on Artificial Intelligence, May 1969.