How to Print a File: An Expert System Approach to Software Knowledge Representation

Peter G. Selfridge
AT&T Bell Laboratories
Room 3C-441
Murray Hill, NJ 07974

Abstract

Representing knowledge of software and software systems is an important research area and a prerequisite to engineering expert-level systems to do software tasks. Printing a file in a UNIX™ environment is an example of a real-world problem that can pose surprising difficulties to UNIX users. The printing of files is also illustrative of a class of software problems characterized by the recombination of existing programs. Automating the printing process involves designing knowledge representations to appropriately capture knowledge of both the printing software and the printing process and designing a reasoning system that uses those knowledge representations in a working implementation. This paper examines the printing problem in detail, presents a model of printing and printing software, and describes an implementation designed to test the model and identify the next set of research issues. The implementation, ESP, for Expert System for Software, successfully automates the printing process and illustrates a knowledge-based approach to software problems.

1. Introduction

Developing knowledge representations for software and software systems is an important research topic and a prerequisite to other difficult research topics and practical problems in Artificial Intelligence. On the practical side, developing expert level systems to do software tasks is increasingly important as programming and using large systems becomes more complicated. Such expert systems will need explicit knowledge representations of software and software components and reasoning mechanisms that use those representations. High-level automatic programming, where existing software modules are re-combined for new purposes, will need knowledge representations that capture the invocation rules, input and output behavior, and side effects of software. Finally, if more advanced intelligent software systems are to be created, models of software as explicit mechanisms and automatic reasoning systems that use those mechanistic models will be necessary.

The goal of this research is to take a specific software domain, that of printing files in UNIX™, develop a model of the domain and of the software used in that domain, develop knowledge representations and reasoning mechanisms based on this model, and finally, test the representations and reasoning mechanisms in a working implementation. A corollary of this research strategy is that the implementation should be specific enough so that it makes clear the next set of research issues that should be addressed.

This paper first examines the printing problem in more detail, and generates some specific criteria that automating printing should address. A model of the printing process is described based on an input/output specification of printing software and some observations are made on the kind of knowledge needed to automate the process. Then, an implementation is described, called ESP, for Expert System for Printing, that uses knowledge representations and reasoning techniques derived from the model. The implementation is illustrated in some annotated examples. Finally, this work is described more generally, compared with other approaches to the representation of software, and used to generate the next set of research questions.

2. The Printing Problem

The UNIX operating system [8, 19] allows programs to be combined so that the output of one program becomes the input to the next program; several programs can be combined into what is called a 'pipeline'. A program designed to be included in a pipeline is called here a *translator*. The following is a pipeline for printing the file 'memo' on a particular printer in our environment:

```
% cat memo | eqn | tbl | troff -mm |
      1       2     3      4a     4b

  dpost -o1-9 | postreverse | lpr -Plw
    5a     5b         6           7
```

This pipeline is interpreted by UNIX as follows. In step 1, the file 'memo' is opened by the program 'cat' and sent as a stream of data to step 2, the translator 'eqn', which translates equation text into 'troff' format, the primary document format. In step 3, the resulting stream is sent to the translator 'tbl', which does the same thing to the table text. In step 4, the result is received by the primary formatting translator, 'troff' [6]. 'Troff' also takes the option indicator '-mm' from

the command line, step 4b, which tells it to use a particular macro package for deciphering certain macro commands present in the text. The output of 'troff' is a 'troff' file, where the text has been completely replaced by troff commands. The final destination of the data stream, step 7, indicated by 'lpr -Plw', is a particular printer, the 'lw' (laser writer) printer. This printer is a 'PostScript' [11,12] printer which takes data in the 'PostScript' format. Therefore, step 5 is required, where the translator 'dpost' translates 'troff' format into 'PostScript' format. 'Dpost' is given another option in step 5b, an option that specifies which subset of the document should be printed; in this case, the pages 1 through 9. Finally, the translator 'postreverse' is needed in step 6 to order the pages properly front to back.

Generating an appropriate pipeline can be difficult for many reasons. First of all, one needs to know all of the different kinds of text present in the file. Second, one needs to know what translators are needed to process the file, how to invoke them, and the order in which they must appear. Third, some options (like '-mm' above) are critical to the final appearance of the printed document and must be included in the proper place. Fourth, an appropriate printer must be chosen and not all printers can print all kinds of files. Fifth, for a given printer, other translators may be needed, and other options may be desired which have to be properly invoked and included in the proper place in the pipeline. Sixth, detecting that a printer is 'down' or otherwise inaccessible, or has a prohibitively long queue of current printing jobs, must be done from time to time. Finally, users typically use a given printer and options most of the time, so that using defaults correctly becomes part of the problem.

Automating the printing process is an important research project for two reasons. First of all, from a practical point of view, such a system would be quite useful. More important, the printing problem is representative of a class of software problems which involve the recombination of existing programs. Recombining existing programs [26] is likely to become a very important mechanism for creating software, as well as aid in the problems of understanding, debugging, and revising software. Progress on the printing problem should identify the next set of research issues relevant to the larger problem of automatic programming by program recombination.

Representing software knowledge for automatic programming has been addressed by a number of researchers. Rich and Waters, and others, have developed the "programmer's apprentice" framework [15-18, 22, 23] and various techniques for representing knowledge of specific program constructs. Their techniques address a lower level of automatic programming and representation of software knowledge, and are not directly applicable to the problem of automatic recombination of existing programs. Wilensky's UNIX consultant (UC) project [23-25] addresses the representation of software knowledge to help the user find particular UNIX commands, but does not address the combining of commands to achieve other goals. Other research into automatic programming almost exclusively use the "transformation approach", which represents programming knowledge as correctness-preserving syntactic transformations applied to a semantically correct program "specification" [2,3,7,20]. Transformation approaches are not applicable to program re-combination, being specifically tailored to the construction of small programs for specific low-level computations.

In summary, the problem of printing files in UNIX and the goal of automating this task is of both practical and theoretical importance, and generates the following specific problems that must be addressed in both a model of the domain and the implementation:

1. how are translators and options chosen?
2. how are they properly ordered?
3. how are printers selected?
4. how can user defaults be handled?
5. how can erroneous printer output be minimized?

3. A Model of File Types and File Printing

This section describes a model of printing files in a UNIX environment. The general model is based on the observation that the file to be printed (including data flowing through a pipeline) can be viewed as an object with a set of properties or types, each type representing one kind of text in the file. With this view, a translator can be modelled as an operator that changes the set of types of the data object, a printer can be modelled as a terminal operator that can only accept data of a specific type, user defaults can be modelled as demons, and the printing process can be modelled as a sequence of operations to detect the file types, select a printer, select the translators, and order them to produce the final pipeline.

With this general model in mind, the specific types of knowledge needed and their general representation need to be identified. There are three categories of "objects" for which knowledge is needed: file types, translators, and printers. The kinds of knowledge are listed here:

Knowledge of files types:
- text patterns that indicate file types

Knowledge of translators:
- translator input type
- translator output type
- translator order
- translator invocation
- possible options

Knowledge of printers:

- printer status
- printer queue length
- printer host
- host status
- printer invocation
- printer input

Each kind of knowledge can be represented as a "frame", or knowledge structure, which is a named object with slots. Two such frames are shown here:

```
FRAME TYPE: translator
  name:      tbl
  input:     tbl
  output:    troff
  order:     10
  help:      "tbl processes 'tbl'
      commands to format tables"

FRAME TYPE: printer
  name:      dp2
  invocation: "lpr -Pdp2"
  input:     postscript
  status:    unknown
  host:      vivace
  h-status:  unknown
  loc:       "coffee room"
  help:      "large, fancy laser printer"
```

For the translator frame, the 'input' slot is used after the file types are detected. For each detected file type in the user file, the translator knowledge base is searched for a translator that can translate that type. The 'output' slot indicates what output type is produced, and this is used to select more translators. The 'order' slot is used to order the translators in the final output command line. (In this domain, translators have a fixed order that can be represented by an integer. That is, the translator 'troff' is given a higher order because it always follows 'tbl' in the command line.) The 'help' slot is used when a user requests information about a particular translator or all translators.

The printer frame is similar. The 'invocation' slot indicates how to invoke the printer in a command line. The 'input' slot indicates the type of input it needs, the 'status', and 'host-status' slots store whether that printer and its computer host are up or down, and the 'help' and 'location' slots are used to provide summary information about the printer. Knowledge about file types and options are similarly encoded in frames.

User defaults, an important part of the goal of automating the printing process, can be modelled as small "demons", which fire when a specific pattern is observed. For example, if the user's default 'troff' macro package is '-mm', then when 'troff' is to be used in the final output, '-mm' can be automatically generated and included.

The actual printing process is modelled as the following 'script', or sequence of operations:

1. determine the printability of the file
2. determine the set of types of the file
3. select a printer if one is not indicated
4. for each type of file, select a translator that will translate the type into one that is eventually printable on the selected printer
5. select other translators if necessary
6. determine the order of the translators and options and construct the final command line

In summary, this model is based on the idea of the flow of typed data and the translation of types into other types. Knowledge of file types, translators, and printers is encoded in frames, and procedural knowledge is encoded in scripts.

4. Implementation

An implementation called ESP, for Expert System for Printing, was written to evaluate the approach described above. ESP consists of 170 OPS5 [5] rules and another 20 routines written in 'C', which encode in procedural form a variety of kinds of low-level knowledge about files and file types. ESP can be used as a stand-alone program to print files, examine the knowledge bases of printers, translators, and options, examine printer queues, remove jobs from printer queues, and get help of various kinds. In addition, it can be called directly from the shell by typing 'print' followed by a file name and optionally, a printer and other option specifications. The details of the implementation will not be presented here; it suffices to say that the frames described in the last section are represented as OPS5 literals and that the 170 rules encode the flow of control. Rather, the performance of ESP is illustrated with regard to the five specific questions listed at the end of section 2.

The first example illustrates ESP selecting translators based on the detected types of the input file, the use of knowledge about the indicated printer to further refine the set of translators, and the use of user defaults. The user requests that ESP print the file 'memo' on the 'xpp2' printer. ESP determines the types of the file 'memo', and uses knowledge of the printer to remove 'troff' from the list of types while retaining nroff, which that printer can handle. This knowledge is derived from the fact the 'xpp2' can only handle 'ascii' text, and there is no translator that can translate 'troff' to 'ascii'. ESP also used knowledge of the user's defaults to select the option '-mm' and include it in the final command line. What the user types is in bold.

```
% print memo xpp2
'memo' CONTAINS ascii tbl troff nroff
      TEXT
APPROPRIATE TRANSLATORS: tbl troff nroff
REMOVING troff - nroff PRINTER
      INDICATED:  xpp2
DEFAULT OPTION -mm SELECTED FOR
      TRANSLATOR nroff
TRANSLATORS AND OPTIONS ORDERED
GENERATED PRINT COMMAND:
cat memo | tbl | nroff -mm | lpr -Pxpp2
%
```

The next example illustrates ESP's response to a more complicated request. ESP uses its knowledge of options to recognize the three options in the user's request. One option is recognized as being in conflict with a default option, and the user is informed that the default will not be used. Knowledge that a 'troff' printer is available, and that 'troff' is generally preferred over 'nroff' is used to select 'troff'. Additional knowledge is used to select 'eqn' ('neqn' is used with 'nroff'). A default printer is selected, a set of translators for handling the file types is selected, and two other translators are selected which are required ('dpost' and 'postreverse'). Finally, the detected options are inserted into the command line using option knowledge.

```
% print nhg -ms -o1-9 -n9
'nhg' CONTAINS so pic neqn eqn tbl
      troff nroff ascii TEXT
APPROPRIATE TRANSLATORS: soelim pic
      neqn eqn tbl troff nroff
REMOVING nroff - troff PRINTER
      AVAILABLE
REMOVING neqn - eqn IS USED WITH TROFF
DEFAULT PRINTER lwc SELECTED FOR troff
INDICATED OPTION -ms OVERRIDES DEFAULT
      OPTION -mm
OTHER TRANSLATORS REQUIRED: REQUIRED
TRANSLATORS AND OPTIONS ORDERED
Generated print command:
cat nhg | soelim | pic | eqn | tbl |
    troff -ms | dpost -n9 -o1-9 |
    postreverse | lpr -Plwc
%
```

The next example shows the use of knowledge to prevent a user from producing erroneous output. ESP prevents the user from printing a directory and an executable file. Then, ESP recognizes that there is no way to print a 'tex' file on the printer 'eps2', and produces an explanation of a sort (producing better explanations will require further research). The user then relies on the default printer, and the request succeeds.

```
% print /usr/pgs/letters
/usr/pgs/letters IS A directory, WHICH
      IS UNPRINTABLE
% print /usr/pgs/bin/maclib
/usr/pgs/bin/maclib IS A executable
      file, WHICH IS UNPRINTABLE
% print letter.tex eps2
'letter.tex' CONTAINS ascii latex TEXT
APPROPRIATE TRANSLATORS: latex
OTHER TRANSLATORS REQUIRED: dvi2ps
I CAN'T TRANSLATE THE OUTPUT OF dvi2ps
      TO PRINT ON eps2.
THE DEFAULT PRINTER FOR LATEX IS lwc.
% print letter.tex
'letter.tex' CONTAINS ascii latex TEXT
APPROPRIATE TRANSLATORS: latex
DEFAULT PRINTER lwc SELECTED FOR latex
OTHER TRANSLATORS REQUIRED: dvi2ps
Generated print command:
cat letter.tex | latex | dvi2ps |
      lpr -Plwc
%
```

In summary, the implementation successfully automated most (but not all) of the printing process, and resulted in a very useful tool. The problems posed at the end of section 2 are all handled adequately, and the output of ESP is sufficient to both print files and allow the user to follow, more or less, the processing required.

6. Discussion

The main problem addressed by this research was to design knowledge representations of how to print files in UNIX, and to test the representations in a working implementation. The kinds of knowledge needed for this task were listed, and the printing process was modelled. This model was used in an implementation, ESP, which included knowledge representations for file types, translators, printers, and options, and a collection of rules that accessed the knowledge base, handled exceptions, and executed subgoals to generate the final UNIX pipeline. ESP is being used daily by a number of technical and non-technical people.

Although quite adequate for the initial printing domain, the underlying model of a software module is relatively shallow. It consists essentially of type knowledge for the input and the output, similar to Perry [9]. This knowledge is adequate for the printing implementation but would not be adequate for a more complicated domain such as information retrieval. There is no knowledge encoding other kinds of side effects nor, more important, a representation of some of the underlying objects. For example, the system has no knowledge that the option '-mm' tells 'troff' to open a particular file in a particular directory. If that file is not available, the system has no way of intelligently dealing with that situation or providing any help to the user. A much more comprehensive model of the underlying objects and behavior, along the lines of Wilensky's UC representation

[25], is needed.

The system has no representation for the invocation of a piece of software; it assumes all translators can be included in a UNIX pipeline. In fact, this was not the case with the programs 'latex' and 'dvi2ps', which had to be embedded in other programs for this to be true. The invocation pattern and side effects of using 'latex' are complicated: 'latex' takes a command line argument which is the first part of the file name, and the extension 'tex' is assumed. In addition, several log files are created as a side effect and particular things happen when certain kinds of errors occur. In order for a system to deal with these kinds of things intelligently, it needs a comprehensive representation of the invocation patterns, side effects, and normal behavior of software and software systems. In order to do this, an underlying process model of the entire environment, including the file system, terminal input and output, and storage allocation is needed.

Research into representing knowledge of software and software systems is an important and growing area, and this work has only touched the surface. Several areas for future work suggestion themselves, partly derived from the limitations of the current system. First of all, a more complicated domain is needed. One possibility is to try to apply the software knowledge representation techniques used here to the problem of information retrieval of software, allowing the retrieval of appropriate software modules based on a functional description of the software and its behavior.

In order to do this, more advanced representation techniques will be necessary. These techniques should first be applied to representing the underlying software environment of UNIX: the file system, UNIX invocation patterns, and terminal interactions, all of which can get very complicated. Once the environment is described, software which is embedded in that environment can be more completely represented.

Finally, this work has not been concerned with how people think about software [4]. A more cognitive approach should elucidate the kinds of models people have about software and software systems (how a file gets printed would be a good initial domain) and should also shed light on how people detect and fix various kinds of errors in the process and in their model of the process. Such knowledge should help us in designing knowledge representations of software and systems to use those representations.

7. Conclusion

This paper has described a certain approach to the representation of knowledge about software. We identified a domain, that of printing files in a UNIX environment, which is an good example of a complex software system, yet tractable from the representation point of view. We explored the kinds of knowledge that needed to be represented, and built representations of the different software components of the printing process. The implementation, ESP, used those representations and achieved almost all of the performance goals. Most important, this work serves to highlight the next set of research issues to be addressed in the area of software knowledge representation, which include deeper representations of software objects, including module invocation and side effects; representation of the underlying software environment, including the file system, memory allocation, and the terminal interface; and investigation of human cognitive models of software.

8. Acknowledgements

I would like to thank Ron Brachman, Dewaynne Perry, and Bruce Ballard for reading and commenting on earlier versions of this paper. Special thanks to Mallory Selfridge for several "hard edits" that were instrumental in improving the overall quality of the paper.

9. References

1. Barth, P., Buthery, S., Barstow, D., The Stream Machine: A Data Flow Architecture, 8th International Software Engineering Conference: 103-110, 1986

2. Barstow, D., Automatic Programming for Streams, IJCAI '85: 232-237, 1985

3. Barstow, D., **Knowledge-Based Program Construction**, North-Holland, 1979

4. Bobrow, D., Ed., **Qualitative Reasoning About Physical Systems**, MIT Press, 1985

5. Brownston, L., Farrell, R., Kant, Elaine, Martin, N., **Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming**, Addison-Wesley, 1986

6. Gehani, N., **Document Formatting and Typesetting on the UNIX System**, Silicon Press, NJ, 1986

7. Goldberg, A.T., Knowledge-Based Programming: A Survey of Program Design and Construction Techniques, IEEE Trans. on SE Se-12: 752-768, 1986

8. Kernighan, B. W., Pike, R., **The UNIX Programming Environment**, Prentice-Hall, NJ, 1984

9. Perry, D. E., Software Interconnection Models, Proceedings of the 9th International Conference on Software Engineering, 1987

10. Pesch, H., Shaller, H., Test Case Generation Using

Prolog, in 8th International Conference on Software Engineering p. 252-258, 1985

11. **PostScript Language Reference Manual**, Adobe Systems Incorporated, published by Addison-Wesley, Inc. 1985

12. **PostScript Language Tutorial and Cookbook**, Adobe Systems Incorporated, published by Addison-Wesley, Inc. 1985

13. Preito-Diaz, R., Neighbors, J. M., Module Interconnection Languages: A Survey, TR 189, ICS UCI August, 1982

14. Neighbors, J.M., The Draco Approach To Constructing Software From Reusable Components IEEE Trans. on SE., vol. 10, no. 5: 564-574, Sept. 1984, also in [21], p. 525-535

15. Rich, C., Inspection Methods in Programming, MIT AI-TR-604, 1981

16. Rich, C., The Layered Architecture of a System for Reasoning About Programs, IJCAI '85: 540-546, 1985

17. Rich, C., A Formal Representation for Plans in the Programmer's Apprentice, IJCAI '81:1044-1052, 1981

18. Rich, C. and Waters, R., editors, **Readings in Artificial Intelligence and Software Engineering**, Morgan Kaufman, 1986

19. Ritchie, D. M., Thompson, K. L., "The UNIX Time-sharing System", CACM, July, 1974

20. Swartout, W. and Balzer, B., On the Inevitable Intertwining of Specification and Implementation, CACM 25:438 - 440, 1982

21. Waters, R.C., The Programmer's Apprentice: Knowledge Based Program Editing, IEEE Trans. on SE SE-8: 1-12, 1982

22. Waters, R. C., KBEMACS: A Step Towards the Programmer's Apprentice, MIT AI-TR-753, 1985

23. Wilensky, R., et al., UC - A Progress Report, Report no. UCB/CSD 87/303, Computer Science Division, UC Berkeley, July, 1986

24. Wilensky, R., Aren, Y., Chin, D., Talking to UNIX in English: An Overview of UC, CACM 27/6 574-593, 1984

25. Wilensky, R., Some Problems and Proposals for Knowledge Representation, Report no. UCB/CSD 87/351, May, 1987

26. IEEE Software, Special Issue on Reuse, January, 1988