N96-12951

MacDoctor: The Macintosh Diagnoser

David B. Lavery
William D. Brooks

Abstract:

When the Macintosh computer was first released, the primary user was a computer hobbyist who typically had a significant technical background and was highly motivated to understand the internal structure and operational intricacies of the computer. In recent years the Macintosh computer has become a widely-accepted general purpose computer which is being used by an ever-increasing non-technical audience. This has lead to a large base of users which have neither the interest nor the background to understand what is happening "behind the scenes" when the Macintosh is put to use - or what should be happening when something goes wrong.

Additionally, the Macintosh itself has evolved from a simple closed design to a complete family of processor platforms and peripherals with a tremendous number of possible configurations. With the increasing popularity of the Macintosh series, software and hardware developers are producing a product for every user's need. As the complexity of configuration possibilities grows, the need for experienced or even expert knowledge is required to diagnose problems. This presents a problem to uneducated or casual users. This problem indicates a new Macintosh consumer need; that is, a diagnostic tool able to determine the problem for the user. As the volume of Macintosh products has increased, this need has also increased.

The NASA Headquarters Office of Aeronautics and Space Technology (OAST) has become intimately aware of these problems and needs as they installed a Macintosh II computer on the desk of every employee (approximately 180 machines). Early in the installation process, the user support staff received calls to assist with a large number of problems common to multiple users. A desire was expressed for some type of aid to help a user recognize and diagnose the most common of the problems, allowing the user support staff to concentrate their talents on the more uncommon (and typically more difficult) problems. Additionally, such an aid could be used as a training assistant for new or novice user support personnel.

With this idea in mind, the authors began a project to identify and implement the knowledge base required to recognize, diagnose, and provide suggested solutions for, the most common problems associated with typical Macintosh use. This paper will present the process used to develop this implementation, from the initial analysis of user support call logs to identify the problem domain, through the use of CLIPS as the inference engine kernel, to the completion and testing of the system prototype.

MacDoctor: The Macintosh Diagnoser

Executive Summary

MacDoctor is the product of a graduate school project to develop a forward chaining, rule-based diagnostic tool to determine the cause, and thus the remedy, if any, of a Macintosh hardware configuration problem. The problem is identified through the traversal of a discrimination network represented in CLIPS rules. Remedies are directly, if not uniquely, addressed by a given problem determination. Future areas of research include automatic network exploration and mapping, predictive diagnosis, domain expansion and user maintenance.

Introduction

When the Macintosh computer was first released, the primary user was a computer hobbyist who typically had a significant technical background and was highly motivated to understand the internal structure and operational intricacies of the computer. In recent years the Macintosh computer has become a widely-accepted general purpose computer which is being used by an ever-increasing non-technical audience. This has lead to a large base of users which have neither the interest nor the background to understand what is happening "behind the scenes" when the Macintosh is put to use - or what should be happening when something goes wrong.

Additionally, the Macintosh itself has evolved from a simple closed design to a complete family of processor platforms and peripherals with a tremendous number of possible configurations. With the increasing popularity of the Macintosh series, software and hardware developers are producing a product for every user's need. As the complexity of configuration possibilities grows, the need for experienced or even expert knowledge is required to diagnose problems. This presents a problem to uneducated or casual users. This problem indicates a new Macintosh consumer need; that is, a diagnostic tool able to determine the problem for the user. As the volume of Macintosh products has increased, this need has also increased.

The NASA Headquarters Office of Aeronautics, Exploration and Technology (OAET) has become intimately aware of these problems and needs as they installed a Macintosh II computer on the desk of every employee (approximately 180 machines). Early in the installation process, the user support staff received calls to assist with a large number of problems common to multiple users. A desire was expressed for some type of aid to help a user recognize and diagnose the most common of the problems, allowing the user support staff to concentrate their talents on the more uncommon (and typically more difficult) problems. Additionally, such an aid could be used as a training assistant for new or novice user support personnel.

With this idea in mind, the authors have initiated a graduate research project to

identify and implement the knowledge base required to recognize, diagnose, and provide suggested solutions for, the most common problems associated with typical Macintosh use. This paper will present the process used to develop this implementation, from the initial analysis of user support call logs to identify the problem domain, through the use of CLIPS as the inference engine kernel, to the completion and testing of the system prototype.

Problem Statement

The objective of this project is to produce an easy-to-use, plain talking diagnostic tool which will be capable of analyzing a user's description of a problem, recognizing the problem condition and suggesting a solution activity. It is noted that Apple and other vendors manufacture products with built-in test and evaluation (BITE) capabilities. However, these are typically designed for board or component-level investigation. The authors intend to address a higher level implementation - a configuration diagnostic rather than a component diagnostic.

The problem is also more complicated than the component BITE testing. Single components are largely fixed in design. Test procedures for such components can be predetermined. At a configuration level, test procedure designs have added complexity in that computer configurations vary greatly depending on the system options and peripherals that the user has chosen for the system.

If an automated tool were made available to help users track down their configuration problems, at least two categories of users of the tool can be identified. The first is the new, non-computer-literate users who will use the tool to identify and correct problem conditions on their local Macintosh systems, and through the use of the tool gain greater degree of computer literacy. The second class of user includes personnel assigned to assist in the diagnosis and correction of problems for a large configuration of Macintosh systems ("help desk" or "user consultant" staffers), who need to quickly become effective and productive in the remote diagnosis of system problems, who would use the tool as both a rapid training aid and a productivity enhancement utility.

Implementation Approach

Early in the definition process for **MacDoctor**, it was realized that a forward chaining diagnosis system would present certain implementation capabilities which would be valuable to the development of the application. Inherent in the design of such systems in the ability to collect an initial set of error conditions from the user, and synthesize a set of possible solutions. As additional information is gathered, invalid solutions are removed, until a final solution set remains. This set can be indexed with confidence factors to indicate the expected precision of the proposed solution. These systems are flexible, both in terms of implementation and operation - as the knowledge base is developed there are few restrictions on the ordering of the knowledge rules, and as the expert system is

used, multiple logic paths may be followed by the user to reach the same solution. The logic structure used in the design of the questions to the user can resemble an inverted tree, and yet the user can provide incomplete or inferred information which allows them to move between then logical branches of the tree and traverse the tree without being constrained by the formalism of the tree structure.

The forward chaining expert system was selected as the best solution for developing the Macintosh diagnoser. Based on that decision, the following implementation decisions were made:

- The CLIPS expert system shell was used to create and develop the knowledge base and antecedent-consequent rule definitions. CLIPS is an extensible expert system shell developed by the NASA Johnson Space Center (JSC), with executable versions for Cray, Cyber, CDC, IBM, PC, VAX and Apollo computers, as well as the target Macintosh platform.

- Problem domain information was obtained from the NASA Headquarters User Support Center (USC) service call logs. The USC provides assistance to approximately 180 Macintosh users at NASA Headquarters, by aiding with problem diagnosis, system repair, training, and general user support. During the past two years of operation, the USC has compiled extensive documentation by logging problem calls and documenting the eventual solutions provided to users. The USC made this documentation available, and a set of typical user problems and questions which have been used has been derived as the initial Macintosh diagnoser problem domain.

- The expert knowledge for solution of the problems comes from two sources. First, the system implementers have over two years of experience with diagnosing Macintosh system and configuration problems, gained through a combination of professional experience and participation with Macintosh users groups (which involves training of new users). This learned knowledge is used extensively to develop the knowledge base. Second, for areas where the developers knowledge may be insufficient, the Systems Engineering Group at the Apple Federal Government Operations office in Reston, Virginia, was contacted and agreed to provide documentation and support similar to that normally supplied to the Apple field engineers.

- Development of system components external to the expert system shell (user interface, internal system status queries, etc.) were developed in the C programming language. The CLIPS expert system shell was developed in C, and readily incorporates external C routines.

Problem Domain Definition

The **MacDoctor** domain of expertise was selected based on the availability of raw data and the familiarity of the developers. The domain selected was the interoffice computer network installed in OAET, which consists of over 180

Macintosh II desktop computers connected via Ethernet. NASA has established a computing facilities support staff (help desk) which is responsible for the handling of hardware and software problems encountered by NASA personnel. Typically the users are not extensively trained in computer technology and thus constitute a population of novice users.

To define the problem domain to be addressed by the **MacDoctor** application, copies of the User Support Center calls logs were obtained, and review of the logs was initiated. 1372 call log entries were reviewed, and the following problem breakdown was derived:

Printing problems - networked LaserWriters	192
Printing problems - direct connect LaserWriters	0
Printing problems - networked ImageWriters	3
Printing problems - direct connect ImageWriters	7
Disk problems - SCSI devices	44
Disk problems - Diskette drives	21
Net/comm problems - mail services	66
Net/comm problems - file servers	18
Net/comm problems - modem services	171
System problems	60
Application problems	57
Finder problems	35
I/O problems	61
Total:	735

Note that the problem breakdown displayed above is a summary of the domain definition that we have created. The granularity of detail worked with is considerably greater. For example, the "printing problems- networked LaserWriters" line item above actually contains 28 distinct elements, each of which represents a unique problem state to be recognized by **MacDoctor**. In total, 180 distinct problems which occur within the domain were identified.

637 calls from the log entries were rejected, as they were determined to be outside the domain of the defined problem. These include items such as: requests for software, requests for specific training, problems pertaining to non-Macintosh systems, etc.

Determining the problem space was the first step. The more significant task was to build the discrimination network which would select the correct problem identification from the problem space. Again the help desks supplied much of the information. Each entry in the help desk log included the staff member's name, the problem as reported to the help desk, the procedure undertaken to identify the problem, the problem as determined by the staff member, and the steps taken to remedy the problem. Examination of the collection of the help desk log entries for each distinct problem showed a similar pattern of diagnosis and remedy. For each problem, the diagnosis and remedy were reviewed by domain experts to insure their validity. This process resulted in classes of problems with each problem represented by a description of the problem, a

unique set of symptoms which the problem will exhibit, and the remedy to the problem. By matching the symptom set, the problem can be identified and the remedy proscribed.

The symptom sets for the various problems were found to intersect to a high degree. A particular symptom could often be exhibited by several different problems. The problems were thus combined into a discriminate network or tree. The root node of the structure represents the most discriminating symptom, that symptom which reduces the problem space the most. For any node to be higher in the tree, this property must be maintained. If this is maintained, traversal of the tree will rapidly converge on the correct diagnosis.

Scope of Solution

It would be impractical to attempt to implement **MacDoctor** with the ability to recognize every problem identified in the problem domain. Instead, it was the developer's intent to sort the problems identified in the domain by frequency of occurrence and then provide an implementation which will address the top 80% of this list. The remaining 20% of the problem space includes items which tend to be either specific to a unique system configuration, or problems which occur with very low frequency.

Field testing of the **MacDoctor** application was arranged with the NASA User Support Center (source of original domain information) once the application knowledge base was established and implemented. The User Support Center agreed to utilize the system as a training aid for new members of the USC staff to increase productivity while the staff members are becoming familiar with the Macintosh installation, and to distribute the application to selected end users for evaluation and knowledge base validation. This field testing is still underway, and feedback from the testing is being used to implement a second iteration of **MacDoctor**.

Application Design

The design of **MacDoctor** separates the overall system into the following parts: user interface, inference engine, expert knowledge representation, and maintenance front-end.

As each of the segments was implemented, the developers were confronted with the issue of how the contents of knowledge base would be divided between the interface driver and the inference engine. These are the options considered:

- Have all the possible queries which may be asked of the user predefined in the interface portion of the application, installed in dedicated dialog boxes. The results of each query are interpreted by the interface portion of the application and either passed to the inference engine for incorporation within rules and further processing, or the interface

portion may act directly upon the results and process additional queries. The advantage of this approach is that the number of communications between the portions of the applications are minimized, and all the queries are precompiled, which will result in minimal execution times. The disadvantage is that any future extensions of the application will require considerable source-level reprogramming and recompiling of the application, and overall modularity of the application is minimized. Additionally, any change in the logic used in the knowledge base will require modification of both the interface and the inference portions of the application.

- Have all the possible queries which may be asked of the user predefined in the interface portions of the application, installed in dedicated dialog boxes. The results of each query are passed back to the inference engine for incorporation within rules and further processing. The advantage of this is faster processing of queries by minimizing the communication required for the inference portion to request a query, resulting in improved execution times. The disadvantage is that future extensions to the application will require source-level reprogramming and recompiling of the application.

- Have all the queries defined within the inference portion of the application, and queries are passed forward to the interface portion as they are needed. The interface portion is basically a small set of dialog box "shells", which accept and display the query strings from the inference portion, and return the query results. The advantage of this is that full modularity of the application is maintained, and that extensions to the knowledge base and modifications of the rule logic will not require recompilation of the application (it should be noted that input to the inference portion of the application will be done via a single text file containing the rule definitions for the knowledge base; therefore, modifications to the rule base will require only the use of a text editor, and not a compiler or development environment). This will significantly ease maintainability of the application. The disadvantage is that query requests from the inference potion of the application to the interface portion will require more communication between the portions, resulting in slightly decreased application performance.

The interface implementation method selected was to develop a general-case query interface driver which will allow the inference engine to pose query text to the inference driver for display. This will allow all of the logic, rule definitions, query text, and suggested solutions to be located in one modular file (permitting easier maintenance and extension), and allow the user interface to automatically handle extensions to the knowledge base without requiring recompilation of the application. This is done at a slight cost of system performance, but the impact to the user is negligible.

Knowledge Representation

Experience so far indicates that through the use of CLIPS we are able to adequately represent the knowledge base required to address the known problems, and a small subset of the knowledge base has been implemented to verify this. Initial efforts concentrated on the implementation of the rules required to recognize and suggest solutions to file server access problems. This problem class was selected as it included most of the major elements common to the problem space (i.e. network connectivity, supplied power, access control, network definition, device selection, etc.). The definitions required to represent the knowledge for this section of the application was stated with 39 rules in about 420 lines of code. As yet undetermined is the best way to encapsulate the knowledge data separately from the knowledge base framework, to allow extension of the rule set without full knowledge of the CLIPS syntax and structure (to allow maintenance of the knowledge base).

By combining the query format with properly structured rules in the knowledge base, the search paths used to move from the initial state to the complete problem space have been structured to emulate a recursive binary tree, where each node is either a query to the user or a fact inferred by the inference engine, each branch is based on the response to the query, and each terminal leaf is a problem state. For example, a node may consist of "is the printer is plugged in?". The set of possible answers determines the number of exits from the node; with this example they might be "yes" or "no". This is analogous to collecting a set of facts, "the printer is plugged in" or "the printer is not plugged in." The answering of the question corresponds to the consequent of a rule. Determining whether or not to fire a rule and test the premise corresponds to testing for the presence of the effects of a parent node's corresponding consequent. Continuing with the example, the parent node is "is the printer turned on?" with possible answers "yes" and "no". The current node, "is the printer plugged in?" is a child connected to the "yes" exit from the parent. In order to visit the child, the parent node must have been visited and exited via the "yes" arc. Mapping this to the rule representation, in order to fire the second rule (child node) the first rule must have asserted facts which allow the premise of the second rule to fire. So, in the example, the premise of the second rule would be "if (printer turned off)". So translation maps answering the node's question (choosing an exit arc) to a rule consequence and the parent's exit arc to a rule premise.

Currently, all queries to the user concerning states of the configuration require responses which can be answered if the user makes some direct observation from the workstation (i.e. "is your network interface turned 'ON' or 'OFF'?"). Some problem conditions exist which cannot be uniquely isolated by direct observation responses. For example, if too many users are logged on to a file server to allow an additional user to log on, the user may not be able to tell if he is not being allowed access due to server overbooking or an invalid user account. Without some external information from the server administrator, the user does not have a mechanism to identify which of these problem conditions is true while sitting at the workstation. Under these conditions, the current system halts and displays a list of all the possible problem conditions which fit the known information and suggests sources for the external information which can

further isolate the exact problem. Future expansion of the system could provide an option to wait for the user to retrieve the information and the proceed.

With regard to the formalization of the rule schemas, the rules have been classified into these categories: phase control, queries, configuration inference, and solution suggestion. These categories are defined as follows:

Phase control rules:

IF current-phase-completed
THEN assert-begin-next-phase

These rules act as flow control "traffic cops" during the execution of the inference engine. The real purpose of including phase control within **MacDoctor** is to force all queries to the user to take place before any suggested solutions are displayed. This is an issue in those cases where the system is diagnosing multiple problems and identifies a solution to one problem before posing all the queries required to isolate the remaining problems.

Query rules:

IF query-phase AND device-state-needed
THEN request-state-from-user
AND assert-device-state

These rules are fired during the query phase to pose questions to the user when information about the state of a device or configuration component is needed. The queries are specifically designed to constrain the user to a "yes/no" or "on/off" response. The "request-state-from-user" attribute is used to define the query string that is displayed to the user and to receive the user response. The "assert-device-state" attribute is used to assert a fact into the fact list which defines the state of the device, based on the response from the user. This fact, when added to the fact list, typically fires either another query rule, a configuration rule, or defines a terminal problem condition.

Configuration inference rules:

IF device-state-known
THEN assert-derived-facts

These rules are fired by facts asserted by the query rules, and are used to define facts inferred from known device states. For example, if a user provides a response which determines that a file server is visible, a configuration inference rule would fire which would infer that the network interface is on, the network is active, and the server is up.

Solution suggestion rules:

IF problem-condition-known

THEN assert-problem-solution

IF problem-solution-known
THEN display-problem-solution

These rules are fired by facts asserted from either query rules or configuration inference rules, and are intended to define solutions to isolated problems and then display the solutions to the user. The "problem-condition-known" attribute is either a problem definition or device state which defines a problem. "Assert-problem-solution" defines the solution text and then "display-problem-solution" displays the text to the user.

Implementation

The implementation of **MacDoctor** was written in Think C 3.2 on a Macintosh II. Macintosh was chosen for its user interface and Think C for its software development environment. The software was based on the general intention to embed the CLIPS rule engine within a C application. CLIPS-to-application communication was accomplished through the creation of a user function which interacted with the user through Macintosh user interface. The user function was defined in the CLIPS environment as a parameter returning function. The function was passed the node's question ("Is the printer plugged in?") and returned the user's response to the question ("yes", "no" etc). Within the CLIPS language, the function call was embedded within an assertion. The assertion statement were of the form:

'(assert (printer-state = (user-dialog "Is printer plugged in?" "yes" "no")))'

The **user-dialog** function was written in C and designed to present to first parameter in a user dialog window with the remaining parameters as answer buttons. The answer buttons are mouse selectable fields on the window. The user-dialog function creates a CLIPS symbol representing the user's selection, such as "yes" or "no".[1] This symbol is returned to the CLIPS environment and is used in the assertion.

Future Development Directions

At this point, the future plans for the development of **MacDoctor** include completion of the knowledge base to allow the application to recognize the aforementioned 80% of the problem domain, and to fully implement the Macintosh interface to the knowledge base and inference engine. Following that several directions are being considered, including:

- Implementation of a "machine learning" capability, whereby **MacDoctor** will be able to record and analyze patterns of user responses which lead to "dead ends" in the knowledge base (i.e. the user describes a problem

[1] Note that redundant attempts to create a CLIPS symbol simply returns the pre-existing symbol.

which **MacDoctor** does not recognize). The application could be given the ability to analyze the response patterns and alert the knowledge base maintainers of the occurrence of an unrecognized problem class. The maintainers can then use this information to extend the knowledge base of the application.

- Augmentation of the information-gathering capabilities of the application which would allow **MacDoctor** to determine several system configuration statistics and conditions instead of requesting all status information from the user. For example, enable the application with the capability to query the Chooser directly to determine the currently selected printer, rather than posing a query to the user requesting the name of the printer.

- Add a solution feedback mechanism which would allow the system to track the solution suggestions presented to the user and verify that the solution corrected the described problem. In those cases where the solution and the actual problem do not match, enable the system with an analysis capability which could determine if an alternative solution in the knowledge base would provide a "more correct" answer, or if an extension to the knowledge base is needed to handle the actual problem.

- Augment the user interface for the solution suggestions to expand the text description of the solution to display drawings and/or animation to better describe the corrective action required by the user. For example, if the suggested solution is to have the user check that the LocalTalk cable is connected to the printer port on the Mac, include an option which would display a short animation sequence illustrating the back of the Macintosh with a LocalTalk cable being connected.

The Authors

Dave Lavery is the Deputy Manager of the Artificial Intelligence and Robotics Research Program for the National Aeronautics and Space Administration (NASA). He is currently a part-time graduate student pursuing a Masters Degree in Computer Science at George Mason University. Contact: 202-453-2720, DLAVERY@NASAMAIL.AMES.NASA.GOV

Bill Brooks is a project manager with Advanced Decision Systems in Rosslyn, Virginia. He is currently a part time graduate student at George Mason University, working on a Masters Degree in Systems Engineering. Contact: 703-243-1611, WBROOKS@POTOMAC.ADS.COM