Journal of Network and Systems Management, Vol. 13, No. 2, June 2005 (© 2005)
DOI: 10.1007/s10922-005-4443-8

Automated Problem Determination Using Call-Stack Matching

Mark Brodie,[1,3] Sheng Ma,[1] Leonid Rachevsky,[1] and Jon Champlin[2]

We present an architecture and algorithms for performing automated software problem determination using call-stack matching. In an environment where software is used by a large user community, the same problem may re-occur many times. We show that this can be detected by matching the program call-stack against a historical database of call-stacks, so that as soon as the problem has been resolved once, future cases of the same or similar problems can be automatically resolved. This would greatly reduce the number of cases that need to be dealt with by human support analysts. We also show how a call-stack matching algorithm can be automatically learned from a small sample of call-stacks labeled by human analysts, and examine the performance of this learning algorithm on two different data sets.

KEY WORDS: Diagnosis; self-managing; machine learning; case database; common problems; help-desk support.

1. INTRODUCTION

Problem determination is a crucial component of systems management. The requirements involved, including help-desk staffing and training, bug fixes and upgrades, and the importance of maintaining customer satisfaction, are a major and ever-increasing cost for any modern enterprise. Tools that can automate any part of the problem determination process constitute an important step in the direction of self-managing systems that may reduce this cost burden. Indeed, before any type of self-healing or automated error correction can be done, a necessary first step is for the system to be able to perform problem determination. If commonly occurring problems can be automatically identified and resolved, this would constitute

[1]IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532.
[2]IBM Lotus Notes External Support, Route 100, Somers, NY 10589.
[3]To whom correspondence should be addressed at IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: mbrodie@us.ibm.com

an essential part of any "automated control loop" implemented in a self-managing system.

Problem determination is difficult and expensive for a number of reasons. The problem symptoms must be manually described by the end-user to the help-desk. Information needed by the help-desk analyst cannot always be obtained from the user, either because the user does not know the information or, even worse, the user may provide incorrect information. Multiple visits by one or more help-desk analysts to the customer location may be required to completely resolve the problem. Different analysts cannot easily take advantage of each other's knowledge, so that when the same problem is experienced multiple times (either repeatedly by the same user or by different users) it may have to be repeatedly resolved. Since popular software may have millions of users, the cost of having to resolve a problem that, unknown to the analyst, has already been resolved by someone else, may be considerable.

In this work we address these issues by describing a self-managing system which partially automates the process of software problem determination. The components of this system:

- automatically collect the problem symptoms from the customer location when a problem occurs.
- match the problem symptoms against a historical database of previously resolved cases.
- generate a proposed resolution from the previously resolved case(s) if a very good match is found.
- add resolved cases to the database, increasing the likelihood of future matches when a problem occurs.

We focus on using program call-stacks as symptoms for the purposes of automated problem diagnosis. The main idea is that although a bug may be exercised in different ways, the resulting call-stacks are likely to be similar to one another. Another advantage of call-stacks is that they can be captured automatically without requiring lengthy questioning of the user in order to determine the problem symptoms.

The main contribution of this work is to show that a simple matching procedure performs well in detecting the most common problems. The data used is collected from real customer environments; the software tested is the widely used Lotus Notes. Although the call-stack matching method has only been applied to problem determination on the client side so far, there is no reason why it cannot be used on the server side as well.

We also show that the matching procedure can be automatically learned from a small set of training data in which different stacks corresponding to the same problem have been labeled by human experts. This work shows that, for cases

where a program has a large community of users, examining program call-stacks often allows the nature of the problem to be determined. Using the call-stacks can be viewed as a first step towards a more sophisticated self-managing system that takes advantage of other forms of information as well.

The problem that we address here is fundamentally different from, and complementary to, software debugging. The differences lie in both the nature of the problems and also the methods of resolution. Automated matching determines whether a problem is a known problem or a new problem, in order to deal with the large number of redundant customer problem reports. This is most useful once the software is already fielded, since it requires no software skills on the part of the user. However, it can also apply in the testing phase when multiple testers are testing the same software and may rediscover the same problems. Software debugging relies on specialized software tools and skills, and is mostly relevant in development and testing. The resolution required is usually some low-level code change, whereas problems identified by automated matching can usually be solved by simply using an already existing solution to the old problem (e.g. a patch).

In general, not all problems can be resolved in this way. New types of problems that do not match anything previously experienced will continue to happen. These problems will need to be resolved by human support staff in the traditional way, possibly including on-site customer visits and debugging of the source code. Also, certain common problems, such as performance degradations, may not be amenable to call-stack matching. These types of problems are usually addressed by decision-tree methods, where the help-desk support staff ask a sequence of questions to elicit more information, the selection of later questions depending on the answers to earlier questions. It is possible that matching on other types of symptoms may allow for automatic diagnosis in some of these cases as well.

However, automated call-stack matching will be able to resolve some, and possibly many, of the cases that human analysts currently deal with. This is because for most, or perhaps even all, software, a high proportion of problem reports are usually generated by a small fraction of bugs—those which are exercised most frequently by the user community. If automated matching can detect these problems, the number of cases that need to be resolved by human experts will be greatly reduced.

The outline of the remainder of this paper is as follows. Section 2 describes the overall system architecture. Section 3 discusses call-stacks and how they are captured. Section 4 shows how a simple matching procedure can be used to group a collection of call-stacks into clusters of similar stacks, which can then be presented to a human expert for further analysis and classification. We illustrate this method on a data set of crash stacks obtained from an actual customer environment, and confirm that a large percentage of crashes are indeed due to a small percentage of problems which repeat themselves. Section 5 shows how a matching procedure

can be efficiently learned using a "nearest neighbor" mechanism from a small set of training data provided by human experts. Section 6 discusses related work and Section 7 presents some conclusions and prospects for future work.

2. ARCHITECTURE

Figure 1 illustrates the overall system architecture. When a problem occurs in the customer environment, a data collection agent gathers information, such as the problem symptoms and environmental information, which may be useful for diagnosis. In this paper, we will focus on the program call-stack, but more generally a wide variety of information, such as which applications are running, the version of the operating system, and so on, could be collected.

The information is then sent to the self-managing component which begins by logging it for record-keeping and problem trend analysis. The data, which may come from a wide variety of platforms, needs to be converted into a standard format for further analysis. The problem symptoms are then matched against a historical case database. If a match is found, the corresponding resolution is generated from the case database and sent to the customer's system administrator as a recommended solution. If a match is not found, a ticket is opened in the traditional trouble-ticketing system so that the problem can be addressed by a human analyst. By adding resolved cases to the database, the matching performance will automatically improve as the system's experience accumulates.

There are many technical challenges in implementing the type of system outlined earlier. In this paper, we address only some of these, in order to test

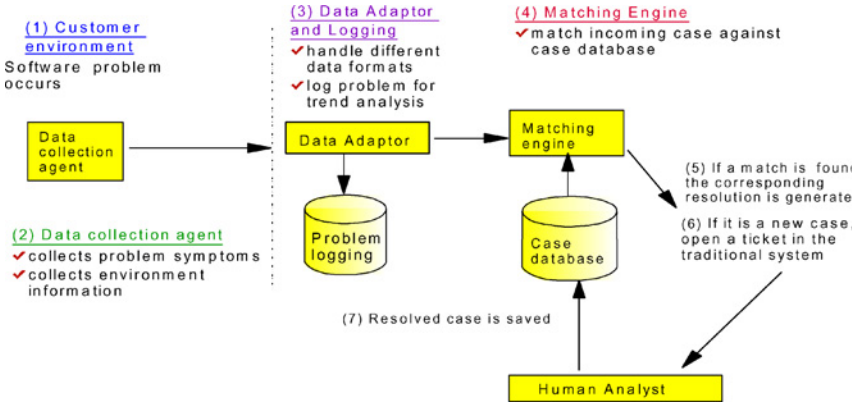

Fig. 1. System architecture.

the viability of this sort of self-managing system. We describe how to collect call-stacks from the customer environment, how to use simple call-stack matching to group the call-stacks into clusters to present to human analysts for classification by problem type, and how to automatically learn an efficient call-stack matching procedure from a small set of call-stacks which have been classified by human analysts into different problem types.

3. CALL-STACKS

The call-stack is the sequence of function or method calls invoked during the program's execution, with the most recent call on the top. Each line of the stack provides information such as the function name, arguments or parameters passed to the function, and the line number of the calling routine. We now illustrate the use of call-stacks in diagnosis, describe how call-stacks can be captured when a program crashes, and discuss the issue of whether call-stack matching should be done locally or centrally.

3.1. Call-Stacks and Program Problems

Call-stack information is potentially very informative about program problems. To illustrate this, consider a very simple program, shown later (Fig. 2), that sets a null pointer and sometime later accesses this pointer, causing a crash. The location where the null pointer is set can be reached along two different paths through the program. Which path is taken may depend on inputs to the program or other uncontrollable factors. The location where the crash takes place may be far removed from the location of the bug.

Later we show two call-stacks recovered when running this program (Fig. 3). The stack on the left shows an execution along path1 through the program and the stack on the right an execution through path2. Both stacks share the part of the stack from the bug location ("set_null_pointer") to the crash location. This common part of both stacks can be regarded as diagnostic for this particular problem.

Of course, actual software is much more complex, with many different bugs and crash points and multiple execution paths both from the beginning of the

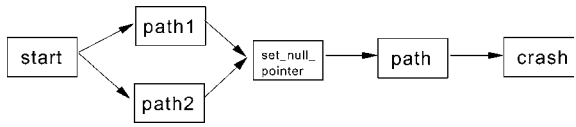

Fig. 2. A simple buggy program.

```
java.lang.NullPointerException
        at crash.error(crash.java:91)
        at crash.pass_pointer2(crash.java:79)
        at crash.pass_pointer1(crash.java:73)
        at crash.set_null_pointer(crash.java:68)
        at crash.path1_2(crash.java:32)
        at crash.path1_1(crash.java:26)
        at crash.path1(crash.java:21)
        at crash.main(crash.java:118)
```

```
java.lang.NullPointerException
        at crash.error(crash.java:91)
        at crash.pass_pointer2(crash.java:79)
        at crash.pass_pointer1(crash.java:73)
        at crash.set_null_pointer(crash.java:68)
        at crash.path2_3(crash.java:53)
        at crash.path2_2(crash.java:48)
        at crash.path2_1(crash.java:43)
        at crash.path2(crash.java:38)
        at crash.main(crash.java:119)
```

Fig. 3. Two call-stacks.

program to each bug location and from each bug location to each crash location. However, if the code is well designed, each bug will tend to generate call-stacks that are similar to one another and quite different from call-stacks caused by other bugs. Thus, examining call-stacks and determining which function calls they have in common may serve as a useful guide for problem diagnosis.

Certain types of problems are more amenable to a call-stack matching approach than others. Performance degradations, caused by environmental conditions interfering with the application (e.g. load spikes, log file size exhausted, etc.), are usually addressed by having the help-desk support staff follow a "script" (essentially a flowchart or decision tree), where they ask the user a sequence of questions to elicit more information. It is not clear whether call-stacks can provide useful information in these cases—here we focus on their use in diagnosing problems with the application itself.

3.2. Capturing Call-Stacks

The techniques for gathering software execution information can be divided into two classes: those that modify target applications (active profiling), and those that do not (passive profiling).

Active profiling methods are based on source code or object code instrumentation. In source code instrumentation, profiling or log routines must be defined and placed in strategic locations in the code, either manually or by means of code-coverage tools. Tools for doing this include BoundsChecker [1] and True Time [2]. In object code instrumentation the binary code is patched with a fixed set of instructions for basic block counting, memory checking, etc. No recompilation is required, since the tools work directly on program binaries. Examples include VTune Performance Analyzer [3] and IBM's PurifyPlus [4].

Source code instrumentation is obvious and simple; however, it may cause a substantial increase in the size of the executable and decrease performance. Binary patching tools need no source code for gathering execution information, but they tend to be very complicated and error-prone [5, 6].

Passive profiling works by 'polling' the application for execution information at regular intervals. It does not alter the structure of the application and therefore is highly reusable—one data-gathering system can be applied to multiple programs. The major disadvantage of passive profiling is the problem of determining the sampling rate. If the sampling rate is too high, the system spends more time in the interrupt service routine than in the application. On the other hand, significant data can be lost if the sampling frequency is too low. Some applications and interfaces that support the passive profiling paradigm include HPROF [7], xdProf [8], and the ICorProfilerCallback interface for .NET applications [9].

We now briefly describe a passive-profiling-like approach for collecting call-stacks from Java applications. The system takes advantage of the Java Virtual Machine Debug Interface (JVMDI) [10]. When the target Java application starts, the JVMDI agent is loaded into the java.exe process using a command line option. The JVMDI client can be notified of interesting occurrences through events—in our case, we subscribe to Java exceptions only. Each time the Java application throws an exception the agent retrieves the information about the class name, failed method, line number, variable names and values. Then it sends this information to the data-collection server through a TCP(IP communication channel (the socket connection is established during the JVMDI agent initialization), as shown in Fig. 4.

The captured call-stacks are shown in our "Stack Trace Viewer." A line number of —1 reflects that the source code is unavailable (Fig. 5).

By responding only to exceptions, this method avoids the problem of having to set the sampling frequency. However, it may result in some slowdown of the application due to increased overhead. It captures not only the method name and line number but also the variable names and values, thereby providing more information than the Java Virtual Machine Profiler Interface (JVMPI), as used in xdProf [8], can provide. Although we will use only the function names to match call-stacks later, potentially, this additional information could also be used to improve call-stack matching.

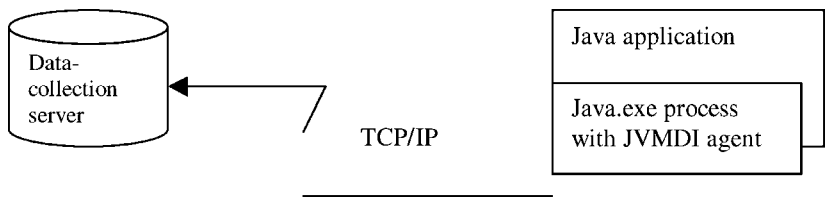

Fig. 4. Collecting Java call-stacks.

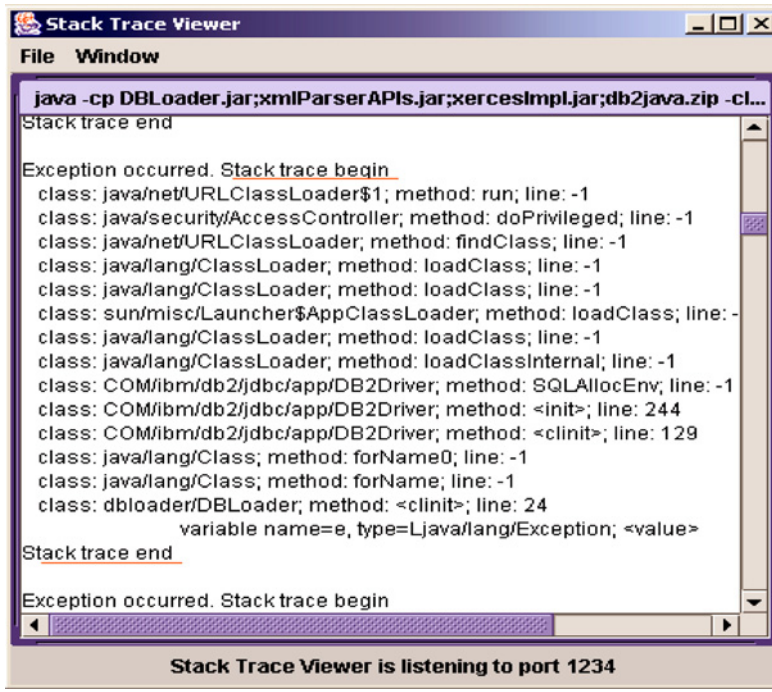

Fig. 5. Captured call-stacks.

3.3. Local or Central Matching

Once the call-stack has been captured at the customer site, the question arises whether matching should be done locally, using only the problems experienced by that same customer, or centrally, using problems experienced by all customers. The advantages of local matching are that it can be done quickly and requires minimal additional infrastructure. A problem determination service residing on a central server requires much greater overhead and must also handle issues such as the confidentiality and privacy of customer data. However, the benefits obtained include a higher probability of a match being found because other customers may have experienced the same problem even if it is a new problem for this particular customer, and also the ability to prevent the problem from occurring at other customers by distributing a fix to all customers.

Whether the benefits of a central server are worth the costs is really an empirical question which may differ from case to case. A hybrid system is also possible—perform initial matching locally at the customer site and only if a match is not found refer the problem to the central server.

4. MATCHING CALL-STACKS

Given different call-stacks obtained from program crashes, how can we determine whether or not the crashes were due to the same underlying cause? A method is needed to compute the "similarity" of call-stacks with one another. A simple way to match two call-stacks is by measuring the length of the longest sequence of consecutive function names that they have in common, as a proportion of the stack lengths. Dividing by the stack length "normalizes" all similarity values to between 0 and 1, so they can be easily compared. This is needed in the following type of situation: if two stacks share a sequence of 10 calls, say, then if both stacks consist only of those 10 calls, similarity is very high, but if the stacks contain many other different calls as well then similarity is low.

If the stacks are completely identical, their match-score will be 100%. If both stacks are of length n, and the longest sequence of common functions has length m, then their simple match-score would be m/n. Finally, if one stack is of length $n1$ and the other of length $n2$, average stack length is used and the match-score would be $m/((n1 + n2)/2)$. Figure 6 shows some simple examples.

This simple method can be improved in a number of ways. For example, we give higher weight to functions occurring nearer the top of the stack, since these are likely to be nearer the location of the problem. Also, functions with slightly different names, for example Alloc and Realloc, may be regarded as "the same" for purposes of matching, or one could consider the "edit distance" (also called the Levenstein distance) between the function names, which counts the number (or cost) of changes that need to be made to transform one string into another. Instead of looking for just the longest matching subsequence, one could consider all matching subsequences, and so on.

4.1. Experiment

In our initial experiment we have a collection of call-stacks whose causes are unknown. We use simple matching to explore how often the identical stack

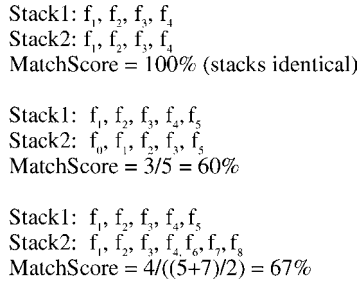

Fig. 6. Simple call-stack matching.

occurs, in order to confirm that most of the software crashes are indeed due to a small number of underlying problems. This also allows the stacks to be collected into groups of identical stacks which can then be presented to a human analyst for further analysis and classification.

The data consists of about 100 call-stacks collected by Lotus Notes' Automatic Data Collection (ADC) mechanism at customer locations. When a Notes client crashes, NSD (Notes System Diagnostics) is called automatically to collect diagnostic data (which includes the call stacks of all threads along with other information like OS version, patch levels, etc.). This is saved to a file on disk. When the client is restarted, it detects that it is restarting after a crash and automatically calls ADC to find the collected NSD output. It parses through it to find the call-stack of the thread that crashed. It then normalizes the stack across platforms (pulls out only the function name, de-mangles C++ mangled names, reverses the call stack on some platforms which read the stack from top to bottom instead of bottom to top, etc.). The normalized stack is sent via email to a repository database where all call-stacks are accumulated and provide the basis for matching.

In our experiment, the match-score between every pair of stacks is computed and all stacks with a 100% match-score are grouped together. This results in about 30 different groups of call-stacks, which are then ordered by the number of stacks they contain. The cumulative frequency of the number of stacks in each group is shown in Fig. 7. The largest group contains 31 identical stacks, the next largest contains 13 stacks, which are identical to one another but different from the stacks in the first group (raising the cumulative frequency to 44), and so on. Eventually, we reach the groups that contain stacks that occur only once.

If we assume that each group of identical stacks corresponds to the same underlying cause, the results clearly show that a large proportion of crashes are caused by a small number of problems. The two most common problems account for about 40% of the crashes, and fewer than half the total problems are responsible for 80% of the crashes. In fact, these estimates are probably conservative because

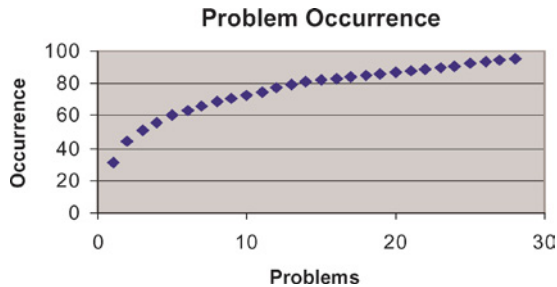

Fig. 7. A small number of problems account for most of the crashes.

stacks in different groups, although different, may in fact sometimes represent the same problem. Nonetheless, this demonstrates the potential of even a very simple method that simply looks for identical stacks. This will allow a large proportion of crashes to be automatically identified, and only a small number of remaining cases will need to be resolved by human analysts.

5. MACHINE LEARNING

5.1. Motivation

A major weakness with the method of matching only identical stacks is that different call-stacks often correspond to the same underlying problem. This is likely to happen because the same bug may be exercised in many different ways. Therefore, we need to determine the part of the call-stack that is most relevant to diagnosing the problem. We can develop simple heuristics for this—for example, functions nearer the top of the stack are more likely to be involved in the cause of the problem than functions near the bottom of the stack. Furthermore, we may be able to elicit knowledge from human analysts describing how they use the call-stack information in their problem-diagnosis methodology, and we could attempt to codify this knowledge into a set of rules that can be implemented automatically by a program. However, such "expert" systems are well known to be very difficult and expensive to develop, modify, and maintain. For example, in the late 1980s DEC spent up to $2 million a year to update its expert system [11].

An alternative approach is to use human analysts to provide a small sample of labeled call-stacks and then use machine learning methods that generalize from this sample to automatically produce a matching procedure that will be able to predict the label, or classification, of other call-stacks that are presented to it. In this scenario, all that is required from the human analysts is to provide, for a small number of call-stacks, the label of each stack; namely, which type of problem caused the crash. If there are different stacks that correspond to the same problem, the machine learning algorithm can automatically extract what the stacks have in common and use that to classify other stacks that are encountered in the future. The advantages of this approach are that no knowledge needs to be obtained from human analysts and laboriously converted into rules and that as new problems are discovered and the knowledge of the human analysts changes, the machine learning method will automatically adjust by learning from a different training set.

5.2. Learning Algorithm

When a new call-stack in encountered for which the true cause is not known, this stack needs to be compared in some way with the labeled stacks for which the true cause is known. If a stack is found that is very similar to the new stack, it is

likely that the cause responsible for that stack is also the cause of the new stack. In this case, the problem has been automatically diagnosed. This is an example of a "nearest-neighbor" algorithm—other possibilities include decision trees, neural networks, or support vector machines.

However, it is computationally very expensive to compare each new stack with all previously encountered stacks, since the database of stacks may be very large. It would be far better to have some sort of "signature" for each type of known problem. Each new stack can then simply be matched against the signatures for the most commonly occurring problem types in order to find the best match.

A machine learning algorithm can be used to compute "signatures" for each type of known problem from a small sample of call-stacks. This sample, called a training set, or training data, consists of labeled call-stacks—the label, or classification, of each stack denotes which problem type the stack corresponds to or was caused by, as determined by a human analyst. Some problem types may have many different stacks, some only a few or even just one. The machine learning algorithm compares the different stacks for each problem type and determines what they have in common. This common part is then stored as the signature for that problem type and stacks encountered in the future can be matched against it in order to find the signature, and hence the problem type, that is the best match. The signature can be learned efficiently using an incremental procedure that compares each stack for with the signature for its problem type and updates the signature. Each stack does not actually have to be compared with all the other stacks. A simple example of this is shown in Fig. 8.

The algorithm we actually use is somewhat more complicated. It finds the "best" sequence of consecutive function names contained in both the signature and the stack. The best sequence is not necessarily the longest—each common sequence of consecutive function names is given a score that depends on both the length of the sequence and its location—sequences near the top of the stack are weighted more highly. The algorithm finds the sequence with the highest weighted score—this is regarded as the "best" matching sequence. The algorithm is described in Fig. 9.

Stack1: $f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8$
Stack2: $f_2, f_3, f_4, f_5, f_7, f_8$
Stack3: $f_1, f_2, f_3, f_4, f_7, f_8, f_9$

Compare Stack1 with Stack 2 and find highest scoring common subsequence:
Signature = f_2, f_3, f_4, f_5

Compare Stack 3 with Signature and find highest scoring common subsequence:
Signature = f_2, f_3, f_4

Fig. 8. Example of learning problem signature.

```
Input: A collection of training stacks for each problem type
Output: A signature for each problem type

Let P = collection of different problem types
Let stacks_i, i in P, be the collection of stacks for each problem type

For each i in P
    signature_i = stacks_i[0]        (initialize signature to first stack)
    For j = 1 to |stacks_i|
      bestScore = 0
      Repeat
       (*) Traverse stacks_i[j] and signature_i until a common element is found, say at
                    stacks_i[j][m] = signature_i[n]
       weight = 1-(m/|stacks_i[j]|)
                (weight = 1 if at top of stack, weight decreases to zero at bottom)
       Continue traversing until stacks_i[j][m+k] != signature_i[n+k]
       common = {signature_i[n], signature_i[n+1], ..., signature_i[n+k-1]}
       score = weight*|common|
       if score>bestScore
              bestSeq = common;   bestScore = score
       Return to (*)
      Until end of stacks_i[j] is reached
      signature_i = bestSeq
Return signature_i, i in P
```

Fig. 9. Algorithm for learning problem signatures.

5.3. Experiments

5.3.1. Data-Sources

For the learning experiments we use two different data sets. The first is obtained from a tool called Laza that is used worldwide by Lotus Notes help-desk analysts to search for software problem reports whose call-stacks closely match the call-stack of the problem they are dealing with. Laza uses a hard-coded matching procedure of first searching for a sequence of seven matching function calls; if that is not found, then six, and so on. There is no guarantee that the different reports returned from this search are caused by the same underlying cause, but for the case of the best matches this is likely to be the case. We obtained 79 Windows and AIX call-stacks in this way, divided into 12 groups. The largest group has 17 versions of the stack, the smallest groups have only 2. The label, or classification, of each stack is the group to which it belongs.

The second data set is obtained from a Lotus Notes database called TechNotes which contains multiple stacks, often from different platforms, which have been identified by human experts as corresponding to the same underlying problem. This is an ideal data set for learning purposes. Unfortunately, the data set is quite

small, containing only 38 Windows and AIX call-stacks. There are 10 different problem types, the largest containing 9 different stacks, the smallest only 2.

5.3.2. Learning Procedure

The learning methodology adopted in the experiments is shown in Fig. 10. First the stacks are preprocessed to extract the sequence of function names—this requires parsing because stacks from different platforms are in different formats. We then adopt the standard methodology used in machine learning to measure the performance of a learning algorithm.

The data is divided randomly into separate training and testing sets. The learning algorithm is then run on the training set, resulting in a set of signatures, one signature for each type of problem represented in the data. At this point, the training phase terminates and testing takes place. Each stack in the testing set is compared with all the problem signatures to find the one which it best matches. This represents the algorithm's prediction as to which problem was the cause of that stack. This is then compared with the "true" cause, as given by the label of the stack. The overall accuracy of the algorithm is the percentage of stacks in the testing set whose label it predicts correctly. This entire procedure is repeated multiple times and the results averaged.

The amount of data used in the training set can be varied from a small to a large fraction of the total data. As the amount of training data increases, the algorithm learns more and so its performance, measured on the testing set,

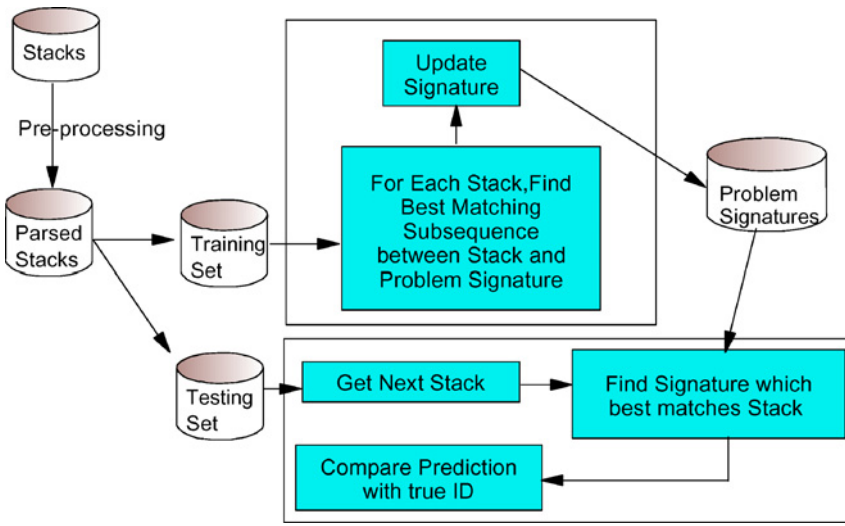

Fig. 10. Learning methodology.

normally improves. However, because the testing set is smaller, the performance measure is generally more variable and less trustworthy.

Note that the division between training and testing sets is done per-problem-type. For example, suppose half the data is used for training and half for testing. If a problem type has 10 stacks, then 5 will be used for training and the other 5 for testing. However, if there are only 4 stacks, 2 will be used for training and the other 2 for testing.

5.3.3. Results

The results of the learning experiments are shown in Fig. 11. As expected, performance, measured as percentage accuracy on the testing set, increases as more data is used for training. On the Laza data set, performance reaches 100% accuracy with about half the data used for training. On the TechNotes data set, performance stabilizes at about 80% but does not improve beyond that. Note that both data sets contain both Windows and AIX call-stacks but this does not confuse the algorithm. Also, when the data sets are combined the performance does not degrade.

The TechNotes data set is clearly much harder to learn than the Laza data set. The most likely reason for this is that the TechNotes data set consists of different stacks that have been identified as being caused by the same underlying problem by human experts. These stacks are sometimes quite different from one another and so seeing only some of them in the training set may be insufficient to be able to accurately classify the remaining ones in the testing set.

The Laza data set is likely to be more representative of stacks collected in a real customer environment. This is because, as we have seen in Section 4, the same or very similar stacks usually occur multiple times. The stacks in the Laza data set are often quite similar to one another and therefore are easier to learn to classify correctly.

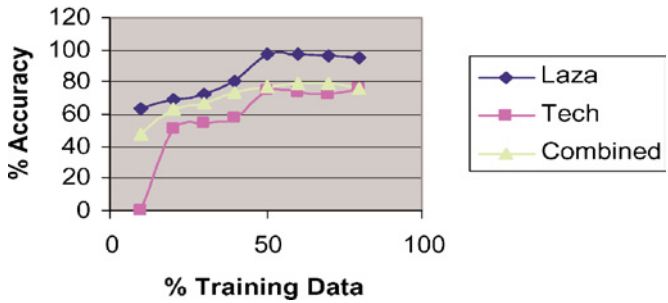

Fig. 11. Learning performance.

6. RELATED WORK

Our work lies at the intersection of several areas: machine learning, call-stacks, and automated software problem diagnosis. Software debugging has a long history, and many applications exist to help developers with program understanding, detecting performance bottlenecks, statically checking code quality, and semi-automated problem diagnosis; e.g. automated testing techniques that operate on the source code to automatically localize failures [12–14]. However, in our situation, the source code is not available and users lack sophisticated software skills. Therefore, we adopt a matching mechanism which does not require any knowledge from the users who report bugs. We also emphasize the learning aspect because we believe that this is an important way to automatically keep up with the latest known bugs/problems.

Several recent works utilize machine learning for software debugging. A statistical debugging technique has been proposed that collects software run-time state information from many users [15]. Decision trees are a well-known machine learning technique which has also been used to build failure models [16, 17]. Program invariants can be dynamically discovered by building detailed execution state profiles using a collection of normal examples [18, 19]. Compared with these approaches, we collect only call-stack information instead of extensive and detailed information from many program executions. Because the order of the functions in a crash stack is important, we use a sequence detection algorithm to learn the call-stack signature related to a problem and a nearest neighbor approach to match new call-stacks with the most similar signature.

The general idea of solving problems by matching symptoms against a historical database is also a well-known technique, known as case-based reasoning (CBR). It has been applied to customer support and help-desk situations [20–22], but these approaches try to find similarities in the problem report information supplied by users, not at the program execution level. They, typically, do not consider call-stacks or the approach of looking for sequences in the data. Furthermore, our approach does not attempt to reproduce human analogical reasoning processes to find the solution, as CBR does. CBR usually attempts to manipulate the solution to an existing problem using logical reasoning to obtain a solution to the new, similar, problem. Here, we focus on learning how to match the problem symptoms so that the similar problems can be accurately identified in the first place.

The collection and use of call-stack information has also been explored. Feng et al. [23] uses call-stack information to develop an anomaly detection algorithm for the purposes of intrusion detection, not problem diagnosis. Lambert and coworkers [24, 25] describe the collection of Java stack traces and discuss various techniques for comparing call-stacks. They use the JVMPI rather than the JVMDI used here, and they also do not use machine learning methods to automatically learn a matching algorithm for the purposes of problem diagnosis.

In summary, although there is a lot of literature in related fields, as far as we know, this is the first work that attempts to use machine learning for automated problem determination using call-stack matching. We study three aspects: instrumentation using existing debugging interfaces to collect call-stacks, matching the call-stacks to diagnose known problems and support human help-desk operation, and machine learning to provide continuous adaptation to new problems.

7. CONCLUSION

Automated problem diagnosis is an important component of any self-managing system. In this paper, we have discussed a first step towards one aspect of this—automatically detecting common problems among many users by matching the problem symptoms, in particular, the program call-stack. This approach complements traditional software debugging and help-desk efforts by reducing the number of cases that human support staff are required to deal with. The use of machine learning techniques allows the system to generalize from a sample of training data to make accurate predictions when it encounters new problems.

There are many directions for further work. The matching algorithm can be enhanced by including additional information besides the function name, taking into account multiple matching subsequences, addressing partial matches, and allowing for the incorporation of domain-specific constraints. Different matching algorithms, each appropriate to a particular operating system or software version, could be learned from sample data using the method we employ here. An instrumented test-bed, in which a variety of specific bugs could be simulated or injected, and the results of the automated diagnosis checked against the "ground truth," would significantly advance our understanding of the strengths and weaknesses of this approach. Finally, the integration of this technology into a comprehensive self-managing system is an exciting prospect for the future.

REFERENCES

1. *BoundsChecker*, http://www.compuware.com/products/devpartner/bounds.htm.
2. *TrueTime*, http://www.hallogram.com/devvb/truetime/.
3. *VTune*, http://developer.intel.com/software/products/vtune/index.htm.
4. *PurifyPlus*, http://www-306.ibm.com/software/awdtools/purifyplus/.
5. S. Shende, A. D. Malony, and R. Ansell-Bell, *Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation*, Proceedings of Parallel and Distributed Processing Techniques and Applications, 2001.
6. "Comparing and contrasting runtime error detection technologies," available from MicroQuill at http://www.microquill.com/heapagent/ha_comp.htm.
7. *HPROF*, http://java.sun.com/developer/TechTips/2000/tt0124.html.
8. *xdProf*, http://xdprof.sourceforge.net/.
9. M. Pietrek, *Under the Hood: The .NET Profiling API and the DNProfiler Tool*, MSDN Magazine, December 2001.

10. *JVMDI*, http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jvmdi-spec.html.
11. D. Crevier, *AI: The Tumultuous History of the Search for Artificial Intelligence*, Basic Books, 1993.
12. A. Zeller and R. Hildebrandt, Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering* 28(2), pp. 183–200, February 2002.
13. J. D. Choi and A. Zeller, *Isolating Failure-Inducing Thread Schedules*, Proceedings of the International Symposium on Software Testing and Analysis, July 2002.
14. H. Cleve and A. Zeller, *Finding Failure Cases through Automated Testing*, Proceedings of the Fourth International Workshop on Automated Debugging, 2000.
15. B. Liblit, A. Aiken, A. Zheng, and M. Jordan, *Sampling User Executions for Bug Isolation*, Workshop on Remote Analysis and Measurement of Software Systems, 2003.
16. M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer, *Failure Diagnosis Using Decision Trees*, International Conference on Autonomic Computing, 2004.
17. A. Podgurski, D. Leon, P. Francis, and M. Minch, *Automated Support for Classifying and Prioritizing Software Failure Reports*, 25th International Conference on Software Engineering, pp. 465, 2003.
18. S. Hangal and M. Lam, *Tracking Down Software Bugs Using Automatic Anomaly Detection*, 24th International Conference on Software Engineering, pp. 291, 2002.
19. M. D. Ernst, J. Cockrell, W. Grisowold, and D. Notkin, dynamically discovering likely program invariants to support program evolution, *IEEE Transactions on Software Engineering*, Vol. 27, No. 2, February 2001.
20. T. Acorn and S. Walden, SMART: Support management reasoning technology for Compaq customer service, *Innovative Applications of Artificial Intelligence*, Vol. 4, 1992.
21. T. Li, S. Zhu, and M. Ogihara, *Mining Patterns from Case Base Analysis*, Workshop on Integrating Data Mining and Knowledge Management, 2001.
22. L. Lewis, *Managing Computer Networks: A Case-Based Reasoning Approach*, Artech House Publishers, 1995.
23. H. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong, *Anomaly Detection Using Call Stack Information*, Proceedings of the 2003 IEEE Symposium on Security and Privacy, pp. 62, 2003.
24. J. Lambert, *Using Stack Traces to Identify Failed Executions in a Java Distributed System*, Masters Thesis, Case Western Reserve University, 2002.
25. J. Lambert and A. Podgurski, *xdProf: A Tool for the capture and analysis of stack traces in a distributed Java system*, International Society of Optical Engineering (SPIE) Proceedings, Vol. 4521, pp. 96–105, 2001.

Mark Brodie is a research staff member in the "Machine Learning for Systems" group at the IBM T.J. Watson Research Center in Hawthorne, NY. He did his undergraduate work in Mathematics at the University of the Witwatersrand in South Africa and received his PhD in Computer Science at the University of Illinois in 2000. His research interests include machine learning, data mining, and problem determination.

Sheng Ma received his BS degree in Electrical Engineering from Tsinghua University, Beijing China, in 1992, and his MS and PhD with honors in Electrical Engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1995 and 1998, respectively. He joined the IBM T.J. Watson Research Center as a research staff member in 1998 and became manager of the "Machine Learning for Systems" group in 2001. His current research interests include machine learning, data mining, network traffic modeling and control, and network and computer systems management.

Leonid Rachevsky is a software systems analyst in the "Machine Learning for Systems" group at the IBM T.J. Watson Research Center in Hawthorne, NY. He obtained his MSc in Mathematics at

Kazan State University and his PhD in Technical Science (Applied Mathematics) at the Kazan Institute of Chemical Engineering in Kazan, USSR (now Russia). He has worked extensively as a software engineer and senior software analyst in Israel, Canada and the United States.

Jon Champlin is an advisory software engineer with the Lotus division of IBM's Software Group. He received a Bachelors of Computer Science from Siena College in 1993. He is part of the external support group and has developed several serviceability features for Lotus Notes/Domino.