KNOWLEDGE BASED PROGRAMMING ASSISTANT, KBPA-1

by

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

Report No. UIUCDCS-F-82-894, Department of Computer Science, University of Illinois, Urbana, April 1982.

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.

1 Introduction.

Program debugging is a time and resource consuming activity. Debugging a program cannot be entirely separated from the problem of understanding it, and program understanding requires the ability to form descriptions. While formal semantic descriptions and formal program verification describe properties of programs, such techniques are not easily applicable to aiding the debugging of programs written by application programmers who have not had formal computer science education. We propose applying knowledge-base management methods and techniques for computer inference to develop an expert system for assisting programmers debugging their programs.

A key issue to the application of expert systems to debugging is the categorization and description of programs and programming errors.

2 A Classification of Debugging.

In this section we examine the approaches to debugging that have been suggested in the literature, discuss errors and various ways to categorize them, and discuss various approaches written by 'human experts' that attempt to help the Fortran programmer.

2.1 The Nature of Errors.

Errors can be characterized by the time of their detection, by the type of faults that cause them, by the process in which the errors are detected, and by the techniques that can be used to avoid them.

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. Sub-classes 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 considered critical.

2.3.3 Overload Errors.

It is often important to test a program to find out what happens if various internal tables, buffers, queues, or other storage areas are filled up to or even beyond their capacity. (Overload errors may well occur in programs used by Fortran application programmers.)

2.3.4 Timing Errors.

This is a category that is usually relevant to real-time systems. (It is unlikely that these errors will occur in user programs in the local IBM Fortran environment.)

2.3.5 Throughput and Capacity Errors.

Once again, this is a category that may be relevant only for real-time systems, although it seems that more batch-oriented programs should be tested in this area. We are concerned here about the performance of the program; even though it produces the correct output, it may take an unacceptable amount of CPU time to do so, or it may use an exorbitant amount of memory, disk space, etc. This is critical for many on-line systems because the performance of a program is often immediately visible to the user in terms of response time. In a batch program, we might still want to specify (and then test) that the program be able to process one transaction per second, that it will take no more than 100,000 bytes of storage, and so forth.

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.

It is easier to find inconsistent pairs of items than large collections of items.

Direct inconsistencies are easier to detect than those that require long chains of inference.

2.4.2 Lexical and Spelling Errors.

Some faults generally caused by mechanical errors in program preparation can be detected purely by the lexical analysis within a compiler. Such faults may appear as ill-formed tokens or as delimiter faults where tokens begin and end with particular symbols (e.g. comments and strings). Many mechanical faults in program production lead to spelling errors, in which tokens are well-formed, but undeclared, identifiers. Morgan claims that 80% of the spelling errors in programs involve insertion, replacement, or deletion of a single character or the tranposition of a pair of characters. In Fortran, spelling errors appear as implicit declarations and may not be detected by the lexical analyser.

2.4.3 Syntactic Errors.

Syntactic errors are the focal point of error detection and diagnosis within compilers. Because syntactic specifications are precise, it is possible to develop parsers that accept exactly the specified langaguages; because they are formal, it is possible to prove that parsers detect any syntactically invalid programs. It is assumed that syntactic errors will not play a major role in an expert consulting system for debugging Fortran programs.

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.

2.4.6 Output Errors.

Despite the successful execution of a program, the output from the program may incorrect. In general, these errors are the most difficult to correct.

2.4.7 Undetected Faults.

Lastly, for completeness we must mention that some faults within a program will generate errors that are not detected, until some very special combination of the input parameters is used.

2.4.8 Error Detection and Correction.

Several studies (Cannon 1975) have shown that the earlier an error is detected in the process of compiling a program the faster it can be corrected. The following results were obtained for the average persistence of faults associated with the various classes of errors in two example programming languages.

lexical errors	1.00 run
syntactic errors	1.34 runs
semantic errors	1.24 runs
run-time errors	5.78 runs
output errors	8.52 runs

The current trends in the development of programming languages is towards producing more reliable languages which detect errors as early as possible. Clearly, the results from this study demonstrate the importance of a good diagnosis and treatment for run-time and output errors in existing and 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.

Assignment statements in their unrestricted form (Strachey). Fortunately, this does not apply to Fortran.

Defaults and implicit type conversions (Hoare). They hide too many program faults. This is a difficult problem in Fortran.

Duplication (Clark and Horning), useless redundancy at its worst. For example the EQUIVALENCE and COMMON statements of Fortran.

2.6 Anomaly Detection.

One approach to the certification of software that has been proposed in industry (Stucki, Osterwiel and Taylor) is to examine the program for peculiar use of programming constructs. Examples of program anomalies are sections of code that are inaccessible, uninitialized variables and infinite loops. Whether this approach might prove useful to an 'expert' debugger might be considered.

2.7 Approaches to Debugging.

The major problem in debugging, perhaps representing 95% of the effort, is the location of the fault in the program that lead to a particular error. Myers categorizes the activity of debugging into several approaches.

2.7.1 Debuging by Brute Force.

This approach is characterized by attempting to debug using the least mental effort and relying on the computer providing information. Examples of techniques in this category are debugging using a storage dump, debuggingg by scattering print statements through the program, and debugging with automated 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

of patterns. Contradictions are frequently important clues. One organizational technique proposed is "The Method" (Brown and Sampson). This approach organizes symptoms into a table with one axis specifying when, where, what, and to what extent the symptoms occur and the other axis describing contradictions.

3) Devise a hypotheses. Study the relationships and clues and form one or more hypothesis about the reason for the error. If a theory cannot be developed devise some more test data. If several theories are devloped, select the most probable first.

4) Prove the hypothesis. Compare the hypothesis with the original data and ensure that it accounts for all the symptoms. Avoid fixing just the symptoms.

5) Now fix the fault in the program.

2.7.3 Debugging by deduction.

An alternative approach to debugging is to use the following algorithm:

1) Enumerate the possible causes and form a hypotheses.
2) Use the data to eliminate possible causes.
3) Refine the hypotheses.
4) Prove the hypotheses.

2.7.4 Debugging by backtracking.

Starting from the detection of the error, incorrect results are used in a backtracking algorithm through the logic of the program. Useful for small programs.

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?

6) How could the error be detected earlier?

7) How was the error found?

2.8 Diagnosis of Software Errors from Symptoms of an Error.

This leads on naturally from the preceding section and has special relevance to heuristic models of de-bugging (see later). Thayer et al. describe several measurements made of software reliability. In particular, in one analysis of Fortran and Assembly Code an attempt is made to categorize a set of symptoms and to determine the probabilities that, given a set of symptoms, a diagnosis can be made of the original programming fault. The causes of an error are divided into twelve categories which are subdivided into further subcategories. Examples of the categories and subcategories chosen are:

1) COMPUTATIONAL ERRORS

1.1) Incorrect operand in equation

1.2) Incorrect use of parenthesis

1.3) Sign convention error

1.4) Units or data conversion error

1.5) Computation produces over/under-flow

1.6) Incorrect/inaccurate equation used

1.7) Precision loss due to mixed mode

1.8) Missing computation

1.9) Rounding or truncation error

2) LOGIC ERRORS

2.1) Incorrect operand in logical expression

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 system aids that gather information:

1) PMD: An error recovery package that attempts to track down the source of an execution error. PMD generates a symbolic trace of a Fortran program. The values of variables may be examined including the values of arrays.

PMD is distributed by CDC and is based on the Leicester Trace Package.

2) Fortran Trace and Debug Directives Six options are permitted and recognized by the Fortran compiler and can be used for debugging. These six options are: ARRAYS, turns on subscript checking, CALLS traces calls to and returns from subroutines, FUNCS which is similar to CALLS except it applies to functions, STORES which traces stores into individual variables, GOTOS checks assigned GOTO statements, and TRACE which provides a trace of gotos, computed gotos, arithmetic IFs and the true side of logical IFs.

3) The CYBER Interactive Debug allows the interactive setting of breakpoints and the examination of the contents of variables using their symbolic names. The interactive debug system is more expensive and may not yield itself to further analysis by an expert system.

A document produced by CSO describes some common causes of error that have been noted to occur by CSO staff. In particular, several of the errors are directly related to the environment in which the Fortran programs are executed; notably the Cyber.

1) invalid subscripts.

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 pecularities: 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 and ways for novice programmers to develop their programming technique. For example, questions of the form:

Did you write this program yourself?

How much programming experience have you?

Does your program include a computed GOTO?

might reveal a naivety in the programmer that is best referred to an instructor. Similarly for an 'expert' consultation that has established that the program entered an infinite loop and that it is the programmer's first real number computation assignment an approprate enquiry might be whether real numbers are used in any comparison for equality. Input to and output from a program are frequently the source of error. Kernighan and Plauger apply several important principles to this area which might be generalized in an 'expert' system:

1) Check input data for validity and plausibility

2) Make sure that data does not violate the limits of the program.

3) Read input until end of file, not by count.

4) Identify input errors and recover if possible. Do not stop on the first error. Do not ignore errors.

5) Use mnemonic input and output. Make input easy to prepare (and easy to prepare correctly). Echo the input and any defaults onto the output; make the output self-explanatory.

A separate section in the book outlines common blunders. Kernighan and Plauger list the following common sources of errors:

1) Variable Initialization

2) Don't stop examining the program after finding one error.

3) Use the debugging facilities of the compiler.

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 design in which the distinction between program-execution and automatic theorem-proving is partly obliterated. Relevance to the automatic programming problem has recently been accentuated by the demonstration by Clark & Tarnlund that a PROLOG program can be proved corect by deducing it as a theorem from its formal specification. Other recent work on formal derivation of programs includes Manna and Waldinger. For a survey of program synthesize methods see Biermann. As a correction to over-emphasizing purely deductive paths, recent projects by Biermann, Summers, Jouannaud & Kodratoff, and Shaw, Swartout & Green demonstrated synthesis of correct LISP programs not by deduction from the formal specification but by computer induction from examples and counterexamples of the transformation to be programmed.

Recent years, with increasing complexity of the tasks attempted, have seen more stress on the role of machine-stored organized knowledge. The PSI system (a typical "Expert System") is the most carefully worked out experimental approach to date. The original trial domain was sorting; more recently the complex task of synthesizing a program for concept-forming was tackled with substantial success. Other work on program understanding includes Wilcox, Davis & Tindall.

Program debugging is related to the recently popular field of program transformations, see e.g. Darlington & Burstall, Loveman, Standish et al., and Wegbreit. However, most of this work has taken a formal approach.

3.2 Commercially oriented work.

One of the main centers of work on commercially-oriented automatic programming is at IBM, Yorktown Heights. A major project for some years has 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 pathology, etc. This similarity of structure is seen particularly clearly in Green's PSI system mentioned earlier.

On the other hand, many expert system projects in applied science at some stage develop a need to provide the domain-specialist with sufficient automatic programming aids to lighten his task in transferring to the machine the bodies of knowledge necessary for its decision-taking. Examples are the Meta-DENDRAL module of the DENDRAL chemistry system and Michalski's AQ11 inductive inference program used for determining decision rules for the PLANT/DC system (soybean diseases). A recent exercise in the same vein was undertaken by T. Niblett and A. Shapiro using Quinlan's ID3 algorithm to machine- synthesise a classification program for a difficult-to-program chess endgame. A need for program synthesis also arises in a slightly different sense when the expert skill deployed by the knowledge-based system has the construction of plans of action as an important component of its consultative task. An example is the SECS program for planning organic chemical syntheses developed by Tod Wipke; other cases arise in planning manipulative movements by assembly robots, in scheduling problems, in computer game-playing and in other applications where means-ends analysis and attainment of goals are central.

A distinguishing feature of our proposal is that this connection comes in twice over. Parts of the system s programming expertise will have been built with the aid of parts of this same expertise.

4 Proposed System.

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
Bicycling	What every bicyclist "knows" (he doesn't really, but relies on subliminal triggering of learned stimulus-response bonds)	Newtonian dynamics plus control theory
Answering questions in arithmetic	What every calculating prodigy knows (mainly in the form of EMYCIN-like situation-action rules)	Axiomatisations such Peano's plus inference procedures.
Meningitis diagnosis and prescribing	What every specialist knows (well modeled by MYCIN)	Anatomical, physiological fluid-dynamical, and pharmacological model of the brain; plus biophysics, chemistry (molecular, organic, inorganic) of pathogens and drugs; plus logical model of everyday life sufficient for interpreting patient histories.

The three examples have been deliberately choosen for being tractable by use of heuristic models. In the present state of the computing and domain-specialist arts, all three are probablly intractable by implementation of deep models alone. The following three examples are different. Here one and the same problem has in each case provoked successful implementations using both kinds of model.

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 <u>Division of Functions and Broad First-Year Objectives.</u>

With this background it becomes straightforward to say who will do what as concerns the technical aspect of the project, thus: <u>heuristic model</u> - 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); <u>deep model</u> - Dershowitz (theoretical basis and trial implementations), Plaisted (theoretical basis).

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

Independently of the recruitment to be made possible by the funds requested the above listing of human resources greatly understates what will over time become available. Already students and teaching assistants under the care of Campbell and of Michalski, who are not named in this document, have embarked on systematic collection of empirical debugging and bug-incidence data. As the project moves into action and its solid financial base becomes known in the Computer Science Department it is bound to exert a gravitating influence on the more active-minded component of our large population of graduate students. This is where much of the implementational hard work will come from, supplementing that of the M.S. student who has already started, the posts for which salaries are requested, and the work contributed by the professorial investigators themselves. For this and other reasons we feel confident of having a heuristically based expert system to show before the end of the first year which will clinch the feasibility of useful user support. During the same period we expect a theoretically based design study to have been completed as the basis for building a deep model, starting in year 2, capable of giving powerful default back-up to the evolving heuristic model. We currently believe that in year 2, given continuation of funds, the heuristic model should be re-vamped to yield a version for PASCAL which by then will have attained solid and fully documented form on the Series 1, complete with conventional debugging aids. Evolution of the model for IBM FORTRAN will continue, and at the heuristic level will inevitably diverge. The degree to which it will then be feasible or desirable for the deep model to "know about" both, or only one, of the divergent heuristic models will form part of the theoretically oriented studies of the first year (Dershowitz, 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)

IF integer arithmetic overflow AND there is a loop that contains a recurrence THEN loop is causing problem (.5 probability)

IF there is integer arithmetic overflow AND there is no loop causing the problem THEN error is caused by exponentiation OR repeated multiplication (.5 probability)

The rules might be applied by the AL/X system to produce the following scenario:

Did you get any run-time error messages? YES

What was it? integer > maxinteger

Do you have a loop in your program that could be infinite? NO

Do you have a loop in your program that has a recurrence with large integer constant coefficient? NO

Do you use exponentiation in your program? NO

Did you have any other messages? NO

I believe (.99 percent probability) that you have an uninitialized global variable in your program.

4.4 Statement of Work.

In the first funding period, we propose:

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.

References

Biermann, A. W.,"Approaches to automatic programming," Advances in Computers, vol. 15, Academic Press, New York, pp. 1-63, 1976.

Biermann, A. W., "The inference of regular LISP programs from examples," IEEE Trans. on Systems, Man and Cybernetics, vol. SMC-8, pp. 585-600, 1978.

Brown, A. R. and Sampson, W. A., Program debugging, MacDonald, London, 1973.

Clark, K. and Tarnlund, S., "A first-order theory of data and programs," Information Processing (ed. B. Gilchrist), pp. 939-944, 1977.

Davis, R., "Generating correct programs from logic specifications," Ph.D. Thesis, University of California, Sant Cruz, CA.

Darlington, J. and Burstall, R. M., "A system which automatically improves programs," Acta Informatica, vol. 6, no. 1, pp. 41-60, 1976.

Dershowitz, N., "The evolution of programs," Ph.D. Thesis, Weizmann Institute of Science, Rehovot, Israel, 1979.

Dershowitz, N. and Manna, Z., "The evolution of programs: Automatic program modification, IEEE Trans. on Software Engineering, vol. SE-3, no. 6, pp. 377-385, 1979.

Eastwood, E., "Control theory and the engineer," Proc. IEE, 115, pp. 203-211, 1968. 1968.

Floyd, R. W., "Toward interactive design of correct programs," Proceedings Information Processing Congress, Ljubljana, Yugoslavia, pp. 7-10, August 1971.

Goldstein, I. P., "Summary of MYCROFT: A system for understanding simple picture programs," A.I. Memo, MIT, Cambridge, MA, May 1974.

Gould, J. D. and Drongowski, P., "A controlled psychological study of computer program debugging," RC-4083, IBM Research Division, Yorktown Heights, NY, 1972.

Gould, J. D., "Some psychological evidence on how people debug computer programs," Int. J. Man-Machine Studies, vol. 7, no. 2, pp. 151-182, 1975.

Green, C. C., "The design of the PSI program synthesis system," Proceedings of the Second International Conference on Software Engineering, San Francisco, CA, October 1976.

Green, C. C. and Barstow, D., "On program synthesis knowledge," Artificial Intelligence, vol. 10, no. 3, pp. 241-280, 1978.

Horning, J. J., "Programming languages," Advanced Course on Computing Systems Reliability, University of Newcastle upon Tyne, 1978.

Ingevaldsson, L., Jackson structured programming: A practical method of program design, Input Two-Nine Ltd.

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.

Shneiderman, B. and Mayer, R., "Syntactic/semantic interactions in programmer behavior: A model & experimental results," J. Comp. & Inf. Sciences, vol. 8, no. 3, pp. 219-238, 1979.

Stallman,R. M. and Sussman, G. J., "Forward reasoning dependency-directed backtracking in a system for computer-aided circuit analysis," Artificial Intelligence, v. 9, no. 2, pp. 135-196, October 1977.

Standish, T. A., Harriman, D. C., Kibler, D. F., and Neighbors, J. M., "Improving and refining program by program manipulation," Memo, Department of Information and Computer Science, University of California, Irvine, CA, February 1976.

Stefanescu, D. C., "Interactive computer-aided debugging," TR-06-78, Center for Research in Computing, Harvard University, Cambridge, MA, 1978.

Stefanescu, D. C., "More on debugging of programs," Memo, Center for Research in Computing, Harvard University, Cambridge, MA.

Summers, P. D., "A methodology for LISP program construction from examples," J. ACM, vol. 24, pp. 161-175, 1977.

Sussman, G. J., A computer model of skill acquisition, American Elsevier, 1975.

Thayer, T. A., Lipow, M., and Nelson, E. C., Software Reliability, North-Holland Publishing Company, pp. 42-46, 160-162, 1978.

Waters, R. C., "A method, based on plans, for understanding how a loop implements a computation," WP 150, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1977.

Wegbreit, B., "Goal-directed program transformation," Conference Record, Third ACM Symposium on Principles of Programming Languages, Atlanta, GA, pp. 153-170, 1976.

Weinberg, G. M., The psychology of computer programming, Van Nostrand, New York, 1971.

Wertz, H. A., "A system to improve incorrect programs," Proc. 4th Software Engineering, pp. 286-193, 1979.

Wilcox, T. R., Davis, A., and Tindall, M., "The design and implementation of a table driven, interactive diagnostic programming system," Comm. ACM, vol. 19, pp. 609-616, 1976.

Pao, Y-H., "A knowledge based engineering approach to power systems monitoring and control," Dept. of E. E., Case Western Reserve University, Cleveland, 1981.

Yourdon, E., Techniques of program structure and design, Prentice-Hall, 1975.

Yourdon, E. and Constantine, L. L., Structured Design, Englewood Cliffs, NJ, Prentice-Hall, pp. 502, 504-5, 511, 1979.