Quickly Finding Known Software Problems via Automated Symptom Matching

Mark Brodie, Sheng Ma
IBM T.J. Watson Research Center
Yorktown Heights, NY

Guy Lohman, Tanveer Syeda-Mahmood
IBM Almaden Research Center
San Jose, CA, USA

Laurent Mignet, Natwar Modani
IBM India Research Lab
New Delhi, India

Mark Wilding
IBM Toronto Development Lab
Markham, Ontario, Canada

Jon Champlin, Peter Sohn
Lotus Development Lab
Westford, MA, USA

Abstract

We present an architecture for and prototype of a system for quickly detecting software problem recurrences. Re-discovery of the same problem is very common in many large software products and is a major cost component of product support. At run-time, when a problem occurs, the system collects the problem symptoms, including the program call-stack, and compares it against a database of symptoms to find the closest matches. The database is populated off-line using solved cases and indexed to allow for efficient matching. Thus problems that occur repeatedly can be easily and automatically resolved without requiring any human problem-solving expertise. We describe a prototype implementation of the system, including the matching algorithm, and present some experimental results demonstrating the value of automatically detecting re-occurrence of the same problem for a popular sofware product.

1. Introduction

Much of the research to date in autonomic and self-managing systems has focused on self-optimizing and self-configuring systems, largely because standard performance metrics are available which can be used to measure progress. Unfortunately, there has been far less investigation into how to develop **self-healing** computer systems, particularly software systems. This is likely a result of the challenge presented by the task of automating how to detect, isolate, identify the root cause of, and repair the full breadth of complex problems that may occur in modern systems, as well as how to predict and avoid such problems.

We can begin to address the complexity of this challenge by distinguishing between two categories of problems: known problems and new problems. Many products report that typically half, and sometimes as many as 90 percent, of all problems reported by users are re-occurrences – or *rediscoveries* – of known problems. While such statistics may seem encouraging, support staffs typically spend a significant amount of time manually determining whether a given problem report is in fact new or not. In aggregate, over a third of all time spent by the service organization of at least one major IBM product is consumed by rediscoveries, i.e., by determining whether the symptoms reported by a user match those of any known problem.

While rediscoveries may seem to be "low-hanging fruit" for automation, matching the problem symptoms provided by a user to a large database of such symptoms is not as straightforward as it may seem. Because the symptoms are still largely collected manually by humans and conveyed verbally or by e-mail, there is much potential for miscommunication. The symptoms provided by the user may or may not be relevant to the problem, and critical symptoms may inadvertently be omitted. Each user – or service analyst – may describe the symptoms imprecisely or in slightly different terminology, and even in a different language! Even automated systems collect different types of information and use different formats, for historical or organizational reasons or because the same code operates slightly differently or reports different diagnostics on different platforms. On the other hand, closely-matching symptoms alone may not necessarily represent the same problem, but could be due to an incomplete "fix" or even accidental re-introduction of a bug during the repair process or in subsequent releases.

Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

Our ultimate goal in this research is to significantly reduce the cost of servicing software products by developing symptom databases that can be populated and searched automatically, i.e., by self-healing agents. But a necessary first step is to solve the more immediate problem of speeding the process by which human service analysts now perform this function, Typically, structured symptoms, particularly those generated by software, are more amenable to automated matching than semi-structured or unstructured symptom descriptions provided by humans, so we began with these and hope to extend our work to the more challenging types of symptoms in the future.

Perhaps the most pervasive and highly-structured type of symptom in software systems is the call stack produced by a hard system failure such as a "trapped" error, a "hang", or an undetected "system crash". The call stack reconstructs the sequence of function calls leading up to the failure via the operating system's stack of addresses that is pushed each time a function is called and popped when it returns. Error-handling routines map each address on this stack back to the function's name and the offset from that function's entry point at which the next call was initiated. This provides the exact address at which the problem occurred (in the case of a crash) or was first detected (i.e., "trapped"), as well as the path through the code by which the system arrived at that address. This information can be extremely useful to anyone attempting to fix the problem, so it is typically captured in human-readable form. More importantly for our purposes, the call stack also defines a kind of signature of instances of this class of problem, which we exploit in this work to search and match known problems in a database.

An example of a call stack is given in Figure 1. In this format, each line gives the function's address, followed by the function name, separated by two underscores from the C++ "mangled" names of the arguments of that function and the offset (note that a few lines in the figure wrap). The first line gives the top of the stack, i.e. the most recently invoked function, and the bottom of the stack gives the entry point (which has been omitted for brevity).

The stack in Figure 1 illustrates a number of features that any stack-matching procedure must take into account. The top few routines are typically common error-handling routines that any error in a component is supposed to call (by coding convention), and are therefore often useless for isolating or identifying the problem. Similarly, the entry function at the bottom of the stack is common to all invocations of this product and hence irrelevant. The stack may also contain recursive calls which may need to be ignored in order not to confuse the matching.

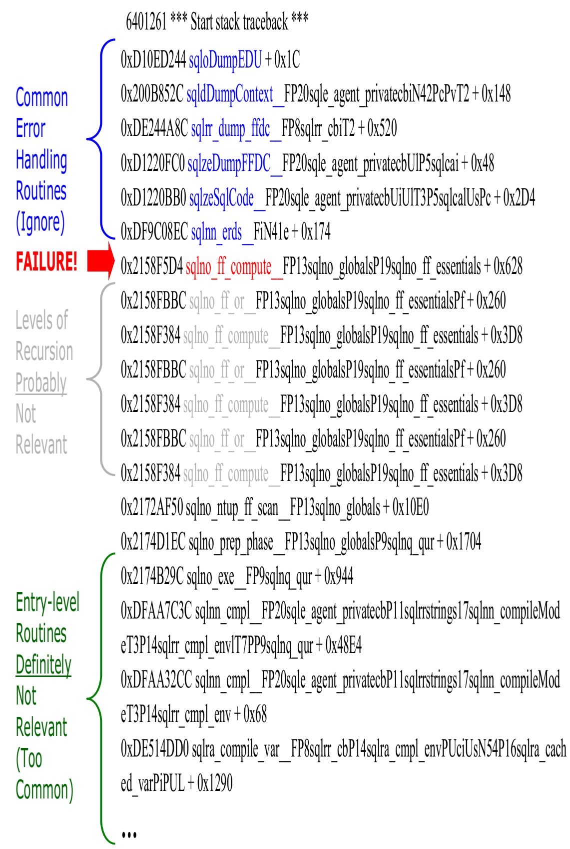

Figure 1: Example of a call stack.

The further up the stack one goes, short of the error handling routines, the more relevant is each function name. The function just below the error handling routines is the function that first detected the error, but it may or may not have caused that error. It's possible that the routines that called it passed an illegal value, or otherwise contributed to the error, so the path to that routine may or may not be relevant to the problem.

Matching a call stack to a database of such call stacks therefore comprises the following steps:

1. Parsing the text field in which the call stack occurs, to identify each function name in the stack (this step may be omitted if the stack is already tagged, e.g. via XML);
2. Removing the error handling routines at the top of the stack and (some of) the common entry routines;
3. Removing recursive function calls if appropriate;
4. Matching the remaining stack to those in the database by a matching algorithm that takes into consideration the position of each function in the remaining stack.

We will describe each of these steps in more detail in this paper, and also detail ways to index the stacks for efficient search without having to compare the target stack against every stack in the database.

The remainder of the paper is organized as follows. We first provide an overview of the architecture of our symptom-matching system in Section 2. Section 3 discusses the design and implementation details of each of the steps, including how the symptom database is populated and indexed, as well as the details of how stack matching is performed. Section 4 presents experimental results for a version of this system that will soon be deployed in the Lotus Notes/Domino 7.0 product. We compare our approach to related work in Section 5, and in Section 6 describe future work and conclusions.

2. Architecture

Figure 2 illustrates the overall process. It has two main parts: the on-line components that collect information from end-users and conduct matching and search operations for a match request, and the off-line components that manage the case database.

The matching and search engine (MSE) is the core of the on-line part. It begins by logging the received request for record-keeping and problem trend analysis. Its main function is to provide matching results to a matching or search request based on the case knowledge base or its indexing database. The request can be submitted by either a software agent or a query interface. The latter, similar to a web search application, provides an interfaces for a user (e.g. a member of the support staff) to submit a matching request and view the returned results - a ranked list of similar cases, with their resolutions and recommended actions. This will help the user to diagnose a problem based on previously solved cases.

Our architecture also supports automated data collection, problem reporting and possibly automated actions. When a problem occurs in the customer environment, a data collection agent automatically gathers diagnosis information, such as the program call-stack, other problem symptoms and environmental information. It then sends a diagnosis request to the matching and search engine and coordinates further actions based on the returned matching results.

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

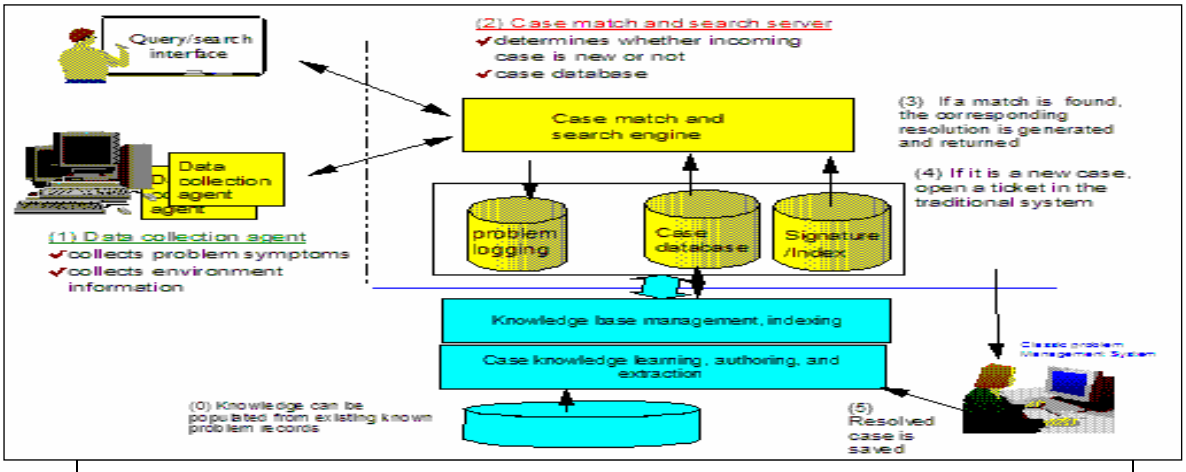

Figure 2: Overview of the architecture of the symptom-matching system

be collected. Further, our current prototype only displays matching results and the recommended actions. It can be extended to support automated actions; e.g. opening a ticket to a problem management system if the case is a new problem, or installing a recommended fix for a known problem.

Knowledge management is an important task to support large scale deployment and maintain the system's effectiveness over time. It includes data management, such as case database indexing for scalable search, as well as the capability to extract cases from existing data sources, support continuous learning to improve matching effectiveness, and authoring and administration interfaces.

3. Implementation

In this section we describe in more detail the following aspects of the symptom-matching system: data collection, case database population and indexing, and call stack matching, including preprocessing steps such as removal of error-handling routines and recursive calls.

3.1. Data-collection

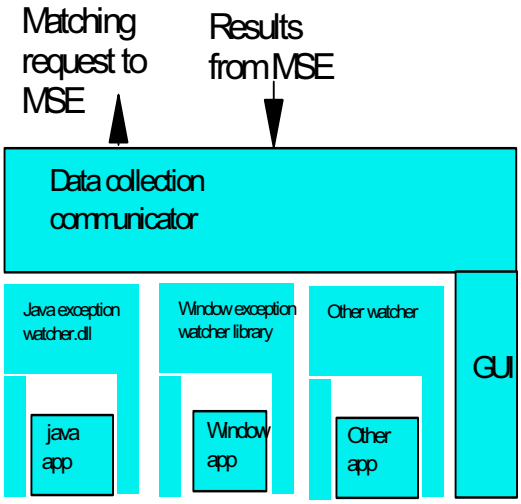

Figure 3: Data-collection agent.

In general, any software product may use its own method to collect problem symptom data. However to create our own testbed for the automated symptom-matching system, we implemented a simple data-collection agent. As shown in Figure 3, the agent has three main components: a communicator that communicates with the matching and searching engine (MSE); a set of watchers that monitor a native application and collect the diagnosis information when a problem such as a crash or exception occurs; and a GUI that interacts with the user to display results.

We have implemented two watchers for Java and Window application respectively. For Java applications we use the JVMDI (Java Virtual Machine Debug Interface) [8]. When the target Java application starts, the JVMDI watcher is loaded into the java.exe process using a command line option. The JVMDI watcher can subscribe to JVM events, in particularly Java exceptions in our implementation. Each time the Java application throws an exception the agent retrieves the information about the class name, failed method, line number, and variable names and values. Such information is handed over to the communicator that in turn submits a request to the MSE.

3.2. Call stack extraction from known problem records

The initial case database against which matching is done can be populated from known problem records. The process employed is the following. An actual call stack in the midst of a problem description appears as shown in Figure 1. We extract the function sequence by looking for consecutive lines that obey the regular expression

address<space>*<func_>+__<rest>*<space>*+<offset>

where address is a string of the form

<Digit>X<HexaDigit>+

and <func> and <rest> are both alphanumeric sequences. The offset is represented by the same regular expression as the address.

In the database creation stage, the following steps, which are described in detail in the following sections, are also performed: stop-word removal (i.e. removal of common functions, e.g. error-handling routines) recursive-function-call removal, indexing (so that

queries can be matched efficiently without searching the entire database) and function frequency estimation, used later in the matching procedure.

3.3. Stop-word Removal

The call stack matching process contains a preprocessing phase before the stack is matched against the case database. Preprocessing is needed to account for error handling routines and other common functions and recursive calls.

Call stacks often include the functions which are either entry points or error handling routines. These occur very frequently across various call stacks without providing any additional information about the problem itself. We call these stop-words, by analogy with common words, such as "the", which are ignored by convential text search-engines.

While the entry points only increase the length of the call stack a little, the error handling routines can have a more profound impact as they are near the top of the stack. For instance if we prune the collection of matches by looking only at the top of the stack, the number of candidate matches would be much larger because the error handling routines are very common across the various call stacks. More dangerous is the case when a new error handling routine is introduced in a new version. This would cause the call stacks for the same problem from previous versions to be rejected as candidate matches. Hence it is critical to remove these stop words from the call stacks prior to matching.

Currently, our system uses a stop-word list provided by a domain expert, but we are exploring the possibility of determining the stop-words automatically by analyzing the frequency and pattern of the occurrences of the function calls near the top and bottom of the call stacks.

3.4. Recursive Function Calls

Recursive function calls occur often in call stacks. For example, a call stack that looks like a()->b()->b()->c() should probably be treated as identical to a call stack a()->a()->b()->b()->b()->c() from the point of view of detection of known problems, as the root cause for the two is likely to be the same. In our system, we remove the recursive function calls to normalize both these call stacks to a()->b()->c().

The pattern of recursion could be a simple sequence of function repeats or a more complex pattern such as a()->b()->c()->b()->c(). For example Figure 4 shows a stack containing a repeat of the functions sqlno_call_sf, sqlno_each_opr, sqlno_call_sf, sqlno_walk_qun. The matching metric should be insensitive to the number of recursive calls so that call stacks with different number of repetitions of the same recursive subsequence should be treated as matches.

sqlzeDumpFFDC sqlzeSqlCode sqlnn_erds sqlno_prop_pipe PIPE sqlno_crule_pipe _root sqlno_crule_pipe sqlno_plan_qun sqlno_call_sf sqlno_each_opr sqlno_call_sf sqlno_walk_qun sqlno_call_sf sqlno_each_opr sqlno_call_sf sqlno_walk_qun sqlno_call_sf sqlno_each_opr sqlno_call_sf	sqlzeDumpFFDC sqlzeSqlCode sqlnn_erds sqlno_prop_pipe PIPE sqlno_crule_pipe_ root sqlno_crule_pipe sqlno_plan_qun sqlno_call_sf sqlno_each_opr sqlno_call_sf sqlno_walk_qun sqlno_call_sf sqlno_each_opr sqlno_call_sf

Figure 4: Illustration of recursion removal.

To handle recursions of arbitrary patterns, we conduct a linear scan of the function sequence looking for possible repeats of a symbol. Suppose the stack is the sequence $\{a_1, a_2, \ldots, a_n\}$. Let {i1, i2, …, ik} be the set of start positions for a symbol a_j such that $a_{i1}=a_{i2}=\ldots=a_{ik}=aj$. Then the subsequence a_{il}-a_j is a recursive pattern if $a_{im}=a_{j+(m-i)}$ for all 1<m<i-l. This can be used to perform recursion removal in $O(n^3)$ in the worst case.

3.5. Matching

The problem of matching a call stack against a case database can be described as follows. Given a query stack $Q=\{q_1, q_2, \ldots, q_m\}$ consisting of a sequence of functions, find the best matching sequence $D=\{d_1, d_2, \ldots, d_n\}$ from the database, or a ranked list of the best matching database sequences.

How can the best matching function sequence be defined? Intuitively, a sequence is a good match if it has considerable overlap with the query sequence. There is a large body of algorithms from the string matching literature for both edit distance - the minimum number of edit operations, insertions, deletions and letter substitutions that transform one string to another [Levenshtein 66] - and biological sequence searching that could be applicable to our problem. However call stack matching contains some distinct features (in addition to the recursive function calls removed above): function gaps are more common than function substitutions, and position in a sequence is important.

Gaps vs. substitutions

First, simple approaches such as finding the longest common subsequence would not be suitable, as they allow arbitrary gaps between contiguous subsequences in defining the overall subsequence. Variants based on edit-distance allow for substitutions as well as insertions. For the case of call stack matching, while gaps or insertions are tolerable (corresponding to extra function calls or missing functions calls), it is unlikely that function names can be substituted and still represent a manifestation of the same problem. Thus insertions and deletions should be emphasized over substitutions. However, a long gap would imply a sequence of different set of function calls in one of the sequences, so that it is unlikely that they represent the same problem.

Position in a sequence

Further, edit distance-based algorithms are insensitive to the position in the string where the match occurs. This is important for call stack matching because, once the common functions (e.g. error handling) are discarded (see section 3.3), the function that is responsible for the problem is likely to occur closer to the top of stack rather than the bottom. Thus the position from the top of stack must be a factor taken into account during sequence matching.

We now present an algorithm to match call stacks keeping the above requirements in mind. Our algorithm is a variant of the Needleman-Wunsch algorithm in bioinformatics [Needleman 70]. It is also based on the dynamic programming principle in composing best matching sequences by extending best matches from prefixes of candidate sequences. Like the Needleman-Wunsch algorithm, it does a global alignment of candidate sequences. It finds the best global alignment as the maximum match consisting of the largest number of residues i.e. subsequences of one sequence that can be matched to the other while allowing for all possible gaps. However, it does not allow substitutions, a key operation in Needleman-Wunsch that is necessary in bioinformatics but not in call stack matching.

The algorithm conducts a global alignment of the query and database call stacks by computing a dynamic programming matrix H, where the element H(i,j) is the optimal score for aligning the sequences up to the ith and jth element in the respective sequences. The higher the score of a path through the matrix, the better the alignment. The matrix H is found by progressively finding the matrix elements, starting at $H_{1,1}$ and proceeding in the directions of increasing i and j. Each element is set according to:

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ H_{i-1,j} - d \\ H_{i,j-1} - d \end{cases}$$

where d is a fixed gap penalty to account for a single function insertion or deletion, and S_{ij} is the match score of the functions at position i and j in the stacks respectively.

It is in the computation of S_{ij} that we model the specifics of the call stack matching problem. Specifically, when $a_i=b_j=f$ we associate a cost with the matching by considering three independent factors, namely, (a) the importance of the function symbol f, (b) the position of the query symbol a_i from the top of stack, and (c) the extent of the gap between the ith symbol in the query sequence with the jth symbol in the database sequence. Each of these independent factors we model using probabilities as follows.

The importance of a function symbol f is given by its discriminability, defined as

$$P_1(f) = 1 - \frac{\text{Number of stacks containing } f}{\text{Total number of stacks}}$$

The position of the query symbol a_i from the top of the stack is given by

$$P_2(a_i = f) = 1 - \frac{i}{m}$$

where m is the length of the query call stack.

The gap between the positions of the query and database call stacks is given by

$$P_3(a_i, b_j) = e^{-\frac{|i-j|}{2}}$$

using an exponential weighting of the gap penalty.

Since P_1, P_2, and P_3 are independent, the overall cost of match S_{ij} is given by their product as:

$$S_{ij} = \begin{cases} P_1(f) * P_2(a_i) * P_3(a_i, b_j) \\ when \quad a_i = b_j = f \\ 0 \quad otherwise \end{cases}$$

3.6. Indexing

The matching algorithm described above compares the query call stack with each call stack in the database. This clearly does not scale well to industrial-sized databases. To more quickly locate the most likely matches, a more scalable approach would be to create an index of the call stacks in the database. The difficulty is deciding upon a key on which to index, so that the index will return a superset of those call stacks that are most likely to be good matches for the query call stack. The matching algorithm can then be applied to that much smaller superset of good candidates. If it were not truly a superset, using the index would result in false negatives (missed matches).

Postulating as before that the top of the stack (after removing the common error handling functions) contains the best discriminators for matching purposes, we constructed keys for a standard B+-tree by hashing the top J function names in the stack. We experimented with J varying from 1 to 4, and found little difference in the results. The hashing of course raises the possibility of collisions (false positives), but we are applying the matching algorithm to all the results returned by the index anyway, so false positives simply end up being ranked very low. If any of the top 4 function names are missing, it's very unlikely to be the same problem.

Creating the index entries is part of the normal population of the database: as new entries are inserted, the top J function names are concatenated, hashed, and stored with the resulting hash code as an additional column in the database over which a standard B+-tree is defined. When a query call stack is submitted, it goes through the exact same processing: it too is parsed and has its error routines removed, so that the key can be constructed by hashing the top J functions. This key is then used to query the database in far less time than it would take to access every call stack in the database. In general it is better to keep J small (say J=2), so that results aren't biased by how the function that first detects the problem is reached. However in practice it's very rare where the top two functions are the same but the third is different – usually the two stacks will be the same.

4. Experiments

In our initial experiment we have a collection of crash-stacks from crashes whose causes are unknown. We use simple matching to explore how often the identical stack 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 such as the Operating System 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 that 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 crash-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 yields 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 Figure 5 below. 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.

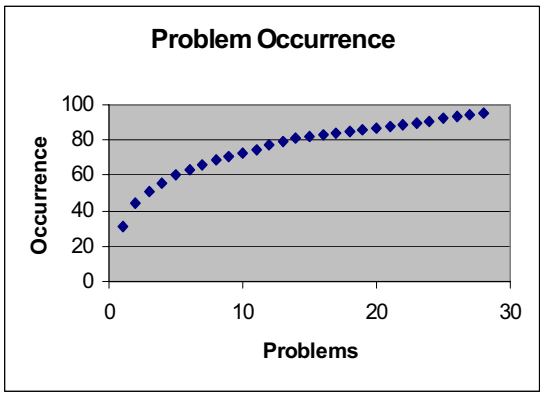

Figure 5 – A small number of problems account for most of the crashes.

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 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. Related Work

Our work lies at the intersection of several areas: Automated software problem diagnosis, case-based reasoning, call stack analysis, and string matching.

Automated testing techniques typically operate on the source code to automatically localize failures [Zeller 02, Choi 02]. However in our situation the source code is not available and users lack sophisticated software skills. Therefore we adopt a matching mechanism that does not require any knowledge from the users who report bugs. Other automated debugging techniques include: collecting software run-time state information from many users [Liblit 03], decision trees to build failure models [Chen 04, Podgurski 03], and discovering program invariants by building detailed execution state profiles using a collection of normal examples [Hangal 02, Ernst 01]. Compared with these approaches, we rely only on call stack information from crashes, instead of extensive and detailed information from many normal program executions.

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 [Acorn 92, Li 01], but these approaches try to find similarities in the problem report information supplied by users, not at the program execution level. The Microsoft Windows Error Reporting Service [Microsoft] collects crash data and groups crashes using module and application names and offsets, but does not consider call stacks or the approach of looking for repeated sequences in the stack.

The collection and use of call stack information has also been explored. [Feng 03] uses call stack information to develop an anomaly detection algorithm for the purposes of intrusion detection, not problem diagnosis. [Lambert 01] describes the collection of Java stack traces and discuss various techniques for comparing call stacks. [Brodie 05] adopts a similar framework to ours but does not do stop-word and recursive function removal and uses a simpler matching algorithm which finds the highest weighted common subsequence of functions.

There is a large body of literature in string matching algorithms, ranging from "Longest Common

Subsequence" to various adaptations of edit distance between two strings - the minimum number of edit operations, insertions, deletions, and letter substitutions that transform one string to another [Levenshtein 66] to suffix trees [Gusfield 98]. The string matching problem has also been addressed extensively in bioinformatics for biological sequence alignment. The subsequent literature on alignment has been enormous, and includes seminal papers such as the original Needleman–Wunsch dynamic programming solution [Needleman 70] and many others since then. Recently, the literature on basic methodology and tools development has been growing rather than shrinking, indicating that the alignment problem is still not solved in bioinformatics. As explained above, our algorithm is a variant of the Needleman-Wunsch algorithm with appropriate modifications for the problem of call stack matching.

6. Conclusion

Automated problem diagnosis is a critical component of any self-managing system and a prerequisite to self-healing. Much unnecessary work can be avoided by first determining whether a problem is already known or not, and automating that step requires methods to match the symptoms of the problem to those in the database of known problems. We have presented in this paper an initial system that enables efficient location and ranking of known problems, in response to a query providing a program call stack. By suitably removing common error and entry routines, indexing the case database, and using sophisticated matching algorithms, we have demonstrated an architecture that scales well for large databases while still having excellent precision and recall. The system is scheduled to be deployed within Lotus Notes/Domino 7.0, and other IBM products are considering its incorporation as well.

In the future, we hope to further refine the matching algorithms, preprocessing, and indexing based upon further testing and customer experience on large databases from multiple products. We also hope to automate the detection of the error functions and other "stop-words", rather than relying upon domain experts to identify them. This will make the algorithm more generic. We also plan to fully integrate a learning component in the system. One can easily envisage a system that learns to adjust its matching procedure over time in response to feedback about which matches were or were not successful. (Some preliminary work in this direction appears in [Brodie 05]).

Call stacks are only one type of symptom, and perhaps the easiest to match. In the future, we plan to extend the symptom database to incorporate a wide variety of symptom types in the database, in order to be able to handle a wider range of problem types and perhaps to exploit synergy between the specification of multiple symptom types to better isolate the best match in the database. Lastly, this symptom matching symptom must be integrated with other components into a comprehensive self-healing system.

References

[Acorn 92] Acorn, T., and Walden, S., *SMART: Support Management Reasoning Technology for Compaq Customer Service*, Innovative Applications of Artificial Intelligence, Volume 4, 1992.

[Brodie 05] Brodie, M., Ma, S., Rachevsky, L, and Champlin, J., *Automated Problem Determination using Call-Stack Matching*, Journal of Network and Systems Management, special issue on self-managing systems, to appear.

[Chen 04] Chen, M., Zheng, A., Lloyd, J., Jordan, M., and Brewer, E., *Failure Diagnosis Using Decision Trees*, International Conference on Autonomic Computing, 2004.

[Choi 02] Choi, J.D. and Zeller, A., *Isolating Failure-Inducing Thread Schedules*, Proceedings of the Interational Symposium on Software Testing and Analysis, July 2002.

[Ernst 01] Ernst, M.D., Cockrell, J., Grisowold W. and Notkin, D., *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, IEEE Transactions on Software Engineering, Volume 27, No. 2, February 2001.

[Feng 03] Feng, H. H., Kolesnikov, O., Fogla, P., Lee, W. and Gong, W., *Anomaly Detection Using Call Stack Information*, Proceedings of the 2003 IEEE Symposium on Security and Privacy, 2003, pg 62.

[Gusfield 98] D. Gusfield, J. Stoye, *Linear time algorithms for finding and representing all the tandem repeats in a string*, Report CSE-98-4, Department of Computer Science, University of California, Davis, 1998

[Hangal 02] Hangal, S. and Lam, M., *Tracking down software bugs using automatic anomaly detection*, 24th International Conference on Software Engineering, 2002, pg 291.

[JVMDI] JVMDI - http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jvmdi-spec.html

[Lambert 01] J. and Podgurski, A., *xdProf: A Tool for the capture and analysis of stack traces in a distributed Java system*, International Society of Optical Engineering (SPIE) Proceedings, Volume 4521, 2001, pg 96-105.

[Levenshtein 66] Levenshtein V.I., *Binary codes capable of correcting deletions, insertions and reversals*, Soviet Physics Doklady 10(8), pp. 707-710.

[Li 01] Li, T., Zhu, S., and Ogihara, M., Mining Patterns from Case Base Analysis, Workshop on Integrating Data Mining and Knowledge Management, 2001.

[Liblit 03] Liblit, B., Aiken, A., Zheng, A., and Jordan, M., *Sampling User Executions for Bug Isolation*, Workshop on Remote Analysis and Measurement of Software Systems, 2003.

[Microsoft] Microsoft Windows Error Reporting Service http://www.microsoft.com/whdc/maintain/WER/ErrClass.mspx#EAAA

[Needleman 70] Needleman, S. B., and C. D. Wunsch. *A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins*. In Journal of Molecular Biology ~8. 1970, pp. 443-453.

[Podgurski 03] Podgurski, A., Leon, D., Francis, P. and Minch, M., *Automated Support for Classifying and Prioritizing Software Failure Reports*, 25th International Conference on Software Engineering, 2003, pg 465.

[Zeller 02] Zeller, A. and Hildebrandt, R., *Simplifying and Isolating Failure-Inducing Input*, IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.