THE APPLICATION OF ARTIFICIAL INTELLIGENCE TECHNIQUES TO SOFTWARE MAINTENANCE

by

WAYNE LOUIS WERBELOW

B.S., University of Wyoming, 1979

A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1985

Approved by:

Roger T. Hartley

Major Professor

CONTENTS

LIST OF FIGURES

1. Introduction

The rising cost of computer software development and specifically software maintenance has generated much concern in recent years. Maintenance is now being viewed as a major limiting factor on the capabilities of software systems for several reasons. First, even though the cost of computer hardware has declined, the cost of professional programmers has increased dramatically. Second, with the programming projects becoming more and more complex, extra programmers are being added to maintain software. Third, more technical background knowledge is being required to make the correct changes to software.

Applications using Artificial Intelligence (AI) techniques have grown significantly in recent years. Intelligent tools have been designed to aid doctors in treating their patients, geologists in drilling for oil, engineers in finding faults in computer systems and scientists in identifying organic compounds [Kobler82]. All of these intelligent tools have a common underlying concept which allows them to handle a mixture of symbolic and heuristic information. So why not apply AI concepts toward helping programmers to program? Researchers in this area have been concentrating on designing knowledge-based intelligent tools, also known as expert systems, which coordinate the individual steps of software development - requirements, specifications, design, code generation, testing and maintenance.

This paper presents a viewpoint of what computer software

maintenance is and how AI techniques might be applied as tools to aid computer programmers in the maintenance of software. Jim Gallagher, in a joint project effort, will focus on the applicability of AI techniques to requirements engineering [Gallagher85].

2. Definitions and the Need for Intelligent Tools in Software Maintenance

This section describes the terminology used in the remainder of this paper. It also explains why intelligent tools are needed to aid in the maintenance of software. First, the traditional software development life cycle is discussed. Most of the tools presented in this paper follow this paradigm. Other tools, like the User Software Engineering (USE) Environment [Wassermen83] and the Knowledge-Based Software Assistant (KBSA) [Green83] follow the rapid prototyping methodology [Connell84]. Rapid prototyping generates the actual program code, through user interaction on a prototype system, from high-level requirements and specifications. Next, software maintenance and its problems are described. Artificial intelligence techniques are described next. These include knowledge-based systems and expert systems. Finally, the reasons why software maintenance would benefit from the use of artificial intelligence tools are explored.

2.1. The Software Development Life Cycle

Software maintenance is just one phase of the software development process. This process is traditionally called the software development life cycle and is comprised of three main phases each of which is broken into various steps [Pressman82]. Phase one is the planning phase and consists of a software plan (feasibility study), requirements analysis (problem identification) and specification (stated solution to the problem). The second phase is the development phase which consists of preliminary design (module definitions, interfaces and data structures), detailed design (procedural descriptions), coding (generation of the software) and testing (unit, integration and validation testing). The third and final phase in the software development life cycle is the maintenance phase (ongoing modification of the software). Many times these steps are not sequentially followed and can overlap (e.g. some code for a program could be written before another program is finished with detailed design). Also, feedback loops from one step to a previous one can result (e.g. problems found during testing could relate back to code changes or even specification and design changes).

2.2. What is Software Maintenance?

"Software maintenance is all those activities associated with a software system after the system has been initially defined, developed, deployed, and accepted as operational." [Dean82] Software maintenance comprises the final phase in the

software life cycle. It is, by far, the most costly and lengthy of all the steps in the life cycle.

The following sections describe the similarity of the maintenance life cycle is to the software development life cycle, the reasons behind maintenance and the current software maintenance environment.

2.2.1. The Software Maintenance Life Cycle

The maintenance phase, although usually defined as being in the software life cycle, can also be looked upon as having its very own life cycle. It can be divided into phases and steps which parallel the software development life cycle. The steps include the following:

1. Re-negotiate the software plan - Is the change feasible? Can it be handled manually or not at all?
2. Re-analyze requirements - identify the problem with the software.
3. Re-specify - identify what the solution will be.
4. Re-design - add more detail to the solution and identify which modules or data structures will change.
5. Re-code - Make the changes to the programs noting that old code may have to change and new code may be added.
6. Re-test - Verify that the solution meets the changed specifications and that it hasn't affected other programs or data indirectly.

Like the software development life cycle steps, the maintenance steps can overlap and include feedback loops to previous steps. In practice, these formal steps are usually not followed and the initial steps are often times ignored completely. This may be advantageous if the change is simple, especially when trying to reduce the time and cost of maintenance. If the change is more complex and possibly modifies the specifications, then the various steps should be followed.

2.2.2. Why is maintenance done?

Most software developments are expensive. So once the development is done and a bug is found or enhancements are requested, then it would be extremely expensive and time-consuming to throw away the software and start over. Therefore, the software system needs to evolve and change over time.

Usually maintenance is associated with a modification request to the software system from users or others affected by the system. This modification request can take on many forms. First, the system needs to be fixed (a bug was found which did not follow the specification or was implemented wrongly). Second, the environment of the system changed (e.g. the operating system changed). Third, the system needs to be rewritten to make it structured or more efficient. And lastly, the system's requirements or specifications have changed (been enhanced or due to weak initial specifications).

This last modification type accounts for almost half of all software maintenance activity [Pressman82].

2.2.3. The Software Maintenance Environment

Software maintenance is performed in a more dynamic environment than other phases of the development cycle - mainly due to its duration and the fact that changes are constantly being made to the software. This can create special problems for the software maintenance programmer.

To most computer programmers, software maintenance is the least desirable function in the software development cycle. Frequently, as soon as a software system is developed and implemented, the designers (usually senior programmers) announce to management that they do NOT want to be placed on a maintenance team. Even if they are assigned to a maintenance task and it has a poor environment (i.e. primitive editors, languages, and compilers), then the good programmers are likely to look for another job elsewhere. Thus, the original designers may no longer be present or accessible and programmers with less expertise must maintain the system.

Another aspect of the maintenance environment is that frequently there is little or no documentation to support the programs. Requirements, specifications and implementation considerations may not be included in the system documentation because they were conveyed in memos or merely by discussions between the designers. Also, if the programs are modified due to changes in requirements or specifications, then the

documents supporting the requirements and specifications should be updated. This is rarely done due to time limitations, lack of interest or because "nobody reads those things anyway!"

The administration of software changes can range from one person making a change, cutting the program into production and not telling anybody about it, to a preferred but complicated and time-consuming modification request procedure. The latter procedure consists of the following steps:

1. A request from the users is entered into the problem tracking system. This system accesses and controls the modification requests stored in a database or file.
2. The request is assigned to a programmer. (Sometimes this is done arbitrarily, so it may get re-assigned later.)
3. The programmer decides when the change can actually be made. (This, in itself, is one of the most difficult decisions a programmer has to make.)
4. The maintenance life cycle is adhered to by generating all the necessary documents, getting the appropriate signatures and, of course, making the change to the program.
5. Finally, the users and the other programmers are informed about the change.

2.3. Artificial Intelligence Techniques

"Artificial Intelligence (AI) is the science and art of automating problem-solving processes that are informal, heuristic and symbolic in nature. The simplest definition of AI is any activity that is performed by a non-human entity (typically a digital computer) and that is usually considered to require intelligence when performed by humans." [McCune83]

The concept of AI has been in existence for about twenty-five years. It grew from theories of mathematical logic and computation to theories of cybernetics and self-organizing systems. AI is entering a new generation which consists of practical applications that are starting to reach the marketplace.

AI techniques such as heuristic reasoning (i.e. "rule of thumb" versus strict algorithms), learning, natural language understanding and knowledge-based representation are being applied as intelligent tools to many different problem areas such as image processing, speech recognition, medical diagnosis, oil drilling and automatic programming. These intelligent tools are comprised of knowledge-based systems and expert systems, which try to understand the problem and give a solution based on the information in their knowledge base and some governing rules or inferences. These are discussed in the next sections.

2.3.1. Knowledge-based Representation

Knowledge representation has been defined as "a combination of data structures (declarative) and interpretive

procedures (procedural) that, if used in the right way in a program, will lead to 'knowledgeable' behavior" [Barr81]. The data structures are usually represented internally by facts or rules in syntactic (formal notation), semantic (related to the meaning) and pragmatic (application specific) formats. A regular database contains only raw data. A knowledge base contains data also but has an inference mechanism to interpret the data into a knowledgeable form [McCalla83].

2.3.2. Expert Systems

Expert systems are knowledge-based systems which are "programs capable of simulating the problem-solving behavior of a human planner by using rules derived from previous experience and empirical observation" [Dyer84]. Expert systems differ from regular programs in the way they are organized. Regular programs consist of structured algorithms and data. Human knowledge, on the other hand, is not as structured. It consists of basic fragments of 'know-how'. These fragments are applied in new and different ways to derive intelligent human decisions. Expert systems, to simulate human knowledge, must retrieve information from a knowledge base using these fragments of knowledge [Hayes84].

A human expert who seems to have all the right answers will use a knowledge base also. The knowledge base in this case may be the expert's brain or it may be books, documents, other experts, or a computer. No matter what the knowledge base is, the expert knows where to find the answer. When

knowledge is transferred from a human expert to another human, it may be forgotten or interpreted wrongly. Yet, if this same knowledge could be transferred from the human expert to a computer that could represent knowledge, then another person could acquire the knowledge and use it. This would make the expert knowledge more widely accessible.

Expert systems have been applied to different disciplines. Table 1 shows just a few examples of expert systems [Kobler82], [Frenkel85] and [Goering84].

2.4. Motivation for Intelligent Tools in Software Maintenance

A good tool will usually help in getting the job done faster and cheaper and help control consistency between different aspects of the job. This is very true when applying tools toward software development and especially maintenance. Current (non-AI) tools like text editors, compilers and debuggers have helped the software industry immensely. Yet, these alone are not going to be good enough. The need for intelligent tools exists because of rising costs, deteriorating quality and to reduce the mundane maintenance tasks commonly required by a typical programmer.

2.4.1. Software Maintenance Cost

The cost of maintaining software has skyrocketed in the past few years mainly due to the rising cost of skilled programmers and the amount of time and effort needed to

TABLE 1

AREA OF APPLICATION	NAME OF SYSTEM	DESCRIPTION
Communication	ACE	Aids in telephone cable maintenance.
Education	GUIDON	Teaches by eliciting and correcting answers to a series of questions.
Engineering	DART	Aids in the diagnosis of computer system faults.
	Drilling Advisor	Diagnosis and correction of oil-well drilling problems.
	XCON	Configures computer systems for DEC.
Geology	PROSPECTOR	Aids in locating potential mineral deposits.
Knowledge Engineering	ART	A frame-based and rule-based expert system development tool.
	EMYCIN	A rule-based consultant derived from MYCIN.
	KEE	A framed-based and rule-based expert system development tool.
	S.1	Easy-to-use frame-based expert system development tool.
	TEIRESIAS	Assists in transferring knowledge to a system from a human expert.
Mathematics	MACSYMA	Aids mathematicians with tasks such as polynomial factoring and symbolic integration.
Medicine	CADUCEUS	Assists diagnoses in internal medicine.
	MYCIN	Identifies certain bacterial infections and suggests treatment. Uses "judgmental" knowledge.
Science	COPY	Looks at a complex tower of blocks and builds a mirror image of it using "vision" and a mechanical arm.
	DENDRAL	Helps in the identification of organic compounds.

maintain the software. These costs are usually measured in millions of dollars for large software systems and in billions of dollars for the U.S. government software systems. Software maintenance costs usually range from 60% to 75% of the total software development costs [Dean82]. To reduce these costs, easy-to-use intelligent tools must be created to improve the productivity of software programmers.

2.4.2. Software Maintenance Quality

Improvements in software quality could potentially be a side-effect of introducing intelligent tools for maintenance due to more consistent programming practices, more structured code, and better program and documentation control. Since everyone will be using the same environment and specifically, the same intelligent tools to modify and generate new code, then most programming errors will be found before the program is implemented into the production environment.

2.4.3. Mundane Programming Chores

Mundane programming tasks can take up a large portion of a programmer's time and effort. These tasks include keeping track of all the different versions of programs, knowing the correct syntax of the programming language(s), remembering the many commands of the program editor(s), documenting programs and using debuggers and compilers. The mundane tasks could begin to disappear when intelligent tools are used. An

intelligent program assistant can help a programmer in many ways.

1. Error Checking - misspellings, wrong parameters. Current systems just check for syntax.
2. Questions Answered - Before making a change, the programmer may ask, how is this program called and what are the parameters?
3. Trivia - The programmer wouldn't need to know all the little facts and intricacies of the program.
4. Debugging - A programmer could ask, which procedure set a certain variable to 3? The tool would execute a trace and then possibly backtrack to find the answer.
5. Program Tracking - Keep track of all current versions of programs and archive old versions.

These helpful aids could allow extra time for the programmer to become more creative in solving the initial problem by simplifying the design, code and modification of the software.

3. Historical Background and Current Research

When talking about computer programming and especially intelligent tools to aid programming, historical background and current research actually start to blend together. Since many projects described in this paper may not be operational at this time, the reference to current research actually refers to the time the article was written.

This section describes intelligent program assistants,

programming environments and miscellaneous maintenance tools which use Artificial Intelligence techniques.

3.1. Intelligent Program Assistants

Tools which help programmers edit programs have been around for some time. Initially they were very primitive allowing only one line to be modified and having a limited scope of commands. Now program editors allow for full screen editing and include a large amount of commands (sometimes too many for most people to remember). Examples are the Visual Editor (vi), used with the UNIX[1] operating system, and IBM's Structured Programming Facility (SPF) running under the MVS operating system. Although these editors are nice and make editing of programs easier, they don't do any syntactic or semantic checking. Some editors do allow the programmer to build macros or supersets of commands to make editing more efficient and to perform checking of the program's syntax.

An analogy has been made concerning program assistants and typists [Shapiro84]. Basic editors are like a non-English speaking typist. The program's text is entered with a word-for-word copy and no changes are made from the original. An English speaking typist would be similar to a syntax-oriented editor which can find misspellings and syntactical errors. Compilers do this for programmers now, but it is usually done after the editing session, so if an

---

1. UNIX is a trademark of AT&T Bell Laboratories.

error shows up on the compiled listing, the programmer has to re-edit the program. A more intelligent program assistant could mimic a typist who is also an English teacher. It would know about the domain of programming on a specific programming language and make general programming suggestions, catch certain types of semantic errors, help stylize and improve the overall flow of the program. Finally, if the program assistant is familiar with the application that the program is dealing with, (similar to a typist who has the same knowledge or background as the author), then it could help develop algorithmic structures and catch certain types of pragmatic (or application specific) errors. The program assistants described in this paper are of the type associated with the typist as an English teacher.

3.1.1. Programmer's Apprentice

The Programmers' Apprentice is the classical example of an intelligent tool to aid programmers with the task of programming. It is being designed and implemented by Charles Rich [Rich78], Howard Shrobe and Richard Waters [Waters82] at the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology (MIT). The basic intent of the Apprentice is that the programmer will still do the difficult parts of design and implementation, while the Apprentice will serve as an expert programmer and critic (as if looking over the programmer's shoulder). It can keep track of details and assist the programmer in documentation, verification,

debugging, and modification of programs. It actually augments the existing environment of text editors, interpreters and compilers instead of replacing it. The programmer has the choice to use the existing environment directly or to use the Apprentice.

The underlying design of the Programmer's Apprentice is a representation or plan which contains all the logical properties of an algorithm and serves as the basis for understanding the knowledge of programming. The plan is not dependent on any specific programming language although current usage has been limited to LISP. The plan can represent a wide variety of programming constructs from simple loops to complex hashing functions.

The Programmer's Apprentice is composed of five integrated modules. Figure 1 shows the architecture of the Programmer's Apprentice [Waters82]. One module is the analyzer which constructs a plan from a given program. It does not matter if the program was created or modified by an ordinary text editor or by the Programmer's Apprentice. This alone allows the Programmer's Apprentice to be implemented into an existing environment. The coder module, which does the opposite of the analyzer, generates program text corresponding to a plan. Pieces of the analyzer and coder modules are program language dependent and would have to be modified for different languages. The drawer module draws graphical representations of plans. The plan library module contains plan fragments which are standard or common data structures or algorithms. The plan editor module allows the

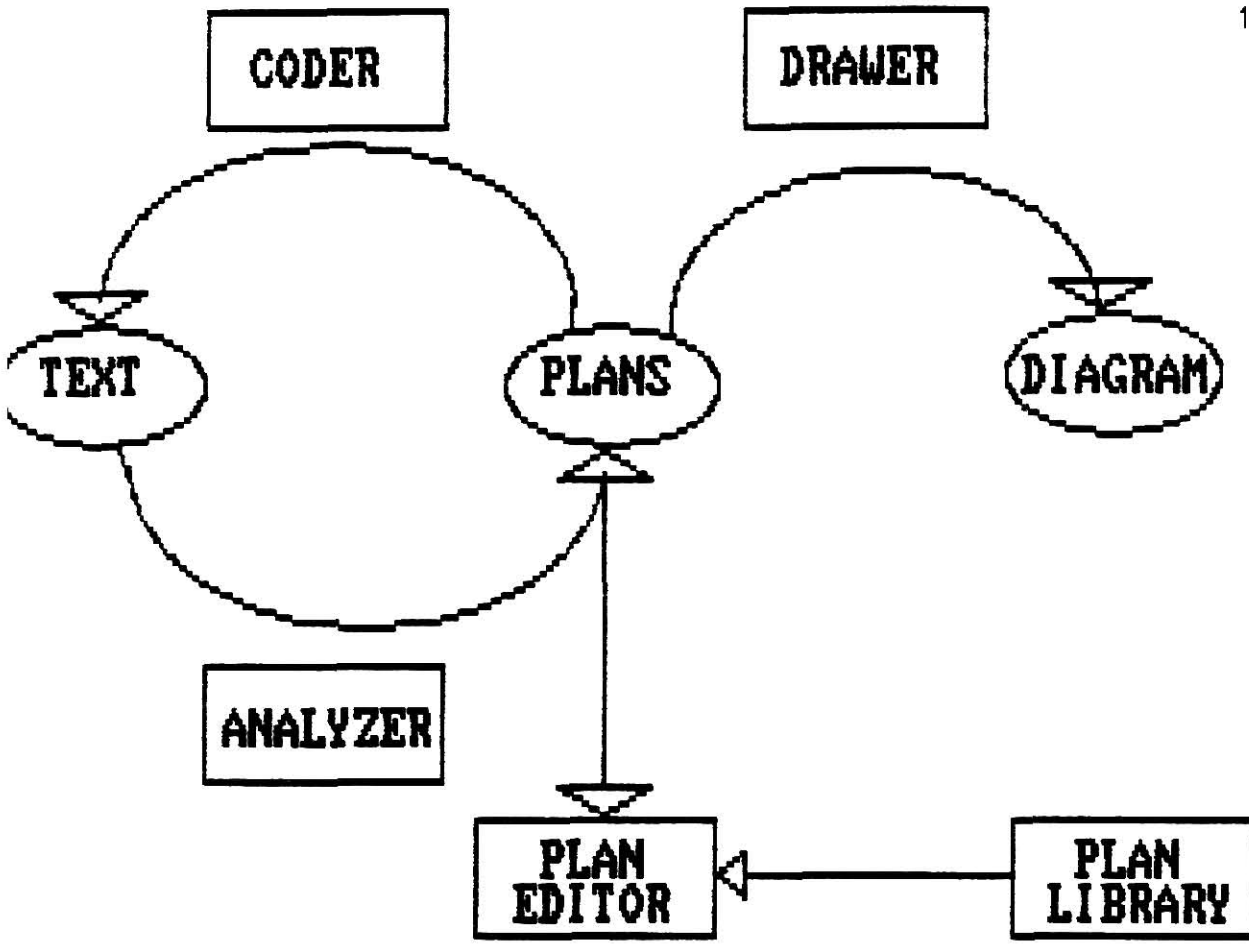

FIGURE 1. THE PROGRAMMER'S APPRENTICE ARCHITECTURE

programmer to modify the program by modifying the plan. Using the plan library to store common plans, a programmer can easily and accurately build or maintain programs. The Programmer's Apprentice is a very ambitious project which will take several years of refinement to produce a workable and efficient product.

3.1.2. Intelligent Program Editor (IPE)

The Intelligent Program Editor (IPE) is another "programmer's apprentice" which allows the programmer to

better understand what the program does (input and output information) and what it contains (data structures and flow control information). Daniel Shapiro, Jeffery Dean, and Brian McCune [Shapiro84] are currently designing and implementing IPE at Advanced Information and Decision Systems in California. Initially, it will only support the Ada programming language. The main idea in IPE is to explicitly represent textual, syntactic and semantic structures in programs. This is done with a knowledge base which contains the representations, and a search mechanism called the Program Reference Language (PRL), which locates pieces of programs based on descriptions from the programmer. Both of these comprise the Extended Program Module (EPM).

The EPM can be thought of as a smart database management system for creating and maintaining programs. The database of program structures contains seven different types or levels of representations of which programs are comprised. First, and most basic, is the textual representation which is concerned with words and delineators and is similar to most basic text editors. Second, is the syntactical representation which provides the vocabulary for programming constructs like "for" loops, parameters and procedures. Next is data and control flow representation which provides the vocabulary relating to the logical structure of the program. The next level of representation is the segmented parse level which defines a program in terms of its data and control flow such as identifying subfunctions of loops. The next level is more abstract and considers programs to be built with common

structures. These are called typical programming patterns (TPPs) and include such notions as list insertions, variable interchanges and hash table functions. These are similar in nature to the plans of the Programmer's Apprentice. The next, and most complex level, is the intentional aggregate representation which associates larger program fragments with key concepts that are supplied by the programmer. A collection of TPPs that are functionally similar may be defined as an intentional aggregate. The last level of representation is the all-important documentation. It spans all the other levels of representation and allows a programmer, for example, to explain why some data flow or loop construct was used.

The Program Reference Language (PRL) allows the programmer to specify a program region to be explained for further understanding by searching four of the database representations. The four levels are the textual, syntactic, segmented parse and the typical programming patterns. They are connected by a code region which associates program features with the physical regions of the program. See Figure 2. Depending on what the programmer needs to know about the program, the PRL will search the appropriate data base(s) and link the corresponding documentation to the findings.

The PRL uses two different search techniques. First, the programmer makes a "stab in the dark" at selecting the region needed. By applying more conditions and restrictions to the initial selection, PRL is able to locate the appropriate information. The second method uses a technique called code

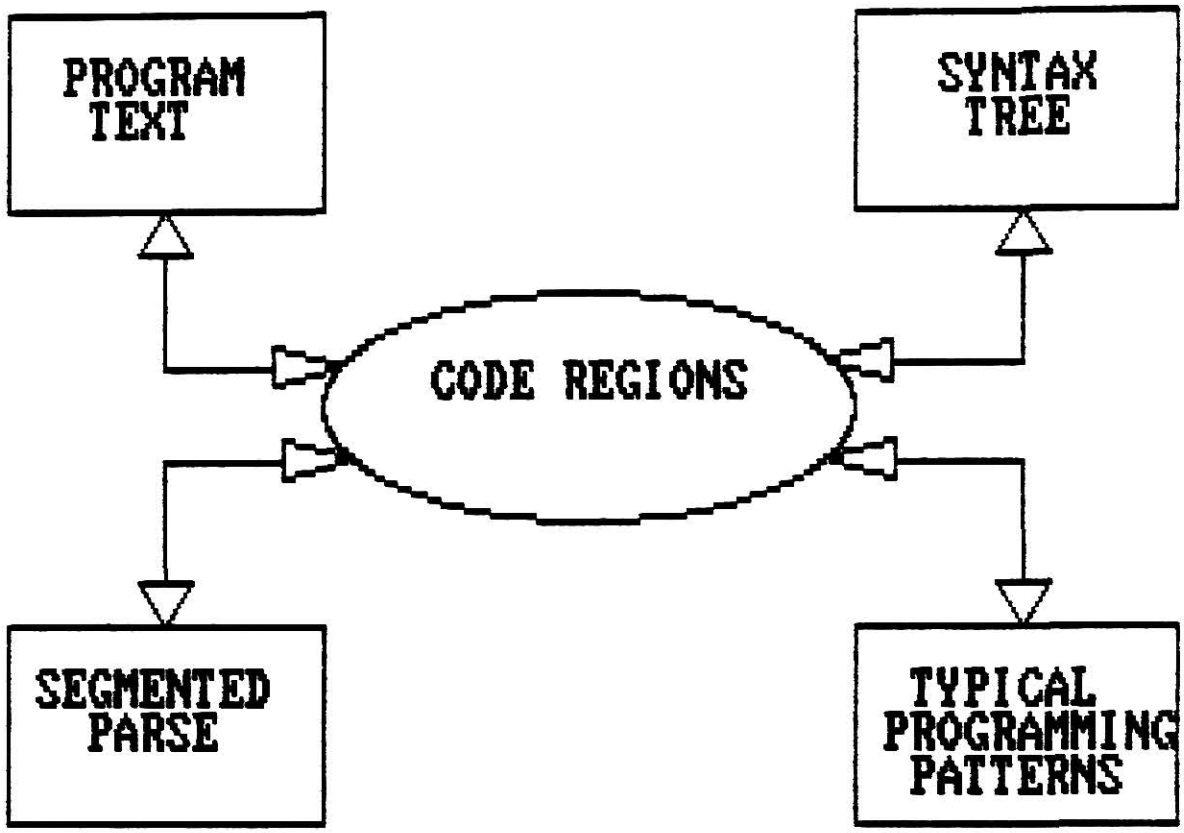

FIGURE 2. THE PRL IMPLEMENTATION FOR IPE

painting which intersects a collection of items together to satisfy the conditions imposed by the programmer. PRL overlays the regions of code with these items and colors (figuratively) some items red and the others yellow. If a region comes up orange, then it has all the properties that were requested (or it is at least close enough).

3.1.3. PROUST

PROUST is a knowledge-based program understanding system which analyzes and "understands" Pascal programs written by

novice programmers. Its developers are W. Lewis Johnson and Elliot Soloway from Yale University. PROUST uses an ordinary Pascal program as input and finds the most likely mapping between the program and the program's requirements and specification. It uses a knowledge base of programming plans (similar to the Programmer's Apprentice plans), some programming strategies and a pool of common program bugs. The PROUST developers define a bug to be "sections of code whose behavior fails to agree with the program specification". "In general, bugs are not properties of programs, but rather are properties of the relationship between programs and intentions." [Johnson84]

As mentioned above, the programming plans which PROUST uses are abstracted procedures or strategies that represent key elements to aid in finding the actual intention of the code. One problem in using plans is that novice programmers may not use the same plan as an expert so the designers have extended PROUST's knowledge base to include extra plans. This extension causes more problems because there are more plans to choose from and the programmer has to spend extra time to determine which plan to use. Also, a program is not necessarily free of bugs simply by adding more plans to the program. Organization of plans then becomes crucial. PROUST solves these problems by using goal decomposition of the programs. A goal decomposition consists of a description of subtasks represented in a hierarchical organization, indications of the interactions and relationships among the subtasks, and a mapping from the subtasks' requirements to the

plans that are used to implement them. The plans which a goal decomposition specifies are then matched against the program and this results in a mapping from the program requirements to the individual statements. The results from a study on the use of PROUST were fairly promising. More than 200 programs were chosen from students and 72% of these were completely analyzed by PROUST. Of those programs analyzed, PROUST was correct 95% of the time. PROUST is only able to understand small Pascal programs (less than two pages in length) at the current time [Johnson 84].

3.1.4. EMACS

EMACS is an extensible display editor which allows a programmer to add new functions or change existing editing functions. Even though, by itself, it is not truly an intelligent tool using AI techniques, it can be modified to include an extension which is an intelligent tool. The tools added could be knowledgeable enough to understand and analyze a program that is being edited. Richard Stallman [Stallman81] of MIT has developed EMACS into a very powerful and successful product. Several hundred companies and universities are using EMACS.

The extensions to EMACS are really functions written in the same language as EMACS. LISP is usually the language base that EMACS uses, although any interpretive programming language can be used as well because as EMACS is executing it can accept new functions and then execute them.

Unfortunately, PL/I, Pascal, C and other compiled languages are more difficult to implement with EMACS and have much less extensibility.

One extension EMACS uses is called TAGS which aids in editing large programs. It records each function or procedure in a program, states what file defines it and the position in the file. This allows the programmer to find the location of a piece of code very easily.

EMACS also contains the ability to self-document the extensions and functions it uses. This is very helpful to those programmers who need to know what the extension or function does.

Another useful function of EMACS is its ability to undo the a command that was entered. This is powerful since it allows a programmer to "test out" what the code or documentation will look like without saving and re-editing the file.

3.2. Software Maintenance Tools in Programming Environments

The software maintenance tools mentioned in this section are not strictly used for the maintenance of software but can also be used in the software development process as well.

3.2.1. UNIX

The UNIX operating system is quickly becoming a standard

throughout the country. Many corporations are converting to it and hundreds of universities have used it for teaching purposes for many years. It was created by Dennis Ritchie and Ken Thompson in the late 70's [Kernighan81].

Although UNIX is not a programming environment which has a lot of integrated, intelligent tools, it is an excellent environment to build such tools. The User Software Engineering Environment (USE), discussed below, uses UNIX as its underlying environment. The Shell environment in UNIX is an extensible, easy-to-use command interpreter which allows the programmer to do high-level functions, such as file manipulation, program compilation and execution of user defined macros. UNIX has some very useful tools with which programmers can easily modify programs and track programs and documentation.

As mentioned earlier, the Visual Editor (vi) is a very powerful full-screen editor which allows a programmer to easily modify a program. Vi will do automatic indention (user defined), search and replace items (locally or globally) and it allows easy access the Shell environment.

The "make" command is a software tool that maintains, updates, and regenerates groups of computer programs. It provides a method to store all the information needed to assemble small programs (usually C programs) into a large, more sophisticated one. A file called a makefile holds the file names of the small programs, the steps necessary to generate the large program, and specifies the dependencies among the files. When make executes the makefile, the date

and time the small programs were last modified are checked and the operations needed to update them are performed in sequence. Then, make continues to create the overall large program.

The Source Code Control System (SCCS) allows the complete tracking of a program's source code and its corresponding documentation. When modifications are made to a program, a new version number is automatically generated and the changes (or deltas) are attached to the old version. This makes any version accessible at any point in time. Documentation can accompany the change detailing what the change was, who made the change, when the change occurred and why it was needed.

The UNIX Consultant (UC) is an intelligent tool which augments the UNIX environment [Wilensky84]. UC is written in FRANZ LISP and PEARL, an AI language. It is a natural language help facility which helps beginning users to learn the UNIX commands and structures. It has a knowledge base of the English language and UNIX commands. New UNIX commands are easily added to the knowledge base by the UC Teacher.

3.2.2. USE

The User Software Engineering (USE) is really a methodology that subscribes to the rapid prototyping paradigm. This methodology is supported by the Unified Support Environment which contains five integrated tools. All five were developed in the UNIX environment by Anthony Wasserman [Wasserman83.

The first tool is called Troll which is a relational database management system using a relational algebra-like language. It is accessed by the other three tools and used for storage and retrieval of information. It does not use a central area of internal storage. Instead, it uses shared UNIX directories or even an individual's private directory as the database. The second tool is RAPID (RApid Prototyping of Interactive Dialogues) which helps the programmer to quickly construct a prototype. It uses state transition diagrams to link data structures and algorithms (pieces of programs very similar to the Programmer's Apprentice plans) to the programmer's dialogue. The third tool is PLAIN (Programming LAnguage for INteraction). It is derived from Pascal and allows the efficient use of string handling, exception handling, pattern matching and Troll database management. The fourth tool is called Focus. It is an easy-to-use, screen-oriented interface to the Troll database. The last tool is really an environment by itself. It is the Interactive Development Environment (IDE) which uses the Troll database to keep track of all development and maintenance activities and to control program versions. It also simplifies the task of compiling, linking and loading of programs. IDE keeps information about the status of the program development or modification so reports can easily be generated for other programmers and management.

3.2.3. Interlisp

Interlisp, like UNIX, is another popular programming environment, especially for those programmers who like and use LISP [Teitelman81]. It is integrated and extensible and contains powerful tools like Masterscope, DWIM, and the Programmer's Assistant.

Masterscope is a tool which interactively analyzes and performs a cross-reference on programs to help a programmer determine if a change will affect other programs. It reports which functions are called, where and how variables are referenced, set and bound, and record declarations. The programmer can then look at the information (held in a database) or have Masterscope search for the individual items needed. It is extensible and can be modified with LISP functions to manipulate the database information. Masterscope can be invoked by the program editor when a question about a change arises. It uses DWIM (described below) to help update automatically and explain any assumptions it made.

DWIM (Do What I Mean) is a very interesting tool for programmers. As soon as DWIM notices a misspelling of a command or LISP function, it then attempts to correct the misspelling. It first searches for a list of valid commands or recently used functions. If it finds a match, it then corrects the command or function and any future misspellings of the same type. The use of DWIM can be modified so that selected parameters could be left out of a program and DWIM would then search and find the appropriate parameter or value and insert it without ever telling the programmer.

The Programmer's Assistant in Interlisp actively records,

(in a history list) the programmer's input, any side effects of the input and the results of operations on the input. There are several commands which can manipulate the data in the history list. The first command, REDO, allows the programmer to repeat one or more operations. The FIX command invokes the Interlisp editor to make changes to the program and then re-executes the changes. The USE command allows a substitution of the input to be made before re-executing the program. The UNDO command, one of the most liked and needed commands in programming, allows the cancellation of effects on certain operations. The programmer can regain lost information or flip between the original program and the changed program.

Interlisp-D is an interactive, user-friendly interface between the programmer and Interlisp. Many windows can be opened up simultaneously on the CRT screen. This allows a programmer to be editting a program and applying Masterscope on another program at the same time.

3.2.4. POPLOG

POPLOG is an integrated programming environment that uses LISP, PROLOG, and POP-11 as its underlying language support. It was initially developed for AI research and teaching at the University of Sussex in Brighton, England by Steve Hardy [Hardy83]. Its use, now, is toward general applications like expert systems and rapid prototyping of software.

POPLOG is also an efficient programming environment. A

programmer working in POPLOG doesn't have to re-compile a whole program after changes are made, instead, the current state of the compile is saved as images so, just the modified piece of the program is re-compiled. Since the editor, compiler, loader, debugger and on-line documentation are integrated into one system, then modifications to programs can be made more efficiently.

The three languages that POPLOG fully supports (LISP, PROLOG and POP-11) allow a programmer to choose the proper language type in which to implement the solution to a problem. All three languages are compiled into an intermediate language and that language is then compiled into machine code. In addition, POPLOG supports the use of Pascal, Ada, C and Fortran routines to be linked in.

The on-line documentation facility simplifies the usage of POPLOG. It contains help files, teach files, reference files and manuals. These files can be accessed by explicitly typing help, teach or ref and then the command, or by highlighting the command in the editor and hitting the help key. A window, with the information requested, will be displayed on the screen.

3.2.5. KBSA

The Knowledge-Based Software Assistant (KBSA) is a rapid prototyping environment which generates code from the requirements specification [Green83]. The KBSA proposes that software developers incrementally refine decisions which

produces the formal specification. During the maintenance phase, the programmer would modify the specification and regenerate the program by "replaying" the development process. A new version number would then be connected to the change. This process would abandon the current notion that maintenance is just "patching" of the implementation and would force the maintenance phase to follow the same guidelines of the development phase.

At the center of the KBSA structure is the framework, consisting of an activities coordinator, which validates, records and coordinates all activities between the support system, project management and the development process, and a knowledge base manager, which contains a semantic model of the entire project.

The support system contains version and access controls, the inference engine (to extrapolate information from the knowledge base), integrated tools and user interfaces (using a workstation with high-resolution graphics).

The project management system coordinates the policies and procedures defined for the development process, aids in task assignments, allows easier communications between all those involved and provides the necessary documentation tools for the entire development process.

3.3. Miscellaneous Maintenance Tools

The following maintenance tools are what most programmers would like to see in a maintenance programming environment.

These are just a select set of tools that designers of programming environments are still researching [Dean82].

3.3.1. Programming Language Stylizer

A programming language stylizer can check programs to make sure they adhere to certain pre-defined standards and style guidelines. These guidelines will not be built-in, but can be created independently to provide uniqueness for different programming projects.

When the stylizer is executed, it will provide a list of violations. The intelligent program editor could then be invoked to help the programmer fix the violations by highlighting the affected areas or by actually making the changes to the program automatically. If the violations are not necessary for various reasons, then the programmer can be allowed to suppress any printing of violations after the first printing of the program.

3.3.2. Change Propagation Detector

During the maintenance of programs, a programmer makes a change and does not realize the change may affect other parts of the program. So, after testing is done and a bug was found (caused from the original change), then another change must be made. This can be a "vicious" cycle and a nightmare to a programmer.

A change propagation detector could help identify

possible side-effects of modifications. It could use the syntax and semantics of the programming language to find the potential bugs. It could easily be able to determine the effects on basic data structures, but could have difficulty with pointers and indirect referencing. The editor could be invoked to highlight the possible bug to let the programmer know it exists. The Masterscope system in the Interlisp programming environment is similar to this concept.

3.3.3. Programming Tutor

Programmers usually need to look up information about the programming language or some aspect of the programming environment. If this information were to be put on-line and allow the computer to "do the talking", then the programmer would not have to interrupt the current session to find the answer.

The programming tutor should have some intelligence built into it. It should know about the programming environment as well as knowing the specific level of expertise of the programmer. The tutor could actually have the programmer write a program, compile it and run it, and simultaneously the programmer is interactively asking questions. The UNIX Consultant is a good example of an intelligent program tutor.

4. Critical Evaluation and Further Study

This section critically evaluates the intelligent tools

described in the previous section. It explains the advantages and the limitations of each tool.

4.1. Intelligent Program Assistants

The Programmer's Apprentice, if implemented by design, will help revolutionize software programming and certainly aid in software maintenance. A major advantage that it has over the other intelligent program assistants, is its ability to be easily integrated into an existing environment. Due to its modularity, only two modules, the analyzer and the coder, would have to change when it is implemented into a different environment with different programming languages. Another advantage is that the programmer is not forced to use the Programmer's Apprentice, but will want to use it anyway because of its power to catch most programming errors. The drawer module allows a user-friendly mode of viewing the plans of data structures and algorithms.

One problem with the Programmer's Apprentice is that it is currently being designed exclusively for LISP and that implementation may be years away. Eventually, more languages need to be included to make it more of a viable product.

The Intelligent Program Editor (IPE) is also modular in design and allows the documentation to be retrieved at any level of representation. The code painting technique, used by the Program Reference Language (PRL) to find a region of code the programmer was looking for, is an interesting and unique method.

A major problem with IPE is that the seven different levels of representation would have to be modified if it were to be applied to other programming languages.

PROUST has some good aspects, but they are overshadowed by its limitations. PROUST can understand programs by mapping the program requirements to individual statements in the program, which directly verifies that the program is correct (assuming the requirements were correct). Another advantage is PROUST can be applied to existing programs without modification to the programs.

Unfortunately, PROUST can only handle very small Pascal programs. This drastically reduces its chances of becoming a good usable tool unless its developers were to allow larger programs in various languages. Also, if the program does not have any requirements or the requirements are outdated, then the programmer would have to create some requirements (possibly from the actual code itself).

EMACS is a good display editor mostly due to its extensibility. Intelligent tools have been written to expand its usefulness. The TAGS function allows the programmer to efficiently find pieces of code to be modified. Also, the self-documentation ability is a nice feature for those programmers who forget what the extension or function is called or what it does.

One drawback of EMACS is that LISP is the only language that is allowed easy extensibility. If a compiled language like Pascal or C is used as the base for EMACS, then the extensions have to be programmed in a dialect of LISP.

Incremental compilers and interpreters for previously compiled languages are being developed. These will then allow EMACS to be extensible for many other languages. Also, EMACS has a very large set of commands for editing. This can be a problem if the programmer forgets the commands and needs to look them up every time.

High-level languages like Pascal, C and COBOL were designed to hide the assembly code of programs so programmers did not have to know assembler. Yet, there were many occasions where we had to look at the assembled code to find out why a COBOL or PL/I program didn't work like it was intended to. Since some of us had very little experience with the assembly language, an expert assembly programmer had to help analyze the program. A limitation, or at least a concern, of using very high-level languages, like Programmer's Apprentice and the Intelligent Program Editor, will be the hiding of what the machine is actually doing. This causes the programmer to be more dependent on the developer and the supplier of the software. To prevent this from happening, the software must be developed so that there isn't a need to know what the machine is doing at a lower level (almost impossible to do), or the software must be extensible to allow for lower level analysis and modification for specific applications.

Future intelligent program assistants, which generate code strictly from the program specification (automatic programming) may not be very popular among programmers because most programmers enjoy designing software. These tools, on the other hand, will be well received by non-programmers or

programmers, who are involved with identifying solutions (requirements and specifications) to problems, because the program could be automatically generated from some high level description (specification).

4.2. Software Maintenance Tools in Programming Environments

The UNIX programming environment is a good environment to use when maintaining software. I have a biased opinion, of course, because UNIX was developed by Bell Laboratories and I work for AT&T. I mainly use UNIX for writing memoranda and sending mail to other users.

The primary advantage of UNIX is its availability and portability. It is becoming very popular and runs on small microcomputers to large mainframes. The make command is a very powerful tool to the maintenance programmer since it "remembers" the dependencies of many programs so the programmer doesn't have to. Another good tool is the Source Code Control System (SCCS). It does an excellent job of keeping track of the source code and documentation of large software projects in the maintenance environment. There have been many times when going back to an older version has helped in certain situations. The UNIX Consultant (UC) has not been as popular as UNIX itself, but it is also a much newer product. It is a good tool when teaching users how to use UNIX.

A drawback of UNIX is that most people find UNIX hard to use due to the staggering number of commands that are

available. This is true of the Visual Editor (vi). There are so many commands to remember that if you don't use it frequently, it becomes difficult to use. That's why the UNIX Consultant comes in handy. Yet, for those who are expert users of UNIX, UC is really not needed. Another difficulty is in the make command. The makefiles are very difficult to build initially for large software projects and require the programmer to decide whether one program is dependent on another program.

The User Software Engineering (USE) environment has some good integrated, modular tools. The primary maintenance tool is the Interactive Development Environment (IDE) which controls the program versions.

The tools in USE lack the intelligence needed to really be able to handle a large programming environment.

The Interlisp environment has some nice features. Masterscope can really help a programmer when trying to search for certain pieces of code. I have spent a lot of time looking through listings of programs looking for a particular section of code. DWIM (Do What I Mean) is needed in every programming environment since it actually tries (and succeeds) in second guessing the programmer. The Programmer's Assistant is especially a good tool because of the UNDO and the other associated commands. There have been many occasions where I wished I could undo certain changes (not starting the editing session over). The Interlisp-D interface allows easy and efficient access to the Interlisp environment.

The main limitation of Interlisp is it only supports the

LISP programming language and most businesses currently do not use LISP or even know what it is. LISP has always had a difficult time gaining a stronghold anywhere but universities and research centers. Maybe by creating such easy-to-use, intelligent tools and programming environments, like Interlisp, businesses will eventually take notice and use LISP or intelligent tools programmed in LISP.

POPLOG's primary advantage is its ability to use several languages in the environment. Allowing routines of other languages to be linked into the environment is very powerful. Also, the capability of stopping a compile, making a change to the source, and re-compiling only the affected piece of the program is very efficient (not only for the machine, but especially for the programmer). The on-line documentation feature, allowing usage at many levels, is also very powerful and allows a user to understand commands easily.

POPLOG is not very well known and has usually been used only in education and research disciplines, but it is now being marketed in the United States.

The Knowledge Based Software Assistant (KBSA) has a good theory behind it. That is, maintenance programming is just "patching" of code and by developing and maintaining a proper specification for a program, then the program can be generated automatically with the correct changes. Also, the maintenance environment closely parallels the development environment.

The KBSA presents a few problems. First, it will only be focusing on extremely large software projects of greater than one million lines of code. Which means it may not be feasible

to use it for smaller projects. Second, it will take up to 15 years to complete the project, assuming the U.S. government will grant them the money needed for the project. Finally, if the software system is very large and the software is changed frequently (i.e. "replayed"), then it could extremely tax the computer hardware (unless it is very large or it is a parallel processing machine).

The maintenance tools in programming environments will need to be even more integrated and more intelligent to be used for the entire software life cycle.

4.3. Miscellaneous Maintenance Tools

All three miscellaneous maintenance tools presented have not yet been developed to their full potential and have limited capabilities (like being dependent on a particular language). They could be excellent tools, though, and should provide the programmer with an easier mechanism to maintain programs.

5. A Practical Application of an Intelligent Tool

The following section describes how just one of the intelligent tools, Masterscope in Interlisp, could be applied (with some modifications) to a "real world" maintenance environment.

5.1. A Maintenance Environment

For the past six years I have been a computer programmer for AT&T in Denver, Colorado. I maintained several large COBOL and PL/I programs which were part of a large order processing software project. I also coordinated the changes and production "cut-ins" (i.e. linking programs into the production environment) for over 500 programs each averaging 1,000 lines of code. At its heyday, it required approximately 30 people to maintain the software. Currently, the product it supports is being phased out and therefore, programmers will not be needed to support it except for occasional problems.

Even though this software project is dying there are many other software projects in AT&T and other companies which could benefit from the intelligent tools described in this paper. This software project serves as an example to show how the applicability of an Artificial Intelligence technique, a change propagation detector, can help in saving a maintenance programmer's time and effort and therefore, the overall cost of maintenance.

This large software system supported an order processing environment for Private Branch Exchange (PBX) telephone systems. These systems were sold to retail businesses, hotels and motels, universities and governments - both domestic and foreign. The PBX systems were configured with various feature packages and memory sizes to fit an individual customer's needs. The foreign market required different features, functions and hardware than the domestic market. Over the years, many enhancements and changes were made to the product

to satisfy the changing customers' requirements. These product changes required changes in the order processing software system which I helped to maintain.

The order processing system served several purposes. First, it allowed orders to be entered from other AT&T locations throughout the country. An order resulted from a questionnaire filled out by a prospective customer. Next, the order was checked for erroneous or mismatched data. If the order had some errors or problems, then changes were made and the order was reprocessed (re-submitted to the computer). This iteration could happen many times because an order was seldom free of errors when it was initially entered. After the order was "clean" of errors, it would continue being processed through the software system and a Customer Order Document (COD) would be generated, which documented all of the customized hardware and software features for an individual customer. Finally, a magnetic cartridge tape containing all of the customer's specific information (or customer translations) was generated. This cartridge tape, which was used to load the PBX's internal memory, and the COD were then sent to the customer's site along with the PBX hardware (cabinets, circuit packs, cabling and telephones). Another aspect of the order processing system was to allow existing PBX's at customer sites to be updated with new features and enhancements. This required the customer to mail in the magnetic cartridge tape. The information (customer translations) was then "pulled off" the tape, merged with new information from an order and then checked for errors. A new

COD and cartridge tape were then mailed back to the customer for re-installation.

The order processing software system contained four major subsystems. The first was the order entry sybsystem written entirely in COBOL. This subsystem used the raw questionnaire data as input. This input information was heavily checked for errors. There were usually several "data checks" for every question in the questionnaire. This subsystem was by far the largest in size of all subsystems and required much more effort and knowledge to maintain it. The main output from the order entry subsystem consisted of intermediate customer translation files and a data file which contained the internal linkage sections from the COBOL programs. These files served as input into subsequent subsystems. The second subsystem generated the COD and was written in COBOL. It used the order entry subsystem's output files and the internal linkage data file as input. It used COBOL's report generation facility to produce a document which reflected the customer's order. The third subsystem, also written in COBOL, created the magnetic cartridge tape containing all of the customer's translation information. The input to this subsystem was the order entry subsystem's output files. The internal linkage data file was not used as input. The fourth subsystem was written in PL/I and allowed an existing customer to add or change features. It was called the blowback subsystem since the old translation information was "blown back" onto a new tape. This blowback subsystem performed functions that were opposite to the cartridge tape generation subsystem. Its input was the old

customer's tape and its output was the intermediate files similar to those produced by the order entry subsystem. These files were actually input to the order entry subsystem which also had the questionnaire data as input.

Problems consistently arose when changes were made to a subsystem without verifying whether or not the changes affected the same subsystem or other subsystems. For example, the information on the cartridge tape was supposed to match the printed information in the COD and what the customer originally ordered on the questionnaire. Frequently, there were mismatches due to one programmer making a change to a subsystem and not telling another programmer about it, assuming it did not affect anybody else's programs. The problem was found and the programs were then modified to bring the cartridge tape and COD into agreement with the questionnaire. This modification process was sometimes extensive as more changes were made and more mismatches were encountered. Another area of inconsistency was between the magnetic cartridge tape subsystem and the blowback subsystem. Since one was written in COBOL and the other in PL/I, a language barrier was created. The cartridge tape subsystem was usually changed first so that new PBX machines would have the newest features. The changes to the blowback subsystem, meanwhile, would usually lag behind.

There were four main reasons why these changes were not caught before being cut into production. First, pressure from the users group and their management forced many quick changes to the system which meant very little time was spent to verify

and test the changes. Changes were made by submitting a batch compile of the COBOL or PL/I program, waiting for the output (sometimes several hours or even a whole day), submitting a batch test order, which may not have been up-to-date, waiting a whole day for the test to execute and then verify its output, and finally, submitting a batch job to link the program into production. If the program had syntax errors or the test failed, then the process was even slower and the users became more impatient. Second, the programmers, including myself, initially had no idea what side-effects could occur from making a "harmless" change. Since the system had very little documentation to support it, we had to stumble along by making a change, testing it (as best as we could) and then linking it into production. At times, the production environment seemed to be our only test vehicle and the users would get upset and frustrated. Third, the cartridge tape subsystem did not use the internal linkage file that the COD subsystem used. This caused an enormous amount of inconsistencies between the cartridge tape and the COD. Changes were made that affected the internal linkage and therefore, the COD was changed but the tape would not be changed. Fourth, the inconsistencies between the blowback subsystem and the cartridge tape subsystem were mainly caused by one person updating the cartridge tape subsystem using COBOL and another person updating the blowback subsystem using PL/I. This was complicated by the same change being made to the different subsystems at different times (often months apart).

Eventually, changes to the programs became easier to control since experience, through trial and error (mostly error), grew into a human "knowledge base". A few key programmers would remember what happened when a change was made to a particular program and the effect it had on other programs in the same subsystem or other subsystems. Other programmers were asked by the key programmers to make changes to a program because another programmer was making a change that would affect that program. Problems still arose since the key programmer would forget about a program that could be affected, not be asked by another programmer if the change affects other programs, or be transferred to another project.

An intelligent tool with the concept of a change propagation detector would have made the whole project (and the users) much easier to work with.

5.2. Intelligent Tool Application

One existing intelligent tool discussed previously, seems suited to the problems in the maintenance environment described above. The tool is Masterscope which is found in the Interlisp programming environment. As described earlier, its purpose is to analyze programs and to build a program cross-reference database. The underlying concept of Masterscope is to provide the user with all the programs that would be affected if a particular change was made. This concept is very similar to a change propagation detection tool.

When applying Masterscope to the maintenance environment, one glaring limitation is recognized. Masterscope is only used in the Interlisp environment and is restricted to the LISP programming language. This, by itself, drastically reduces its usefulness as a general tool. In fact, most of the intelligent tools being designed and implemented today will only help a very small percentage of the existing software maintenance projects. This is mainly because the majority of research on intelligent tools is being done by researchers who work with LISP and Prolog and recognize the potential those languages have in Artificial Intelligence. It is also due to the fact that the researchers like shortcuts too, so they use their tools just for themselves initially.

Another problem with Masterscope, when applied to the maintenance environment, is its lack of support for application specific knowledge. Actually, most intelligent tools have not yet reached this goal of making the tool smart enough to discern what is right for a particular application. An enormous amount of knowledge would have to be stored for this purpose. An example would be: If a change is made to the order entry subsystem, could it affect the COD and the tape? If it affects the COD, it may not necessarily affect the tape (i.e. changes in descriptions of features but no customer translation changes). If it affects the tape, does it have any affect on the blowback subsystem? All these factors and decisions would have to be made while the program is being edited. Right now it doesn't seem feasible to expect such an application specific tool to be built with the current

hardware and software. Therefore, research in this area is still continuing.

For the purpose of this paper, assume that Masterscope was designed to be used in a COBOL or PL/I environment and that it had the capability to understand and analyze application specific information. This would make it a very powerful and extremely valuable tool in today's software maintenance environment. It could determine whether or not a change would affect other programs and other subsystems and also provide reasons why programs should or should not be updated due to any peculiarities.

Designing a new Masterscope for COBOL or PL/I should seemingly be easier because all of the variables and data structures are declared in a special section at the beginning of each program. This would allow Masterscope to find all the occurrences of the same variable or data structure throughout the programs. If the variable names are different between programs (which is usually the case), but are used in the same context, then Masterscope could note this fact and recall it later when a change affects the variable. In the maintenance environment described above, the modified Masterscope would have to know both COBOL and PL/I so that it could bridge the gap between the blowback subsystem and the other subsystems. This could create interesting design problems, but it should be attainable.

COBOL has a linkage declaration section for any global variables that are passed between programs and subsystems. This linkage section would have to be checked for changes

since its structure must be identical throughout all of the programs that use it. This linkage section usually contains a lot of key information about the application itself and could be used to Masterscope's benefit to help determine the proper changes to the right programs. In the maintenance environment, the linkage section contained the customized hardware configuration and any feature the customer ordered.

COBOL is a verbose language (wordy, but with limited commands) which can handle variable and procedure names up to 32 characters long. It is supposed to be a self-documenting language, but it only seems that way when the programmer is thinking up new variable names. These variable names may be interpreted differently by other programmers later. We always had problems finding what would be affected by a certain change because of very little existing documentation. If a database of connected documentation did exist (like some fortunate maintenance projects have), then Masterscope could be modified to also detect and possibly update any changes to the documentation. One drawback with this is that the documentation database would need to be extremely large for a large project like the one I described. The cross-reference database would be large also. Luckily, computer hardware is continually allowing more and more information to be stored so this may not be as much of a problem in the future.

Almost all of the problems in the above maintenance environment would be solved by applying a modified Masterscope as an intelligent tool. Changes to programs would no longer require massive guess-work on the programmer's part.

Masterscope would recognize changes in a subsystem, determine if changes are required in the other subsystems and alert the appropriate programmers. Inconsistencies between subsystems would disappear since Masterscope would recognize what programs used certain variables and data structures.

Masterscope would be an interactive system which would allow the programmers to immediately find where all of the changes need to be applied and could provide reasons why. By using an integrated, intelligent editor, the programmer could quickly and efficiently make the necessary changes, update the documentation, and begin testing. There would be fewer (if any) iterations of program editing and therefore the modified program could be placed into production sooner and with a higher level of confidence that it would execute properly.

6. Conclusion

With the cost of software maintenance increasing rapidly, new techniques and tools will be needed to help software maintenance programmers modify code efficiently and effectively. These tools must be integrated, intelligent and extensible.

Artificial Intelligent techniques, when applied to the software maintenance problem, address most of these issues. The intelligent program assistants and programming environments described in this paper have made a gallant effort to reduce the cost of software maintenance. Yet, most are still on the drawing board and won't be available for many

years, or are being used now, but have limited capabilities (e.g. supports only one programming language).

Over the course of the next decade, I envision intelligent software tools being developed and used just as basic editors and compilers are developed and used today. The software programmer could evolve into someone who knows how to solve problems (at a much higher level of thinking), instead of someone who only knows how to code and maintain a program.

BIBLIOGRAPHY

[Barr81] Barr, Avron; Feigenbaum, Edward A. The Handbook of Artificial Intelligence, Vols. 1-2, Kaufman, Los Altos, California, 1981.

[Connell84] Connell, John; Brice Linda "Rapid Prototyping" Datamation, August 1984, pp. 93-100.

[Dean82] Dean, Jeffery S.; McCune, Brian P. Advanced Tools for Software Maintenance, Advanced Information & Decision Systems, Mountain View, California, December 1982.

[Dyer84] Dyer, Charles A. "Expert Systems in Software Maintainability", 1984 Proceedings Annual Reliability and Maintainability Symposium, 1984, pp. 295-299.

[Frenkel85] Frenkel, Karen A. "Toward Automating the Software Development Cycle", Communications of the ACM, Vol. 28, No. 6, June 1985, pp. 578-589.

[Gallagher85] Gallagher, Jim The Application of Artificial Intelligence Techniques to Requirements Engineering, Kansas State University, 1985.

[Goering84] Goering, Richard "Do-it-yourself Development Tools Speed AI Applications" Computer Design, December 1984, pp. 29-39.

[Green83] Green, Cordell; Luckham, David; Blaxer, Robert; Cheatham, Thomas; Rich, Charles Report on a Knowledge Based Software Assistant, Kestrel Institute, Palo Alto, California, 1983.

[Hardy83] Hardy, Steve; Sloman, Aaron "POPLOG: A Multi-Purpose Program Development Environment" Microprocessor Software Quarterly, November 1983, pp. 1-34.

[Hayes84] Hayes-Roth, Frederick "The Knowledge-Based Expert System: A Tutorial", Computer, Vol. 17, No. 9, September 1984, pp. 11-28.

[Johnson84] Johnson, W. Lewis; Soloway, Elliot "PROUST: Knowledge-Based Program Understanding", Proceedings of the Seventh International Conference on Software Engineering, March 1984, pp. 369-380.

[Kernighan81] Kernighan, Brian W.; Mashey, John R. "The UNIX Programming Environment", Computer, Vol. 14, No. 4, April 1981, pp. 12-24.

[Kobler82] Kobler, Dr. Virginia; McDaniel, Ms. Bonnie G. "Expert Systems: Status and Perspectives" Compsac 1982, pp. 565-570.

[McCalla83] McCalla, Gordon; Cercone, Nick "Approaches to Knowledge Representation", Computer, Vol. 16, No. 10, October 1983, pp. 12-18.

[McCune83] McCune, Brian P.; Dean, Jeffery S. "Trends for Advanced Software Tools", IEEE Electronics and Aerospace Conference, 1983, pp. 291-298.

[Pressman82] Pressman, Roger Software Engineering: A Practitioner's Approach, McGraw Hill, 1982.

[Rich78] Rich, Charles; Shrobe, Howard E. "Initial Report on a Lisp Programmer's Apprentice" IEEE Transactions on Software Engineering, November 1978, pp. 456-467.

[Shapiro84] Shapiro, Daniel G.; Dean, Jeffrey S.; McCune, Brian P. "A Knowledge Base for Supporting An Intelligent Program Editor", Proceedings of the Seventh International Conference on Software Engineering, March 1984, pp. 381-386.

[Stallman81] Stallman, Richard M. "EMACS: The Extensible, Customizable, Self-Documenting Display Editor", Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, June 1981, pp. 147-156.

[Teitelman81] Teitelman, Warren; Masinter, Larry "The Interlisp Programming Environment", Computer, Vol. 14, No. 4, April 1981, pp. 25-34.

[Wasserman83] Wasserman, Anthony I. "The Unified Support Environment: Tool Support for the User Software Engineering Methodology", IEEE Softfair - Software Development: Tools, Techniques and Alternatives, 1983, pp. 145-153.

[Waters82] Waters, Richard C. "The Programmer's Apprentice: Knowledge Based Program Editing" IEEE Transactions on Software Engineering, January 1982, pp. 1-12.

[Wilensky84] Wilensky, Robert; Arens, Yigel; Chin, David "Talking to UNIX in English: An Overview of UC", Communications of the ACM, Vol. 27, No. 6, June 1984, pp.574-593.

THE APPLICATION OF ARTIFICIAL INTELLIGENCE TECHNIQUES TO SOFTWARE MAINTENANCE

by

WAYNE LOUIS WERBELOW

B.S., University of Wyoming, 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1985

ABSTRACT

This paper examines current research which applies Artificial Intelligence (AI) techniques to the software maintenance task. Definitions of what is meant by software maintenance and AI techniques (used by intelligent tools) are given. Current effort is divided into producing intelligent program assistants, software maintenance tools in large programming environments and miscellaneous maintenance tools. A critical evaluation of these efforts are discussed and the advantages and limitations are examined. A practical application of an intelligent tool is given to support the need of applying AI techniques to software maintenance.