Static Analysis, Code Transformation and Runtime Profiling for Self-healing

Mohammad Muztaba Fuad, Debzani Deb and Jinsuk Baek
Department of Computer Science
Winston-Salem State University, Winston-Salem, NC 27110, USA
{fuadmo, debd, baekj}@wssu.edu

***Abstract*—A self-healing application brings itself into a stable state after a failure put the software into an unstable state. For such self-healing software application, finding fix for a fault is a grand challenge. Asking the user to provide fixes for every fault is bad for productivity, especially when the users are non-savvy in technical aspect of computing. If failure scenarios come into existence, the user wants the runtime environment to handle those situations autonomically. This paper presents a new technique of finding self-healing actions by matching a fault scenario to already established fault models. By statically analyzing the code and transforming it in a way to allow the program to profile itself, it is possible to capture runtime parameters and execution pathways during runtime. The transformed program then can establish stable execution models that can be used later to match with an unstable execution scenario. Experimentation and results are presented that showed that even with additional overheads; this technique can prove beneficial for autonomically healing faults and reliving system administrators from repeated and routine troubleshooting situations.**

***Index Terms*—Self-adaptive application, Autonomic computing, Code transformation, Fault similarity.**

I. Introduction

Today's computing environments are complex and heterogeneous tangle of hardware, middleware and software from multiple vendors that is becoming increasingly difficult to program, integrate, install, configure, tune, and maintain. This leads to the idea of autonomic computing [1]-[2] where the complexity and the management of such systems is handled by the system itself. One aspects of such self-management is self-healing where the system corrects itself after a failure situation. In real life, the end users of today's big and complex software are left with the task of managing the system when the computational task is faltering due to failures that they cannot fix because of lack of computing knowledge. To provide users with a self-managed system and to achieve the goal of self-healing, we need to provide two supports. Firstly, since we do not want the developers worry about incorporating the self-healing features, appropriate code transform and injection techniques are needed at the object code level (merely because for older systems, the source code may not be available). Secondly, there should be a way to identify a specific failure and distinguish it from other similar failures with different root causes. None of these tasks is trivial and requires extensive effort. Our past experience on code transformations [3]-[4] showed that the first of the above-mentioned supports is achievable. In this paper, we are going to present an approach to correctly identify and distinguish failures in a software application so that self-healing can be provided autonomically. This support is necessary if we want to envision a true self-healing system where the self-management feature will actively categorize new faults, learn patterns for cause-affect relationships, perform management actions for identified faults and even try to predict what to do in case an unknown fault appears in the system.

The rest of the paper is organized as follows. Section II talks about the motivation behind this work and discusses on related work. Section III introduces code transformation technique and discusses costs associated with such transformation. Section IV provides details of the runtime operation of the transformed code. Section V discusses about fault similarity and their relationship. Section VI shows results from experimentations and finally Section VII concludes the paper.

II. Motivation And Related Work

The motivation of this research comes from the fact that it is a non-trivial task to automate the process of making a regular user application into a self-healing application. To achieve this goal, proper code transformation techniques (to add required functionalities) are needed. Our previous investigation [3]-[4] in this research area showed that it is viable to do code transformations to inject autonomic properties into existing applications. It also showed that relieving the end-user from the complex programming interfaces and metaphors, the application is more eligible to users who do not have advanced knowledge in programming paradigms.

When a runtime software system requires management due to any kind of failure or user action, an effective fix (management action) should be identified and applied quickly. Failure can occur due to system faults or due to performance bottlenecks. In most cases, the root cause of such failure is human ignorance or mismanagement. It is therefore compelling to have

doi:10.4304/jcp.8.5.1127-1135

systems that self-manage itself and reduce the day-to-day involvement of humans in the operation of the system. Matching a fault with the database of existing faults is also not trivial [5] since the same fault can occur for multiple different reasons and in different circumstances. By keeping track of what the program is doing during fault free runs and comparing that with situation where the program is not running smoothly; fault models can be generated. Also, after a few times of interactions with a particular software application, users naturally express a great deal of their goals, preferences, and personality. That way, patterns of usage and failures scenarios can be learned over time by monitoring such application.

In the last couple of years, software fault healing (specially, adaptive or self-*) was forced back to the spotlight because of current software's inherent complexity and requiring more expert human intervention and man-hour to manage and maintain these software. There is a plethora of research work addressing this issue and some of the research related to this work is presented here. One major difference between the work presented in this paper and other related work is that we establish domain specific inherent relationships among faults in a system to find the root cause of a fault by looking in to the similarity/dissimilarity between faults by using an established execution trace history of the program. We believe that, building up these relationships between faults and root causes will allow the system us to match fault scenarios faster and allow systems to provide true self-healing features. Another factor that distinguish this research is that the added functionality for self-healing is added without the need for any extra programming on behalf of the application developers.

Ding, et al. [6] proposes a black-box approach of software development that automatically diagnoses several classes of application faults using the application's runtime behaviors. Their approach collects application's runtime signature as we do, but instead of concatenating previous execution traces to form signatures, we generalize traces and formulate signature. Also, instead of manual invocation of diagnosis process, we incorporate that as part of the system (by means of code injection), so that in an unstable state, the injected codes try to diagnose and solve the problem itself with minimum or no human intervention.

Yuan, et al. [7] tries to find correlation of an unstable application state with a list of solved cases. Instead of using vague text descriptions to identify problem situations, the authors employ statistical techniques to match an unknown fault to a set of known fault situations. An obvious difference with our approach is that we trace the target application in a regular execution to establish known execution paths and signatures and once an unknown case is identified the injected code handles all related healing procedures. In this work, an already establish list of 100 top faults and their root causes are given (specific to a certain problem domain) and an unknown cases is matched with this list of cases and the one which is close to the unknown fault is provided to the user so that the user can take control and solve that particular situation.

Cook, et al. [5] presents a similar approach that directly matches an unknown fault to a list of known fault. We think both of these work are good foundation for further extension and we utilizes both these ideas in our approach and extended by creating relationship between faults and signatures and also by automating the process of healing the software once the fault is diagnosed.

Binkley, et al. [8] presented a work where the authors used text similarity measures to predict fault from existing application log data. Although their approach is innovative, we cannot employ that in our problem domain because of lack of fault data and logs and because of dissimilarity between the underlying data.

III. Code Transformations

The goal of this research is to inject self-healing related autonomicity into non-autonomous object-oriented code, whose source code is no longer available (Even with source code, this technique can be used by compiling and converting the source code to byte code). Therefore the code must be analyzed and the autonomic functionality should be inserted in such a manner that it is separated from the service functionality of the user application. The technique presented in this section injects self-healing capabilities into byte code without any user involvement so that at run time, the application can heal itself after transient software faults. Fig. 1 shows a general overview of the application life cycle. The user application is statically analyzed and appropriate code segments and hooks are injected on the object code and a modified (the computation logic of the user application remains same) version of the application is produced, which acts as an independent entity (autonomous entity). Although the Fig. 1 shows the code inserts and the original code of the same size, there is no relationship and the overall code inflation depends on different factors.

Major characteristic of the code transformation process is as follows:

- Java byte code is used as the target for application's programming domain.
- The code transformation uses standard Object Oriented Programming (OOP) metaphors (Class, Object and Methods) to inject code segments. Therefore, with minor modification, the techniques can be ported to other interpreted OOP language.
- A proxy based structure is used which encapsulates one or more runtime objects into a single manageable entity that communicates with other such entities in the system with a single communication channel. Having an encapsulating proxy object allow us to incorporate the autonomic functionalities seamlessly into the user objects with the help of sensors, actuators and control interfaces.
- The granularity for adding self-healing is per method basis. There are more methods in the proxy class than

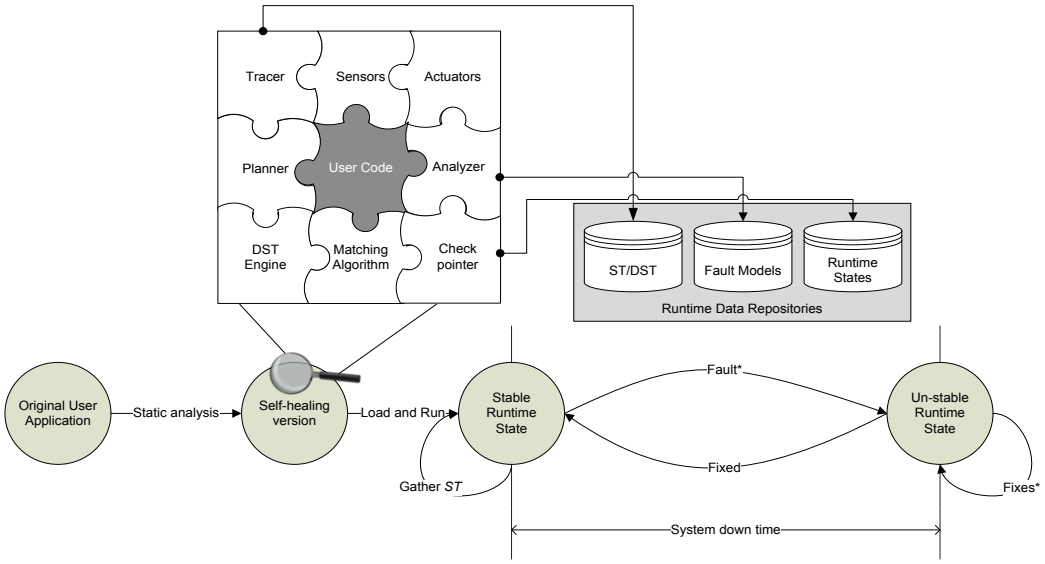


Figure 1. Application's life cycle.

the original class to interact with the environment and also to manipulate the objects itself. The original methods are overridden with the following structure:

Pre-processing
Original method call
Post-processing

Any direct field access is converted to *getter* and *setter* methods to facilitate generation of Signature-Traces (STs). Instead of instrumenting each method, a wrapper method is created to ensure the consistency of the existing line number table for debugging purpose. Since system class methods cannot be modified, they are not considered during trace generation.

- To restart seamlessly from a crash or after a runtime exception, the internal state (fields, method parameters and local variables, next byte code to be executed) of the object has to be made persistent. According to the given user policy, instrumentation points in the code are determined and appropriate method calls are inserted to checkpoint the current state of the object.
- Polymorphic method calls are not considered because of the complexity to determine the origin of a method call in a polymorphic call. This would require a full-fledged stack oriented emulator as in the Java Virtual Machine. Creating one such emulator is out of the scope of this research. Similarly reflection in the user code is not considered due to the added complexity in the byte code-rewriting phase.
- To save the signature information of any running method inside the object, local variable values have to be saved during check pointing along with other necessary information, such as line number of the next instruction to be executed, value of actual parameters and value of any object field. To gather the current state of the object, all the local variables in the current scope along with any fields and class variable's values need to be saved. As the Java compiler does not save any local variable information (such as type information, variable scope etc.), the static analyzer gathers this information by analyzing the byte code and adds that information as a byte code attribute to each method.
- To handle any runtime exceptions, the method body is encapsulated by another try-catch block (which catches the *Throwable* super class) to give the method another opportunity to continue after the statement where the exception is thrown. However, adding a new try-catch block introduces different semantic and algorithmic scenarios, which are addressed at byte code level to fully support runtime exceptions.

During runtime, a clone of the running method is created (by using the injected code during static analysis) where, at the beginning of the code, new code is inserted to reinitialize the object with a consistent state and to restart the method at the point where the fault occurs. Byte code sequences are generated and added at the end of any existing code in the constructor to avoid inheritance related constraints placed on the constructor (such as calls to *super* must occur first before any other statement). For methods, a new *Object* array is introduced as a local variable and the length of the array is set to the total number of local variables inside that method. Care needs to be taken in calculating the number of local variables inside a method since any arguments to that method are also counted as local variables and the nature of the method (static or instance) also dictates how those local variables are numbered. Therefore, the *maxLocal* attribute in the method is updated with the newly calculated number of local variables. At a checkpoint location, the byte code sequence is inserted that assigns each local variable with in the scope of that position to the cells of the *Object* array declared previously. However, the challenge is to wrap each of these variables (primitive types) with proper *Object* classes and assign them to individual cells of the *Object* array. The local variable attribute is used to determine each of the variable's type and then appropriate wrapper classes are

used. For reference types (including arrays), nothing extra needs to be done since standard polymorphism takes precedence in such assignment. The call to the *checkpoint* method inside the *statusObject* class is then generated in the byte code and added to the sequence generated thus far. To pass all arguments to that method, each of the arguments are loaded first in the runtime stack in the same order as the formal parameters, before actually invoking the method. This is to ensure the correct semantics of the stack-based JVM execution model. The *Object* array, current method name, current byte code offset and the current object reference are passed as arguments to the *checkpoint* method. Adding new codes in existing methods (including constructors) introduces new challenges and forces recalculation of the maximum stack depth during execution and consequently the corresponding byte code attribute (*maxStack*) needs updating. Without recalculating and updating this attribute, the resulting code will not be verified by the Java Runtime Verifier [9] and will not be executed by the JVM.

Because new proxy classes are created and segments of byte code are inserted into the exiting byte code, we see an average of 35% to 51% inflation in the resulting code size depending on different factors and user level policies. However, we believe that, for the sake of the transparency and the added dynamics in the system, this added space penalty is worth taking. Experimentations were made to find the timing requirement of such delegated method call using proxy based autonomic elements and affect of the injected code on the runtime performance of the user application. The timing increase was negligible compared to the size of the code inflation and can be minimized by different optimization techniques.

IV. Runtime Operation

Once the static analyzer produces the transformed code, it is executed in the similar fashion as the original program. The modified program runs in two different states:

a) *Stable State*: In stable state, the tracer keeps all records of execution and establishes signature of stable execution paths of the program. The check-pointer keeps track of the application in regard of its current states (field values, object states, execution point etc.), so that once the software needs to restore to the last stable state, it can do that after healing whatever failure it encounters. The modified application collects runtime parameters (signature) and execution pathways (traces) and stores it in an object called *Signature-Trace* or *ST*. A *signature* of a program is its runtime state including field values, object states, open files, environment information etc. We can represent this information as an *n-gram* based representation as in [6]; however will not be adequate to hold the relationships and rank of those attributes for generalization purpose. A *trace* is program execution path that records method invocation and stack traces. One of the challenges is to determine the granularity of the trace or the execution pathways. To generate such trace data, proper code segments have to be inserted at that granularity level. Fig. 2 shows the code inflation due to different granularity levels for a specific user application. Although, statement level captures the most information related to an execution pathway, it is extremely hard to inject code to track every statement execution because of branching statements and repeated code blocks. Even with the simplest program, the code inflation was exponential. As the granularity level becomes coarser, the fewer details can be gathered related to that execution pathway. There is a distinctive tradeoff

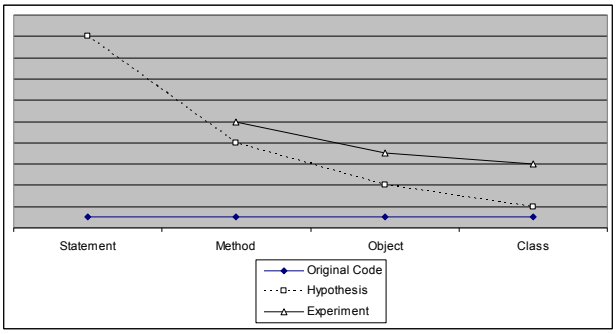
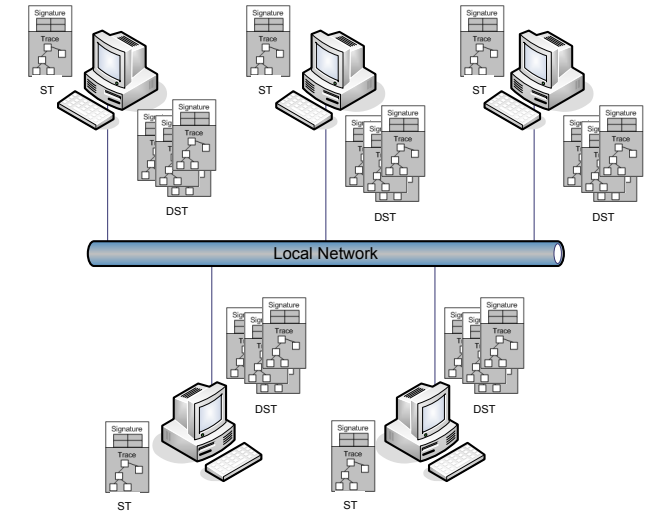


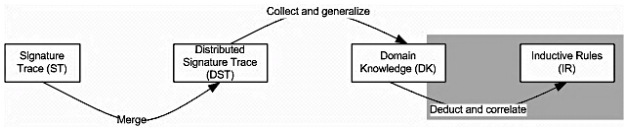
Figure 2. Trace granularity and code inflation relationship.

exists here between runtime details and code inflation and a granularity level has to be selected that satisfy our needs. After extensive analysis, we selected method level granularity for tracing our execution pathways. After a stable run, the acquired signature and trace is merged into a data structure that we are denoting as *Signature-Trace* or *ST*. After each stable run, the generated *STs* are compared and generalizations are made. To make generalizations, we do not just pick the common entities or generalize on the value of a field or parameter. We rank each entity by their occurrences and sort them accordingly. However, to limit the inflation of the data structure, a threshold is set and if the size goes beyond the threshold limit, the least ranking entity is deleted. *ST*s are created for every single run of the application and in regular intervals are shared among the machines in the system. Each machine merges all gathered *ST*s and generates *Distributed ST* or *DST*. Initially, the system treats *DST*s as the sole source for domain knowledge. However, with enough run of the application and generation of enough *DST*s, generalizations are made and global *Domain Knowledge* (*DK*) database is generated from which fixes have to be deduced for different fault scenarios. It is to be noted that, once *DK* is generated, each machines need to update its *DST* with the more generalized version of any *ST* combinations from *DK*. Eventually, well-formed fault scenarios are generalized and put into the fault model database. This whole process is shown in Fig. 3. One point to be noted from Fig. 3(a) is that the target application has to have different running instances for faster

generations of *DSTs*. Also, in Fig. 3(b), the last two steps of the knowledge transformation is proposed but haven't been implemented fully.



(a)



(b)

Figure 3. System view and knowledge transformation.

b) *Unstable State*: Once there is a failure, the application suspends execution (by virtue of the transformed code) and moves to an unstable state. In this state, the injected code takes charge of the program and added components take charge of appropriate functionalities to heal the application back to a stable state.

In traditional self-management architecture, domain experts specify rules that map symptoms to fixes in an *if-then-else* format. Previously defined static rules might work well for simple systems where all possible failures are known in advance and a universal fix can quickly solve most failures. To extend such static approach, we proposed a new concurrent algorithm that employ feedback-driven loop to find the best solution scenario (s) for previously unseen failure. The overall algorithm to find fixes is presented bellow. Step 1 and 2 are concurrent processes for better response of the algorithm. As noted earlier, after couple of runs, *DST* is replaced by *DK* and therefore the algorithm only mentions about *DST*.

1. *Collect local application's signature trace (ST).*
   - A. *Merge $ST_{i..n}$ to form Distributed Signature Trace (DST)*
   - B. *In regular intervals, share DST_i with other machines in the domain running the same software.*
   - C. *If a new DST is received, then update local DST_i with more generalized scenarios from the received DST.*
2. *If a failure, f is detected:*
   - A. *Save the current system state and treat the system as unstable.*
   - B. *The fault matching algorithm analyze the failure signature trace (ST_c) and match it with all possible DST and generate a list of fixes $f_1 \ldots f_n$ with existing success rate of those fixes.*
   - C. *The list of possible fixes will be applied one (f_x) at a time to the application.*
     - i. *If f_x results in a stable state, increase the success rate of that ST and continue to step H.*
     - ii. *If f_x results in an unstable state and part of the DST exactly matches the ST_c, then apply the corresponding fix for that part of DST and continue as step C.i. The algorithm has to be aware of the depth of this kind of recursive try and should have a threshold value.*
     - iii. *If f_x results in an unstable state and there is a DST, which partially matches the ST_c, then mark that as a candidate ST for future processing.*
   - D. *For all candidate ST, find the subset (size n) with the highest success rate.*
   - E. *Calculate the distance of ST_c with the members of the subset and find the ST with the lowest distance. Different distance formulas can be used to calculate this distance.*
   - F. *Apply the newly found fix (in the closest ST) and*
     - i. *if it results in a stable state, continue to step G, otherwise,*
     - ii. *if not exceeded time limit (threshold) then refresh DST with others in the system and continue to step C*
     - iii. *otherwise continue to step H.*
   - G. *Resume the application.*
   - H. *If all of the above fails then save the current status of the application, let the administrator know about the fault and log of the injected code.*

Timing requirement for Step 1 is discussed in Section VI (Fig. 6). Step 2 depends on several factors, such as, size of *DST*, time to save current state, length of the current trace, time to match etc. Because of these factors, timing requirement for this step varies widely. Since Step 1 and Step 2 are concurrent process, during unstable state of the program, Step 1 ceases operation so that no unstable *ST* corrupts the current *DST*.

Not all faults can be healed automatically or even recovered from. This paper is concerned with transient faults (network outage, memory overload, disk space outage etc.) that occur after the program is deployed. Such faults could result from problems in the user code (such as class loader related exceptions), in the underlying physical system (such as IO exceptions) or network connection (such as communication and server

exceptions) or in the run-time environment (such as access control exceptions). Non-transient faults, caused by bugs in the user code (logical errors), user generated custom exceptions or faults generated due to the functional aspect of the program are outside the control of this approach and should be addressed by the system administrator or the developer of the user program.

V. Fault similarity and Relationships

Along with building *DST*s, the system also develops fault models to find similarity between a known fault and an unseen fault. A fault model is the collection of generalized signature traces and possible fixes for a specific fault scenario. Once a fault has occurred, it is analyzed and related *ST*s are tagged with that particular fault. For well-known faults, domain engineers can specify fault models early at the life cycle of the program. With more execution and fault occurrences, model for each fault will get enriched with the different execution scenarios and associated signature-traces. Once there are a substantial number of fault models acquired, a previously unseen fault can be matched with the faults in the model database by analyzing how similar each *ST* of two individual fault models is. As shown in Fig. 4 (a), the signature-traces of each fault in the fault model databases are matched and depending on that, the faults are arranged in a two-dimensional acyclic graph with weighed edges as shown in Fig. 4 (b).

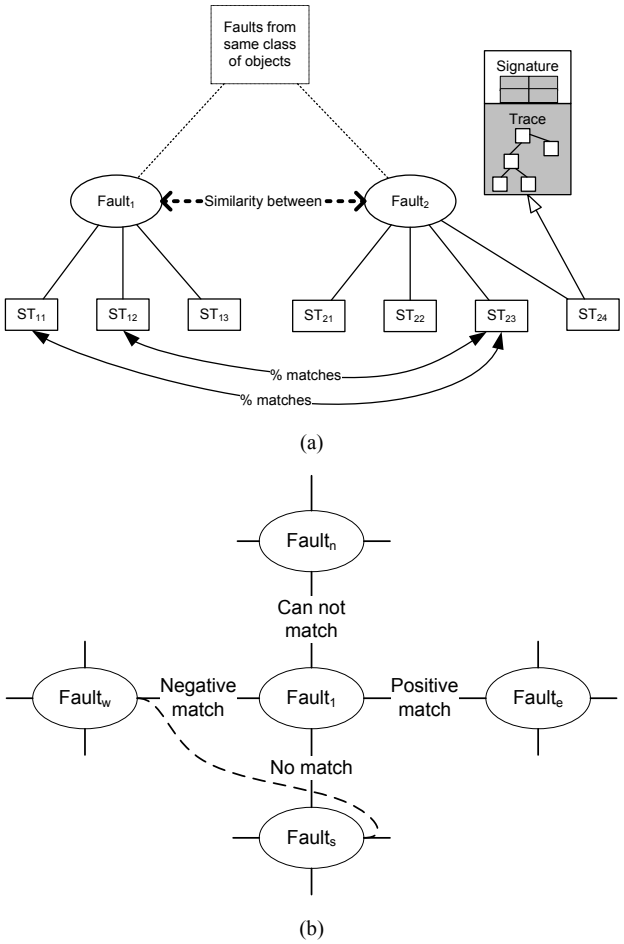


Figure 4. Fault similarity and relationships.

The measurements, which have to be made for the classification algorithm to work is as follows:

- *Positive match*: Number of *STs* that matched (within a certain error margin) between faults and if number of positive matches is greater than negative matches.
- *Negative match*: Number of incomparable STs and if number of negative matches ate higher than positive matches.
- *Cannot match*: Faults are from two different classes of objects with no common structure between them.
- *No match*: Faults from same class of objects without match in their *STs* or missing attributes in their signature.

Generally, faults from the same class of objects show positive matches, however, because of similar patterns in execution path and environment parameters, sometimes two different classes of faults can have a positive match between them (as shown by the dotted line). This complicates the graph and the matching process as this leads to a multiple dimension graph, intertwined with links between nodes in different dimensions. To minimize this situation, we added the following relationship refinement:

- Family tree: If the fault classes have semantic relationships, such as inheritance. This minimizes (can-not match and no-match cases) or strengthens (negative or positive match cases) links between faults. Faults from the same parent class are treated specially and commonalities are noticed between two sibling fault models for deducing a fix.

Once there is a stable fault model graph in the system, the following algorithm finds a match for a newly occurred fault.

// Argument: A fault, f, which is to be matched
// Returns: Fix, F_x
classify (fault f)
1. Put every fault model entity in a candidate set.
2. For every candidate, F_c in the fault model, do
2.1. Calculate the percent match between F_c and f.
- *a) if exact match (within some error margin), returns find_fix(F_c, f)*
- *b) if positive match, then discard all negative matched entities of F_c from the set and record this percent match.*
- *c) if negative match, then discard all positive matched entities of F_c from the set and record this percent match*
- *d) if can-not match or no-match then ignore.*

3. Sort all percent match and
- *a) pick the top most positive matched fault, F_p and return find_fix(F_p, f).*
- *b) if there are no positive matches, then select the lowest negative matched fault F_n and return find_fix(F_n, f).*
- *c) if no match at all, inform system administrator for intervention.*

find_fix(..) matches the fault scenario with the candidate fault model and find the attached fix with that

model. At the beginning, when the fault model is being developed, there will be a high degree of mismatches and frequent invocation to the system administrator. However, gradually this will improve as the fault model evolves to more matured and established state.

VI. EXPERIMENTATION AND EVALUATION

To evaluate the effectiveness of the transformed self-managed system, a robust and quantitative benchmarking methodology is needed. However, developing such a benchmark methodology is a non-trivial task [10] given the many evaluation issues and environment criteria to be resolved. If there were such a benchmarking methodologies, then the effectiveness will be quantitatively measured by comparing the performance of the proposed technique against benchmarks associated with the current modes of operation (without the self-healing approach). However, since that is not feasible with the current state of benchmarking technologies for self-managed systems; we can deploy techniques, such as [11], to measure the effectiveness of the proposed algorithm.

Java enterprise application server GlassFish [12] is selected as the target application and Java Platform Debugger Architecture [13] is being used to gather runtime application status to generate the program signatures and traces. Inside GlassFish application server, we run a custom server program within which we injected different faults to check our approach. This configuration was then cloned in multiple machines in the system and was executed simultaneously.

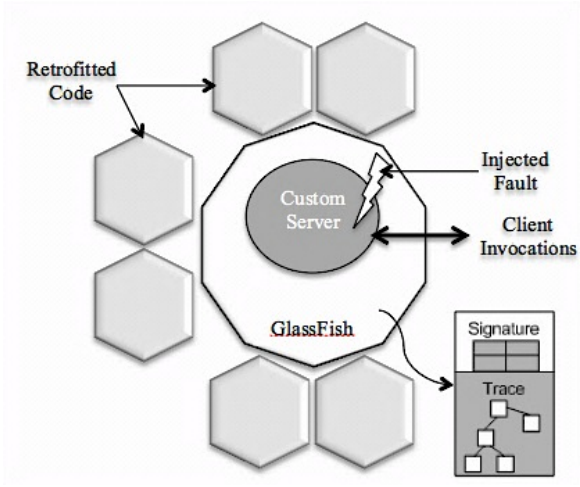


Figure 5. Experimental Setup.

As shown in Fig. 5, *STs* from GlassFish were generated and the custom server program was crashed inside GlassFish with injected faults so that we can match GlassFish's execution during a stable and unstable run. The custom program, which is run inside GlassFish, is a simple server that accepts incoming client request and serves those requests. By injecting faults in that server program, we can see what GlassFish does and develop and test our algorithm.

At first, we measure and analyze the time it takes for gathering and storing *ST*s and merging them into DST. Fig. 6 shows the timing requirements to gather ST and merge them to form DST. To calculate the times, we run five clone instances of the GlassFish in five different machines, ten times and then average was taken. The overhead to collect ST is negligible however the time requirement to merge STs into DST is substantial. To overcome this, we can stop tracing a given pathway any further, when there is a stable fault model in place that include this pathway. Instead of sharing the entire DST across the network for updating, an incremental updating algorithm is being developed to minimize this overhead. This algorithm only propagates the changes made to the DST between runs to other machine.

Next we looked into the affect of this overhead with relation to the size of the DST. As it is evident from Fig. 7, the overhead to gather ST is nearly constant regardless off overall size of the DST. However, as the size of the DST increases, the overhead to merge them also increases. As described above, an incremental merging algorithm will minimize this time for bigger DST. One thing to notice in this graph is that each size of the DST in the horizontal axis is gathered after different number of runs of GlassFish (1, 10, 50, 100 and 500 total runs).

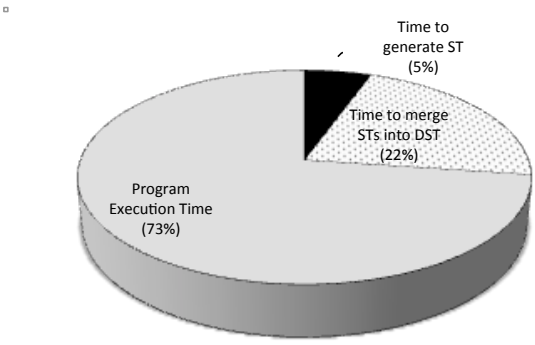


Figure 6. Time requirements for generating ST and DST.

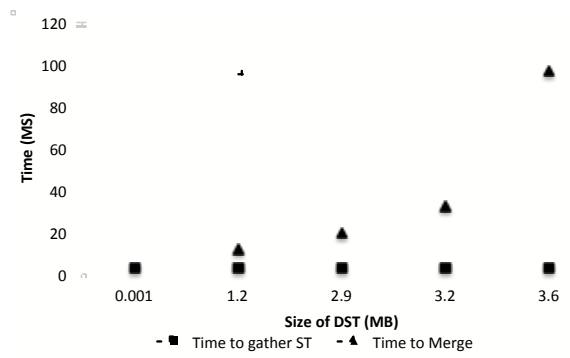


Figure 7. Effect of DST size on timing requirement.

Fig. 8 shows the space requirements for the resultant DST. Although with smaller number of runs, the space requirement is greater, it eases a lot as the number of runs

increases and similar STs are combined together. We anticipate that the size of the DST will hover around a constant size after enough runs of the software. One thing to remember here is to reduce the communication overhead; incremental propagation of the DST is necessary across the machines instead of broadcasting the entire DST across the machines.

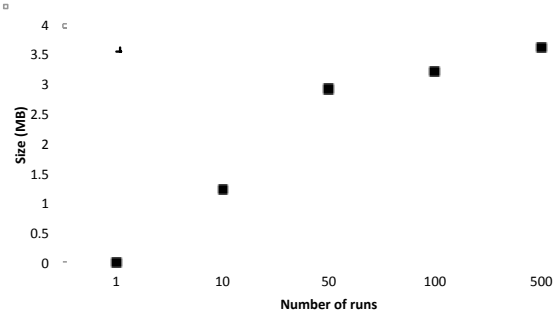


Figure 8. Space requirement of DST.

To find how efficiently the matching algorithm works, we injected a fault for which the ST is already established in the DST (manually inserted for testing). Since we know that the algorithm should return a match for that, we measure how long it took to do that on different size of DST. The matching time for a known fault was proportional to the size of the DST. With more occurrence of the same fault, the algorithm ranks the faults (and associated STs), so later occurrences can be matched even faster. To find an unforeseen fault, several issues need to be addressed for devising experimentations. Following is a list of such issues and possible solution.

Issue: How to deviate a ST from a known ST and use that deviated ST to find a close match?

Solution: This is crucial to check how good the algorithm is performing in identifying unforeseen fault scenarios. By changing the depth of the trace or by modifying signatures, we deviated STs from their instances in the DST. We used percent match to

Issue: How to devise an accuracy matrix for partial match?

Solution: We set a threshold of percent match for partial matched STs. Although this is a static threshold, a dynamic threshold controlled by user level policy can be implemented.

Issue: How to find the timeout threshold to match unknown faults?

Solution: Extensive analysis should be made to find a threshold that balances response time and matching accuracy. Currently a static threshold is used which is chosen to satisfy particular response time requirements.

Issue: Should timing requirement be addressed in case of an unknown fault is categorized within a certain threshold?

Solution: This depends on the system level priority and currently a rigid response time is set as the priority of the matching algorithm.

Our goal is not to make the program run without any human intervention. We think that is impractical and even unattainable. We envision a system where the software heals autonomically most mundane and repeated faults themselves and only invoke the system administrator in case of completely new or miss-matched faults. That would certainly minimize system administrator's workload and will allow him/her to concentrate more high-level management issues. The technique presented in this paper will allow that.

VII. Conclusions

Day-to-day maintenance of software systems is a grand challenge due to the fact that the runtime environment changes continuously. Users of such systems want to run their application and do not want to worry about the mundane task of system management in the face of a failure. If such management scenarios come into existence, the user wants the runtime environment to handle those situations autonomically. This paper presents a new technique of matching unknown fault scenarios to already established fault models to self-heal user applications. By capturing runtime parameters and execution pathways, stable execution models are established and later are used to match with an unstable execution scenario.

Self-adaptive computing is a new paradigm where computing systems possess the capability of self-management. Building a self-adaptive version of existing software from scratch is desirable however is not always an option. Mostly because of the cost and time associated with a major development like that. It is tremendously beneficial to programmers if such adaptive behaviors can be added automatically and transparently into existing code. This paper presents one such technique where the self-healing support is provided transparently and the added functionalities are incorporated into existing user application by appropriate code analysis, transformation and profiling techniques. Results from experimentations are also presented that showed the usability of the proposed technique for self-healing applications.

References

[1] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," *IBM Corporation*, October 15, 2001.

[2] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, Vol. 36, No. 1, pp. 41-50, 2003.

[3] M. M. Fuad, "Code Transformation Techniques and Management Architecture for Self-manageable Distributed Applications," *Proceedings of the Twentieth International Conference on Software Engineering and Knowledge Engineering*, USA, pp. 315-320, 2008.

[4] M. M. Fuad and M. J. Oudshoorn, "Transformation of Existing Programs into Autonomic and Self-healing Entities," *Proceedings of the 14th IEEE International*

Conference on the Engineering of Computer Based Systems, USA, pp. 133-144, 2007.
[5] B. Cook, S. Babu, G. Candea and S. Duan, "Toward Self-Healing Multitier Services," Technical Report of the Duke University, 2005.
[6] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang, "Automatic Software Fault Diagnosis by Exploiting Application Signatures," *Proceedings of the 22nd Conference on Large installation System Administration Conference*, USENIX Association, 2008.
[7] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang and W. Ma, "Automated Known Problem Diagnosis with Event Traces," *SIGOPS Operating System Review*, Vol. 40, No. 4, 2006.
[8] D. Binkleya, H. Feildb, D. Lawriea and M. Pighinc, "Increasing diversity: Natural language measures for software fault prediction," *Journal of Systems and Software*, Vol. 8, Issue 11, 2009.
[9] Java Virtual Machine Specification, http://java.sun.com/docs/books/jvms/secondedition/html/VMSpecTOC.doc.html.
[10] A. Brown, J. Hellerstein, M. Hogstrom, T. Lau, S. Lightstone, P. Shum, M. Yost, "Benchmarking autonomic capabilities: Promises and pitfalls," *Proceedings of the 1st International Conference on Autonomic Computing*, IEEE NSF, May 2004.
[11] R. Griffith, "The 7U Evaluation Method: Evaluating Software Systems via Runtime Fault-Injection and Reliability, Availability and Serviceability (RAS) Metrics and Models," *Ph. D. Thesis*, Columbia University, Aug 2008.
[12] GlassFish Open Source Application Server, http://glassfish.java.net/.
[13] Java Platform Debugger Architecture, JPDA, http://java.sun.com/javase/technologies/core/toolsapis/jpda.

Mohammad Muztaba Fuad is an Assistant Professor of Computer Science at Winston-Salem State University. A native of Bangladesh, he did his Ph. D. in Computer Science from Montana State University, USA in 2007 and Masters of Computer Science from University of Adelaide, Australia in 2001. He served as Technical Program Committee member and reviewer for many books, journals, conferences and workshops. His research interests include self-adaptive computing, distribute and parallel systems and software engineering. He is a member of ACM.

Debzani Deb is an Assistant Professor in the Department of Computer Science at the Winston Salem State University (WSSU), USA. She received her Ph.D. (May, 2008) from the Department of Computer Science at Montana State University, USA, where her research was in the area of Self-managed Software Systems. Her research interests are distributed and high performance computing, software engineering, algorithms, autonomic computing, data mining and programming languages. Before joining to WSSU she worked as a faculty at Indiana University of Pennsylvania, University of North Carolina at Greensboro and Montana State University.

Jinsuk Baek is Associate Professor of Computer Science at the Winston-Salem State University (WSSU), Winston-Salem, NC. He received his B.S. and M.S. degrees in Computer Science and Engineering from Hankuk University of Foreign Studies (HUFS), Korea, in 1996 and 1998, respectively and his Ph.D. in Computer Science from the University of Houston (UH) in 2004. Dr. Baek was a post doctorate research associate of the Distributed Multimedia Research Group at the UH. He acted as a consulting expert on behalf of Apple Computer, Inc in connection with Rong and Gabello Law Firm which serves as legal counsel to Apple computer. He has served on Editor of the KSII Transactions on Internet and Information Systems, and The Smart Computing Review. He also served or currently serving as a reviewer and Technical Program Committee for many important Journals/Conferences/Symposiums/Workshop in Computer Communications Networks area. His research interests include scalable reliable communication protocols, computer systems, mobile computing, wireless sensor networks, and network security protocols. He is a member of the IEEE.