Learning States and Rules for Time Series Anomaly Detection

Stan Salvador, Philip Chan, and John Brodie
Dept. of Computer Sciences Technical Report CS-2003-05
Florida Institute of Technology
Melbourne, FL 32901
{ssalvado, pkc, jbrodie}@cs.fit.edu

ABSTRACT
In this paper we investigate machine learning techniques for discovering knowledge that can be used to monitor the operation of devices or systems. Specifically, we study methods for generating models that can detect anomalies in time series data. The normal operation of a device can usually be characterized in different temporal states. To identify these states, we introduce a clustering algorithm called Gecko that can automatically determine a reasonable number of clusters using our proposed "L" method. We then use the RIPPER classification algorithm to describe these states in logical rules. Finally, transitional logic between the states is added to create a finite state automaton. Our empirical results, on data obtained from the NASA shuttle program, indicate that the Gecko algorithm is comparable to a human expert in identifying states and our overall system can track normal behavior and detect anomalies.

Categories and Subject Descriptors
I.2.1 [**Learning**]: Induction.

General Terms
Algorithms, Verification.

Keywords
clustering, segmentation, anomaly detection, rule generation, time series data, state transition logic, expert systems.

1. INTRODUCTION
Motivation. An expert system contains a knowledge base that allows it to reason proficiently in a specific domain. These knowledge-intensive systems are often used to help humans monitor and control critical systems in real-time. For example, NASA uses expert systems to monitor various devices on the space shuttle. However, populating an expert system's knowledge base by hand is known to be a time-consuming process. In this paper we investigate machine learning techniques for generating knowledge that can monitor the operation of devices or systems. Specifically, we study methods for generating models that can detect anomalies in time series data.

The normal operation of a device can usually be characterized in different temporal states. Segmentation or clustering techniques can help identify the various states, however, most methods directly or indirectly require a parameter to specify the number of segments/clusters in the time series data. The output of these algorithms is also not in a logical rule format, which is commonly used in expert systems for its ease of comprehension and modification. Furthermore, the relationships between these states needs to be determined to allow tracking from one state to another and to detect anomalies.

Problem. Given a time series depicting a system's normal operation, we desire to learn a model that can detect anomalies and can be easily read and modified by human users. We investigate a few issues in this paper. First, we want a clustering algorithm that can dynamically determine a reasonable number of clusters, and hence the number of states for our purposes. These states should be relatively comparable to those identified by human experts. Second, we would like these states to be characterized in logical rules so that they can be read and modified with relative ease by humans. The rules should be general enough to cover normal data points not seen in the training data. Third, given the knowledge of the different states, we wish to describe the relationship among them for tracking normal behavior and detecting anomalies.

Approach. To identify states, we introduce Gecko, which is able to cluster time series data and automatically determine a reasonable number of clusters (states). Gecko consists of a top down partitioning phase to find initial sub-clusters and a bottom-up phase which merges them back together. The appropriate number of clusters is automatically determined by what we call the "L" method. To characterize the states in logical rules, we use the RIPPER [2] classification rule learning algorithm. Since different states can sometimes overlap in the one-dimensional input space, additional attributes are derived to help characterize the states. To track normal behavior and detect anomalies, we construct a finite state automaton (FSA) with the identified states.

Our main contributions are: (1) we demonstrate a way to perform time series anomaly detection via automatically generated states and rules that can easily be understood and modified by humans; (2) we introduce an algorithm named Gecko for clustering time series data and the L method for dynamically finding a reasonable number of clusters--the L method is general enough to be used with other hierarchical divisive/agglomerative clustering algorithms; (3) we integrate RIPPER and state transition logic to generate a complete anomaly detection system; 4) our empirical evaluations, with data from NASA, indicate that Gecko performs comparably with a NASA expert and the overall system can track normal behavior and detect anomalies.

The next section gives an overview of related work. Section 3 provides a detailed explanation of our system, which includes the

components: Gecko (clustering), RIPPER (rule generation), and state transition logic. Section 4 contains experimental evaluations of the component algorithms as well as the overall anomaly detection system, and Section 5 summarizes our study.

2. RELATED WORK

Clustering Algorithms. There are four main categories of clustering algorithms: partitioning, hierarchical, density-based, and grid-based. Partitioning algorithms, for example *K*-means, and PAM [10], iteratively refine a set of *k* clusters, which could take a long time to converge. Density-based algorithms, e.g., DBSCAN [4] and DENCLUE [7], are able to efficiently produce clusters of arbitrary shape and are also able to handle noise. If the density of a region is above a specified threshold, it is assigned to a cluster, otherwise it is considered to be noise. However, sharp spikes in time series data are sometimes important features and could be incorrectly determined to be noise by a density-based clustering algorithm. Hierarchical algorithms can be agglomerative and/or divisive. The agglomerative (bottom-up) approach repeatedly merges two clusters, while the divisive (top-down) approach repeatedly splits a cluster into two. ROCK [6] and Chameleon [8] are hierarchical algorithms that differ mostly in their similarity functions, which favor spherical and non-spherical clusters (respectively). Grid-based algorithms such as WaveCluster [12] reduce the clustering space into a grid of cells which enables efficient clustering of very large datasets. This is useful for clustering a large amount of very concentrated data, but not for one-dimensional time series data. Existing clustering algorithms are not designed to cluster time series data. Our Gecko algorithm is a hierarchical algorithm that clusters time series data by adding constraints to the merging and splitting of clusters.

Determining the Number of Clusters. Clustering algorithms usually require a stopping condition to be explicitly given as a parameter(s). This parameter is either *k* (the number of clusters to return) or some other ad-hoc algorithm specific parameter. Forcing a user to specify the number of clusters to return requires either detailed pre-existing knowledge of the data set, or time-consuming trial and error. The majority of existing methods to automatically determine the number of clusters to return involve a brute-force method of repeatedly running a clustering algorithm over a range of the parameter *k*, and then very inefficiently assessing the quality of each set of clusters that is produced. The Gap statistic [14] is an example of such a brute-force method that is far too inefficient to be practical in data sets of non-trivial size. The existing methods also favor spherical clusters and are less suitable to time series data. Two methods were proposed to determine the number of non-spherical clusters to return: cluster stability [11] and Monte Carlo cross evaluation [13]. However, these methods have not yet been shown to work well in practice.

Segmentation Algorithms. Segmentation algorithms usually take time series data as input and produce a Piecewise Linear Representation (PLR) as output. PLR is a set of consecutive line segments that tightly fit the original data points. Segmentation algorithms are somewhat related to clustering algorithms in that each segment can be thought of as a cluster. However, due to the linear representation bias, segmentation algorithms usually produce a finer grain partitioning than clustering algorithms, so the clusters that are produced may not represent natural clusters. There are three common approaches [9]. First, in the Sliding Window approach, a segment is grown until the error of the line is above a specified threshold, then a new segment is started. Second, in the Top-down approach, the entire time series is recursively split until the desired number of segments is reached, or an error threshold is reached. Third, the Bottom-up approach starts off with *n*/2 segments, the 2 most similar adjacent segments are repeatedly joined until the desired number of segments is reached, or an error threshold is reached. Bottom-up Segmentation (BUS) will be evaluated with our proposed ideas.

Determining the Number of Segments. Segmentation algorithms are most commonly used to create a fine approximation of a time series for compression, and are not interested in finding genuine clusters. Methods have been developed to automatically determine the maximum number of segments that can be created without over-fitting the data [15], but no current segmentation algorithms attempt to find a relatively small set of segments that would correspond to natural clusters.

Rule Generation. To characterize the states into logic rules, we can use classification rule or association rule algorithms. However, association rule algorithms generate *all* associations that exceed a user-specified confidence and support, which are not necessary since we just need to characterize the states. Furthermore, we want a succinct characterization so that the overhead of matching the states is relatively small.

Anomaly Detection. Much of the work in time series anomaly detection relies on models that are not easily readable and hence modifiable by humans for tuning purposes. Examples include a set of normal sequences [3] and adaptive resonance theory [1].

3. APPROACH

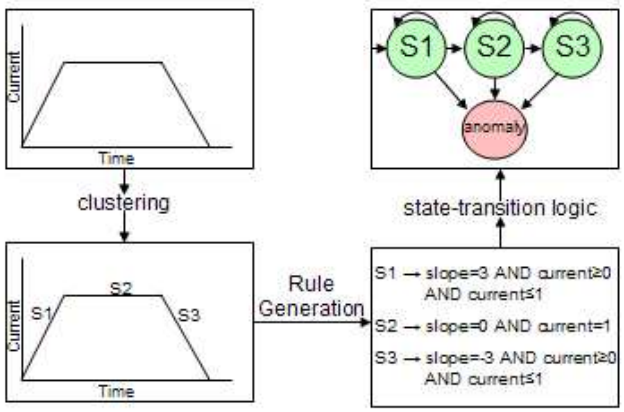

Figure 1. Main steps in time series anomaly detection.

The input to our overall anomaly detection system is a time series signature (such as the current vs. time graph at the top left corner of Figure 1) which is considered to be a "normal." The output of the overall system is a set of rules that implement state transition logic on an expert system, and are able to determine if other time series signatures deviate significantly from the learned signature. Since the learned signature is the "normal" model, any significant deviation from this model is considered an anomaly. The overall architecture of the anomaly detection system consists of three parts: clustering, rule generation (characterization), and state transition logic. The clustering phase is performed by the newly developed clustering algorithm "Gecko," which is able to identify

distinct phases in a time series signature. After the clustering algorithm identifies all of the major phases or states of the time series, rules are created for each state by an implementation of the RIPPER algorithm. The rules and additional logic for transition between the states constitute a finite state automaton, which is the expert system. It raises an anomaly if the data stream being monitored differs significantly from the learned (normal) model.

In order to prevent data signatures that are shifted to the left or right (shifted in the time dimension) from causing false anomalies, time is not considered when creating rules to describe each state. This also allows steady-state conditions (long horizontal phases which usually signify on or off) to occur for an indefinite amount of time without triggering anomalies. Time should not be used in classification because it is too inflexible to expect all states to start and end at the same time or for all states to always last for exactly the same amount of time. In other words, the actual time value where states begin and end is not important, only the relative ordering of the states and the normal attribute (non-time) values of the states are important. As an example, state S2 in Figure 1 would be able to continue indefinitely without triggering an anomaly, (it is a steady-state condition) if time is not used for state S2's classification. However, using only a single original data measurement (such as current in Figure 1) makes it difficult to create rules that unambiguously classify data points. For example, if 'current' is the only attribute value that is used during classification, all data points in state S1 and S3 would be indistinguishable from each other because they have identical current values. An ambiguous classification would make state transition logic difficult because the state that a data point belonged to would not be clear. To help distinguish between the different states, pre-processing of the raw data is performed to generate new attributes. The new derived values that are generated are the slopes and the 2nd derivatives of all original measurements. Using the slope of the current, along with the original 'current' measurements, states S1 and S3 can be easily distinguished between each other. We discuss the main parts in the next three sections.

3.1 Gecko – Data Clustering

Gecko is a newly developed clustering algorithm that is able to cluster time series data. While segmentation algorithms typically create only a fine linear approximation of time series data, Gecko divides time series data into "genuine" clusters. This optimum number of clusters is automatically determined by the algorithm and does not require user input. This is a departure from other clustering and segmentation algorithms which require either the number of clusters or some arbitrary threshold to determine how many clusters or segments should be produced.

3.1.1 The Method

Gecko uses a 2-pass method that is a combination of both agglomerative and divisive hierarchical clustering. The first is a top-down pass that partitions the data into a large number of sub-clusters. This is followed by a bottom-up pass that merges the sub-clusters back together. The first top-down pass determines all of the potential boundary areas between clusters, which then enables the second bottom-up pass to focus only on the relative similarity of clusters. Note that hierarchical clustering algorithms are very similar to top-down/bottom-up segmentation. The main difference is that hierarchical clustering is more generalized than hierarchical segmentation and any number of methods can be used to determine similarity, while segmentation is limited to the error of a segment's best-fit line.

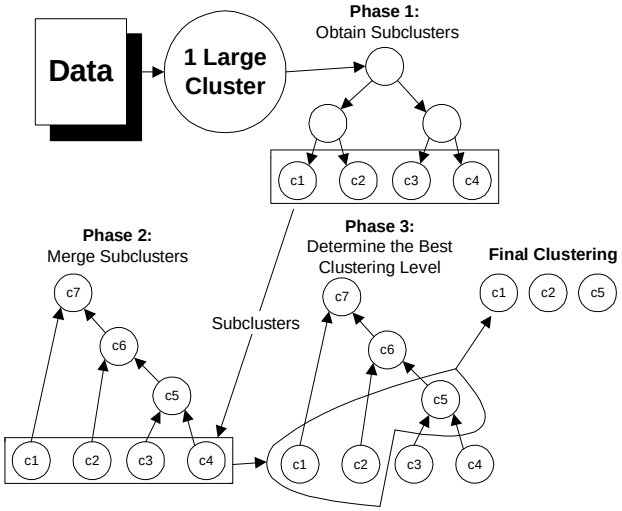

Figure 2. Overview of the Gecko Algorithm.

The Gecko algorithm consists of three phases. The first phase creates many small sub-clusters by initially putting all of the data points in one cluster, and repeatedly splitting the largest cluster until all of the clusters can no longer be divided without becoming smaller than a specified parameter *s*. The second phase takes all of the sub-clusters and repeatedly merges the two most similar clusters until all of the data is once again in the same cluster. Information about each merge is recorded in a dendrogram (tree data structure). This dendrogram contains clustering information about all clustering levels, from clusterings containing a single cluster to the initial fine grain clustering produced by phase 1. Using the information that is stored in the dendrogram, phase 3 is able to quickly determine the 'best' number of clusters that should be extracted from the dendrogram.

The Gecko Algorithm (overview)
Input: *D* // time series data
s // the minimum cluster size
Output: *c** clusters

Phase 1:
1. build a *k*-nearest neighbor graph of *D* (*k*=2**s*)
2. recursively bisect the graph until the size of each sub-cluster is between *s* and 2.2**s*

Phase 2:
3. recursively merge the sub-clusters together until only one cluster remains - a dendrogram is created

Phase 3:
4. find *c**, an appropriate number of clusters to return, by using the L method.
5. extract *c** clusters from the dendrogram and return them

3.1.2 Phase 1: Create Sub-Clusters

In the first phase, many small sub-clusters are created by a method that is very similar to the one used by Chameleon [8], with the

exception that Gecko forces cluster boundaries to be non-overlapping in the time dimension. The sub-clusters are created by initially placing all of the data points in a cluster, and repeatedly splitting the largest cluster until all of the clusters are too small to be split again without violating the minimum possible cluster size *s*.

To determine how to split the largest cluster, a *k*-nearest neighbor graph is built in which each node in the graph is a time series data point (measurements taken at a time-interval), and each edge is the similarity between two data points. Only the slopes of the original values (original sensor readings) are used to determine similarity, and not the original values themselves. Using only the slope will tend to produce sub-clusters that have constant slope, which produces sub-clusters that are as close to straight lines as possible. The *k*-nearest neighbor graph is constructed by creating an edge from every vertex to each of its *k* nearest (most similar) neighbors. The parameter *k* is not an input parameter. It is derived from *s* (smallest possible cluster size), and is defined to be 2*s*. Due to the importance of time, the *k* nearest points of a data point can be assumed to be the *k*/2 points on each size of the point according to the time axis. By using this graph the similarity between groups of points (clusters) can be determined by computing the edge cut (sum of the edges) between the two groups. Similarity between two points is defined to be ln(1.0/*distance*+1), where *distance* is the Euclidean distance (or any other distance method) between the two points. However any reasonable inverse mapping between distance and similarity can be used. If the graph is split where the edge-cut is the smallest, then the two newly separated clusters will be dissimilar to each other and have high internal similarity.

Since all boundaries between clusters are cut cleanly by the time axis with no overlap, the typically NP-hard problem of graph bisection is trivialized, and the optimal min-cut partitioning of a cluster can be quickly determined in fewer than |cluster|-1 edge-cut checks (where |cluster| is the number of data points contained in the cluster). There is no need for heuristics, because all possible edge-cut possibilities can be quickly computed with efficient data structures.

3.1.3 Phase 2: Repeatedly Merge Clusters

Phase 1 produces many small and similarly-sized sub-clusters that are as dissimilar to each other as possible. In phase 2, the most similar pair of adjacent (in time) clusters are repeatedly merged until only one cluster remains. To determine which adjacent pair of clusters are the most similar, representative points are generated for each cluster and the two adjacent clusters with the closest representative points are merged. The reduction of a cluster into a single representative point for comparison with other clusters does not adversely effect the quality of the merging decision because the clusters are internally homogeneous, and can therefore be accurately represented by a single point. This reduction causes a substantial improvement in efficiency because only *c*-1 representative points (where *c* is the current number of clusters) need to be compared to determine which adjacent pair of clusters are most similar.

The representative point of a cluster contains a value for the slope of every original attribute in the data other than time. The slope values are computed by fitting a line to all of the data points of an original attribute. Experimentation has shown that also using the original data points and second derivatives in the representative points does not to improve the quality of the clustering. So essentially, Gecko generates clusters based only on the slopes of the original data. If a human is asked to pick out several distinct phases of a time series graph, he is likely to divide the graph into flat regions and transitions between flat regions. This eyeball method of clustering is also essentially clustering by slope. Segmentation also relies exclusively on slope: if a minimum-error line (segment) is well fitted to a set of points it means that the segment has a consistent slope.

However, if raw slope values are used in the representative points, then the "distance" between clusters with slope values 100 and 101 would be the same as the distance between clusters with slope values 0 and 1. Differences in slopes that are near zero need to be emphasized because the same absolute change in slope can triple a small value, and be an insignificant increase for a large value. Relative differences between slopes cannot be measured by the percentage increase because in the preceding example, the percentage increase from 0 to 1 is undefined. Gecko uses representative values of slopes to compute the "distance" between two slopes. The representative value of slope is computed by the equation:

$$\text{Representative Slope} = \begin{cases} \ln(slope+1) & if\ slope \geq 0 \\ -\ln(-slope+1) & if\ slope < 0 \end{cases}$$

This equation emphasizes slopes near zero and decreases the effect of changes in slope when the slope values are large. Whenever a slope value is squared, its representative slope value (approximately) doubles. In the preceding example of comparing 2 pairs of clusters with slopes {100, 101} and {0, 1} the representative values of their slopes are {4.615, 4.625} and {0, 0.693}. This accurately reflects the relative difference between raw slopes and not the absolute difference. Taking the difference between the adjusted slope values gives a good distance measurement between clusters.

3.1.4 Phase 3: Determine the Best Clustering Level

Hierarchical clustering algorithms typically only keep track of the current set of clusters, and store no information about previous sets of clusters. In order to determine when to stop merging (and thus the number of clusters to return), a stopping condition needs to be explicitly specified as a parameter. The stopping condition can be either a number of clusters to return or a threshold that stops the merging when the measurement of distance (or similarity) between the last pair of clusters being joined is above (or below) some threshold. No static stopping condition is likely to produce good results across varied data sets. Some time series data sets are much more complex than others that require more clusters to accurately depict them. In addition, some data sets also contain more noise than others or have a different scaling, which leads to inconsistent results when using a constant error threshold parameter as a stopping condition. The method that Gecko uses to determine the best number of clusters takes only a small amount of analysis after a single pass of the clustering algorithm.

In order to make an intelligent decision about which number of clusters produces the best clustering, the merging process must be continued all the way to one cluster. Taking advantage of the nature of hierarchical clustering algorithms, it is possible to

efficiently store many clustering possibilities at the same time. A tree data structure can be created during the merging process in which the leaf nodes store the initial sub-clusters, and for each merge, the newly created merged cluster is represented by a node that contains pointers to the two clusters that were combined to create it. The use of such a tree structure enables efficient analysis of all clustering possibilities, which can be used to determine the best number of clusters to return. When this number is determined, the optimal set of c^* clusters can be directly extracted from the tree without any backtracking or a 2nd pass through the merging process.

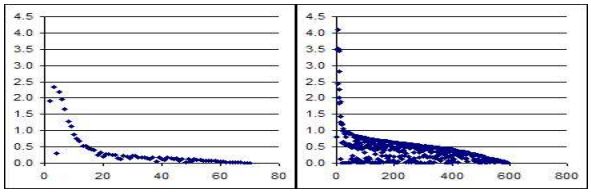

Figure 3. Sample Plots of *'# of clusters* vs. *merge distance'*.

To determine a good number of clusters to return, the distances of all merges during phase 2 are analyzed. The basic shape of the '*# of clusters* vs. *merge distance*' graph is shown in Figure 3. In this graph, the *x*-axis is the number of clusters from 2 to the number of sub-clusters generated by phase 1. The *y*-axis is the distance of the two closest clusters when there are *x* clusters. Each data-point is the distance of a single merge, and the entire graph is generated in only once pass of the clustering algorithm. This graph can vary significantly depending on the data set but they all contain a similar "L" shape curve. Figure 3 shows two such graphs generated from two very different data sets. The graph on the right side was generated from a much larger data set that contained more noise. All '*# of clusters* vs. *merge distance*' graphs have three distinctive areas: a rather flat region to the right, a near-vertical region to the left, and a curved transition area in the middle.

Starting from the right end, where the phase 2 merging process begins, there are many very similar clusters to be merged and the trend continues to the left in a rather straight line for some time. In this region, many clusters are similar to each other and should be merged.

Another distinctive area of the graph is on the far left side where the merge distances grow very rapidly (moving from right to left, which is the order that the merging occurs). This rapid increase in distance indicates that very dissimilar clusters are being merged together, and that the quality of the clustering is becoming poor because clusters are no longer internally homogeneous. If the best available remaining merges start becoming increasingly poor, it means that too many merges have already been performed and the optimal clustering has been passed.

The optimal number of clusters is therefore in the curved area, or the "elbow" (also known as the knee) of the graph. This elbow region is between the low distance merges that form a nearly straight line on the right side of the graph, and the quickly increasing region on the left side. In this area, the merges that are being performed are merges of transition clusters between the more obvious clusters. Clusterings in this region contain highly homogeneous clusters, as well as some number of transition clusters between them. Detecting at what number of clusters this region is will therefore give a good number of clusters to return that is totally dependent on the data and does not rely on static or ad-hoc parameters.

To determine the location of the transition area or elbow of the graph, we take advantage of a property that exists in these '*# of clusters* vs. *merge distance*' graphs. The regions to both the right and the left of the curved section of the graph (see Figure 3) are approximately linear. If a line is fitted to the right side and another line is fitted to the left side, then the intersection of those two lines will be in the transition area and can be used as the number of clusters to return. Figure 4 depicts an example.

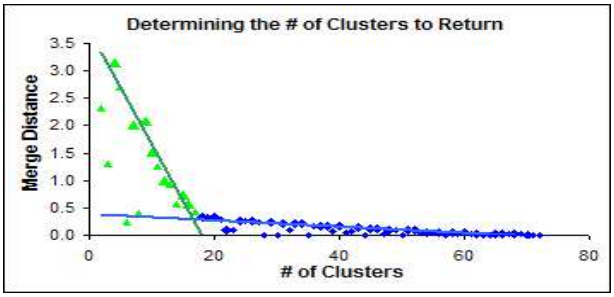

Figure 4. Finding the number of clusters by the L method.

To create these two lines that will intersect at the transition area and indicate a good number of clusters to return, we will find a pair of lines that most closely fit the curve. Figure 5 shows all possible pairs of best fit lines for a graph that contains seven data points (seven clusters were repeatedly merged into a single cluster). Each line must contain at least 2 points, and must start at either end of the data. Both lines together cover all of the data points, so if one line is small, the other is large to cover the rest of the remaining data-points. The lines cover sequential sets of points, so the total number of line pairs is not exponential, but only *numOfSubclusters*-4. Of the four possible line pairs in Figure 5, the pair that fits their respective data points with the minimum amount of error is the pair on the bottom left.

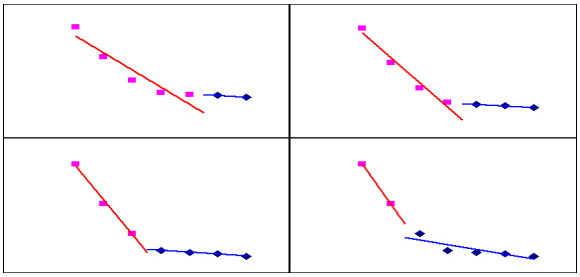

Figure 5. All possible pairs of best-fit lines.

Consider a '# of clusters vs. merge distance' graph produced by recursively merging b sub-clusters into a single cluster. The x-axis varies from 2 to b, hence there are b-1 data points (i.e., b-1 possible merges of clusters) in the graph. Let L_c and R_c be the left and right sequences of data points partitioned at $x=c$; that is, L_c has points with $x=2...c$, and R_c has points with $x=c+1…b$, where $c=3…b-2$. Equation 1 defines the total root mean squared error $RMSE_c$, when the partition of L_c and R_c is at $x=c$:

$$RMSE_c = \frac{c-1}{b-1} \times RMSE(L_c) + \frac{b-c}{b-1} \times RMSE(R_c) \quad [1]$$

where $RMSE(L_c)$ is the root mean squared error of the best-fit line for the sequence of points in L_c (and similarly for R_c). The weights are proportional to the lengths of L_c (c-1) and R_c (b-c). We seek the value of c, c^, such that $RMSE_c$ is minimized, that is:

$$c\hat{} = \arg\min_c \quad RMSE_c \qquad [2]$$

The x-intercept of the two minimum-error lines that minimize the total root mean squared error is:

$$c^* = \frac{yInt[bestFitLine(L_c)] - yInt[bestFitLine(R_c)]}{slope[bestFitLine(R_c)] - slope[bestFitLine(L_c)]} \qquad [3]$$

The x-intercept calculated by Equation 3 is used as the number of clusters to return. Another possibility not evaluated in this paper is to use $c^*=c\hat{}$.

This method to determine the number of clusters to return is general, and can also be used to determine the number of clusters in other hierarchical clustering and hierarchical segmentation algorithms (either bottom-up or top-down) that use different measures of distance or similarity at each merge. In Gecko, distance is the y-axis, however if a similar graph is produced by an algorithm that merges based on similarity, it would be flipped around the x-axis. The elbow of the curve would be found in this instance just as effectively with no modifications to the L method algorithm.

The '*# of clusters* vs. *merge distance*' graph contains a portion where the data (moving to the left) reaches a maximum and starts moving down (see Figure 4). This occurs because far too many 'natural' clusters have been merged together into one cluster at this point, that the representative slope values start to approach zero (a very long time series' slope usually begins to converge towards zero). The points that are merged after the maximum point destroy the natural "L" shape of the graph and should not be considered when fitting 2 lines to the graph.

3.2 RIPPER – Rule Generation

We have adapted RIPPER [2] to generate human readable rules that characterize the states identified by the Gecko algorithm. The RIPPER algorithm is based on the Incremental Reduce Error Pruning (IREP) [5] over-fit-and-prune strategy. The IREP algorithm is a 2-class approach, where the data set must first be divided into two subsets. The first subset contains examples of the class whose characteristics are desired (the positive example set) and the other subset contains all other data samples (the negative example set). Our implementation of RIPPER acts as an outer loop for the IREP rule construction.

The input to RIPPER is the data produced by Gecko which contains time series data classified into c^* states. RIPPER will execute the IREP algorithm c^* times, once for each state. At each execution of IREP, a different state is considered to be the positive example set and the rest of the states form the negative example set. This results a set of rules in for each state. To describe the relationship among these states, state transition logic is identified as discussed in the following section.

3.3 State Transition Logic

The upper right-hand quadrant of Figure 1 depicts a simplified state transition diagram for a signal containing just three states. The state transition logic is described by three rules for each state corresponding to each of the three possible state transition conditions on each input data point. These three rules can be summarized as the following three simple "if-then" statements:

- If input matches current state's characteristics Then remain in current state.
- If input matches the next state's characteristics Then transition to the next state.
- If input matches neither the current state's nor the next state's characteristics Then transition to an error state.

The essential element of each of these three rules is the antecedent condition, which characterizes the data points belonging to each state. This antecedent condition for each state is obtained from the RIPPER rule generation process. The state transition logic simply needs to glue together the proper antecedents to formulate the above three transition rules for each of the c^* states identified by Gecko.

Unfortunately, our state transition logic needs to be somewhat more complex. In the domain of devices and systems we are attempting to monitor, sensors may sometimes report short-term, transient, anomalous values – false alarms. In order for our approach to be more robust in handling these transients, we have added extra counting/threshold logic to the transition from a normal state to the error state. Before the error state is actually entered one of two additional criteria must be satisfied: either (1) the number of consecutively observed anomalous values must exceed a specified threshold; or (2) the total number of anomalous values observed has exceeded another threshold. Thus an anomalous condition is not annunciated unless the observed values have been improper for some length of time. We have provided similar logic for the transition from a normal state to its normal successor. That is, we will not transition to a new normal state unless the threshold number of consecutive new values for that state has been observed; or the total number of new values for that state exceeds a threshold. These 4 threshold values are parameters to our state transition logic generation process.

4. EMPIRICAL EVALUATION

The goal of this evaluation is to demonstrate the ability of the Gecko algorithm to identify states in real time series data, show that RIPPER is able to characterize these states, and also to show that our overall system is able to detect anomalies. The data used to evaluate the component algorithms and the overall system is 10 time series data sets obtained from NASA. The data sets are signatures of a valve from the space shuttle program. Each data set contains between 1,000 and 20,000 current measurements.

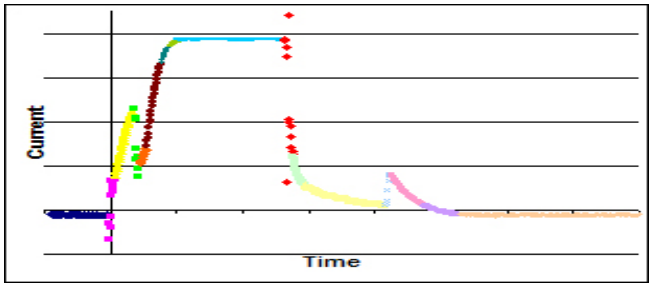

Figure 6. A sample valve time series with clusters from Gecko.

These 10 data sets contain both signatures of valves that are operating normally, and also signatures of valves that are under stress or damaged. These particular types of valves must perform within a strict tolerance, and because a valve's signature often changes over time, they must be frequently tested by NASA engineers to ensure they are working properly. The current method used to test these valves involves an expert comparing a valve's signature to a known normal signature, and determining if there is any significant variation. We would like to demonstrate in this evaluation that our anomaly detection system is able to perform the job of this human expert an automatically determine if a valve is operating normally.

4.1 Identifying States with Gecko

4.1.1 Procedures and Criteria

The experimental procedure for evaluating Gecko consists of three parts. First, Gecko and a valve expert from NASA independently cluster the 10 data sets. The expert is given an un-clustered graph of each data set and is asked to draw lines between what he thinks are clusters. This allows us to determine if the number of clusters that is automatically determined by the Gecko algorithm is comparable to the number of clusters produced by the human expert. Second, both Gecko and an existing algorithm cluster the 10 data sets. Then have a NASA engineer rate the quality of each clustering from 1 to 10, without informing him which output is from which algorithm. The existing algorithm that is used is a bottom-up segmentation algorithm (BUS). The BUS segmentation algorithm is unable to determine how many clusters to return without being given an input parameter that specifies either the number of clusters or an error threshold. It was impossible to find a static error threshold that didn't produce horrible results on over half of the data sets, so for each data set the input parameter that specifies the number of clusters to return was set to the same number that Gecko automatically generates. Forcing both algorithms to produce the same number of clusters makes for a better test of the comparative quality of the clusters they produce. Third, the valve expert is asked to go over all of the Gecko data sets that he rated in step 2, and explain his evaluation.

Gecko was run with the default parameter for each data set: minimum cluster size s=10. However, data set number 10 contained a large amount of noise and s needed to be increased to 25 to prevent phase two from considering large noisy areas to be many clusters alternatively moving up and down.

4.1.2 Results and Analysis

The first part of Gecko's evaluation was to compare the number of clusters it produced to the number produced by an expert human. A summary of the results is shown in Table 1. Gecko was able to identify a number of clusters that was within the range specified by the expert to be a 'reasonable range' (for datasets 5-10 the expert did not provide a range and we extrapolated from his clustering for that data set and his ranges for data sets 1-4). However, the human expert consistently created clusterings with fewer clusters than the Gecko algorithm. Despite the difference in the number of clusters produced, the clusterings produced are actually quite similar. Gecko generally identifies the same major clusters as the valve expert, but also produces several 'transition' clusters between the more obvious clusters.

Table 1. Clusterings produced by Gecko and a human expert.

	Gecko	**NASA Human Expert**	
Data Set	*# of clusters*	*# of clusters*	*Reasonable Range*
1	16	11	9-20
2	16	10	9-20
3	14	10	9-20
4	12	10	9-20
5	13	7	(6-15)
6	10	5	(5-10)
7	7	6	(6-11)
8	16	10	(9-19)
9	16	12	(10-20)
10	15	11	(9-16)

The next task performed by the NASA engineer was to rate the clusterings produced by Gecko and an existing algorithm. The existing algorithm used for comparison is a standard implementation of bottom-up segmentation BUS that initially creates $n/2$ segments (each segment is of length 2), and uses root-mean squared error when determining the errors of lines. For each data set, the BUS algorithm was made to produce the same number of clusters as the Gecko algorithm. Table 2 contains the scores given for all 10 pairs of clusterings.

Table 2. Clustering quality of Gecko and BUS

Data Set	**Gecko**	**BUS**
1	10	2
2	10	3
3	9	3
4	10	3
5	10	3
6	10	3
7	8	8
8	9	5
9	9	7
10	10	6
Average Score	**9.5**	**4.3**

Gecko's average score was 9.5, while the bottom-up segmentation algorithm's average score was only 4.3. In addition to the increased clustering quality, Gecko was also able to determine a suitable number of clusters with fewer input parameters. It is also interesting that the data sets that Gecko received a perfect score on (which signifies a clustering as good as the human expert's clustering) often differed notably in the number of clusters generated. For example, Gecko produced nearly twice as many clusters as the human expert for data set 5, and Gecko still got a perfect rating. This suggests that there is often a range of "very good" numbers of clusters to return, and there is no single correct number of clusters.

A major reason that Gecko performed so much better is because Gecko determines the initial segments/clusters much more

effectively than BUS. Gecko and BUS are both bottom-up hierarchical algorithms and start their merging process from an initial partitioning of the data. BUS initially partitions the data by creating as many small clusters as possible by initially putting every two points into a cluster. This means that wherever there is a very sharp cluster boundary, there is a 50% chance the BUS's initial segments will straddle the boundary. These small errors often cause more errors during the merging process and the overall clustering quality suffers. The initial partitioning produced by Gecko in its first phase, is careful make sure that all important cluster boundaries occur only on the edges of clusters.

The final part of Gecko's evaluation was a discussion with the NASA engineer about why he gave each score. This also indicated another advantage of the Gecko algorithm over BUS. In data sets with regions that have very high slopes, BUS divides them into too many clusters. This is because of the way that BUS measured the errors of lines. When clusters are merged together by keeping the total error of the best fit lines to a minimum, there is a bias favoring merging clusters together that are horizontal and have low slopes. This is because when the error of a line is computed using the root mean squared error, the vertical distance from the point to the best-fit is what is being measured. Thus, lines that are nearly vertical may seem visually to be a nearly perfect fit, but the vertical distances from the points to the line can be huge. Gecko does not suffer from this problem.

Our implementation of Gecko on a PC is able to cluster a 1,000 point data sets in 7 seconds. A 20,000 point data set takes approximately 7.5 minutes to cluster. However, sampling can be performed to increase the execution time without any effect on the quality of the output unless the user wishes to discover very small clusters that would be smoothed over by over-sampling. About 90% of the execution time is due to phase 1 of the Gecko algorithm. Building a k-nearest neighbour graph and recursively bisecting it is much more complex than the merging method used in the second phase.

4.2 Characterizing states with RIPPER

4.2.1 Procedures and Criteria

The RIPPER algorithm was tested by characterizing the clusters produced by Gecko for each of the 10 valve data sets. The accuracy of the characterization produced by RIPPER was determined in two steps: (1) for every cluster of a data set (training): create a rule that characterizes the cluster by using 90% of the cluster's data points; (2) For the remaining 10% of the data (testing): see if the cluster that these unseen data points belong to can be correctly determined by the learned rule.

To facilitate the error analysis of temporally adjacent clusters, we do not separate the unseen test data from the training data during testing. The 10% unseen data simulates minor normal variations not observed during training. We group errors into four types:

1. **Contradiction**: The exact same data point (looks the same because time is ignored) in training was classified during clustering to be in two different clusters. This is probably not a clustering error. It is most likely two different data points in flat regions that have other clusters between them. However, because time is ignored during classification, there is no way to tell the points apart. This will not be an issue during the state transition logic because the finite state automaton will only need to know if the data point is in the current or next state.
2. **Uncovered point**: The point is not covered by any of the rules. This is obviously an error. However, it is more likely to occur in this specific evaluation than in actual practice. In this test, 10% of the data was unseen during training. If several successive points in a transition region are not trained on during the training phase, it could be difficult to predict them during testing because no similar points were seen in training. The state transition logic can compensate for uncovered points.
3. **Wrong rule**: The point is covered by a rule, but not by either the correct rule, or a rule that is adjacent to it. This is a rather significant error. In a real system it would most likely indicate that the point is anomalous. An error threshold counter could be used in the state transition logic to force several 'wrong rule' points to occur in a row before signaling an anomaly. This would make sure stray 'wrong rule' points do not trigger anomalies.
4. **Poor transition**: The point is not covered by the correct rule, but is covered by a rule either immediately before or after it. This could be a transition point which is very close to belonging in two clusters at the same time. This can be dealt with in the state transition logic by having a transition threshold that requires several transition points in a row to perform a transition to a new state.

Furthermore, to determine what effect derived attributes have on the quality of the rules that RIPPER produces, the following 3 tests are performed by varying the input data points to contain: original attributes only, original attributes + slopes, and original attributes + slopes + second derivatives.

4.2.2 Results and Analysis

RIPPER was able to accurately characterize the vast majority of the test data points. The frequency of each kind of error on all data sets when using the original attribute as well as two derived attributes (slope and second derivative) is shown in Table 3.

Table 3. Classification Errors from RIPPER.

Data Set	Contrad-iction	Not Covered	Wrong Rule	Poor Transition	Total Err.
1	0%	8%	3%	0%	11%
2	0%	3%	0%	3%	6%
3	0%	2%	2%	3%	7%
4	0%	4%	1%	0%	5%
5	0%	0%	0%	0%	0%
6	0%	1%	0%	2%	3%
7	0%	2%	0%	0%	2%
8	0%	1%	1%	0%	2%
9	0%	3%	1%	0%	4%
10	0%	0.4%	0.1	0.5%	1.0%
Avg %	0.0%	2.4%	0.8%	0.8%	4.1%

The ability of RIPPER to accurately characterize data is largely dependent on the attributes of the data points. Using only the original attribute "current," over 1 in 5 data points were not able to be accurately characterized by the rules. The number of errors

goes down with each extra attribute that is added to the data. This can be clearly seen in Table 4. This occurs because each new attribute gives RIPPER the ability to more accurately classify the data. It is analogous to trying to describe your position on the earth using only latitude vs. using latitude and longitudes.

Table 4. Errors with different attributes.

Attributes	Contradiction	Not Covered	Wrong Rule	Poor Transition	Total Err.
Original	10.2%	0.0%	8.3%	4.1%	22.5 %
Orig + Slope	3.2%	1.7%	1.6%	0.9%	4.7%
Orig + Slope + 2ndDer	0%	2.4%	0.8%	0.8%	4.1%

4.3 Overall System (FSA)

4.3.1 Procedures and Criteria

The overall anomaly detection system was tested by using the rules generated by RIPPER to implement a finite state automaton on a time series stream of input. If the finite state automaton is unable to process the input stream through each state in the correct order of: "S1→S2→S3→…→S*n*", then the input stream is rejected and is considered to contain an anomaly.

In order to test whether the anomaly detection system works correctly we performed three kinds of tests: (1) **Self-tracking:** Use 90% of the data points to create rules, and then use 100% of the data fed into the expert system to see if the state transitions would trigger properly without detecting any anomalies. (2) **Normal operation**: Use all of a normal valve's data to learn its signature, and then monitor another valve that is also operating normally. This case should also not trigger any anomalies. (3) **Detecting anomalies**: Use all of a properly functioning valve's data to learn its normal signature, and then take signatures of valves that are damaged slightly and run it through the anomaly detection system. The damaged valves should trigger anomalies.

4.3.2 Results and Analysis

Self-tracking. The baseline test of the anomaly detection system is provide an incomplete sampling of a time series signature (random 90%) to characterize, will it be able to monitor the entire time series signature without triggering any anomalies. This determines if the anomaly detection system is able to detect similar time series with minor variations.

An error point in Table 5 is any point that is unexpected in the state transition logic. This means that the point is neither in the current state or the following state. Time series data often contains noise and minor variations. For this reason, anomalies must not be triggered by only a single data point that does not agree with the model contained in the finite state automaton. We use a threshold counter which only reports an anomaly if a certain number of consecutive error points are found. The last column in Table 5 shows what the minimum consecutive error threshold (*CE*) must be set to for the anomaly detection system to not report an anomaly. A value of 1 in this last column means that the anomaly detection system will correctly not report an anomaly as long as *CE* >= 1.

Table 5. Self-tracking of a time series.

Data Set	Error Pts	Min. Error Threshold
1	1.1%	2
2	0.8%	2
3	0.7%	1
4	0.5%	1
5	0.0%	0
6	0.4%	1
7	0.3%	1
8	0.2%	1
9	0.4%	1
10	1.1%	21
Avg	0.55%	4.0

In this experiment both the "consecutive transition threshold" (*CT*) and the "consecutive error threshold" (*CE*) were set to zero. This causes every possible state transition to be made and every error point throws an anomaly. This enabled easy computation of the number of error points. Data set number 10 performs poorly in this test because the FSA transitions prematurely near the end of its signature and starts reporting many anomalies, the results for this data set can be improved by increasing *CT* to prevent it from transitioning too early on a spurious data point.

Normal Operation. This test will show that the anomaly detection system's model of the normal signature is general enough to recognize that a different normal time series contains no anomalies. In this test, the anomaly detection system trained on data set 1, and then tested on data set 2. Both of these data sets are of normally operating valves that contains minor (but visible) differences. The "consecutive transition threshold" (*CT*) parameter was set to 2, and *CE* was set to 10 (minimum possible cluster size *s*=*10*). This means than two consecutive points believed to be in the next state are needed to perform a state transition and ten consecutive points believed to be errors are needed to declare that the time series contains anomalies.

The system was able to successfully transition through the states, without declaring any anomalies. Of 979 data points, 61 (2.6%) were error--they were not believed to belong to the current state, nor to be transition points belonging to the following state. However, since a consecutive number of errors greater than *CE* was never encountered, an anomaly was never triggered.

Detecting Anomalies. In this test, the two data sets containing time series signatures of valves operating normally (data sets 1 and 2) were used to develop the normal models. Each normal model was then run against the remaining data sets. Data sets 3-10 contain signatures of damaged or otherwise ill-behaved valves and should be determined to be anomalous by the anomaly detection system.

For each of the 16 tests, the anomaly detection system was able to determine that signatures contained anomalies. Additionally, the system is able to inform the user of the state number where the signature differs from the model. The state where the errors occurred varied from state 2 to state 12. Thus, the system does not only give a yes/no answer to whether a time series contains anomalies, but it is also able to explain to the user where the

anomaly occurred. Also, because the rules generated by RIPPER are in a human-readable format, the user can look at the rule for the state where the error occurred and understand exactly why the system reported the anomaly.

5. CONCLUDING REMARKS

We have detailed our approach to time-series anomaly detection by discovering and characterizing the states of a time series, and adding transition logic between these states to construct a finite state automaton, which is used to track normal behavior and detect anomalies. The proposed Gecko clustering algorithm is designed to cluster time series data, and uses our proposed L method to automatically determine a reasonable number of clusters efficiently. The rules generated for each state by the RIPPER algorithm can be easily understood and modified by humans. (Moreover, the generated rules can be in a format used by the SCL expert system shell at ICS, which is our collaborator on this NASA project.)

Our empirical evaluations have shown that the L method used by the Gecko algorithm returns a number of clusters that is similar to the number that is generated by a human expert. When the human expert was asked to rate Gecko's clusterings from 1-10, Gecko's clusterings were given perfect ratings on 6 of 10 data sets. A perfect rating signifies that Gecko's clustering is equally as good as the human expert's clustering. For comparison, the bottom-up segmentation algorithm was also tested, and was only given an average rating of 4.3. RIPPER was able to create accurate rules to describe the states in which only an average of about 4% of the data points had a possibly vague classification. Such a small number of potential errors have little adverse effect on the state transition logic's ability to correctly track a signature. Even small error and transition threshold values would be able to compensate for this small amount of errors. In fact, during the self-tracking tests of the finite state automation, nearly all of the data sets required the consecutive error threshold value to be set to no greater than 2 to correctly process the signature without detecting an anomaly. The overall anomaly detection system was able to detect anomalies in every signature that was from a 'damaged' valve, and was also able to monitor a 2nd normal valve without detecting any anomalies.

We plan to further evaluate our approach with more datasets from NASA; issues include building a model from multiple datasets collected at different times and datasets with different measurements. We plan to study how the L method performs with other hierarchical clustering algorithms. To dynamically set the thresholds used in the state transition logic, we can hold out part of the training data and find thresholds that prevent errors on the unseen portion of the data.

6. ACKNOWLDEGEMENTS

This research is partially supported by NASA. We thank Bobby Ferrell and Steven Santuro at NASA for providing the data sets, helpful comments, and evaluations. We also thank Brian Buckley and Steve Creighton at ICS for help integrating our algorithms into their SCL expert system.

7. REFERENCES

[1] Caudell, T. & Newman, D. (1993). An Adaptive Resonance Architecture to Define Normality and Detect Novelties in Time Series and Databases. In *Proc. IEEE World Congress on Neural Networks*, pp. IV166-176.

[2] Cohen, W. (1995). Fast Effective Rule Induction, *ICML*.

[3] Dasgupta, D. & Forrest S. (1996). Novelty Detection in Time Series Data using Ideas from Immunology. In *Proc. Fifth Intl. Conf. on Intelligent Systems*.

[4] Ester M., Kriegel H., Sander J., & Xu X. (1996) A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. 3rd KDD*.

[5] Furnkranz J. & Wildmer G. (1994). Incremental reduced error pruning. In *Proc. ICML*.

[6] Guha Sl, Rastogi R., & Shim K. (1999) ROCK: a robust clustering algorithm for categorical attributes. In *15th Int'l Conf. on Data Engineering*.

[7] Hinneburg A. & Keim D. (1998) An Efficient Approach to Clustering in Large Multimedia Databases with Noise. *AAAI*.

[8] Karypis G., Han E. & Kumar V. (1999) Chameleon: A hierarchical clustering algorithm using dynamic modeling. *IEEE Computer*, 32(8) pp. 68-75.

[9] Keogh E., Chu S., Hart D., & Pazanni M. (2001). An Online Algorithm for Segmenting Time Series. In *Proc. IEEE Intl. Conf. on Data Mining*, pp. 289-296.

[10] Ng R. & Han J. (1994). Efficient and effective clustering method for spatial data mining. In *Proc.VLDB*, pp 144-155.

[11] Roth V., Lange T., Braun M. & Buhmann J. A Resampling Approach to Cluster Validation.

[12] Seikholeslami, G., Chatterjee, S. & Zhang, A. (1998). WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases. *Proc. of the 24th VLDB*.

[13] Smyth, P. (1996). Clustering Using Monte-Carlo Cross-Validation. In *Proc. 2nd KDD*, pp.126-133.

[14] Tibshirani, R., Walther, G. & Hastie, T. (2000). Estimating the number of clusters in a dataset via the Gap statistic.

[15] Vasko, K. & Toivonen, T. Estimating the number of segments in time series data using permutation tests.