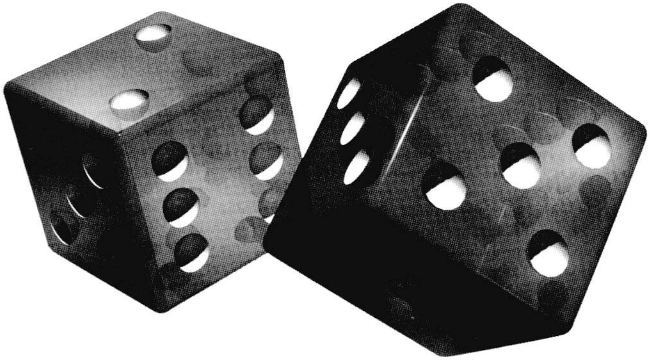
Lisa Burnell and Eric Horvitz



Structure and Chance: *Melding Logic and Probability for Software Debugging*

Software errors abound in the world of computing. Sophisticated computer programs rank high on the list of the most complex systems ever created by humankind. The complexity of a program or a set of interacting programs makes it extremely difficult to perform offline verification of run-time behavior. Thus, the creation and maintenance of program code is often linked to a process of incremental refinement and ongoing detection and correction of errors. To be sure, the detection and repair of program errors is an inescapable part of the process of software development. However, run-time software errors may be discovered in fielded applications days, months, or even years after the software was last modified—especially in applications composed of a plethora of separate programs created and updated by different people at different times. In such complex applications, software errors are revealed through the run-time interaction of hundreds of distinct processes competing for limited memory and CPU resources. Software developers and support engineers responsible for correcting software problems face difficult challenges in tracking down the source of run-time errors in complex applications. The information made available to engineers about the nature of a failure often leaves open a wide range of possibilities that must be sifted through carefully in searching for an underlying error.

Expert debugging often requires the investigative skills and honed intuitions of a Sherlock Holmes. Seasoned engineers narrow down possibilities and prioritize a search for the source of a program bug by carefully gathering important clues and considering the relevance of these pieces of evidence to the nature and location of program errors. Evidence may include erroneous run-time behaviors, debugging output provided by the operating system, and the logical structure of a program.

Deterministic information derived from the logical structure of a program and the values of program variables is critical for narrowing down the causes and locations of software errors. However, in addition to such deterministic information, engineers with long-term experience with debugging software systems also take advantage of their strong intuitions or "hunches" about the relative likelihoods that a problem is being caused by one or more types of error, based on what they see and their previous experiences. Although these uncertain intuitions are typically framed by information about the logical structure of portions of the code being explored, they are based on knowledge acquired over time about the ways that particular classes of software have been discovered to fail in the past.

The salience of uncertain reasoning in program debugging, in combination with the development, over the last decade, of expressive probabilistic representations and inference methods [6, 8, 11] stimulated us to investigate methods that take advantage of both logical and probabilistic inference to support the process of software debugging. We have applied automated reasoning methods to problems with maintaining and refining large, complex pieces of software that are used and refined over many years. Such *corporate legacy software* typically has a long history of evolution, growing and changing with contributions from many software engineers over time. In many cases, software may be poorly documented and incompletely tested.

Problems in legacy software may be detected years after a software update or modification. Frequently, people charged with the task of debugging find that engineers responsible for particular portions of program code are far removed from the code they created. Software engineers may have long since been promoted to other positions or left the company. In such situations, automated tools promise to deliver significant payoffs in increasing the efficiency of program understanding and debugging.

Our work was carried out as part of the Dump Analysis and Consulting System (DAACS) Project, a research effort centered at The University of Texas at Arlington and at the former Knowledge Systems Group of American Airlines in Fort Worth [3, 4]. We have focused specifically on the problem of identifying sequences of instructions that could harbor the source of run-time problems arising in the American Airlines Sabre airline reservation system, the most widely used time-shared reservation system in the world. The Sabre system runs under the IBM TPF operating system, which is widely used for transaction processing. As part of the research effort, several DAACS programs were developed to assist software engineers to determine sources of software errors in Sabre.

In this article, we will first describe the Sabre system and review the problem of interpreting and correcting problems with Sabre software as detected at run-time by the operating system. Then, we will summarize the fundamental problems that cause particular classes of operating system errors and highlight the relevance of methods for reasoning under uncertainty for diagnosing problems with legacy software. We will describe the DAACS advisory system, dwelling first on the logical structural analysis employed to identify execution paths that may contain the problem that led to a detected run-time error. Then we will focus on the uncertainty analysis that takes as input the results of the preceding logical analysis. The uncertainty analysis assigns likelihoods to alternate execution paths and to the different classes of error, given evidence about the structure and context of the paths. Finally, we review an example in detail and present some empirical results obtained through an informal validation of the system on Sabre run-time errors.

Sabre Problem Domain

The Sabre reservation system typically engages in thousands of transactions per second to support clients around the world. Hundreds of millions of transactions per day provide a real-world test for the correctness of the millions of lines of program code that define the functionality of the Sabre system. Experience shows that it is very difficult to thoroughly test Sabre software at design time. Syntactic and semantic errors continually slip through software development. As testimony to this problem, hundreds of program errors are noted in a typical week of Sabre operation. Many of these software problems arise in recently developed or modified program code. Other run-time errors uncover long-standing problems that come to attention only after special cases are handled or after rare interactions occur among programs that coexist in memory.

Operating Systems and Memory Management

We have concentrated on diagnosing the source of an abnormal termination of mainframe assembler language programs. Abnormal termination occurs when a program or subprogram in a time-shared environment violates an operating system constraint. For example, an attempt may have been made by a process to reference an area of memory that is not allocated to that process. In this article, we focus on problems with program segments attempting to reference pages of memory that are protected by other processes.

In response to a request to Sabre for, say, a list of airline fares, a collection of program segments are activated to process the query. Each segment may be thought of as a function or subroutine within a traditional program. At activation, all of the segments are loaded into memory and a structure called the entry control block (ECB) is created in memory. All segments of the program can access this memory. The ECB contains control and status information and a small amount of memory for a work area ("scratch data"). Sixteen fixed-size areas of memory called core blocks may be dynamically requested or freed by any of the program segments. The location, size, and status (free versus assigned or held) of all core blocks are recorded in the ECB. Working memory is divided into protected and unprotected pages, indicated by the protect keys of 0 and 1, respectively. As working memory is filled, the operating system makes new protected areas available to programs by setting the segment protect keys to 1 when the memory is free for use. Free pages of memory may be reprotected by other processes.

When an operating system detects a memory access violation, data is collected and formatted in a trace of recent history called a memory dumpfile, or dump. Engineers typically debug complex mainframe assembler language problems by poring over dumps from programs that terminate abnormally. A typical dumpfile contains a snapshot of relevant parts of memory at the time an illegal operation occurred. This trace includes information about the register contents, processor status, program counter, program object code, and the memory location and contents of program data areas.

Software engineers, armed with a memory dumpfile and a program listing, pursue the source of a problem by identifying the specific instruction that led directly to the program termination and then tracing backward through the operation of the program along execution paths in search of the principal cause of the termination. A valid *execution path* is a sequence of program instructions that could have been executed, depending on the outcome of conditional branch instructions.

Debugging Sabre software frequently involves decision-making under uncertainty because the information gleaned from a memory dump and recent history of a program's operation before the system encountered an illegal operation is often an incomplete description of the software error. We may be uncertain about the identity of the execution path taken to reach the error. Frequently, several alternative paths must be considered. Indeed, much of the effort of debugging is expended on identifying the failing path. Analysts may also be uncertain about the different errors possible given an identified execution path. Besides identifying and tracing the structure underlying a problem via the identification of execution paths, expert engineers may use other information, such as the detailed composition of execution paths, and their personal knowledge about the prior likelihood of various classes of error, given the type of operating system violation noted.

Classes of Run-Time Error

Several categories of fundamental software errors are detected and recorded by the TPF operating system used for Sabre. Memory addressing problems account for a majority of the errors in the software. There are several types of memory addressing problems. An illegal memory reference occurs when an instruction references (through either a read or a write operation) an address in a page that is protected by the operating system for use by other programs or by the operating system itself. If the instruction attempts to write to such an illegal address, the error is called a protection exception. If the instruction is attempting to read from an illegal address, it is called a read protection exception. A get core error occurs when memory already allocated to a program is reallocated to the program. Conversely, a release core error occurs when a block of memory that is not allocated to a program is freed or released by that program. Core corruption refers to errors that occur when a program segment writes to an invalid area of memory but are not immediately detected by the operating system. In many of these cases, an operating system error occurs somewhere downstream when a side effect of the invalid memory change causes an operating system violation.

Moving beyond memory addressing errors, data exception errors occur when data is referenced that is not in the expected format. For example, the system may expect packed decimal but see an integer value. Another class of problems, called application time-out errors, occur when a program segment executes for more than a predetermined amount of time, e.g., 500 milliseconds. These errors will occur if a program segment generates an infinite loop. Errors that lead to looping include failure to account for all cases so as to ensure that a looping procedure will terminate.

Various errors in Sabre can be attributed to underlying patterns of programming oversight. For example, looping errors in Sabre stem partly from incomplete testing of error-handling situations, especially within special code that is added to handle rare situations. Testing and development tend to focus on common cases; rare situations and exotic error-handling typically do not receive a great deal of attention and validation from software developers. Thus, code developed for handling rare situations can be a source of errors. Other software problems leading to looping are errors in incrementing or, more generally, in adjusting the values of variables. This type of error includes such problems as a software engineer erroneously using a multiplication operator rather than an intended addition.

Program Understanding and Debugging

Our work draws on techniques and motivations from the area of program understanding. Program understanding tasks in debugging vary depending on the complexity of the programs being analyzed and on the level of program abstraction available for debugging. For example, large time-shared assembler programs provide challenges not usually encountered in microcomputer-based programs compiled from higher-level structured languages. With assembler programs, variables are pointers that can access any part of memory and the code is unstructured. Also, input-output, execution traces, and intended behavioral descriptions are typically unavailable and are difficult to derive. Therefore, to debug time-shared assembler programs, engineers must often examine program structure and infer intended program behavior.

Researchers have been conducting ongoing related work on automated program understanding (APU). APU tools build abstract representations of a program and related information to facilitate reasoning (see, for example, [5, 9, 12]). Most program understanding systems seek to match portions of programs to prototypical implementation plans. Applications pursued by APU researchers include student programming tutors, design recovery and reuse of software, and program language translation. In many APU projects, designers have sought to develop methods of interpreting and understanding an entire program segment. In the realm of automated program debuggers, the APU task involves understanding just enough about program behavior to identify failures in the code (see, for example, [2, 10]; additional related work and a comparison to our method are given in [4]). In theory, the debugging task is not as ambitious as that of comprehensive program understanding. However, APU debugging systems must operate on real-world software as it is written and can draw only on the information available in the existing memory dumpfiles. In the case of Sabre, the software we seek to understand is unstructured assembler code that has been modified over several decades and has never been tested comprehensively.

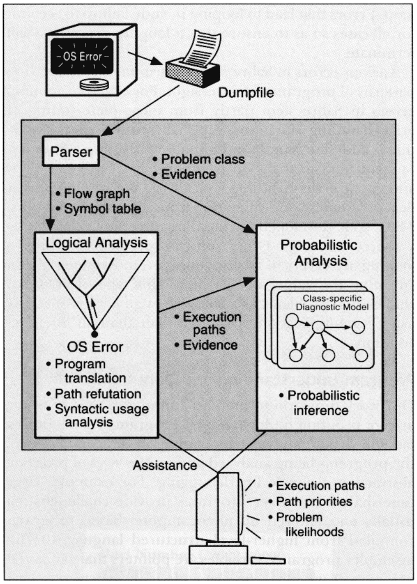


Figure 1. DAACS Architecture. This schematized overview of the logical and probabilistic components portrays how operating system errors are analyzed to create a graph of feasible execution paths. After the logical analysis, evidence and candidate paths are examined with belief networks to compute the likelihoods that paths harbor an error and the root causes for each path.

Overview of the DAACS System

Our goal is to assist analysts to determine the root error in a program segment. The root error is the initial instruction or sequence of instructions upstream in a segment that led to an erroneous situation detected by the operating system. In pursuit of the root error, DAACS employs several phases of analysis. We divide the DAACS system into two main components of problem-solving: prediagnosis and diagnosis. Prediagnosis refers to the basic parsing of the dumpfile information. The prediagnosis module extracts information about the type of operating system violation as well as additional relevant symptoms from a program dumpfile. These symptoms include those that experts first examine to generate their initial hypotheses about the software problem. As an example, the initial task in debugging illegal memory reference errors is to decode the offending instruction and determine the area of memory that instruction registers are referencing. These checks can be performed very quickly and with limited reasoning. In prediagnosis, the specific run-time event that caused the protection exception is determined, but the underlying, or root, cause remains unknown.

The function of the diagnosis module is to assist an engineer to narrow down the possible root causes of a run-time error. Diagnosis is composed of two phases of analysis: a deterministic, or logical, phase of analysis followed by a probabilistic phase (Figure 1). The logical phase determines feasible execution paths from information gleaned in prediagnosis and attempts to narrow the problem down to a single root error. If the logical phase fails to identify the problem, a set of paths is passed to the probabilistic phase. The probabilistic reasoning component is used to compute the probabilities of alternative hypotheses for each execution path identified by the logical reasoning component. The output of the diagnosis module is the root error, or a probabilistic ordering over paths, and the likely diagnoses and the evidence that contributed to belief in these diagnoses for each path.

Logical Inference Methods

At execution time, only a single execution path is taken by the computer. Unfortunately, it is frequently not possible to identify the path taken on the way to a run-time error. Operations at loops and transfer points branch the possible flow of control from one sequence of instructions to others based on the dynamic values assigned to variables. We may not have access to the dynamic values that defined control at run time. In these cases, we must consider the ways that the flow of control may have branched before an error was detected.

Logical constraints, derived from the structure of a program, can be employed to prune the space of paths down to those that were feasible in a particular situation. The logical component of the DAACS system analyzes the structure of a program, using information provided in a dumpfile. This deterministic phase of analysis considers blocks, looping, and branching instructions to identify feasible paths associated with an error.

Let us briefly focus on the identification of feasible partial execution paths. The partial execution paths of interest are sequences of instructions that could have been executed before a bad instruction was reached. Generating partial execution paths identifies relevant blocks of instructions to consider. The concept of generating partial execution paths with deterministic rules is known as *program slicing* [13]. In the process of generating a control flow graph, a symbol table is generated that defines all variables and the dependencies among them. The symbol table, the control flow graph, and the data in the dumpfile are used to rule out impossible paths.

Execution paths are typically constructed using a directed-graph representation of the program segment called a *control flow graph* (see [1] for a detailed description). Each node of the graph, called a basic block, contains a set of instructions that are executed in sequence. The only entrance into a basic block is via the first instruction of that block, and the only way to exit from the block and move to other blocks is via the last instruction in the block. The control flow graph representation facilitates the identification of transfer points and loops. From this information, feasible execution paths can be identified.

To construct an execution path for analysis, syntactic structures (listed in Table 1 in the following section) are matched against the control flow graph. A syntactic structure is a template or sequenced pattern of relevant events—all the events that could logically have caused a failure downstream. Syntactic structures are used to prove that large portions of a program are unrelated to erroneous run-time activity. For example, the SET-BAD syntactic structure matches if one or more paths exist in which the only relevant events are an initialization of the bad register (a set) and the instruction where the program terminated (the bad instruction).

Let us consider an attempt to debug an illegal memory reference. Assume that the operating system has noticed that a register (R2) has pointed to an illegal address. DAACS' logical analysis employs rules for chaining along an execution path and for pruning away irrelevant portions of code from consideration. The system considers the following events:

- **Bad instruction:** The instruction where the failure was noted by the operating system.
- **Set:** An instruction in which the bad register has been previously initialized; an example of a set is the instruction "LR R2, R5" where the value of register R2 is set to the contents of register R5. The new value is not dependent on the old value contained in register R2. Thus, any instructions that occurred prior to this instruction can be safely ignored.
- **Adjust:** An instruction in which the bad register value is modified but is somehow dependent on the previous value stored in the register. As an example, the instruction "LA R2, 3(R2)" adds three to the previous value stored in R2.
- **Use:** An instruction where the bad register is used to address a location in memory. For example, "L R5,20(R2)" loads register R5 with the value stored at 20(R2). For illegal memory references, if the value in R2 had been invalid, this instruction would have failed, so we know that the value in R2 was pointing to a legal area of memory when this instruction was executed. Any instructions that occurred prior to this instruction can be safely ignored.
- **Procedure call:** A procedure call to another program segment. The dumpfile may not have access to the instructions in the latter segment. If the other program segment has exited memory, for example, the memory dump will not contain the hexadecimal representation of the program to analyze. Given no other information, DAACS must assume that the register value could have been modified in the segment called. Thus, the call becomes a relevant event.

To construct relevant execution paths, we chain back from the sentry event noticed by the operating system as an indication of an error. Starting at the block of instructions containing the bad instruction, we trace the execution backward, in the direction opposite to that of program control flow, and continue to add additional blocks of the program until we reach either an instruction that uses or sets the bad register or the first block in the program. If the bad instruction is inside one or more loops, we must consider two cases: the loop was completely executed one or more times or the loop had not completed. These cases generate at least two possible paths.

Examples of execution paths created by DAACS are shown in Figure 2. In both (a) and (b), the instruction where the program abnormally terminated occurs in BLOCK 4 and there are two feasible complete execution

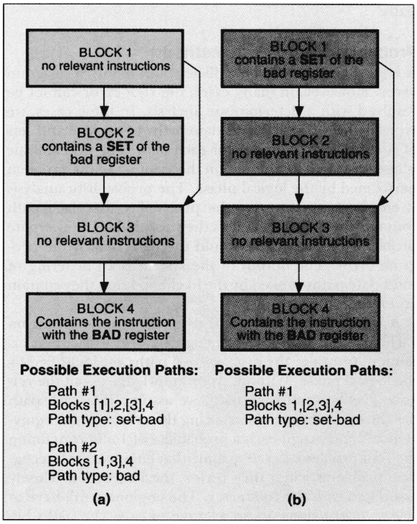


Figure 2. Sample execution paths. Two possible execution paths are treated distinctly in (a) and are combined in (b), based on the syntactic structure of the bad register.

paths that reach BLOCK 4. The first complete execution path is the sequence of blocks 1, 2, 3, and 4. The second complete execution path is the sequence of blocks 1, 3, and 4.

In Figure 2(a), the only relevant instructions in the program are the SET instruction in BLOCK 2 and the BAD instruction in BLOCK 4. By matching the syntactic structures against the control flow graph, we find two possible partial execution paths can be formed—one in which the SET instruction is executed and one in which it is not executed. Thus, for this case, DAACS must analyze two execution paths. In Figure 2(b), the relevant instructions are the same as in Figure 2(a), with the exception that the SET instruction occurs in BLOCK 1 instead of in BLOCK 2. In this case, only one partial execution path is formed, because the same sequence of relevant instructions—the SET and the BAD instructions—will be executed whether we execute the instructions in blocks 1, 2, 3, and 4 or, bypassing BLOCK 2, take the branch that executes the instructions in blocks 1, 3, and 4.

DAACS can sometimes fail to identify paths containing the root cause of the error because the structural analysis may fail to find all feasible paths. This situation can occur if the program segment contains a branch-on-register instruction. Such an instruction branches the program to a location specified in a dynamically assigned register. If DAACS cannot determine the value this register contained when the instruction was executed, it will ignore the branch instruction and thus fail to create a feasible path.

Probabilistic Inference Methods

A logical analysis can be sufficient to identify a program error. However, in many cases, the root error cannot be resolved with a deterministic analysis. In these cases, we typically have a set of candidate paths to explore and sets of root errors to consider for each path. The probabilistic phase leverages its analysis on the framing of the problem performed by the logical phase. The probabilistic analysis is employed to compute the probability that each path contains the errors as well as the probabilities of alternate problems on each path should the path harbor the program error. The output of the system is an ordering of candidate paths ranked by the likelihood that they contain an error.

A ranking over paths and over the types of error on each path can help software engineers prioritize their attention. Consider the case where n paths are identified by the logical phase. Without prior knowledge about the relative likelihood of each path, we assume that each path has the same chance of harboring the error ("event equivalence") and assign each a probability of $1/n$ of containing the root problem. Let us assume that engineers will recognize problems when they review the instructions closely associated with the root error. The engineers will have to review instructions on an average of $(n + 1)/2$ paths before discovering the error, or about half the candidate paths. Sorting the paths by making available more informative probability distributions over the candidate paths can significantly decrease the average number of paths reviewed. In addition, prioritizing the alternate problems for each path by probability can speed the time it takes to rule out paths.

In the probabilistic component of DAACS, we employ belief networks (see "Bayesian Networks," in this issue) to compute the likelihood of alternate root causes of an error given the type of error noted by the operating system and several classes of observations. Belief networks—directed acyclic graphs with nodes representing random variables and directed arcs representing probabilistic dependencies among those variables—are an efficient way to represent joint probability distributions, because they facilitate the explicit expression of independence among variables. The representation also allows experts to graphically structure and review causal relationships among variables they are familiar with.

In practice, probabilistic-inference algorithms are applied to belief networks to compute the probability distributions over the values of variables in the network conditioned on the specified values of some subset of variables (see reviews of this work in [6, 8, 11]. In most applications of belief networks, users are most interested in reviewing probabilities computed for values of nodes representing variables that are hidden from direct inspection, such as diseases in a patient. We set the values of nodes in the network that represent variables we have observed—such as a patient's symptoms and test results—and perform Bayesian updating on all of the nodes to revise the probabilities over the values of the hidden variables. In the case of the DAACS probabilistic phase, for each path we are given a set of findings about the error and structure of the path and we are interested in the likelihood that the path contains the error and in the probabilities of different kinds of error should the path indeed harbor the error.

The structure and probabilistic relationships for the belief networks in the DAACS system were acquired from experienced analysts who analyze on a daily basis dumpfiles of programs with which they are not intimately familiar. The structure of a sample belief network is shown in Figure 3. Knowledge-acquisition sessions were conducted over a 12-month period with four expert-level and two intermediate-level dumpfile debuggers. Two chief experts, each with more than 10 years of experience, resolved disagreements and validated the structure and assessments of the belief network models. In the current version of DAACS, we assume that there is single root error and represent alternative root errors as values of a ROOT ERROR variable. To construct the belief networks, we assessed from experts probabilities of the form: p(*path finding* $x = j$|ROOT ERROR $= y$) for all findings x and root errors y. During analysis with the probabilistic phase, we perform Bayesian inference on the belief networks to compute the probabilities p(ROOT ERROR $= y$|*set of path findings* X) for each path.

We found it useful to build distinct belief networks for specific contexts defined by the general classes of illegal memory reference errors noted by the operating system. Custom tailoring the models for different contexts helped to focus the modeling and scoring of the belief networks and promised to provide more discriminating diagnostic models. At run time, DAACS selects the appropriate belief network depending on the context noted. Belief networks relating sets of key observations to root causes were constructed for the following classes of problems:

• PAST HELD: A base register is pointing past the end of a held core block. The register is "near" (as defined by experts) a valid area of memory.
• ADDRESS PAST: An address is pointing past the end of a held core block. The base register is pointing in a core block, but the base register + displacement + index register is pointing outside the core block.
• LENGTH PAST: An address + length (e.g., in a move character instruction) is pointing past the end of a held core block. The address is pointing in a core block, but the address + length is pointing outside the core block.
• REGISTER 0: The base register contains the value zero.
• NO REFERENCE: The base register is not in close proximity to any valid area of memory.
• NOT HELD: The base register is pointing inside a core block that has been freed.

The root errors considered in the belief networks were structured and assessed in terms of types of errors, each type referring to a set of specific hypotheses. These abstract types capture the level at which actual errors are investigated and reported. An example of a specific entity in the BAD LOOP type is "Loop index register never initialized to a proper value."

The root error hypothesis classes considered in one of the DAACS belief networks includes the following errors:

• BAD SET: Instructions on the path have improperly initialized the erroneous variable;
• BAD ADJUST: Instructions on the path have improperly modified a value stored in the variable (e.g., addition, multiplication, shift instructions);
• ENTERED BAD: The variable is not initialized or modified on the path, and so was assigned an invalid value before entering the segment being analyzed;
• BAD LOOP: There are errors in loop index initializations, adjustments, or exit tests that led to the erroneous value of the variable;
• GOOD PATH: No errors occurred on this path.

Observations about the program and paths considered in the DAACS belief networks include:

• SYNTACTIC STRUCTURE: The structure of a path.
• CLOSE REGS: The proximity of values of variables to the value assigned to the erroneous variable. The appearance of close values is an indication of BAD SET errors.
• NEG REGS: The presence of large negative values, which support BAD ADJUST and BAD LOOP errors.
• BORDER PROXIMITY: A measure of the distance to the end of a valid memory block that is being addressed by actions along a path, e.g., a variable initialized prior to a loop entry to point near the end of a block indicates a faulty initialization, particularly if the loop construction itself shows no signs of errors.
• CORE CORRUPTION EVIDENCE: Key locations in the ECB are examined for "unusual" (as defined by experts) values. For example, if the first byte of any core block reference word (which records the starting address, length, and allocation status of a core block) is not zero, core corruption of the ECB may have occurred. Corruption evidence strongly supports, but does not prove, the other-segment hypothesis.

The values of some of the belief network variables are displayed in Table 1. The definitions of some of these distinctions rely on specifications of such details as measures and distances in programs. For example, the notion of border proximity is useful in the PAST HELD problem context. Border proximity is a measure of the proximity of an instruction that initializes a register to the end of a held core block. It captures the notion that "fencepost" errors—errors that occur when limits are not carefully

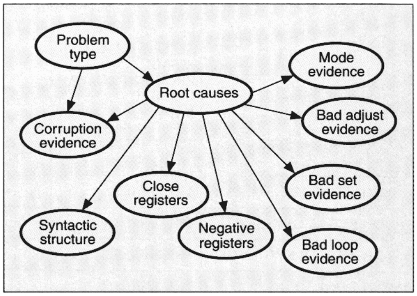


Figure 3. Sample belief network used for inferring the likelihoods of memory reference errors. The PROBLEM TYPE node refers to the class of problem. This node is implicit in DAACS because the belief networks are conditioned on a problem context, that is, selected based on the class of observed run-time failure.

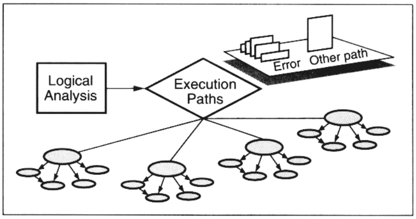


Figure 4. Belief networks tailored to the noted software problem class are applied to each feasible execution path identified in the structural analysis. The bar graph portrays the use of a probabilistic analysis to compute the probability that the software fault is either on the path being examined or on another path. Information is also computed on the relative likelihood of different problems, given that the problem exists on the path being analyzed.

Table 1. Values of variables represented in one of the DAACS belief networks.

ROOT CAUSE	GOOD PATH BAD ADJUSTMENT BAD SET ENTERED BAD OTHER SEGMENT BAD LOOP BLOCK RELEASED BAD ADDR MODE	A mutually exclusive set of hypotheses about the error for a given path.
SYNTACTIC STRUCTURE	SET BAD USE ADJUST BAD USE PROC BAD SET ADJUST BAD USE PROC ADJUST BAD SET PROC BAD SET PROC ADJUST BAD PROC ADJUST BAD USE BAD BAD	Possible syntactic structures for a path. For example, a SET ADJUST BAD structure means that on the path being analyzed, the only relevant events are an initialization of the bad register (a set), one or more adjust-type instructions (e.g., incrementing the register by 2), and the instruction where the program terminated (the bad instruction).
CLOSE REGISTERS	TRUE/FALSE	Do any of the other registers have a value that is close to the value in the bad register?
NEGATIVE REGISTERS	TRUE/FALSE	Do any registers have a negative value?
BAD SET EVIDENCE	TRUE/FALSE	Is there evidence that the bad register was initialized improperly?
BAD ADJUST EVIDENCE	TRUE/FALSE	Is there evidence that the bad register was modified improperly?
BAD LOOP EVIDENCE	TRUE/FALSE	Is there evidence of loop construct problems, e.g., failure to initialize a loop index register or a faulty exit test?
BAD MODE EVIDENCE	TRUE/FALSE	Is there evidence that the program is operating in the wrong addressing (24- or 31-bit) mode?
CORRUPTION EVIDENCE	TRUE/FALSE	Do key locations in the ECB contain unusual values?

assigned—are more likely when initializations and loops occur near memory borders. For example, the closer an initialization is to the end of a held core block, the stronger the belief is that the initialization will be faulty. The belief that a BAD SET error is true is increased when the observation BORDER PROXIMITY is noted in the PAST HELD context. The BAD SET hypothesis cannot be logically proven using information about border proximity; we do not know if the programmer intended to initialize the register close to the end of a core block and programmed with care about the memory constraints. However, the finding is a positive update on the likelihood of a BAD SET error.

As schematized in Figure 4, should the logical phase fail to isolate a single error, a belief network is selected for the problem context at hand and is loaded into the system. For each execution path identified by the logical phase, evidence from the dumpfile and the path analysis are used to instantiate the belief network, inference is performed, and likelihoods are assigned to alternate hypotheses.

One of the values of the root error node is the diagnosis GOOD PATH, referring to the situation where the path does not contain the problem. This value allows us to consider the likelihood that each path contains the problem and to sort the paths by likelihood. The hypotheses for each path include the hypothesis GOOD PATH. The belief that a path contains the error is $1 - p(\text{ROOT CAUSE} = \text{GOOD PATH} | \textit{path findings } X)$. For n independent paths identified by the logical phase of the analysis, we assert event equivalence and assume equal prior probabilities $1/n$ before inference. After each execution path is analyzed, the probability of error is normalized over all paths so that the probability of error on all paths sums to 1. The system provides an ordering over the likelihood of paths harboring the error, and over the likelihoods of different types of error for each path, given that it contains the error.

To date, we have made several assumptions in DAACS research. First, we make the assumption that problems leading to observations about invalid memory access are located on a single execution path. This assumption is violated when some paths share some relevant structure. Second, we assume all paths are equally likely to harbor the error. As we will describe later, we are investigating means for relaxing this assumption. Third, we assume the paths identified by the logical analysis serve as the complete set of possibilities. As we mentioned earlier, in some cases we may not generate all valid paths. Finally, we have not represented probabilistic dependencies among evidence conditioned on different root causes in the current version of DAACS. Despite our assumptions, we found the results of the system to be promising.

A Sample Session

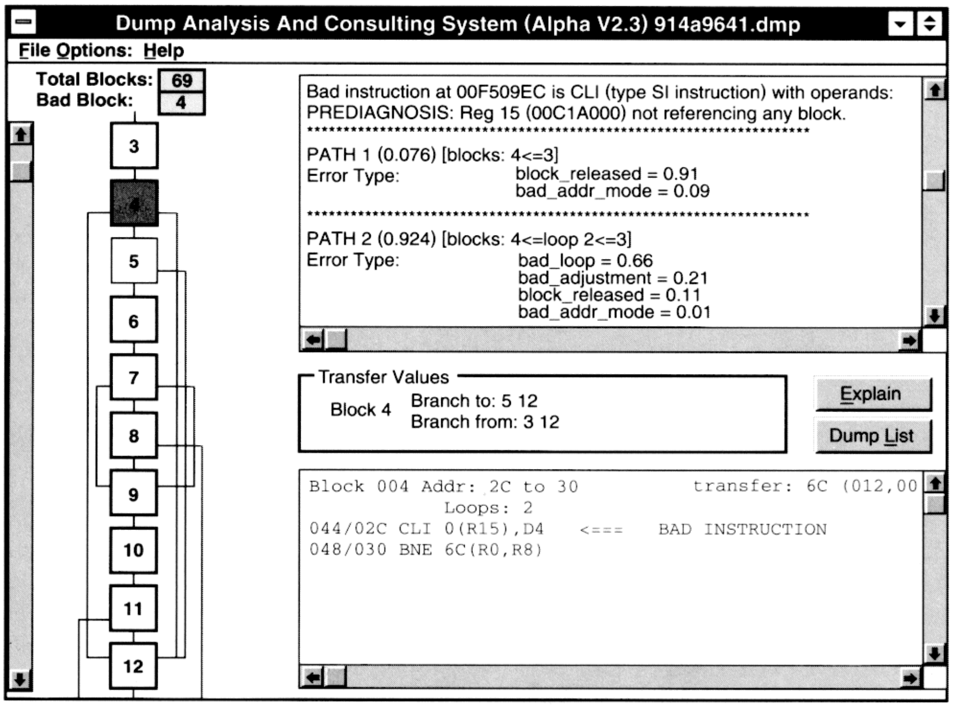
Let us consider the diagnosis of a rather straightforward illegal memory reference in which a bad register is not referencing any valid area of memory, nor is the reference close to any valid memory area. Figure 5 shows the main DAACS screen for this example. The user interface displays the control flow graph, including all blocks and control flow. Branches between the blocks at the left side of the figure appear as lines. The blocks on the right side are forward-pointing branches, indicating transfer to higher-numbered blocks. Branches on the left side of the blocks are backward-pointing branches, indicating that a transfer is made to lower-numbered blocks. Such pointers identify the occurrence of a loop. By pointing with a mouse and clicking on a path number, the user interface highlights the control flow for that path. By clicking on a block, the user interface highlights control flow for that block, and the immediate predecessor and successors in the "Transfer Values" window are listed. Clicking on a block also displays the instructions in the block in the lower right window. Additionally, the bad instruction, register contents, symptoms, and diagnosis are presented, and options for dumpfile listing and for generating an explanation or description of the findings are available to the user.

In our example, the top right window displays the information that the bad instruction is a CLI (Compare Logical Immediate) at address 00F509EC. The problem indicated is that register 15 is not referencing any core block. The control flow graph on the left of the screen highlights the block containing the bad CLI instruction. In this case, the DAACS logical analysis generates two feasible paths to the error. On path 1, the relevant events are an initialization of register 15 in block 3 and the bad instruction. This gives us a syntactic structure of SET-BAD.

```
************************************************************
EVIDENCE COLLECTION FOR PATH 1 (likelihood = 0.076):
  Reverse path is:  4 <= 3
  Path type:        set, bad
  Findings:
     Offset 034: FLIPC affects Level CR1. Can't prove that R15 reference.
     Offset 038: R15 set to A74935, address of 15(R1).
                 Referencing HELD core block CR0
************************************************************
EVIDENCE COLLECTION FOR PATH 2 (likelihood = 0.924):
  Reverse path is:  4 <= LOOP 2 <= 3
  Path type:        use, adjust, bad
  Findings:
     Offset 08C: NEGATIVE REGISTER R14 used in BCT instruction
     Offset 06C: R15 adjusted by 1
     Offset 084: R15 adjusted by 1
     Offset 222: R15 adjusted by UNKNOWN value
     Offset 234: R15 adjusted by UNKNOWN value
************************************************************
```

Figure 5. Sample output from a DAACS analysis of an illegal memory reference error. Some auxiliary information provided by DAACS has been deleted for simplification.

Figure 6. Findings reported by DAACS for the illegal memory reference example.



```
Bad instruction at 00F509EC is CLI (type SI instruction) with operands:
PREDIAGNOSIS: Reg 15 (00C1A000) not referencing any block.
************************************************************
PATH 1 (0.076) [blocks: 4<=3]
Error Type:             block_released = 0.91
                        bad_addr_mode = 0.09
************************************************************
PATH 2 (0.924) [blocks: 4<=loop 2<=3]
Error Type:             bad_loop = 0.66
                        bad_adjustment = 0.21
                        block_released = 0.11
                        bad_addr_mode = 0.01

Transfer Values
  Block 4   Branch to: 5 12
            Branch from: 3 12

Block 004 Addr: 2C to 30          transfer: 6C (012,00
             Loops: 2
044/02C CLI 0(R15),D4     <===   BAD INSTRUCTION
048/030 BNE 6C(R0,R8)
```

The relevant instructions for path 2 are the bad instruction, which is marked as both a BAD and a USE based on the loop structure, and on multiple adjust type instructions within the loop. This gives us a syntactic structure of USE-ADJUST-BAD. Before we consider path-specific evidence, the prior probability that each of these two paths contains the error is set to 1/2, or 0.50.

After collecting and reasoning about evidence found on each path, DAACS reports that the second path is significantly more likely than the first. To understand these results, let's examine the findings reported by DAACS. When we click on the "Explain" button, the justification for the diagnosis appears as displayed in Figure 6.

For the display of the belief in hypotheses for an individual path, DAACS treats the path as the actual execution path and relays information about the relative likelihoods of different types of root problem. In our example, the diagnostic output for Path 1 indicates that if this was the path that was taken, the only hypotheses are BLOCK RELEASED and BAD ADDR MODE, and the former is much more likely than the latter. Internally, DAACS keeps track of the probability that each path is error free—that the problem is on a different path. These probabilities are used to prioritize the paths.

For Path 1, DAACS reports that register 15 (R15) is set near (within 15 bytes) the top of an allocated core block. Thus, BAD SET EVIDENCE is set to false. The syntactic structure is noted to be SET-BAD, and BAD ADJUST EVIDENCE is set to false. Likewise, this path contains no loops, so BAD LOOP EVIDENCE is set to false. There are no calls to other segments between the initialization in block 3 and the bad instruction, so the OTHER SEGMENT hypothesis is also ruled out. There is no evidence of corruption, and no registers have a value close to register R15. However, a FLIPC on core block CR1 is executed before the bad instruction. A FLIPC swaps two core blocks, which may deallocate a previously allocated block of memory. We do not have enough information in this case to determine if this affects R15, because the dump does not contain the starting address of CR1. If this were the only execution path in the program, the FLIPC instruction would be the most likely explanation for the error, which explains why the BLOCK RELEASED hypothesis is assigned a large likelihood ($p = 0.91$).

In Path 2, DAACS reports on the four adjust-type instructions that may have been executed on this path. Because DAACS cannot determine by what value R15 was adjusted in two of the instructions (offsets 222 and 234), there is moderate support for the BAD ADJUST hypothesis. The syntactic structure rules out the BAD-SET and OTHER SEGMENT hypotheses. An important finding on this path is the negative register R14, which we can determine is being used as a loop exit condition in the BCT instruction at offset 08C. DAACS concludes that a BAD LOOP is the most likely diagnosis for this path and that there is considerable evidence to support this conclusion relative to the evidence found for Path 1.

The next step in diagnosis could be to pursue the most likely error on the most likely path—in this case by determining why register 14 used in the loop on Path 2 is negative. New syntactic structures and execution paths for register 14 would need to be constructed, and a belief network for diagnosing loop index problems would be loaded. The current implementation does not carry the level of diagnosis this far. However, such multilevel diagnosis is feasible.

Empirical Validation

We validated the DAACS advisory system informally by considering the savings in the time required by analysts using the system to identify a root error. We also tested the accuracy of the system's recommendations. A minimum of two analysts were used for the tests. The experience level of the analysts ranged from three months to over two years. All tests were conducted with program segments running in the Sabre reservation system and with actual real-time errors. Test dumps were selected at random from those available over a period of a year and a half. Moreover, all comparisons between the performances of analysts with and without the DAACS system assumed that the analysts were diagnosing the dumpfile to the same depth as DAACS.

For 65% of the cases examined, DAACS correctly diagnosed the root error solely with the logical analysis, based on the program structure data in the dumpfile. In another 10% of the cases, the probabilistic analysis diagnosed the correct path, identified the root error as the most likely error type, and reported the proper events on the path that led to the failure. In the remaining 25% of the cases, the system did not assign the highest probability to the actual error. For a large fraction of these cases, problems were noted to be centered within the logical phase of the analysis. In many of these cases, the logical inference procedures failed to identify feasible paths containing the root errors.

Time-saving measures performed to date include estimates on debugging efficiency obtained from intermediate- to expert-level analysts using the system and estimates based on the number of instructions that must be examined to diagnose the error. These estimates need to be refined with additional, controlled studies. Nonetheless, the results to date are promising. Analysts participating in the validation used the system for a minimum of one month during the course of their normal duties, which included dumpfile debugging. Based on this usage, the analysts estimated a 25% to 50% reduction in debugging time over that of their current method.

We have also been interested in the average number of instructions that must be examined to debug software with and without DAACS advice. The DAACS approach shows promise for helping to minimize the number of instructions reviewed by engineers. In the 75% of cases where the DAACS system correctly identified the leading root cause, the system reported, on average, 4.25 relevant instructions for users to review. Although we have not rigorously studied the average numbers of instructions reviewed by software engineers in the course of debugging Sabre problems, the average number of instructions in successful DAACS analyses appears to be significantly smaller. Program segments come in two sizes, the more common being 1,055 bytes, with an average instruction length of four bytes. Thus, we estimate that a program segment contains approximately 250 instructions. Although expert analysts no doubt have the ability to selectively focus on a fraction of the total number of instructions, we believe that the average number of instructions they review is significantly greater than four per case.

Summary and Future Directions

We described our work on integrating logical and probabilistic automated inference methods to assist engineers with diagnosis of the source of run-time errors in software. Our approach centers on the use of logical program understanding methods to generate a set of feasible execution paths and the use of probabilistic methods to prioritize the paths for examination and to determine the likelihood of errors on each path. Our initial exploration and validation of automated inference for debugging has highlighted the promise of employing an amalgam of logical and probabilistic reasoning methods for software diagnosis. We are interested in seeing the DAACS methodology extended in several ways. We believe that descendants of the DAACS system can perform with greater completeness and accuracy through the extension of the probabilistic models, with richer distinctions about the functionality, source, degree of verification, and age of the program code. Also, probability and logical reasoning methods can be interleaved in a variety of different ways. For example, uncertain reasoning can assist with the task of identifying execution paths. Probabilistic methods promise to be an important adjunct to the structural execution path identification procedures used in the current version of DAACS, especially for scaling up program analyses beyond the current focus on program segments. It may be useful to introduce a probabilistic analysis into the path identification procedures so as to consider the likelihood that a path that cannot be refuted with a logical analysis is irrelevant to an error. Also, probabilistic analyses can help us to determine the likelihood that alternate feasible execution paths were taken. In particular, information about the structure of a program, including such evidence as classes of interaction at branching points, can be useful in determining the likelihoods that specific candidate execution paths were traversed at run time. Such inferences would allow us to relax our assumption that valid paths have equal prior probabilities of being the actual run-time path. Additionally, we are interested in the use of decision-theoretic methods to focus the attention of path-identification analyses, to identify cost-effective evidence-gathering strategies, and to prioritize debugging tasks for a time-pressured software engineer [7]. Finally, we are excited about recent research in learning belief-network models from data. We expect that Bayesian methods for the automated construction of diagnostic models will one day enable us to harness case libraries of stored information about software errors for use in adaptive systems that can assist people with program understanding and debugging.

Acknowledgments

We are indebted to the many people whose support has contributed to this work, especially the members of the Realtime Coverage organization at American Airlines, including Casey Deller, Joe Burcham, Sandy Roach, Larry Buell, Daryl Cox, and many others. Special thanks also to Lynden Tennison and the members of the former American Airlines SCS Knowledge Systems group. We also thank David Heckerman, Jim Martin, Mark Peot, Bjarne Steensgaard, and the anonymous reviewers for their comments and suggestions. This work was partly supported by the Rockwell Science Center Palo Alto Laboratory IR&D Project on Decision Making Under Uncertainty.

References

1. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
2. Arbon, R.G., Atkinson, L., Chen, J., and Guida, C.A. TPF dump analyzer: A system to provide expert assistance to analysts in solving run-time program exceptions by deriving program intention from a TPF assembly language program. In *Proceedings of Innovative Application of Artificial Intelligence 4* (San Jose, Calif., July 12–16). AAAI, Menlo Park, Calif., 1992, pp. 71–88.
3. Burnell, L.J., and Horvitz, E.J. A synthesis of logical and probabilistic reasoning for program understanding and debugging. In *Proceedings of the 9th Conference on Uncertainty in Artificial Intelligence* (Washington, D.C., July 9–11). Morgan Kaufmann, San Mateo, Calif., 1993, pp. 285–291.
4. Burnell, L.J., and Talbot, S.E. Incorporating probabilistic reasoning in a reactive program debugging system. *IEEE Expert 9,* 1 (Feb. 1994), 15–20.
5. Hartman, J. Technical introduction to the First Workshop on Artificial Intelligence and Automated Program Understanding. In *Workshop Notes of the AAAI-92 Workshop Program: AI & Automated Program Understanding,* L. Van Sickle and J. Hartman, eds. (San Jose, Calif., July 12–16). AAAI, Menlo Park, Calif., 1992, pp. 8–30.
6. Heckerman, D., Horvitz, E., and Nathwani, B. Toward normative expert systems: Part I. The Pathfinder Project. *Methods Inf. Med. 31,* 90–105.
7. Horvitz, E.J. Reasoning under varying and uncertain resource constraints. In *Proceedings of the 7th National Conference on Artificial Intelligence* (Minneapolis, Minn., Aug. 21–26). Morgan Kaufmann, San Mateo, Calif., 1988, pp. 111–116.
8. Horvitz, E.J., Breese, J.S., and Henrion, M. Decision theory in expert systems and artificial intelligence. *Int. J. Approximate Reasoning, 2* (1988), pp. 247–302.
9. Kozaczynski, W., Letovsky, S., and Ning, J. A knowledge-based approach to software system understanding. In *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference* (Syracuse, N.Y., Sept. 22–25). IEEE, Los Alamitos, Calif., 1991, pp. 162–170.
10. Lenz, N.G., and Saelens, S.F.L. A knowledge-based system for MVS dump analysis. *IBM Syst. J. 30,* 3 (July 1991), 336–350.
11. Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufman, San Mateo, Calif., 1988.
12. Selfridge, P.G. Knowledge representation support for a software information system. In *Proceedings of the 7th IEEE Conference on AI Applications* (Miami Beach, Fla., Feb. 24–28). IEEE, Los Alamitos, Calif., 1991, pp. 134–140.
13. Weiser, M., and Lyle, J. Experiments on slicing-based debugging aids. In *Experimental Studies of Programmers,* E. Soloway and S. Iyengar, eds. Ablex, Norwood, N.J., 1986, pp. 187–197.

Continued on page 57

JOHN S. BREESE is a senior researcher in the Decision Theory Group at Microsoft Research. Current research interests focus on developing tools and methods for decision-theoretic reasoning, in particular the integration of normative methods with symbolic processing techniques.

KOOS ROMMELSE is a researcher in the Decision Theory Group at Microsoft Research. Current interests are in the development of tools for probabilistic inference, troubleshooting, and learning Bayesian networks from data. **Authors' Present Address:** Microsoft Research, One Microsoft Way 9S, Redmond, WA 98052-6399; email: heckerma, breese, koosr@microsoft.com.

Continued from page 41

About the Authors:

LISA BURNELL is a doctoral candidate in computer science and a faculty associate at The University of Texas at Arlington. Current interests include uncertain reasoning, intelligent decision-theoretic systems, and automated program understanding. **Author's Present Address:** Department of Computer Science and Engineering, The University of Texas at Arlington, 416 Yates Street, Nedderman Hall, Room 300, Arlington, TX 76019-0015; email: burnell@csr.uta.edu.

ERIC HORVITZ is a senior researcher in the Decision Theory Group at Microsoft Research. Current interests include decision-theoretic methods for time-critical decision-making and diagnosis, and applying probability and utility to solving problems in operating systems, user interfaces, information retrieval, and medicine. **Author's Present Address:** Microsoft Research, One Microsoft Way, Redmond, WA 98052-6399; email: horvitz@microsoft.com.

Continued from page 48

ference (TREC-2) (Gaithersburg, Md.). The National Institute of Standards and Technology, 1994, pp. 1–20.

5. Heckerman, D.E. A tractable algorithm for diagnosing multiple diseases. In *Proceedings of the 5th Workshop on Uncertainty in Artificial Intelligence* (Detroit, Mich.). 1989, pp. 162–173.
6. Howard, R.A., and Matheson, J.E. Influence diagrams. In *Readings on the Principles and Applications of Decision Analysis*, R.A. Howard and J.E. Matheson, Eds. Strategic Decisions Group, Menlo Park, Calif., 1981, pp. 721–762.
7. Maron, M.E., and Kuhns, J.L. On relevance, probabilistic indexing, and information retrieval. *J. ACM 7* (1960), 216–244.
8. Pearl, J. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann, San Mateo, Calif., 1988.
9. Robertson, S.E., and Sparck Jones, K. Relevance weighting of search terms. *J. Am. Soc. Inf. Sci. 27* (May–June 1976), 129–146.
10. Salton, G. *The SMART Retrieval System—Experiments in Automatic Document Processing.* Prentice-Hall, Fort Lee, N.J., 1971.
11. Turtle, H.R., and Croft, W.B. Inference networks for document retrieval. In *Proceedings of the 13th International Conference on Research and Development on Information Retrieval* (Brussels, Belgium), 1990, pp. 1–24.
12. Turtle, H.R., and Croft, W.B. Evaluation of an inference network-based retrieval model. *ACM Trans. Info. Sys., 9(3)* (1991), pp. 187–222.
13. van Rijsbergen, C.J. *Information Retrieval.* Butterworth, London, 1979.

ROBERT FUNG is a principal at Prevision Inc., a company that provides software tools and consulting services in Bayesian networks and influence diagrams. Current research interests include inference in Bayesian networks, probabilistic and decision-theoretic diagnosis and planning, and learning probabilistic representations from data. He received his doctorate from the Engineering-Economic Systems Department of Stanford University. **Author's Present Address:** Prevision Inc., 2817 Almeria St., Davis, CA 95616; email: fung@prevision.com

BRENDAN DEL FAVERO is a graduate student in Engineering-Economic Systems at Stanford University. Current research interest is in applying Bayesian methods to information retrieval. **Author's Present Address:** email: bdf@leland.stanford.edu