Towards Dependability in Everyday Software Using Software Telemetry

Kenny C. Gross* Scott McMaster+ Adam Porter+ Aleksey Urmanov* Lawrence G. Votta*
*Sun Microsystems Inc., +University of Maryland
{Kenny.Gross, Aleksey.Urmanov,Larry.Votta}@sun.com, {aporter,scottmcm}@cs.umd.edu

ABSTRACT

Application-level software dependability is difficult to ensure. Thus it's typically used only in custom systems and is achieved using one-of-a-kind, handcrafted solutions. We are interested in understanding whether and how these techniques can be applied to more common, lower-end systems. To this end, we have adapted a condition-based maintenance (CBM) approach called the Multivariate State Estimation Technique (MSET). This approach automatically creates sophisticated statistical models that predict system failure well before failures occur, leading to simpler and more successful recoveries. We have packaged this approach in the Software Dependability Framework (SDF). The SDF consists of instrumentation and data management libraries, a CBM module, performance visualization tools, and a software architecture that supports system designers. Finally, we evaluated our framework on a simple video game application. Our results suggest that we can cheaply and reliably predict impending runtime failures and respond to them in time to improve the system's dependability.

1 Introduction

Dependability is essential to modern software systems. Developers typically try to ensure dependability by monitoring key runtime variables and, when these variables exceed established limits, intervening in ways that return the variables to appropriate values. In practice, these key variables and their limits are highly constrained in their number and complexity to limit computational overhead, are determined through manual, offline analysis, and are difficult to change if system behaviors or usage patterns change. Needless to say, this process is error-prone and costly. Success often comes, if at all, only through ad hoc solutions, hand-coded by highly experienced developers, tuned and verified through trial and error.

As a result, these approaches have been used mainly on high-end, safety-critical systems such as nuclear power plant control, telephone switching, or military applications. The truth, however, is that many systems could benefit from improved dependability (e.g., e-commerce systems, network middleware, personal productivity software). Unfortunately, many of these systems are widely distributed to users with differing and unknown usage patterns making it difficult to develop appropriate dependability strategies.

In effect, our need to create dependable systems is large and growing, but our tools, techniques, and processes to cost-effectively do so lag far behind. Our research provides one first step towards understanding and improving software engineering support for dependable "everyday" systems.

1.1. Implementing Software Dependability

Although details differ from application to application, at a high level many systems achieve dependability through a continuous process of self-assessment, problem identification and corrective action.

Self-assessment. Developers add instrumentation to the system to continuously monitor its runtime state. The resulting data is called a telemetry stream. In general, telemetry data comes in two flavors: environmental and domain. Environmental telemetry is data observed or derived from the system's runtime environment. Examples include CPU and memory usage, disk activity, cache misses, and network latency. Domain telemetry, on the other hand, is application-specific. For example, telephony systems monitor "failed call attempts", while e-commerce systems monitor "average transaction time." Key challenges here include understanding what data to collect, with simple enough instrumentation so as not to unacceptably perturb the application.

Problem detection and identification. Telemetry data are monitored to determine whether a problem has or will soon occur, and in some cases what kind of problem it is. Problem detection is accomplished using threshold values or windows for key system variables. We call these values and ranges, alarm conditions. If at any time alarm conditions are exceeded, then the system is in an undesirable state and corrective action is needed. The key challenges are to identify problems early and accurately, so that there is time to take small, likely to be successful, corrective actions, while minimizing false alarms.

Corrective action: Once a problem has been identified, the system takes corrective action to fix the problem, ideally returning the system to a non-problem state. When applied at runtime, corrective actions must have a fairly immediate effect and typically involve limiting access to the system or changing the character of work already in progress in the system. A key challenge here is to determine the right corrective actions that ensure improved dependability.

This paper makes several research contributions. First, we describe a generic approach to creating dependable systems. This approach relies on sophisticated statistical techniques (MSET) to automatically model the system's underlying failure prediction modes. Next, we present the Software Dependability Framework (SDF), which provides components needed to implement our generic approach. We then use the SDF framework and MSET tools to create a simple dependable system. Finally, we evaluate the overall process on that system.

2 MSET Pattern Recognition

Successful, dependable systems must be able to (1) detect problems, (2) isolate and contain them, and (3) recover while still maintaining a high level of service. We believe that designing and implementing systems so they emit telemetry data creates a powerful framework for improving software dependability. This will only be true, however, if we can somehow use the telemetry data to reliably predict impending failure.

As mentioned earlier, we leverage a Condition Based Maintenance (CBM) approach called the Multivariate State Estimation Technique (MSET) [Gross et al., 2002, Cassidy et al., 2002] to automatically model and predict impending system failures based on telemetry data.

The general CBM approach has been used successfully in complex mechanical systems. These strategies continuously monitor telemetry data generated from transducers measuring physical variables, e.g. distributed temperatures, vibration levels, fluid flow rates and pressures, motor currents, etc. in order to understand the "health" of the system. If a group of telemetry producing machines can be operated to failure, then we can use analytical methods to identify one or more variables that are strongly correlated with residual life. These "predictor variables" may then be used for real-time health monitoring to proactively annunciate impending failure before it occurs so that the evolving degradation can be terminated and potentially costly consequences avoided.

This basic rationale for CBM methods that exploit the Multivariate State Estimation Technique (MSET) is as follows: In general, if we can identify single predictor variables that are correlated with residual life, we can designate a threshold value for each such variable beyond which we actuate an alarm or trigger an automated corrective action for asset protection, safety margin, or simply to maintain acceptable QoS. These predictors are typically application specific. For instance temperature thresholds in a nuclear plant that actuate the Plant Protection System and automatically insert control rods; or call-volume thresholds for a telephony switching system that initiate automatic rerouting of new incoming calls to avoid system overload events.

If, instead, we can identify pairs of variables that in combination have a good bivariate correlation with residual life, then we can often improve the "power" of the approach[1]. An example might be a machine for which a combination of high temperature and high vibration could be much more highly correlated with time-to-failure than either temperature or vibration separately.

[1] The power of an algorithm is a measure of its sensitivity in detecting "real" events, thereby enhancing asset protection or safety margin, while avoiding false alarms, which in turn enhances availability, operating efficiency, and return on investment (ROI).

In fact, power will generally continue to improve as long as we continue to find new variables that in combination have some predictive power as to the health of the system.

The MSET approach was originally developed for proactive health monitoring of complex engineering systems in the commercial nuclear industry [Gross et al., 1997, Singer et al., 1997]. As part of this research we have adapted it to the telemetry and analysis of complex software systems. MSET has been beneficially applied to mitigate complex latch-contention phenomena in large OLTP enterprise servers [Cassidy et al., 2001] and to proactively detect the onset of software aging phenomena in large, multi-user web servers [Gross et al., 2002].

The prior work however, has several limitations that we are now beginning to address. First, applying MSET was extremely labor-intensive. We had no tools or libraries that would have allowed us to implement the approach quickly. Second, the problems these systems were experiencing centered on "software aging" – basically resource leakage. These phenomena generally express themselves over periods of days and weeks. For many systems, for example those involving human controllers, we also need to handle events on the scale of minutes and seconds (or faster).

3 The Software Dependability Framework

3.1 Overview

To support the development of dependable systems, we have constructed and evaluated a software dependability framework (SDF). The SDF provides several components and facilities for implementing the generic dependability strategy described in Section 1.1. These facilities include:

Instrumentation and data management libraries: Instrumentation is used to identify the true dynamic behavior of programs. This instrumentation is intended to be lightweight from the point of view of individual programs, configurable at runtime, and to the largest extent possible, transparent to the application programmer. Also, since we will often instrument multiple deployed program instances, we also provide libraries for transferring and aggregating data from multiple sources.

Statistical modeling tools: These tools turn raw telemetry data into information that we can use. This happens in two ways. Initially, we use MSET off board, not-in-real-time to build models (i.e., learn data patterns) that predict impending failure reliably. Later, these models are dynamically linked back into the running system to do onboard, real-time problem detection. Of course, both uses can happen simultaneously: e.g., existing models used onboard, while at the same time sensitivity parameters are tuned and optimized off board.

Corrective action strategy support tools: Once we can proactively detect problems, the next step is to correct them. Corrective action strategies will be highly application-

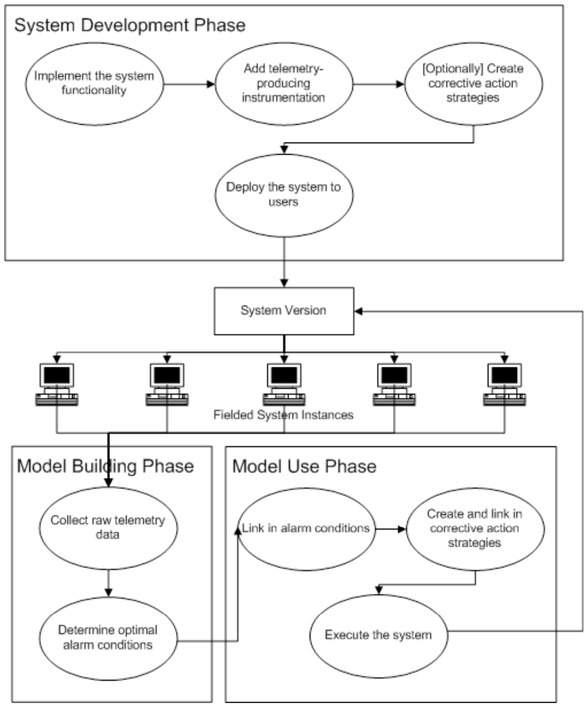

System Development Phase
Implement the system functionality
Add telemetry-producing instrumentation
[Optionally] Create corrective action strategies
Deploy the system to users
System Version
Fielded System Instances
Model Building Phase
Collect raw telemetry data
Determine optimal alarm conditions
Model Use Phase
Link in alarm conditions
Create and link in corrective action strategies
Execute the system

Figure 1: High Level View of the Telemetry Process

specific. Still, once defined, they need to be validated and tuned. We have created visualizations to support this.

A software architecture: To glue everything together, we defined a cohesive software architecture that identifies the necessary components, interfaces, and protocols expected by the rest of the framework.

3.2 Process View

As depicted in Figure 1 our approach has three phases: system development, model building, and model use.

3.2.1 Phase1: System Development

Implement system functionality. Before implementing the required functionality, developers design the system according to the SDF software architecture described later in Section 3.3.1.

Add telemetry-producing instrumentation. Developers use facilities provided by the instrumentation and data management libraries (see Section 3.3.2) to create an instrumented version of the system. At runtime this instrumentation generates environmental and domain telemetry data. If developers intend to process telemetry off board, they must configure the system with the location of the processing site and the transmission frequency in mind.

Create corrective action strategies. Developers create components that encapsulate the actions to be taken when alarm conditions are detected at runtime. At this stage, systems will typically be unable to raise meaningful alarms. In this case, this step may also be done at a later time.

Deploy the system to users. The instrumented system is deployed to end-users.

3.2.2 Model Building Phase

Collect telemetry data. As it runs, each instance of the instrumented system generates telemetry data. If the system is configured to allow telemetry streaming outside the firewall for remote monitoring, the data are forwarded to one or more off board sites for processing. If security policies prevent real-time transmission of telemetry data off site, then the data are directed to a “black box flight recorder” file that may be transferred by other means for remote processing (e.g. via an automated email script, via FTP, or even manually transported out of the datacenter on removable media).

Determine model parameters. Once enough data arrives, the server(s) process it using the methods described in Sections 2 and 3. The result is a parameterized statistical model $P(t, X_1,X_2,..,X_n)$, giving the probability of runtime failure within t time units, given the current values of predictor variables $X_1,X_2,..,X_n$. For telemetry systems monitoring configurations characterized by noisy process variables, there is usually a tradeoff between the sensitivity for annunciation of process anomalies that are likely to lead to failure (and hence the length of advance warning), and the number of false alarms raised. One of our objectives is to maximize warning time while minimizing false alarms.

3.2.3 Model Use Phase

Link in specific model. After analyzing the parameterized model, developers customize the model by choosing a specific time-interval and probability threshold that will cause an alarm to be raised. For example, developers might want to trigger an alarm if the probability of failure within the next 30 seconds is greater than 0.75. When multiple models are desired, the process is repeated. Next, the customized model is coded into a software component that will be statically or dynamically linked into deployed system instances.

Link in corrective action strategies. Corrective action strategies are software components that can respond to specific alarms. When those alarms are raised, the component executes code intended to avert impending failure.

Execute the system. At runtime, certain framework components observe the system telemetry stream, raising alarms as appropriate. Corrective action strategy components can then execute in response to them. The result is a system that significantly reduces the probability of system failure. According to user and developer preferences, data collection can be continued through the model use phase so that the model may be periodically calibrated and refined.

3.3 Implementation View

3.3.1 Software Architecture

To use this framework, developers build their systems according to the SDF software architecture. Its components are described below and shown in Figure 2.

Telemetry Source. Telemetry sources generate telemetry data and pass it to the Telemetry Stream upon request. Since domain telemetry and environmental telemetry require different support facilities, we encapsulate that support in two different objects: FRU and Environment Monitor. FRUs manage domain telemetry, while Environment Monitors manage environmental telemetry.

FRU. Application components are encapsulated in software field-replaceable units (FRUs). FRUs are the focal

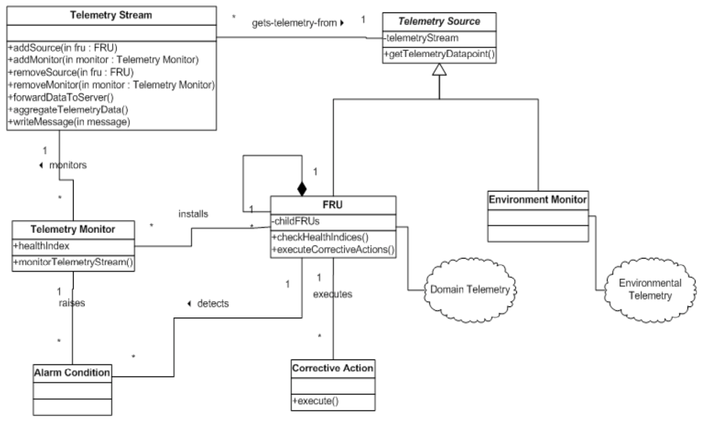


Figure 2: High Level View of the SDF Framework

point of the dependable system and, therefore, perform many functions. Besides generating telemetry data and passing it to the Telemetry Stream, FRUs also aggregate telemetry data from other FRUs that it logically contains. This allows telemetry data to be calculated at different levels of abstraction: entire system, subsystem, or an individual FRU. FRUs also poll Telemetry Monitors for alarm conditions and dispatch appropriate corrective actions as necessary.

Environment Monitor. Environment Monitors collect environmental telemetry. Examples include components that watch memory usage, thread counts, and network traffic. Because of their application-independent nature, environment monitors are generally portable from system to system within our architecture. Designers may choose and install any environment monitors that may provide insight into their system. Like FRUs, environment monitors register with and are monitored by the Telemetry Stream.

Telemetry Stream. The Telemetry Stream represents a time-synchronized aggregation of environmental telemetry from the system's host platform along with domain telemetry collected from each FRU. While a system may contain many Telemetry Sources, it has only one Telemetry Stream. This stream periodically queries all registered FRUs and environment monitors for their telemetry data. This process may be started and stopped as requested by the application.

The Telemetry Stream is also responsible for forwarding telemetry data to a telemetry server for processing. The forwarding location and the transmission frequency are set by the application. Depending on the telemetry sampling rate and the amount of data generated by telemetry sources, the telemetry stream may contain large amounts of data. To transmit this data across the network, care must be taken to minimize the size of the data stream.

Telemetry Monitor. Telemetry Monitors observe the system's telemetry stream at regular intervals and calculate a health index based on the data in the stream. In calculating the health index, a telemetry monitor may choose to incorporate simple heuristics or sophisticated statistical models. At any given time, a system may have many telemetry monitors and therefore many health indices. Each FRU attaches one or more telemetry monitors of interest and periodically checks to see if any of them have raised an alarm condition. FRUs may also detach telemetry monitors when they are no longer needed.

Alarm Condition. Alarm conditions represent the detection of problem behavior. Telemetry Monitors define alarm conditions based on their models of the system. The alarm conditions are subsequently detected by interested FRUs, which react by executing corrective actions.

Corrective Action. Corrective action objects encapsulate actions taken in response to an alarm condition. Typically, these actions are designed to return the system to a non-problem state. Practical examples include proactive restarts or system rejuvenation 4, refusing connections, and discarding work.

3.3.2 Instrumentation and Data Mgmt. Libraries

To support this architecture we have created several reusable classes and libraries. They are currently implemented in Java for the Java 2 Platform.

Telemetry Stream. The Telemetry Stream class is implemented as a singleton and maintains its own thread to minimize interference with the system's functionality. Users configure this class to control how often it aggregates telemetry data and where it forwards that data. Users may start and stop the Telemetry Stream as desired.

Telemetry Source. Telemetry Sources are objects that contribute data to telemetry streams. We have therefore defined a Java interface that controls this behavior. This interface contains a single method that returns telemetry data as a collection of name-value pairs. Telemetry Sources must also register themselves with the Telemetry Stream object.

Telemetry Monitor. Concrete objects that observe the Telemetry Stream and calculate system health indices based on that data extend the Telemetry Monitor class. Telemetry Monitors are registered with the Telemetry Stream and are notified when new telemetry data are collected. FRUs may use Telemetry Monitor methods to check for alarm conditions as desired.

EnvironmentMonitor: ProcCounter. We developed an environment monitor telemetry source that uses the Java Virtual Machine Profiling Interface (JVMPI) 1. This particular class collects and outputs method-entry events from selected classes at regular intervals as the application runs. As is desired for environment monitors, this component is completely decoupled from the core application code. Other environment monitors that capture different environmental telemetry can easily be added.

3.3.3 Statistical Modeling and Corrective Action Strategy Support Tools

The framework contains a set of tools that process telemetry data off board in order to build statistical models that reliably predict impending system failure. The outputs of these tools are optimized models that will be coded into Telemetry Monitor components. These tools are described in detail in Section 2.

The framework relies on telemetry monitor components to raise alarms when system failure is likely. Developers must also write corrective action components. These components respond to specific alarms and encapsulate actions intended to return the system to control. The framework also includes some visualization tools that help developers evaluate how well their corrective actions work.

4 Proof-of-Concept Study

As an initial proof-of-concept, we conducted a small-scale study of our framework, examining several high level questions. In particular we wanted to understand how easy was it to use our framework and whether its components are sufficient to create a simple dependable system. To answer these questions we conducted three studies: building a simple dependable system using the SDF; using telemetry data to build prediction models; and using prediction models to yield dependable system behavior.

4.1 System Development Phase

The application for this study is an implementation of a computer game similar to Tetris 2. Tetris is a well-known interactive puzzle game. The user is presented with a steady stream of falling shapes and must translate and rotate them so that they form horizontal lines as they are stacked. There are seven different shapes that comprise four blocks each. As continuous, horizontal rows are created, they are removed from the game, giving the user more space to work toward the goal of creating more rows. It is possible to complete up to four rows with one shape, and more points are awarded for completing multiple rows simultaneously. The shapes fall more rapidly as the game progresses, making it increasingly difficult for the user to correctly position them. The game is lost when the height of the highest block goes beyond the top of the game area.

Although Tetris is not a large safety-critical application, it is a useful starting point for our research. First, it mimics larger dependable systems that occasionally require human intervention, perception, cognition, and actions that are both deliberate and judicious to maintain or restore equilibrium. Blocks continuously entering the workspace represent work that must be processed. Full rows represent completed jobs, losing the game represents system overload, and the goal is to keep playing.

Tetris provides an ideal simulation of human-control tasking that continuously increases in complexity and response-time requirements until the human reaches cognitive overload conditions. As such, Tetris has been used for several years by psychologists [see, e.g. Thach, 1996] to study complex visuo-motor task situations where researchers can evaluate in a safe environment concepts involving human cognition, eye-hand coordination, decision making under conditions where new information is becoming available at a relentlessly accelerating rate, context-response linkage, and prioritizing actions to take while actively deciding what actions to not take to minimize the probability of failing (i.e. game over). Tetris also provides a good setting for evaluating quantitative metrics for estimating "residual life" of systems in which the most likely cause of failure is cognitive overload. For Tetris, regardless of a participant's skill, "cognitive overload" is eventually reached. In fact, even another computer program might have a hard time playing Tetris indefinitely (optimizing play under certain conditions is known to be NP-complete [Demaine, 2002]).

Our modified version of Tetris was written in Java. The game functionality is implemented primarily in one main class and two helper classes consisting of around 550 lines of code when taken together. The full system with reusable SDF-provided code, user-provided instrumentation and corrective action strategies, and original application code consists of approximately 1200 lines of code.

We instrumented our application to periodically output environmental and domain telemetry. For environmental telemetry we used the ProcCounter class to capture the number of times that each method in the application was called. For domain telemetry, we captured a variety of game state variables, including the height of the highest block, the number of shapes created, the block density, and the hole count (unfilled grid spaces lying under dropped blocks).

Qualitatively speaking, we experience no noticeable runtime performance degradation as a result of our instrumentation. However, in future work on other systems, we plan to quantify and minimize the performance penalty incurred by telemetry instrumentation. As we developed the SDF infrastructure classes, we also experimented with building predictive models, coding them up, linking them into the Tetris application, and developing and using different corrective action strategies.

Overall, the experience was straightforward. The key lesson is how a well-behaved FRU in our architecture should be written. Because telemetry data requests come from a separate thread (the thread run by the Telemetry Stream), a FRU must do two things to ensure that monitors watching the telemetry stream see a consistent picture of its state. First, it must prepare and return a deeply cloned snapshot of its state avoiding object references to state variables, because the FRU's state can be changed on the main thread before processing by the telemetry stream and associated telemetry monitors. Second, while preparing a telemetry response, a FRU must block any requests that may change its state, again to ensure that the telemetry data points are consistent with each other. Because of the need to block, it becomes important for the FRU to prepare the telemetry response as quickly as possible so that processing can resume on the main thread. Thus, any complicated calculations involving telemetry data should be performed by telemetry monitors and not by the FRUs themselves.

4.2 Model Building Phase

While developing the SDF framework, we explored using MSET-inspired statistical tools on software telemetry data.

4.2.1 Hypothesis

The research hypothesis behind this study is that our tools can be used on software telemetry data to automatically build reliable predictors of impending system failure.

4.2.2 Data Collection

For this study we asked 34 CS students to play Tetris for approximately one hour each. We gave them a hand-instrumented Tetris application that collected telemetry data every five seconds as the game was played. We captured procedure count data as well as Tetris domain telemetry as described in Section 4.1. After the students finished playing we collected the resulting data for analysis. In total we collected and analyzed data from 323 games. This data then became the basis for the model-building phase.

4.2.3 Threats to Validity

Threats to Internal Validity. The effectiveness of MSET may be affected by data sampling rates, or peculiarities in the data due to abnormal playing styles (e.g., players who try to lose quickly). We made no effort to examine different sampling rates, but we carefully explained our goals to the participants and examined the data for each closely to identify abnormal behavior.

Threats to External Validity. The generalization of our results is bounded by the degree to which the Tetris application represents a model of a complex and long-running system. At this stage of the research, we felt it would only complicate matters to use a more complex system. We will address this issue by repeating these studies on more complex subject programs in the future.

4.2.4 Data and Analysis

To demonstrate the usefulness of applying a simple pattern recognition approach to the telemetry data we performed a detailed correlation analysis of all games to identify and rank the variables that are most closely correlated with "residual life", defined as the time remaining before failure (game over). This analysis demonstrated that the best single variable predictor of residual life is the height of the packed pieces, and the bivariate predictors of residual life are height and hole count. In particular, we are interested in whether the models do, in fact, predict failures when and only when they occur in the data we collected. To illustrate this procedure we select one prototypic game for detailed analysis (game 1 by Player 8).

The game is over when the height metric reaches the top of the screen. We informally define the red zone as a set of states $\{X_i\}$ of the game starting from which the game will be over in less than T_z seconds (residual life) with probability greater than 0.5. Remaining states are called the green zone. We want to generate alarms when entering the red zone, while minimizing the probabilities of false alarms (Type I error) and missed alarms (Type II error).

We begin by building a simple model that relates each state of the game X_i with the probability p_i such that starting with this state the game will be over in T_z seconds. This can be done by building a linear logistic model with p_i being the proportion of cases when starting from state X_i for which the game is over in less than T_z seconds. Such a model can be fit using collected metrics of several games played by the same player. A linear logistic model is given by:

$$p_i = 1 \Big/ \left(1 + e^{-(a + b_1 x_1 + b_2 x_2 + \cdots + b_m x_m)}\right)$$

We shall seek to demonstrate the improvements obtained by going to models of higher dimensionality in terms of (1) sensitivity for detecting impending failure and (2) avoidance of false-alarms and missed-alarms. Results of the analysis follow.

In Figure 3, two metrics are plotted as time-series. The

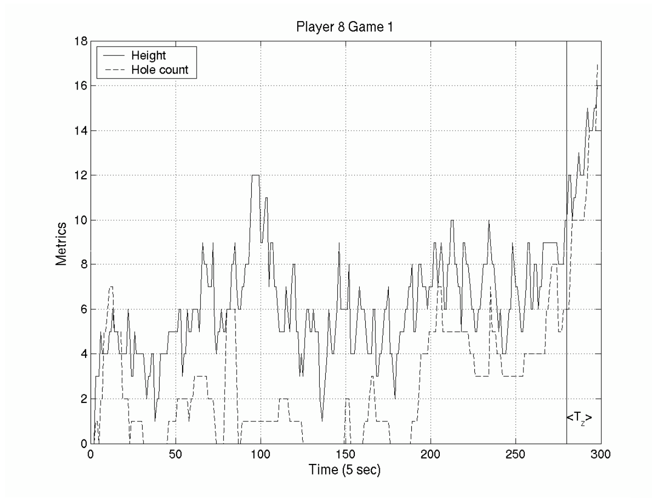


Figure 3: Height and Hole Count for Player 8 Game 1

dark vertical line marks T_z=1.5 min before the end of the game. The states of the game on the right of the vertical line are the states starting from which this game ends in less that T_z seconds. The states on the left of the vertical line are the states from which this game lasts longer that T_z seconds. This division can be used to calculate the proportions required for fitting the logistic model. For each state X_i we count how many times the game was in this state and how many of those lie on the right of the vertical line. This gives a proportion p_i of cases for state X_i for which the game is over in less that T_z seconds. This proportion is a rough estimate of the probability that the game will be over in less than T_z seconds if the current state is X_i.

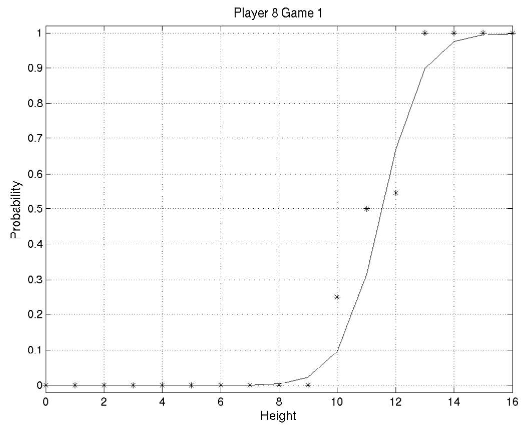


Figure 4: Linear logistic model w/ height regressor

Figure 4 shows the fit of a linear logistic model with one regressor, height. The stars represent calculated proportions for the game to be over in less than T_z seconds, given a particular state represented by only one metric, height. The solid line is the fitted model that gives the probability of the game to be over in less than T_z seconds as a function of height.

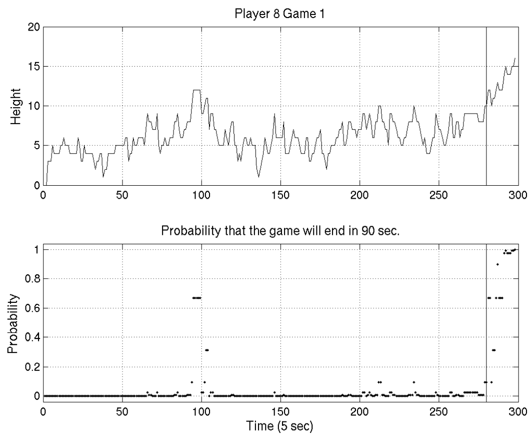


Figure 5: Model 1 Predictions

Figure 5 shows predictions made by Model 1. The upper subplot shows height from the beginning to the end of the game. The lower plot shows the probability that the game will be over in less than T_z seconds. We see that at approximately T=100 the model-predicted probability of gameover becomes high, but the game continued much longer. This is a false alarm from the one-regressor model.

Figure 6 shows the actual alarms raise for entering the red zone (symbols) defined as the states leading to gameover in less than T_z seconds with probability greater than 0.5. We see false alarms at approximately T=100. Examples are the symbols just after the dark vertical line.

We can now compare the performance of Model 2 constructed with bivariate regressors: height and hole density.

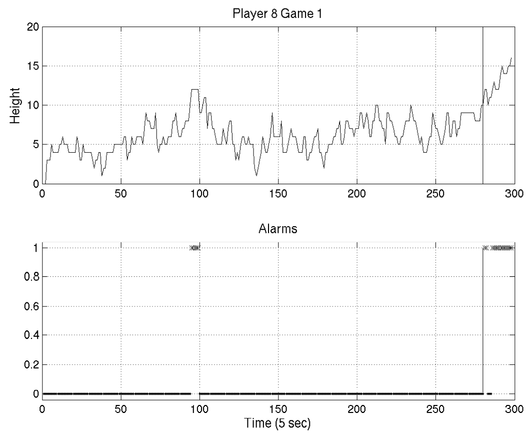


Figure 6: Model 1 Alarms

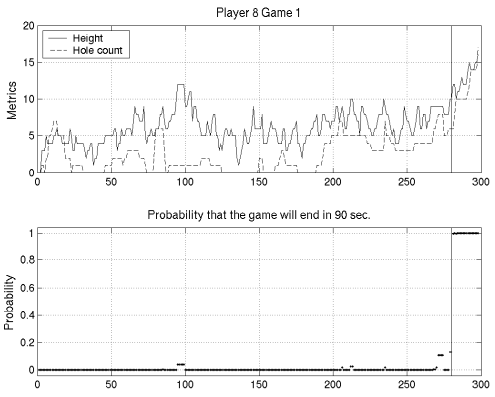
Figure 7 shows the 3D surface depicting the linear logistic regression fit by model 2.

Figure 8 shows predictions of the probability of gameover in T_z seconds as the game progresses from the beginning to the end. Notice that with model 2, the probability of gameover at T=100 is now very low as compared to predictions made using Model 1.

Finally, Figure 9 shows actual alarms for entering the red zone produced using model 2 with 2 regressors. There are no false alarms in the vicinity of T=100, and no missed alarms following transition into the red zone. This is a significant improvement over predictions by model 1.

4.3 Model Use Phase

After developing the MSET models, we also explored whether and how these models and corresponding corrective action strategies would improve system dependability.



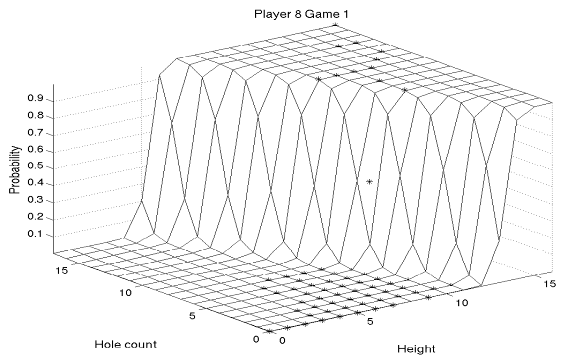


Figure 7: Model 2 linear logistic regression

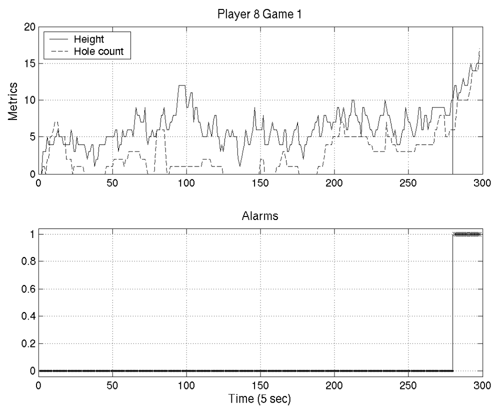


Figure 9: Model 2 Alarms

4.3.1 Hypothesis

The research hypothesis behind this study is that the models developed in the last study will be reliable even in new contexts and that corrective action strategies actually improve system dependability.

4.3.2 Operational Model

Experimental Platform. For this study, we modified the Tetris application used in the previous study. In addition to the telemetry data already captured, we instrumented the models and the corrective action strategies. We linked both models into the application, but allowed only one to raise alarms during any one game. We also increased the sampling rate to once per second to more precisely view the system's behavior.

We implemented a single corrective action strategy, which was to clear the bottom 4 rows from the grid and to reset the game speed. This strategy takes into account the two fundamental ways to prevent a system from going into overload, limiting access to the system and changing the character of work already in progress in the system

Evaluation Criteria. To evaluate the models, we examined how well they predicted impending failure in this new context and then examined whether the models and corrective actions allow players to play longer and score more points. Since the models effectively work by trading some points now in hope of scoring more points in the future, we want to use corrective actions if and only if they're actually necessary. In Tetris terms, we want to minimize the number of partial rows thrown away (by the corrective action strategy) while maximizing the number of rows completed. Different corrective action strategies will create different cost-benefit tradeoffs. To quantify this, we came up with an alternative score where each completed row added one point and each row thrown away while executing a corrective action lost one point. We also assessed a penalty of 1 point for each partially-filled row at each loss (or end of game).

4.3.3 Data Collection

In this study, we asked 10 CS grad students to use the modified Tetris application. None of these students had participated in the previous study. Each participant played 4, 15-minute periods. The first period was considered a warm up. In the second period the participants used the application without the corrective action strategy enabled (NoModel). The third and the fourth periods involved the corrective action strategy enabled applications. At the start of the third game, one model was randomly selected and used throughout that game. Since the models and strategies we are using make it impossible to lose the game, we automatically end the game after 15 minutes. The fourth game was the same, but used the remaining model (Model-1 is the univariate model, Model-2 the bivariate).

During each game we logged both models' health indices at each interval. In all, we collected data from 52 games in the second period (with no corrective action) and 12 games each from the third and fourth periods (with corrective action) using 12 participants.

4.3.4 Data Analysis

To understand whether the predictive models from the previous study were useful in this one, we examined when the models would have raised alarms in the NoModel games. Figure 10 shows that both models would have triggered alarms before loss occurred and that on the average Model-2 would have raised alarms 82s from failure, while Model-1 would have raised alarms 64s from failure. Since the models were built to predict failure 90s away, we consider Model-2 to be closer to the desired behavior.

The results obtained by having participants play with no corrective actions, and with corrective actions actuated by Model 1 and Model 2 are summarized here:

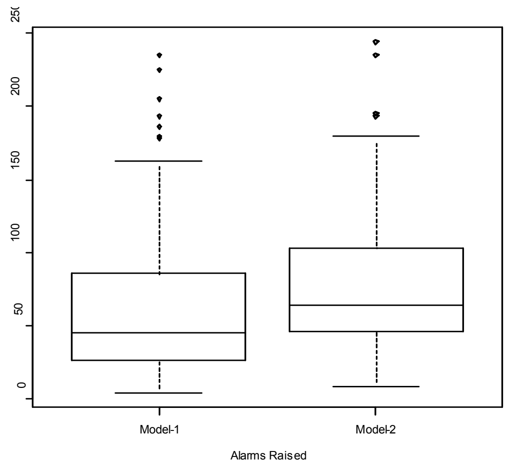


Figure 10: Advance warning by model type

Figure 11 shows the avg work performed with and without models. No Model games had an average alternative score of only 10. They processed 72 rows, but lost frequently. Model-1 (83 processed and 21 thrown away) games processed 62 more and Model-2 games (80 processed and 25 thrown away) processed 55 more. Clearly, the models allowed longer, more effective play.

The conclusion from this proof-of concept investigation is that it possible (even in human-controlled) systems to enhance overall system operation and maintenance strategies by a combination of continuous system telemetry coupled with pattern recognition. The pattern recognition module can detect the incipience or onset of problems proactively and trigger automated corrective actions.

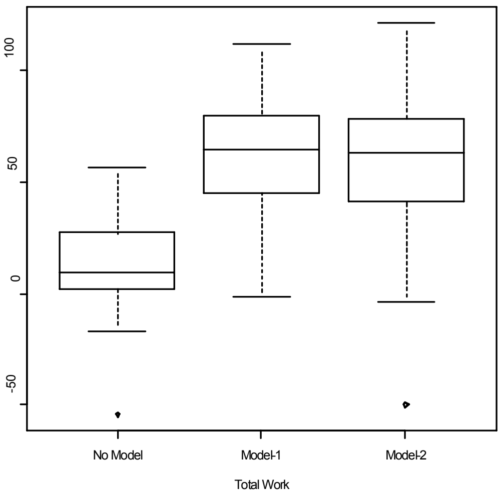


Figure 11: Total work by model type

5 Related Work

Our software dependability work is motivated by research in predictive detection and software engineering Several researchers, have understood the difficulty of building dependable systems. The Eternal system for fault-tolerant CORBA is designed to enable the transparent integration of fault tolerance into existing applications 16. France and Georg [13] separate fault tolerance code from the core application code and weave it together using aspect-oriented techniques. Kalbarczyk et al. [18] describe a Chameleon framework that allows different levels of availability requirements to be simultaneously supported in a networked environment using configurable software ARMOR modules that are tuned for specific failure modes. Xu et. al. [14] provide a framework that allows systems to trade off dependability and efficiency characteristics at runtime based on design-time choices. Our framework also has the goal of making it easy for application developers to get the benefits of sophisticated dependability infrastructure without making major modifications to their code bases.

Our work is also influenced by work on predictive detection, such as the Pinpoint system [15]. This work explores techniques for analyzing time series data to understand its effect on certain outcomes of events

Researchers in [11-13] have investigated the use of predictive algorithms for a closely related objective of enhancing performance for improved manageability of computing systems. That research is leading to new system-management innovations, improving the performance and quality-of-service of complex uniprocessor and distributed-processor systems. Our work focuses on software dependability improvement for human-in-the-loop scenarios where humans need information in a low-pressure setting with an uncluttered, prioritized format to avoid "cognitive overload" mistakes that can quickly compound into complete system failure. In such settings where humans remain a part of the decision/action process, we introduce here a quantitative methodology for continuously assessing the "residual life" of a system, which reflects not only the overall health of the system but the likelihood of being able to perceive and react to incoming information when system upset events occur that may increase the rate incoming messages. The real value of the methodology introduced in this investigation is the ability of the SDF to automate the decision as to when it is very unlikely that the present mitigation actions (which may be a combination of human actions and those from automated system management systems) will avoid complete system failure. At this point, and with quantitative confidence factors, the system can reject incoming work and/or kill off the lowest priority processes until equilibrium has been re-established. Finally, another difference between the contributions in [11-13] and the approach we adopted is the use of nonlinear, nonparametric regression as embodied in MSET. Conventional predictive algorithms are based on threshold-type rules. When there are three or more telemetry metrics to be monitored, MSET has a significant advantage over conventional threshold-limit rules as it is sensitive to anomalies in the correlation patterns between and among monitored dynamic variables. Thus, sensor "stuck-at" failure events and other degradation modes wherein telemetry is faulty but nevertheless will not trip a threshold are causes of misidentifications in conventional predictive algorithms. MSET, by monitoring the correlation structure among monitored metrics, gives sensitive alarms when signals go out of bounds (as would conventional system management solutions); but also when anomalies appear in the correlation patterns among the monitored signals, even while those signals are still within their normal range of variation.

6 Summary

Modern software systems increasingly need to be dependable. Yet the software engineering techniques and tools for achieving it are limited, relying almost exclusively on one-of-a-kind solutions and ad hoc optimizations. Consequently, there is a great need for software engineering tools and techniques that make it easier for developers to build dependability into their systems.

We have developed the SDF to improve this situation. This framework is based on a generic dependability strategy in which systems are instrumented to produce runtime telemetry data. These data are then analyzed by MSET, which automatically produces statistical models intended to predict impending failures well before they occur. These models are then fed back to the system to allow a system to monitor itself at runtime, raising alarms when the system is believed to be approaching failure. To support this process we developed tools, libraries, and an architecture complete with reusable components.

We also presented a simple proof-of-concept study in which we used our framework to instrument a simple application. We then asked a number of subjects to use the application, from which we automatically developed failure-predicting models. We evaluated these models and demonstrated that they indeed reliably predicted failure. Next we linked the models back into the application and asked a different set of users to execute it. We found that the models worked well in this new context - even though the application is completely driven by human input.

For future work, we need techniques to efficiently transmit telemetry data over the network for off board analysis by centralized servers. We plan to quantitatively analyze the overhead generated by telemetry data collection. Given that more frequent telemetry sampling may allow us to build better models of the system, but only at the price of increased speed and space overhead, we will analyze the cost/benefit tradeoff in telemetry sampling rates. We intend to scale techniques up to larger systems in domains such as ecommerce servers and applications.

7 References

1. Java Virtual Machine Profiler Interface (JVMPI). Sun Microsystems. Feb.,1999
2. Tetris.com.The Tetris Company, LLC. 2003.
3. C. Talbot, “On-Condition Replacement,”, Proc. MARCON 2001, “Maintenance and Reliability in the 21st Century”, Gatlinburg, TN, (May 6-9, 2001).
4. K. Validyanathan, R. E. Harper, S. W.d Hunter, and K. S. Trivedi, “Analysis and Implementation of Software Rejuvenation in Cluster Systems,” ACM Sigmetrics 2001/Performace 2001, June 2001.
5. K. C. Gross, K. Mishra, R. L. Bickford, “Proactive Detection of Software Aging Mechanisms in Performance-Critical Computers,” Proc. 27th Ann. NASA SW Eng. Sym., Greenbelt, MD, (Dec 4-6, 2002.)
6. E. D. Demaine, S. Hohenberger, and D. Liben-Nowell, “Tetris is Hard, Even to Approximate,” Tech. Rept. MIT-LCS-TR-865, MIT (Oct 21, 2002).
7. K. Cassidy, K. C. Gross, and A. Malekpour. "Advanced Pattern Recognition for Detection of Complex Software Aging Phenomena in Online Transaction Processing Servers," Int'l Performance and Dependability Symposium, Washington, DC, June 23-26, 2002.
8. K. C. Gross, R. M. Singer, S. W. Wegerich, J. P. Herzog, R. VanAlstine, and F. Bockhorst “Application of a Model-based Fault Detection System to Nuclear Plant Signals,” Proc. 9th Intnl. Conf. On Intelligent Systems Applications to Power Systems, pp. 66-70, Seoul, Korea (July 6-10, 1997).
9. R. M. Singer, K. C. Gross, J. P. Herzog, R. W. King, and S. Wegerich, “Model-Based Nuclear Power Plant Monitoring and Fault Detection: Theoretical Foundations,” Proc. 9th Intnl. Conf. On Intelligent Systems Applications to Power Systems, pp. 60-65, Seoul, Korea (July 6-10, 1997).
10. Distributed Operations and Management, 2001. "A Statistical Approach to Predictive Detection," Joseph L. Hellerstein, Fan Zhang and Perwez Shahabuddin, Computer Networks, January, 2000.
11. "Rule Induction of Computer Events," Ricardo Vilalta, Sheng Ma, and Joseph L. Hellerstein, Distributed Operations and Management, 2001.
12. "A Statistical Approach to Predictive Detection," Joseph L. Hellerstein, Fan Zhang and Perwez Shahabuddin, Computer Networks, January, 2000.
13. R. France and G. Georg. “An Aspect-Based Approach to Modeling Fault Tolerance Concerns”. CSU Tech Report, 2002.
14. J. Xu, A. Bondavalli and F. Di Giandomenico. "Dynamic Adjustment of Dependability and Efficiency in Fault-Tolerant Software," in Predictably Dependable Computing Systems, pp. 155-172, Brussels, Springer Verlag, 1995. ISBN 3-540-59334-9
15. Chen, M., E. Kiciman, E. Fratkin, E. Brewer and A. Fox. Pinpoint: Problem Determination in Large, Dynamic, Internet Services. Proceedings of the International Con. on Dependable Systems and Networks (IPDS Track), Washington D.C., 2002.
16. P. Narasimhan, Ph.D. Dissertation, Technical Report #99-18, Department of Electrical and Computer Engineering, University
17. Kalbarczyk, Z.T., Iyer, R.K., Bagchi, S., Whisnant, K. “Chameleon: A Software Infrastructure for Adaptive Fault Tolerance”. IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 6, June 1999.
18. Thach, W. T. On the specific role of the cerebellum in motor learning and cognition: Clues from PET activation and lesion studies in man. Behavioral and Brain Sciences 19(3): 411-431, 1996.