KNOWLEDGE BASED PROGRAMMING ASSISTANT

by

D. G. Badger
R. Campbell
N. Dershowitz
M. T. Harandi
A. Laursen
R. S. Michalski
D. Michie
R. Penka
M. Simmonds

Proposal to IBM Palo Alto Scientific Center, November, 1980.

UILU-ENG 82 1708

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

Knowledge based programming assistant
(Proposal to IBM Palo Alto Scientific Center: Nov. 1980)

by

D.G. Badger*, R. Campbell, N. Dershowitz, M.T. Harandi, A. Laursen,
R.S. Michalski, D. Michie, R. Penka*, M. Simmonds*

* University of Illinois Computing Services Organisation

FILE NO. UIUCDCS-F- 82-894
April 1982

SUMMARY

The knowledge-based programming assistant (KBPA) is an expert system to aid programmers in the debugging of their programs. This proposal seeks support to study and construct such a system by applying knowledge-base management methods and techniques for computer inference (deductive and inductive) to debugging rules and error statistics. The knowledge will be gathered from expert programmers, professional programming advisors, information supplied by surveys, and analysis of the actual debugging process. Specific goals include:

- Identification and analysis of classes of programming errors (compile-time and run-time) amenable to KBPA.
- Development of decision rules for the diagnosis of these errors, representative of the expertise of experienced programmers and programming consultants.
- Implementation of these rules in rule-processing software systems developed locally and already shown effective in other application domains.
- Use of such implementations to study the particular problems posed by the debugging domain.
- Development of an experimental KBPA for small computers in the field.

2.2 Characterization of Errors by the Time of their Detection.

Two main classes of errors can be distinguished: those that are discovered at run-time and those that are discovered at compile-time. Subclasses of these errors can be distinguished, for example by identifying in which pass of the compiler they were detected.

2.3 Characterization of Errors by the Type of Faults.

Yourdon gives an extensive characterization of program errors. His characterization is partially reproduced below and comments have been inserted to describe their applicability to the Fortran debugging situation:

2.3.1 Logic Errors.

Logic errors are normally the most common type of computer bug and most of our testing efforts are justifiably directed towards these bugs. For the purpose of our discussion, we can assume a logic error to be a solid repeatable bug. If a given test input exposes the presence of a bug, then the same input, when presented to the program a second time, should expose the same bug in the same way.

2.3.2 Documentation Errors.

There are some documentation applications where a programming error can be just as serious as a logic error. In most cases, we would be more concerned with errors in user documentation specifically when the documentation incorrectly tells a user how to prepare input for the program, how to operate the program, and how to use and interpret the output from the program. There are also situations where errors in the technical documentation could be con-

2.3.6 Fall-back and Recovery Errors.

For a number of programs, the concept of recovery and fallback is quite critical. (However, this category does not apply to the Fortran user in the environment in question.)

2.3.7 Hardware Errors and System Software Errors.

(Again, this category is not applicable.)

2.3.8 Standards Errors.

Finally, some people suggest that programs should be tested to ensure that they adhere to various programming standards: that they are modular, well-commented, free of nonstandard programming statements. (Not directly applicable to error diagnosis, although such methods might provide an approach to an expert guiding a user to correct his program see Kernighan and Plauger.)

2.4 Characterization of Errors by the Process in which they are Detected.

Horning provides a characterization of errors based on the process in program debugging when they are discovered.

2.4.1 Inspection.

One way of detecting errors is human inspection of the programs. Not very much is known about the psychology of program readability (see Weissman) but a few general things can be said about inconsistency detection.

Inconsistent items are easier for the human reader to dicover if they are located close to each other. Machines are better at global analysis.

2.4.4 Static Semantic Errors.

These errors may be detected by the semantic routines of a compiler. Unlike syntactic errors, there may be no formal definition of valid programs. Most static semantic errors are detected from the context in which they occur. For example, languages which are strongly typed (Pascal) can detect errors of assignment between variables with different possible ranges of values. Assignment from a real to an integer is invalid in Pascal, for example. The Fortran compiler does not provide much static error detection because of the lack of redundancy in the specification of algorithms in the language. Thus, many static semantic errors that would be discovered in more recent programming languages go undiscovered in Fortran and become 'run-time' errors. Fortran assignment of a real to an integer is valid, for example.

2.4.5 Run-time Error Detection.

Run-time errors are detected by either the run-time system of the programming language or the operating system within whose environment the program is being executed. Such errors may not directly reveal the location of the fault that caused them. Detection of the errors relies on information encoded into the program by the compiler and the environment in which the program is to execute. For example, range checking can be included in the object code of the compiled program and range errors reported to the run-time system. Division by zero or addressing a location outside of the storage area allocated for the program may be detected by the operating system and reported to the run-time system.

future languages where diagnosis by the compiler and run-time system is not possible. In the next section we will discuss various attempts by 'experts' to provide a set of diagnostic and debugging rules for run-time and output errors.

2.5 Distinguishing Error Prone Language Features.

One established method of examining a program for possible errors is to inspect closely constructs that are error prone. Several studies have examined programming language features to abstract general principles about language design and reliability. These provide a useful framework for the design of rules in an expert system. A short list of such analysis is offered below:

GO TO statements (see Dijkstra). In Fortran most of the desired control constructs must be built out of the use of conditional statements and GO TO statements. Knuth points out that certain uses of the GO TO statement are less error prone than others. In particular, use of the computed GO TO and non-nested control flow constructions might be valuable input for an expert diagnosis.

Global variables (see Wulf and Shaw). In particular, the use of COMMON and EQUIVALENCE in Fortran.

Pointers (Hoare). Pointers are not permitted in Fortran, but programs employing list structures constructed from arrays and indices provides a danger area.

Selection by position (Ichbiah and Rissen). Long parameter lists are principal offenders.

debugging tools such as symbolic dumps, traces and interactive debuggers.

Using a storage dump to locate the error is inefficient and difficult for the following reasons. It is difficult to relate a large mass of information to the variables and code of the program. The dump represents a static picture of the program, not the dynamic one which corresponds to the program's execution. The dump is rarely produced at the moment the program generates the error, but only later after the error is detected. Scattering print statements offers little better debugging features. In particular this approach may produce large amounts of information, the extra statements may cause errors, and the method is hit or miss.

The automated debugging tools also generate significant amounts of data and are hit and miss.

There is experimental evidence given by Gould and Drongowski that debugging aids do not 'assist the debugging process, and that, in terms of the speed and accuracy of finding the error, people who use their brains rather than a set of "aids" seem to exhibit superior performance' (see Myers.) Clearly, here is a challenge to test the abilities of 'expert' systems. Brute force methods are counter-productive and debugging requires the application of reasoning. The brute force methods may be successfully used to supplement reasoned debugging of programs.

2.7.2 Debugging by Human-oriented Induction.

The inductive approach to debugging follows the following algorithm.

1) Locate the pertinent data. Enumerate test cases that work and do not work.

2) Organize the data. Structure the data to facilitate detection

2.7.5 Debugging by testing.

After the detection of an error, modify the test case that generated the error slightly with the objective of pinpointing the program logic conditions under which the error occurs.

2.7.6 Further Debugging Principles.

The following comments are also frequently made about debugging (e.g. Myers).

1) Where there is a bug there is likely to be another.

2) Fix the error, not the symptom.

3) The probability of a fix is not 100%.

4) The probability of the fix being correct drops as the program size increases.

2.7.7 On improving error analysis.

It would appear from many articles that it is important to obtain a clear understanding of the problems of a set of users in developing programs in a particular environment. Particular errors occur very frequently and their probability of arising can be a source of information directing the search for the fix. In particular, a careful analysis of debugging activities should record the following kinds of information for study:

1) When was the error made?

2) Who made the error?

3) What was done incorrectly?

4) How could the error have been prevented?

5) Why wasn't the error detected earlier?

2.2) Logic activities out of sequence

2.3) Wrong variable being checked

2.4) Missing logic or condition tests

2.5) Too many/few statements in loop

2.6) Loop iterated incorrect number of times

2.7) Duplicate logic

The error categories provide a very good starting point for the work on an expert system. The symptoms are divided into twenty five categories. The following is a sample of some of these symptoms:

Data Overflow

Incorrect Processing (Incorrect answer.)

Abort

Premature Program Exit

Loop

Too Much Output Produced

The results from their analysis do not encourage the conclusion that diagnosis can be made simply from the symptoms. They conclude that the combinations of symptoms would need to include not only the symptom description, but also certain auxiliary data such as branch execution frequency tables, a list of set/use discrepancies, and other dynamically obtained data.

2.9 Information Available for use by an Expert Debugger System.

The Computer Services Office provides a number of utilities that may be used to obtain information about the behavior of a Fortran program they help satisfy some of the demands required in a diagnosis of program errors. In addition to the compiler error messages, there are three different run-time

2) uninitialized variables.

3) unresolved external references.

4) subprogram calling problems; the wrong argument count, incorrect argument types, incorrect argument dimensions, the changing of constant parameters.

5) misuse of the common block.

6) CDC peculariti:s: necesity to rewind files, need to use OUTPUT statement on PROGRAM statement, DATA statement conflicts with type, array subscript assumptions.

7) use of illegal values (undefined, infinity,etc.)

8) array storage confusion.

More information can be found in the CSO document, CYBER FORTRAN DEBUGGING.

2.10 An Example Expert Guide to Debugging Fortran Programs.

Kernighan and Plauger's "The Elements of Programming Style" provides a very interesting description of the most common failings of Fortran programmers. In this section we examine this book as an example of an expert aiding the user to produce more reliable and 'better' programs. Such a collection of rules might be used to aid the programmer rewriting faulty parts of his software in a manner to eliminate further mistakes. Kernighan and Plauger provide 62 rules that, if followed, will aid in improving the clarity and reliability of a program written in Fortran. The rules provide a guideline for when and how certain (often misused) programming language constructs should be used in an application. In the context of an automated Fortran system, these rules provide a starting point for suggestions to correct a program. They also provide a mechanism to indicate sources of possible errors

4) Watch out for off-by-one errors.

5) Take care to branch the right way on equality.

6) Examine loops that exit to the same place from side and bottom.

7) Make sure your code "does nothing" gracefully.

8) Test programs at their boundary conditions.

9) Check some answers by hand.

10) 10.0 times 0.1 is hardly ever 1.0.

11) Don't compare floating point numbers solely for equality

12) Check array references do not go out of bounds.

13) Do not count with floating point numbers.

3 Knowledge Based Systems.

3.1 Artifical Intelligence Applied to Programming.

Much Artificial Intelligence work has roots in automatic programming. Floyd's proposals were among the earliest of those to envisage a "programmer's apprentice" model. They were followed by a succession of experimental implementations leading to modest but definite practical successes and some unifying principles. Work at MIT by Goldstein, Sussman, Ruth, Waters, and Rich & Strobe, and other work by Stefanescu and Schneiderman & McKay, all gravitated around the debugging problem, classifying this into (a) failures of the program to correspond to the specification and (b) errors of the specification itself. Other, more formal, approaches to debugging include work by Katz & Manna, Sagiv, and Dershowitz & Manna.

A predicate-logic-based stream of ideas which inspired these investigations triggered in the United Kingdom a departure of programming language

been the development by Patricia Goldberg's group of a system for business data processing, in which the task is described in a very high level, non-procedural dataflow language. A program is then generated: if the problem specification was incomplete the user is prompted to provide the missing information, and if the problem formulation is inconsistent the system will conduct a dialogue with the user to define the problem more precisely.

Mention should be made of some recent UK work aimed at automating the construction of commercial data processing programs by the "Michael Jackson" methodology. Jackson's method is a manual technique for constructing programs, starting with specifications of the manual technique of the input and output files. Coleman at UMIST observed that since the file structure could be expressed in BNF, the programs could be synthesized by a syntax analyzer-generator. His co-worker Hughes investigated in more detail the precise class of problems to which this technique applies, showing that automatic generation is possible only if the files can be described by regular expressions. Further work on the automated production of programs for distributed computing systems is in progress.

3.3 Relation to Expert Systems.

The connection between automatic synthesis and debugging of programs and work on "knowledge-based" or "expert" systems arises in two different ways. On the one hand, there is the idea of implementing programming expertise as an interactive system: this is essentially the "programmer's apprentice" concept. Not surprisingly the design principles show essential similarity to those governing the implementation of knowledge-based expertise in other domains of mental skill--clinical consultation, mass spectrometry, plant

4.1 Heuristic Models and Deep Models.

Our plan of work and division of functions follows a particular way of partitioning the structure of expert behavior into two major components, namely a heuristic model and a deep model. Great conveniences attend treating these two models as separate. The dichotomy reflects the fact that (with some variations from one domain to another) a consultant or other person with a highly trained question-answering skill can usually answer the overwhelming majority of client's questions "off the top of his head", just as a Fisher or a Karpov when playing under the time constraints of lightning chess can usually find a master move in five seconds (less than 200 binary decisions in terms of human compute time).

The discovery that a domain specialist's top-of-the-head skill can be mimicked by relatively simple and uniform computational structures, based on pattern-driven situation-action rules, is what has led to the recent rapid development of expert systems such as MYCIN, PROSPECTOR, PUFF, VM, SACON, and others. It is not generally understood that given an appropriate choice of domain (the de-bugging domain is actually unlikley to be such a case) a sufficient level of expertise can be implemented using a heuristic model alone, with no need for the more difficult and costly task of constructing an adequate deep model. By a "deep" model we mean a representation of the domain's logical and causal structure together with procedures for search, calculations, and reasoning, with which recommendations for action may be dug out. In other domains (we believe debugging to be one of them) the ideal system incorporates both types of model as seperate sub-programs, and picks them appropriatley. Some well-studied examples will help.

SKILL	HEURISTIC MODEL	DEEP MODEL
Pole-balancing under constraints	Chambers & Michie's BOXES, 1968. 225 pattern-directed rules.	Eastwood, 1968. Newtonian dynamics plus control theory.
Control of large electrical power systems (Pao et al. Case Western Res. U)	Associative store of pattern-driven rules.	Highly developed logical and numerical model of electrical power system.
Fault-diagnosis in electro-mechanical systems (U Illin.)	Rouse and Hunt's "S-rules" mapping from symptomatic patterns to decisions.	Rouse & Hunt's "T-rules" embodying the logical causal domain structure.

The simplest way to combine the two kinds of model in a single system is for the deep model to be used as default to answer those questions where top-of-the head rules fail i.e. whenever exit occurs from the heuristic model through failure to find an adequate pattern match. Retrospective analysis of the first case tabulated above (pole-balancing) showed that such combination would have greatly improved the economics of each, but in different ways, the run-time skill of BOXES and the run-time costs of Eastwood's program. As is inevitably the case in general, implementation costs of the deep model were large compared with those of the heuristic model.

4.2 Division of Functions and Broad First-Year Objectives.

With this background it becomes straightforward to say who will do what as concerns the technical aspect of the project, thus: heuristic model - Michie (as software manager), Harandi (as assistant manager), Penka and Simmonds (domain specialists for FORTRAN), Campbell (domain specialist for PASCAL), Michalski (knowledge engineer, rule-acquistion), Laursen (knowledge engineer, implementation); deep model - Bershowitz (theoretical basis and trial implementations), Plaisted (theoretical basis).

It should be emphasised here, and will be emphasised again, that

Plaisted).

We propose initially constructing a prototype expert system which will embody a heuristic model of debugging. It will be based on measurements and analyses of programming errors and faults. The intention will be to achieve a quick but convincing demonstration of feasibility in parallel with longer term studies directed towards a "deep" model. The system will be built from locally available software written in Pascal starting with already field-tested versions of AQ11 and INDUCE-1 (Michalski's group) and AL/X and ID3 (Michie's group).

Based on experience with other complex domains, we would expect to build a knowledge-base of several hundred rules before attaining truly expert performance. These rules will for the most part be hand-crafted and tested, although progress with computer induction ("programming by examples") in Michalski's and Michie's groups is such as to offer opportunities for taking some short cuts. The following set of rules illustrate what a "rule" might look like in the AL/X system for building and executing heuristic models. The syntax and composition of the rules are, for the purposes of this exposition, arbitary.

4.3 Sample Consultation.

RULES:

IF error is "integer > maxinteger" THEN there is an uninitialized global variable (.9 probability)

IF error is "integer > maxinteger" THEN there is integer arithmetic overflow (.1 probability)

1) A survey of programming faults and errors generated by programmers of intermediate skills. The survey will be conducted in our local student environment and we expect that the majority of programs will be written by people with one or more years experience in programming. The survey will be performed for two target languages: Fortran and Pascal. The survey material will include analysis of job streams, individual programs, and theoretical analysis of the language.

2) A study of the feasibility of expert debugging systems based upon the results of the survey.

3) Appropriate modification of our current expert system software to accommodate the required work.

4) A working rule base for one of the domains, namely FORTRAN/IBM 4136.

5) The construction of an example rule-based expert that generates advice for a limited, but realistic, subset of errors. Particular emphasis will be placed on developing a 'friendly' and interactive expert system that requires little prior knowledge for its use.

6) A study of the feasibility of a small expert debugging system suitable for mini-computers such as the Series 1 that would support software language products.

7) A theoretical design study of the desirable properties of a "deep model" of de-bugging and a critique from this standpoint of the evolving heuristic model.

8) Quarterly two-page progress reports accompanied by visits to the IBM Palo Alto Center to present progress.

Jouannaudi, J. and Kodratoff, Y., "An automatic construction of LISP programs, by transformations of functions synthesized from their input-output behavior," to appear.

Katz, S. M. and Manna, Z.,"Towards automatic debugging of programs," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975.

Kernigan, B. W. and Plauger, P. J., The elements of programming style, McGraw-Hill Book Company, 1974.

Kernigan, B. W. and Myers, G., The art of software testing, John Wiley and Sons, 1979.

Kowalski, R., Logic for problem solving, Artificial Intelligence Series, no. 7, North-Holland.

Loveman, D. B., "Program improvement by source-to-source transformation," J. ACM, vol. 24, no. 1, pp. 121-145, 1977.

Manna, Z. and Waldinger, R. J., "Synthesis: Dreams => Programs," IEEE Software Engineering, vol. SE-5, no. 4, pp. 294-328, July 1979.

Manna, Z. and Waldinger, R. J., "A deductive approach to program synthesis," TOPLAS, vol. 2, no. 1, pp. 90-121, January 1980.

Rich, C. and Shrobe, H. E., "Initial report on a Lisp programmer's apprentice," IEEE Trans. on Software Engineering, vol. SE-4, no. 6, pp. 456-467, 1978.

Rouse, W. B. and Hunt, R. M., "A fuzzy rule-based model of human problem solving in fault diagnosis tasks," Working paper, Coordinated Science Laboratory, University of Illinois, Urbana, IL.

Ruth, G. R., "Intelligent program analysis," Artificial Intelligence, vol. 7, pp. 65-85, 1976.

Ruth, G. R., "Protosystem 1: An automatic programming system prototype," TM-72, Laboratory for Computer Science, MIT, Cambridge, MA, July 1976.

Sacerdoti, E. D., A structure for plans and behavior, American Elsevier, 1977.

Sagiv, Y., "A study of the automatic debugging of programs," Master's thesis, Weizmann Institute of Science, Rehovot, Israel, August 1976.

Shaw, F. E., Swartout, W. R., and Green, C. C., "Inferring LISP programs from examples," IJCAI, pp. 260-267.

Shneiderman, B. and McKay, D., "Experimental investigations of computer program debugging and modification," 6th Int. Cong. of the Intl. Ergonomics Assoc., College Park, MD, July 1976.

Shneiderman, B., "Perceptual and cognitive issues in the syntactic/semantic model of programmer behavior," Comp. Sci. (W. Camm & R. E. Granda, eds.), pp. 65-77, July 1978.

BIBLIOGRAPHIC DATA SHEET	1. Report No. ~~UIUCDCS-P-82-894~~ Working Paper KBPA-1	2.	3. Recipient's Accession No.
4. Title and Subtitle Knowledge based programming assistant			5. Report Date April 1982 6.
7. Author(s) D.G. Badger*, R.Campbell, N.Dershowitz, M.T.Harandi, A.Laursen, R.S.Michalski, D.Michie, R.Penka*, M.Simmonds*			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Dept. of Computer Science University of Illinois, Urbana, IL. 61801 * Iniversity of Illinois Computing Services Organisation			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address IBM Scientific Center, 1530 Page Mill Road, Palo Alto, CA. 94304			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

A proposal to develop programming tools to aid in the location and correction of errors in programs. A heuristic model will be developed to aid the identification of compile-time errors(and run-time errors which can be diagnosed "Symptomatically") and a " deep" model to identify logical errors.

17. Key Words and Document Analysis. 17a. Descriptors

debugging
heuristic model
deep model
logic errors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
internal distribution members of KBPA project	UNCLASSIFIED	35
	20. Security Class (This Page) UNCLASSIFIED	22. Price