

Overview of the Programmer's Apprentice

Charles Rich, Howard E. Shrobe, and Richard C. Waters

Artificial Intelligence Laboratory and the Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Mass. 02139

ABSTRACT - This note gives an overview of the Programmer's Apprentice system being developed at MIT. This system is conceived as being midway between an aid to improved programming methodology and an automatic programming system. A programmer and the apprentice work together throughout all phases of the development and maintenance of a program. The programmer does the difficult parts of design and implementation, while the apprentice acts as a junior partner and critic, keeping track of details and assisting the programmer wherever possible. A key feature of the apprentice is its ability to understand the logical structure of a program so that it can interact with the programmer in a meaningful way.

1. Introduction

The Programmer's Apprentice (PA) is an interactive system for helping programmers manage the complexity of evolutionary programming. It is being designed and implemented at MIT by Charles Rich, Howard Shrobe and Richard Waters (only some of the modules in the system are actually running at the present time). The intent is that the programmer will do the hard parts of design and implementation, while the PA will act as a junior partner and critic, keeping track of details and assisting the programmer in the documentation, debugging, and modification of his program. In order to cooperate with the programmer in this fashion, the PA must be able to "understand" what is going on. The central achievement of the Programmer's Apprentice project has been the development of a "plan" representation which facilitates the cataloging of basic programming knowledge.

The "plan" for a program is a network of operations interconnected by data flow and control flow links. This representation is a better vehicle for expressing the logical interrelationships in a program than the programming language source code. A plan is segmented into a hierarchy of subsegments, each corresponding to a conceptual unit of behavior. The plan specifies how each non-terminal segment is constructed out of the segments contained within it. Explicit dependency links are used to record the goal-subgoal and prerequisite relationships between a segment and its subsegments. Taken together, these links summarize how the behavior of a segment follows from the behavior of its subsegments and the way they are connected by control flow and data flow. A final aspect of the plan representation is that there may be more than one plan for a program or part of a program, representing different points of view.

The plan representation is also used in the PA system to construct a library of common algorithms and data structure implementations. In general, the PA attempts to understand a programmer's code in terms of these familiar plans.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

2. A Scenario of Use of the PA

The following imagined dialogue between a programmer and the PA is presented in order to illustrate the intended operation of the system. (Comments discussing the scenario are printed in *italics*.) The scenario illustrates the following basic areas in which the PA can assist a programmer:

- 1) **DOCUMENTATION:** The PA provides automatic, permanent, and in-depth documentation of the program. The PA remembers not only explicit commentary supplied by the programmer with the code, but also a substantial body of derived information describing the logical structure underlying the program.
- 2) **DEBUGGING:** Any discrepancy between the PA's understanding of the programmer's intent and the actual operation of the program is reported to the programmer as a potential bug.
- 3) **MANAGING EVOLUTION:** Based on its knowledge of the logical relationships between parts of a program, the PA is able to determine what parts of a program are affected by a proposed design change. It can use this information to warn the programmer of impending difficulties.

The scenario traces the design, coding, and subsequent modification of a program which deletes an entry from a hash table. The scenario picks up in the middle of a session at a point where the programmer has already made many design choices and told them to the PA. In particular, he has stated the input-output specifications for the subroutine DELETE and chosen the basic data structures to be used. The hash table is implemented as an array of linked lists called "buckets". Each of the entries in a bucket is a dotted-pair whose CAR is the key for the entry.

Programmer: The plan for the routine DELETE has three main steps: the first one hashes the input key to get an index; the second one fetches the bucket corresponding to that index from the table; and the third splices out the entry in the bucket whose key part is equal to the input key.

[The only feature of this scenario which is not a current research goal of the Programmer's Apprentice project is the use of natural English dialogue. Although the system should be able to support a sophisticated natural language "front-end", no effort has been expended in this direction to date.]

The PA uses the programmer's outline of the design in order to build a plan representing the program's intended structure. References to basic programming techniques such as "splicing out" (deleting elements from a linked list) trigger the retrieval of plans from the plan library.]

PA: I think that there is a bug in this design. If the input key is equal to the key part of the first entry in a bucket then the first entry will not be spliced out.

[After constructing a plan, the PA uses its reasoning abilities in order to check for bugs. Above, the PA finds a bug and reports it to the programmer, framing its response in terms of the "splicing" plan. In general, it is the responsibility of the programmer to supply corrections.]

Programmer: Modify the plan as follows. Add a final step which compares the bucket after splicing with the bucket before splicing and stores the new bucket in the table under the index if the two initial cells are not identical.

PA: OK, the plan now seems correct.

[The programmer then types in the code for DELETE.]

Programmer:

```
(defun delete (key) ; hash table delete routine.
  (prog (bkt1 bkt2)
    (setq bkt1 (tbl (hash key))
          bkt2 (bkt-delete key bkt1))
    (cond ((not (eq bkt1 bkt2))
           ; check if first entry deleted.
           (store (tbl (hash key)) bkt1)))))
```

[The PA analyzes the code written by the programmer in order to construct a plan for it. Comparing this plan with the plan directly described by the programmer, the PA discovers an implementation bug. The bug is so simple that the PA can suggest a correction.]

PA: This code is not consistent with your plan. When the first entry is deleted, the code stores the bucket before splicing, rather than the bucket after splicing. Did you mean:
"(STORE (TBL (HASH KEY)) BKT2)"?

Programmer: Yes, fix the code.

[Some time later it is necessary to make a modification to the original design. The PA helps the (perhaps new) programmer maintain consistency.]

Programmer: I want to make a change in the representation of buckets in the hashing programs. A bucket now has two parts: a count field and a list of entries as before. The count field will contain an integer equal to the length of the list of entries. Let a bucket be a dotted-pair whose CAR is the count and whose CDR is the list of entries.

[The PA inspects the logical dependencies in the plans for the programs being worked on in order to determine the specific ways in which the programs can be affected by the proposed changes.]

PA: The proposed change will affect DELETE as follows: After the bucket is fetched, it is passed to BKT-DELETE which does the actual deletion. However, BKT-DELETE requires a list of entries while in this new implementation, a bucket is no longer just a list of entries.

Programmer: Change the specifications of BKT-DELETE so that it expects as input a bucket of the new type, rather than a list of entries.

[... and so the session continues with the PA looking over the programmer's shoulder.]

3. Current Status of the PA

Rich and Shrobe [1] laid out the basic idea of a plan and the initial design of the PA. Since that time Rich, Shrobe and Waters have been working together on further aspects of the theory along with design and implementation of the PA system.

Rich's work [2] centers on the plan library. He is using the plan representation to formalize a number of common techniques in non-numerical programming involving trees, lists, and arrays. He is also studying the use of this library in the PA system for analyzing already-written programs and to provide high level abstractions for interactive program synthesis.

Shrobe [3] has implemented a deductive module, called REASON, which can reason about programs represented by plans. REASON is a dependency based system which maintains an explicit record of its control information within the same formalism as is used to represent facts. An accompanying article in these proceedings describes how REASON reasons about destructive modifications to complex and potentially shared data structures.

Waters [4] has implemented a system which can analyze the code for a program and produce a structured plan for the entire program. The basic idea behind Waters' work (which is described in detail in an accompanying article in these proceedings) is that plans for most programs are built up in a small number of stereotyped ways, and that features in the code for a program can be used to determine the structure of the plan for the program.

Our goal for the immediate future is to construct a prototype system which exhibits the kind of behavior shown in the scenario. To do this, an interactive module must be built, and the modules which have already been constructed must be connected together into an integrated system. Looking further ahead, additional modules (such as a simple program synthesis module, and one dealing with efficiency issues) will be added to the PA, and the existing ones strengthened so that the PA can take over more responsibility from the human programmer.

REFERENCES

- [1] C. Rich & H.E. Shrobe, "Initial Report on a Lisp Programmer's Apprentice", IEEE Trans. on Software Engineering, SE-4, No. 6, November, 1978.
- [2] C. Rich, "A Library of Programming Plans with Applications to Automated Analysis, Synthesis and Verification of Programs", forthcoming PhD thesis, MIT AI Lab, 1979.
- [3] H.E. Shrobe, "Reasoning and Logic for Complex Program Understanding", MIT/AI/TR-583, June 1979.
- [4] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, April 1979.