Identifying Software Problems Using Symptoms

Inhwan Lee　Ravishankar K. Iyer　Abhay Mehta

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W Main St., Urbana, IL 61801

Tandem Computers Incorporated
14231 Tandem Boulevard
Austin, TX 78728

Abstract

This paper presents an approach to automatically identify recurrent software failures using symptoms, in environments where many users run the same software. The approach is based on observations that the majority of field software failures in such environments are recurrences and that failures due to a single fault often share common symptoms. The paper proposes the comparison of failure symptoms, such as stack traces and symptom strings, as a strategy for identifying recurrences. This diagnosis strategy is applied using the actual field software failure data. The results obtained are compared with the diagnosis and repair logs by analysts. Results of such comparisons using the failure, diagnosis, and repair logs in two Tandem system software products show that between 75% and 95% of recurrences can be identified successfully by matching stack traces and symptom strings. Less than 10% of faults are misdiagnosed. These results indicate that automatic identification of recurrences based on their symptoms is possible.

1 Introduction

A field software failure can occur due to a known fault, a newly found fault, or an unidentified fault. Here these failures are referred to as a "recurrence," "first occurrence," or "unidentified," respectively. [Lee93a] showed that about 72% of reported field software failures in Tandem systems are recurrences. Considering that a quick succession of failures at a site, which are likely to be due to the same fault, are often reported in a single failure report, the actual percentage of recurrences can be higher. Recurrences are not unique in Tandem systems. A similar situation exists in IBM systems [Adams84] and AT&T systems [Levendel]. This shows that the software development process is not the only important factor. Recurrences can seriously degrade software dependability in the field.

Recurrences exist for several reasons. First, designing and testing a fix of a problem can take a significant amount of time. In the meantime, recurrences can occur at the same site or at other sites. Second, the installation of a fix sometimes means a planned outage. This may force users to postpone the installation and cause recurrences. Third, a purported fix of a problem can fail. Finally and probably most importantly, users who did not experience problems due to a certain fault often hesitate to install an available fix for fear that doing so will cause new problems, as is sometimes the case with fixes.

The impact of recurrences are: 1) more failures than predicted based on the number of faults, 2) wasted resources due to repeated data collection, reporting, and diagnosis of the same problem, and 3) delayed service to users even if solutions to problems are available. Preventive maintenance, which refers to the process of fixing a software fault in a user system when the fault did not cause a problem in the system, can potentially reduce the number of recurrences. But it costs resources. Besides, faults in a fix can cause new problems in user systems. Based on the failure and shipment data in IBM products, [Adams84] proposed that preventive maintenance be limited to a small number of highly visible faults. This result and the above reasons for recurrence indicate that recurrences will continue to be a significant part of field software failures.

In this paper, we present an approach to automatically identify recurrences based on their symptoms. The approach is based on an observation that failures due to the same fault often share common symptoms [Lee93a]. Specifically, we propose the comparison of stack traces and symptom strings as a strategy for identifying (i.e., diagnosing) recurrences. A stack trace is the history of procedure calls made by the active process at the time of a failure. It represents the software function that detected a problem. A symptom string uniquely identifies the code location at which a problem was detected. We applied the proposed diagnosis strategy using the failure data from two Tandem system software products. We then compared the results obtained with the actual Tandem diagnosis and repair logs. Results of the comparison showed that between 75% and 95% of recurrences can be identified successfully by matching stack traces and symptom strings. Less than 10% of faults are misdiagnosed. These results indicate that recurrences can be identified automatically based on their symptoms.

The diagnosis strategy is currently being implemented as an automatic diagnosis tool. The tool is envisioned to monitor many user systems connected

0363-8928/94 $3.00 © 1994 IEEE

by an on-line alarm system. Given a failure alarm, the tool will extract a stack trace and a symptom string from the failed machine, compare these with those from past failures, and determine whether the failure is a recurrence or due to a new fault. Such a fully-integrated tool is not up and working at this point. The benefits of developing and using such a diagnosis tool are 1) saving the wasted human effort of reporting and diagnosing the same problem repeatedly and 2) identifying an available fix or a workaround rapidly.

2 Related Work

Measurements on software errors have been performed by researchers. Some recent studies are the following. A census of Tandem system availability [Gray90] has shown that, as the reliability of hardware and maintenance improves significantly, software becomes the major source (62%) of outages in the Tandem system. [Sullivan91] investigated software defects and their impact on system availability using the data from the IBM/MVS system. An approach to use observed software defects to provide feedback on the development process was proposed in [Chillarege92]. [Lee93b] discussed a methodology for analyzing operating system fault tolerance and demonstrated the methodology through three case studies.

Symptoms of faults in computer systems have been studied using error logs. An information organization and data reduction concept, called *tuple*, for fault prediction was developed in [Tsao83]. Separation of an error log into transient and intermittent events, and failure prediction based on the shape of the interarrival time function were discussed in [Lin90]. A probabilistic methodology for recognizing the symptoms of persistent problems was proposed and illustrated using error data collected from an IBM 3081 and two CYBER systems [Iyer90].

Failure diagnosis attempts to locate the underlying faults of failures. Symptom directed diagnosis of system faults was discussed in [Maxion85]. [Latham86] discussed an expert system to help in analyzing crashes of the VMS operating system using the crash dump files and system event logs as data. [Maxion93] studied the detection and discrimination of network faults based on network traffic signatures. The *recreate* problem in identifying and diagnosing software failures in the field was discussed in [Chillarege93].

3 Measurements

The Tandem NonStop[1] system is a message-based multiprocessor system designed for on-line transaction processing (OLTP). The Tandem system software halts the processor on which it is running when it detects a nonrecoverable error. When a processor halt occurs, a memory dump is taken from the halted processor and sent to Tandem in the form of a Tandem Product Report (TPR). All diagnosis actions taken by analysts including the log of memory dump analysis are appended to each TPR.

[1]NonStop is a trademark of Tandem Computers Inc.

Failures in two Tandem system software products are used in this study. One product implements the low-level functions to support database applications and is referred to as DB in this paper. The other product implements network communication functions and is referred to as DC. These products run as processes and serve requests from user applications. Among analysts, DB is known to be robust, while DC is known to be not robust.

We first extracted all user-generated TPRs caused by faults in the two system software products for the past few years. We then extracted all preceding TPRs due to the same causes. During the measurement period, the products were modified many times due to bugfixes and minor enhancements. There was also a major revision. Both products are written in Transaction Application Language (TAL), which is similar to C. The size of each product is on the order of 10^5 lines of commented source code. In this paper the terms *processor halt* and *failure* were used interchangeably.

4 Diagnosis Strategy

A memory dump captures the processor state at the time of a failure. Given a dump, analysts investigate key failure symptoms such as the software function being executed, the apparent reason for the halt, and the error pattern (see Figure 1). Based on the symptoms, they attempt to identify the underlying fault by reasoning back through the error generation and propagation process. This diagnosis requires experience, a detailed knowledge of the operating system, and extensive reasoning. Although software failure diagnosis is a complex task that is hard to automate, it has been observed that failures due to the same software fault often have identical stack traces [Lee93a], suggesting that it may be possible to identify recurrences based on their symptoms.

A diagnosis strategy consists of a set of common symptoms and associated matching scheme to be used for identifying recurrences. The diagnosis strategy is determined once by off-line evaluation.

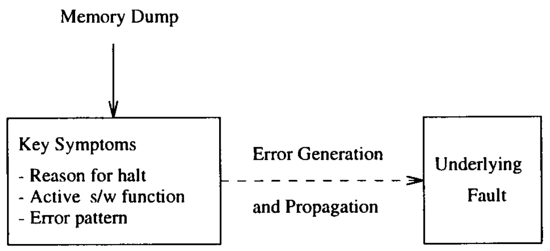

Figure 1: Software Failure Diagnosis

4.1 Common Symptoms

A question to ask is: What are the symptoms that are usually shared by the failures due to the same fault? Such common symptoms will be useful for identifying recurrences. Our experience shows that failures due to the same fault often share two types of common

```
Halt code → %00104 into PAGEFAULT
TOS →       %00365 into GET_ACCTENTRY
            %00220 into GET_FILEOP_INPUTBUF
            %00052 into READ_SETUP
            %00015 into READ_RQST
            %00446 into MAINLOOP
```

Figure 2: Sample Stack Trace with Offsets

symptoms: 1) certain local and shared data (*data-oriented symptoms*) and 2) code that was being executed (*code-oriented symptoms*). Code-oriented symptoms capture information such as the active process at the time of failure, the software function being executed (i.e., stack trace), and the exact code location where a problem was found (i.e., symptom string). Examples of data-oriented symptoms are the values of parameters passed between procedures in a stack trace and the state of certain local and global variables. In this study, we focused on the use of stack traces and symptom strings because we used failure reports (i.e., TPRs) generated by analysts, not actual dumps. Full data-oriented symptoms were not usually recorded in the failure reports, although they were available from the dumps.

Figure 2 shows a stack trace extracted from a failure. Each line represents a procedure, and the associated number represents the offset of the code location (i.e., the machine instruction) that called the next procedure from the beginning of the procedure, in octal words. In Figure 2 the (system) process that halted the processor is normally sitting in the procedure MAINLOOP. When the process receives a request, it serves the request by calling necessary procedures. In this case, the process detected a nonrecoverable error during the execution and halted the processor on which it is running. The set of procedures shown in Figure 2 is a stack trace for the failure. Each software failure has its stack trace.

The first line from the top shows an error handling procedure. There is an error handling procedure and associated *halt code* for each type of problem detection defined by software developers and system designers. In the sample shown in Figure 2, the error handling procedure shows that a page fault occurred while executing a code section in which a page fault is not supposed to occur. The actual stack trace consists of the procedure names beginning from the second line. The stack trace represents the software function that detected the problem. It is not necessarily related to the location of the underlying fault. The first procedure from the top, except for the error handling procedure, is called the procedure at the top of the stack (TOS) in this study.

The procedure at the TOS and the associated offset (i.e., "%00365 into GET_ACCTENTRY" in Figure 2), when combined with the software version information, uniquely identifies the code location at which problems were detected. The software version needs to be known because the procedure offset may change due to bug fixes or enhancements. DB developers designed the code such that when errors are detected by consistency checks (i.e., explicit software checks), an ASCII string (called symptom string) is inserted at the designated location of the process stack before asserting a processor halt, so that analysts can read it and recognize the location of problem detection regardless of software version. The symptom string consists of three parts that identify the source file name, the procedure name, and the software check that detected a problem.

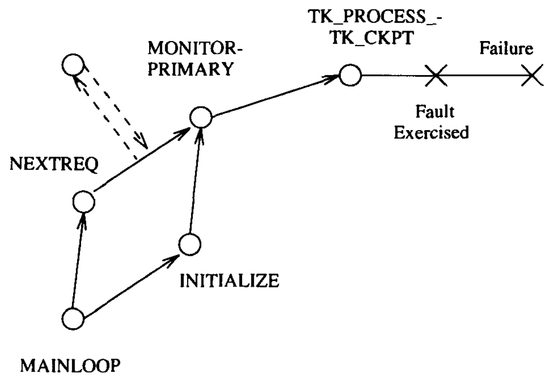

Figure 3: Detection near Faulty Code

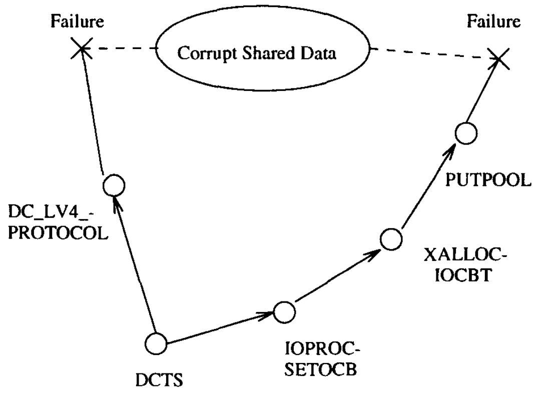

Figure 4: Detection after Corruption in Shared Data

Two extremes exist. First, a software fault can cause failures with different symptoms as illustrated in Figure 3. The figure shows a case in which a problem was detected near the faulty code section. A circle represents a procedure call and an arrow represents the execution within a procedure. The figure shows a failure in which the base procedure MAINLOOP called the procedure NEXTREQ, which in turn called the procedure MONITORPRIMARY. MONITORPRIMARY called the procedure TK_PROCESS_TK_CKPT in which a fault was exercised and a halt was asserted. In another failure, the same thing happened except that MAINLOOP reached MONITORPRIMARY through the procedure INITIALIZE. This was also shown in the figure. The chain of procedure calls forms a stack trace and is represented by a set of connected solid arrows in the figure. The dotted arrows represent a pair of a procedure call and return that does not explicitly appear in a stack trace. Because the software structure is modular, there can be different program paths to reach the faulty code section. Figure 3 shows two such paths. Each of the paths gives a distinct stack trace.

Figure 4 shows a case in which a wide range of corruption occurred in shared data. The dotted lines represent accesses to the shared data. The underlying fault was a developer's misunderstanding of data structure. In this case, any software function can detect some of the errors and assert a halt. This would lead to widely different stack traces, problem detection locations, and error patterns. Figure 4 shows two very different stack traces.

The second extreme to consider is that different faults can cause failures with identical symptoms. There was a case in which a processor halt was asserted while executing the procedure DC_LV4_PROTOCOL, which was called by the base procedure DCTS. The underlying fault was not providing a routine to handle a rare but legitimate sequence of events, which led the system to an inconsistent state. This failure scenario and the left-hand-side stack trace in Figure 4 show that different faults can cause identical symptoms (i.e., identical stack traces in this case).

4.2 Matching

Once a set of common symptoms is determined, the next question is: How do we compare failure symptoms (i.e., particular values of the common symptoms that were chosen to be used for the diagnosis)? Three types of matching can be considered: *complete matching*, *partial matching*, and *weighted matching*. Complete matching means that two failures are declared to be due to the same fault if their failure symptoms (e.g., two stack traces extracted from the two failures) are identical. Partial matching means two failures are declared to be due to the same fault if their failure symptoms are within a certain distance from each other, based on a predefined measure of distance. Partial matching can allow us to make a certain tradeoff under the two extremes discussed in Section 4.1. This issue will be discussed further in the next subsection and Section 6. Weighted matching is necessary when using several types of common symptoms. In weighted matching, a measure of similarity of two failures is determined by comparing their values of each type of common symptom. These measures are then combined to form an overall measure that represents the similarity of two failures in their symptoms, based on their weights. The weights for different types of common symptoms can be determined by an iterative performance evaluation and based on the knowledge of software structure and functionality.

In this study, we used the complete and partial matching of stack traces. Since the symptom string is a single piece of information, only complete matching can be used for matching symptom strings.

4.3 Evaluation Method

To evaluate the effectiveness of the proposed diagnosis strategy under the extremes described in Section 4.1, we considered *fault clusters* and *symptom clusters*. A fault cluster consists of all failures due to a fault. In this study fault clusters were formed based on Tandem diagnosis and repair logs. Given a set of failures, the set of fault clusters is unique. A symptom cluster consists of all failures that share certain common symptoms. As far as the diagnosis is concerned, failures in the same symptom cluster are regarded as the manifestations of the same fault. Each choice of common symptoms and associated matching scheme (i.e., each diagnosis strategy) may give a new set of symptom clusters.

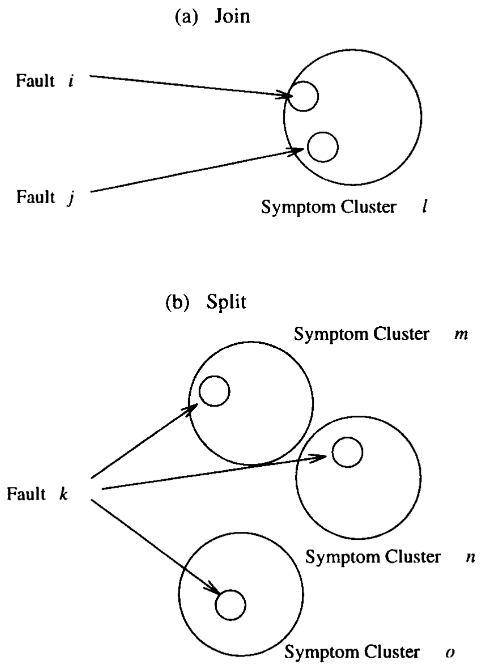

Figure 5: Join and Split

A one-to-one correspondence between fault clusters and symptom clusters would be ideal, but hard to achieve. We considered two general situations to describe the imperfectness of a diagnosis strategy: *join* and *split* (Figure 5). A join means that failures due to more than one fault are grouped into a single symptom cluster. From the perspective of the diagnosis, it represents the possibility of a misdiagnosis. Two scenarios are possible. First, a failure due to a new fault can be declared as a recurrence of a previously reported fault. Second, a recurrence of a fault can be declared as a recurrence of another fault. A split means that failures due to a single fault are divided into multiple symptom clusters. From the perspective of the diagnosis, it represents a repeated diagnosis of the same fault because it means that a recurrence is declared as a first occurrence.

Let's assume that using a particular diagnosis strategy leads to N joins and M splits. Also let J_i be the number of unique faults involved in the i-th join, and S_j be the number of symptom clusters involved in the j-th split. Then, the following measures of efficiency can be defined:

$$\begin{aligned} F_{msdx} &\equiv F_{misdiagnosis,max} \\ &\equiv Maximum\ number\ of\ faults\ misdiagnosed \\ &= \sum_{i=1}^{N} (J_i - 1) \qquad (1) \end{aligned}$$

$$\begin{aligned} F_{rpdx} &\equiv F_{repeated-diagnosis,max} \\ &\equiv Maximum\ number\ of\ repeated\ diagnoses \\ &= \sum_{i=1}^{M} (S_i - 1) \qquad (2) \end{aligned}$$

$$\begin{aligned} S_{crdn} &\equiv S_{correct-diagnosis,min} \\ &\equiv Min.\ number\ of\ recurrences\ diagnosed\ correctly \\ &= Number\ of\ recurrences - F_{rpdx} \qquad (3) \end{aligned}$$

The actual number of misdiagnoses can be smaller than $F_{misdiagnosis,max}$ for the following reasons:

- Overlaps in joins and splits: For example, two faults can generate two symptom clusters as a result of two joins and two splits. In this case, the actual number of misdiagnoses is at most one, not two as calculated from Equation 1.
- Nonoverlap of fault manifestation windows: Even if two faults cause failures with identical symptoms, if one fault appears after the other is completely fixed in the field, there can be no misdiagnosis. In this study, if the last failure due to a fault and the first failure due to another fault occurred more than six months apart in such cases, we assumed that there is no misdiagnosis.

$F_{repeated-diagnosis,max}$ and $S_{correct-diagnosis,min}$ provide a maximum and a minimum respectively because of the first reason listed above for $F_{misdiagnosis,max}$.

Note that partial matching uses a less strict rule than complete matching in building symptom clusters and therefore generates fewer symptom clusters. This means that, when compared with complete matching, partial matching leads to a greater or equal number of joins and a lesser or equal number of splits. Therefore, partial matching can be used to increase the probability of correct diagnosis, at the cost of increasing the probability of misdiagnosis.

4.4 Cost of Misdiagnosis

A question to ask here is: What is the cost of misdiagnosis in an automated diagnosis environment? Consider that two faults (faults A and B) cause failures with identical symptoms. Fault A already caused a failure, and a fix for the fault is available. When fault B causes a failure for the first time, it will be treated as a recurrence of fault A, and the fix for fault A will be recommended by the tool. Then a concern is: What if fault B keeps causing failures? A similar concern exists in the case of an incorrect fix. Consider that a purported fix of fault C fails to fix the fault. When fault C causes another failure with identical symptoms at another site, the tool will declare it as a recurrence of fault C and recommend the incorrect fix. As a result, fault C may keep causing failures.

Both of these situations can be handled by associating each fault in the failure database with the software version information that is supposed to contain a fix for the fault. With this information, when fault B or C causes a failure at a site that installed a fix for the fault, the tool will realize that the failure is due to another fault or due to an incorrect fix and recommend the diagnosis of the problem by human analysts. In the first situation, after a fix for fault B is made, when fault A or B causes a failure, the tool will recommend the installation of fixes for both faults A and B. In both situations, the cost of a misdiagnosis is the time between initial incorrect diagnosis and eventual correct diagnosis. Considering the implementation of a diagnosis strategy as an automatic tool, more emphasis can be put on reducing $F_{misdiagnosis,max}$ than on increasing $S_{correct-diagnosis,min}$.

5 Diagnosis Environment

Figure 6 illustrates the type of automatic diagnosis environment envisioned. The diagnosis tool is connected with many user systems by an on-line alarm system. All previously reported failure symptoms and the associated information such as underlying faults and fixes are stored in a database. On a failure alarm, the tool accesses the system that sent the alarm, extracts the values of the common symptoms (i.e., a stack trace and a symptom string), and compares them with those of previously reported faults in the database. If a match is found in the database, it is declared as a recurrence of the corresponding fault; otherwise, it is declared to be due to a new fault. In the case of recurrence, it also identifies an available fix. After the diagnosis, the database is updated with new failure data. The diagnosis strategy is determined a priori, by off-line evaluation. The tool is built based on the selected diagnosis strategy.

The environment shown in Figure 6 involves connections with many user systems and a database, and cooperation with other software service tools. Such a fully-integrated environment is not up and working at this point, although some individual parts exist. Note that, in such a diagnosis environment, the terms *matching* and *clustering* can be used interchangeably. That is, "found a matching symptom in the database" (see Figure 6) and "clustered together with a symptom in the database" will have identical meanings.

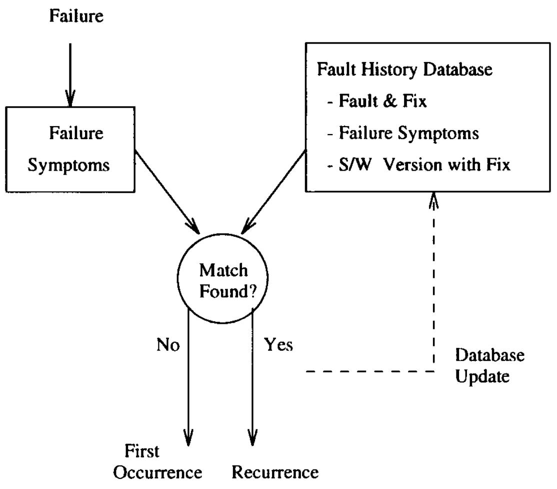

Figure 6: Diagnosis Environment Envisioned

Problem Detection		Fraction (%)
Consistency Checks		81
detection within DB	(72)	
detection outside DB	(9)	
Virtual Memory Protection		14
detection within DB	(13)	
detection outside DB	(1)	
Hang		5

Table 1: Problem Detection Profile (DB)

Common Symptom	#Joins	#Splits	F_{msdx}	S_{crdn}
Stack Trace (78 TPRs, 39 faults)	2	9	2{0}	28

{}: nonoverlap of fault manifestation windows

Table 2: Complete matching of Stack Traces (DB)

6 Evaluation of Diagnosis Strategies Using Field Data

This section evaluates the effectiveness of the proposed diagnosis strategy using the field failure data in two Tandem software products. The thrust of the evaluation is to investigate the range of effectiveness of the proposed diagnosis strategy and its variations. Ideally we would have evaluated the strategies using all failures. We used failures in two products due to time constraint. Given this limit, we selected two products with widely different reputations among Tandem analysts in terms of their quality, hoping that an evaluation using failures in the two products would give us a range of effectiveness.

6.1 Evaluation Using Failures in DB

Table 1 shows a breakdown of the 152 failures in DB, based on how the problems were detected. The numbers inside parentheses represent a further subdivision inside a class. The failures occurred due to 55 unique faults. The table shows that about 85% percent of the problems were detected while executing the DB code and 72% of the problems were detected by the consistency checks in DB. Only the 130 failures detected while executing the DB code were considered because these failures and the failures detected outside DB naturally have different code-oriented symptoms.

6.1.1 Matching Stack Traces

Although the stack trace exists in all failures, not all TPRs contained stack traces. This usually happened when there were many recurrences due to a single fault. In TPRs reporting later occurrences, analysts sometimes just left pointers to the TPRs that analyzed previous occurrences, rather than describing the detailed symptoms. Our experience shows that this is more likely to happen when later occurrences share the same symptoms with early occurrences. Out of 130 TPRs, 78 contained stack traces. These failures occurred due to 39 unique faults. Note that the recurrence rate in the data set became much lower than its actual value. The average number of procedures in a stack trace (i.e., the average length of a stack trace) was 5.7.

Table 2 shows the effectiveness of the diagnosis when symptom clusters were constructed by the complete matching of stack traces. The table shows that, with the complete matching of stack traces, at least 72% (28 out of 39) of the recurrences could have been identified correctly. (We think that this percentage would be higher if all TPRs contained stack traces.) The cost of using such a diagnosis is the misdiagnosis of at most two faults. In each join, two different faults affected the processor state in the same manner: a table entry was missing due to the faults. The problems were detected when attempting to locate a nonexisting entry. They were detected at an identical location while executing the same function. So the joins were unavoidable with code-oriented symptoms. The data showed that, in each join, the two faults had nonoverlapping manifestation windows. Therefore the actual number of misdiagnoses was zero, which is shown inside a pair of braces in Table 2. Including the halt code in constructing symptom clusters had negligible effect: it decreased $S_{correct-diagnosis,max}$ by one. This was because many failures were detected by consistency checks and had identical halt codes.

Partial matching can reduce the number of splits at the cost of increasing the number of joins. We investigated the patterns of stack traces in the nine splits in Table 2. The splits were mainly due to different program paths to reach the same errors. As a result, different stack traces causing the splits often had an identical procedure at the TOS. Two common patterns of differences in the splits were: 1) stack traces were much different and 2) stack traces were the same except for minor differences in the middle. Based on

Heuristics	#Joins	#Splits
Differ-by-one	7(+5)	5(-4)
Differ-by-one & the same procedure at the TOS	3(+1)	5(-4)
Contain-the-other	10(+8)	9(0)
Contain-the-other & the same procedure at the TOS	3(+1)	9(0)

Table 3: Partial Matching of Stack Traces (DB)

these patterns, the following heuristics were considered for the partial matching of stack traces:

1. If two stack traces with the same length differ from each other by no more than one procedure, group them into the same symptom cluster. This heuristic is called *differ-by-one*. Note that repeated applications of this heuristic can cluster together stack traces that differ by more than one procedure.
2. Apply the differ-by-one heuristic only if the procedures at the TOS are the same.
3. If one stack trace includes all procedures in the other without regard to their order, group them into the same symptom cluster. This heuristic is called *contain-the-other*.
4. Apply the contain-the-other heuristic only if the procedures at the TOS are the same.

Table 3 shows the results of the partial matching of stack traces. The numbers inside the parentheses indicate the differences from the numbers when complete matching is used (Table 2). The table shows that the procedure at the TOS is a useful common symptom. Including it prevented the increase in the number of joins appreciably. With the "differ-by-one and same procedure at the TOS" heuristic, at least 87% (34 out of 39) of the recurrences could have been identified correctly. The number of joins increased by one, but the actual number of misdiagnoses was still zero due to the nonoverlap of fault manifestation windows. The contain-the-other heuristic was not effective.

6.1.2 Matching Symptom Strings

The results in the previous subsection indicated that the code location at which a problem is detected can be a useful common symptom. As described in Section 4.1, the DB symptom string uniquely identifies the code location of problem detection, regardless of software version. In fact, DB developers have been using this information as an aid for software failure diagnosis [Tandem92]. All 110 TPRs reporting failures detected by the DB consistency checks (see Table 1) contained symptom strings. These TPRs were due to 39 unique faults.

Table 4 shows the effectiveness of the diagnosis when symptom clusters were formed using symptom strings. Since a symptom string is a single piece of information, only complete matching is possible. Table 4

Common Symptom	#Joins	#Splits	F_{msdx}	S_{crdn}
Symptom string (110 TPRs, 39 faults)	5	4	8{1}	67

{}: nonoverlap of fault manifestation windows

Table 4: Matching Symptom Strings (DB)

Common Symptom	#Joins	#Splits	F_{msdx}	S_{crdn}
Procedure at the TOS (130 TPRs, 43 faults)	5	5	13	82
Procedure at the TOS & offset (110 TPRs, 40 faults)	3	11	4	54

Table 5: Matching Variations of the Symptom String (DB)

shows that at least 94% (67 out of 71) of the recurrences could have been identified correctly, at the cost of misdiagnosis of less than eight faults. The data showed that the maximum number of misdiagnoses was actually one, considering the nonoverlap in fault manifestation windows.

The hypothesis that matching symptom strings was as effective as the complete matching of stack traces in terms of successful diagnosis was rejected, indicating that matching symptom strings was more effective in terms of successful diagnosis for the measured period in DB (see Table 2 and Table 4). The hypothesis was tested using the binomial test at the 5% significance level, by treating the diagnosis of recurrences as Bernoulli trials.[3] The hypothesis that matching symptom strings was as effective as the complete matching of stack traces in terms of misdiagnosis was not rejected by the same test at the same level. A caution for the observations is that the two tables used for the comparison were generated using data sets with different recurrence rates, because analysts did not always record stack traces in TPRs.

A limitation in using symptom strings is that the symptom string exists only when problems are detected by consistency checks. (This is discussed further in Section 6.2.) Note that a stack trace always exists, even in failures due to nonsoftware faults.

We also used two variations of the symptom string to construct symptom clusters: 1) procedure at the TOS, and 2) procedure at the TOS and associated offset. These symptoms always exist. Table 5 shows the results. Although the three sets of TPRs used to generate Table 4 and Table 5 were different, we can make several observations. Compared to the use of symptom strings, using the procedure at the TOS increased $F_{misdiagnosis,max}$ because some problems due

[3]Whether a diagnosis strategy is better than another can be answered by conducting a hypothesis test. We addressed such issues lightly because we used failures in only two products.

Problem Detection	Fraction (%)	
Consistency Checks		51
detection within DC	(33)	
detection outside DC	(19)	
Virtual Memory Protection		46
detection within DC	(31)	
detection outside DC	(15)	
Hang		3

Table 6: Problem Detection Profile (DC)

Common Symptom	#Joins	#Splits	F_{msdx}	S_{crdn}
Stack Trace	13	11	21	77
Stack Trace & halt code	10	11	16{6}	77

{}:nonoverlap of fault manifestation windows and using subprocedure traces

Table 7: Complete matching of Stack Traces (DC – 166 TPRs due to 59 faults)

to different faults were detected at different locations in the same procedure. Using the procedure at the TOS and associated offset increased the number of splits appreciably because the same code location had different offset values in different software versions. One interesting observation here is that the number of joins has decreased. This was because of the nonoverlap of fault manifestation windows between different faults in a join. Due to the code changes between the windows, although they were detected at an identical location, they showed different offsets.

6.2 Evaluation Using Failures in DC

Table 6 shows a breakdown of 258 failures caused by 72 unique faults in DC. Compared with the problem detection in DB (Table 1), two observations can be made. First, the percentage of the problems detected by consistency checks was lower. Second, a greater percentage of the problems was detected while executing non-DC code. These observations corroborate with the analysts' suspicion that this product is less robust. The evaluation was conducted using 166 failures that were detected while executing the DC code and that contained stack traces. These failures occurred due to 59 unique faults. The average number of procedures in a stack trace was 3.6.

6.2.1 Matching Stack Traces

Table 7 shows the effectiveness of the diagnosis when the complete matching of stack traces was used. Using halt codes along with stack traces reduced the number of joins while not increasing the number of splits. This was because the percentage of the problems detected by consistency checks was lower. So, the halt code, which represents how problems were detected, became a useful common symptom. In the subsequent analysis, failures with different halt codes were not grouped into the same symptom cluster.

Reason for Split	#Splits	F_{rpdx}
Data corruption	4	23
Different calling sequence	6	6
Data dependence	1	1

Table 8: Breakdown of Splits (DC)

In four of the ten joins in Table 7, problems caused by different faults were detected at an identical code location while executing the same software function. With just code-oriented symptoms, resolution of the joins was not possible. In the remaining six joins, problems were detected at different locations in the same procedure. These joins were mainly due to big procedures that detected errors due to different faults. The existence of big procedures is attributed to the language's support of subprocedures, callable only within a procedure. The data showed that, with the use of subprocedure traces within the procedure at the TOS, $F_{misdiagnosis,max}$ is reduced to 8, without affecting $S_{correct-diagnosis,min}$. This suggests that the effectiveness of the diagnosis may be improved by reasonably sizing procedures.

The maximum number of misdiagnoses was reduced again to 6, considering the nonoverlap of fault manifestation windows. With the complete matching of stack traces, halt codes, and subprocedure traces in the procedure at the TOS, at least 72% (77 out of 107) of the recurrences could have been identified correctly, at the cost of misdiagnosis of at most six faults. There was no significant difference in the performance of the complete matching of stack traces in the two products in terms of successful diagnosis, but the complete matching of stack traces was more effective in DB than in DC in terms of misdiagnosis (see Table 2 and the second row of Table 7). These observations were obtained using the binomial test at the 5% significance level. Again, a caution here is that the recurrence rate in the data set used for DB was lower.

Table 8 shows a classification of the 11 splits in Table 7, based on their major reasons for the splits. *Data corruption* means that a fault caused corruption in a shared data area. If such a corruption occurs, errors can be detected while executing many software functions, which is why a fault causes different stack traces. There were two complex faults (i.e., two splits) which caused corruption in shared data. It took a while to diagnose the problems and, in the meantime, the faults caused failures with 23 different stack traces. That is, the two faults accounted for 21 $F_{repeated-diagnosis,max}$.

Different calling sequence means that the differences in stack traces are attributed to different program paths to reach and detect the same errors. *Data dependence* means that depending on the actual values of errors and the machine state, a problem is detected at different (but typically close to each other) code locations. In the actual case, the difference in stack traces was one extra procedure at the TOS. This type

Heuristics[2]	#Joins	#Splits	F_{msdx}	S_{crdn}
Differ-by-one	(+4)	(0)	(+12)	(+7)
Differ-by-one & the same proc. at the TOS	(+1)	(0)	(+3)	(+6)
Contain-the-other	(+2)	(-1)	(+5)	(+6)
Contain-the-other & the same proc. at the TOS	(0)	(0)	(+3)	(+4)
Extra-proc-at-TOS	(+3)	(-2)	(+3)	(+2)

Table 9: Partial Matching of Stack Traces (DC)

of differences in stack traces could also be observed in some data corruption cases. For example, when a software function accesses a corrupt data region, depending on the actual values of errors and the machine state, a problem could be detected after an additional procedure call, after a return to the previous procedure, or within that procedure. With this observation we added the fifth heuristic for the partial matching of stack traces:

5. Given two stack traces, if one is longer than the other by one and the difference is an additional procedure at the TOS, group them into the same symptom cluster. This heuristic is called *extra-proc-at-TOS*.

Table 9 shows the effectiveness of the diagnosis when the partial matching of stack traces was used. The numbers inside the parentheses indicate the differences from the numbers when complete matching is used (the second row of Table 7). Subprocedure traces were not used here. All heuristics increased $S_{correct-diagnosis,min}$, but not drastically, indicating that the partial matching hueristics could not completely capture the randomness in failure symptoms caused by data corruption. This suggests that the error containment capability of software can be a factor that affects the effectiveness of the diagnosis. The increases in $F_{misdiagnosis,max}$ were mainly due to short stack traces (with length of three or less) that easily caused joins when partial matching was used. Table 9 shows that the procedure at the TOS helped to suppress the increase in the number of joins in DC, too.

6.2.2 Matching Symptom Strings

The product DC did not provide the symptom string. Although not all TPRs recorded the failed software version, it was possible to determine whether two problems were detected at the same code location, using the information in TPRs (stack traces, offsets, halt codes, and textual descriptions by analysts) and the actual code. So, in the following evaluation, it was assumed that the symptom string existed in all failures. We formed symptom clusters using the following three

[2]To avoid an excessive increase in the number of joins, the differ-by-one hueristic was not applied to the stack traces of length one, and the contain-the-other hueristic was not applied to the stack traces of length one or two.

Common Symptom	#Joins	#Splits	F_{msdx}	S_{crdn}
Proc. at the TOS	15	10	25	89
Symptom string	8	12	11{6}	83
Symptom string & stack trace	6	14	6	70

{}: nonoverlap of fault manifestation windows

Table 10: Matching Variations of the Symptom String (DC – 166 TPRs due to 59 faults)

symptoms, listed in the increasing order of strictness:

1. Procedure at the TOS.
2. Symptom string.
3. Symptom string and stack trace.

Table 10 shows that, by matching the symptom string and halt code, at least 78% (83/107) of the recurrences could have been identified correctly, at the cost of the misdiagnosis of at most six faults. For the measured period, there was no significant difference between the complete matching of stack traces and the matching of symptom strings in their performance in DC (see the second rows of Table 7 and Table 10). Comparing Table 4 and the second row of Table 10, the matching of symptom strings was more effective in DB than in DC in terms of successful diagnosis, but it showed similar performance in the two products in terms of misdiagnosis. These observations were again obtained using the binomial test at the 5% significance level.

6.2.3 Machine Code Symptom String

Now the question is: How does an automatic tool compare the two code locations of problem detection in DC? It can be encouraged to implement the DB-style symptom string in all products. But the percentage of failures that have the symptom string (i.e., the percentage of failures that are detected by consistency checks) seems to depend on the quality of software. Besides, the value of the percentage can be estimated after the software is released to the field.

Here we propose the use of a *machine code symptom string*. It is defined as the machine instructions in the binary form, before and after the code location of problem detection. Just like a stack trace, it always exists. (There can be rare cases in which we cannot compare machine code symptom strings if two detection locations are at different edges of two memory pages and the connecting pages are not available.) A possible strategy is to use the DB-style symptom string if available and otherwise to use the machine code symptom string.

7 Conclusions

In this paper, we presented an approach to automatically identify recurrent software failures using symptoms, in environments where many customers run the same software. The approach is based on our observations that about 72% of reported field software failures in Tandem systems are recurrences and that failures due to the same fault often share common symptoms. Specifically, we proposed the comparison of stack traces and symptom strings as a strategy for identifying recurrences. We applied this strategy using failures in two Tandem system software products and compared the results obtained with actual Tandem diagnosis and repair logs by analysts.

The results of the comparison showed that between 75% and 95% of recurrences can be identified successfully by matching stack traces and symptom strings. Less than 10% of faults are misdiagnosed. These results indicate that automatic identification of recurrences using symptoms is possible. In an automated diagnosis environment, the cost of a misdiagnosis is the time between initial incorrect diagnosis and eventual correct diagnosis. The benefits of developing and using a tool that implements such a diagnosis strategy are 1) saving the wasted human effort of reporting and diagnosing the same problem repeatedly and 2) identifying an available fix or a workaround rapidly. The results of the evaluation suggested that the error containment capability of the software can be a factor that determines the effectiveness of the approach. Proper sizing of procedures can also be a factor when using stack traces.

We would like to point out several areas of future work. First, more diagnosis strategies need to be investigated. For example, the use of data-oriented symptoms needs to be investigated. Second, it is necessary to use failures from more software products for the evaluation because, in real environments, many products run together and the effects of faults can cross the boundaries between the products. Failures due to nonsoftware faults also need to be included, because whether a failure is due to a software fault is often unclear. Third, numerical results reported in this paper are specific to the measurements. However, the two measured products consist of many small procedures and are written in a high-level language, which is common in many system software products around today. Our experience shows that there are no special requirements for the software to satisfy, for the approach to be effective. Still, further experiments are necessary to determine how well the numbers will project to other system software products. Also, it will be interesting to investigate the effectiveness of the approach for application software products.

Acknowledgements

We thank Tandem Computers Incorporated, in particular Gil Pitt, for their assistance in conducting this study. We also thank Ram Chillarege and Timothy Tsai for their comments on the paper. This research was supported in part by Tandem, in part by the Office of Naval Research under Grant N00014-91-J-1116, and in part by NASA grant NAG-1-613. The content of this paper does not necessarily reflect the position or policy of the government or Tandem, and no endorsement should be inferred.

References

[Adams84] E. N. Adams, "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Development*, Vol. 28, No. 1, Jan. 1984.

[Chillarege92] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal Defect Classification–A Concept for In-Process Measurements," *IEEE Trans. Software Engineering*, Vol. 18, No. 11, Nov. 1992, pp. 943-956.

[Chillarege93] R. Chillarege, B. Ray, A. Garrigan, and D. Ruth, "The Recreate Problem in Software Failures," *Proc. Fourth Int. Symp. Software Reliability Engineering*, 1993.

[Gray90] J. Gray, "A Census of Tandem System Availability between 1985 and 1990," *IEEE Trans. Reliability*, Vol. 39, No. 4, Oct. 1990, pp. 409-418.

[Iyer90] R. K. Iyer, L. T. Young, and Iyer, P. V., "Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data," *IEEE Trans. Computer*, Vol. 39, No. 4, Apr. 1990.

[Latham86] B. Latham and M. W. Swartwout, "CD_x–Crash Diagnostician for VMS," *Expert Systems and Knowledge Engineering*, T. Bernold(editor), Elsevier Science Publishers B. V. (North-Holland), 1986.

[Lee93a] I. Lee and R. K. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System," *Proc. 23rd Int. Symp. Fault-Tolerant Computing*, Toulouse, France, 1993, pp. 20-29.

[Lee93b] I. Lee, D. Tang, R. K. Iyer, and M.-C. Hsueh, "Measurement-Based Evaluation of Operating System Fault Tolerance," *IEEE Trans. Reliability*, Vol. 42, No. 2, June 1993, pp. 238-249.

[Levendel] Y. Levendel, Private communications.

[Lin90] T.-T. Lin and D. P. Siewiorek, "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis," *IEEE Trans. Reliability*, Vol. 39, No. 4,Oct. 1990, pp. 419-432.

[Maxion93] R. A. Maxion and R. T. Olszewski, "Detection and Discrimination of Injected Network Faults," *Proc. 23rd Int. Symp. Fault-Tolerant Computing*, Toulouse, France, 1993, pp. 198-207.

[Maxion85] R. A. Maxion and D. P. Siewiorek, "Symptom Based Diagnosis," *Int. Conf. Computer Design*, 1985, pp. 294-297.

[Sullivan91] M. S. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability–A Study of Field Failures in Operating Systems," *Proc. 21st Int. Symp. Fault-Tolerant Computing*, June 1991, pp. 2-9.

[Tandem92] *Smart Dumps External Specification*, Tandem Computers Inc., 1992.

[Tsao83] M. M. Tsao, *Trend Analysis and Fault Prediction*, Ph. D. Dissertation, Department of Electrical Engineering, Carnegie-Mellon University, May 1983.