Using Search Methods for Selecting and Combining Software Sensors to Improve Fault Detection in Autonomic Systems

Maxim Shevertalov, Kevin Lynch, Edward Stehle, Chris Rorres, and Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
{max, kev, evs23, spiros}@drexel.edu, crorres@cs.drexel.edu

Abstract

Fault-detection approaches in autonomic systems typically rely on runtime software sensors to compute metrics for CPU utilization, memory usage, network throughput, and so on. One detection approach uses data collected by the runtime sensors to construct a convex-hull geometric object whose interior represents the normal execution of the monitored application. The approach detects faults by classifying the current application state as being either inside or outside of the convex hull. However, due to the computational complexity of creating a convex hull in multi-dimensional space, the convex-hull approach is limited to a few metrics. Therefore, not all sensors can be used to detect faults and so some must be dropped or combined with others.

This paper compares the effectiveness of genetic-programming, genetic-algorithm, and random-search approaches in solving the problem of selecting sensors and combining them into metrics. These techniques are used to find 8 metrics that are derived from a set of 21 available sensors. The metrics are used to detect faults during the execution of a Java-based HTTP web server. The results of the search techniques are compared to two hand-crafted solutions specified by experts.

***Keywords*-genetic algorithms; genetic programming; search based software engineering; autonomic computing**

I. Introduction

Complex software systems have become commonplace in modern organizations and are critical to their daily operations. They are expected to run on a diverse set of platforms, while interoperating with a wide variety of applications and servers. Although there have been advances in the engineering of software, faults still regularly cause system downtime. The downtime of critical applications can create additional work, cause delays, and lead to financial loss [**?**]. Faults are difficult to detect before an executing system reaches a point of failure, as the first symptom of a fault is often system failure itself. While it is unrealistic to expect software to be fault-free, actions can be taken to reinitialize the software, quarantine specific software features, or log the software's state prior to the failure.

Many fault detection approaches rely on runtime metrics such as CPU utilization, memory consumption, and network throughput. One detection approach, created by the authors, uses information collected by the runtime sensors and constructs a convex-hull geometric object that represents the normal execution of the monitored application [**?**]. The convex-hull approach then detects faults by classifying the state of an executing application as inside or outside of the convex hull. If the state is classified as being outside of the hull it is considered to be failing.

Due to the computational complexity of creating a convex hull in multi-dimensional space (one dimension per metric), the convex-hull approach is limited to a few metrics. Therefore, not all sensors can be used to detect faults and some must be dropped or combined with others. For example, instead of using `heap memory` and `non-heap memory` as individual metrics, these sensors can be summed together to create a `total memory` metric. The problem of selecting and combining sensors to create a limited number of metrics can be formulated as a search problem.

This paper compares the effectiveness of genetic-programming, genetic-algorithm, and random-search approaches to solving the sensor selection and combination problem described above. These techniques are used to find 8 metrics derived from a set of 21 available sensors. The metrics are used to detect faults during the execution of NanoHTTPD [**?**], which is a Java-based HTTP server.

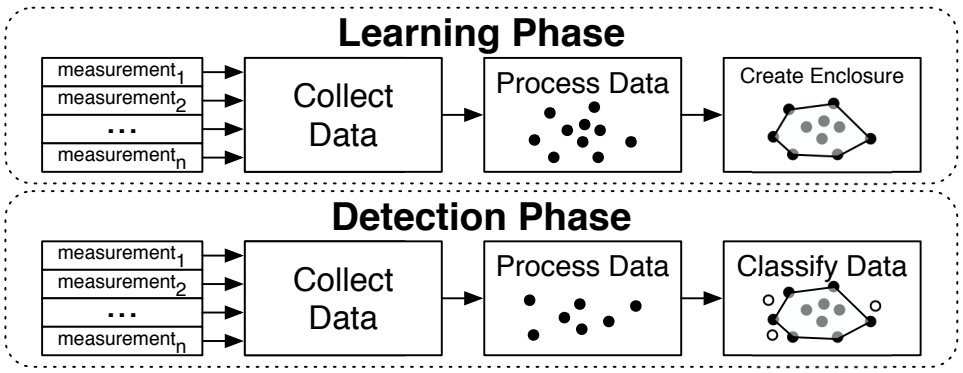

Figure 1. During the learning phase, the convex-hull approach collects and processes runtime measurements. It then constructs a convex-hull enclosure that represents the normal execution of the monitored system. During the detection phase, runtime measurements are collected and processed. Then, each data point (vector of metrics) is classified as normal (gray points) if it is inside the enclosure, or anomalous (white points) if it is outside of the enclosure.

The solutions found by the search techniques are targeted to a specific set of 9 seeded faults and are compared to each other as well as two hand-crafted solutions specified by domain experts. This paper illustrates the relative effectiveness of search based software engineering techniques as well as the solutions proposed by domain experts.

The rest of the paper is organized as follows: Section II presents the background information on fault detection, specifically the convex-hull method; Section III describes the search techniques used in this work; Section IV compares the solutions found using search to each other as well as to the two hand-crafted solutions; finally Section V states conclusions and plans for future work.

II. Background

Autonomic computing is a branch of software engineering concerned with creating software systems capable of self-management [**?**]. One of the aspects of autonomic computing, and the focus of this work, is fault detection. Fault detection is a step toward creating a self-healing software system, meaning a system that is aware when a fault has occurred and can correct it.

Existing methods of detecting software faults fall into two categories, signature-based methods and anomaly-based methods [**?**]. Signature-based methods detect faults by matching measurements to known fault signatures. These techniques can be used in static fault-checking software such as the commercial antivirus software McAfee [**?**] and Symantec [**?**], as well as network intrusion detection systems such as Snort [**?**] and Netstat [**?**]. These techniques can also be used to detect recurring runtime faults [**?**].

If a set of known faults exists, then training a system to recognize these faults will typically lead to better fault detection. However, that system is unlikely to recognize faults it has not seen before. For example, a fault caused by a zero-day virus is unlikely to be detected by commercial antivirus software because there are no known patterns to recognize.

Anomaly-based methods learn to recognize the normal runtime behavior of the monitored system and classify anomalous behavior as potentially faulty. The advantage of using anomaly-based methods is that they can detect previously unseen faults. However, they risk incorrectly treating any newly encountered good states as faulty. This occurs when insufficient training data is supplied to the method.

Typically, anomaly-detection methods begin by collecting sensor measurements of a system that is executing normally. Then, they construct a representation of the monitored system and compare any future measurements against that representation. A naïve approach will assume that all sensors are independent and determine the safe operating range for each of them. In other words, during the learning phase, this method will record the maximum and minimum values of each sensor and then classify the system as faulty when any of the measurements fall outside of their determined ranges.

While the naïve approach is capable of detecting some faults, it can fail when the metrics are dependent. Therefore, more sophisticated fault detection techniques assume that there are dependencies between metrics. Many of these techniques employ statistical machine learning [**?**], [**?**], [**?**], [**?**]. A common approach is to use sensor correlations to represent a monitored system. During detection, if the correlations between sensors become significantly different from the learned correlations, the system is classified to be

in a faulty state.

A. Convex-Hull Approach to Fault Detection

The convex-hull method is an anomaly-detection method and, thus, can detect faults not seen before. However, unlike the previously discussed anomaly-detection methods, it does not use statistical machine learning. Instead, it uses computational geometry and works well independent of whether the metrics are dependent or not.

Figure 1 illustrates the phases of the convex-hull fault-detection technique, which consists of a training phase and a detection phase. During the training phase, the normal execution behavior of the monitored system is modeled by constructing a convex-hull enclosure around good measurements (*i.e.*, when faults were not observed). During the detection phase, observed measurements are classified as normal or anomalous based on whether they are inside or outside of the convex hull.

The convex hull for a finite set of points is the smallest convex polygon that contains all of the points. A convex polygon is a polygon such that for every two points inside the polygon the straight-line segment between those points is also inside of the polygon. A convex hull for a finite set of points in one-dimensional space (one metric) is the line segment between the largest and smallest values. In two-dimensional space (two metrics) one can think of each point as a peg on a board and the convex hull as an elastic band that snaps around all of those pegs. In three-dimensional space (three metrics) one can think of the convex hull as an elastic membrane that encompasses all of the points.

A problem with the convex-hull method, and the reason for this work, is that it is computationally expensive to construct a hull in higher dimensions given a large set of points. As more metrics are considered, the process of computing the convex hull becomes more time consuming. Once constructed, however, the hull can classify new points as in or out at a rate of about 100 classifications a second. The QHull library [**?**], which this work uses for all of its convex hull calculations, is limited to 8 dimensions. Due to this constraint, a set of sensors must be turned into a set of no more than 8 metrics. The QHull library is the premier convex hull computational library that is used in many mathematical tool kits. Even with its limitation to 8 metrics, it is still the fastest library for computing convex hulls in higher than 3 dimensions.

Evolutionary algorithms [**?**] have shown promise in finding good solutions in a large space of potential answers. Evolutionary algorithms begin their process by constructing a random population of answers. Next they evaluate all of the solutions using a fitness function. The population is then recombined via a breeding process to construct the following generation. Given a new population the process is repeated until some acceptable answer is found.

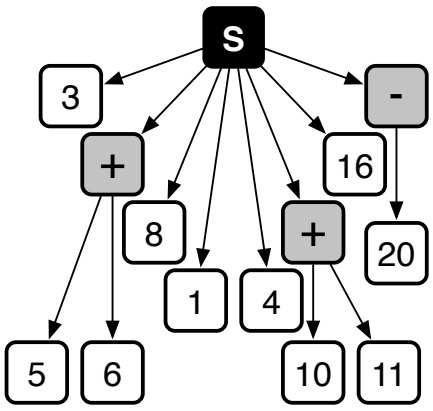

Figure 2. The chromosome is represented as a tree structure. It has three types of nodes: a root node (black); an expression node (gray); and a terminal node (white).

Given the problem of selecting and combining sensor measurements into metrics, our conjecture was that an evolutionary algorithm would find a good solution. The search space is large and the dependencies between different sensors is not known. For these and other reasons, which will become clear further in the paper, an evolutionary algorithm was chosen as the start of our work.

In our previous paper [?], we compared the convex hull approach to several statistical approaches with favorable results. During that work we selected metrics based on our intuition. This extends that work and attempts to find a better set of metrics using search based techniques.

III. Search Techniques

The search library used in this work was implemented using the Ruby programming language [**?**]. While it was initially developed as a GP library, it supports GA and random search as special cases of the GP.

The description of the library used to find a solution is separated into the following parts: Section III-A presents the chromosome representation; Section III-B describes how the initial population is created; Section III-C presents the process of evaluating a population of chromosomes; finally Section III-D describes how new populations are generated.

A. Chromosome Representation

Figure 2 illustrates the chromosome representation as a tree structure. Each chromosome represents a function that

```
TERMINAL => {1|2|3|...|21}

   UEXP => {- EXP}

   BEXP => {- EXP EXP|+ EXP EXP}

    EXP => {UEXP|DEXP|TERMINAL}

      S => {s EXP EXP EXP EXP
            EXP EXP EXP EXP}
```

Figure 3. The grammar used to generate the initial random population. Generation begins with the `S` rule and proceeds until all branches end with a `TERMINAL` node.

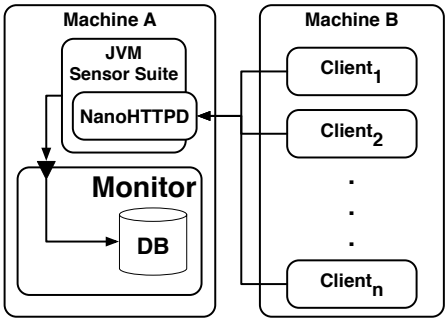

Figure 4. The experimental testbed contains two machines. The first is used to run NanoHTTPD and the autonomic framework. The second houses clients that access NanoHTTPD.

accepts a set of 21 sensor measurements as input and produces a set of 8 metrics as output. The tree is constructed using one of four nodes. The `root node` has 8 branches, one for each metric. `Binary operation nodes` can be either addition or subtraction. `Unary operation nodes` negate the subtrees linked from them. `Terminal nodes` contain the `ID` value of a sensor.

Using the tree structure to represent the entire solution, as opposed to each metric individually, allows the algorithm to consider how metrics are related to each other. Simply finding the best eight metrics independent of each other does not lead to good results because many of the metrics are dependent and tend to have similar classification effectiveness.

Note that if the `root node` is forced to decompose directly into `terminal nodes`, the problem changes from a GP trying to evolve 8 metric formulas to a GA trying to find the optimal subset of 8 sensors.

B. Creating the Initial Population

The initial population is created using the grammar [**?**], [**?**] of prefix expressions presented in Figure 3. The production begins with the `root node`, which is expanded into 8 `EXP` nodes. When expanding a node with several options, one of the options is chosen at random. Once all of the productions are expanded into `TERMINALS`, the generated abstract syntax tree (AST) is simplified into the chromosome format, as demonstrated by Figure 2.

It has been shown that restricting the tree height of the initial chromosomes can lead to good convergence rates [**?**]. Therefore, the initialization accepts in a parameter k that specifies the maximum height of a random tree. When a branch of the tree being generated reaches a height k, the generation algorithm automatically chooses a `terminal node`. Because each production, except `EXP`, is expanded into an `EXP`, which has a production `TERMINAL` as one of its options, the generated AST is guaranteed to have a maximum height of $k + 1$.

C. Evaluating a Population

A chromosome represents a function that translates a point in 21-dimensional space into a point in 8-dimensional space. Each chromosome is evaluated based on how well it can classify data collected during the execution of NanoHTTPD as normal or anomalous. To perform this evaluation three data sets are needed. The first set, labeled as the `training set`, is a set of points used to construct a convex hull that represents normal execution. The second set, labeled as the `normal set`, is a set of points collected during the normal execution of NanoHTTPD. These points are not part of the `training set` and are used to determine the false positive rate. A false positive occurs when a point known to have been sampled during the normal execution of NanoHTTPD is outside of the convex hull and is therefore falsely classified as an anomaly. The third set, labeled as the `error set`, is a set of points collected during the faulty execution of NanoHTTPD. These points are used to determine the false negative rate. A false negative occurs when a point known to have been sampled during the faulty execution of NanoHTTPD is inside of the convex hull and is therefore incorrectly classified as normal. The `normal` and the `error` sets are divided into two subsets. The first is used during the training phase and second is used to evaluate the final result.

To create the three data sets described above, a case study using NanoHTTPD was conducted. Figure 4 presents the design of the testbed used in this case study. One machine is used to host NanoHTTPD and Aniketos, our autonomic computing framework. Another machine manages clients that request resources from NanoHTTPD. A JVM sensor suite is used to monitor NanoHTTPD's execution and report measurements to the monitoring server. The

monitoring server stores the gathered data and processes it using the QHull library [?].

NanoHTTPD was chosen for this case study because it is open source and manageable in size, thus making it easy to modify and inject with faults. It is a web server that hosts static content. NanoHTTPD spawns a new thread for each HTTP client request. If a thread crashes or goes into an infinite loop, it does not compromise NanoHTTPD's ability to serve other files.

The goal of this case study is to replicate realistic operating conditions. To this end, it uses resources and access patterns of the Drexel University Computer Science Department website. Nine weeks worth of website access logs were collected and all resources accessed in those logs were extracted and converted into static HTML pages. Out of the nine weeks of logs, three weeks were chosen at random to be used in the case study. These were replayed against the statically hosted version of the website and provided the case study with realistic workload access patterns.

One week of request logs was used to create the `training set`. Another week of request logs was used to create the `normal set`. A third week of request logs was used as background activity during the fault-injection experiments, thus creating the `error set`. All measurements were normalized such that each metric's value ranged between 0 and 100.

NanoHTTPD was injected with nine faults. These faults represent coding errors, security vulnerabilities, and attacks. Two of the most common coding errors are the infinite-loop and the infinite-recursion faults. An infinite-loop fault is presented as a `while` loop that iterates indefinitely. Two versions of this fault were created. One, where each iteration of the loop does nothing, and the second, where each iteration of the loop performs a `sleep` operation for 100ms. The goal of the slow-infinite-loop fault is to create a more realistic scenario during which an infinite loop is not overwhelming the CPU. Similar to the infinite-loop fault, the infinite-recursion fault also has two versions, a regular and a slow one that performs a `sleep` operation for 100ms. An infinite recursion is presented as a function calling itself until the thread running it crashes due to a `StackOverflowError` exception.

Another fault injected into NanoHTTPD was the memory-leak fault. Two versions of this fault were created. The first version performed a realistic memory leak and leaked strings containing the requested URLs by adding them to a `vector` stored in memory. The second version of the memory-leak fault doubled the size of the leak `vector` with each request.

Log explosion [**?**] is another problem common to the server environment and was injected into NanoHTTPD. The log-explosion fault causes NanoHTTPD to write to a log file continuously until there is no more space left on the hard drive. While this does not cause the web server to crash, the log-explosion fault does cause a degradation in the performance of the server.

In addition to faults due to coding errors, two security attacks were perpetrated against NanoHTTPD. In the first, NanoHTTPD was injected with a spambot trojan [**?**]. Once triggered, the spambot began to send spam email message packets to an outside server at a rate of 3 email messages per second. Each message was 1 of 3 spam messages chosen at random. The messages varied in length between 166 and 2325 characters each.

The second attack perpetrated against NanoHTTPD was a denial of service (DOS) attack [**?**]. During the DOS attack several processes that continuously requested resources from NanoHTTPD were launched from two separate machines. The attack escalated with about 100 new processes created every second. It continued until NanoHTTPD could no longer process valid user requests.

Each fault was triggered by a specific URL request. For example, the memory leak fault was triggered by accessing the "`/memleak`" resource via a web browser. Faults were triggered after a minimum of 1 minute of fault-free execution.

The entire case study took about three weeks to execute. Most of that time was spent executing fault-free experiments and the memory-leak experiment. While all other fault experiments took only a few minutes, the memory-leak fault took several hours before NanoHTTPD failed irreversibly.

Given these 3 data sets, the fitness of the chromosome was calculated using the following function:

$$f(x) = \frac{a}{b} + \left(1 - \frac{c}{d}\right)$$

Where x is the chromosome being evaluated, a is the number of points from the `normal set` classified as inside the hull, b is the total number of points in the `normal set`, c is the number of points from the `error set` classified as inside the hull, and d is the total number of points in the `error set`. The function will return a maximum value of 2.0 if every point is classified correctly and a minimum value of 0.0 if every point is misclassified. Note that this fitness function weights the false-positive and false-negative rates equally.

Using the error data in selecting a set of metrics alters the goals of Aniketos slightly. While this does not completely transform it from an anomaly detection approach to a signature based approach, it certainly uses the faults as a guide. As one of the goals of this work was to evaluate the performance of expertly chosen metrics, this modification is acceptable.

Once collected, the three data sets had to be randomly subsampled because at their original size the QHull library took too long to compute the convex hull. The original `training set` contained 604,800 sample points, and was randomly subsampled down to 18,000. This dramatically decreased the hull computation time from several

hours to several minutes in the average case. Note that a new hull has to be computed for each chromosome in a population. Therefore, without subsampling, an application of search algorithms would have been infeasible.

However, even when using a reduced set, the time to compute the convex hull varied from a few seconds to several minutes. Therefore, any evaluation that took over 10 minutes was stopped and that chromosome was given a fitness value of 0. This effectively bred out slow chromosomes and consequently the evaluation sped up with each generation.

To further speed up the evaluation process it was distributed over several computation nodes. Five servers with 16 cores each were available for our use. Because the servers are a shared resource, the number of available computation nodes varied between 30 and 80. At its peek performance, the algorithm was able to evaluate a generation every nine minutes, on average.

Because the QHull library was written in `C`, the evaluation function was wrapped in a `C` program that took the chromosome and 3 data sets as input and produced the value of the chromosome as the output. The main controller was implemented in `Ruby` and used the provided `SSH` library to distribute the work across several machines.

The `Ruby` process was initialized with the required information to form an SSH connection to each of the computation nodes. Then, during evaluation, a pool of available nodes was maintained. To evaluate a chromosome, a node from the pool was chosen. Given the node, Ruby created a `green thread`, meaning a thread managed by the `Ruby interpreter`, and used the `SSH` library to connect to the machine that hosted the computation node. Once the connection was made, the `Ruby` process created a new process on that node and executed the `C` program used to evaluate a chromosome. The process was set to timeout after 10 minutes. When the computation was complete, or stopped due to a timeout, the reference to that node was placed back into the pool to be reused.

In addition to the time constraints, not all chromosomes produced a `training set` that could be turned into an 8 dimensional hull. The `QHull` library expects the data to have dimensionality, meaning that the produced hull is expected to have a non-zero width in every dimension. Given the nature of the produced solution it is likely that a chromosome may reduce the original 21-dimension `training set` so that the 8-dimension training set breaks the dimensionality constraint. This occurs when a metric has no width, or two or more metrics are correlated with one another. Any chromosome that produced a set that could not be used to create a convex hull was replaced by a new one. The new chromosome was computed randomly if it replaced a chromosome during the first generation and bred if it replaced a chromosome during subsequent generations.

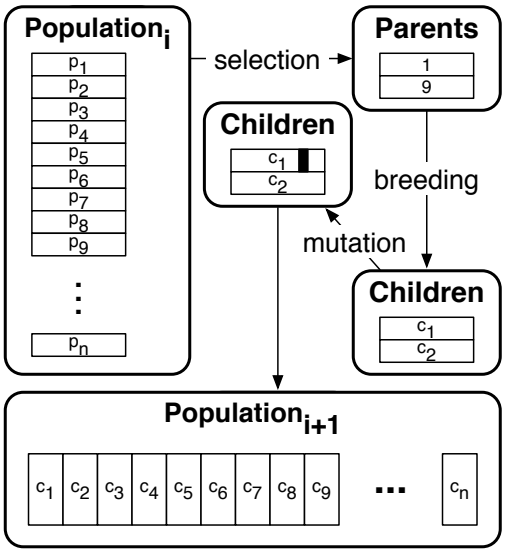

Figure 5. The breeding process to produce a new ($i + 1$) population. Parents are selected via roulette wheel selection and bred 95% of the time. The children chromosomes are then mutated 1% of the time before being placed into the next generation.

D. Generating a New Population

A new generation of 100 chromosomes is bred via the process in Figure 5. Parents are chosen via roulette wheel selection. A fitter chromosome will have a higher probability in being selected. Once selected and bred, chromosomes are placed back into the population so that they can be selected again.

Two selected chromosomes are bred with a probability of 95%, otherwise they are passed into the new population unchanged. The breeding function is a modified single cross-over function. Given that the `root node` points to 8 subtrees, if a traditional single cross-over function was used, most of the changes would occur in the metrics with many branches, leaving metrics that contained a single sensor unmodified. For example, consider a chromosome in which 6 out of 8 metrics led to a single sensor, while the other two had complicated functions with 20 unique branches each. In this case, the probability of choosing a cross-over point that modified one of the 6 single sensor metrics would be only 12%.

The modification added to the single cross-over function is that two metrics, one from each chromosome, are first chosen at random. These represent one of 8 subtrees directly under the `root` node. Once the subtrees are selected a cross-over operation is performed using those subtrees. The links from the `root node` to either of the subtrees are considered during the cross-over, therefore an

		Training Sets			
		1	2	3	average
Arithmetic	score	1.990	1.993	1.988	1.990
	false positive	1.0%	0.6%	1.2%	0.9%
	false negative	0.0%	0.1%	0.1%	0.1%
Subset Selection	score	1.992	1.987	1.992	1.990
	false positive	0.8%	1.3%	0.8%	1.0%
	false negative	0.0%	0.0%	0.0%	0.0%
Random Arithmetic	score	1.987	1.962	1.971	1.973
	false positive	1.3%	3.8%	2.9%	2.7%
	false negative	0.0%	0.0%	0.0%	0.0%
Random Subset Selection	score	1.973	1.987	1.986	1.982
	false positive	1.4%	1.3%	1.4%	1.4%
	false negative	1.3%	0.0%	0.0%	0.4%
Handcrafted$_1$	score	1.837	1.918	1.870	1.875
	false positive	4.0%	3.7%	4.1%	3.9%
	false negative	12.4%	4.5%	9.0%	8.6%
Handcrafted$_2$	score	1.682	1.683	1.683	1.683
	false positive	0.3%	0.6%	0.3%	0.4%
	false negative	31.4%	31.2%	31.4%	31.3%

Table I. The fitness of the best chromosomes discovered using various search techniques. Handcrafted$_1$ is the result when using 7 metrics provided by a software engineer. Handcrafted$_2$ is the result when using 6 metrics provided by a system administrator.

entire tree can be exchanged.

Once the new chromosomes are generated, they are mutated with the probability of 1%. If a mutation operation occurs, a node in the chromosome is chosen at random and modified. If the selected node is a `binary operator node`, it is replaced by a new `binary operator node` chosen at random and based on the provided grammar. If the selected node is a `negation operator`, the node is removed from the tree. If the selected node is a `terminal node`, it is replaced by a new `terminal node` chosen at random and based on the grammar.

IV. Search Algorithms Evaluation

Table I illustrates the results of several search experiments and compares these results to two handcrafted solutions. Due to the time constrains described in Section III, the training set had to be sub-sampled. In order to verify the results, three different sub-sampled sets were used in these experiments. Each training set was generated by choosing 18,000 points randomly from the full training set.

Four different search experiments were conducted as part of this work. All experiments were stopped after 12 hours of execution and the best solution found by each is presented in Table I. The first, labeled `Arithmetic` in

```
TERMINAL => {1|2|3|...|21}

EXP => {TERMINAL}

S => {s EXP EXP EXP EXP
      EXP EXP EXP EXP}
```

Figure 6. A modification of the original grammar from Figure 3, changes the problem from a GP to a GA.

Software Engineer
Total CPU Time
Thread Count
TCP Bytes Read + TCP Bytes Written
Number of Open TCP Accept Sockets
Loaded Class Count
Heap Memory + Non-heap Memory
Average Stack Depth

System Administrator
Total CPU Time
Thread Count
TCP Bytes Read + TCP Bytes Written
Number of Open TCP Accept Sockets
Number of Closed TCP Accept Sockets
Daemon Thread Count

Figure 7. Metrics developed by the domain experts. The software engineering expert is one of the the authors of this paper. The system administrator expert is the manager of the computing resources for Drexel University's Computer Science department.

Table I finds the solution via the GP search described in Section III. The second, labeled `Subset Selection`, solves the subset selection problem using the same algorithm as the GP with a different grammar. The grammar is altered such that the root node expands into 8 terminal nodes, thus changing the problem from a GP to GA. This new grammar is presented in Figure 6. The third experiment, labeled `Random Arithmetic`, creates several random chromosomes via the process described in Section III-B and simply selects the fittest chromosome as the solution. The fourth experiment, labeled `Random Subset Selection`, creates several random chromosomes using the same grammar used by the `Subset Selection` experiment (Figure 6). Results using two different handcrafted chromosomes are also presented in Table I.

This work began by having two domain experts, a software engineer and a system administrator, choose up to 8 metrics from a list of 21 sensors, presented in Figure 7.

	3D	4D	5D
score	1.855	1.956	1.988
false positive	2.6%	0.8%	1.2%
false negative	11.9%	3.6%	0.0%

Table II. The best fit subset of 3, 4, and 5 sensors found via exhaustive search.

Notice that while there is some overlap between these two sets of metrics, they were created from two different perspectives. After discussing the selection with the system administrator, it was clear that he was far more concerned with how a failure in the web server would affect all other services he was managing. Therefore, he focused on metrics that expressed server load. The software engineer was more concerned with detecting failures internal to NanoHTTPD. Therefore, he chose metrics such as `Average Stack Depth` and `Total Memory`.

Given that the metrics chosen by the domain experts varied significantly in their results, we decided to automate the metric selection process. Because both experts chose a combination of sensors for at least one of their metrics, the GP search approach was chosen first.

When running the GP process we were surprised by how well it performed. The best GP result was found after only twelve generations. This led us to speculate that using the GP may be excessive and not necessary for finding a good solution. It appears that the problem in not a "needle in a haystack" type problem that GPs are so good at solving. Therefore, we decided to simplify the problem to subset selection.

The best subset selection solution was found during the 36^{th} generation. However, good solutions appeared as early as the 8^{th} generation. This led us to further explore the search space via random search.

While random search was not able to find solutions that were as good as those found via the guided search techniques, it was able to get surprisingly close. Upon further examination, roughly 0.3% of the solutions explored by random search had a value greater than 1.95. Each random search experiment explored about 3,500 solutions.

Having observed such good results using random search, we wanted to further investigate the search space. To that end, we conducted three exhaustive-search experiments. Taking a single subsample as the training set, we computed the value of every possible combination of 3, 4, and 5 metrics. More metrics would be impractical to compute. The best solutions for each are presented in Table II. Histograms demonstrating the break down of all solutions are presented in Figure 8. As expected, higher dimensions provided better results. However, the

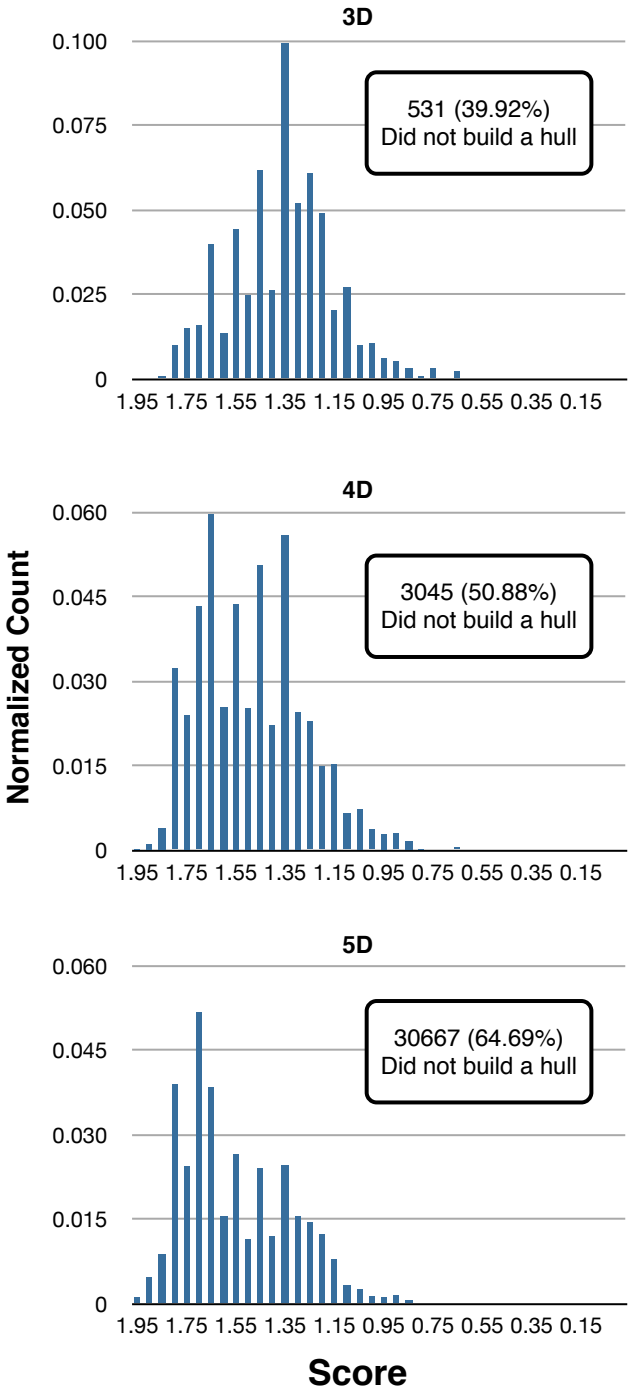

Figure 8. Normalized histograms showing the values of all subsets of 3, 4, and 5 sensors.

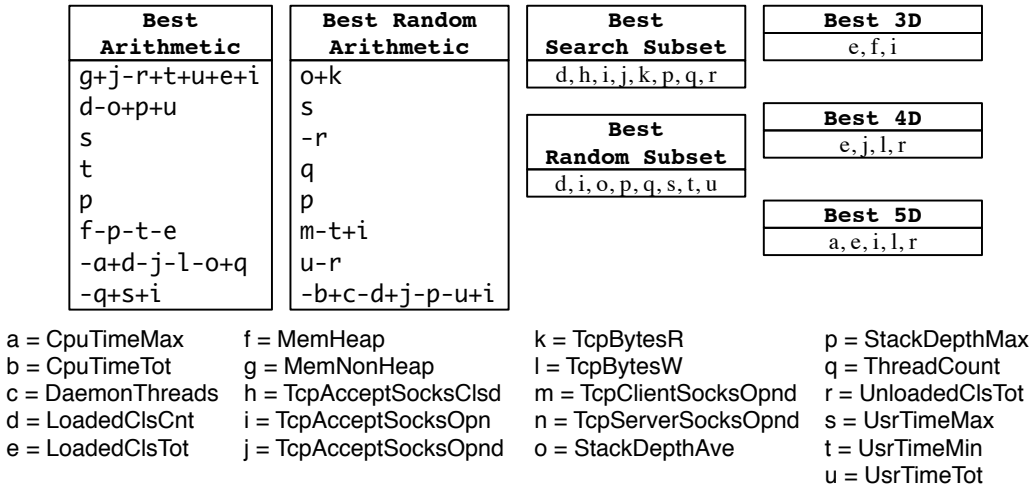

Figure 9. The most fit solutions found by the various search algorithms discussed in this paper.

high quality of the results when using only 5 metrics was a surprise. Since 5 metrics produced such good results it is not surprising that the search algorithms were able to find equally good solutions in 8 dimensions.

In addition to comparing the quality of the results it is also interesting to compare the best metrics chosen during each experiment. Figure 9 summarizes the best solutions found in each case. While several of the sensors can be observed in most solutions, there is no single sensor that stands out as the most important. This is probably because some sensors are correlated with one another and therefore can be used interchangeably. For example, each new request causes NanoHTTPD to start a new thread and open a new TCP socket. Therefore, the `ThreadCount` and `TcpAcceptSocksOpn` sensors are highly correlated.

These experiments where repeated with `training sets` as small as 500 points. In all cases the results were analogous to those reported here. However, these experiments took 4 hours, as opposed to 12 hours, to search through as many solution as the experiments using training sets of 18,000 points.

V. Conclusions and Future Work

This paper presents a search-based approach for deriving a set of metrics from a larger set of sensors. These metrics are intended to be used in a computational-geometry technique for run-time fault detection in an autonomic computing framework. The technique treats run-time measurements as points in n-dimensional space and constructs a convex hull to represent the normal execution of the application being monitored. Then, during fault detection, it categorizes new points as anomalous when they fall outside of the convex hull. This technique is limited by the QHull library to 8 metrics.

A case study using a Java-based web served name NanoHTTPD was conducted. During the case study, search-based techniques found solutions that were superior to those chosen by the domain experts. However, the solutions were found with relatively little computation and, upon further exploration, it became evident that many good solutions exist in the space being searched. This is best observed by the fact that an exhaustive search of every 5 sensor combination was able to find a solution equal to those found by the various search techniques described in this paper.

This observation can be explained because many of the sensors are dependent with one another and are therefore redundant. This is likely to be a consequence of the simplicity of NanoHTTPD. NanoHTTPD is a bare-bones web server that does not have any advanced functionality for managing its resources.

The results presented in this paper are promising as they demonstrate the relative effectiveness of search-based software-engineering techniques as compared to solutions proposed by domain experts. Even using a system as simple as NanoHTTPD, the computer significantly outperformed the domain experts. The search algorithms were able to discover non-obvious esoteric relationships between sensors. We expect the difference between the quality of human and computer-generated solutions to increase as more sophisticated applications begin using this technique.

In future work we will test this hypothesis by employing the techniques presented in this paper on industrial-strength software systems. These will include more complex web servers that perform some form of resource

management, application servers such as TomCat [**?**] and GlassFish [**?**], as well as standalone desktop applications.

It is also worth noting that the fitness function used in this work weighs the false-positive and false-negative rates equally. While this may be useful in the general case, there are scenarios in which a failure may be more costly than false fault detection. In such scenarios the fitness function would need to be adjusted to take that cost into account.

Acknowledgments

The authors would like to thank the Lockheed Martin Corporation for sponsoring this research.